MIPSpro™ Fortran Language Reference Manual, Volume 1

007–3692–006

Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

Record of Revision

Version	Description
1.0	December 1993 Original Printing.
1.1	June 1994 Online-only revision. Includes minor updates and corrections to revision 1.0.
2.0	October 1995 This printing supports the CF90 compiler release 2.0 running on Cray PVP systems, Cray T3E systems, and SPARC systems. The implementation of features on Cray T3E systems is deferred. Sections 11 through 14 of revision 1.0 are now part of the *CF90 Fortran Language Reference Manual, Volume 1*, SR–3903. Appendix sections A through G of revision 1.0 are now part of the *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905.
3.0	May 1997 This printing supports the Cray Research CF90 3.0 release, running on UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 compiler 7.2 release, running on the IRIX operating system. The implementation of features on IRIX operating system platforms is deferred. Sections 9 and 10 of revision 2.0 are now part of the *MIPSpro Fortran Language Reference Manual, Volume 2*.
3.0.1	August 1997 This online revision supports the Cray Research CF90 3.0.1 release, running on UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 compiler 7.2 release, running on the IRIX operating system. Includes minor corrections and updates to revision 3.0.
3.0.2	March 1998 This online revision supports the Cray Research CF90 3.0.2 release, running on UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 compiler 7.2.1 release, running on the IRIX operating system. Includes minor corrections and updates to revision 3.0.1.

3.1	August 1998 This online revision supports the Cray Research CF90 3.1 release, running on UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 compiler 7.2.1 release, running on the IRIX operating system. Includes minor updates and corrections to revision 3.0.2.
3.2	January 1999 This revision (007–3692–004) supports the CF90 3.2 release, running on the UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 7.3 release, running on the IRIX operating system. It includes major updates to revision 3.1.
005	July 1999 This revision supports the CF90 3.3 release, running on the UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 7.3 release, running on the IRIX operating system. It includes minor updates to revision 3.2.
006	September 2002 This revision supports the MIPSpro Fortran 90 release 7.4 running on the IRIX operating system version 6.5 and later.

Contents

Figures

Tables

About This Guide

This manual describes the Fortran language as implemented by the MIPSpro Fortran 90 compiler. This compiler implements the Fortran standard.

The compiler was developed to support the Fortran standard adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). This standard, commonly referred to in this manual as *the Fortran standard*, is ISO/IEC 1539–1:1997. Because the Fortran standard is, generally, a superset of previous standards, the compiler will compile code written to previous standards.

Note: The Fortran 95 standard is a revision to the Fortran 90 standard. The standards organizations continue to interpret the Fortran standard for vendors. To maintain conformance to the Fortran standard, SGI may need to change the behavior of certain compiler features in future releases based upon the outcomes of interpretations to the standard.

Related Compiler Publications

This manual is one of a set of manuals that describes the compiler. The complete set of manuals is as follows:

- *MIPSpro Fortran Language Reference Manual, Volume 1*. Chapters 1 through 8 correspond to sections 1 through 8 of the Fortran standard.
- *MIPSpro Fortran Language Reference Manual, Volume 2*. Chapters 1 through 6 of this manual correspond to sections 9 through 14 of the Fortran standard.
- *MIPSpro Fortran Language Reference Manual, Volume 3*. This manual contains compiler information that supplements the Fortran standard. The standard is the complete, official description of the language. This manual also contains the complete Fortran syntax in Backus-Naur form (BNF).

Compiler Messages

You can obtain compiler message explanations by using the online `explain`(1) command.

Compiler Man Pages

In addition to printed and online prose documentation, several online man pages describe aspects of the compiler. Man pages exist for the library routines, the intrinsic procedures, and several programming environment tools.

You can print copies of online man pages by using the pipe symbol with the `man`(1), `col`(1), and `lpr`(1) commands. In the following example, these commands are used to print a copy of the `explain`(1) man page:

```
% man explain | col -b | lpr
```

Each man page includes a general description of one or more commands, routines, system calls, or other topics, and provides details of their usage (command syntax, routine parameters, system call arguments, and so on). If more than one topic appears on a page, the entry in the printed manual is alphabetized under its primary name; online, secondary entry names are linked to these primary names. For example, `egrep` is a secondary entry on the page with a primary entry name of `grep`. To access `egrep` online, you can type `man grep` or `man egrep`. Both commands display the `grep` man page to your terminal.

Related Fortran Publications

The following commercially available reference books are among those that you can consult for more information on the history of Fortran and the Fortran language itself:

- Adams, J. C., W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 95 Handbook : Complete ISO/ANSI Reference*. MIT Press, 1997. ISBN 0262510960.
- Chapman, S. *Fortran 90/95 for Scientists and Engineers*. McGraw Hill Text, 1998. ISBN 0070119384.
- Chapman, S. *Introduction to Fortran 90/95*. McGraw Hill Text, 1998. ISBN 0070119694.
- Counihan, M. *Fortran 95 : Including Fortran 90, Details of High Performance Fortran (HPF), and the Fortran Module for Variable-Length Character Strings*. UCL Press, 1997. ISBN 1857283678.
- Gehrke, W. *Fortran 95 Language Guide*. Springer Verlag, 1996. ISBN 3540760628.
- International Standards Organization. *ISO/IEC 1539–1:1997, Information technology — Programming languages — Fortran*. 1997.

- Metcalf, M. and J. Reid. *Fortran 90/95 Explained*. Oxford University Press, 1996. ISBN 0198518889.

Obtaining Publications

To order a document, call +1 651 683 5907. SGI employees may send e-mail to `orderdsk@sgi.com`.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
DEL or DELETED	The DEL or DELETED notation indicates that the feature being described has been deleted from the Fortran standard. The compiler supports these features, but it issues a message when a deleted feature is encountered.
EXT or EXTENSION	The EXT or EXTENSION notation indicates that the feature being described is an extension to the Fortran

	standard. The compiler issues a message when extensions are encountered.
OBS or OBSOLESCENT	The OBS or OBSOLESCENT notation indicates that the feature being described is considered to be obsolete in the Fortran standard. The compiler supports these features, but it issues a message when an obsolescent feature is encountered.
xyz_list	When *_list* is part of a syntax description, it means that several items may be specified. For example, *xyz_list* can be expanded to mean *xyz* [, *xyz*] ... .
*scalar_*	When *scalar_* is the first item in a syntax description, it indicates that the item is a scalar, not an array, value.
_name	When *_name* is part of a syntax definition, it indicates that the item is a name with no qualification. For example, the item must not have a subscript list, so `ARRAY` is a name, but `ARRAY(I)` is not.
(R*nnnn*)	Indicates that the Fortran 90 standard has rules regarding the characteristic of the language being discussed. All rules are numbered, and the numbered list appears in the *MIPSpro Fortran Language Reference Manual, Volume 3*. The numbering of the rules in the *MIPSpro Fortran Language Reference Manual, Volume 3* matches the numbering of the rules in the standard. The forms of the rules in the *MIPSpro Fortran Language Reference Manual, Volume 3* and the BNF syntax class terms that are used may differ from the rules and terms used in the standard.
`POINTER`	The term `POINTER` refers to the Fortran `POINTER` attribute.
Cray pointer	The term *Cray pointer* refers to the Cray pointer data type extension.
Fortran Fortran standard	These terms refer to the current Fortran standard, which is the Fortran 95 standard. For situations when it might otherwise be confusing, a specific standard is

mentioned along with its numeric identifier (FORTRAN 77, Fortran 90, Fortran 95).

BNF Conventions

This section describes some of the commonly used Backus-Naur Form (BNF) conventions.

Terms such as *goto_stmt* are called *variable entries*, *nonterminal symbols*, or simply, *nonterminals*. The metalanguage term *goto_stmt*, for example, represents the `GO TO` statement, as follows:

goto_stmt	**is**	`GO TO` *label*

The syntax rule defines *goto_stmt* to be `GO TO` *label*, which describes the format of the `GO TO` statement. The description of the `GO TO` statement is incomplete until the definition of *label* is given. *label* is also a nonterminal symbol. A further search for *label* will result in a specification of *label* and thereby provide the complete statement definition. A *terminal* part of a syntax rule is one that does not need further definition. For example, `GO TO` is a terminal keyword and is a required part of the statement form. The complete BNF list appears in the *MIPSpro Fortran Language Reference Manual, Volume 3*.

The following abbreviations are commonly used in naming nonterminal keywords:

Abbreviation	Term
arg	argument
attr	attribute
char	character
decl	declaration
def	definition
desc	descriptor
expr	expression
int	integer

op	operator
spec	specifier or specification
stmt	statement

The term **is** separates the syntax class name from its definition. The term **or** indicates an alternative definition for the syntactic class being defined. The following example shows that *add_op*, the add operator, may be either a plus sign (+) or a minus sign (-):

add_op	**is**	+
	or	-

Indentation indicates syntax continuation. If a rule does not fit on one line, the second line is indented. This is shown in the following example:

dimension_stmt	**is**	DIMENSION [::] *array_name* (*array_spec*) [, *array_name* (*array_spec*)] ...

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library World Wide Web page:

  http://techpubs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.
- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy., M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

We value your comments and will respond to them promptly.

Introduction

For a programming language, Fortran has existed for a long time. It was one of the first widely used high-level languages and was the first programming language to be standardized. It is still the premier language for scientific and engineering computing applications. This chapter provides compatibility information and introduces the newest additions to the Fortran language.

1.1 FORTRAN 77 Compatibility

Because of the large investment in existing software written in Fortran, the Fortran standards committee decided to include the entire previous FORTRAN 77 standard in the Fortran 90 standard. In the Fortran 95 standard, however, some features are deleted. Even though the Fortran 95 standard deleted some features, the compiler has not deleted any features. Features from older standards that are deleted are honored, but they generate an ANSI message from the compiler.

Fortran 95 restricts the behavior of some features that were processor dependent in FORTRAN 77. Therefore, a program that conforms to the FORTRAN 77 standard and uses processor-dependent features can also conform to the Fortran 95 standard and yet behave differently than with some FORTRAN 77 systems. In the following situations, the Fortran 95 interpretation is different from that of FORTRAN 77:

- Fortran 95 contains more intrinsic functions than FORTRAN 77 did, and Fortran 95 has a few intrinsic subroutines. Therefore, a standard-conforming FORTRAN 77 program can have a different interpretation under this standard if it invokes an external procedure that has the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an `EXTERNAL` statement as recommended for nonintrinsic functions.
- If a named variable that was not in a common block was initialized in a `DATA` statement, it has the `SAVE` attribute in Fortran 95. In FORTRAN 77, if the value of the variable was changed or became undefined, its value on reentry into a procedure was processor dependent. The compiler treats variables initialized in `DATA` statements as if they had appeared in a `SAVE` statement. MIPSpro F77 saves the variables in `DATA` statements on the heap instead of the stack.
- In FORTRAN 77, an input list could never require more characters than were present in a record during formatted input. Fortran 95 does not dictate this

restriction for cases in which the PAD= specifier is YES; in these cases, the input record is padded with as many blanks as necessary to satisfy the input item and the corresponding format. For more information on this, see the *MIPSpro Fortran Language Reference Manual, Volume 3*.

- FORTRAN 77 permitted a processor to supply extra precision for a real constant when it was used to initialize a DOUBLE PRECISION data object in a DATA statement. Fortran 95 does not permit this.
- The format of a floating point zero written with a G edit descriptor is different in Fortran 95. The floating-point zero was written with an E*w*.*d* edit descriptor in FORTRAN 77, but it is written with an F*w*.*d* edit descriptor in the compiler. FORTRAN 77 output cannot be changed. Therefore, different compare files must be retained for FORTRAN 77 and Fortran 95 programs that use the G edit descriptor for floating-point output.

1.2 Fortran 90 Compatibility

The Fortran 90 language standard introduced new data types, new operators, and new meanings for the existing operators and assignment. It provided ways for programmers to extend Fortran. These facilities allow programmers to create abstract data types by defining new types and the operations to be performed on them. Modules were introduced into Fortran as a convenient way to package these new data types and their operations. Modules can be used by the same programmer in different applications or can be distributed to several users on the same or different projects.

The Fortran 90 standard (ISO/IEC 1539:1991) described the syntax and semantics of the Fortran programming language. The standard addressed certain aspects of the Fortran processing system, but it did not address others. When specifications were not covered by the standard, the interpretation was processor dependent; that is, the processor defined the interpretation, but the interpretation for any two processors did not need to be the same. Typically, programs that rely on processor-dependent interpretations are not portable.

The Fortran 90 standard declared several features to be obsolescent. In the Fortran 95 standard, some of Fortran 90's obsolescent features are deleted. The compiler has not deleted any features. The compiler issues an ANSI message whenever a deleted feature is used, however. For more information on deleted and obsolescent features, see the *MIPSpro Fortran Language Reference Manual, Volume 3*.

1.2.1 Fortran 95 language standard

Fortran 95 continues the evolutionary model introduced in Fortran 90 by deleting several of the features marked as obsolescent in Fortran 90 and identifying a few new obsolescent features. For information on these features, see the *MIPSpro Fortran Language Reference Manual, Volume 3*.

Fortran 95 is a relatively minor evolution of standard Fortran, with the emphasis in this revision being upon correcting defects in the Fortran 90 standard. This new standard also provides interpretation for a number of questions that have arisen concerning Fortran 90 semantics and syntax. For example, the Fortran 95 `SIGN`(3i) intrinsic function behaves differently from the Fortran 90 `SIGN`(3i) function if the second argument is negative real zero.

In addition to corrections and clarifications, Fortran 95 contains several extensions to Fortran 90. The major extensions are as follows:

- The `FORALL` statement and construct.
- `PURE` and `ELEMENTAL` procedures.
- Pointer initialization and structure default initialization.
- Additional intrinsic procedures. A Fortran 90 program may have a different interpretation under the Fortran 95 standard if it invokes an external procedure that has the same name as one of the new standard intrinsic procedures unless that procedure is specified in an `EXTERNAL` statement or an interface body.

The Fortran 95 standard specifies the following information:

- Syntax of Fortran statements and forms for Fortran programs
- Semantics of Fortran statements and Fortran programs
- Specifications for correct input data
- Appearance of standard output data

The standard does not specify the following information:

- The way in which each Fortran compiler is written
- Operating system facilities defining the computing system
- Methods used to transfer data to and from peripheral storage devices and the nature of the peripheral devices

- Behavior of vendor extensions
- Size and complexity of a Fortran program and its data
- Hardware or firmware used to run the program
- The way values are represented and the way numeric values are computed
- Physical representation of data
- Characteristics of tapes, disks, and various storage media

1.2.2 Program Conformance

A program conforms to the standard if all the statements are syntactically correct, execution of the program causes no violations of the standard (such as dividing by zero), and all the input data is in the correct form.

Compiler extensions to the Fortran standard appear in notes throughout the text.

1.2.3 Processor Conformance

In the Fortran standard, the term *processor* means the combination of a Fortran compiler and the computing system that executes the code. A processor conforms to the standard if it compiles and executes programs that conform to the standard, provided that the Fortran program is not too large or complex for the computer system in question.

Options on the `f90`(1) command line can direct the compiler to flag nonstandard usage. For more information on the command lines, see the `f90`(1) man page or the *MIPSpro Fortran 90 Commands and Directives Reference Manual*. When the option is in effect, the compiler prints messages for extensions to the standard that are used in the program. As required by the standard, the compiler also flags the following items and provides the reason that the item is being flagged:

- Obsolescent features
- Deleted features
- Kind type parameters not supported
- Violations of any syntax rules and the accompanying constraints
- Characters not permitted by the processor

- Illegal source form
- Violations of the scope rules for names, labels, operators, and assignment symbols

These conformance requirements were not present in previous Fortran standards.

The compiler includes extensions to the Fortran standard. Because the compiler processes programs according to the standard, they are considered to be standard-conforming processors. When the option to note deviations from the Fortran standard is in effect, extensions to the standard are flagged with ANSI messages when detected at compile time.

1.2.4 Portability

One of the main purposes of a standard is to describe how to write portable programs. However, some things are standard-conforming but not portable; for example, a program that computes a very large number like 10^{250}. Certain computing systems will not accommodate a number this large. Such a number could be part of a standard-conforming program, but it might not run on all systems; therefore, it might not be portable. Another example is a program that uses a deeper nesting of control constructs than is allowed by a particular compiler.

Fortran Concepts and Terms

Terms are used in a precise way to describe a programming language, so this chapter introduces the fundamental terms needed to understand Fortran. You can consult the glossary in this manual for definitions of terms.

One of the major concepts involves the organization of a Fortran program. This topic is introduced in this chapter by presenting the high-level syntax rules for a Fortran program, which includes the principal constructs and statements that form a program. This chapter also describes the order in which constructs and statements must appear in a program and concludes with an example of a short Fortran program.

2.1 Scope and Association

In examining the basic concepts in Fortran, it helps to trace some of the important steps in its evolution. The results of the first few steps are familiar to Fortran programmers, but the later ones become relevant only when the new features of Fortran are used.

The first version of Fortran produced in the late 1950s did not have user-defined subroutines or functions, but it did contain intrinsic functions.

Programmers soon realized the benefits of isolating definitive chunks of code into separate units known as function and subroutine subprograms. This not only provided a mechanism for structuring a program but permitted subprograms to be written once and then be called many times by the same program (or even be used by more than one program). Equally important, subprograms could be compiled separately.

With this powerful tool came complications. For example, if both the main program and a subprogram use the variable named X, what is the connection between them? The answer is that, in general, there is no connection between X in the main program and X in a subprogram. Subprograms are separately compilable; an X in a different subprogram is not even known at compile time, so the simplest thing to do is have no connection between variables with the same name in different program units. Thus, if two different programmers work on different program units, neither one needs to worry about names picked by the other. This idea is described by saying that the two Xs have different *scope*.

A subroutine could be written to do a summation of the first hundred integers and print the results. For example:

```
  SUBROUTINE TOTAL
  M = 0
  DO I = 1, 100
     M = M + I
  END DO
  WRITE(6, 9) M
9 FORMAT(I10)
  RETURN
  END
```

With this subroutine available, the main program could be written as follows:

```
CALL TOTAL
STOP
END
```

Suppose you decide that the subroutine would be more generally useful if it computed the sum but did not print it, as follows:

```
SUBROUTINE TOTAL
M = 0
DO I = 1, 100
   M = M + I
END DO
RETURN
END
```

A first attempt to use this subroutine might produce the following erroneous program:

```
  CALL TOTAL
  WRITE(6, 9) M
9 FORMAT(I10)
  STOP
  END
```

This does not work because the variable `M` in the subroutine has nothing to do with variable `M` in the main program. A connection between the two values should exist, so when subroutines and functions were introduced, two schemes were provided to communicate values between them and other program units: procedure arguments and common blocks.

The following two complete programs communicate values in two ways. The first, program ARGSUM, uses a subroutine argument, and the second, program COMSUM, uses a common block to communicate values. The names in the different program units identify completely separate variables, yet their values are communicated from one to the other by using either arguments or common blocks, so the name of the variable holding the sum in the subroutine need not be the same as the corresponding variable in the calling program.

```
  PROGRAM ARGSUM
  CALL TOTAL(M)
  WRITE(6, 9) M
9 FORMAT(I10)
  END

  SUBROUTINE TOTAL(ITOTAL)
  ITOTAL = 0
  DO I = 1, 100
     ITOTAL = ITOTAL + I
  END DO
  END
```

Program COMSUM, which follows, performs the same computation as program ARGSUM:

```
  PROGRAM COMSUM
  COMMON /CB/ M
  CALL TOTAL
  WRITE(6, 9) M
9 FORMAT(I10)
  END

  SUBROUTINE TOTAL
  COMMON /CB/ ITOTAL
  ITOTAL = 0
  DO I = 1, 100
     ITOTAL = ITOTAL + I
  END DO
  END
```

To describe even these simple cases and appreciate how they work already requires the introduction of some terms and concepts. To precisely describe the phenomenon that the subroutine variable ITOTAL is not known outside the subroutine, the concept

of *scope* is used. Because the scope of a variable in a common block is local, the scope of ITOTAL includes only the subroutine — not the main program.

The scope of ITOTAL is the subroutine; the scope of the variable M is the main program. However, the scope of the common block name CB is global. *association* is used to describe the connection between M in the main program and ITOTAL in the subroutine. In the first example it is *argument association*, and in the second it is *storage association.*

To summarize, the *scope* of a variable is that part of the program in which it is known and can be used. Two variables may have the same name and nonoverlapping scopes; for example, there may be two completely different variables named X in two different subprograms. Association of variables means that there may be two different names for the same object; this permits sharing values under certain conditions.

With arguments available, it is natural to generalize the computation somewhat to allow the upper limit of the sum (100 in the example) to vary. Also, a function is more natural in this case than a subroutine, because the object of the computation is to return a single value. These changes produce the following program:

```
PROGRAM PTOTAL
INTEGER TOTAL
PRINT *, TOTAL(100)
END

FUNCTION TOTAL(N)
INTEGER TOTAL
TOTAL = 0
DO I = 1, N
   TOTAL = TOTAL + I
END DO
END
```

In this example, the scope of N is function TOTAL, but when function TOTAL is called from the main program in the PRINT statement, the value of N, through argument association, becomes 100. The scope of variable I is function TOTAL. The scope of function TOTAL is the whole program, but its type must be declared in the main program because by the implicit typing rules, TOTAL is not of type integer. Also note that there is a function named TOTAL, with global scope, and a variable named TOTAL is local to the function. The use of identifier TOTAL determines whether it is the local variable TOTAL or the global function name TOTAL. When TOTAL is used with an argument list, it is the function name; when used inside the function

subprogram defining the function TOTAL, it is the local variable. Variable TOTAL computes and stores the value that is returned as the value of the function TOTAL.

It is possible to rewrite the example using internal procedures:

```
PROGRAM DO_TOTAL
  PRINT *, TOTAL(100)
  CONTAINS

  FUNCTION TOTAL(N)
     INTEGER TOTAL
     TOTAL = 0
     DO I = 1, N
        TOTAL = TOTAL + I
     END DO
  END FUNCTION TOTAL

END PROGRAM DO_TOTAL
```

This is similar to the previous example, but the function is placed prior to the END statement of the main program and the CONTAINS statement is inserted to mark the beginning of any internal functions or subroutines. In this case, the function TOTAL is not global but is local to the program DO_TOTAL. Also, the FUNCTION statement for TOTAL and the specifications that follow it specify TOTAL as a function of type integer with one integer argument N. The type of TOTAL must not be declared in the specification part of the program DO_TOTAL; to do so would create a duplicate declaration of TOTAL. The information about the type of the function and type of the argument is called the *interface* to the internal function.

To illustrate some other rules about scoping and association related to internal procedures, the example can be changed back to one that uses a subroutine, but one that is now internal.

```
PROGRAM DO_TOTAL
  INTEGER TOTAL
  CALL ADD_EM_UP(100)
  PRINT *, TOTAL
  CONTAINS

   SUBROUTINE ADD_EM_UP(N)
   TOTAL = 0
   DO I = 1, N
      TOTAL = TOTAL + I
```

```
   END DO
   END SUBROUTINE ADD_EM_UP

END PROGRAM DO_TOTAL
```

The preceding example shows that TOTAL in the internal subroutine and TOTAL in the main program are, indeed, the same variable. It does not need to be declared type integer in the subroutine. This is the result of *host association,* wherein internal procedures inherit information about variables from their host, which is the main program in this case. Variable I does not appear in the main program, so its scope is the internal subroutine.

Data declarations and procedures can be placed in a *module*. Then they can be used by other parts of the program. This scheme is illustrated using the summation function example again, as follows:

```
MODULE TOTAL_STUFF
CONTAINS
   FUNCTION TOTAL(N)
   INTEGER TOTAL, N, I
   TOTAL = 0
   DO I = 1, N
      TOTAL = TOTAL + I
   END DO
   END FUNCTION TOTAL
END MODULE TOTAL_STUFF

PROGRAM DO_TOTAL
USE TOTAL_STUFF
PRINT *, TOTAL(100)
END PROGRAM DO_TOTAL
```

The module and the program could be in different files. They can be compiled like subroutines, but unlike subroutines, the module must be available to the compiler when program DO_TOTAL is compiled.

The scope of variables N and I is function TOTAL; N gets its value 100 by argument association. The module name TOTAL_STUFF is global and any program can use the module, which causes the type and definition of function TOTAL to become available within that program. This is called *use association.*

When more extensive examples are constructed using such features as internal procedures within a procedure in a module, there is a need to have a deeper

understanding of scope and association. These topics are introduced briefly in the following section and discussed in more detail in the *MIPSpro Fortran Language Reference Manual, Volume 2*.

2.1.1 Scoping Units

The scope of a program entity is that part of the entire executable program in which that entity is known, is available, and can be used. Some of the parts of a program that constitute the scope of entities are called *scoping units*. Scoping units range in extent of inclusiveness from parts of a statement to an entire program unit.

Some entities have scopes that are something other than a scoping unit. For example, the scope of a name, such as a variable name, can be global to an executing program. For more information on name scope and scoping units, see the *MIPSpro Fortran Language Reference Manual, Volume 2*.

2.1.2 Association

The term *association* describes how different entities in the same program unit or different program units can share values and other properties. It is also a mechanism by which the scope of an entity is made larger. For example, argument association allows values to be shared between a procedure and the program that calls it. Storage association, set up by the use of `EQUIVALENCE` and `COMMON` statements, for example, allows two or more variables to share storage, therefore values, under certain circumstances. Use association and host association both allow entities described in one part of a program to be used in another part of the program. Use association makes entities defined in modules accessible, and host association makes entities in the containing environment available to an internal or module procedure. The complete description of all categories of association are described in the *MIPSpro Fortran Language Reference Manual, Volume 2*.

2.2 Program Organization

A collection of program units constitutes an executable program. Program units can contain other smaller units.

2.2.1 Program Units

A Fortran program unit is one of the following:

- Main program
- External subprogram (subroutine or function)
- Module
- Block data

A Fortran program must contain one main program and can contain any number of the other types of program units. Program units define data environments and the steps necessary to perform calculations. Each program unit has an `END` statement to terminate the program unit. Each program unit also has an identifying initial statement, but the identifying statement for a main program is optional.

An external subprogram (a function or a subroutine) performs a task or calculation on entities available to the external subprogram. These entities can be the arguments to the subprogram that are provided in the reference, entities defined in the subprogram, or entities made accessible by other means, such as common blocks. A `CALL` statement invokes a subroutine. A function is invoked when its value is needed in an expression. The computational process that is specified by a function or subroutine subprogram is called a *procedure*. It may be invoked from other program units of the Fortran program. Neither module nor block data program units are executable, so they are not considered to be procedures. A main program also contains executable constructs, but it is not classified as a procedure.

A block data program unit contains only data definitions. It specifies initial values for a restricted set of data objects.

A module contains public definitions that can be made accessible to other program units and subprograms. It also contains private definitions that are accessible only within the module. These definitions include data definitions, type definitions, definitions of subprograms known as *module subprograms,* and specifications of procedure interfaces. Module subprograms can be either subroutine or function subprograms. The procedures they define are called *module procedures*. Module subprograms can be invoked by other module subprograms in the module or by other program units that access the module.

A main program, external subprograms, and module subprograms may contain internal subprograms, which can be either subroutines or functions. The procedures they define are called *internal procedures*. Internal subprograms must not themselves contain internal subprograms, however. The main program, external subprogram, or module subprogram that contains an internal subprogram is referred to as the internal subprogram's *host*. Internal subprograms may be invoked by their host or by other internal subprograms in the same host.

A statement function also defines a procedure. Executable program units, module subprograms, and internal subprograms can all contain statement functions.

Figure 2-1, page 15, illustrates the organization of a sample Fortran program. Large arrows represent use association with the USE statement at the arrow tip. Small arrows represent subprogram references with the call at the arrow base.

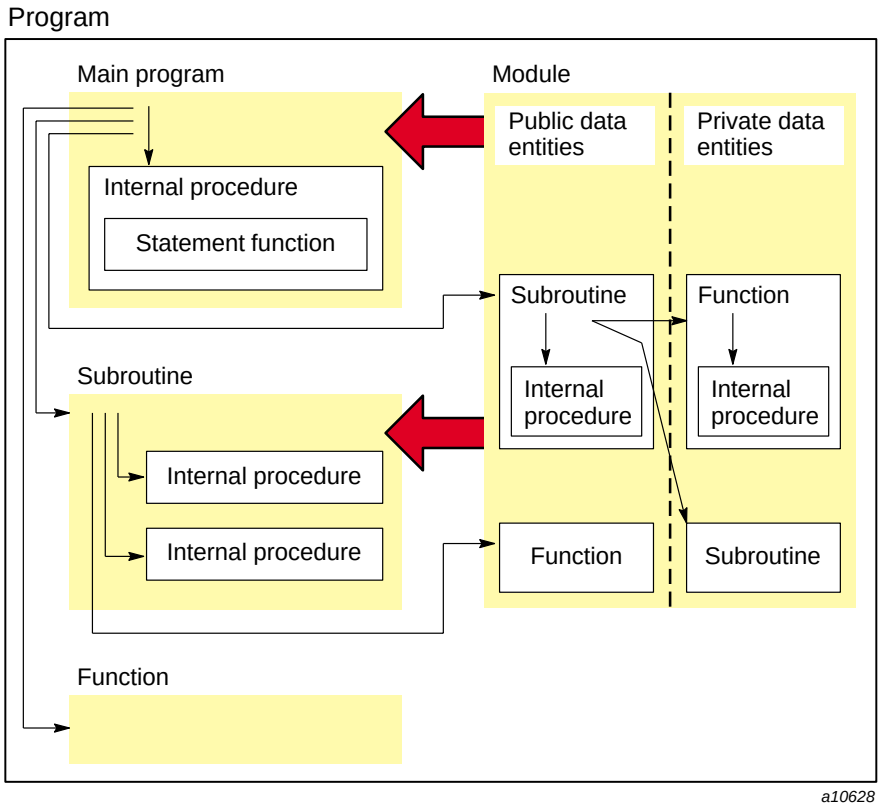

Figure 2-1 Program packaging example

Program units and subprograms are described more fully in the *MIPSpro Fortran Language Reference Manual, Volume 2*.

2.2.2 Packaging

The packaging of a program is an important design consideration when planning a new Fortran application.

The most important benefit of packaging is the capability to hide information. Entities can be kept inaccessible except where they are actually needed. This provides some protection against inadvertent misuse or corruption, thereby improving program reliability. Packaging can make the logical structure of a program more apparent by hiding complex details at lower levels. Programs are therefore easier to comprehend and less costly to maintain. The Fortran features that provide these benefits are internal procedures and modules.

As previously noted, internal procedures can appear in a main program, external subroutines, external functions, and module subprograms. They are known only within their host. The name of an internal procedure must not be passed as an argument, and an internal procedure must not itself be the host of another internal procedure. However, a statement function can appear within an internal procedure. Thus, in some ways, internal procedures are like external procedures and in other ways they are like statement functions.

Modules provide the most comprehensive opportunities to apply packaging concepts, as illustrated in Figure 2-1, page 15. In addition to several levels of organization and hiding, the entities specified in a module (types, data objects, procedures, interfaces, and so on) can be kept private to the module or made available to other scoping units by use association.

2.3 Data Environment

Before a calculation can be performed, its data environment must be developed. The data environment consists of data objects that possess certain properties, attributes, and values. The steps in a computational process generally specify operations that are performed on operands (or objects) to create desired results or values. Operands can be constants, variables, constructors, or function references; each has a data type and (if defined) a value. In some cases the type can be assumed by the processor; in other cases it may be declared. A data object has attributes other than type. Chapter 4, page 67, discusses data type in detail; Chapter 5, page 115, discusses the other attributes of program entities; and Chapter 6, page 183, and Chapter 7, page 211, describe how data objects are used.

2.3.1 Data Type

The Fortran language provides five intrinsic data types: real, integer, complex, logical, and character, and it lets you define additional types. Sometimes it is natural to organize data in combinations consisting of more than one type. For example, assume that a program is written to monitor the patients in a hospital. For each patient, certain information must be maintained, such as the patient's name, room number, temperature, pulse rate, medication, and prognosis for recovery. Because all of this data describes one object (a particular patient), it would be convenient to have a means to refer to the aggregation of data by a single name. In Fortran, an aggregation of data values of different types is called a *structure*. To use a structure, a programmer must first define the type of the structure. After the new type is defined, any number of structures of that type can be declared. This mechanism might seem slightly cumbersome if only one such structure is needed in a program, but usually several are needed. The following is an example of a user-defined type with three components:

```
TYPE PATIENT
   INTEGER              PULSE_RATE
   REAL                 TEMPERATURE
   CHARACTER(LEN = 300) PROGNOSIS
END TYPE PATIENT
```

After type `PATIENT` is defined, objects (structures) of the type can be declared. For example:

```
TYPE(PATIENT) JOHN_JONES, SALLY_SMITH
```

2.3.2 Kind

Some intrinsic types have more than one representation (or kind). Fortran provides both single- and double-precision representations of the real and complex types; chapter 4 describes these data representations in detail. Fortran permits more than one representation for the integer, logical, and character types. The compiler supports more than one representation for the integer and logical types, but they support only a single character type (ASCII). Alternative representations for the integer type permit different ranges of integers.

2.3.3 Dimensionality

Single objects, whether intrinsic or user-defined, are scalar. Even though a structure has components, it is considered to be a scalar. A set of scalar objects, all of the same

type, may be arranged in patterns involving columns, rows, planes, and higher-dimensioned configurations to form arrays. It is possible to have arrays of structures. An array may have a maximum of seven dimensions. The number of dimensions is called the *rank* of the array. It is declared when the array is declared and cannot change. The size of the array is the total number of elements and is equal to the product of the extents in each dimension. The shape of an array is determined by its rank and its extents in each dimension. Two arrays that have the same shape are said to be *conformable*. The following are examples of array declarations:

```
REAL COORDINATES(100, 100)
INTEGER DISTANCES(50)
TYPE(PATIENT) MATERNITY_WARD(20)
```

In Fortran, an array is treated as an object and is allowed to appear in an expression or be returned as a function result. Intrinsic operations involving arrays of the same shape are performed element-by-element to produce an array result of the same shape. There is no implied order in which the element-by-element operations are performed.

A portion of an array, such as an element or section, can be referenced as a data object. An array *element* is a single element of the array and is scalar. An array *section* is a subset of the elements of the array and is itself an array.

2.3.4 Dynamic Data

There are three sorts of dynamic data objects in Fortran: pointers, allocatable arrays, and automatic data objects.

Data objects in Fortran can be declared to have the `POINTER` attribute. Pointer objects must be associated with a target before they can be used in any calculation. This is accomplished by allocation of the space for the target or by assignment of the pointer to an existing target. The association of a pointer with a target can change dynamically as a program is executed. If the pointer object is an array, its size and shape could change dynamically, but its rank is fixed by its declaration. The following is an example of pointer array declaration and allocation:

```
REAL, POINTER :: LENGTHS(:)
ALLOCATE(LENGTHS (200))
```

Following execution of an `ALLOCATE` statement, `LENGTHS` points at (is associated with) an array with 200 elements. The elements as yet have no values.

An array can be declared to have the `ALLOCATABLE` attribute. Space must be allocated for the array before it can be used in any calculation. The array can be

deallocated and reallocated with a different size as the program executes. The size and shape can change at each allocation, but the rank is fixed by the declaration. The following is an example of allocatable array declaration and allocation:

```
REAL, ALLOCATABLE :: LENGTHS(:)
ALLOCATE (LENGTHS(200))
```

The similarities of these examples reflect the similarity of some of the uses of allocatable arrays and pointers, but pointers have more functionality. Pointers can be used to create dynamic data structures, such as linked lists and trees. The target of a pointer can be changed by reallocation or pointer assignment. The extents of an allocatable array can be changed only by deallocating and reallocating the array. If the values of the elements of an allocatable array are to be preserved, a new array must be allocated and the values moved to the new array before the old array is deallocated.

An automatic data object can be an array or a character string. Automatic objects can be declared in a subprogram. These local data objects are created on entry to the subprogram and disappear when the execution of the subprogram completes. They are useful in subprograms for temporary arrays and characters strings whose sizes are different for each reference to the subprogram. The following is an example of a subprogram unit with automatic array TEMP:

```
SUBROUTINE SWAP_ARRAYS(A, B)
   REAL, DIMENSION(:) :: A, B
   REAL, DIMENSION(SIZE(A)) :: TEMP

   TEMP = A
   A = B
   B = TEMP
END SUBROUTINE SWAP_ARRAYS
```

A and B are assumed-shape array arguments; that is, they take on the shape of the actual argument. TEMP is an automatic array that is created the same size as A on entry to subroutine SWAP_ARRAYS. SIZE(3i) is an intrinsic function that is permitted in a declaration statement.

2.4 Program Execution

During program execution, constructs and statements are executed in a prescribed order. Variables become defined with values and can be redefined later in the

execution sequence. Procedures are invoked, perhaps recursively. Space can be allocated and later deallocated. Pointers can change their targets.

2.4.1 Execution Sequence

Program execution begins with the first executable construct in the main program. An executable construct is an instruction to perform one or more of the computational actions that determine the behavior of the program or control the flow of the execution of the program. It can perform arithmetic, compare values, branch to another construct or statement in the program, invoke a procedure, or read from or write to a file or device. When a procedure is invoked, its execution begins with the first executable construct after the entry point in the procedure. On normal return from a procedure invocation, execution continues where it left off. The following are examples of executable statements:

```
     READ(5, *) Z, Y
     X = (4.0 * Z) + BASE
     IF (X > Y) GO TO 100
     CALL CALCULATE(X)
100  Y = Y + 1
```

Unless a branching statement or control construct is encountered, executable statements are executed in the order in which they appear in a program unit until a `STOP`, `RETURN`, or `END` statement is executed. Branch statements specify a change in the execution sequence and consist of the various forms of `GO TO` statements, a procedure reference with alternative return specifiers, and input/output (I/O) statements with branch label specifiers such as `ERR=`, `END=`, and `EOR=` specifiers. The control constructs (`IF`, `CASE`, and `DO`) can cause internal branching implicitly within the structure of the construct. Chapter 8, page 295, discusses control flow within a program in detail.

2.4.2 Definition and Undefinition

Most variables have no value when execution begins; they are considered to be undefined. Exceptions are variables that are initialized in `DATA` statements, type declaration statements, or variables whose type is a structure for which full default initialization is specified in the structure definition (all components must be fully initialized); these are considered to be defined. A variable can acquire a value or change its current value, typically by the execution of an assignment statement or an input statement. Thus it can assume different values at different times, and under some circumstances it can become undefined. *Defined* and *undefined* are the Fortran

terms that are used to specify the definition status of a variable. The *MIPSpro Fortran Language Reference Manual, Volume 2*, describes the events that cause variables to become defined and undefined.

A variable is considered to be defined only if all parts of it are defined. For example, all elements of an array, all components of a structure, or all characters of a character string must be defined; otherwise, the array, structure, or string is undefined. Fortran permits zero-sized arrays and zero-length strings; these are always considered to be defined.

Pointers have both a definition status and an association status. When execution begins, the association status of all pointers is undefined unless one of the following conditions is present:

- The pointers are initialized with a `NULL`(3i) intrinsic procedure in a type declaration statement.
- The pointers are initialized in a `DATA` statement.
- The pointer is a component with default initialization specified.

During execution a pointer can become nullified by the execution of a `NULLIFY` statement, in which case its association status becomes *disassociated*, or it may become associated with a target by the execution of an `ALLOCATE` or pointer assignment statement, in which case its association status becomes *associated*. Even when the association status of a pointer is defined, the pointer is not considered to be defined unless the target with which it is associated is defined. Pointer targets become defined in the same way that other variables becomes defined, typically by the execution of an assignment or input statement. When an allocatable array is allocated by the execution of an `ALLOCATE` statement, it is undefined until some other action occurs that causes it to become defined with values for all array elements.

2.4.3 Dynamic Behavior

Fortran supports the following types of dynamic behavior:

- Recursion
- Allocation and deallocation
- Pointer assignment

Many algorithms can be expressed with the use of *recursion*, which occurs when a subroutine or function references itself, either directly or indirectly. The keyword

RECURSIVE must be present in the SUBROUTINE or FUNCTION statement if the procedure is referenced recursively. Recursive subroutines and functions are described in the *MIPSpro Fortran Language Reference Manual, Volume 2*.

No space exists for a pointer until the pointer is allocated or associated with an existing target. No space exists for an allocatable array until the array is allocated. The rank of array pointers and allocatable arrays is fixed by declaration, but the extents in each dimension (and thus the size of the arrays) is determined during execution by calculation or from input values.

The ALLOCATE and DEALLOCATE statements give Fortran programmers mechanisms to configure objects to the appropriate shape. Only pointers and allocatable arrays can be allocated. It is not possible to deallocate an object unless it was previously allocated, and it is not possible to deallocate a part of an object unless it is a pointer component of a structure. It is possible to inquire whether a pointer is currently associated and whether an allocatable array is currently allocated. Chapter 5, page 115, describes the declaration of pointers and allocatable arrays. Chapter 6, page 183, describes the ALLOCATE and DEALLOCATE statements. The *MIPSpro Fortran Language Reference Manual, Volume 2*, describes the ASSOCIATED(3i) intrinsic inquiry function for pointers and the ALLOCATED(3i) intrinsic inquiry function for allocatable arrays.

Pointers are more flexible than allocatable arrays, but they are also more complicated and less efficient. A pointer need not be an array; it can be a scalar of any type. A pointer need not be associated with allocated space; any object with the TARGET attribute can become a pointer target. A pointer assignment statement is provided to associate a pointer with a target (declared or allocated). It makes use of the symbol => rather than the single character =. In all other respects, pointer assignment statements are executed in the same way that ordinary assignment statements are executed, except that instead of assigning a value they associate a pointer with a target, as is shown in the following example:

```
REAL, TARGET :: VECTOR(100)
REAL, POINTER :: ODDS(:)
   . . .
ODDS => VECTOR(1:100:2)
```

The pointer assignment statement associates ODDS with the odd elements of VECTOR. The following assignment statement defines each odd element of VECTOR with the value 1.5:

```
ODDS = 1.5
```

Later in the execution sequence, pointer `ODDS` could become associated with a different target by pointer assignment or allocation, as long as the target is a one-dimensional, real array. Chapter 7, page 211, describes the pointer assignment statement.

2.5 Summary of Forms

This section shows the forms of the higher-level components of a Fortran program. The notation used in most of the forms is the same as that used to show the syntax forms in all the remaining sections of this manual. The complete Backus-Naur form (BNF), as given in the Fortran standard, is included in the *MIPSpro Fortran Language Reference Manual, Volume 3*.

2.5.1 Program Units

Fortran defines a *program_unit* as follows:

executable_program	**is**	*program_unit* [*program_unit*]...
program_unit	**is**	*main_program*
	or	*external_subprogram*
	or	*module*
	or	*block_data*
main_program	**is**	[*program_stmt*] [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_program_stmt*
external_subprogram	**is**	*functional_subprogram*
	or	*subroutine_subprogram*

function_subprogram	**is**	*function_stmt* [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_function_stmt*
subroutine_subprogram	**is**	*subroutine_stmt* [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_subroutine_stmt*
module	**is**	*module_stmt* [*specification_part*] [*module_subprogram_part*] *end_subroutine_stmt*
block_data	**is**	*block_data_stmt* [*specification_part*] *end_block_data_stmt*
specification_part	**is**	[*use_stmt*]. . . [*implicit_part*] [*declaration_construct*]. . .
implicit_part	**is**	[*implicit_part_stmt*]. . . *implicit_stmt*
implicit_part_stmt	**is**	*implicit_stmt*
	or	*parameter_stmt*
	or	*format_stmt*
	or	*entry_stmt*
declaration_construct	**is**	*derived_type_def*
	or	*interface_block*
	or	*type_declaration_stmt*
	or	*specification_stmt*
	or	*parameter_stmt*
	or	*format_stmt*
	or	*entry_stmt*

	or	*stmt_function_stmt*
execution_part	**is**	*executable_construct* [*execution_part_construct*]...
execution_part_construct	**is**	*executable_construct*
	or	*format_stmt*
	or	*data_stmt*
	or	*entry_stmt*
internal_subprogram_part	**is**	*contains_stmt* *internal_subprogram* [*internal_subprogram*]...
internal_subprogram	**is**	*function_subprogram*
	or	*subroutine_subprogram*
module_subprogram_part	**is**	*contains_stmt* *module_subprogram* [*module_subprogram*]...
module_subprogram	**is**	*function_subprogram*
	or	*subroutine_subprogram*
specification_stmt	**is**	*access_stmt*
	or	*allocatable_stmt*
	or	*common_stmt*
	or	*data_stmt*
	or	*dimension_stmt*
	or	*equivalence_stmt*
	or	*external_stmt*
	or	*intent_stmt*
	or	*intrinsic_stmt*
	or	*namelist_stmt*
	or	*optional_stmt*
	or	*pointer_stmt*

	or	*save_stmt*
	or	*target_stmt*
executable_construct	**is**	*action_stmt*
	or	*case_construct*
	or	*do_construct*
	or	*forall_construct*
	or	*if_construct*
	or	*where_construct*
action_stmt	**is**	*allocate_stmt*
	or	*arithmetic_if_stmt*
	or	*assign_stmt*
	or	*assigned_goto_stmt*
	or	*assignment_stmt*
	or	*backspace_stmt*
	or	*call_stmt*
	or	*close_stmt*
	or	*computed_goto_stmt*
	or	*continue_stmt*
	or	*cycle_stmt*
	or	*deallocate_stmt*
	or	*endfile_stmt*
	or	*end_function_stmt*
	or	*end_program_stmt*
	or	*end_subroutine_stmt*
	or	*exit_stmt*
	or	*forall_stmt*
	or	*goto_stmt*
	or	*if_stmt*

or *inquire_stmt*

or *nullify_stmt*

or *open_stmt*

or *pause_stmt*

or *pointer_assignment_stmt*

or *print_stmt*

or *read_stmt*

or *return_stmt*

or *rewind_stmt*

or *stop_stmt*

or *where_stmt*

or *write_stmt*

2.5.2 Main Program

Typically, a main program takes the following form:

```
[ PROGRAM program_name ]
   [ specification_part ]
   [ execution_part ]
   [ CONTAINS
   internal_subprogram
   [ internal_subprogram ] ... ]
   END [ PROGRAM [ program_name ] ]
```

2.5.3 External Subprogram

Typically, an external subprogram takes one of the following forms:

```
function_stmt
   [ specification_part ]
   [ execution_part ]
   [ CONTAINS
   internal_subprogram
   [ internal_subprogram ] ... ]
   end_function_stmt
```

```
subroutine_stmt
   [ specification_part ]
   [ execution_part ]
   [ CONTAINS
   internal_subprogram
   [ internal_subprogram ] ... ]
   end_subroutine stmt
```

2.5.4 Module

Typically, a module takes the following form:

```
MODULE module_name
   [ specification_part ]
   [ CONTAINS
   module_subprogram
   [ module_subprogram ] ... ]
   END [ MODULE [ module_name ] ]
```

2.5.5 Block Data

Typically, a block data program unit takes the following form:

```
BLOCK DATA [ block_data_name ] [ specification_part ]
   END [ BLOCK DATA [ block_data_name ] ]
```

2.5.6 Internal Subprogram

Typically, an internal subprogram takes one of the following forms:

function_stmt [*specification_part*] [*execution_part*] *end_function_stmt*

subroutine_stmt [*specification_part*] [*executable_part*] *end_subroutine_stmt*

2.5.7 Procedure Headings

The FUNCTION and SUBROUTINE statements are defined as follows:

[*prefix*] FUNCTION *function_name* ([*dummy_argument_list*])
  [RESULT (*result_name*)]

[*prefix*] SUBROUTINE *subroutine_name* [([*dummy_argument_list*])]

2.5.8 Procedure Endings

The END FUNCTION and END SUBROUTINE statements are defined as follows:

END [FUNCTION [*function_name*]]

END [SUBROUTINE [*subroutine_name*]]

2.5.9 Specification Constructs

A specification part generally can contain any of the following. Some program units or subprograms may not allow some of these specification constructs. See the descriptions for each program unit or subprogram description for restrictions.

- USE statements
- PARAMETER statements
- FORMAT statements
- ENTRY statements
- Derived type definitions
- Interface blocks
- Type declaration statements

- Other specification statements

2.5.10 Derived-type Definition

Typically, derived-type definitions take the following form:

```
TYPE [ [, access_spec] :: ] type_name
   [ PRIVATE ]
   [ SEQUENCE ]
   type_spec [ [, component_attr_spec_list ] :: ] component_decl_list ...
   [ type_spec [ [, component_attr_spec_list ] :: ]
      component_decl_list ... ] ...
   END TYPE [ type_name ]
```

2.5.11 Interface Block

Typically, an interface block takes the following form:

```
INTERFACE [ generic_spec ]
   [ interface_body ] ...
   [ MODULE PROCEDURE procedure_name_list ] ...
   END INTERFACE
```

2.5.12 Specification Statements

This section lists the general forms for specification statements. The BNF used here is an abbreviated format and is used only in this section. The specific formats are described in later sections of this manual. The specification statements are as follows:

```
ALLOCATABLE [ :: ] allocatable_array_list

AUTOMATIC automatic_list ...

COMMON [ / [ common_block_name ] / ] common_block_object_list

DATA data_statement_object_list / data_statement_value_list /

DIMENSION array_dimension_list

EQUIVALENCE equivalence_set_list

EXTERNAL external_name_list

FORMAT ([ format_item_list ])

IMPLICIT implicit_spec

INTENT (intent_spec) [ :: ] dummy_argument_name_list

INTRINSIC intrinsic_procedure_name_list

NAMELIST / namelist_group_name / namelist_group_object_list

OPTIONAL [ :: ] optional_object_list

PARAMETER (named_constant_definition_list)

POINTER [ :: ] pointer_list

PUBLIC [ [ :: ] module_entity_name_list ]

PRIVATE [ [ :: ] module_entity_name_list ]

SAVE [ [ :: ] saved_object_list ]

TARGET [ :: ] target_list
```

```
USE module_name [ , rename_list ]

USE module_name , ONLY : [ access_list ]

type_spec [ [, attr_spec ] ... :: ] object_declaration_list

VOLATILE entity_decl_list
```

ANSI/ISO: The Fortran standard does not specify the AUTOMATIC or VOLATILE attributes.

2.5.13 Type Declaration Statements

Typically, the type declaration statements take the following form:

```
TYPE (type_name)

LOGICAL [ ([ KIND= ] kind_parameter) ]

CHARACTER*char_length

CHARACTER (KIND= kind_parameter [, LEN= length_parameter ])

CHARACTER ([ LEN= ] length_parameter [, [KIND= ] kind_parameter ])

CHARACTER [ ([ LEN= ] length_parameter) ]

COMPLEX [ ([ KIND= ] kind_parameter) ]

DOUBLE PRECISION

REAL [ ([ KIND= ] kind_parameter) ]

INTEGER [ ([ KIND= ] kind_parameter) ]
```

2.5.14 Attribute Specifications

Section 2.5.12, page 30, introduced *attr_spec*. Typically, attribute specifications take the following form:

```
ALLOCATABLE

AUTOMATIC

DIMENSION (array_spec)

EXTERNAL

INTENT (intent_spec)

INTRINSIC

OPTIONAL

PARAMETER

POINTER

PRIVATE

PUBLIC

SAVE

TARGET

VOLATILE
```

ANSI/ISO: The Fortran standard does not specify the `AUTOMATIC` or `VOLATILE` attributes.

2.5.15 Execution Part

An execution part can contain the following:

- *action_statement*
- *case_construct*
- *do_construct*
- *if_construct*
- *where_construct*

2.5.16 Action Statements

Typically, action statements take the following form:

```
ALLOCATE (allocation_list [ , STAT= scalar_integer_variable ])

ASSIGN label TO scalar_integer_variable

BACKSPACE external_file_unit

BACKSPACE (position_spec_list)

BUFFER IN (id, mode) (start_loc, end_loc )

BUFFER OUT (id, mode) (start_loc, end_loc )

CALL subroutine_name [ ([ actual_argument_spec_list ]) ]

CLOSE (close_spec_list)

CONTINUE

CYCLE [ do_construct_name ]

DEALLOCATE (name_list [ , STAT= scalar_integer_variable ])

ENDFILE external_file_unit

ENDFILE (position_spec_list)

EXIT [ do_construct_name ]

FORALL forall_header forall_assignment_stmt

GO TO label

GO TO (label_list) [ , ] scalar_integer_expression

GO TO scalar_integer_variable [ [ , ] (label_list) ]
```

```
IF (scalar_logical_expression) action_statement

IF (scalar_numeric_expression) label, label, label

INQUIRE (inquire_spec_list) [ output_item_list ]

NULLIFY (pointer_object_list)

OPEN (connect_spec_list)

PAUSE [ access_code ]

PRINT format [, output_item_list ]

READ (io_control_spec_list) [ input_item_list ]

READ format [, input_item_list ]

RETURN [ scalar_integer_expression ]

REWIND external_file_unit

REWIND (position_spec_list)

STOP [ access_code ]

WHERE (array_logical_expression) array_assignment_statement

WRITE (io_control_spec_list) [ output_item_list ]

pointer_variable => target_expression

variable = expression
```

2.5.17 CASE Construct

Typically, CASE constructs take the following form:

```
SELECT CASE (case_expr)
   [ CASE case_selector
       block ] ...
   [ CASE DEFAULT
       block ]
   END SELECT
```

2.5.18 DO Construct

Typically, DO constructs take the following form:

```
DO [ label ]
    block
   end_do

DO [ label ] [ , ] do_variable = scalar_numeric_expr,
      scalar_numeric_expr [ , scalar_numeric_expr ]
    block
   end_do

DO [ label ] [ , ] WHILE (scalar_logical_expr)
    block
   end_do
```

2.5.19 IF Construct

Typically, IF constructs take the following form:

```
IF (scalar_logical_expr) THEN
       block
   [ ELSE IF (scalar_logical_expr) THEN
       block ] ...
   [ ELSE
       block ]
   END IF
```

2.5.20 FORALL Construct

Typically, FORALL constructs take the following form:

```
FORALL forall_header
   [ assignment_stmt ] |
   [ forall_assignment_stmt ] |
   [ forall_construct ] |
   [ forall_stmt ] |
   [ pointer_assignment_stmt ] |
   [ where_construct ] |
   [ where_stmt ]
   END FORALL
```

2.5.21 WHERE Construct

Typically, WHERE constructs take the following form:

```
[where_construct_name:] WHERE (mask_expr)
       [ where_body_construct ] ...
   [ ELSEWHERE (mask_expr) [ where_construct_name ]
       [ where_body_construct ] ...]
   [ ELSEWHERE
       [ where_body_construct ] ...]
   END WHERE [where_construct_name]
```

2.6 Ordering Requirements

Within program units and subprograms, there are ordering requirements for statements and constructs. The syntax rules in the previous section do not fully describe the ordering requirements. They are illustrated in both Figure 2-2, page 39, and Figure 2-3, page 40. Generally, data declarations and specifications must precede executable constructs and statements, although FORMAT, DATA, NAMELIST, and ENTRY statements can appear among the executable statements. USE statements, if any, must appear first. Internal or module subprograms, if any, must appear last following a CONTAINS statement.

ANSI/ISO: The Fortran standard does not allow `NAMELIST` statements to be intermixed with executable statements and constructs. The compiler eases this restriction.

In Figure 2-2, page 39, a vertical line separates statements and constructs that can be interspersed; a horizontal line separates statements that must not be interspersed.

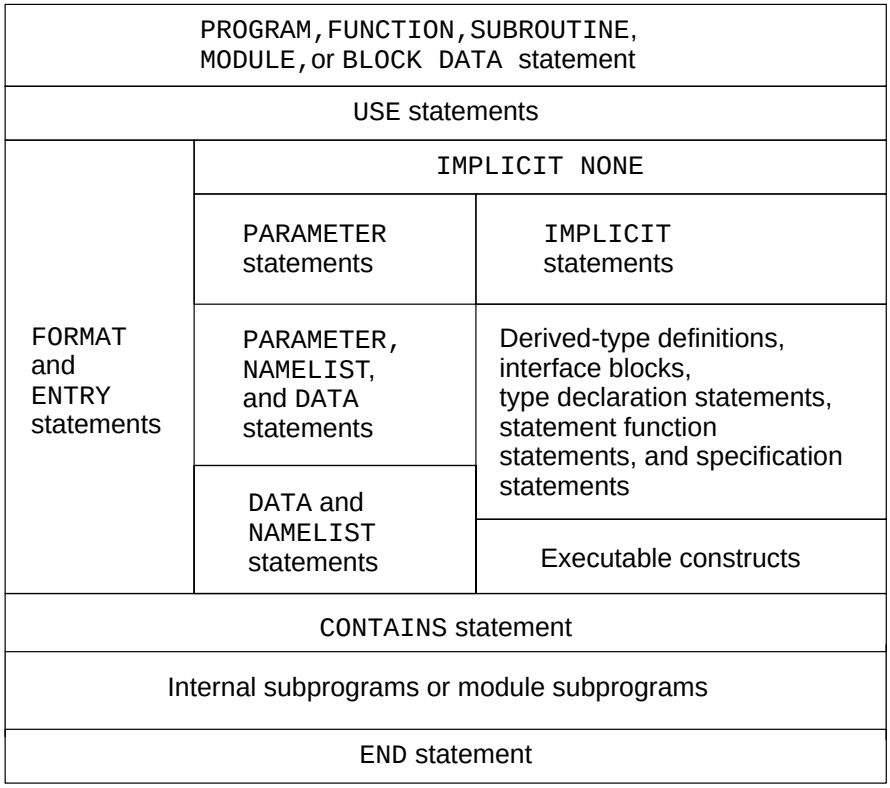

Figure 2-2 Requirements on statement ordering

There are restrictions on the places where some statements may appear. Figure 2-3, page 40, summarizes these restrictions.

Note: Miscellaneous declarations are PARAMETER statements, IMPLICIT statements, type declaration statements, and specification statements.

Scoping unit	Main program	Module	Block data	External subprog	Module subprog	Internal subprog	Interface body
USE statement	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ENTRY statement	No	No	No	Yes	Yes	No	No
FORMAT statement	Yes	No	No	Yes	Yes	Yes	No
Misc. declarations (see note)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DATA statement	Yes	Yes	Yes	Yes	Yes	Yes	No
Derived-type definition	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface block	Yes	Yes	No	Yes	Yes	Yes	Yes
Statement function	Yes	No	No	Yes	Yes	Yes	No
Executable statement	Yes	No	No	Yes	Yes	Yes	No
CONTAINS	Yes	Yes	No	Yes	Yes	No	No

Note: Miscellaneous declarations are PARAMETER statements, IMPLICIT statements, type declaration statements, and specification statements.

a10843

Figure 2-3 Restrictions on the appearance of statements

2.7 Example Fortran Program

The following simple Fortran program consists of one program unit, the main program. Three data objects are declared: H, T, and U. These become the loop indexes in a triply-nested loop structure that contains an IF statement that conditionally executes an I/O statement.

```
PROGRAM SUM_OF_CUBES
! This program prints all 3-digit numbers that
! equal the sum of the cubes of their digits.
INTEGER :: H, T, U
DO H = 1, 9
```

```
  DO T = 0, 9
    DO U = 0, 9
    IF (100*H + 10*T + U == H**3 + T**3 + U**3) &
      PRINT "(3I1)", H, T, U
    END DO
  END DO
END DO
END PROGRAM SUM_OF_CUBES
```

This Fortran program produces the following output:

```
153
370
371
407
```

Language Elements and Source Form

This chapter describes the language elements that a Fortran statement can contain. Language elements consist of lexical tokens, which include names, keywords, operators, and statement labels. Rules for forming lexical tokens from the characters in the Fortran character set are also presented.

3.1 Compiler Character Set

The compiler character sets contain the following characters:

- The Fortran character set, which includes the 26 uppercase letters of the alphabet, the corresponding 26 lowercase letters, and several special characters.
- The numerical digits `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`.
- The underscore character (_).
- The newline and tab characters, which are control characters and have no graphic representation. These characters are part of the processor-dependent control characters as allowed by the standard.

Table 3-1, page 43, shows the special characters of the character sets.

Table 3-1 Special characters

Graphic	Character name	Graphic	Character name
	Blank	:	Colon
=	Equals	!	Exclamation point
+	Plus	"	Quotation mark
-	Minus	%	Percent
*	Asterisk	&	Ampersand
/	Slash	;	Semicolon
(	Left parenthesis	<	Less than

Graphic	Character name	Graphic	Character name
)	Right parenthesis	>	Great than
,	Comma	?	Question mark
'	Apostrophe	$	Currency symbol
.	Decimal point or period		

The 52 letters define the syntax class *letter*.

In most cases, the compiler is not case sensitive; lowercase letters are considered the same as uppercase letters. In the following cases, however, lowercase and uppercase letters are considered to have different data values:

- Within a character constant
- Within a quotation mark, apostrophe, or H edit descriptor
- Within a Hollerith constant (nonstandard)

ANSI/ISO: Hollerith data is not included in the Fortran standard.

The compiler considers the following two statements to be equivalent:

```
PRINT *, N
Print *, n
```

The compiler distinguishes between uppercase and lowercase letters in the FILE= or NAME= specifier in an OPEN or an INQUIRE statement.

The digits 0 through 9 define the syntax class *digit*. The digits are assumed to be decimal numbers when used to describe a numeric value, except in binary, octal, and hexadecimal (BOZ) literal constants or input/output (I/O) records corresponding to B, O, or Z edit descriptors.

For example, in the following DATA statement, the digits of the first constant are decimal digits, those of the second constant are binary digits, and those of the third are hexadecimal digits:

```
DATA  X, I, J / 4.89, B'1011', Z'BAC91' /
```

The underscore can be used to make names more readable. For example, in the identifier NUMBER_OF_CARS, each underscore is used to separate the obvious English words. It is a significant character in any name. An underscore cannot be used as the first character of a name, but it can be the last character. An underscore is also used to separate the kind value from the actual value of a literal constant (for example, 123_2).

There are 22 special characters used for operators like multiply and add. They are also used as separators or delimiters in Fortran statements. Separators and delimiters make the form of a statement unambiguous.

Fortran's treatment of uppercase and lowercase letters can lead to portability problems when calling subprograms written in other languages. The problem occurs because the standard does not specify the case of letters used for external names. The following program fragment illustrates the problem:

```
EXTERNAL  FOO
  . . .
CALL  FOO
  . . .
END
```

The compiler converts external names into lowercase and append an underscore, so the compiler would use foo_ as the external name. If the subprogram were written in C, which is case sensitive, and if foo were written in lowercase, the external name used in C would then be different from the name produced by the compiler.

The NAME compiler directive allows you to specify a case-sensitive external name in a Fortran program. You can use this directive, for example, when writing calls to C routines. For more information on the NAME directive, see the *MIPSpro Fortran 90 Commands and Directives Reference Manual*.

3.2 Lexical Tokens

A statement is constructed from low-level syntax. The low-level syntax describes the basic language elements, called *lexical tokens,* in a statement. A lexical token is the smallest meaningful unit of a statement and can consist of 1 or more characters. Tokens are names, keywords, literal constants (except for complex literal constants), labels, operator symbols, comma, =, =>, :, ::, ;, %, and delimiters. A literal of type complex consists of several tokens. Examples of operator symbols are + and //.

Delimiters are pairs of symbols that enclose parts of a statement. The following symbol pairs are delimiters:

```
/ ... /
( ... )
(/ ... /)
```

In the following statements, the slashes distinguish the value list from the object list in a DATA statement, the parentheses are delimiters that mark the beginning and end of the argument list in the CALL statement, and the pairs (/ and /) mark the beginning and end of the elements of an array constructor:

```
DATA  X, Y / 1.0, -10.2/
CALL PRINT_LIST(LIST, SIZE)
VECTOR = (/ 10, 20, 30, 40 /)
```

3.2.1 Statement Keywords

Statement keywords appear in uppercase letters in the syntax rules. Some statement keywords identify the statement, such as in the following DO statement:

```
DO I = 1, 10
```

DO is a statement keyword that identifies the DO statement. Other keywords identify parts of a statement such as ONLY in a USE statement or WHILE in one of the forms of a DO construct, as follows:

```
DO WHILE( .NOT. FOUND )
```

Others specify options in the statement such as IN, OUT, or INOUT in the INTENT statement.

There are three statements in Fortran that have no statement keyword. They are the assignment statement, the pointer assignment statement, and the statement function.

Some sequences of capital letters in the formal syntax rules are not statement keywords. For example, EQ, in the lexical token .EQ., and EN, as an edit descriptor, are not statement keywords.

A dummy *argument keyword*, a different sort of keyword, is discussed in the *MIPSpro Fortran Language Reference Manual, Volume 2*.

3.2.2 Names

Variables, named constants, program units, common blocks, procedures, arguments, constructs, derived types (types for structures), namelist groups, structure components, dummy arguments, and function results are among the elements in a program that have a name. Fortran permits up to 31 characters in a name.

	character	**is**	*alphanumeric_character*
		or	*special_character*
	alphanumeric_character	**is**	*letter*
		or	*digit*
		or	*underscore*
EXT		**or**	*currency_symbol*
EXT		**or**	*at_sign*
	underscore	**is**	_
EXT	*currency_symbol*	**is**	$
EXT	*at_sign*	**is**	@
	name	**is**	*letter* [*alphanumeric_character*]...

Note: The @ sign is not supported by the compiler.

A name must begin with a letter and can consist of letters, digits, and underscores. The compiler allows you to use the underscore (_), and dollar sign ($) in a name, but none of these can be the first character of a name. Use $ is not recommended, however, because it is intended for internal use.

The following are examples of names:

```
A
CAR_STOCK_NUMBER
A__BUTTERFLY
Z_28
TEMP_
```

ANSI/ISO: The Fortran standard does not allow the dollar sign ($) character in a name. The at sign (@) is not included in the standard character set.

3.2.3 Constants

A *constant* is a syntactic notation for a value. The value can be of any intrinsic type; that is, it can be a numeric (integer, real, or complex) value, a character value, or a logical value. A *constant* is defined as follows:

	constant	**is**	*literal_constant*
		or	*named_constant*
	literal_constant	**is**	*int_literal_constant*
		or	*real_literal_constant*
		or	*logical_literal_constant*
		or	*complex_literal_constant*
		or	*char_literal_constant*
		or	*boz_literal_constant*
EXT		**or**	*typeless_constant*
EXT	*typeless_constant*	**is**	*octal_typeless_constant*
		or	*hexadecimal_typeless_constant*
		or	*binary_typeless_constant*
EXT	*octal_typeless_constant*	**is**	*digit* [*digit*...] B
		or	O" *digit* [*digit*...] "
		or	O' *digit* [*digit*...] '
		or	" *digit* [*digit*...] "O
		or	' *digit* [*digit*...] 'O
EXT	*hexadecimal_typeless_constant*	**is**	X' *hex_digit* [*hex_digit*...]'
		or	X" *hex_digit* [*hex_digit*...] "

		or	' *hex_digit* [*hex_digit*...] 'X
		or	" *hex_digit* [*hex_digit*...] "X
		or	Z' *hex_digit* [*hex_digit*...] '
		or	Z" *hex_digit* [*hex_digit*...] "
EXT	*binary_typeless_constant*	**is**	B' *digit* [*digit*...] '
		or	B" *digit* [*digit*...] "
	constant	**is**	*literal_constant*
		or	*named_constant*
	named_constant	**is**	*name*
	int_constant	**is**	*constant*
	char_constant	**is**	*constant*

The following notes pertain to the preceding format:

- *digit* must have one of the values 0 through 7 in *octal_typeless_constant*
- *digit* must have a value of 0 or 1 in *binary_typeless_constant*
- The B, O, X, and Z characters can be in uppercase or lowercase.

A value that does not have a name is a *literal constant*. The following are examples of literal constants:

```
1.23
400
( 0.0, 1.0 )
"ABC"
B'0110110'
.TRUE.
```

The B, O, and Z constants are described in Section 4.3.1.3, page 77.

No literal constant can be array-valued or of derived type. Section 4.3, page 75 describes the formats of literal constants in more detail.

A value that has a name is called a *named constant* and can be of any type, including a derived type. A named constant can also be array-valued. In the following statements, X_AXIS and MY_SPOUSE are examples of named constants:

```
REAL, DIMENSION(2), PARAMETER :: X_AXIS = (/ 0.0, 1.0 /)
TYPE(PERSON), PARAMETER :: MY_SPOUSE = PERSON( 39, 'PAT' )
```

Note that the entity on the right of the equal sign is not itself a constant but a constant expression. The forms for defining named constants are described in more detail in Section 5.5.1, page 150.

3.2.4 Operators

Operators are used with operands in expressions to produce other values. The following are examples of intrinsic operators:

Operator	Representation
`*`	Multiplication of numeric values
`//`	Concatenation of character values
`==`	Comparison for equality (same as `.EQ.`)
`.OR.`	Logical disjunction
`.NOT.`	Logical negation

The *intrinsic_operator*s are defined as follows:

intrinsic_operator	**is**	*power_op*
	or	*mult_op*
	or	*add_op*
	or	*concat_op*
	or	*rel_op*
	or	*not_op*
	or	*and_op*
	or	*or_op*
	or	*equiv_op*
power_op	**is**	**
mult_op	**is**	*

		or	/
	add_op	**is**	+
		or	-
	concat_op	**is**	//
	rel_op	**is**	.EQ.
		or	.NE.
		or	.LT.
		or	.LE.
		or	.GT.
		or	.GE.
EXT		**or**	.LG.
		or	==
		or	/=
		or	<
		or	<=
		or	>
		or	>=
EXT		**or**	<>
	not_op	**is**	.NOT.
EXT		**or**	.N.
	and_op	**is**	.AND.
EXT		**or**	.A.
	or_op	**is**	.OR.
EXT		**or**	.O.
	equiv_op	**is**	.EQV.
		or	.NEQV.
EXT	*exclusive_disjunct_op*	**is**	.XOR.
EXT		**or**	.X.

The abbreviations .A., .O., .N., and .X. are synonyms for .AND., .OR., .NOT., and .XOR., respectively. If the abbreviated operator is overloaded in an interface block as a defined operator, the abbreviated form of the intrinsic operator cannot be used in any scope in which the defined operator is accessible.

ANSI/ISO: The Fortran standard does not specify the .A., .O., or .N., abbreviations for the logical operators, nor does it specify the .XOR. operator or its .X. abbreviation. The Fortran standard does not specify the .LG. or <> operators.

You can define operators in addition to the intrinsic operators. User-defined operators begin with a period (.), followed by a sequence of up to 31 letters, and end with a period (.), except that the letter sequence must not be the same as any intrinsic operator defined by the Fortran standard or the logical constants .FALSE. or .TRUE.

3.2.5 Statement Labels

A *label* can be used to identify a statement. A label consists of 1 to 5 decimal digits, one of which must be nonzero. If a statement has a label, it is uniquely identified and the label can be used in DO constructs, CALL statements, branching statements, and I/O statements.

Leading zeros in a label are not significant. In other words, the labels 020 and 20 are considered to be the same label. A statement label is local to a scoping unit. This means that a program unit can contain more than one definition of the same label as long as the labels are defined in different scoping units. For example, a main program and a contained internal subprogram can both define the same statement label. The cases in which duplicate labels can be used in the same program unit are explained as part of the general treatment of the scope of entities in the *MIPSpro Fortran Language Reference Manual, Volume 2*.

The following are examples of statements with labels:

```
100 CONTINUE
 21 X = X + 1.2
101 FORMAT (1X, 2F10.2)
```

Fortran syntax does not permit a statement with no content. This is sometimes referred to as a blank statement. Such a statement is always treated as a comment; therefore, if such a statement is created, it must not be labeled. For example, each of the following lines is nonstandard Fortran. They generate an error message from the compiler:

```
10
X=0;101;
```

3.3 Source Form

A Fortran program consists of statements, comments, and `INCLUDE` lines. This collection of statements, comments, and `INCLUDE` lines is called *source text*. A statement consists of one or more lines of source text and is constructed from low-level syntax.

The lines within a program unit (except comment lines) and the order of the lines are, in general, significant.

Because all program units terminate with their own `END` statement, lines following such an `END` statement are never part of the preceding program unit; they are part of the program unit that follows. `END` statements can be continued.

ANSI/ISO: The Fortran standard does not describe `END` statement continuation.

There are two source forms for writing source text: free source form and fixed source form.

Note: The Fortran standard has declared fixed source form to be obsolescent. The preferred alternative is free source form.

Fixed source form is the default for Fortran source files with a `.f` or `.F` suffix. Free source form is the default source for Fortran source files with a `.f90` or `.F90` suffix. The compiler allows you to use the `FIXED` and `FREE` compiler directives to switch from one source form to the other within a program unit. The `f90`(1) command line allows you to override the source form implied by the input file suffix. See the *MIPSpro Fortran 90 Commands and Directives Reference Manual*, for information on the `f90`(1) command line and on the `FIXED` and `FREE` compiler directives. Section 3.4, page 63, describes a way to write Fortran statements so that the source text is acceptable to both free and fixed source forms.

ANSI/ISO: The Fortran standard does not describe compiler directives or command lines.

Characters that form the value of a character literal constant, a Hollerith constant, or a character string edit descriptor (quotation mark, apostrophe, or H edit descriptor) are in a *character context.* The characters in a character context do not include the delimiters used to indicate the beginning and end of the character constant or string. Also, the ampersands (&) in free source form, which are used to indicate that a character string is being continued and used to indicate the beginning of the character string on the continued line, are never part of the character string value and thus are not in character context.

ANSI/ISO: The Fortran standard does not describe Hollerith constants.

The rules that apply to characters in a character context are different from the rules that apply to characters in other contexts. For example, blanks are always significant in a character context but are never significant in other parts of a program written using fixed source form. The following code fragment illustrates this:

```
CHAR = CHAR1 // "Mary K. Williams"
! The blanks within the character string
! (within the quotation marks) are significant.
! The next two statements are equivalent
! in fixed source form.
DO2I=1,N
DO 2 I = 1, N
```

Comments can contain any printable character.

3.3.1 Free Source Form

In free source form, the only restriction on statement positioning within a line is that the line itself cannot contain more than 132 characters. A blank character is significant and may be required to separate lexical tokens.

Blank characters are significant everywhere, but a sequence of blank characters outside a character context is treated as a single blank character. They can be used freely between tokens and delimiters to improve the readability of the source text. For example, the following two statements are interpreted identically:

```
SUM=SUM+A(I)
SUM = SUM + A (I)
```

Each line can contain from 0 through 132 characters.

The exclamation mark (!), not in character context, indicates the beginning of a comment that ends with the end of the line. A line can consist of nothing but a comment. Comments, including the !, are ignored and do not alter the interpretation of statements in any way.

The compiler supports compiler directives. *Compiler directives* are lines inserted into source code that specify actions to be performed by the compiler. Other compilers may treat compiler directive lines as comments. For more information on compiler directives, see the *MIPSpro Fortran 90 Commands and Directives Reference Manual*.

A line whose first nonblank character is an exclamation mark is called a *comment line.*

The following is an example of a comment line:

```
! Begin the next iteration.
```

The following is an example of a statement with a trailing comment:

```
ITER = ITER + 1   ! Begin the next iteration.
```

An ampersand (&), not in a character context, is a continuation symbol and must be followed by one of the following:

- Zero or more blanks.
- A comment and the end of the line. If the line following this line is not a comment line; it is a *continuation line*.

The following is an example of a continued line and a continuation line:

```
FORCE = G * MASS1 *  & ! This is a continued line.
       MASS2 / R**2 ! This is a continuation line.
```

The compiler allows a statement to be continued with up to 99 continuation lines.

ANSI/ISO: If you are using free source form, the Fortran standard allows no more than 39 continuation lines within a statement.

Comment lines cannot be continued. That is, the ampersand as the last character in a comment is part of the comment and does not indicate continuation.

A line with only blank characters or with no characters is treated as a comment line.

More than one statement or partial statement can appear on a line. The statement separator is the semicolon (;), provided it is not in a character context. Multiple

successive semicolons on a line with or without blanks intervening are considered as a single separator. The end of a line is also a statement separator, but a semicolon at the end of a line that is not part of a comment is considered as a single separator. Essentially, a null statement is a legal Fortran statement. The following statements show use of the semicolon:

```
! The semicolon is a statement separator.
X = 1.0;  Y = 2.0
! However, the semicolon below, at the end of a
! line, is not treated as a separator and is
! ignored.
Z = 3.0;
! Also, consecutive semicolons are treated as one
! semicolon, even if blanks intervene.
Z = 3.0; ; W = 4.0
```

A label can appear before a statement, provided that it is not part of another statement, but it must be separated from the statement by at least one blank. Examples:

```
10 FORMAT(10X,2I5)                ! 10 is a label
   IF (X == 0.0) 200 Y = SQRT(X)  ! Label 200 is
                                  ! not allowed.
```

Any printable character can be used in character literal constants and character string edit descriptors.

The compiler supports only the ASCII character set. The *MIPSpro Fortran Language Reference Manual, Volume 3*, describes the ASCII character set.

3.3.1.1 The Ampersand (&) As a Continuation Symbol

The ampersand (&) is used as the continuation symbol in free source form. If it is the last nonblank character in a line after any comments are deleted and it is not in a character context, the statement is continued on the next line that does not begin with a comment. If the first nonblank character on the continuing line is an ampersand, the statement continues after the ampersand; otherwise, the statement continues with the first position of the line. The ampersand or ampersands used as the continuation symbols are not considered part of the statement. For example, the following statement takes two lines (one continuation line) because it is too long to fit on one line:

```
STOKES_LAW_VELOCITY = 2 * GRAVITY * RADIUS ** 2 *  &
      (DENSITY_1 - DENSITY_2) / (9 * COEFF_OF_VISCOSITY)
```

The leading blanks on the continued line are included in the statement and are allowed in this case because they are between lexical tokens.

The double ampersand convention must be used to continue a name, a character constant, or a lexical token consisting of more than 1 character split across lines. The following statement is the same statement as in the previous example:

```
STOKES_LAW_VELOCITY = 2 * GRAVITY * RADIUS ** 2 * (DEN&
      &SITY_1 - DENSITY_2) / (9 * COEFF_OF_VISCOSITY)
```

However, splitting names across lines makes the code difficult to read and is not recommended.

Ampersands can be included in a character constant. Only the last ampersand on the line is the continuation symbol, as illustrated in the following example:

```
LAWYERS = "Jones & Clay & &
&Davis"
```

The value of this constant is `Jones & Clay & Davis`. The first two ampersands are in a character context; they are part of the value of the character string.

`END` statements cannot be continued.

3.3.1.2 Blanks As Separators

Blanks in free source form cannot appear within tokens, such as names or symbols consisting of more than 1 character, but blanks can be used freely in format specifications. For example, blanks cannot appear between the characters of multicharacter operators such as `**` and `.NE.`. Format specifications are an exception because blanks can appear within edit descriptors such as `BN`, `SS`, or `TR`.

The compiler treats tabs and single blanks as equivalent in free source form, except in character literal strings.

ANSI/ISO: The Fortran standard does not allow tabs in Fortran source files.

A blank must be used to separate a statement keyword, name, constant, or label from an adjacent name, constant, or label. For example, the blanks in the following statements are required:

```
INTEGER SIZE
PRINT 10,N
DO I=1,N
```

Adjacent keywords require a blank separator in some cases (for example, CASE DEFAULT). In other cases, two adjacent keywords can be written either with or without intervening blanks (for example, BLOCK DATA).

Blank separators are optional in the following keywords:

- BLOCK DATA
- DOUBLE PRECISION
- ELSE IF
- END BLOCK DATA
- END DO
- END FILE
- END IF
- END INTERFACE
- END MODULE
- END PROGRAM
- END SELECT
- END SUBROUTINE
- END TYPE
- END WHERE
- GO TO
- IN OUT
- SELECT CASE

Blank separators are mandatory in the following keywords:

- `CASE DEFAULT`
- `DO WHILE`
- `IMPLICIT` *type_spec*
- `IMPLICIT NONE`
- `INTERFACE ASSIGNMENT`
- `INTERFACE OPERATOR`
- `MODULE PROCEDURE`
- `RECURSIVE FUNCTION`
- `RECURSIVE SUBROUTINE`
- `RECURSIVE` *type_spec*
- *type_spec* `FUNCTION`
- *type_spec* `RECURSIVE`

Blanks are not mandatory in all cases, but blank separators between statement keywords make the source text more readable and clarify the statements. Generally, if common rules of English text are followed, everything will be correct. For example, blank separators in the following statements make them quite readable, and the blanks in `DOUBLE PRECISION` and `END FUNCTION` are optional:

```
RECURSIVE FUNCTION F(X)
DOUBLE PRECISION X
END FUNCTION F
```

3.3.1.3 Sample Program, Free Source Form

The following is a sample program in free source form. Note that the numbers and the line at the top are not part of the program; the vertical bars to the left of the program are also not part of the program. These graphics are included to show the columns this program uses in free source form.

```
 123456789.......
 ------------------------------------------------
|PROGRAM LEFT_RIGHT
```

```
| REAL  X(5), Y(5)
|       ! Print arrays X and Y
|       PRINT 100, X, Y
|       100 FORMAT (F10.1, F10.2, F10.3, F10.4,  &
|                   F10.5)
|   . . .
|END
```

3.3.2 Fixed Source Form

Fixed source form is position oriented on a line using the historical Fortran conventions for position that were used on punched cards.

Note: The Fortran standard has declared fixed source form to be obsolescent. The preferred alternative is free source form.

By default, statements or parts of statements must be written between positions 7 and 72, inclusive. The `f90`(1) command line includes options that allow you to specify different line lengths. For information on specifying different line lengths, see the *MIPSpro Fortran 90 Commands and Directives Reference Manual*.

Regardless of the command line specification, character positions 1 through 6 are reserved for special purposes.

ANSI/ISO: The Fortran standard does not allow a compiler to recognize characters beyond column 72.

Blanks are not significant in fixed source form except in a character context. For example, the following two statements are identical:

```
D O  10  I = 1, L O O P E N D
DO 10 I = 1, LOOPEND
```

A `C` or `*` in position 1 identifies a comment. In this case, the entire line is a comment and is called a *comment line*. A `!` in any position except position 6 and not in a character context indicates that a comment follows to the end of the line. Comments are not significant.

A line with only blank characters or with no characters is treated as a comment line.

Multiple statements on a line are separated by one or more semicolons. Semicolons can occur at the end of a line, and these are ignored.

Any character (including `!` and `;`) other than blank or 0 in position 6 indicates that the line is a continuation of the previous line. Such a line is called a *continuation line.* The text on the continuation line begins in position 7. There can be no more than 99 continuation lines for one statement in fixed source form. The first line of a continued statement is called the *initial line*.

ANSI/ISO: If you are using fixed source form, the Fortran standard allows no more than 19 continuation lines within a statement.

Statement labels can appear only in positions 1 through 5. A label can appear only on the initial line of a continued statement. Thus, positions 1 through 5 of continuation lines must contain blanks.

The compiler allows you to continue an `END` statement, as follows:

```
 E
&N
&D
```

ANSI/ISO: The Fortran standard states that an `END` statement must not be continued.

The characters `END` can appear as the only characters on the initial line of a statement, as follows:

```
 END
&FILE(10)
```

ANSI/ISO: The Fortran standard states that `END` cannot be an initial line of a statement other than an `END` statement.

Any character from the supported compiler character set (including graphic and control characters) can be used in character literal constants and character edit descriptors. Although the Fortran standard permits a processor to limit the use of control characters (such as the newline) to such character contexts, the compiler imposes no such limitation.

3.3.2.1 Tab Character

You can substitute the tab character for spaces at the beginning of a line, but it is not actually converted to spaces. The compiler uses the tab as follows:

- If a tab is the first character on a line, the next character determines how the line is interpreted. A nonzero digit indicates a continuation line. Otherwise, this line is the initial line of a statement.
- A statement label, if present, must precede the first tab.
- The compiler treats the tab character in a statement the same way it treats a blank character.

A tab character is not converted to spaces, so the exact visual placement of tabbed statements depends on the utility you use to edit or display text.

Note: A tab counts as 1 character even though it may expand to more than one space in the listing or editor. Thus, statements that include tabs may appear to have data beyond column 72 (or 80) in the editor or listing.

3.3.2.2 `D` or `d` character (EXTENSION)

The compiler allows a `D` or `d` character in column one. This notation directs the compiler to replace the `D` or `d` with a blank and treat the rest of the line as a source statement. This can be used, for example, for debugging purposes if the rest of the line contains a `PRINT` statement.

This functionality is controlled through the `-d_lines` option on the compiler. For more information on these options, see your `f90`(1) man page.

Note: The Fortran standard does not describe using `D` or `d` characters in column 1.

3.3.2.3 Sample Program, Fixed Source Form

The following is a sample program in fixed source form. Note that the numbers and the line at the top are not part of the program; the vertical bars to the left of the

program are also not part of the program. These graphics are included to show the columns this program uses in fixed source form.

```
 123456789.....
-------------------------------------------
|      PROGRAM LEFT_RIGHT
|      REAL  X(5), Y(5)
|C     Print arrays X and Y
|      PRINT 100, X, Y
|  100 FORMAT (F10.1, F10.2, F10.3, F10.4,
|     1        F10.5)
|          . . .
|      END
```

3.4 Portable Source Form

For portability in many cases, such as an included file, it is desirable to use a form of the source code that is valid and equivalent for either free source form or fixed source form. Such a source form can be written by obeying the following rules and restrictions:

- Limit labels to positions 1 through 5, and statements to positions 7 through 72. These are the limits required in fixed source form.
- Treat blanks as significant. Because blanks are ignored in fixed source form, using the rules of free source form will not impact the requirements of fixed source form.
- Use the exclamation mark (`!`) for a comment, but do not place it in position 6, which indicates continuation in fixed source form. Do not use the `C` or `*` forms for a comment.
- To continue statements, use the ampersand in both position 73 of the line to be continued and in position 6 of the continuation. Positions 74 to 80 must remain blank or contain only a comment. Positions 1 through 5 must be blank. The first ampersand continues the line after position 72 in free source form and is ignored in fixed source form. The second ampersand indicates continuation in fixed source form and in free source form indicates that the text for the continuation of the previous line begins after the ampersand.
- The compiler allows you to switch between fixed and free source forms within a file or include file by using the `FIXED` and `FREE` compiler directives. All compiler

directives should begin in column 1. For more information on compiler directives, see the *MIPSpro Fortran 90 Commands and Directives Reference Manual*.

3.4.1 Sample Program, Use with Either Source Form

The following is a sample program that is acceptable in either fixed or free source form. Note that the numbers and the line at the top are not part of the program; the vertical bars to the left of the program are also not part of the program. These graphics are included to show the columns this program uses in fixed source form.

```
 123456789.....                                          73
 ----------------------------------------------.........
|      PROGRAM LEFT_RIGHT
|      REAL  X(5), Y(5)
|!     Print arrays X and Y
|      PRINT 100, X, Y
|  100 FORMAT (F10.1, F10.2, F10.3, F10.4,           &
|     &        F10.5)
|          . . .
|      END
```

3.5 The INCLUDE Line

Source text can be imported from another file and included within a program file during processing. An `INCLUDE` line consists of the keyword `INCLUDE` followed by a character literal constant. The following is an example of an `INCLUDE` line:

```
INCLUDE  'MY_COMMON_BLOCKS'
```

The specified text is substituted for the `INCLUDE` line during compilation and is treated as if it were part of the original program source text. The compiler allows you to specify search path names on the `f90`(1) command line for locating files to be included. For more information on `INCLUDE` lines, see the *MIPSpro Fortran 90 Commands and Directives Reference Manual*.

The `INCLUDE` line provides a convenient way to include source text that is the same in several program units. For example, the specification of interface blocks or objects in common blocks may constitute a file that is referenced in the `INCLUDE` line.

The format for an `INCLUDE` line is as follows:

```
INCLUDE character_literal_constant
```

The *character_literal_constant* used cannot have a kind parameter that is a named constant.

The INCLUDE line is a directive (but not a compiler directive) to the compiler; it is not a Fortran statement.

The INCLUDE line is placed where the included text is to appear in the program.

The INCLUDE line must appear on one line with no other text except possibly a trailing comment. There can be no statement label.

The INCLUDE lines can be nested. That is, a second INCLUDE line may appear within the text to be included. The Fortran anstandard does not specify the permitted level of nesting, and the compiler imposes no limit. The text inclusion cannot be recursive at any level. For example, included text A cannot include text B, which includes text A.

A file intended to be referenced in an INCLUDE line cannot begin or end with an incomplete statement.

An example of a program unit with an INCLUDE line follows:

```
PROGRAM MATH
REAL, DIMENSION(10,5,79) :: X, ZT!  Some arithmetic
INCLUDE 'FOURIER'!  More arithmetic
   . . .
END
```

The source text in the file FOURIER in effect replaces the INCLUDE line.

3.6 Low-level Syntax

The basic lexical elements of the language consist of the classes *character*, *name*, *constant*, *intrinsic_operator*, *defined_operator*, and *label*. These are defined as follows:

defined_operator	**is**	*defined_unary_op*
	or	*defined_binary_op*
	or	*extended_intrinsic_op*
defined_unary_op	**is**	. *letter* [*letter*] . . . .
defined_binary_op	**is**	. *letter* [*letter*] . . . .
extended_intrinsic_op	**is**	*intrinsic_operator*
label	**is**	*digit* [*digit* [*digit* [*digit* [*digit*]]]]

Data Types

Fortran was designed to give scientists and engineers an easy way to solve problems by using computers. Statements could be presented that looked like formulas or English sentences. For example, the following statement might be performing typical numeric calculations:

```
X = B + A * C
```

As another example, the following statement could specify that a certain action is to be taken based on a logical decision:

```
IF (LIMIT_RESULTS .AND. X .GT. XMAX) X = XMAX
```

And the following statement could be used to communicate the results of a calculation to a scientist or engineer in a meaningful way:

```
PRINT *, "CONVERGENCE REACHED"
```

Each of these statements performs a task that uses a different type of data:

Task	Data Type
Calculating typical numeric results	Numeric data
Making decisions	Logical data
Explaining	Character data

The preceding list shows the commonly needed data types, and the Fortran standard provides for them.

The compiler supports additional data types. This preserves compatibility with other vendor's systems. These additional types are as follows:

- Cray pointer
- Boolean (or typeless)

ANSI/ISO: The Fortran standard does not specify Cray pointer, Cray character pointer, or Boolean data types.

Anything provided by the language is *intrinsic* to the language. Types that are not intrinsic to the language can be specified by a programmer. The programmer-specified types are built of (or derived from) the intrinsic types and thus are called *derived types*. The Fortran data types are categorized in Figure 4-1.

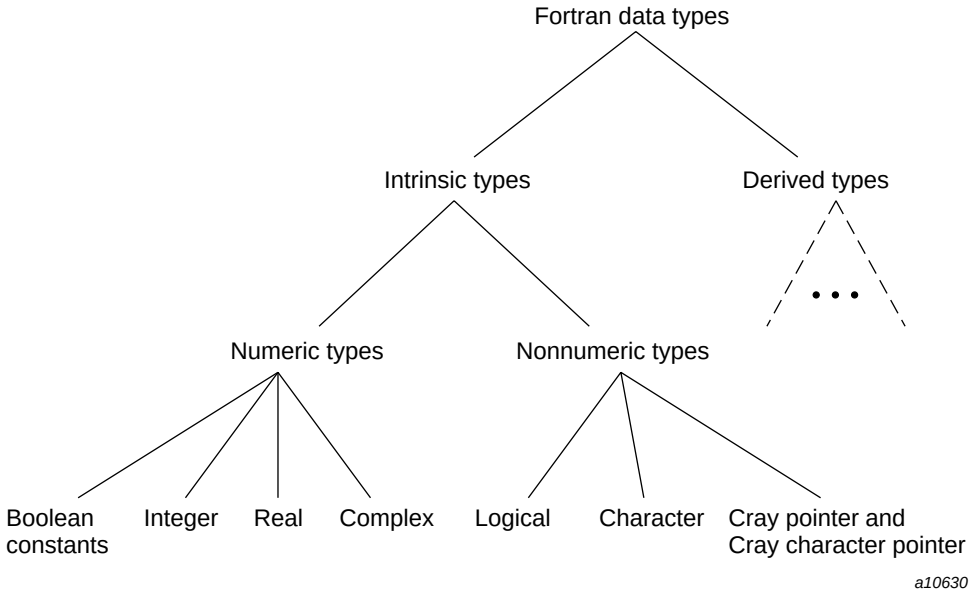

Figure 4-1 Fortran data types

As the following list shows, the type of the data determines the operations that can be performed on it:

Data Type	Operations
Real, complex, integer, Boolean	Addition, subtraction, multiplication, division, exponentiation, negation, comparison, masking expressions
Logical	Negation, conjunction, disjunction, and equivalence
Character	Concatenation, comparison
User defined	User defined

Cray pointer, Cray character pointer	Addition, subtraction, and `LOC( )`function

The intrinsic types have the appropriate built-in (intrinsic) operations. You must define the operations performed on user-defined data types.

This chapter explains the Fortran data types. It describes each of the intrinsic types, and it explains derived types and the facilities provided by the language that allow you to define types and declare and manipulate objects of these types in ways that are analogous to the ways in which objects of the intrinsic types can be manipulated.

4.1 Building the Data Environment for a Problem Solution

When envisioning a computer solution to a problem you focus on the operations that must be performed and the order in which they must be performed. It is a good idea, however, to consider the variables you will need before you determine all the computational steps that are required. The variables that are chosen, together with their types and attributes, sometimes determine the course of computation, particularly when variables of user-defined type are involved.

4.1.1 Choosing the Type and Other Attributes of a Variable

There are several decisions to make about a variable in a program. If the variable is of an intrinsic type, the intended use of the variable will readily determine its type, making this an easy decision. While type is the most important attribute of a variable, there are other attributes. Certainly it is necessary to decide very early whether the variable is to be a single data object (a scalar) or an array. Fortran provides many new facilities for manipulating arrays as objects, making it possible to specify computations as straightforward array operations.

Because Fortran provides allocatable arrays and pointers, it is not necessary to decide at the outset how big an array must be. In fact, determining sizes can be postponed until the finished program is executed, when sizes can be read in as input or calculated. Setting aside space for an array can be deferred until the appropriate size needed for a particular calculation is known.

Another decision that can be made about a variable is its accessibility. Control of accessibility is a feature available in modules. If the variable is needed only within the module, then it can be kept private or hidden from other program units. This prevents it from being corrupted inadvertently. This feature can be used to make Fortran programs safer and more reliable.

In addition to type, dimensionality, dynamic determination, and accessibility, there are other attributes that can be applied to data objects. The attributes that are permitted depend on where and how the object is to be used; for example, there are a number of attributes that can be applied only to subprogram arguments. Chapter 5, page 115, describes all of the attributes of data objects.

4.1.2 Choosing the Kind of a Variable of Intrinsic Type

After the type of a variable is decided, you may need to consider which kind of the type to use. Each of the intrinsic types can be specified with a *kind parameter* that selects a processor-dependent representation of objects of that type and kind. If no kind parameter is specified, the default kind is assumed.

Note: Depending on your hardware platform, the compiler may support more than one kind for each data type.

Fortran requires a processor to support at least two kinds for the real and complex types and at least one kind for the other three intrinsic types. The compiler supports several kinds of real, complex, logical, and integer data types.

The Fortran data types are as follows:

- Real. Programs with `REAL` and `DOUBLE PRECISION` declarations are not numerically portable across machine architectures with different word sizes. The compiler chooses a representation for the real type that is efficient on the target machine. For example, a representation that fits into 32 bits is used on machines with 32-bit words, and a representation that fits into 64 bits is used on machines with 64-bit words.

  A kind parameter gives you access to and control over the use of different machine representations of real values in order to make a program more portable. For example, a kind parameter in a `REAL` declaration can specify a required minimum precision, as follows:

  ```
  REAL(KIND=SELECTED_REAL_KIND(10,50)) :: REAL_VALUE
  ```

  When a program is run on a 32-bit machine, it uses two words to contain variable `REAL_VALUE`. When the same program (without any changes) is run on a 64-bit machine, one word is used to contain variable `REAL_VALUE`.

Fortran treats double-precision real as a separate kind of real. There are two ways to declare real variables: one is with a `REAL` statement specifying a nondefault kind and the other is with a `DOUBLE PRECISION` statement.

- Complex. Fortran uses a `COMPLEX` attribute with a nondefault kind parameter to specify double-precision complex.
- Character. The Fortran standard's kind type parameter values allow a single character to occupy more than one byte. The compiler supports only the ASCII character set, though, so they have no nondefault character kind.
- Integer. Alternative representations of integer data provide an integer kind with a very large range. The compiler supports several integer kinds.

4.1.3 Choosing to Define a Type for a Variable

Sometimes it is easier to think about an essential element of a problem as several pieces of related data, not necessarily all of the same type. Arrays can be used to collect homogeneous data (all of the same type) into a single variable. A *structure* is a collection of nonhomogeneous data in a single variable. To declare a structure, it is first necessary to define a type that has components of the desired types. The structure is then declared as an object of this user-defined (or derived) type.

An example of objects declared to be of user-defined type was given in Section 2.3.1, page 17. It is repeated here. First a type, named `PATIENT`, is defined. Next, two structures, `JOHN_JONES` and `SALLY_SMITH`, are declared:

```
TYPE PATIENT
   INTEGER         PULSE_RATE
   REAL            TEMPERATURE
   CHARACTER *300  PROGNOSIS
END TYPE PATIENT

TYPE(PATIENT)     JOHN_JONES, SALLY_SMITH
```

Type `PATIENT` has three components, each of a different intrinsic type (integer, real, and character). In practice, a type of this nature probably would have even more components, such as the patient's name and address, insurance company, room number in the hospital, and so on. For purposes of illustration, three components are sufficient. `JOHN_JONES` and `SALLY_SMITH` are structures (or variables) of type `PATIENT`. A type definition indicates names, types, and attributes for its components; it does not declare any variables that have these components. Just as with the

intrinsic types, a type declaration is needed to declare variables of this type. Because there is a type definition, though, any number of structures can be created that have the components specified in the type definition for `PATIENT`; subprogram arguments and function results can be of type `PATIENT`; there can be arrays of type `PATIENT`; and operations can be defined that manipulate objects of type `PATIENT`. Thus, the derived-type definition can be used merely as a way to specify a pattern for a particular collection of related but nonhomogeneous data; but, because the pattern is specified by a type definition, a number of other capabilities are available.

4.2 What Is Meant by *type* in Fortran

Knowing exactly what is meant by *type* in Fortran becomes more important now that you can define types in addition to the intrinsic types. A data type provides a means to categorize data and determine which operations can be applied to the data to get desired results. The following exist for each data type:

- A name
- A set of values
- A set of operations
- A form for constants of the intrinsic types and constructors for derived types

4.2.1 Data Type Names

Each of the intrinsic types has a name supplied by the standard. The names of derived types must be supplied in type definitions. The name of the type is used to declare entities of the type unless the programmer chooses to let the processor determine the type of an entity implicitly by the first character of its name. Chapter 5, page 115, describes declarations and implicit typing.

4.2.2 Data Type Values

Each type has a set of valid values. The logical type has only two values: true and false. The integer type has a set of integral numeric values that can be positive, negative, or zero. For complex or derived types, the set of valid values is the set of all combinations of the values of the individual components.

The kind of an intrinsic type determines the set of valid values for that type and kind. For example, there is more than one integer data type: the default type and shorter integer types. The shorter integer types have values that are a subset of the default integer values. The kind of a type is referred to as a *kind parameter* or *kind type parameter* of the type. The character data type has a length parameter as well as a kind parameter. The length parameter specifies the number of characters in an object, and this determines the valid values for a particular character object. Derived types do not have parameters, even though their components may.

4.2.3 Data Type Operations

For each of the intrinsic data types, a set of operations with corresponding operators is provided by the language. These are described in Chapter 7, page 211.

You can specify new operators and define operations for the new operators. The form of a new operator is an alphabetic name of your choice delimited by periods. These new operators are analogous to intrinsic operators such as `.GT.`, `.AND.`, and `.NEQV.`. For example, you might specify and define the operations `.PLUS.`, `.REMAINDER.`, and `.REVERSE.`. In defining the operation, the types of allowable operands must be specified. Such new operations can apply to objects of intrinsic type and in these cases extend the set of operations for the type. You would more frequently be defining operations for objects of derived type.

You cannot redefine an intrinsic operation, but you can define meanings for intrinsic operator symbols when at least one operand is not of an intrinsic type or for intrinsic operands for which the intrinsic operation does not apply. For example, consider the expression `A + B`. If both `A` and `B` are of numeric type, the operation is defined intrinsically and cannot be redefined. However, if either `A` or `B` is of derived type or nonnumeric type, the plus operation between `A` and `B` is not defined intrinsically, and you can provide a meaning for the operation. New operations are defined by functions with the `OPERATOR` interface. These are described in the *MIPSpro Fortran Language Reference Manual, Volume 2*.

Assignment is defined intrinsically for each intrinsic and derived type. Structure assignment is component-by-component intrinsic or pointer assignment, though this can be replaced by a defined assignment. No other intrinsically defined assignment, including array assignment, can be redefined. Beyond this, any assignment between objects of different type may be defined with the `ASSIGNMENT` interface as described in the *MIPSpro Fortran Language Reference Manual, Volume 2*.

4.2.4 Forms for Constants and Constructors

The language specifies the syntactic forms for literal constants of each of the intrinsic types. Syntactic mechanisms (called derived-type constructors) specify derived-type values. As shown in Figure 4-2, page 74, the form indicates both the type and a particular member of the set of valid values for the type.

Syntax	Type	Value
1	integer	1
103.1 or 1.031E2	real	103.1
(1.0, 1.0)	complex	1 + *i*
.TRUE.	logical	true
"Hello"	character	Hello
PATIENT (70, 99.7, "Recovering")	patient	70 99.7 Recovering

Figure 4-2 Forms of constants and constructors

If a constant is not of default kind, some indication of its kind must be included in its syntactic form. This form is the default literal constant separated from the kind value by an underscore. Kind specifications follow integer, real, and logical values. Kinds are known to the compiler as integer values, but if a program is to be portable, the actual numbers should not be used because the kind values depend on the processor. Instead, a kind value should be assigned to a named constant, and you should use the name.

In the following examples, `DOUBLE` and `HIGH` are named constants for kind values:

Real `1.3141592653589_DOUBLE`

Complex `(1.75963_HIGH, -2.0)`

The kind of a complex constant is determined by the kind of its parts. Section 4.3.3.3, page 84, describes the form for complex literal constants.

4.3 Intrinsic Data Types

The default real kind, default integer kind, and default logical kind are all stored in one storage unit. Default complex (which is really two default reals) and double-precision real data is stored in two storage units.

Note: This chapter describes each of the intrinsic types. The descriptions include a simple statement form to show how objects of these types can be declared. These simple forms are not complete. If they are used to construct statements, the statements will be correct, but other variations are permitted. Section 5.1, page 117, contains the complete formats. The kind parameter that appears in the formats is limited to a scalar integer initialization expression, which is described in Section 7.2.9.2, page 252.

4.3.1 Integer Type

The name of the integer type is `INTEGER`. The following format shows how integer objects can be declared:

```
INTEGER [ ([ KIND = ] kind_param) ] [ [ , attribute_list ] :: ] entity_list
```

Examples:

```
INTEGER :: X
INTEGER :: COUNT, K, TEMPORARY_COUNT
INTEGER(SHORT) :: PARTS
INTEGER, DIMENSION(0:9) :: SELECTORS, IX
```

4.3.1.1 Values

The integer data type has values that represent a subset of the mathematical integers. The intrinsic inquiry function `RANGE` provides the decimal exponent range for integers of the kind of its argument. Only one kind of integer is required by the standard, but the compiler supports several. Values that overflow in storage may be truncated.

If *kind_param* is specified, it must have one of the following values: 1, 2, 4, or 8. The default *kind_param* is specific to your hardware platform. These values are shown in Table 4-1, page 76. Options to the `f90`(1) command allow you to change the size and storage aspects of integer values. See the*MIPSpro Fortran 90 Commands and Directives Reference Manual*, for information on changing default kind parameter values.

Table 4-1 Integer kind values

kind_param	Value range	Size / Storage
1	$-2^7 \le n < 2^7$	8 bits / 8 bits
2	$-2^{15} \le n < 2^{15}$	16 bits / 16 bits
4 (default)	$-2^{31} \le n < 2^{31}$	32 bits / 32 bits
8	$-2^{63} \le n < 2^{63}$	64 bits / 64 bits

In Table 4-1, the Size information refers to the size according to the integer model defined on the `MODELS`(3i) man page.

Table 4-2, page 76, shows power-of-10 values that approximate the power-of-2 values shown in Table 4-1.

Table 4-2 Exponent equivalents

2^n	10^k
2^7	10^2
2^{15}	10^4
2^{31}	10^9
2^{45}	10^{13}
2^{52}	10^{15}
2^{63}	10^{18}

The `RANGE`(3i) intrinsic function returns the decimal exponent range of a given number. For more information on this intrinsic function, see the `RANGE`(3i) man page.

The `KIND`(3i) intrinsic function can be used to determine the kind parameter of its integer argument.

The `SELECTED_INT_KIND`(3i) intrinsic function returns the integer kind parameter required to represent as many decimal digits as are specified by the function argument. If there is no such integer type available on your system, –1 is returned.

The following statement declares `I` and `J` to be integer objects with a representation method that permits at least five decimal digits; that is, it includes all integers between -10^5 and 10^5:

```
INTEGER (SELECTED_INT_KIND (5)) I, J
```

4.3.1.2 Operators

There are both binary and unary intrinsic operators for the integer type. Binary operators have two operands and unary operators have only one. The binary arithmetic operations for the integer type are: `+`, `-`, `*`, `/`, and `**`. The unary arithmetic operations are `+` and `-`. The relational operations (all binary) are: `.LT.`, `<`, `.LE.`, `<=`, `.EQ.`, `==`, `.NE.`, `/=`, `.GE.`, `>=`, `.GT.`, and `>`. The result of an intrinsic arithmetic operation on integer operands is an integer entity. The result of an intrinsic relational operation is a logical entity of default logical kind.

4.3.1.3 Format for Constant Values

An integer constant is a string of decimal digits, optionally preceded by a sign, and optionally followed by an underscore and a kind parameter. An integer constant that must not have a kind type parameter is defined as follows:

signed_digit_string	**is**	[*sign*] *digit_string*
digit_string	**is**	*digit* [*digit*]...

The format of a signed integer literal constant (which may have a kind type parameter) is defined as follows:

signed_int_literal_constant	**is**	[*sign*] *int_literal_constant*
int_literal_constant	**is**	*digit_string* [_ *kind_param*]
kind_param	**is**	*digit_string*
	or	*scalar_int_constant_name*
sign	**is**	+
	or	-

The signed integer literal constant often takes the following format:

[*sign*] *digit_string* [_ *kind_param*]

In the following examples, assume that LONG and SHORT are named constants with values that are valid integer kind parameters:

```
42
999999999999999999999_LONG
+64
10000000
-258_SHORT
```

Integer constants are interpreted as decimal values. However, in a DATA statement, it is possible to initialize an object with a value that is presented as if it had a nondecimal base. The allowed forms are unsigned binary, octal, and hexadecimal constants, and are defined as follows:

boz_literal_constant	**is**	*binary_constant*
	or	*octal_constant*
	or	*hex_constant*

The compiler supports binary, octal, and hexadecimal constants in other contexts; for more information on these forms see Section 4.3.6, page 89.

A *binary_constant* is defined as follows:

binary_constant	**is**	B ' *digit* [*digit*] ... '
	or	B " *digit* [*digit*] ... "

You must specify 0 or 1 for *digit*.

An *octal_constant* is defined as follows:

octal_constant	**is**	`O ' digit [ digit ] ... '`
	or	`O " digit [ digit ] ... "`

You must specify a value from 0 through 7 for *digit*.

A *hex_constant* is defined as follows:

hex_constant	**is**	`Z' hex_digit [ hex_digit ] ...'`
	or	`Z " hex_digit [ hex_digit ] ... "`
hex_digit	**is**	*digit*
	or	`A`
	or	`B`
	or	`C`
	or	`D`
	or	`E`
	or	`F`

You must specify a value from 0 through 9 or one of the letters `A` through `F` (representing the decimal values 10 through 15) for *digit*. The compiler supports lowercase letters, so the hexadecimal digits `A` through `F` can be represented by their lowercase equivalents, `a` through `f`.

In these constants, the binary, octal, and hexadecimal digits are interpreted according to their respective number systems. For example, all of the following have a value equal to the decimal value 10:

```
B"1010"
O'12'
Z"A"
```

4.3.2 Real Type

The name of the real data type is `REAL`. The name `DOUBLE PRECISION` is used for another kind of the real type. You can use one of the following formats to declare objects of real type:

```
REAL [ ([ KIND = ] kind_param) ] [ [ , attribute_list ] :: ] entity_list

DOUBLE PRECISION [ [ , attribute_list] :: ] entity_list
```

Examples:

```
REAL X, Y
REAL(KIND = HIGH), SAVE :: XY(10, 10)
REAL, POINTER :: A, B, C
DOUBLE PRECISION DD, DXY, D
```

4.3.2.1 Values

The values of the real data type approximate the mathematical real numbers. The set of values varies from processor to processor. The Fortran standard requires a processor to support at least two approximation methods for the real type. The compiler provides three, so there are three kind values for the real type.

If *kind_param* is specified, it must have one of the following values: 4, 8, or 16. The default *kind_param* and the values associated with the kind values are specific to your hardware platform. These values are shown in Table 4-3, page 80. Options to the `f90`(1) command allow you to change the size and storage aspects of real values. See the *MIPSpro Fortran 90 Commands and Directives Reference Manual* for information on changing default kind parameter values.

Table 4-3 Real and complex kind values

kind_param	Value range	Size and Storage	Operating system
4 (default)	$2^{-125} \leq n < 2^{128}$	32 bits	IRIX
8	$2^{-1021} \leq n < 2^{1024}$	64 bits	IRIX
16	$2^{-967} \leq n < 2^{1023}$	128 bits	IRIX

Table 4-4, page 81, shows power-of-10 values that approximate some of the power-of-2 values shown in Table 4-3.

Table 4-4 Exponent equivalents

2^n	10^k
2^{128}	10^{38}
2^{1024}	10^{308}
2^{8189}	10^{2466}
2^{16384}	10^{4932}

The KIND(3i) intrinsic function can be used to determine the kind parameter of its real argument. The intrinsic functions PRECISION(3i) and RANGE(3i) return the decimal precision and exponent range of the approximation method used for the kind of the argument. The intrinsic function SELECTED_REAL_KIND(3i) returns the kind value required to represent as many digits of precision as specified by the first argument and the decimal range specified by the optional second argument.

The following statement declares X to have at least five decimal digits of precision and no specified minimum range:

```
REAL(SELECTED_REAL_KIND(5)) X
```

The following statement declares Y to have at least eight decimal digits of precision and a range that includes values between 10^{-70} and 10^{70} in magnitude:

```
REAL(SELECTED_REAL_KIND(8, 70)) Y
```

4.3.2.2 Operators

The intrinsic binary arithmetic operators for the real type are: +, -, *, /, and **. The intrinsic unary arithmetic operators are: + and -. The relational operators are: .LT., <, .LE., <=, .EQ., ==, .NE., /=, .GE., >=, .GT., >, .LG., and <>. The result of an intrinsic arithmetic operation on real operands is a real entity. If one of the operands of an arithmetic operation is an integer entity, the result is still a real entity. The result of an intrinsic relational operation is a logical entity of default logical kind.

ANSI/ISO: The Fortran standard does not describe the .LG. or <> operators.

4.3.2.3 Forms for Constants

A real constant is distinguished from an integer constant by containing either a decimal point, an exponent, or both. The format for a *signed_real_literal_constant* is defined as follows:

signed_real_literal_constant	**is**	[*sign*] *real_literal_constant*
real_literal_constant	**is**	*significand exponent_letter exponent* [*_kind_param*]
	or	*digit_string exponent_letter exponent* [*_kind_param*]
significand	**is**	*digit_string* . [*digit_string*]
	or	. *digit_string*
exponent_letter	**is**	`E`
	or	`D`
	or	`Q`
exponent	**is**	*signed_digit_string*

A signed real literal constant can take one of the following forms:

[*sign*] *digit_string exponent_letter exponent* [*_kind_param*]

[*sign*] *whole_part* . [*fraction_part*] [*exponent_letter exponent*] [*_kind_param*]

[*sign*] . *fraction_part* [*exponent_letter exponent*] [*_kind_param*]

For *whole_part*, specify a *digit_string*. For *fraction_part*, specify a *digit_string*. For *exponent*, specify a *signed_digit_string*.

If both a *kind_param* and an *exponent_letter* are present, the *exponent_letter* must be `E` or `e`. If a *kind_param* is present, the real constant is of that kind; if a `D` or `d` exponent letter is present, the constant is of type double-precision real; if a `Q` exponent is present, the constant is of type quad-precision real; otherwise, the constant is of type default real.

A real constant can have more decimal digits than will be used to approximate the real number.

Examples of signed real literal constants are as follows:

```
-14.78
+1.6E3
2.1
-16.E4_HIGH
0.45_LOW
.123
3E4
2.718281828459045D0
```

In the preceding example, the parameters `HIGH` and `LOW` must have been defined, and their values must be kind parameters for the real data type permitted by the compiler.

If a real literal constant has a kind parameter, it takes precedence over an exponent letter. Consider the following specification:

```
1.6E4_HIGH
```

The example's code fragment will be represented by the method specified for `HIGH`, even though `1.6E4` would be represented by a different method.

4.3.3 Complex Type

The name of the complex type is `COMPLEX`. A format for declaring objects of this type is as follows:

`COMPLEX [([ KIND = ]` *kind_param*`) ] [[ , `*attribute_list*` ] :: ] `*entity_list*

Examples:

```
COMPLEX CC, DD
COMPLEX(KIND = single), POINTER :: CTEMP(:)
```

4.3.3.1 Values

The complex data type has values that approximate the mathematical complex numbers. A complex value is a pair of real values; the first is called the *real part* and the second is called the *imaginary part*. Each approximation method used to represent data entities of type real is available for entities of type complex with the same kind parameter values. Therefore, there are three approximation methods for complex.

When a complex entity is declared with a kind specification, this kind is used for both parts of the complex entity. There is no special double-precision complex declaration, as such. If no kind parameter is specified, the entity is of type default complex which corresponds to default real. The SELECTED_REAL_KIND(3i) intrinsic function may be used in a declaration of a complex object.

If *kind_param* is specified, it must have one of the following values: 4, 8, or 16. The default kind is specific to your hardware platform. The values, the ranges (for both the real and imaginary portions of the number), and the defaults supported are the same as those for type real, and these values are shown in Table 4-3, page 80. See the *MIPSpro Fortran 90 Commands and Directives Reference Manual* for information on changing default kind parameter values.

For information on power-of–10 and power-of–2 equivalent values, see Table 4-4, page 81.

In the following statement, CX must be represented by an approximation method with at least 8 decimal digits of precision and at least a decimal exponent range between 10^{-70} and 10^{70} in magnitude for the real and imaginary parts:

```
COMPLEX(SELECTED_REAL_KIND(8, 70)) CX
```

4.3.3.2 Operators

The intrinsic binary arithmetic operators for the complex type are: +, -, *, /, and **. The intrinsic unary arithmetic operators are: + and -. The intrinsic relational operators are: .EQ., ==, .NE., and /=. The arithmetic operators specify complex arithmetic; the relationals compare operands to produce default logical results. The result of an intrinsic arithmetic operation on complex operands is a complex entity. If one of the operands is an integer or real entity, the result is still a complex entity.

4.3.3.3 Form for Constants

A complex literal constant is written as two literal constants that are real or integer, separated by a comma, and enclosed in parentheses, as follows:

(*real_part* , *imag_part*)

The compiler allows the real and imaginary portions of a complex literal constant to be named constants.

ANSI/ISO: The Fortran standard does not specify the use of named constants as the real or imaginary components of a complex literal constant.

The format for a *complex_literal_constant* is defined as follows:

complex_literal_constant	**is**	(*real_part* , *imag_part*)
real_part	**is**	*signed_int_literal_constant*
	or	*signed_real_literal_constant*
imag_part	**is**	*signed_int_literal_constant*
	or	*signed_real_literal_constant*

Examples:

```
(3.0, -3.0)
(6, -7.6E9)
(3.0_HIGH, 1.6E9_LOW)
```

A real kind parameter can be specified for either one of the two real values. If a different real kind parameter is given for each of the two real values, the complex value will have the kind parameter that specifies the greater precision, unless the kind parameters specify the same precision. If both parts are integer, each part is converted to default real. If one part is of integer type and the other is of real type, the integer value is converted to the kind and type of the real value.

4.3.4 Logical Type

The name of the logical type is `LOGICAL`. A format for declaring objects to be of this type is as follows:

```
LOGICAL [([ KIND = ] kind_param) ] [[ , attribute_list ] :: ] entity_list
```

Examples:

```
LOGICAL IR, XT
LOGICAL(KIND = SMALL), SAVE :: XMASK (3000)
```

4.3.4.1 Values

The logical data type has two values that represent true and false. The Fortran standard requires processors to provide one logical kind, but the compiler provides other kinds. Each logical item occupies one word. (An object of default logical type must occupy the same unit of storage as an object of default real type.) The `KIND`(3i) intrinsic function can be used to determine the kind number of its argument. There is no intrinsic function analogous to the functions `SELECTED_INT_KIND`(3i) and `SELECTED_REAL_KIND`(3i).

If specifying a *kind_param*, it must have one of the following values: 1, 2, 4, or 8. The default *kind_param* and the values associated with the kind values are specific to your hardware platform. These values are shown in Table 4-5. See the *MIPSpro Fortran 90 Commands and Directives Reference Manual* for information on changing default kind parameter values.

Table 4-5 Logical kind values

kind_param	Size / Storage	Operating system
1	8 bits / 8 bits	IRIX
2	16 bits / 16 bits	IRIX
4 (default)	32 bits / 32 bits	IRIX
8	64 bits / 64 bits	IRIX

4.3.4.2 Operators

The intrinsic binary operators for the logical type are as follows: conjunction (`.AND.`), inclusive disjunction (`.OR.`), logical equivalence (`.EQV.`), and logical nonequivalence (`.NEQV.`). The intrinsic unary operation is negation (`.NOT.`). The exclusive disjunction operator for the compiler is `.XOR.`

ANSI/ISO: The Fortran standard does not specify the `.XOR.` exclusive disjunction operator.

4.3.4.3 Form for Constants

There are only two logical literal constants. They can be followed by an underscore and a kind parameter. The format for a *logical_literal_constant* is defined as follows:

logical_literal_constant	**is**	`.TRUE.` [*_kind_param*]
	or	`.FALSE.` [*_ kind_param*]

If a *kind_param* is not specified, the type of the constant is default logical. Examples are as follows:

```
FALSE.
TRUE._WORD
```

If `.T.` and `.F.` are not defined operators in a compilation unit, the compiler recognizes `.T.` as an abbreviation for `.TRUE.` and `.F.` for `.FALSE.`

ANSI/ISO: The Fortran standard does not specify the use of `.T.` or `.F.`.

4.3.5 Character Type

The name of the character type is `CHARACTER`. Declarations for objects of this type may take several different forms. One of these forms is as follows:

```
CHARACTER [ ([ LEN = ] length_parameter [, [ KIND = ] kind_param ]) ]
   [ [ , attribute_list ] :: ] entity_list
```

The *length_parameter* can be an asterisk (`*`) or a specification expression, which is described in Section 7.2.9.3, page 254. The various forms of the `CHARACTER` statement are described in Section 5.1.6, page 124, but the following examples use the form specified previously:

```
CHARACTER(80) LINE
CHARACTER(*) GREETING
CHARACTER(LEN = 30, KIND = ASCII), DIMENSION(10) :: C1
```

4.3.5.1 Values

The character data type has a set of values composed of character strings. A character string is a sequence of characters, numbered from left to right 1, 2, ..., *n*, where *n* is the length of (number of characters in) the string. Both length and kind are type parameters for the character type. If no length parameter is specified, the length is 1. A character string can have length 0. The maximum length permitted for character strings is 2, 097, 151. Although the Fortran standard permits a processor to provide more than one character kind, the compiler supports only one, ASCII. Thus, the compiler does not support any nondefault character kinds. If *kind_param* is specified, it must have the value 1.

The Fortran standard specifies only a partial collating sequence because it is concerned only that operations that compare character objects containing only characters from the Fortran character set will be portable across different processors. Because the compiler supports the ASCII character set, they follow the Fortran standard's collating requirements. The intrinsic functions `ACHAR`(3i) and `IACHAR`(3i) convert between numeric values and ASCII characters. The intrinsic functions `LGT`(3i), `LGE`(3i), `LLE`(3i), and `LLT`(3i) provide comparisons between strings based on the ASCII collating sequence.

4.3.5.2 Operators

The binary operation concatenation (`//`) is the only intrinsic operation on character entities and has a character entity as a result. A number of intrinsic functions are provided that perform character operations. These are described in the *MIPSpro Fortran Language Reference Manual, Volume 2*. The intrinsic relational operators on objects of type character are `.LT.`, `<`, `.LE.`, `<=`, `.EQ.`, `==`, `.NE.`, `/=`, `.GE.`, `>=`, `.GT.`, and `>`. The relational operations can be used to compare character entities, but because of possible processor-dependent collating sequences, care must be taken if the results are intended to be portable.

4.3.5.3 Form for Constants

A character literal constant is written as a sequence of characters, enclosed either by apostrophes or quotation marks. The format for a *char_literal_constant* is as follows:

char_literal_constant	**is**	[*kind_param* _] ' [*ASCII_char*] ... '
	or	[*kind_param* _] " [*ASCII_char*] ... "

Note that, unlike the other intrinsic types, the kind parameter for the character literal constant precedes the constant.

If the string delimiter character (either an apostrophe or a quotation mark) is required as part of the constant, two consecutive such characters with no intervening blanks serve to represent a single such character in the string, for example:

```
"DON'T"
'DON''T'
```

These two examples have the value `DON'T`. A zero-length character constant can be written as `""` or `''`. The quotation marks or apostrophes are immediately adjacent to each other.

4.3.6 Boolean Type (EXTENSION)

A Boolean constant represents the literal constent of a single storage unit. There are no Boolean variables or arrays, and there is no Boolean type statement.

ANSI/ISO: The Fortran standard does not describe Boolean values.

Boolean type differs, depending on your platform, as follows:

- A bitwise logical expression has an integer result, with each of its bits representing the result of one or more logical operations on the corresponding bit of the expression's operands. When an operand of a binary arithmetic or relational operator is Boolean, the operation is performed as if the Boolean operand is of type integer. If both operands are of type Boolean, the operation is performed as if they were of type integer.

  No user-specified or intrinsic functions generate Boolean results.

Boolean and logical types differ in the following ways:

- Variables, arrays, and functions can be of logical type, and there is a `LOGICAL` type statement.
- A logical variable or constant represents only one value of true or false (rather than separate bit values), and a logical expression yields one true or false value.
- Logical entities are invalid in arithmetic, relational, or bitwise logical expressions, while Boolean entities are valid. (Note, however, that results of relational expressions are logical.)

A Boolean constant can be written as an octal, hexadecimal, or Hollerith value. There is no form for binary digits. Boolean constants can represent up to 256 bits of data. This size limit corresponds to the size of the largest numeric type, `COMPLEX(KIND = 16)`. The ultimate size and make-up of the constant is dependent on its context. The constant is truncated or padded to match the size of the type implied by its context. These forms use the notation described in the following sections.

4.3.6.1 Octal Form

The octal form contains 1 to 86 digits (0 through 7) in either of the following two forms:

- It can be a string of digits followed by the letter `B` or `b`, as in `177B`.
- It can be a quoted string of digits followed by the letter `O` or `o`, as in `"177"O`.
- It can be a quoted string of digits preceded by the letter `O` or `o`, as in `O"177"`.

The 86 digits in a Boolean value correspond to the internal representation of four 64-bit words or eight 32-bit words. If all 86 digits are specified, the leftmost octal digit must be only 0 or 1, representing the content of the leftmost bit position (bit 0 in the first word) of the value. Each successive octal digit specifies the contents of the next three bit positions. The last octal digit specifies the content of the rightmost three bit positions, which are bits 61, 62, and 63 of the last (fourth or eighth) word. Blanks are ignored in fixed source form. Blanks are significant in free source form for the *ddd*`B` syntax. That is, `1 777B` would not be treated as a single value in free source form, and it would most likely result in a syntax error message.

A Boolean value represented by fewer than 86 octal digits is right justified; that is, it represents the rightmost bits of 256 bits.

When context is taken into account, the value is truncated on the left if it is too large for the context type. It is padded with 0 on the left if it is too small for the context type.

Note: For a literal constant, the letter `B` indicates octal digits; in an I/O format specification, the `B` descriptor indicates binary digits.

Examples:

Boolean constant	**Internal representation (octal) for a 64-bit word**
`0B`	`0000000000000000000000`

77740B	0000000000000000077740

Statement example:

```
I = 1357B
```

4.3.6.2 Hexadecimal Form

The hexadecimal form consists of 1 to 64 hexadecimal digits consisting of `0` through `9`, `A` through `F`, or `a` through `f` in either of the following two forms:

- It can be specified as the letter `X` or `x` followed by a string of hexadecimal digits enclosed in apostrophes or quotations marks, as in `X"FFF"`.
- It can be specified with the `X` or `x` trailing the quoted string of digits, as in `"FFF"X`.

When a Boolean value contains 64 hexadecimal digits, the binary equivalents correspond to the content of each bit position in 4 64-bit words or eight 32-bit words.

A Boolean value represented by fewer than 64 hexadecimal digits is right justified; that is, it represents the rightmost bits of 256 bits.

An optional unary minus sign (`-`) is allowed within the quoted string.

When context is taken into account, the value is truncated on the left if it is too large for the context type. It is padded with 0 on the left if it is too small for the context type.

Examples:

Boolean constant	**Internal representation (octal) for a 64-bit word**
X'ABE'	0000000000000005276
X"-340"	1777777777777776300
X'1 2 3'	0000000000000000443
X'FFFFFFFFFFFFFFFF'	1777777777777777777

Statement examples:

```
J = X"28FF"
K = X'-5A'
```

4.3.6.3 Hollerith Form

A Hollerith constant is type Boolean. When a character constant is used in a bitwise logical expression, the expression is evaluated as if the value were Hollerith. A Hollerith constant can have a maximum of 32 characters.

When context is taken into account, the truncation or padding depends on the type of Hollerith syntax used.

For Hollerith using the letter `H`, the value is truncated on the right if it is too large for the context type. It is padded with blanks on the right if it is too small for the context type.

For Hollerith using the letter `L`, the value is truncated on the right if it is too large for the context type. It is padded with 0 on the right if it is too small for the context type.

For Hollerith using the letter `R`, an error occurs on IRIX systems if the Hollerith constant is greater than 4 characters.

4.3.7 Cray Pointer Type (EXTENSION)

A Cray pointer is a variable whose value is the address of another entity, which is called a pointee. The Cray pointer type statement declares both the pointer and its pointee.

ANSI/ISO: The Fortran standard does not describe Cray pointer values.

Cray pointers are declared as follows:

```
POINTER (pointer, pointee)
```

Fortran pointers are declared as follows:

```
POINTER :: [ object_name ]
```

The two kinds of pointers cannot be mixed.

You can use pointers to access user-managed storage by dynamically associating variables and arrays to particular locations in a block of storage. Cray pointers do not provide convenient manipulation of linked lists because, for optimization purposes, it

is assumed that no two pointers have the same value. Cray pointers also allow the accessing of absolute memory locations.

The range of a Cray pointer or Cray character pointer depends on the size of memory for the machine in use.

Restrictions on Cray pointers are as follows:

- A Cray pointer cannot be pointed to by another Cray or Fortran pointer; that is, a Cray pointer cannot also be a pointee or a target.
- A Cray pointer cannot appear in a `PARAMETER` statement or in a type declaration statement that includes the `PARAMETER` attribute.
- A Cray pointer variable cannot be declared to be of any other data type.
- A Cray character pointer cannot appear in a `DATA` statement.
- An array of Cray pointers is not allowed.
- A Cray pointer cannot be a component of a structure.

Restrictions on Cray pointees are as follows:

- A Cray pointee cannot appear in a `SAVE`, `DATA`, `EQUIVALENCE`, `COMMON`, `AUTOMATIC`, or `PARAMETER` statement.
- A Cray pointee cannot be a dummy argument; that is, it cannot appear in a `FUNCTION`, `SUBROUTINE`, or `ENTRY` statement.
- A function value cannot be a Cray pointee.
- A Cray pointee cannot be a structure component.

Note: Cray pointees can be of type character, but their Cray pointers are different from other Cray pointers; the two kinds cannot be mixed in the same expression.

The Cray pointer is a variable of type Cray pointer and can appear in a `COMMON` list or be a dummy argument in a subprogram.

The Cray pointee does not have an address until the value of the Cray pointer is defined; the pointee is stored starting at the location specified by the pointer. Any change in the value of a Cray pointer causes subsequent references to the corresponding pointee to refer to the new location.

Cray pointers can be assigned values in the following ways:

- A Cray pointer can be set as an absolute address. For example:

  ```
  Q = 0
  ```

- Cray pointers can have integer expressions added to or subtracted from them and can be assigned to or from integer variables. For example:

  ```
  P = Q + 100
  ```

However, Cray pointers are not integers. For example, assigning a Cray pointer to a real variable is not allowed.

The (nonstandard) LOC(3i) intrinsic function generates the address of a variable and can be used to define a Cray pointer, as follows:

```
P = LOC(X)
```

The following example uses Cray pointers in the ways just described:

```
SUBROUTINE SUB(N)
COMMON POOL(100000), WORDS(1000)
INTEGER BLK(128), WORD64
REAL A(1000), B(N), C(100000-N-1000)
POINTER(PBLK,BLK), (IA,A), (IB,B), &
      (IC,C), (ADDRESS,WORD64)
ADDRESS = LOC(WORDS) + 64
PBLK = LOC(WORDS)
IA = LOC(POOL)
IB = IA + 1000
IC = IB + N
```

BLK is an array that is another name for the first 128 words of array WORDS. A is an array of length 1000; it is another name for the first 1000 elements of POOL. B follows A and is of length N. C follows B. A, B, and C are associated with POOL. WORD64 is the same as BLK(17) because Cray pointers are byte addresses and the INTEGER elements of array BLK are each 4 bytes long.

If a pointee is of a noncharacter data type that is one machine word or longer, the address stored in a pointer is a word address. If the pointee is of type character or of a data type that is less than one word, the address is a byte address. The following example also uses Cray pointers:

```
PROGRAM TEST
CHARACTER X(10), Y(10), A(10)
POINTER (P,X), (Q,Y)

P = LOC(A(1))
Q = LOC(A(2))
I = P
J = Q

IF ( (J-I) .NE. (Q-P) )  THEN
   PRINT *, 'Not a byte addressable machine'
ELSE
   PRINT *, 'Byte addressable machine'
ENDIF
END
```

For purposes of optimization, the compiler assumes that the storage of a pointee is never overlaid on the storage of another variable; that is, it assumes that a pointee is not associated with another variable or array. This kind of association occurs when a Cray pointer has two pointees, or when two Cray pointers are given the same value. Although these practices are sometimes used deliberately (such as for equivalencing arrays), results can differ depending on whether optimization is turned on or off. You are responsible for preventing such association. For example:

```
POINTER(P,B), (P,C)
REAL X, B, C
P = LOC(X)
B = 1.0
C = 2.0
PRINT *, B
```

Because `B` and `C` have the same pointer, the assignment of 2.0 to `C` gives the same value to `B`; therefore, `B` will print as 2.0 even though it was assigned 1.0.

As with a variable in common storage, a pointee, pointer, or argument to a `LOC`(3i) intrinsic function is stored in memory before a call to an external procedure and is read out of memory at its next reference. The variable is also stored before a `RETURN` or `END` statement of a subprogram.

4.4 Derived Types

Unlike the intrinsic types that are defined by the language, you must define derived types. These types have the same utility as the intrinsic types. For example, variables of these types can be declared, passed as procedure arguments, and returned as function results.

A derived-type definition specifies a name for the type; this name is used to declare objects of the type. A derived-type definition also specifies components of the type, of which there must be at least one. A component can be of intrinsic or derived type; if it is of derived type, it can be resolved into components, called *ultimate components*. These ultimate components are of intrinsic type and can be pointers.

The direct components of a derived type are as follows:

- The components of the type
- For any nonpointer component that is of a derived type, the direct components of that derived type.

If the type definition appears in a module, the type definition may contain the keywords `PUBLIC` or `PRIVATE`. Generally, entities specified in a module can be kept private to the module and will not be available outside the module. This is true of data objects, module subprograms, and type definitions. By default, entities specified in a module are available to any program unit that accesses the module; that is, they have `PUBLIC` accessibility by default. This default can be changed by inserting a `PRIVATE` statement ahead of the specifications and definitions in the module. Individual entities can be specified to have either the `PUBLIC` or `PRIVATE` attribute regardless of the default. For a type definition, this can be accomplished by a `PUBLIC` or `PRIVATE` specifier in the `TYPE` statement of the type definition. The keyword `PRIVATE` can be used in two ways in type definitions in a module. One way makes the entire type private to the module; the other way allows the type name to be known outside the module, but not the names or attributes of its components. A separate `PRIVATE` statement that mentions the type name or a `PRIVATE` specifier in the `TYPE` statement of the type definition provides the first of these. An optional `PRIVATE` statement inside the type definition provides the second. See Section 4.4.1, page 97, for examples of a private type and a public type with private components.

A type definition can contain a `SEQUENCE` statement. The Fortran standard allows a processor to rearrange the components of a derived type in any convenient order. However, if a `SEQUENCE` statement appears inside the type definition, the type is considered to be a *sequence type*. In this case, the processor must allocate storage for the components in the declared order so that structures declared to be of the derived

type can appear in COMMON and EQUIVALENCE statements. See Section 4.4.1, page 97, for an example of a sequence type.

Default initialization is specified for a component of an object of derived type when initialization appears in the component declaration. The object is initialized as specified in the derived type definition even if the definition is private or inaccessible. Default initialization applies to dummy arguments with INTENT (OUT) and function return values. Unlike explicit initialization, default initialization does not imply that the object has the SAVE attribute. If a component has default initialization, it is not required that default initialization be specified for other components of the derived type.

A derived type has a set of values that is every combination of the permitted values for the components of the type. The language provides a syntax for constants of the intrinsic types; it provides a somewhat similar mechanism, called a *structure constructor*, to specify a value for a derived type. A constructor can be used in the following places:

- In PARAMETER statements and in type declaration statements to define derived-type named constants
- In DATA statements to specify initial values
- As structure-valued operands in expressions

User-defined functions and subroutines must be used to define operations on entities of derived type. Thus, the four properties of the intrinsic types (possession of a name, a set of values, a set of operations, and a syntactic mechanism to specify constant values) are also provided for derived types.

4.4.1 Derived Type Definition

A derived type definition gives a derived type a name and specifies the types and attributes of its components. A derived type definition begins with a TYPE statement, ends with an END TYPE statement, and has component declarations in between. The following example defines type PATIENT:

```
TYPE PATIENT
   INTEGER          PULSE_RATE
   REAL             TEMPERATURE
   CHARACTER*(300)    PROGNOSIS
END TYPE PATIENT
```

The format of a *derived_type_def* is as follows:

```
TYPE [ [, access_spec ] :: ] type_name
   [ private_sequence_stmt ] ...
   component_def_stmt
   [ component_def_stmt ] ...
   END TYPE [ type_name ]
```

A derived-type definition is defined as follows:

derived_type_def	**is**	*derived_type_stmt* [*private_sequence_stmt*] ... *component_def_stmt* [*component_def_stmt*] ... *end_type_stmt*
private_sequence_stmt	**is**	PRIVATE
	or	SEQUENCE
derived_type_stmt	**is**	TYPE [[, *access_spec*] ::] *type_name*
end_type_stmt	**is**	END TYPE [*type_name*]
component_def_stmt	**is**	*type_spec* [[, *component_attr_spec_list*] ::] *component_decl_list*
component_attr_spec	**is**	POINTER
	or	DIMENSION (*component_array_spec*)
component_array_spec	**is**	*explicit_shape_spec_list*
	or	*deferred_shape_spec_list*
component_decl	**is**	*component_name* [(*component_array_spec*)] [* *char_length*] [*component_initialization*]

For *access_spec*, specify either PRIVATE or PUBLIC.

The *component_array_spec* must be a deferred-shape array if the POINTER attribute is present; otherwise, it must be an explicit-shape array.

The name of the derived type must not be the same as any intrinsic type or locally accessible name in the same class; it has the scope of local names declared in the

scoping unit, which means that it is accessible by use or host association in other scoping units. A component name has the scope of the type definition only; another type definition in the same scoping unit may specify the same component name. For more information on local entities and scope, see the *MIPSpro Fortran Language Reference Manual, Volume 2*.

If the `END TYPE` statement is followed by a name, it must be the name specified in the `TYPE` statement.

A type can be defined only once within a scoping unit.

A `PRIVATE` statement must not appear more than once in a type definition.

A `SEQUENCE` statement must not appear more than once in a type definition

If `SEQUENCE` is present, all derived types specified as components must also be sequence types.

The keywords `PUBLIC` and `PRIVATE` can appear only if the definition is in the specification part of a module.

There must be at least one component definition statement in a type definition.

No component attribute can appear more than once in a specified component definition statement.

A component can be declared to have the same type as the type being defined only if it has the `POINTER` attribute.

An array component without the `POINTER` attribute must be specified with an explicit-shape specification where the bounds are integer constant expressions.

If a component is of type character with a specified length, the length must be an integer constant specification expression. If the length is not specified, it is 1.

If *component_initialization* is specified, a double colon separator (`::`) must appear before the *component_decl_list*.

If `=>` appears in a *component_initialization*, the `POINTER` attribute must appear in the *component_attr_spec_list*. If `=` appears in a *component_initialization*, the `POINTER` attribute cannot appear in the *component_attr_spec_list*.

If *initialization_expr* appears for a nonpointer component, that component in any object of the type is initially defined or becomes defined as specified in *MIPSpro Fortran Language Reference Manual, Volume 2* with the value determined from *initialization_expr*. The *initialization_expr* is evaluated in the scoping unit of the type

definition. The evaluation rules are the same as if the component were a *variable=initialization_expr*. If *component_name* is a type for which *default_initialization* is specified for a component, the *default_initialization* specified by *initialization_expr* overrides the default initialization specified for that component. Explicit initialization in a type declaration statement overrides default initialization. An object of a type with default initialization must not be specified in a DATA statement.

The following example shows a derived-type definition with four components (three integer and one character):

```
TYPE COLOR
   INTEGER :: HUE, SHADE, INTENSITY
   CHARACTER(LEN = 30) :: NAME
END TYPE COLOR
```

The following is a format for declaring variables of derived type:

```
TYPE (type_name) [ [ , attribute_list ] :: ] entity_list
```

For example, variables of type COLOR can be declared as follows:

```
TYPE(COLOR) MY_FAVORITE
TYPE(COLOR) RAINBOW(7)
TYPE(COLOR), DIMENSION (100) :: SELECTIONS
```

The object MY_FAVORITE is a structure. The objects RAINBOW and SELECTIONS are arrays of structures.

Note that the initial statement of a type definition and the statement used to declare objects of derived type both begin with the keyword TYPE. The initial statement of a type definition is called a *derived-type statement*, and the statement used to declare objects of derived type is called a TYPE statement. The type name in a derived-type statement is not enclosed in parentheses, whereas the type name in a TYPE statement is.

A component of a structure is referenced using a percent sign, as in the following template:

```
parent_structure % component_name
```

Examples:

```
MY_FAVORITE % HUE
RAINBOW(3) % NAME
```

The following examples show definitions of derived types. Each example illustrates a different aspect of a type definition:

- A derived type with a component of a different derived type
- A derived type with a pointer component
- A derived type with a pointer component of the type being defined
- A private type definition
- A public type definition with private components

Example 1: A derived type can have a component that is of a different derived type. Type WEATHER in the following example has a component of type TEMPERATURES.

```
TYPE TEMPERATURES
   INTEGER :: HIGH, LOW
END TYPE TEMPERATURES

TYPE WEATHER
   CHARACTER(LEN = 32) :: CITY
   TYPE(TEMPERATURES) :: RANGE(1950:2050)
END TYPE WEATHER

TYPE(WEATHER) WORLDWIDE(200)
```

WORLDWIDE is an array of type WEATHER. Components of an element of the array are referenced as follows:

```
WORLDWIDE(I)%CITY = "Nome"
WORLDWIDE(I)%RANGE(1990)%LOW = -83
```

Example 2: A derived type can have a component that is a pointer, as follows:

```
TYPE ABSTRACT
   CHARACTER(LEN = 50) TITLE
   INTEGER NUM_OF_PAGES
   CHARACTER, POINTER :: TEXT(:)
END TYPE ABSTRACT
```

Any object of type ABSTRACT will have three components: TITLE, NUM_OF_PAGES, and TEXT. TEXT is a pointer to an array of character strings, each of which is of length one. The array size is determined during program execution. The space for the target of TEXT can be allocated, or TEXT can be pointer-assigned to existing space. For information on the ALLOCATE statement, see Section 6.5.1, page 202. For information on pointer assignment, see Section 7.5.3, page 280.

Example 3: A derived type can have a pointer component that is of the type being defined. This is useful in creating linked lists and trees, as follows:

```
TYPE LINK
   REAL VALUE
   TYPE(LINK), POINTER :: PREVIOUS
   TYPE(LINK), POINTER :: NEXT
END TYPE LINK
```

Example 4: A type definition in a module can be kept private to the module, as follows:

```
TYPE, PRIVATE :: FILE
   INTEGER DRAWER_NO
   CHARACTER(LEN = 20) FOLDER_NAME
   CHARACTER(LEN = 5) ACCESS_LEVEL
END TYPE FILE
```

When a module that contains this type definition is accessed by another scoping unit, the type FILE is not available.

Example 5: A type definition can be public while its components are kept private, as follows:

```
MODULE COORDINATES
   TYPE POINT
      PRIVATE
      REAL X, Y
   END TYPE POINT
      ...
END MODULE COORDINATES
```

In a program unit that uses module COORDINATES, variables of type POINT can be declared. Values of type POINT can be passed as arguments. If the program unit is a function, a value of type POINT can be returned as the result. However, the internal

structure of the type (its components) is not available. If the type `POINT` is changed to the following, no other program unit that uses `COORDINATES` will need to be changed:

```
TYPE POINT
   PRIVATE
   REAL RHO, THETA
END TYPE POINT
```

If a subprogram dummy argument is of derived type, the corresponding actual argument must be of the same type. There are two ways in which objects in different scoping units can be declared to be of the same type. Two data entities have the same type if they are declared with reference to the same type definition. The definition can appear in a module that is accessed or, in the case of an internal or module procedure, in the host scoping unit.

```
MODULE SHOP
   TYPE COMPONENT
      CHARACTER(LEN = 20) NAME
      INTEGER CATALOG_NUM
      REAL WEIGHT
   END TYPE COMPONENT
   TYPE(COMPONENT) PARTS(100)
CONTAINS
   SUBROUTINE GET_PART(PART, NAME)
      TYPE(COMPONENT) PART
      CHARACTER(LEN = *) NAME
      DO I = 1, 100
         IF (NAME .EQ. PARTS(I)%NAME) THEN
            PART = PARTS(I)
            RETURN
         END IF
      END DO
      PRINT *, "Part not available"
      PART%NAME = "none"
      PART%CATALOG_NUM = 0
      PART%WEIGHT = 0.0
   END SUBROUTINE GET_PART
      . . .
END MODULE SHOP

PROGRAM BUILD_MACHINE
   USE SHOP
```

```
   TYPE(COMPONENT) MOTOR(20)
   TOTAL_WEIGHT = 0.0
   CALL GET_PART(MOTOR(1), "VALVE")
   IF (MOTOR(1)%WEIGHT .NE. 0) THEN
      TOTAL_WEIGHT = TOTAL_WEIGHT + MOTOR(1)%WEIGHT
   ELSE
      . . .
   ENDIF
   . . .
END PROGRAM BUILD_MACHINE
```

Module procedure GET_PART has access to the type COMPONENT because the type definition appears in its host. Program BUILD_MACHINE has access to the type because it uses module SHOP. This allows a variable of the type, such as MOTOR(1), to be passed as an actual argument.

The other way to declare data entities in different scoping units to be of the same type is provided for programmers who choose not to use a module. Instead of a single type definition in the module, a sequence type can be defined in each of the scoping units that need access to the type. Each of the type definitions must specify the same name; the SEQUENCE property; have no private components; and have components that agree in order, name, and attributes. If this is the case, data entities declared in any of these scoping units to be of the named type are considered to be of the same type. In the following, program BUILD_MACHINE is restated to illustrate the differences between the two ways:

```
PROGRAM BUILD_MACHINE
   TYPE COMPONENT
      SEQUENCE
      CHARACTER(LEN = 20) NAME
      INTEGER CATALOG_NUM
      REAL WEIGHT
   END TYPE COMPONENT
   TYPE(COMPONENT) PARTS, MOTOR(20)
   COMMON /WAREHOUSE/ PARTS(100)
   TOTAL_WEIGHT = 0.0
   CALL GET_PART(MOTOR(1), "VALVE")
   IF (MOTOR(1)%WEIGHT .NE. 0) THEN
      TOTAL_WEIGHT = TOTAL_WEIGHT + MOTOR(1)%WEIGHT
   ELSE
      . . .
   ENDIF
```

```
      . . .
END PROGRAM BUILD_MACHINE
SUBROUTINE GET_PART(PART, NAME)
   TYPE COMPONENT
      SEQUENCE
      CHARACTER(LEN = 20) NAME
      INTEGER CATALOG_NUM
      REAL WEIGHT
   END TYPE COMPONENT
   TYPE(COMPONENT) PART, PARTS
   CHARACTER(LEN = *) NAME
   COMMON /WAREHOUSE/ PARTS(100)
   DO I = 1, 100
      IF (NAME .EQ. PARTS(I)%NAME) THEN
         PART = PARTS(I)
         RETURN
      END IF
   END DO
   PART%NAME = "none"
   PART%CATALOG_NUM = 0
   PART%WEIGHT = 0.0
   PRINT *, "Part not available"
END SUBROUTINE GET_PART
   . . .
```

In this example, type COMPONENT in program BUILD_MACHINE and type COMPONENT in subroutine GET_PART are the same because they are sequence types with the same name; have no private components; and have components that agree in order, name, and attributes. This allows variables of the type to appear in COMMON and be passed as arguments. Note that this example is less concise, particularly if more procedures need to access the type definition, and therefore may be more error prone than the previous example.

Type COMPONENT is a sequence type because its definition contains a SEQUENCE statement. If all of the ultimate components of a sequence type are of type default integer, default real, double-precision real, default complex, or default logical, and are not pointers, the type is a *numeric sequence type*. An object of numeric sequence type can be equivalenced to default numeric objects.

If all of the ultimate components of a sequence type are of type character and are not pointers, the type is a *character sequence type*. An object of character sequence type may be equivalenced to character objects.

A pointer component of a derived type can have as its target an object of that derived type. The type definition can specify that in objects declared to be of this type, such a pointer is default initialized to disassociated. In the following example, type NODE is created and is used to construct linked lists of objects of type NODE:

```
TYPE NODE
   INTEGER :: VALUE
   TYPE (NODE), POINTER :: NEXT_NODE => NULL ( )
END TYPE
```

Initialization need not be specified for each component of a derived type. For example:

```
TYPE DATE
   INTEGER DAY
   CHARACTER (LEN = 5) MONTH
   INTEGER :: YEAR = 1994     ! PARTIAL DEFAULT INITIALIZATION
END TYPE DATE
```

In the following example, the default initial value for the YEAR component of TODAY is overridden by explicit initialization in the type declaration statement:

```
TYPE (DATE), PARAMETER :: TODAY = DATE (21, "Feb.", 1995)
```

4.4.2 Derived Type Values

The set of values of a derived type consists of all combinations of the possibilities for component values that are consistent with the components specified in the type definition.

4.4.3 Derived Type Operations

Any operation involving a derived-type entity must be defined explicitly by a function with an OPERATOR interface. Assignment, other than the intrinsic assignment provided for entities of the same derived type, must be defined by a subroutine with an ASSIGNMENT interface. See the *MIPSpro Fortran Language Reference Manual, Volume 2*, for a description.

Suppose it is desirable to determine the number of words and lines in a section of text. The information is available for each paragraph. A type named PARAGRAPH is defined as follows:

```
TYPE PARAGRAPH
   INTEGER NUM_WORDS, NUM_LINES
   CHARACTER(LEN = 30) SUBJECT
END TYPE PARAGRAPH
```

Suppose that it is now desirable to define an operator for adding the counts associated with the paragraphs. The following OPERATOR interface is required for the function that defines the addition operation for objects of type PARAGRAPH:

```
INTERFACE OPERATOR (+)
   MODULE PROCEDURE ADDP
END INTERFACE
```

The following definition of addition for objects of type PARAGRAPH adds the words and lines, but it does nothing with the component SUBJECT because that would have no useful meaning:

```
TYPE(PARAGRAPH) FUNCTION ADDP(P1, P2)
   TYPE(PARAGRAPH) P1, P2
   INTENT(IN) P1, P2
   ADDP%NUM_WORDS = P1%NUM_WORDS + P2%NUM_WORDS
   ADDP%NUM_LINES = P1%NUM_LINES + P2%NUM_LINES
END FUNCTION ADDP
```

If the following variables were declared, the expression BIRDS+BEES would be defined and could be evaluated in the module subprogram as well as any program unit accessing the module:

```
TYPE(PARAGRAPH) BIRDS, BEES
```

4.4.4 Syntax for Specifying Derived-type Constant Expressions

When a derived type is defined, a structure constructor for that type is defined automatically. The structure constructor is used to specify values of the type. It specifies a sequence of values, one for each of the components of the type. A structure constructor whose values are all constant expressions is a derived-type constant expression. (This is why a derived-type value is formed by a constructor. There is no such thing as a structure constant; there are only structure constructors, some of which may be constant expressions.) A named constant of user-defined type can be assigned such a value. Structure constructors are described in Section 4.5, page 108.

A component of a derived type can be an array. In this case a mechanism called an array constructor is used to specify that component of the type. Array constructors are described in Section 4.6, page 110, and a general discussion of arrays can be found in Section 6.4, page 191.

4.5 Structure Constructors

A *structure constructor* is a mechanism that is used to specify a value of a derived type by specifying a sequence of values for the components of the type. If a component is of derived type, an embedded structure constructor is required to specify the value of that component. If a component is an array, an embedded array constructor is required to specify the values for that component.

A structure constructor is the name of the type followed by a sequence of component values in parentheses. For example, a value of type `COLOR` (from Section 4.4.1, page 97) can be constructed with the following structure constructor:

```
COLOR(I, J, K, "MAGENTA")
```

For information on derived types, see Section 4.4.1, page 97.

The format for a *structure_constructor* is defined as follows:

structure_constructor	**is** *type_name* (*expr_list*)

There must be a value in the expression list for each component.

The expressions must agree in number and order with the components of the derived type. Values may be converted (in the same way they would be for an assignment statement) to agree in type, kind, length, and, in some cases, rank, with the components. The conversions permitted are those for intrinsic assignment, in which the component is the variable on the left and the expression is the one given in the structure constructor corresponding to the component. Rank must be conformable according to the rules of assignment conformance. That is, the shapes must conform or the expression can be a scalar broadcast to an array component.

If a component is an explicit-shape array (that is, a nonpointer array), the array constructor for it in the expression list must be the same shape as the component.

If a component is a pointer, the value for it in the expression list must evaluate to an allowable target for the pointer. A constant is not an allowable target.

A structure constructor must not appear before that type is defined.

The structure constructor for a private type or a public type with private components is not available outside the module in which the type is defined.

If the values in a structure constructor are constants, you can use the structure constructor to specify a named constant, as in the following example:

```
PARAMETER( TEAL = COLOR(14, 7, 3, "TEAL") )
TYPE(COMPONENT), PARAMETER :: NO_PART = COMPONENT("none", 0, 0.0) )
```

Following are several examples of structure constructors for types with somewhat different components:

- A type with a component that is of derived type
- A type with an array component
- A type with a pointer component

Example 1: A structure constructor for a type that has a derived type as a component must provide a value for each of the components. A component may be of derived type, in which case a structure constructor is required for the component. In the following example, type RING has a component of type STONE:

```
TYPE STONE
    REAL           CARETS
    INTEGER        SHAPE
    CHARACTER(30)  NAME
END TYPE STONE

TYPE RING
    REAL           EST_VALUE
    CHARACTER(30)  INSURER
    TYPE (STONE)   JEWEL
END TYPE RING
```

If OVAL is a named integer constant, an example of a structure constructor for a value of type RING is as follows:

```
RING (5000.00, "Lloyds", STONE(2.5, OVAL, "emerald") )
```

Example 2: If a type is specified with an array component, the value that corresponds to the array component in the expression list of the structure constructor must conform with the shape of the array component. For example, type ORCHARD has an array component as follows:

```
TYPE ORCHARD
    INTEGER               AGE, NUM_TREES
    CHARACTER(LEN = 20)   VARIETY(10)
END TYPE
```

Assume the following declarations:

```
CHARACTER(LEN = 20) CATALOG(16, 12)
PARAMETER(LEMON = 3)
```

A structure constructor for a value of type ORCHARD is as follows:

```
ORCHARD (5, ROWS * NUM_PER_ROW, CATALOG(LEMON, 1:10) )
```

Example 3: When a component of the type is a pointer, the corresponding structure constructor expression must evaluate to an object that would be an allowable target for such a pointer in a pointer assignment statement. Assume that the variable SYNOPSIS is declared as follows:

```
CHARACTER, TARGET :: SYNOPSIS(4000)
```

The following value of the type ABSTRACT (from Section 4.4.1, page 97) can then be constructed:

```
ABSTRACT("War and Peace", 1025, SYNOPSIS)
```

A constant expression cannot be constructed for a type with a pointer component because a constant is not an allowable target in a pointer assignment statement.

4.6 Array Constructors

An array constructor is used to specify the value of an array. More precisely, an *array constructor* is a mechanism that is used to specify a sequence of scalar values that is interpreted as a rank-one array. Syntactically, it is a sequence of scalar values and implied-DO specifications enclosed in parentheses and slashes. As with structures, there is no such thing as an array constant. There are only array constructors, some of which may be constant expressions, as follows:

```
REAL VECTOR_X(3), VECTOR_Y(2), RESULT(100)
   . . .
RESULT(1:8) = (/ 1.3, 5.6, VECTOR_X, 2.35, VECTOR_Y /)
```

The value of the first eight elements of RESULT is constructed from the values of VECTOR_X and VECTOR_Y and three real constants in the specified order. If a rank-two or greater array appears in the value list, the values of its elements are taken in array element order. If it is necessary to construct an array of rank greater than one, the RESHAPE(3i) intrinsic function can be applied to an array constructor.

The format for an *array_constructor* is as follows:

array_constructor	**is**	(/ *ac_value_list* /)
ac_value	**is**	*expr*
	or	*ac_implied_do*
ac_implied_do	**is**	(*ac_value_list*, *ac_implied_do_control*)
ac_implied_do_control	**is**	*ac_do_variable* = *scalar_int_expr*, *scalar_int_expr* [, *scalar_int_expr*]
ac_do_variable	**is**	*scalar_int_variable*

Each *ac_value* expression in the array constructor must have the same type, kind type, and length parameters. In particular, this means that if each *ac_value* is a character literal constant, each constant must have the same length.

The type and type parameters of an array constructor are those of its *ac_value* expressions.

If the *ac_implied_do* yields no values, the array is a rank one, zero-sized array.

An *ac_do_variable* must be a scalar integer named variable. This variable has the scope of this *ac_implied_do*.

If an *ac_implied_do* is contained within another *ac_implied_do*, they must not have the same *ac_do_variable*.

Three possibilities for an *ac_value* are as follows:

- It can be a scalar expression, as is each *ac_value* in the following:

  ```
  (/ 1.2, 3.5, 1.1 /)
  ```

- It can be an array expression, as is each *ac_value* in the following:

```
(/ A(I, 1:3), A(I+1, 6:8) /)
```

- It can be an implied-DO specification, as in the following:

```
(/ (SQRT(REAL(I)), I = 1, 9) /)
```

The possibilities can be mixed in a single array constructor, as follows:

```
(/ 1.2, B(2:6,:), (REAL(I), I = 1, N), 3.5 /)
```

If an *ac_value* is an array expression, the values of the elements of the expression in array element order become the values of the array constructor. For example, the values that result from the example in possibility 2 are as follows:

```
(/ A(I,1), A(I,2), A(I,3), A(I+1,6), A(I+1,7), A(I+1,8) /)
```

For more information on array element order, see Section 6.4.7, page 199.

If an *ac_value* is an implied-DO specification, it is expanded to form a sequence of values under control of the *ac_do_variable* as in the DO construct. For example, the values that result from the example in possibility 3 are as follows:

```
(/1.0, 1.414, 1.732, 2.0, 2.236, 2.449, 2.645, 2.828, 3.0/)
```

For more information on the DO construct, see Section 8.5, page 305.

If every expression in an array constructor is a constant expression, the array constructor is a constant expression as in the example above. Such an array constructor can be used to assign a value to a named constant, as follows:

```
REAL X(3), EXTENDED_X(4)
PARAMETER(X = (/ 2.0, 4.0, 6.0 /) )
REAL, PARAMETER :: EXTENDED_X = (/ 0.0, X /) )
```

The following are examples of array constructors. Examples 1 and 2 demonstrate the construction of arrays; examples 3 and 4 demonstrate the construction of values of derived type when the type has an array component:

1. A constructor for a rank two array
2. A constructor for an array of derived type
3. A constructor for a value of derived type with an array component
4. A constructor for a value of derived type with a rank two array component

Example 1: To create a value for an array of rank greater than one, the RESHAPE(3i) intrinsic function must be used. With this function, a one-dimensional array may be reshaped into any allowable array shape.

```
Y = RESHAPE(SOURCE = (/ 2.0, (/ 4.5, 4.0 /), Z /),  &
      SHAPE = (/ 3, 2 /))
```

If Z has the value given in possibility 1 above, then Y is a 3 by 2 array with the following elements:

```
2.0     1.2
4.5     3.5
4.0     1.1
```

Example 2: It might be necessary to construct an array value of derived type.

```
TYPE PERSON
   INTEGER AGE
   CHARACTER(LEN = 40) NAME
END TYPE PERSON

TYPE(PERSON) CAR_POOL(3)

CAR_POOL = (/ PERSON(35, "SCHMITT"),  &
      PERSON(57, "LOPEZ"), PERSON(26, "YUNG") /)
```

Example 3: When one of the components of a derived type is an array, then an array constructor must be used in the structure constructor for the derived type. Suppose that the definition for type COLOR is as follows, which differs slightly from that stated previously:

```
TYPE COLOR
   INTEGER PROPERTIES(3)
   CHARACTER(LEN = 30) NAME
END TYPE COLOR
```

The following value of the revised type COLOR can be constructed:

```
COLOR((/ 5, 20, 8 /), "MAGENTA")
```

Example 4: A derived type might contain an array of rank two or greater, as follows:

```
TYPE LINE
   REAL     COORD(2, 2)
```

```
   REAL     WIDTH
   INTEGER  PATTERN
END TYPE LINE
```

The values of COORD are the coordinates x_1, y_1 and x_2, y_2 representing the end points of a line. WIDTH is the line width in centimeters. PATTERN is 1 for a solid line, 2 for a dashed line, and 3 for a dotted line. An object of type LINE is declared and given a value as follows:

```
TYPE(LINE) SLOPE
   . . .
SLOPE = LINE(RESHAPE((/ 0.0, 1.0, 0.0, 2.0 /), (/ 2, 2 /)), 0.1, 1)
```

The RESHAPE(3i) intrinsic function is used to construct a value that represents a solid line from (0,0) to (1,2) of width 0.1 centimeters.

Declarations

Declarations are used to specify the type and other attributes of program entities. The attributes that an entity possesses determine how the entity can be used in a program. Every variable and function has a type, which is the most important of the attributes; type is discussed in Chapter 4, page 67. However, type is only one of a number of attributes that an entity may possess. Some entities, such as subroutines and namelist groups, do not have a type but may possess other attributes. In addition, there are relationships among objects that can be specified by `EQUIVALENCE`, `COMMON`, and `NAMELIST` statements. Declarations are used to specify these attributes and relationships.

Generally, Fortran keywords are used to declare the attributes for an entity. The following list summarizes these keywords:

Attribute	**Keyword**
Type	`INTEGER`, `REAL` (and `DOUBLE PRECISION`), `COMPLEX`, `LOGICAL`, `CHARACTER`, `TYPE` (user-defined name)
Array properties	`DIMENSION`, `ALLOCATABLE`
Pointer properties	`POINTER`, `TARGET`
Setting values	`DATA`, `PARAMETER`
Object accessibility and use	`PUBLIC`, `PRIVATE`, `INTENT`, `OPTIONAL`, `SAVE`, `AUTOMATIC`
Procedure properties	`EXTERNAL`, `INTRINSIC`

The attributes are described and illustrated in turn using either of the two forms that attribute specifications can take: entity-oriented and attribute-oriented.

For objects that have a type, other attributes can be included in the type declaration statement. For example:

```
INTEGER, SAVE :: A, B, C
```

Collecting the attributes into a single statement is sometimes more convenient for readers of programs. It eliminates searching through many declaration statements to locate all attributes of a particular object. Emphasis can be placed on an object and its attributes (entity-oriented declaration) or on an attribute and the objects that possess the attribute (attribute-oriented declaration), whichever is preferred by a programmer.

In both forms, dimensionality can be specified as an attribute or as an attachment to the object name.

The following are examples of entity-oriented declaration statements:

```
REAL, DIMENSION(20), SAVE :: X
```

or

```
REAL, SAVE :: X(20)
```

The following are examples of attribute-oriented declaration statements:

```
REAL X
DIMENSION X(20)
SAVE X
```

or

```
REAL X(20)
SAVE X
```

If attributes are not declared for a data object, defaults apply. Generally, if an attribute is not specified for an object, it is assumed that the object does not possess the attribute. However, each data object has a type, and if this is not explicitly specified, it is assumed from the first letter of its name. You can use the `IMPLICIT` statement to specify any intrinsic or user-defined type for an initial letter or a range of initial letters. The `IMPLICIT NONE` statement, on the other hand, removes implicit typing and thus requires explicit type declarations for every named data object in the scoping unit.

Fortran provides dynamic data objects that can be sized at the time a program is executed. These include allocatable arrays and objects with the `POINTER` attribute. They also include automatic data objects (arrays of any type and character strings) that are created on entry into a procedure. Only objects whose size may vary are called automatic.

Other declarations (`NAMELIST`, `EQUIVALENCE`, and `COMMON`) establish relationships among data objects. The `NAMELIST` statement is used to name a collection of objects so that they can be referenced by a single name in an input/output (I/O) statement. `EQUIVALENCE` provides references to storage by more than one name. `COMMON` provides a mechanism to share storage among the different units of a program.

5.1 Type Declaration Statements

A type declaration type statement begins with the name of the type, optionally lists other attributes, then ends with a list of variables that possess these attributes. In addition, a type declaration statement may include an initial value for a variable. If the PARAMETER attribute is specified on a type statement, the statement must include the value of the named constant.

The *type_declaration_stmt* is defined as follows:

	type_declaration_stmt	**is**	*type_spec* [[, *attr_spec*]... ::] *entity_decl_list*
	type_spec	**is**	INTEGER *kind_selector*
EXT		**or**	INTEGER* *length_value*
		or	REAL *kind_selector*
EXT		**or**	REAL* *length_value*
		or	DOUBLE PRECISION
EXT		**or**	DOUBLE PRECISION* *length_value*
		or	COMPLEX *kind_selector*
EXT		**or**	COMPLEX* *length_value*
		or	CHARACTER *char_selector*
		or	LOGICAL *kind_selector*
EXT		**or**	LOGICAL* *length_value*
		or	TYPE (*type_name*)
EXT		**or**	POINTER (*pointer_name*, *pointee_name* [(*array_spec*)])
			[, (*pointer_name*, *pointee_name* [(*array_spec*)])] ...
	attr_spec	**is**	PARAMETER
		or	*access_spec*
		or	ALLOCATABLE
EXT		**or**	AUTOMATIC
		or	DIMENSION (*array_spec*)

	or	`EXTERNAL`
	or	`INTENT (`*intent_spec*`)`
	or	`INTRINSIC`
	or	`OPTIONAL`
	or	`POINTER`
	or	`SAVE`
	or	`TARGET`
access_spec	**is**	`PUBLIC`
	or	`PRIVATE`
entity_decl	**is**	*object_name* `[(`*array_spec*`)] [ * `*char_length*` ] [ `*initialization*` ]`
	or	*function_name* `[ * `*char_length*` ]`
kind_selector	**is**	`([ KIND=]` *scalar_int_initialization_expr*`)`
initialization_expr	**is**	`=`*intialization_expr*
	or	`=>NULL()`

The double colon symbol (`::`) is required in a type declaration statement only when the type declaration statement contains two or more attributes or when it contains an *initialization_expr*.

If `=>` appears in *initialization*, the object must have the `POINTER` attribute. If `=` appears in *initialization*, the object cannot have the `POINTER` attribute.

The type specification can override or confirm the implicit type indicated by the first letter of the entity name according to the implicit typing rules in effect.

The same attribute cannot appear more than once in a given type declaration statement.

An entity must not be assigned any attribute more than once in a scoping unit.

The value specified in a kind selector must be a kind type parameter allowed for that type by the compiler.

The character length option can appear only when the type specification is `CHARACTER`.

An initialization expression must be included if the PARAMETER attribute is specified.

A function name must be the name of an external function, an intrinsic function, a function dummy procedure, or a statement function.

An array function result name must be specified as an explicit-shape array unless it has the POINTER attribute, in which case it must be specified as a deferred-shape array. For information on array properties, see Section 5.3, page 131.

Other rules and restrictions pertain to particular attributes; these are covered in the sections describing those attributes. The attributes that can be used with the attribute being described are also listed. The simple forms that appear in the following sections to illustrate attribute specification in a type declaration statement show the attribute being described first in the attribute list, but attributes can appear in any order. If these simple forms are used to construct statements, the statements will be correct, but other variations are permitted.

The following examples show type declaration statements:

```
REAL A(10)
LOGICAL, DIMENSION(5,5) :: MASK_1, MASK_2
COMPLEX :: CUBE_ROOT = (-0.5, 0.867)
INTEGER, PARAMETER :: SHORT = SELECTED_INT_KIND(4)
INTEGER(SHORT) :: K           ! Range of -9999 to 9999
REAL, ALLOCATABLE :: A1(:, :), A2(:, :, :)
TYPE(PERSON) CHAIRMAN
TYPE(NODE), POINTER :: HEAD_OF_CHAIN, END_OF_CHAIN
REAL, INTENT(IN) :: ARG1
REAL, INTRINSIC :: SIN
REAL, POINTER, DIMENSION (:) :: S => NULL()
```

5.1.1 Integer

An INTEGER statement declares the names of entities to be of type integer. If a kind selector is present, it specifies the representation method. For more information on integer type, see Section 4.3.1, page 75.

The compiler supports the following formats for declaring objects of this type:

```
INTEGER [([ KIND = ] kind_param) ] [[ , attribute_list ] :: ] entity_list

INTEGER * length_value [ [ , attribute_list ] :: ] entity_list
```

For *kind_param* values, see Section 4.3.1, page 75. The *length_value* values correspond to the *kind_param* values and are 1, 2, 4 (default), and 8.

The following are examples of entity-oriented declaration statements:

```
INTEGER, DIMENSION(:), POINTER :: MILES, HOURS
INTEGER(SHORT), POINTER :: RATE, INDEX
```

The following are examples of attribute-oriented declaration statements:

```
INTEGER :: MILES, HOURS
INTEGER(SHORT) :: RATE, INDEX
DIMENSION :: MILES(:), HOURS(:)
POINTER :: MILES, HOURS, RATE, INDEX
```

ANSI/ISO: The Fortran standard does not specify the `INTEGER*`*length_value* syntax. It is recommended that this syntax not be used in any new code.

5.1.2 Real

A `REAL` statement declares the names of entities to be of type real. If a kind selector is present, it specifies the representation method. For more information on real type, see Section 4.3.2, page 79.

The compiler supports the following formats for declaring objects of this type:

```
REAL [ ([ KIND = ] kind_param) ] [ [ , attribute_list ] :: ] entity_list
```

```
REAL * length_value [ [ , attribute_list ] :: ] entity_list
```

For *kind_param* values, see Section 4.3.2, page 79. The *length_value* values correspond to the *kind_param* values; on IRIX systems, the values are as follows: 4 (default), 8, and 16.

The following examples show entity-oriented declaration statements:

```
REAL(KIND = HIGH), OPTIONAL :: VARIANCE
REAL, SAVE :: A1(10, 10), A2(100, 10, 10)
```

The following examples show attribute-oriented declaration statements:

```
REAL(KIND = HIGH) VARIANCE
REAL A1(10, 10), A2(100, 10, 10)
OPTIONAL VARIANCE
SAVE A1, A2
```

ANSI/ISO: The Fortran standard does not specify the REAL*_length_value_ syntax. It is recommended that this syntax not be used in any new code.

5.1.3 Double Precision

A DOUBLE PRECISION statement declares the names of entities to be of real type with a representation method that represents more precision than the default real representation. The DOUBLE PRECISION statement is outmoded because REAL with the appropriate kind parameter value is equivalent. A kind selector is not permitted in the DOUBLE PRECISION statement. For more information on the real data type, see Section 4.3.2, page 79.

The compiler supports the following formats for declaring objects of this type:

```
DOUBLE PRECISION [ [ , attribute_list ] :: ] entity_list
```

```
DOUBLE PRECISION * 16 [ [, attribute_list ] :: ] entity_list
```

The following examples show entity-oriented declaration statements:

```
DOUBLE PRECISION, DIMENSION(N,N) :: MATRIX_A, MATRIX_B
DOUBLE PRECISION, POINTER :: C, D, E, F(:, :)
```

The following examples show attribute-oriented declaration statements:

```
DOUBLE PRECISION :: MATRIX_A, MATRIX_B, C, D, E, F
DIMENSION :: MATRIX_A(N, N), MATRIX_B(N, N), F(:, :)
POINTER :: C, D, E, F
```

If DOUBLE is a named integer constant that has the value of the kind parameter of the double-precision real type on the target platform, the preceding entity-oriented declaration statements could be written as follows:

```
REAL (DOUBLE), DIMENSION (N,N) :: MATRIX_A, MATRIX_B
REAL (DOUBLE), POINTER :: C, D, E, F(:,:)
```

ANSI/ISO: The Fortran standard does not specify the DOUBLE PRECISION*16 syntax. It is recommended that this syntax not be used in any new code.

5.1.4 Complex

A COMPLEX statement declares the names of entities to be of type complex. If a kind selector is present, it specifies the representation method. For more information on complex type, see Section 4.3.3, page 83.

The compiler supports the following formats for declaring objects of this type:

```
COMPLEX [([ KIND = ] kind_param)] [ [ , attribute_list ] :: ] entity_list
```

```
COMPLEX * length_value [ [, attribute_list ] :: ] entity_list
```

For *kind_param* values, see Section 4.3.3, page 83. The *length_value* values correspond to the *kind_param* values in the following manner:

kind_param	***length_value***
4 (default)	8 (default)
8	16
16	32

The following examples show entity-oriented declaration statements:

```
COMPLEX(KIND = LOW), POINTER :: ROOTS(:)
COMPLEX, POINTER :: DISCRIMINANT, COEFFICIENTS(:)
```

The following examples show attribute-oriented declaration statements:

```
COMPLEX(KIND = LOW) :: ROOTS(:)
COMPLEX :: DISCRIMINANT, COEFFICIENTS(:)
```

```
POINTER :: ROOTS, DISCRIMINANT, COEFFICIENTS
```

ANSI/ISO: The Fortran standard does not specify the `COMPLEX*`*length_value* syntax. It is recommended that this syntax not be used in any new code.

5.1.5 Logical

A `LOGICAL` statement declares the names of entities to be of type logical. If a kind selector is present, it specifies the representation method. For more information on logical type, see Section 4.3.4, page 85.

The compiler supports the following formats for declaring objects of this type:

```
LOGICAL [ ([ KIND = ] kind_param)] [ [, attribute_list ] :: ] entity_list

LOGICAL * length_value [ [, attribute_list ] :: ] entity_list
```

For *kind_param* values, see Section 4.3.4, page 85. The *length_value* values correspond to the *kind_param* values in the following manner:

kind_param	***length_value***
1	1
2	2
4 (default)	4 (default)
8	8

The following examples show entity-oriented declaration statements:

```
LOGICAL, ALLOCATABLE :: MASK_1(:), MASK_2(:)
LOGICAL(KIND = WORD), SAVE :: INDICATOR, STATUS
```

The following examples show attribute-oriented declaration statements:

```
LOGICAL MASK_1(:), MASK_2(:)
LOGICAL (KIND = WORD) INDICATOR, STATUS
ALLOCATABLE MASK_1, MASK_2
SAVE INDICATOR, STATUS
```

ANSI/ISO: The Fortran standard does not specify the `LOGICAL*`*length_value* syntax. It is recommended that this syntax not be used in any new code.

5.1.6 Character

A `CHARACTER` statement declares the names of entities to be of type character. For more information on character type, see Section 4.3.5, page 87.

The following is a format for declaring objects of this type:

```
CHARACTER [ char_selector ] [ [ , attribute_list ] :: ] entity_list
```

The components of this format are defined as follows:

	char_selector	**is**	*length_selector*
		or	`(LEN =` *type_param_value*`, KIND =` *kind_value*`)`
		or	`(`*type_param_value*`, [ KIND = ]` *kind_value*`)`
		or	`(KIND =` *kind_value* `[, LEN =` *type_param_value* `])`
	length_selector	**is**	`([ LEN = ]` *type_param_value*`)`
OBS		**or**	`*` *char_length* `[ , ]`
	char_length	**is**	`(`*type_param_value*`)`
		or	*scalar_int_literal_constant*
	type_param_value	**is**	*specification_expr*
		or	`*`

The optional comma in a *length_selector* is permitted only if no double colon separator appears in the type declaration statement.

A character type declaration can specify a character length that is a nonconstant expression if it appears in a procedure or a procedure interface if it is not a component declaration in a derived-type definition. The length is determined on entry into the procedure and is not affected by any changes in the values of variables

in the expression during the execution of the procedure. A character object declared this way that is not a dummy argument is called an *automatic data object*.

The length of a named character entity or a character component in a type definition is specified by the character selector in the type specification unless there is a character length in an entity or component declaration; if so, the character length specifies an individual length and overrides the length in the character selector. If a length is not specified in either a character selector or a character length, the length is 1.

If the length parameter has a negative value, the length of the character entity is 0.

If a scalar integer literal constant is used to specify a character length, it must not include a kind parameter. (This could produce an ambiguity when fixed source form is used.)

A length parameter value of * can be used only in the following ways:

- To declare a dummy argument of a procedure, in which case the dummy argument assumes the length of the associated actual argument when the procedure is invoked.
- To declare a named constant, in which case the length is that of the constant value.
- To declare the result variable for an external function. Any scoping unit that invokes the function must declare the function with a length other than *, or it must access such a declaration by host or use association. When the function is invoked, the length of the result is the value specified in the declaration in the program unit referencing the function. Note that an implication of this rule is that a length of * must not appear in an `IMPLICIT` statement.
- To declare a character pointee.

A function name must not be declared with a length of * if the function is an internal or module function; or if it is array-valued, pointer-valued, or recursive; or if it is a `PURE` function.

An interface body can be specified for a dummy or external function whose result is of type `CHAR*(*)` only if the function is not invoked. This is because the characteristics must match in both places.

The length of a character-valued statement function or statement function dummy argument of type character must be an integer constant expression.

The following examples show entity-oriented character type declaration statements:

```
CHARACTER(LEN = 10, KIND = ASCII), SAVE :: GREETING(2)
CHARACTER(10) :: PROMPT = "PASSWORD?"
CHARACTER(*), INTENT(IN) :: HOME_TEAM, VISITORS
CHARACTER*(3), SAVE :: NORMAL_1, LONGER(9)*20, NORMAL_2
CHARACTER :: GRADE = "A"
```

The following examples show attribute-oriented character type declaration statements:

```
CHARACTER(LEN = 10, KIND = ASCII) :: GREETING
CHARACTER(10) :: PROMPT
CHARACTER(*) :: HOME_TEAM, VISITORS
CHARACTER*(3) :: NORMAL_1, LONGER*20, NORMAL_2
CHARACTER GRADE
SAVE :: GREETING, NORMAL_1, LONGER, NORMAL_2
DIMENSION GREETING(2), LONGER(9)
INTENT(IN) :: HOME_TEAM, VISITORS
DATA PROMPT / "PASSWORD?" /, GRADE / "A" /
```

5.1.7 Derived Type

A TYPE declaration statement declares the names of entities to be of the specified user-defined type. The type name appears in parentheses following the keyword TYPE. For more information on derived types, see Section 4.4, page 96.

The following is a format for declaring objects of user-defined type:

```
TYPE (type_name) [ [ , attribute_list ] :: ] entity_list
```

The following examples show entity-oriented derived type declaration statements:

```
TYPE(COLOR), DIMENSION(:), ALLOCATABLE :: HUES_OF_RED
TYPE(PERSON), SAVE :: CAR_POOL(3)
TYPE(PARAGRAPH), SAVE :: OVERVIEW, SUBSTANCE, SUMMARY
```

The following examples show attribute-oriented derived type declaration statements:

```
TYPE(COLOR) :: HUES_OF_RED
TYPE(PERSON) :: CAR_POOL(3)
TYPE(PARAGRAPH) :: OVERVIEW, SUBSTANCE, SUMMARY
DIMENSION :: HUES_OF_RED(:)
```

```
ALLOCATABLE :: HUES_OF_RED
SAVE :: CAR_POOL, OVERVIEW, SUBSTANCE, SUMMARY
```

An object of derived type (a structure) must not have the PUBLIC attribute if its type is private.

A structure constructor must be used to initialize an object of derived type. Each component of the structure constructor must be an initialization expression. For more information on structure constructors, see Section 4.5, page 108.

Note: Variables declared to be Cray pointers and pointees through the Cray POINTER statement cannot be declared as components of derived types.

5.1.8 Cray Pointer (EXTENSION)

The Cray POINTER statement declares one variable to be a Cray pointer (that is, to have the Cray pointer data type) and another variable to be its pointee; that is, the value of the Cray pointer is the address of the pointee. This statement has the following format:

```
POINTER (pointer_name, pointee_name [ (array_spec) ])
  [, (pointer_name, pointee_name [ (array_spec) ])] ...
```

pointer_name — Pointer to the corresponding *pointee_name*. *pointer_name* contains the address of *pointee_name*. Only a scalar variable can be declared type Cray pointer; constants, arrays, statement functions, and external functions cannot.

pointee_name — Pointee of corresponding *pointer_name*. Must be a variable name, array declarator, or array name. The value of *pointer_name* is used as the address for any reference to *pointee_name*; therefore, *pointee_name* is not assigned storage. If *pointee_name* is an array declarator, it can be explicit-shape (with either constant or nonconstant bounds) or assumed-size.

array_spec — If present, this must be either an *explicit_shape_spec_list*, with either constant or nonconstant bounds) or an *assumed_size_spec*.

Example:

```
POINTER(P,B),(Q,C)
```

This statement declares Cray pointer `P` and its pointee `B`, and Cray pointer `Q` and pointee `C`; the pointer's current value is used as the address of the pointee whenever the pointee is referenced.

An array that is named as a pointee in a Cray `POINTER` statement is a pointee array. Its array declarator can appear in a separate type or `DIMENSION` statement or in the pointer list itself. In a subprogram, the dimension declarator can contain references to variables in a common block or to dummy arguments. As with nonconstant bound array arguments to subprograms, the size of each dimension is evaluated on entrance to the subprogram, not when the pointee is referenced. For example:

```
POINTER(IX, X(N,0:M))
```

ANSI/ISO: The Fortran standard does not specify the Cray `POINTER` statement or data type. Variables declared to be pointers and pointees through the Cray `POINTER` statement cannot be declared as components of derived types.

In addition, pointees must not be deferred-shape or assumed-shape arrays. An assumed-size pointee array is not allowed in a main program unit.

5.2 Implicit Typing

Each variable, named constant, and function has a type and a name. If the type is not declared explicitly, it is assumed from the first letter of the name. This method of determining type is called *implicit typing*. In each scoping unit, there is in effect a mapping of each of the letters `A, B, ..., Z` (and corresponding lowercase letters) to one of the accessible types or to no type. `IMPLICIT` statements in a scoping unit can be used to specify a mapping different from the default mapping. If a new mapping for a letter is not specified in an `IMPLICIT` statement, the default mapping continues to apply for that letter.

An `IMPLICIT NONE` statement specifies that there is no mapping for any letter and thus all variables, named constants, and functions must be declared in type

declaration statements. If the host of a scoping unit contains the `IMPLICIT NONE` statement and the scoping unit contains `IMPLICIT` statements for some letters, the other letters retain the null mapping. This is the only situation in which some initial letters specify an implied type and other initial letters require explicit declarations.

A program unit is treated as if it had a host with the mapping shown in Figure 5-1. That is, each undeclared variable or function whose name begins with any of the letters `I`, `J`, `K`, `L`, `M`, or `N` is of type integer and all others are of type real.

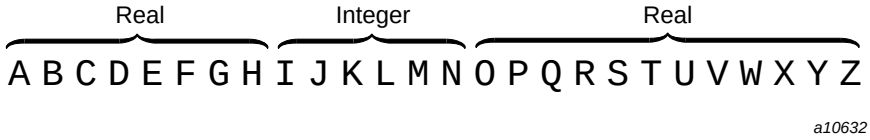

Figure 5-1 Default implicit mapping for a program unit

The `IMPLICIT` statement is defined as follows:

	implicit_stmt	**is**	`IMPLICIT` *implicit_spec_list*
		or	`IMPLICIT NONE`
EXT		**or**	`IMPLICIT UNDEFINED`
	implicit_spec	**is**	*type_spec* (*letter_spec_list*)
	letter_spec	**is**	*letter* [- *letter*]

If `IMPLICIT NONE` appears, it must precede any `PARAMETER` statements and there must be no other `IMPLICIT` statements in the scoping unit.

If the - *letter* option appears in a letter specification, the second letter must follow the first alphabetically.

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 scoping unit.

ANSI/ISO: The Fortran standard does not include the `IMPLICIT UNDEFINED` syntax. `IMPLICIT UNDEFINED` is equivalent to `IMPLICIT NONE`, but it is recommended that the `IMPLICIT UNDEFINED` syntax not be used in new code.

An `IMPLICIT` statement can be used to specify implicit mappings for user-defined types as well as for intrinsic types.

The `IMPLICIT` statement specifies that all variables, named constants, and functions beginning with the indicated letters are assigned the indicated data type (and type parameters) implicitly. For example, consider the following statement:

```
IMPLICIT COMPLEX (A-C, Z)
```

In this statement, all undeclared variables, named constants, and functions beginning with the letters `A`, `B`, `C`, and `Z` are of type default complex. If this is the only `IMPLICIT` statement, undeclared variables, named constants, and functions beginning with `I` through `N` will still be of type integer; undeclared variables, named constants, and functions beginning with `D` through `H` and `O` through `Y` will be of type real.

As another example, consider the following statement:

```
IMPLICIT NONE
```

In this statement, there is no implicit typing in the scoping unit. Each variable and named constant local to the scoping unit, and each external function used in the scoping unit, must be declared explicitly in a type statement. This statement is useful for detecting inadvertent misspellings in a program because misspelled names become undeclared rather than implicitly declared.

The following examples show `IMPLICIT` statements:

```
IMPLICIT INTEGER (A-G), LOGICAL(KIND = WORD) (M)
IMPLICIT CHARACTER*(10) (P, Q)
IMPLICIT TYPE(COLOR) (X-Z)
```

The additional complexity that implicit typing causes in determining the scope of an undeclared variable in a nested scope is explained in the *MIPSpro Fortran Language Reference Manual, Volume 2*.

5.3 Array Properties

An array object has the DIMENSION attribute. An array specification determines the array's rank, or number of dimensions. The extents of the dimensions may be declared or left unspecified. If they are left unspecified, the array must also have the ALLOCATABLE or POINTER attribute, or it must be a dummy argument.

5.3.1 Array Specifications

There are four formats that an *array_spec* can take:

array_spec	**is**	*explicit_shape_spec_list*
	or	*assumed_shape_spec_list*
	or	*deferred_shape_spec_list*
	or	*assumed_size_spec*

The maximum rank of an array is 7. A scalar is considered to have rank 0.

An array with a *deferred_shape_spec_list* must have the POINTER or ALLOCATABLE attribute.

An array with an *assumed_shape_spec_list* or an *assumed_size_spec* must be a dummy argument.

5.3.1.1 Explicit-shape Arrays

An explicit-shape array has bounds specified in each dimension. Each dimension is specified by an *explicit_shape_spec*, which is defined as follows:

explicit_shape_spec	**is**	*lower_bound* **:** *upper_bound*
lower_bound	**is**	*specification_expr*
upper_bound	**is**	*specification_expr*

For more information on *specification_expr*s, see Section 7.2.9.3, page 254.

The number of sets of bounds specified is the number of dimensions (rank) of the array.

If the lower bound is omitted, the default value is 1.

The value of a lower bound or an upper bound can be positive, negative, or 0.

The subscript range of the array in a dimension is the set of integer values between and including the lower and upper bounds, provided the upper bound is not less than the lower bound. If the upper bound is less than the lower bound, the range is empty, the extent in that dimension is 0, and the size of the array is 0.

The expression for a bound could involve variables that cause the expression to have different values each time the procedure in which it is declared is executed. If so, the array must be a dummy argument, a function result, or an automatic array, in which case the actual bounds are determined when the procedure is entered. The bounds of such an array are unaffected by any redefinition or undefinition of the specification variables during the execution of the procedure.

The following are examples of entity-oriented explicit-shape array declarations:

```
REAL Q(-10:10, -10:10, 2)  ! in a main program

SUBROUTINE EX1(Z, I, J)             ! in a
  REAL, DIMENSION(2:I + 1, J) :: Z  ! subroutine
```

The following are examples of attribute-oriented explicit-shape array declarations:

```
REAL Q
DIMENSION Q(-10:10, -10:10, 2)  ! in a main program

SUBROUTINE EX1(Z, I, J)    ! in
   REAL Z                  ! a
   DIMENSION Z(2:I + 1, J) ! subroutine
```

5.3.1.2 Assumed-shape Arrays

An assumed-shape array is a dummy argument that takes the shape of the actual argument passed to it. An *assumed_shape_spec* has the following format:

assumed_shape_spec	**is**	[*lower_bound*] :	

The *lower_bound* of the assumed-shape array is the specified lower bound, if present; otherwise it is 1.

The rank is equal to the number of colons in the *assumed_shape_spec_list*.

The upper bound is the extent of the corresponding dimension of the associated array plus the lower bound minus 1.

An assumed-shape array must not have the POINTER or ALLOCATABLE attribute.

The following example shows an entity-oriented, assumed-shape array declaration:

```
REAL, DIMENSION(2:, :) :: X
```

The following example shows an attribute-oriented, assume-shaped array declaration:

```
SUBROUTINE EX2(A, B, X)
   REAL A(:), B(0:), X
   DIMENSION X(2:, :)
   INTENT(IN) A, B
      . . .
```

As another example, assume that EX2 is called by the following statement:

```
CALL EX2(U, V, W(4:9, 2:6))
```

Dummy argument X is an array with bounds (2:7, 1:5). The lower bound of the first dimension is 2 because X is declared to have a lower bound of 2. The upper bound is 7 because the dummy argument takes its shape from the actual argument W.

5.3.1.3 Deferred-shape Arrays

A deferred-shape array is either an array pointer or an allocatable array. An array pointer is an array that has the POINTER attribute. Its extent in each dimension is determined when the pointer is allocated or when a pointer assignment statement for the pointer is executed. An allocatable array is an array that has the ALLOCATABLE attribute. Its bounds, and thus its shape, are determined when the array is allocated. In both cases a colon specifies the declared bound; that is, the format of a *deferred_shape_spec* is defined as follows:

deferred_shape_spec	**is**	:

The rank is equal to the number of colons in the *deferred_shape_spec_list*.

The bounds of an allocatable array are specified in an ALLOCATE statement when the array is allocated.

The lower bound of each dimension of an array pointer is the result of the LBOUND(3i) intrinsic function applied to the corresponding dimension of the target. The upper bound of each dimension is the result of the UBOUND(3i) intrinsic function applied to the corresponding dimension of the target. This means that if the bounds are determined by allocation of the pointer, you can specify them. If the bounds are determined by pointer assignment, there are two possible interpretations:

- If the pointer target is a named whole array, the bounds are those declared in the array declaration or those specified when the array was allocated.
- If the pointer target is an array section, the lower bound is 1 and the upper bound is the extent in that dimension.

The bounds and shape of an array pointer or allocatable array are unaffected by any subsequent redefinition or undefinition of variables involved in determination of the bounds.

The following examples show entity-oriented, deferred-shape array declarations:

```
REAL, POINTER :: D(:, :), P(:) ! array pointers
REAL, ALLOCATABLE :: E(:)      ! allocatable array
```

The following examples show attribute-oriented, deferred-shaped array declarations:

```
REAL D
DIMENSION D(:, :), P(:), E(:)
POINTER D, P
ALLOCATABLE E
```

5.3.1.4 Assumed-size Arrays

An assumed-size array is a dummy argument array whose size is assumed from that of the associated actual argument. Only the size is assumed. The rank, extents, and bounds (except for the upper bound and extent in the last dimension) are determined by the declaration of the dummy array. The rules for argument association between an actual argument and an assumed-size array are as follows:

- They must have the same initial array element.

- Successive array elements are storage associated. For information on storage association, see Section 5.10, page 170.
- Declarations for the dummy argument determine the rank. They also determine lower bounds for all dimensions and the extents and upper bounds for all dimensions except the last.
- The size of the actual argument determines the size of the dummy argument as explained in this section.

The format of an *assumed_size_spec* is defined as follows:

assumed_size_spec **is** [*explicit_shape_spec_list*,] [*lower_bound* :] *

The rank of an assumed-size array is the number of explicit-shape specifications plus one.

The size of an assumed-size array is determined as follows:

- If the actual argument associated with the assumed-size dummy argument is an array of any type other than character, the size is that of the actual array.
- If the actual argument associated with the assumed-size dummy array is an array element of any type other than character with a subscript order value of *v* in an array of size *x*, the size of the dummy argument is *x* - *v* +1. For information on array element order, see Section 6.4.7, page 199.
- If the actual argument is a character array, character array element, or a character array element substring, and if it begins at character storage unit *t* of an array with *c* character storage units, the size of the dummy array is MAX(INT((*c* - *t* +1)/ *e*), 0) where *e* is the length of an element in the dummy character array.

If *r* is the rank of the array, the bounds of the first *r*-1 dimensions are those specified by the explicit-shape specification list, if present. The lower bound of the last dimension is the specified lower bound, if present; otherwise it is 1.

The expression for a bound may involve variables that cause the expression to have different values each time the procedure in which it is declared is executed. If so, the bounds are unaffected by any subsequent redefinition or undefinition of such variables involved in the determination of the bounds.

A function result must not be an assumed-size array.

An assumed-size array must not appear in a context where the shape of the array is required, such as a whole array reference.

The following examples show entity-oriented, assumed-size array declarations:

```
SUBROUTINE EX3(N, S, Y)
   REAL, DIMENSION(N, *) :: S
   REAL Y(10, 5, *)
      . . .
```

The following examples show attribute-oriented, assumed size array declarations:

```
SUBROUTINE EX3(N, S, Y)
   REAL S, Y
   DIMENSION S(N, *), Y(10, 5, *)
      . . .
```

5.3.2 DIMENSION Attribute and Statement

The dimensions of an array can be specified by the appearance of a DIMENSION attribute or by the appearance of an array specification following the name of the array in a type declaration statement. Both a DIMENSION attribute and an array specification following the name can appear in a declaration statement. In this case, the array specification following the name overrides the array specification following the DIMENSION attribute. A format for a type declaration statement with a DIMENSION attribute is as follows:

type, DIMENSION (*array_spec*) [, *attribute_list*] :: *entity_decl_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

ALLOCATABLE
AUTOMATIC (EXTENSION)
INTENT
OPTIONAL
POINTER
PARAMETER
PRIVATE
PUBLIC
SAVE

TARGET
VOLATILE (EXTENSION)

The type declaration can also contain an *initialization_expr*. An array specification can also appear following a name in several different kinds of statements to declare an array. They are DIMENSION, ALLOCATABLE, POINTER, TARGET, and COMMON statements.

The DIMENSION statement is the statement form of the DIMENSION attribute and is defined as follows:

dimension_stmt	**is**	DIMENSION [::] *array_name* (*array_spec*) [, *array_name* (*array_spec*)] ...

The DIMENSION statement also confers the DIMENSION attribute. It is subject to the same rules and restrictions as the DIMENSION attribute.

The following examples show entity-oriented declarations:

```
INTEGER, DIMENSION(10), TARGET, SAVE :: INDICES
INTEGER, ALLOCATABLE, TARGET :: LG(:, :, :)
```

The following examples show attribute-oriented declarations:

```
INTEGER INDICES, LG(:, :, :)
DIMENSION INDICES(10)
TARGET INDICES, LG
ALLOCATABLE LG
SAVE INDICES
```

The following examples show the array specification in other statements:

```
INTEGER INDICES, LG
TARGET INDICES(10), LG
ALLOCATABLE LG(:, :, :)
SAVE INDICES
```

The following example shows the array specification in a COMMON statement:

```
COMMON /UNIVERSAL/ TIME(80), SPACE(20, 20, 20, 20)
```

5.3.3 ALLOCATABLE Attribute and Statement

Arrays are the only objects that can have the ALLOCATABLE attribute. An allocatable array is one for which the bounds are determined when an ALLOCATE statement is executed for the array. These arrays must be deferred-shape arrays. The following is a format for a type declaration statement with an ALLOCATABLE attribute:

```
type, ALLOCATABLE [, attribute_list ] :: entity_decl_list
```

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

DIMENSION (with deferred shape)
PRIVATE
PUBLIC
SAVE
TARGET
VOLATILE (EXTENSION)

The format of the ALLOCATABLE statement is defined as follows:

```
allocatable_stmt    is  ALLOCATABLE [::] array_name [ (deferred_shape_spec_list)]
                            [ ,array_name [ (deferred_shape_spec_list) ] ] ...
```

The array must not be a dummy argument or function result.

If the array is given the DIMENSION attribute elsewhere, the bounds must be specified as colons (deferred shape).

The ALLOCATABLE statement also confers the ALLOCATABLE attribute. It is subject to the same rules and restrictions as the ALLOCATABLE attribute.

The following examples show entity-oriented declarations:

```
REAL, ALLOCATABLE :: A(:, :)
LOGICAL, ALLOCATABLE, DIMENSION(:) :: MASK1
```

The following examples show attribute-oriented declarations:

```
REAL A(:, :)
LOGICAL MASK1
```

```
DIMENSION MASK1(:)
ALLOCATABLE A, MASK1
```

5.4 POINTER Properties

Most attributes, when applied to an object, add characteristics that the object would not have otherwise. The POINTER attribute, in some sense, removes a characteristic that an object has. Ordinarily, an object has storage space set aside. If the object has the POINTER attribute, it has no space initially and must not be referenced until space is associated with it. An ALLOCATE statement creates new space for a pointer object. A pointer assignment statement permits the pointer to borrow the space from another object. The space that becomes associated with a pointer is called the pointer's *target*. The target can change during the execution of a program. A pointer target is either an object or part of an object declared to have the TARGET attribute; or it is an object or part of an object that was created by the allocation of a pointer. A pointer can be assigned the target (or part of the target) of another pointer.

Another way of thinking about a pointer is as a *descriptor* that contains information about the type, type parameters, rank, extents, and location of the pointer's target. Thus, a pointer to a scalar object of type real is different from a pointer to an array of user-defined type.

5.4.1 POINTER Attribute and Statement

The following is a format for a type declaration statement with a POINTER attribute:

type, POINTER [, *attribute_list*] :: *entity_decl_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

AUTOMATIC (EXTENSION)
DIMENSION (with deferred shape)
OPTIONAL
PRIVATE
PUBLIC
SAVE
VOLATILE (EXTENSION)

The POINTER statement also provides a means for declaring pointers. Its format is defined as follows:

pointer_stmt	**is** POINTER [::] *object_name* [(*deferred_shape_spec_list*)] [, *object_name* [(*deferred_shape_spec_list*)]] ...

The target of a pointer can be a scalar or an array.

An array pointer must be declared as a deferred-shape array.

A pointer must not be referenced or defined unless it is associated with a target that can be referenced or defined. (A pointer on the right-hand side of a pointer assignment is not considered a pointer reference.)

The POINTER statement also confers the POINTER attribute. It is subject to the same rules and restrictions as the POINTER attribute.

The following example shows an entity-oriented declaration:

```
TYPE(NODE), POINTER :: CURRENT
REAL, POINTER :: X(:, :), Y(:)
```

The following example shows an attribute-oriented declaration:

```
TYPE(NODE) CURRENT
REAL X(:, :), Y(:)
POINTER CURRENT, X, Y
```

5.4.2 TARGET Attribute and Statement

Only an object with the TARGET attribute can become the target of a pointer during execution of a program. If an object does not have the TARGET attribute or has not been allocated, no part of it can be accessed through a pointer. The following is a format for a type declaration statement with a TARGET attribute:

type, TARGET [, *attribute_list*] :: *entity_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

ALLOCATABLE
AUTOMATIC (EXTENSION)
DIMENSION
INTENT
OPTIONAL
PRIVATE
PUBLIC
SAVE
VOLATILE (EXTENSION)

The type declaration statement can also contain an *initialization_expr*.

The TARGET statement also provides a means for specifying pointer targets. It has the following format:

target_stmt **is** TARGET [::] *object_name* [(*array_spec*)]
[, *object_name* [(*array_spec*)]] ...

The TARGET statement also confers the TARGET attribute. It is subject to the same rules and restrictions as the TARGET attribute.

The following examples show entity-oriented declarations:

```
TYPE(NODE), TARGET :: HEAD_OF_LIST
REAL, TARGET, DIMENSION(100, 100) :: V, W(100)
```

The following examples show attribute-oriented declarations:

```
TYPE(NODE) HEAD_OF_LIST
REAL V, W(100)
DIMENSION V(100, 100)
TARGET HEAD_OF_LIST, V, W
```

5.4.3 AUTOMATIC Attribute and Statement (EXTENSION)

The AUTOMATIC attribute specifies stack-based storage for a variable or array. Such variables and arrays are undefined upon entering and exiting the procedure. The following is the format for the AUTOMATIC specification:

type, AUTOMATIC [, *attribute_list*] :: *entity_list*

attribute_list For *attribute_list*, specify a variable name or an array declarator. If an *attribute_list* item is an array, it must be declared with an *explicit_shape_spec* with constant bounds. If an *attribute_list* item is a pointer, it must be declared with a *deferred_shape_spec*.

If an *attribute_list* item has the same name as the function in which it is declared, the *attribute_list* item must be scalar and of type integer, real, logical, complex, or double precision.

If the *attribute_list* item is a pointer, the `AUTOMATIC` attribute applies to the pointer itself and not to any target that may become associated with the pointer.

Subject to the rules governing combinations of attributes, *attribute_list* can contain the following:

`DIMENSION`
`TARGET`
`POINTER`
`VOLATILE` (EXTENSION)

The following entities cannot have the `AUTOMATIC` attribute:

- Pointers or arrays used as function results
- Dummy arguments
- Statement functions
- Automatic array or character data objects

An *attribute_list* item cannot have the following characteristics:

- It cannot be defined in the scoping unit of a module.
- It cannot be a common block item.
- It cannot be specified more than once within the same scoping unit.
- It cannot be initialized with a `DATA` statement or with a type declaration statement.
- It cannot also have the `SAVE` attribute.
- It cannot be specified as a Cray pointee.
- It cannot be specified on an object that appears in an `AUXILIARY` or `SYMMETRIC` compiler directive.

ANSI/ISO: The Fortran standard does not specify the `AUTOMATIC` attribute or statement, nor does it provide a means to explicitly declare automatic variables as automatic. The Fortran standard does not specify compiler directives.

5.5 Data Initialization and the `DATA` Statement

An entity can be initialized in a type declaration statement. When an initialization expression appears in a declaration for an object that does not have the `PARAMETER` attribute, the object (which is a variable) is assigned the specified initial value. This object is a variable with *explicit initialization*. Alternatively, explicit initialization can be specified in a `DATA` statement unless the variable is of a derived type for which default initialization has been specified.

Note: The Fortran standard has declared that the placement of `DATA` statements amongst executable statements is obsolescent.

The same rules apply to the assignment of the initial value as apply when an assignment statement is executed. For example, if the variable is of type real but the value is an integer value, the variable will be assigned the real equivalent of the integer value. If the kind of the variable is different from the kind of the value, the value will be converted to the kind of the variable. Array constructors and broadcast values can be used to initialize arrays, and structure constructors can be used to initialize variables of user-defined type. The format of a type declaration statement that provides an initial value for a variable is as follows:

```
type [, attribute_list ] :: object_name [ (array_spec) ]
   [ * char_length ] = initialization_expr
```

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
DIMENSION
POINTER
PRIVATE
PUBLIC
SAVE
```

TARGET
VOLATILE (EXTENSION)

For example:

```
INTEGER :: I = 0
```

The PARAMETER attribute can also appear in the *attribute_list*, but in this case, the object is declared to be a named constant.

The value associated with the name cannot be changed during the execution of the program. For example, PI or E can be associated with the familiar mathematical constants to provide more convenient access to these values. Named constants are also used to assign names to values (such as a sales tax rate) that could change at some later time. When a change is necessary, it can be made at one place in the program rather than every place where the value is used. The program can be recompiled to effect the change.

An array name that appears in a declaration statement that contains an *initialization_expr* must have its dimensionality declared in the same statement or a previous statement.

Initialization of a variable in a type declaration statement or any part of a variable in a DATA statement implies that the variable has the SAVE attribute unless the variable is in a named common block. The automatically acquired SAVE attribute may be reaffirmed by the appearance of SAVE as an attribute in its type declaration statement or by inclusion of the variable name in a separate SAVE statement.

The DATA statement is defined as follows:

data_stmt	**is**	DATA *data_stmt_set* [[,] *data_stmt_set*]...
data_stmt_set	**is**	*data_stmt_object_list* / *data_stmt_value_list* / [[,] *data_stmt_object_list* / *data_stmt_value_list* /] ...
data_stmt_object	**is**	*variable*
	or	*data_implied_do*
data_stmt_value	**is**	[*data_stmt_repeat* *] *data_stmt_constant*
data_stmt_constant	**is**	*scalar_constant*
	or	*scalar_constant_subobject*

		or	*signed_int_literal_constant*
		or	*signed_real_literal_constant*
		or	*structure_constructor*
		or	*boz_literal_constant*
		or	`NULL()`
EXT		**or**	*typeless_constant*
	data_stmt_repeat	**is**	*scalar_int_constant*
		or	*scalar_int_constant_subobject*
	data_implied_do	**is**	(*data_i_do_object_list*, *data_i_do_variable*= *scalar_int_expr*, *scalar_int_expr*[, *scalar_int_expr*])
	data_i_do_object	**is**	*array_element*
		or	*scalar_structure_component*
		or	*data_implied_do*
	data_i_do_variable	**is**	*scalar_int_variable*
EXT	*typeless_constant*	**is**	*octal_typeless_constant*
		or	*hexadecimal_typeless_constant*
		or	*binary_typeless_constant*
EXT	*octal_typeless_constant*	**is**	*digit* [*digit*`...`]`B`
		or	`O"` *digit* [*digit*`...`]`"`
		or	`O'` *digit* [*digit*`...`]`'`
		or	`"` *digit* [*digit*`...`]`"O`
		or	`'` *digit* [*digit*`...`]`'O`
EXT	*hexadecimal_typeless_constant*	**is**	`X'`*hex_digit* [*hex_digit*`...`]
		or	`'X"` *hex_digit* [*hex_digit*`...`]`"`
		or	`'`*hex_digit* [*hex_digit*`...`]`'X`
		or	`"`*hex_digit* [*hex_digit*`...`]`"X`
		or	`Z'`*hex_digit* [*hex_digit*`...`]`'`
		or	`Z"`*hex_digit* [*hex_digit*`...`]`"`

EXT	*binary_typeless_constant*	**is**	B'*bin_digit* [*bin_digit*...]
		or	'B"*bin_digit* [*bin_digit*...] "

The following notes pertain to the preceding format:

- *digit* must have one of the values 0 through 7 in *octal_typeless_constant*
- *digit* must have a value of 0 or 1 in *binary_typeless_constant*
- The B, O, X, and Z characters can be in uppercase or lowercase.
- The *scalar_structure_component* must contain at least one *part_ref* that contains a *subscript_list*.

Note that a constant value cannot be an array constructor. An array can be initialized by using the array name as the *data_stmt_object* and supplying values for all the elements of the array using a *data_implied_do*.

ANSI/ISO: The Fortran standard does not specify the *typeless_constant*. If an object is of type character or logical, the constant used for initialization must be of the same type.

If an object is of type real or complex, the corresponding constant must be of type integer, real, or complex. The compiler permits a default real object to be initialized with a BOZ typeless, BOZ, typeless, or character (used as Hollerith) constant. No conversion of the BOZ value, typeless value, or character constant takes place.

The compiler permits an integer object to be initialized with a BOZ, typeless, or character (used as Hollerith) constant in a type declaration statement. The compiler also allows an integer object to be initialized with a typeless or character (used as Hollerith) constant in a DATA statement.

ANSI/ISO: The Fortran standard does not specify typeless or character (used as Hollerith) constants in initializations, nor does it allow BOZ constants to be used in type declaration statement initializations.

If an object is of derived type, the corresponding structure constructor must be of the same type.

The value of the constant, structure constructor (in a DATA statement), or initialization expression (in a type declaration statement) must be such that its value could be

assigned to the corresponding variable using an intrinsic assignment statement. The variable becomes initially defined with the value of the constant.

A variable, or the same part of a variable, must not be initialized more than once in an executable program.

Note: If a variable is initialized more than once in a program, the order of initialization is not guaranteed. The compiler cannot enforce and does not adhere to an order for initialization when multiple initializations appear in source code. The load order can also affect the value of a variable that is initialized multiple times, which means that the final value can vary from loader to loader. Such code does not necessarily port from platform to platform.

An object declared to be of a derived-type that has default initialization cannot be specified in a `DATA` statement. This object can be initialized in a type specification statement. The initialization in the type specification statement overrides the default initialization.

The following items cannot be initialized:

- A dummy argument
- An object made accessible by use or host association
- A function result
- An automatic object
- An allocatable array
- An external or intrinsic procedure

ANSI/ISO: The Fortran standard does not allow initialization of objects in named common blocks except from within a `BLOCKDATA` program unit. The Fortran standard does not allow initialization of objects in a blank common block.

For an object being initialized, any subscript, section subscript, substring starting point, or substring ending point must be an integer initialization expression.

Each component of a structure constructor used for initialization must be an initialization expression.

If the variable being initialized has the POINTER attribute, then *data_stmt_constant* must be NULL(). The pointer has an initial association status of disassociated.

A variable that appears in a DATA statement and is thereby declared and typed implicitly can appear in a subsequent type declaration statement only if that declaration confirms the implicit declaration. An array name, array section, or array element appearing in a DATA statement must have had its array properties established previously.

If a DATA statement constant value is a named constant or a structure constructor, the named constant or derived type must have been declared previously in the scoping unit or must have been made accessible by USE or HOST association.

An array element or structure component that appears in a DATA statement must not have a constant parent.

The DATA statement repeat factor value must be positive or zero. If it is a named constant, the value must have been specified in a prior statement in the scoping unit that contains the DATA statement or must have been made accessible by use or host association.

In a *scalar_constant_subobject* that is a *data_stmt_repeat*, any subscript must be an initialization expression.

In a *scalar_constant_subobject* that is a *data_stmt_constant*, any subscript, substring starting point, or substring ending point must be an initialization expression.

A subscript in an array element of an implied-DO list must contain as operands only constants or DO variables of the containing implied-DO s.

The scalar integer loop control expressions in an implied-DO must contain as operands only constants or DO variables of the containing implied-DO s. Each operation must be an intrinsic operation.

The data object list is expanded to form a sequence of scalar variables. An array or array section is equivalent to the sequence of its array elements in array element order. A *data_implied_do* is expanded to form a sequence of array elements and structure components, under the control of the implied-DO variable, as in the DO construct. A zero-sized array or an implied-DO with an iteration count of 0 contributes no variables to the expanded list, but a character variable declared to have zero length does contribute a variable to the list.

The data value list is expanded to form a sequence of scalar values. Each value must be a constant or constant expression (structure constructor). If a value is represented

by a named constant, the named constant must be specified prior to the DATA statement. A DATA statement repeat factor indicates the number of times the following constant value is to be included in the sequence. If the repeat factor is 0, the following value is ignored.

If a *data_stmt_constant* is a *boz_literal_constant*, the corresponding object must be of type integer. A *data_stmt_constant* that is a *boz_literal_constant* is treated as if the constant were an *int_literal_constant* with a *kind_param* that specified the representation method with the largest decimal exponent range supported.

Scalar variables and values of the expanded sequence must be in one-to-one correspondence. Each value specifies the initial value for the corresponding variable. The lengths of the two expanded sequences must be the same.

ANSI/ISO: If the last item in the *data_object_list* is an array name, the value list can contain fewer values than the number of elements in the array. Any element that is not assigned a value is undefined.

The following examples show type declaration statement initializations:

```
CHARACTER(LEN = 10) :: NAME = "JOHN DOE"
INTEGER, DIMENSION(0:9) :: METERS = (/ (0, I = 1, 10) /)
TYPE(PERSON) :: ME = PERSON(21, "JOHN SMITH"), &
   YOU = PERSON(35, "FRED BROWN")
INTEGER :: BIRD(3) = 1
REAL :: SKEW(100,100) = RESHAPE ( (/((1.0, K = 1, J-1), &
       (0.0, K = J, 100), J = 1, 100)/), (/ 100, 100 /) )
```

The following are examples of DATA statement initializations:

```
CHARACTER*10 NAME
INTEGER METERS(0:9)
DATA NAME /"JOHN DOE"/, METERS /10*0/

TYPE(PERSON) ME, YOU
DATA ME /PERSON(21, "JOHN SMITH")/
DATA YOU%AGE, YOU%NAME /35, "FRED BROWN"/
INTEGER BIRD(3)
DATA BIRD /3*1/
REAL SKEW(100, 100)
DATA ((SKEW (K, J), K = 1, J-1), J = 1, 100) /4950 * 1.0/
DATA ((SKEW (K, J), K = J, 100), J = 1, 100) /5050 * 0.0/
```

In both forms, the character variable NAME is initialized with the value JOHN DOE with padding on the right because the length of the constant is less than the length of the variable. All ten elements of the integer array METERS are initialized to 0; an array constructor is used in the type declaration statement form; a repeat factor is used for the DATA statement form. ME and YOU are structures declared using the user-defined type PERSON defined in Section 4.6, page 110. In both forms ME is initialized using a structure constructor. In the DATA statement form YOU is initialized by supplying a separate value for each component.

In the type declaration statement form, the value 1 is broadcast to all 3 elements of BIRD. In the DATA statement form, a value must be supplied for each element of BIRD.

In both forms, the two-dimensional array SKEW is initialized so that the lower triangle is 0 and the strict upper triangle is 1. The RESHAPE(3i) intrinsic function is required in the first form because SKEW is of rank 2. Repeat factors are used in the second form.

5.5.1 PARAMETER Attribute and Statement

A constant can be given a name in a type declaration statement with the PARAMETER attribute or in a PARAMETER statement. The following is a format for a type declaration statement with a PARAMETER attribute:

```
type, PARAMETER [, attribute_list ] :: name = initialization_expression
```

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
DIMENSION
PRIVATE
PUBLIC
```

The *initialization_expression* must be present.

More than one named constant can be specified in a single type declaration statement; see the examples in this section.

The named constant becomes defined with the value determined from the initialization expression in accordance with the rules for intrinsic assignment. Any named constant that appears in the initialization expression must meet one of the following conditions:

- Be defined previously in this type declaration statement or in a previous type declaration statement
- Be accessible through host or use association

If the named constant is an array, it must have its array properties declared in this statement or in a previous statement in the same scoping unit.

The PARAMETER statement also provides a means of defining a named constant. Its format is defined as follows:

parameter_stmt	**is**	PARAMETER (*named_constant_def_list*)
named_constant_def	**is**	*named_constant* = *initialization_expr*

The PARAMETER statement also confers the PARAMETER attribute. It is subject to the same rules and restrictions as the PARAMETER attribute.

The PARAMETER attribute must not be specified for dummy arguments, functions, or objects in a common block.

A named constant that appears in a PARAMETER statement and is thereby declared and typed implicitly may appear in a subsequent type declaration statement only if that declaration confirms the implicit declaration.

A named array constant appearing in a PARAMETER statement must have had its array properties established previously.

A named constant must not appear in a format specification because of a possible ambiguity.

The following examples show entity-oriented declarations:

```
INTEGER, PARAMETER :: STATES = 50
INTEGER, PARAMETER :: M = MOD(28, 3),  &
       NUMBER_OF_SENATORS = 2 * STATES
```

The following examples show attribute-oriented declarations:

```
INTEGER STATES, M, NUMBER_OF_SENATORS
PARAMETER(STATES = 50)
PARAMETER(M = MOD(28, 3),  &
      NUMBER_OF_SENATORS = 2 * STATES)
```

5.6 Object Accessibility and Use

Several attributes indicate where an object can be accessed and how it can be used. Some of these attributes apply only to objects in a module and others apply only to dummy arguments or other variables that are declared in a subprogram.

5.6.1 PUBLIC and PRIVATE Attributes and Statements

The PUBLIC and PRIVATE attributes control access to type definitions, variables, functions, and named constants in a module. The PUBLIC attribute declares that entities in a module are available outside the module by use association; the PRIVATE attribute prevents access outside the module by use association. The default accessibility is PUBLIC, but it can be changed to PRIVATE.

The following formats are for type declaration statements with PUBLIC and PRIVATE attributes:

```
type, PUBLIC [, attribute_list ] :: entity_decl_list

type, PRIVATE [, attribute_list ] :: entity_decl_list
```

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

ALLOCATABLE
DIMENSION
EXTERNAL
INTRINSIC
PARAMETER
POINTER
SAVE
TARGET
VOLATILE (EXTENSION)

The type declaration statement can also contain an *initialization_expr*.

PUBLIC and PRIVATE specifications can also appear in the derived-type statement of a derived-type definition in a module to specify the accessibility of the type definition, as shown in the following:

```
TYPE, PUBLIC :: type_name

TYPE, PRIVATE :: type_name
```

If a PRIVATE statement appears inside a type definition, it specifies that, although the type may be accessible outside the module, its components are private.

For more information on derived-type definitions, see Section 4.4.1, page 97.

PUBLIC and PRIVATE statements provide another means for controlling the accessibility of variables, functions, type definitions, and named constants. PUBLIC and PRIVATE statements can control the accessibility of some entities that do not have a type; these are subroutines, generic specifiers, and namelist groups. The formats for PUBLIC and PRIVATE statements are defined as follows:

access_stmt	**is**	*access_spec* [[::] *access_id_list*]
access_spec	**is**	PUBLIC
	or	PRIVATE
access_id	**is**	*use_name*
	or	*generic_spec*

Specify one of the following for *generic_spec*:

- *generic_name*
- OPERATOR (*defined_operator*)
- ASSIGNMENT (=)

PUBLIC and PRIVATE statements can appear only in a module.

The PUBLIC and PRIVATE statements also confer the PUBLIC or PRIVATE attribute. They are subject to the same rules and restrictions as the PUBLIC and PRIVATE attributes.

A *use_name* can be the name of a variable, procedure, derived type, named constant, or namelist group.

Generic specifications are explained further in the *MIPSpro Fortran Language Reference Manual, Volume 2*. The following are examples of PUBLIC and PRIVATE statements that might be used with generic specifications:

```
PUBLIC HYPERBOLIC_COS, HYPERBOLIC_SIN     ! generic names
PRIVATE MY_COS_RAT, MY_SIN_RAT            ! specific names
PRIVATE MY_COS_INF_PREC, MY_SIN_INF_PREC  ! specific names
PUBLIC :: OPERATOR ( .MYOP. ), OPERATOR (+), ASSIGNMENT (=)
```

Only one PUBLIC or PRIVATE statement with an omitted *access_id* list is permitted in the scoping unit of a module. It determines the default accessibility of the module.

The default accessibility of entities defined in a module is PUBLIC. A PUBLIC statement without an *access_id* list can appear in the module to confirm the default accessibility. A PRIVATE statement without an *access_id* list can appear in the module to change the default accessibility.

A procedure that has a generic identifier that is public is accessible through the generic identifier even if its specific name is private. The converse is also true. That is, a module procedure that is public, but whose generic identifier is private, is still accessible through its specific name.

A module procedure that has an argument of a private type or a function result of a private type must be private and must not have a generic identifier that is public.

The following examples show entity-oriented declarations:

```
REAL, PUBLIC :: GLOBAL_X
TYPE, PRIVATE :: LOCAL_DATA
   LOGICAL :: FLAG
   REAL, DIMENSION(100) :: DENSITY
END TYPE LOCAL_DATA
```

The following examples show attribute-oriented declarations:

```
REAL GLOBAL_X
PUBLIC GLOBAL_X
TYPE LOCAL_DATA
   LOGICAL FLAG
   REAL DENSITY
   DIMENSION DENSITY(100)
END TYPE LOCAL_DATA
PRIVATE LOCAL_DATA
```

The following example shows a public type declaration with private components:

```
TYPE LIST_ELEMENT
   PRIVATE
   REAL VALUE
   TYPE(LIST_ELEMENT), POINTER :: NEXT, FORMER
END TYPE LIST_ELEMENT
```

The following example shows how to override the default accessibility:

```
MODULE M
   PRIVATE
   REAL R, K, TEMP(100)           ! R, K, and TEMP are private
   REAL, PUBLIC :: A(100), B(100) ! A and B are public
END MODULE M
```

5.6.2 INTENT Attribute and Statement

The INTENT attribute specifies the intended use of a dummy argument. If specified, it can help detect errors, provide information for readers of the program, and give the compiler information that can be used to make the code more efficient. It is particularly valuable in creating software libraries.

Some dummy arguments are referenced but not redefined within the subprogram; some are defined before being referenced within the subprogram; others can be referenced before being redefined. INTENT has three forms: IN, OUT, and INOUT, which correspond respectively to the preceding three situations.

If the intent of an argument is IN, the subprogram must neither change the value of the argument nor must the argument become undefined during the course of the subprogram. If the intent is OUT, the subprogram must not use the argument before it is defined, and it must be definable. If the intent is INOUT, the argument can be used to communicate information to the subprogram and return information; it must be definable. If no intent is specified, the use of the argument is subject to the limitations of the associated actual argument. For example, the actual argument may be a constant (for example, 2) or a more complicated expression (for example, N+2), and in these cases the dummy argument can be referenced but not defined.

The following is a format for a type declaration statement with an INTENT attribute:

type, INTENT (*intent_spec*) [, *attribute_list*] :: *decl_list*

For *intent_spec*, specify one of the following arguments:

```
IN
OUT
INOUT
```

The *attribute_list* can contain the following attributes:

```
DIMENSION
OPTIONAL
TARGET
VOLATILE (EXTENSION)
```

The `INTENT` statement also provides a means of specifying an intent for an argument. Its format is defined as follows:

intent_stmt	**is**	`INTENT (`*intent_spec*`) [ :: ]` *dummy_arg_name_list*
intent_spec	**is**	`IN`
	or	`OUT`
	or	`INOUT`

The `INTENT` attribute can be specified only for dummy arguments.

An `INTENT` statement can appear only in the specification part of a subprogram or interface body.

An intent must not be specified for a dummy argument that is a dummy procedure because it is not possible to change the definition of a procedure. Intent for a dummy pointer must not be specified either.

The `INTENT` statement also confers the `INTENT` attribute. It is subject to the same rules and restrictions as the `INTENT` attribute.

If an argument is of a type that is default initialized when it is declared with `INTENT(OUT)`, the components that are initialized are defined when the procedure is invoked.

An assumed-size array with `INTENT(OUT)` cannot be a type for which default initialization is specified.

The following examples show entity-oriented declarations:

```
SUBROUTINE MOVE(FROM, TO)
   USE PERSON_MODULE
   TYPE(PERSON), INTENT(IN) :: FROM
   TYPE(PERSON), INTENT(OUT) :: TO

SUBROUTINE SUB(X, Y)
   INTEGER, INTENT(INOUT) :: X, Y
```

The following examples show attribute-oriented declarations:

```
SUBROUTINE MOVE(FROM, TO)
   USE PERSON_MODULE
   TYPE(PERSON) FROM, TO
   INTENT(IN) FROM
   INTENT(OUT) TO

SUBROUTINE SUB(X, Y)
   INTEGER X, Y
   INTENT(INOUT) X, Y
```

5.6.3 OPTIONAL Attribute and Statement

The OPTIONAL attribute allows a procedure reference to omit arguments with this attribute. The PRESENT(3i) intrinsic function can be used to test the presence of an optional argument in a particular invocation and this test can be used to control the subsequent processing in the procedure. If the argument is not present, the subprogram can supply a default value or it can use an algorithm that is not based on the presence of the argument.

The following is a format for a type declaration statement with an OPTIONAL attribute:

type, OPTIONAL [, *attribute_list*] :: *entity_decl_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
DIMENSION
EXTERNAL
INTENT
POINTER
TARGET
```

VOLATILE (EXTENSION)

The OPTIONAL statement also provides a means for specifying an argument that can be omitted. Its format is defined as follows:

optional_stmt	**is**	OPTIONAL [::] *dummy_arg_name_list*

The OPTIONAL attribute can be specified only for dummy arguments.

An OPTIONAL statement can appear only in the scoping unit of a subprogram or interface body.

The OPTIONAL statement also confers the OPTIONAL attribute. It is subject to the same rules and restrictions as the OPTIONAL attribute.

The following examples show entity-oriented declarations in a program fragment:

```
CALL SORT_X(X = VECTOR_A)
   . . .
SUBROUTINE SORT_X(X, SIZEX, FAST)
   REAL, INTENT(INOUT) :: X (:)
   INTEGER, INTENT(IN), OPTIONAL :: SIZEX
   LOGICAL, INTENT(IN), OPTIONAL :: FAST
      . . .

   INTEGER TSIZE

      . . .
   IF (PRESENT(SIZEX)) THEN
      TSIZE = SIZEX
   ELSE
      TSIZE = SIZE(X)
   END IF

   IF (.NOT. PRESENT(FAST) .AND. TSIZE > 1000) THEN
      CALL QUICK_SORT(X)
   ELSE
      CALL BUBBLE_SORT(X)
   END IF
      . . .
```

The following examples show attribute-oriented declarations to be inserted in the same program fragment:

```
SUBROUTINE SORT_X(X, SIZEX, FAST)
   REAL X(:)
   INTENT(INOUT) X
   INTEGER SIZEX
   LOGICAL FAST
   INTENT(IN) SIZEX, FAST
   OPTIONAL SIZEX, FAST
      . . .
   INTEGER TSIZE
      . . .
```

5.6.4 SAVE Attribute and Statement

Variables with the SAVE attribute retain their value and their definition, association, and allocation status after the subprogram in which they are declared completes execution. Variables without the SAVE attribute cannot be depended on to retain their value and status, although the compiler treats named common blocks as if they had the SAVE attribute. The SAVE attribute should always be specified for an object or the object's common named block, if it is necessary for the object to retain its value and status.

Objects declared in a module can be given the SAVE attribute, in which case they always retain their value and status when a procedure that uses the module completes execution. Objects in modules must be in continual use in order to retain their values.

Objects declared in recursive subprograms can be given the SAVE attribute. Such objects are shared by all instances of the subprogram.

Any object that is data initialized (in a DATA statement or a type declaration statement) has the SAVE attribute by default.

The following is a format for a type declaration statement with a SAVE attribute:

type, SAVE [, *attribute_list*] :: *entity_decl_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

ALLOCATABLE

DIMENSION
POINTER
PRIVATE
PUBLIC
TARGET
VOLATILE (EXTENSION)

The type declaration statement can also contain an *initialization_expr*, but it cannot have the PARAMETER attribute.

The SAVE statement provides a means for specifying the SAVE attribute for objects and also for named common blocks. Its format is defined as follows:

save_stmt	**is**	SAVE [[::] *saved_entity_list*]
saved_entity	**is**	*data_object_name*
	or	/ *common_block_name* /

A SAVE statement without a saved entity list is treated as though it contained the names of all items that could be saved in the scoping unit. The compiler allows you to insert multiple SAVE statements without entity lists in a scoping unit.

ANSI/ISO: The Fortran standard permits only one SAVE statement without an entity list in a scoping unit.

If SAVE appears in a main program as an attribute or a statement, it has no effect.

The following objects must not be saved:

- A procedure
- A function result
- A dummy argument
- A named constant
- An automatic data object
- An object in a common block

- A namelist group

A variable in a common block cannot be saved individually; the entire named common block must be saved if you want any variables in it to be saved.

A named common block saved in one scoping unit of a program is saved throughout the program.

ANSI/ISO: The Fortran standard states that if a named common block is saved in one scoping unit of a program, it must be saved in every scoping unit of the program in which it is defined (other than the main program).

If a named common block is specified in a main program, it is available to any scoping unit of the program that specifies the named common block; it does not need to be saved.

The SAVE statement also confers the SAVE attribute. It is subject to the same rules and restrictions as the SAVE attribute.

The following example shows an entity-oriented declaration:

```
CHARACTER(LEN = 12), SAVE :: NAME
```

The following example shows an attribute-oriented declaration:

```
CHARACTER*12 NAME
SAVE NAME
```

The following example shows saving objects and named common blocks:

```
SAVE A, B, /BLOCKA/, C, /BLOCKB/
```

5.6.5 VOLATILE Attribute and Statement

The VOLATILE attribute and statement specifies that the value of an object is unpredictable. The object's value can change without visible assignment by the program, and it's value can be affected by external events. The presence of this statement prevents the compiler from optimizing references to specified variables, arrays, and common blocks of data.

The following format is for a type declaration statement with the VOLATILE attribute:

type, VOLATILE [, *attribute_list*] :: *entity_decl_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

ALLOCATABLE
AUTOMATIC (EXTENSION)
DIMENSION
INTENT
OPTIONAL
POINTER
PRIVATE
PUBLIC
SAVE
TARGET

The *entity_decl_list* can include the name of a common block, enclosed in slash characters (for example, /*common_block_name*/).

The format for the VOLATILE statement is as follows:

EXT	*volatile_stmt*	**is**	VOLATILE *entity_decl_list*
EXT	*entity_decl_list*	**is**	*data_object_name*
EXT		**or**	/*common_block_name*/

The following example shows a type declaration statement that specifies the VOLATILE attribute:

```
INTEGER, VOLATILE :: D, E
```

In the following example, the named common block, BLK1, and the variables D and E are volatile. Variables P1 and P4 become volatile because of the direct equivalence of P1 and the indirect equivalence of P4. The code that shows this is as follows:

```
PROGRAM TEST
LOGICAL(KIND=1) IPI(4)
INTEGER(KIND=4) A, B, C, D, E, ILOOK
INTEGER(KIND=4) P1, P2, P3, P4
COMMON /BLK1/A, B, C
```

```
VOLATILE /BLK1/, D, E
EQUIVALENCE(ILOOK, IPI)
EQUIVALENCE(A, P1)
EQUIVALENCE(P1, P4)
```

The presence of a VOLATILE attribute or statement can inhibit some optimizations because it asserts that the compiler must perform loads and stores from the specified objects. As an example, consider the following code fragment:

```
J = 1
DO I = 1,100000
  IF (J.EQ.2) PRINT 'FOO'
END DO
```

If the preceding code were included in a Fortran program, the compiler might remove the statement `IF (J.EQ.2) PRINT 'FOO'` because J is loop invariant and because J was previously assigned the value 1. If J were declared VOLATILE, the compiler would perform all loads of J because something else might affect the value of J.

ANSI/ISO: The Fortran standard does not describe the VOLATILE attribute or statement.

A variable or common block must be declared VOLATILE if it can be read or written to in a way that is not visible to the compiler. This would be the case in the following situations:

- If an operating system feature is used to place a variable in shared memory so that it can be accessed by other programs, the variable must be declared VOLATILE.
- If a variable is accessed or modified by a routine called by the operating system when an asynchronous event occurs, the variable must be declared VOLATILE.
- If a variable might be written by one thread and then read by a different thread, it must be marked VOLATILE.

If an array is declared VOLATILE, each element in the array is VOLATILE. If a common block is declared VOLATILE, each variable in the common block is VOLATILE.

If an object of derived type is declared VOLATILE, its components are VOLATILE.

If a pointer is declared VOLATILE, the pointer itself is VOLATILE.

A VOLATILE statement must not specify a procedure, function result, or NAMELIST group name.

5.7 Procedure Properties

If an external or dummy procedure is to be an actual argument to a subprogram, the procedure name must be declared EXTERNAL. (A *dummy procedure* is a dummy argument that is a procedure.) If an external procedure has the same name as an intrinsic procedure, again the name must be declared EXTERNAL. When this occurs, the intrinsic procedure of that name is no longer accessible to that program unit. If an intrinsic procedure is to be an actual argument, the name of the procedure must be declared INTRINSIC. The *MIPSpro Fortran Language Reference Manual, Volume 2*, discusses further the usage of these attributes.

Because only functions, not subroutines, are declared to have a type (the type of the result), only function names can appear in type declaration statements. The EXTERNAL and INTRINSIC attributes in type declaration statements therefore apply only to functions. The EXTERNAL and INTRINSIC statements can be used to specify properties of subroutines, and the EXTERNAL statement can specify block data program units. For information on block data program units, see the *MIPSpro Fortran Language Reference Manual, Volume 2*.

5.7.1 EXTERNAL Attribute and Statement

The EXTERNAL attribute in a type declaration statement indicates that a name is the name of an external function or a dummy function and permits the name to be used as an actual argument.

The following is a format for a type declaration statement with an EXTERNAL attribute:

```
type, EXTERNAL [, attribute_list ] :: function_name_list
```

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
OPTIONAL
PRIVATE
PUBLIC
```

An interface block can be used to describe the interface of an external function. A function described by an interface block has the EXTERNAL attribute by default, so the function name cannot also be given the EXTERNAL attribute by any other means. Note that an interface block specifies the EXTERNAL attribute for all procedures in the interface block, with the exception of module procedures specified in MODULE PROCEDURE statements within the block. For information on interface blocks, see the *MIPSpro Fortran Language Reference Manual, Volume 2*.

The EXTERNAL statement provides a means for declaring subroutines and block data program units, as well as functions, to be external. Its format is defined as follows:

external_stmt	**is**	EXTERNAL *external_name_list*

Each external name must be the name of an external procedure, a dummy argument, or a block data program unit.

If a dummy argument is specified to be EXTERNAL, the dummy argument is a dummy procedure.

The EXTERNAL statement also confers the EXTERNAL attribute. It is subject to the same rules and restrictions as the EXTERNAL attribute.

The following examples of entity-oriented declarations:

```
SUBROUTINE SUB(FOCUS)
   INTEGER, EXTERNAL :: FOCUS
   LOGICAL, EXTERNAL :: SIN
```

The following example shows an attribute-oriented declaration:

```
SUBROUTINE SUB (FOCUS)
   INTEGER FOCUS
   LOGICAL SIN
   EXTERNAL FOCUS, SIN
```

FOCUS is declared to be a dummy procedure. SIN is declared to be an external procedure. Both are functions. To declare an external subroutine, the EXTERNAL statement or an interface block must be used because a subroutine does not have a type, and thus its attributes cannot be specified in a type declaration statement. The specific and generic name SIN of the intrinsic function SIN is no longer available to subroutine SUB.

5.7.2 INTRINSIC Attribute and Statement

The INTRINSIC attribute in a type declaration statement indicates that a name is the name of an intrinsic function and permits the names of some intrinsic functions to be used as actual arguments.

The following is a format for a type declaration statement with an INTRINSIC attribute:

type, INTRINSIC [, *attribute_list*] :: *intrinsic_function_name_list*

For *attribute_list*, specify one of the following attributes:

PRIVATE
PUBLIC

The INTRINSIC statement provides a means for declaring intrinsic subroutines, as well as functions. Its format is defined as follows:

intrinsic_stmt **is** INTRINSIC *intrinsic_procedure_name_list*

Each *intrinsic_procedure_name* must be the name of an intrinsic procedure.

A name must not be declared to be both EXTERNAL and INTRINSIC in a scoping unit.

A type can be specified for an intrinsic function even though it has a type as specified in the *MIPSpro Fortran Language Reference Manual, Volume 2*. If a type is specified for the generic name of an intrinsic function, it does not remove the generic properties of the function name.

The INTRINSIC statement also confers the INTRINSIC attribute. It is subject to the same rules and restrictions as the INTRINSIC attribute.

The compiler has implemented intrinsic procedures in addition to the ones required by the standard. These procedures have the status of intrinsic procedures, but programs that use them may not be portable. It is recommended that such procedures be declared INTRINSIC to allow other processors to diagnose whether or not they are intrinsic for those processors.

The following is an example of an entity-oriented declaration:

```
REAL, INTRINSIC :: SIN, COS
```

The following is an example of an attribute-oriented declaration:

```
REAL SIN, COS
INTRINSIC SIN, COS
```

Because the interfaces of intrinsic procedures are known to the compiler, it is not necessary to specify a type for them, but it is not incorrect to do so.

5.8 Automatic Data Objects

Automatic data objects are especially useful as working storage in a procedure. These objects can be declared only in procedures or procedure interfaces; they are created when the procedure is entered and disappear when the procedure completes execution. They can be created the same size as an argument to the procedure, so they can be tailored to each invocation.

The following are the three kinds of automatic data objects:

- Automatic arrays of any type
- Objects of type character
- Local variables and arrays not in a common block or module and not declared with the SAVE attribute

An *automatic array or character data object* is one with a specification that depends on the value of a nonconstant expression and is not a dummy argument. Automatic arrays are those whose size depends on a variable used in a bound expression. The size of an automatic array or character data object is not known at compile time. The size is calculated at execution time, and storage is allocated upon entry into the procedure. Storage is freed upon exit from the procedure. Local variables and arrays may be declared with the AUTOMATIC attribute. For more information on the AUTOMATIC attribute, see Section 5.4.3, page 141.

The following are examples of automatic data objects:

```
SUBROUTINE SUB (N, DUMMY_ARRRAY)
COMMON /CB/ K
INTEGER AUTO_ARRAY(N)      ! Automatic array.
CHARACTER(LEN=K*2) CH      ! Automatic character variable.
INTEGER DUMMY_ARRAY(K,N)   ! Not an automatic array
```

```
                                ! because it is a dummy
                                ! argument, not a local array.. . .
END SUBROUTINE
```

An automatic array or character data object is one with a specification that depends on the value of a nonconstant expression and is not a dummy argument. Automatic arrays are those whose size depends on a value used in a bound expression.

An automatic array or character data object is similar to an object declared with the AUTOMATIC attribute. For both items, storage is allocated when the procedure is entered and deallocated when the procedure is exited. The differences between these types are as follows:

- The size of an automatic array or character data object is not known at compile time. The size is calculated at execution time, and storage is allocated upon execution of the procedure. Automatic arrays and character data objects cannot be declared with the AUTOMATIC attribute.
- The size of objects declared with the AUTOMATIC attribute must be known at compile time. Storage is allocated with the initial stack allocation upon entry to the procedure.

In Fortran, the term *automatic array or character object* does not include noncharacter scalar local variables or arrays with constant bounds. For an array, the extents in each dimension are determined when the procedure is entered. For a character object, the length is determined when the procedure is entered. Apart from dummy arguments, this is the only character object whose length can vary. For arrays, extents can vary for allocatable arrays and array pointers as well as dummy arguments. An automatic array or character object is not a dummy argument, but it is declared with a specification expression that is not a constant expression. The specification expression can be the length of the character object or the bounds of the array. For variables declared with the AUTOMATIC attribute, the variables must be scalar or array values with constant bounds, and they cannot be declared in a common block or module. These variables are allocated on the stack. Automatic objects cannot be saved or initialized.

In the following example, C is an automatic array and MESSAGE is an automatic character object:

```
SUBROUTINE SWAP_ARRAYS(A, B, A_NAME, B_NAME)
   REAL, DIMENSION(:), INTENT(INOUT) :: A, B
   CHARACTER(LEN = *), INTENT(IN)     :: A_NAME, B_NAME
```

```
   REAL C(SIZE (A))
   CHARACTER (LEN = LEN(A_NAME) + LEN(B_NAME) + 17) MESSAGE

   C = A
   A = B
   B = C

   MESSAGE = A_NAME // " and " // B_NAME // " are swapped"
   PRINT *,  MESSAGE
END SUBROUTINE SWAP_ARRAYS
```

ANSI/ISO: The Fortran standard does not provide a means to explicitly declare automatic variables as automatic.

5.9 NAMELIST Statement

A NAMELIST statement establishes the name for a collection of objects that can then be referenced by the group name in certain I/O statements. The NAMELIST statement is defined as follows:

namelist_stmt	**is**	NAMELIST / *namelist_group_name* / *namelist_group_object_list* [[,] / *namelist_group_name* / *namelist_group_object_list*] ...
namelist_group_object	**is**	*variable_name*

A variable in the variable name list must not be an array dummy argument with nonconstant bounds, a variable with assumed character length, an automatic object, a pointer, a Cray pointer, an object of a type that has a pointer component at any level, an allocatable array, or a subobject of any of the preceding objects.

ANSI/ISO: The Fortran standard does not describe Cray pointers.

If a namelist group name has the PUBLIC attribute, no item in the namelist group object list can have the PRIVATE attribute or have private components.

The namelist group name cannot be a name made accessible by USE association.

The order in which the data objects (variables) are specified in the NAMELIST statement determines the order in which the values appear on output.

A namelist group name can occur in more than one NAMELIST statement in a scoping unit. The variable list following each successive appearance of the same namelist group name in a scoping unit is treated as a continuation of the list for that namelist group name.

A variable can be a member of more than one namelist group.

A variable must have its type, type parameters, and shape specified previously in the same scoping unit, or it must be determined by implicit typing rules. If a variable is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm the implicit type and type parameters. The following is an example of a NAMELIST statement:

```
NAMELIST /N_LIST/ A, B, C
```

5.10 Storage Association

Generally, the physical storage units or storage order for data objects cannot be specified. However, the COMMON, EQUIVALENCE, and SEQUENCE statements provide sufficient control over the order and layout of storage units to permit data to share storage units.

The COMMON statement provides a means of sharing data between program units. The EQUIVALENCE statement provides a means whereby two or more objects can share the same storage units.

Fortran modules, pointers, allocatable arrays, and automatic data objects provide additional tools for sharing data and managing storage. The SEQUENCE statement defines a storage order for structures. This permits structures to appear in common blocks and be equivalenced. The SEQUENCE statement can appear only in derived-type definitions to define sequence types. The components of a sequence type have an order in storage sequences that is the order of their appearance in the type definition.

5.10.1 Storage Units

Fortran includes numeric and character storage units for numeric and character data. The nondefault types (user-defined types and pointers), however, are stored in

unspecified storage units. These unspecified storage units are used for pointers, objects of nondefault type, and structures that contain components that are of nondefault types or are pointers. This unit is different for each different sort of object. A pointer occupies a single unspecified storage unit that is different from that of any nonpointer object and can be different for each combination of type, type parameters, and rank.

There are two kinds of structures, sequence structures and nonsequence structures, depending on whether or not the type definition contains a `SEQUENCE` statement. A nonsequence structure occupies a single unspecified storage unit that is different for each type. The three kinds of sequence structures are as follows:

- Numeric sequence structures (containing only numeric and logical entities of default kind)
- Character sequence structures (containing only character entities)
- Sequence structures (containing a mixture of components including objects that occupy numeric, character, and unspecified storage units)

Table 5-1, page 171, lists objects of various types and attributes and the storage units they occupy.

Table 5-1 Types, attributes, and storage

Types and attributes of object	Storage units
Default integer	1 numeric
Default real	1 numeric
Logical	1 numeric
Double precision	2 numeric
Default complex	2 numeric
Character of length 1	1 character
Character of length *s*	*s* characters
Nondefault integer	1 unspecified
Real other than default real or double precision	1 unspecified
Nondefault complex	1 unspecified
Nonsequence structure	1 unspecified

Types and attributes of object	Storage units
Numeric sequence structure	*n* numeric, where *n* is the number of numeric storage units the structure occupies
Character sequence structure	*n* characters, where *n* is the number of character storage units the structure occupies
Sequence structure	The sum of the storage sequences of all the ultimate components of the structure.
Any type with the `POINTER` attribute	1 unspecified
Any intrinsic or sequence type with the `DIMENSION` attribute	The size of the array times the number of storage units for the type (will appear in array element order)
Any nonintrinsic or nonsequence type with the `DIMENSION` attribute	One unspecified storage unit for each element of the array
Any type with the `POINTER` attribute and the `DIMENSION` attribute	1 unspecified

5.10.2 Storage Sequence

A *storage sequence* is an ordered sequence of storage units. The storage units can be elements in an array, characters in a character variable, components in a sequence structure, or variables in a common block. A sequence of storage sequences forms a composite storage sequence. The order of the storage units in such a composite sequence is the order of the units in each constituent taken in succession, ignoring any zero-sized sequences.

Storage is associated when the storage sequences of two different objects have some storage in common. This permits two or more variables to share the same storage. Two objects are *totally associated* if they have the same storage sequence; two objects are *partially associated* if they share some storage but are not totally associated.

5.10.3 EQUIVALENCE Statement

To indicate that two or more variables will share storage, they can be placed in an equivalence group in an EQUIVALENCE statement. If the objects in an equivalence group have different types or type parameters, no conversion or mathematical relationship is implied. If a scalar and an array are equivalenced, the scalar does not have array properties and the array does not have the properties of a scalar. The format of the EQUIVALENCE statement is defined as follows:

equivalence_stmt	**is**	EQUIVALENCE *equivalence_set_list*
equivalence_set	**is**	(*equivalence_object*, *equivalence_object_list*)
equivalence_object	**is**	*variable_name*
	or	*array_element*
	or	*substring*

An equivalence object must not be one of the following items:

- A dummy argument
- A Fortran pointer
- An allocatable array
- A structure containing a pointer at any level
- An automatic data object
- A function name, result name, or entry name
- A named constant
- A structure component
- A Cray pointee
- An object made accessible by USE association.
- A subobject of any of the preceding
- An object with the TARGET attribute.

An equivalence group list must contain at least two items.

Subscripts and substring ranges must be integer initialization expressions. A substring cannot have a length of zero.

If an equivalence object is of type default integer, default real, double-precision real, default complex, default logical, or numeric sequence type, all objects in the set must be of these types.

If an equivalence object is of type character or character sequence type, all objects in the set must be type character. The lengths do not need to be the same.

If an equivalence object is of sequence type other than numeric or character sequence type, all objects in the set must be of the same type.

ANSI/ISO: The compiler allows equivalencing of character data with noncharacter data. The Fortran standard does not address this. It is recommended that you do not perform equivalencing in this manner, however, because alignment and padding differs across platforms, thus rendering your code less portable.

If an equivalence object is of intrinsic type other than default integer, default real, double-precision real, default complex, or default logical, all objects in the set must be of the same type with the same kind type parameter value.

The use of an array name unqualified by a subscript list in an equivalence set specifies the first element of the array; that is, `A` means the first element of `A`.

An `EQUIVALENCE` statement must not specify that the same storage unit is to occur more than once in a storage sequence. For example, the following is illegal because it would indicate that storage for `X(2)` and `X(3)` is shared:

```
EQUIVALENCE (A, X(2)), (A, X(3))
```

An `EQUIVALENCE` statement must not specify the sharing of storage units between objects declared in different scoping units.

An `EQUIVALENCE` statement specifies that the storage sequences of the data objects in an equivalence set are storage associated. Any nonzero-sized sequences in the set have the same first storage unit, and any zero-sized sequences are storage associated with one another and with the first storage unit of any nonzero-sized sequences. This causes storage association of the objects in the group and may cause storage association of other data objects.

Example 1: The following code causes the alignment illustrated in Figure 5-2:

```
CHARACTER(LEN = 4) :: A, B
CHARACTER(LEN = 3) :: C(2)
EQUIVALENCE (A, C(1)), (B, C(2))
```

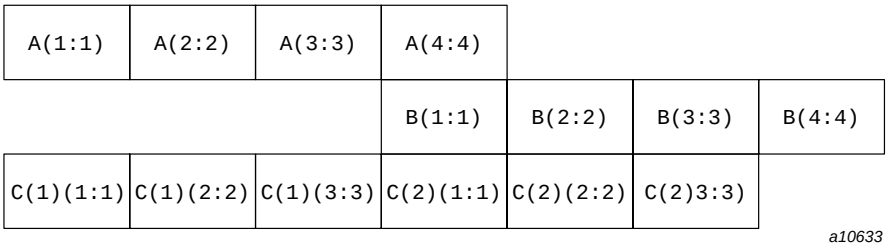

Figure 5-2 Character alignment example

As a result, the fourth character of `A`, the first character of `B`, and the first character of `C(2)` all share the same character storage unit.

Example 2: Figure 5-3, page 175, illustrates alignment of the following two numeric arrays:

```
REAL, DIMENSION(6) :: X, Y
EQUIVALENCE (X(5), Y(3))
```

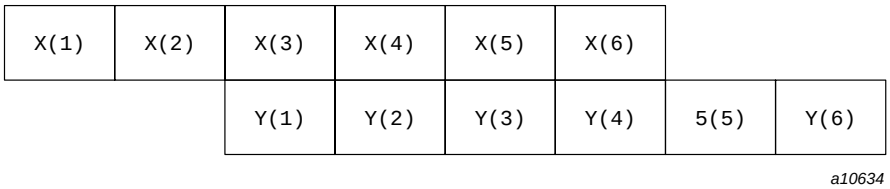

Figure 5-3 Numeric array alignment example

5.10.4 `COMMON` Statement

The `COMMON` statement establishes blocks of storage called *common blocks* and specifies objects that are contained in the blocks. Two or more program units can share this space and thus share the values of variables stored in the space. Thus, the `COMMON` statement provides a global data facility based on storage association. Common

blocks can be named, in which case they are called *named common blocks,* or they can be unnamed, in which case they are called *blank common.*

Common blocks can contain mixtures of storage units and can contain unspecified storage units; however, if a common block contains a mixture of storage units, every declaration of the common block in the program must contain the same sequence of storage units, thereby matching types, kind type parameters, and attributes (`DIMENSION` and `POINTER`). The format of the `COMMON` statement is defined as follows:

common_stmt	**is**	`COMMON [ / [` *common_block_name* `] / ]` *common_block_object_list* `[ [ , ] / [` *common_block_name* `] /` *common_block_object_list* `] ...`
common_block_object	**is**	*variable_name* `[ (`*explicit_shape_spec_list*`) ]`

A *common_block_object* must not be one of the following items:

- A dummy argument
- An allocatable array
- An automatic object
- A function name, result name, or entry name
- A `USE` associated variable
- A `HOST` associated variable
- A sequence structure with default initialization
- A nonsequence structure

The appearance of two slashes with no common block name between them declares that the objects that follow are in blank common.

A common block name or an indication of blank common can appear more than once in one or more `COMMON` statements in the same scoping unit. The object list following each successive block name or blank common indication is treated as a continuation of the previous object list.

A variable can appear in only one common block within a scoping unit.

If a variable appears with an explicit-shape specification list, it is an array, and each bound must be a constant specification expression.

Only a named common block can be saved. Individual variables in the common block cannot be saved.

For each common block, a common block storage sequence is formed. It consists of the sequence of storage units of all the variables listed for the common block in the order of their appearance in the common block list. The storage sequence may be extended (on the end) to include the storage units of any variable equivalenced to a variable in the common block. Data objects storage associated with a variable in a common block are considered to be in that common block. The size of a common block is the size of its storage sequence including any extensions of the sequence resulting from equivalence association.

Within an executable program, the common block storage sequences of all nonzero-sized common blocks with the same name have the same first storage unit. Zero-sized common blocks are permitted. All zero-sized common blocks with the same name are storage associated with one another. The same is true for all blank common blocks except that because they can be of different sizes, it is possible for a zero-sized blank common block in one scoping unit to be associated with the first storage unit of a nonzero-sized blank common block in another scoping unit. In this way, many subprograms can use the same storage. They can specify common blocks to communicate global values or to reuse and conserve storage. `USE` association or `HOST` association can cause these associated objects to be accessible in the same scoping unit.

A nonpointer object of type default integer, default real, double-precision real, default complex, default logical, or numeric sequence type must become associated with only nonpointer objects of these types.

A nonpointer object of type character or character sequence must become associated with only nonpointer objects of these types.

If an object of numeric sequence or character sequence type appears in a common block, it is as if the individual components were enumerated in order directly in the common block object list.

A nonpointer object of sequence type other than numeric or character sequence type must become associated only with nonpointer objects of the same type.

A nonpointer object of intrinsic type other than default integer, default real, double precision real, default complex, default logical, or character must become associated

only with nonpointer objects of the same type with the same kind type parameter value.

A pointer must become associated only with pointers of the same type, type parameters, and rank.

Note: An object with the TARGET attribute can become storage associated only with another object that has the TARGET attribute and the same type and type parameters.

The compiler treats named common blocks and blank common blocks identically, as follows:

- Variables in blank common and variables in named common blocks can be initialized.
- Named common blocks and blank common are always saved.
- Named common blocks of the same name and blank common can be of different sizes in different scoping units.

ANSI/ISO: The Fortran standard lists the following differences between blank common and named common blocks:

- Variables in blank common must not be initially defined in type declaration statements or DATA statements.
- A named common block is not saved unless it is named in a SAVE statement.
- Named common blocks of the same name must be of the same size in all scoping units.

Consider the code in the following example:

```
SUBROUTINE FIRST
   REAL B(2)
   COMPLEX C
   LOGICAL FLAG
   TYPE COORDINATES
      SEQUENCE
      REAL X, Y
      LOGICAL Z_O      ! ZERO ORIGIN?
   END TYPE COORDINATES
   TYPE(COORDINATES) P
```

```
   COMMON /REUSE/ B, C, FLAG, P

   REAL MY_VALUES(100)
   CHARACTER(LEN = 20) EXPLANATION
   COMMON /SHARE/ MY_VALUES, EXPLANATION
   SAVE /SHARE/

   REAL, POINTER :: W(:, :)
   REAL, TARGET, DIMENSION(100, 100) :: EITHER, OR
   INTEGER(SHORT) :: M(2000)
   COMMON /MIXED/ W, EITHER, OR, M
      . . .
END SUBROUTINE
SUBROUTINE SECOND
   INTEGER, PARAMETER :: SHORT = 2
   INTEGER I(8)
   COMMON /REUSE/ I

   REAL MY_VALUES(100)
   CHARACTER(LEN = 20) EXPLANATION
   COMMON /SHARE/ MY_VALUES, EXPLANATION
   SAVE /SHARE/

   REAL, POINTER :: V(:)
   REAL, TARGET, DIMENSION(100000) :: ONE, ANOTHER
   INTEGER(SHORT) :: M(2000)
   COMMON /MIXED/ V, ONE, ANOTHER, M  ! ILLEGAL
   . . .
END SUBROUTINE
```

Common block REUSE has a storage sequence of 8 numeric storage units. It is used to conserve storage. The storage referenced in subroutine FIRST is associated with the storage referenced in subroutine SECOND, as follows:

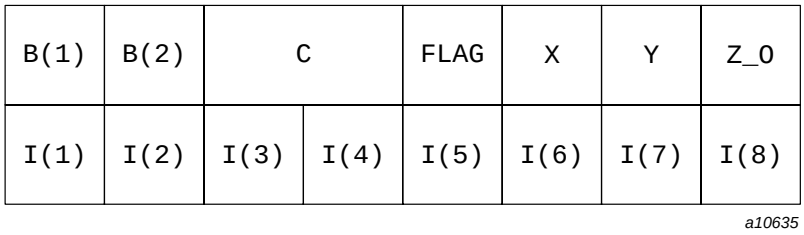

Figure 5-4 Storage of REUSE in FIRST and SECOND

The Fortran standard does not guarantee that the storage is actually retained and reused because, in the absence of a SAVE attribute for REUSE, some compilers can release the storage when either of the subroutines completes execution. The compiler treats named common blocks as entities that are contained in a SAVE statement.

Common block SHARE contains both numeric and character storage units and is used to share data between subroutines FIRST and SECOND.

The declaration of common block MIXED in subroutine SECOND is illegal because it does not have the same sequence of storage units as the declaration of MIXED in subroutine FIRST. The array pointer W in FIRST has two dimensions; the array pointer V in SECOND has only one. With common blocks, it is the sequence of storage units that must match, not the names of variables.

5.10.5 Restrictions on Common and Equivalence

An EQUIVALENCE statement must not cause two different common blocks to become associated and must not cause a common block to be extended by adding storage units preceding the first storage unit of the common block. For example, the following code is legal and results in the alignment illustrated in Figure 5-5, page 181:

```
COMMON A(5)
REAL B(5)
EQUIVALENCE (A(2), B(1))
```

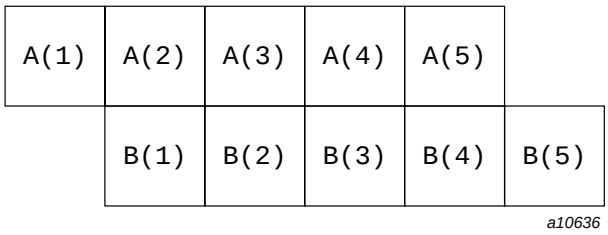

Figure 5-5 Alignment resulting from correct code

On the other hand, the following code is not legal because it would place `B(1)` ahead of `A(1)`, as is illustrated in Figure 5-6:

```
EQUIVALENCE (A(1), B(2))
```

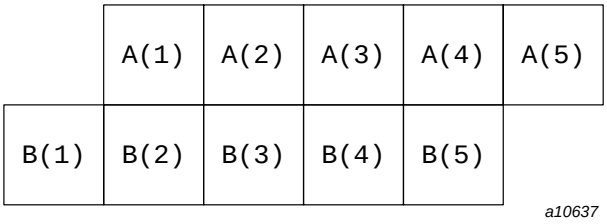

Figure 5-6 Alignment resulting from incorrect code

`COMMON` and `EQUIVALENCE` statements can appear in a module. The name of a public data object from a module must not appear in a `COMMON` or `EQUIVALENCE` statement in any scoping unit that has access to the data object.

`EQUIVALENCE` association must not cause a derived-type object with default initialization to be associated with an object in a common block.

Using Data

Chapter 5, page 115, explains how data objects are created and how their attributes are specified. This chapter explains how these objects can be used. The appearance of the name or designator where its value is required is a *reference* to the object. When an object is referenced, it must be defined; that is, it must have a value. The reference makes use of the value. Consider the following two statements:

```
A = 1.0
B = A + 4.0
```

In the first statement, the constant value `1.0` is assigned to the variable `A`. It does not matter whether `A` was previously defined with a value or not; it now has a value and can be referenced in an executable statement. In the second statement, `A` is referenced; its value is obtained and added to the constant `4.0` to obtain a value that is then assigned to the variable `B`. The appearances of `A` in the first statement and `B` in the second statement are not considered to be references because their values are not required. The appearance of `A` in the second statement is a reference.

A data object can be a constant or a variable. Variables and constants can be scalar objects (with a single value) or arrays (with any number of values, all of the same type). Strictly speaking, there is no such thing as an array constant. An array constructor made up of all constant values is a *constant expression*, not an array constant. Also note that a derived type is a scalar value. Similar to an array constructor, a derived type constructor composed of all constant values is a constant expression, not a structure constant.

Arrays are said to be dynamic if their size can change. Automatic arrays are discussed in Section 5.8, page 167; they are created on entry to a procedure, and their sizes are determined at that time. Allocatable arrays or pointer arrays can change size as well. The declared rank cannot change, but the extents of the dimensions may change with each reallocation or pointer assignment.

If a variable or constant is a portion of another object, it is called a *subobject*. A subobject can be one of the following items:

- An array element
- An array section
- A structure component

- A substring

A variable is referenced by its name, whereas a subobject is referenced by a *designator*. A designator indicates the portion of an object that is being referenced. Each subobject is considered to have a parent and is a portion of the parent. Each of the subobjects is described in this chapter.

This chapter also explains how to create and release pointers and allocatable arrays by using the `ALLOCATE` and `DEALLOCATE` statements. In addition, you can disassociate pointers from any target object by using the `NULLIFY` statement.

A reference to a variable or subobject is called a *data reference*. Guidelines exist for determining whether a particular data reference is classified as a character string, character substring, structure component, array, array element, or array section. These classifications are perhaps of more interest to compiler writers than to users of the language, but knowing how a data reference is classified makes it clearer which rules and restrictions apply to the reference, and easier to understand some of the explanations for the formation of expressions. Briefly, character strings and substrings must be of type character. Arrays have the `DIMENSION` attribute. Some data references can be classified as both structure components and array sections. In general, if a data reference contains a percent sign (%), it is a structure component, but its actual classification can be determined by other factors such as a section subscript or the rightmost element of the reference. If a substring range appears in a data reference, it must appear at the right end of the reference; the reference is considered to be a substring unless some component of the reference is an array section, in which case the data reference is considered to be an array section that just happens to have elements that are substrings. For a component reference to be classified as an array element, every component must have rank zero and a subscript list must appear at the right end of the reference. Section 6.1, page 184, through Section 6.4.5, page 194, contain many examples that demonstrate how these guidelines for classification apply.

6.1 Constants and Variables

A constant has a value that cannot change; it can be a literal constant or a named constant (parameter). As explained in Chapter 4, page 67, each of the intrinsic types has a form that specifies the type, type parameters, and value of a literal constant of the type. For user-defined types, there is a structure constructor to specify values of the type. If all of the components of a structure constructor are constants, the resulting derived-type value is a constant expression. Array constructors are used to form array values of any intrinsic or user-defined type. If all array elements are

constant values, the resulting array is a constant array expression. A reference to a constant is always permitted.

A variable has a name such as `A` or a designator such as `B(I)`, and may or may not have a value. If it does not have a value, it must not be referenced. Variables are defined as follows:

variable	**is**	*scalar_variable_name*
	or	*array_variable_name*
	or	*subobject*
subobject	**is**	*array_element*
	or	*array_section*
	or	*structure_component*
	or	*substring*
logical_variable	**is**	*variable*
default_logical_variable	**is**	*variable*
char_variable	**is**	*variable*
default_char_variable	**is**	*variable*
int_variable	**is**	*variable*
default_int_variable	**is**	*variable*

A *logical_variable* must be of type logical, and a *default_logical_variable* must be of type default logical. A *char_variable* must be of type character and a *default_char_variable* must be of type default character. The compiler does not support any nondefault character types. An *int_variable* must be of type integer and a *default_int_variable* must be of type default integer.

Variables can be of any type, except for Boolean (typeless). There are contexts in which a variable must be of a certain type. In some of these cases, terms, such as *logical_variable*, *character_variable*, or Cray pointer, provide precise limitations.

ANSI/ISO: The Fortran standard does not specify Boolean (typeless) constants or Cray pointers.

A subobject with a constant parent is not a variable.

A single object of any of the intrinsic or user-defined types is a *scalar*. A set of scalar objects, all of the same type and type parameters, can be arranged in a pattern involving columns, rows, planes, and higher-dimensioned configurations to form an array. An array has a rank between one and seven. A scalar has rank zero. In simple terms, an array is an object with the DIMENSION attribute; a scalar is not an array. For example:

```
TYPE PERSON
   INTEGER AGE
   CHARACTER(LEN = 40) NAME
END TYPE PERSON

TYPE(PERSON) FIRECHIEF, FIREMEN(50)
CHARACTER*(20) DISTRICT, STATIONS(10)
```

The following data references are classified as indicated by the comments on each line:

```
DISTRICT      ! character string
DISTRICT(1:6) ! substring
FIRECHIEF%AGE ! structure component
FIREMEN%AGE   ! array of integers
STATIONS      ! array of character strings
STATIONS(1)   ! array element (character string)
STATIONS(1:4) ! array section of character strings
```

The following code segment shows that a subobject can have a constant parent:

```
CHARACTER(*), PARAMETER :: MY_DISTRICT = "DISTRICT 13"
CHARACTER(2) DISTRICT_NUMBER
DISTRICT_NUMBER = MY_DISTRICT(10:11)
```

DISTRICT_NUMBER has the value 13.

6.2 Substrings

A *character string* consists of zero or more characters. Even though it is made up of individual characters, a character string is considered to be scalar. As with any data type, it is possible to declare an array of character strings, all of the same length.

A *substring* is a contiguous portion of a character string that has a starting point and an ending point within the character string. It is possible to reference a substring of a character variable or constant.

Substrings are defined as follows:

substring	**is**	*parent_string* (*substring_range*)
parent_string	**is**	*scalar_variable_name*
	or	*array_element*
	or	*scalar_structure_component*
	or	*scalar_constant*
substring_range	**is**	[*scalar_int_expr*] : [*scalar_int_expr*]

The *parent_string* of a substring must be of type character. The *substring* is of type character.

The *scalar_int_expr* at the left in the *substring_range* is the starting position. The *scalar_int_expr* at the right in the *substring_range* is the ending position.

A *substring* is the contiguous sequence of characters within the string, beginning with the character at the starting position and ending at the ending position. If the starting position is omitted, the default is 1; if the ending position is omitted, the default is the length of the character string.

The length of a character string or substring can be 0, but not a negative number. Zero-length strings result when the starting position is greater than the ending position. The formula for calculating the length of a string is as follows:

```
MAX (ending_position - starting_position + 1, 0)
```

The first character of a parent string is at position `1` and the last character is at position *n* where *n* is the length of the string. The starting position of a substring must be greater than or equal to `1` and the ending position must be less than or equal

to the length *n*, unless the length of the substring is `0`. If the parent string is of length `0`, the substring must be of length `0`.

Example 1:

```
CHARACTER*(14) NAME
NAME = "John Q. Public"
NAME(1:4) = "Jane"
PRINT *, NAME(9:14)
```

In example 1, `NAME` is a scalar character variable, a string of 14 characters, that is assigned the value `John Q. Public` by the first assignment statement. `NAME(1:4)` is a substring of four characters that is reassigned the value `Jane` by the second assignment statement, leaving the remainder of the string `NAME` unchanged; the string `NAME` then becomes `Jane Q. Public`. The `PRINT` statement prints the characters in positions 9 through 14, in this case, the surname, `Public`.

Example 2:

Assume the following definition and declarations:

```
TYPE PERSON
   INTEGER AGE
   CHARACTER(LEN = 40) NAME
END TYPE PERSON

TYPE(PERSON) FIRECHIEF, FIREMEN(50)
CHARACTER(20) DISTRICT, STATIONS(10)
```

The following are all substrings:

```
STATIONS(1)(1:5)    ! array element as parent string
FIRECHIEF%NAME(4:9) ! structure component as parent string
DISTRICT(7:14)      ! scalar variable as parent string
'0123456789'(N:N+1) ! character constant as parent string
```

In example 2, the reference `STATIONS(:)(1:5)` is permitted. It is an array whose elements are substrings, but it is not considered to be a substring reference. Even though the entire array is indicated, this reference is considered to be an array section reference. The description is in Section 6.4.5, page 194. `STATIONS(1:5)(1:5)` is also permitted. It is an array section whose elements are substrings. Whenever an array is constructed of character strings and any part of it (other than the whole object) is referenced, an array section subscript must appear before the substring range specification, if any. Otherwise, the substring range specification will be treated as an

array section specification because the two have the same form. `STATIONS(1:5)` is an array section reference that references the entire character strings of the first five elements of `STATIONS`. The last line of the example is a substring where the parent is a constant and the starting and ending positions are variable.

6.3 Structure Components

A *structure* is an aggregate of components of intrinsic or derived types. It is itself an object of derived type. The types and attributes of the components are specified in the type definition; they can be scalars or arrays. Each structure has at least one component. There can be arrays of structures. In example 2 (see Section 6.2, page 187), `FIRECHIEF` is a structure; `FIREMEN` is an array of structures of type `PERSON`.

A component of a structure is referenced by placing the name of the component after the name of the parent structure, separated by a percent sign (`%`). For example, `FIRECHIEF % NAME` references the character string component of the variable `FIRECHIEF` of type `PERSON`.

A structure component is a data reference and is defined as follows:

data_ref	**is**	*part_ref* [% *part_ref*] ...
part_ref	**is**	*part_name* [(*section_subscript_list*)]
structure_component	**is**	*data_ref*

For a data reference to be considered a structure component reference, there must be more than one part reference and the rightmost part reference must be a part name. If the rightmost component is of the following form, the reference is considered to be an array section or array element (the simplest form of a section subscript list is a subscript list):

part_name (*section_subscript_list*)

The rules for forming a *section_subscript_list* and a *subscript* are provided in Section 6.4, page 191.

In a data reference, each part name except the rightmost must be of derived type.

In a data reference, each part name except the leftmost must be the name of a component of the derived-type definition of the type of the preceding part name.

In a part reference containing a section subscript list, the number of section subscripts must equal the rank of the part name.

It is possible to create a structure with more than one array part, but in a data reference to the structure, there must not be more than one part reference with nonzero rank.

In a data reference, a part name to the right of a part reference with nonzero rank must not have the POINTER attribute. It is possible to declare an array of structures that have a pointer as a component, but it is not possible to reference such an object as an array.

The rank of a part reference consisting of just a part name is the rank of the part name. The rank of a part reference of the following form is the number of subscript triplets and vector subscripts in the list:

part_name (*section_subscript_list*)

The rank is less than the rank of the part name if any of the section subscripts are subscripts other than subscript triplets or vector subscripts. The shape of a data reference is the shape of the part reference with nonzero rank, if any; otherwise, the data reference is a scalar and has rank zero.

The *parent structure* in a data reference is the data object specified by the leftmost part name. If the parent object has the INTENT, TARGET, or PARAMETER attribute, the structure component has the attribute. The type and type parameters of a structure component are those of the rightmost part name. A structure component is a pointer only if the rightmost part name has the POINTER attribute. Typically, an object cannot have both the TARGET and POINTER attributes. However, a structure that has the TARGET attribute can contain components that have the POINTER attribute.

Example 1: Assume the following type definition and structure declarations:

```
TYPE PERSON
   INTEGER AGE
   CHARACTER(LEN = 40) NAME
END TYPE PERSON
. . .
TYPE(PERSON) FIRECHIEF, FIREMEN(50)
```

In this example, structure components are as follows:

```
FIRECHIEF%AGE    ! scalar component of scalar parent
FIREMEN(J)%NAME  ! component of array element parent
```

```
FIREMEN(1:N)%AGE ! component of array section parent
```

Example 2: If a derived-type definition contains a component that is of derived type, then a reference to an ultimate component can contain more than two part references as do the references in the first two PRINT statements in the following example:

```
TYPE REPAIR_BILL
   REAL PARTS
   REAL LABOR
END TYPE REPAIR_BILL
. . .
TYPE VEHICLE
   CHARACTER(LEN = 40) OWNER
   INTEGER MILEAGE
   TYPE(REPAIR_BILL) COST
END TYPE VEHICLE
. . .
TYPE(VEHICLE) BLACK_FORD, RED_FERRARI
. . .
PRINT *, BLACK_FORD%COST%PARTS
PRINT *, RED_FERRARI%COST%LABOR
PRINT *, RED_FERRARI%OWNER
```

6.4 Arrays

An *array* is a collection of scalar elements of any intrinsic or derived type. An object of any type that is specified to have the DIMENSION attribute is an array. All of the elements of an array must have the same type and kind parameter. There can be arrays of structures. The value returned by a function can be an array. The appearance of an array name or designator has no implications for the order in which the individual elements are referenced unless array element ordering is specifically required.

6.4.1 Array Terminology

An array consists of elements that extend in one or more dimensions to represent columns, rows, planes, and so on. There can be up to seven dimensions in an array declaration. The number of dimensions in an array is called the *rank* of the array. The number of elements in a dimension is called the *extent* of the array in that dimension. The *shape* of an array is determined from the rank and the extents; to be precise, the

shape is a vector where each element of the vector is the extent in the corresponding dimension. The *size* of an array is the product of the extents; that is, it is the total number of elements in the array.

Example 1: Consider the following statement:

```
REAL X(0:9, 2)
```

The rank of `X` is `2` because `X` has two dimensions. The extent of the first dimension is `10`; the extent of the second dimension is `2`. The shape of `X` is `10` by `2`, that is, a vector of two values, `(10, 2)`. The size is `20`, the product of the extents.

An object can be given the `DIMENSION` attribute in a type declaration or in one of several other specification statements.

Example 2: Consider the following statements show some ways of declaring that `A` has rank `3` and shape `(10, 15, 3)`:

```
DIMENSION A(10, 15, 3)
REAL, DIMENSION(10, 15, 3) :: A
REAL A(10, 15, 3)
COMMON A(10, 15, 3)
TARGET A(10, 15, 3)
```

Arrays of nonzero size have a lower and upper bound for each dimension. The *lower bound* is the smallest subscript value for a dimension; the *upper bound* is the largest subscript value for that dimension. The default lower bound is 1 if the lower bound is omitted in the declaration. Array bounds can be positive, zero, or negative.

Example 3: Consider the following statement:

```
REAL Z(-3:10, 12)
```

The first dimension of `Z` ranges from `-3` to `10`, that is, `-3`, `-2`, `-1`, `0`, `1`, `2`, . . ., `9`, `10`. The lower bound is `-3`; the upper bound is `10`. In the second dimension, the lower bound is `1`; the upper bound is `12`.

The bounds for array expressions are described in Section 7.2.8.4, page 247.

6.4.2 Whole Arrays

Some arrays are named. The name is either an array variable name or the name of a constant. If the array name appears without a subscript list or section subscript list,

all of the elements of the array are referenced and the reference is considered to be a *whole array* reference.

6.4.3 Array Elements

An *array element* is one of the scalar elements that make up an array. A subscript list is used to indicate which element is referenced. Assume that `A` is declared to be a one-dimensional array, as follows:

```
REAL, DIMENSION(10) :: A
```

`A(1)` refers to the first element, `A(2)` to the second, and so on. The number in the parentheses is the subscript that indicates which scalar element is referenced. Assume that `B` is declared to be a seven-dimensional array, as follows:

```
REAL B(5, 5, 5, 5, 4, 7, 5)
```

`B(2,3,5,1,3,7,2)` refers to one scalar element of `B`, indexed by a subscript in each dimension. The set of numbers that indicate the position along each dimension in turn (in this case, `2,3,5,1,3,7,2`) is called a *subscript list*.

6.4.4 Array Sections

Sometimes only a portion of an array is needed for a calculation. It is possible to refer to a selected portion of an array as an array; this portion is called an *array section*. A *parent array* is the whole array from which the portion that forms the array section is selected.

An array section is specified by an array variable name and a section subscript list that consists of subscripts, triplet subscripts, or vector subscripts. At least one subscript must be a triplet or vector subscript; otherwise, the reference indicates an array element, not an array section. The following example uses a section subscript to create an array section:

```
REAL A(10)
   . . .
A(2:5) = 1.0
```

The parent array `A` has 10 elements. The array section consists of the elements `A(2)`, `A(3)`, `A(4)`, and `A(5)` of the parent array. The section `A(2:5)` is an array itself and the value `1.0` is assigned to all four of its elements.

6.4.5 Format of Array Elements and Array Sections

The format of an array element is a data reference, which is defined as follows:

array_element	**is**	*data_ref*

For a data reference to be classified as an array section, exactly one part reference must have nonzero rank, and either the final part reference must have a section subscript list with nonzero rank or another part reference must have nonzero rank.

The format of an array section is a data reference followed by an optional substring range enclosed in parentheses. This is defined as follows:

array_section	**is**	*data_ref* [(*substring_range*)]

The format of a substring range is found in Section 6.2, page 187.

A part name in a data reference can be followed by an optional section subscript list, as follows:

subscript	**is**	*scalar_int_exp*
section_subscript	**is**	*subscript*
	or	*subscript_triplet*
	or	*vector_subscript*
subscript_triplet	**is**	[*subscript*] : [*subscript*] [: *stride*]
stride	**is**	*scalar_int_expr*
vector_subscript	**is**	*int_expr*

Each *subscript* and *stride* must be a *scalar_int_expr*. A *vector_subscript* must be an integer array expression of rank one.

For a data reference to be classified as an array element, every part reference must have rank zero and the last part reference must contain a subscript list.

In an array section that is a data reference followed by a substring range, the rightmost part name must be of type character.

In an array section of an assumed-size array, the second subscript must not be omitted from a subscript triplet in the last dimension.

A section subscript must be present for each dimension of an array. If any section subscript is simply a subscript, the section will have a lesser rank than its parent.

If any part of a reference is an array section, the reference is considered to be an array section reference. In a data reference, there may be at most one part with rank greater than zero.

Examples of array elements and array sections are as follows:

```
ARRAY_A(1,2)        ! array element
ARRAY_A(1:N:2,M)    ! rank-one array section
ARRAY_B(:,:,:)(2:3)  ! array. elements are substrings of
                     ! of length 2
SCALAR_A%ARRAY_C(L)                    ! array element
SCALAR_A%ARRAY_C(1:L)                  ! array section
SCALAR_B%ARRAY_D(1:N)%SCALAR_C         ! array section
ARRAY_E(1:N:2)%ARRAY_F(I,J)%STRING(K)(:) ! array section
```

In the last example above, each component of the type definition is an array and the object ARRAY_E is an array. The reference is valid because each component in the reference is scalar. The substring range is not needed because it specifies the entire string; however, it serves as a reminder that the last component is of type character.

The following examples demonstrate the allowable combinations of scalar and array parents with scalar and array components.

```
TYPE REPAIR_BILL
   REAL PARTS(20)
   REAL LABOR
END TYPE REPAIR_BILL

TYPE(REPAIR_BILL) FIRST
TYPE(REPAIR_BILL) FOR_1990(6)
```

Scalar parent:

```
 1. FIRST % LABOR          ! structure component
 2. FIRST % PARTS(I)       ! array element
 3. FIRST % PARTS          ! component (array-valued)
```

```
 4. FIRST % PARTS(I:J)     ! array section

 5. FOR_1990(K) % LABOR    ! structure component
 6. FOR_1990(K) % PARTS(I) ! array element
 7. FOR_1990(K) % PARTS    ! component (array-valued)
 8. FOR_1990(K) % PARTS(I:J) ! array section
```

Array parent:

```
 9. FOR_1990 % LABOR       ! component and array section
10. FOR_1990 % PARTS(I)   ! array section
11. FOR_1990 % PARTS      ! ILLEGAL
12. FOR_1990 % PARTS(I:J) ! ILLEGAL

13. FOR_1990(K:L) % LABOR ! component and array section
14. FOR_1990(K:L) % PARTS(I) ! array section
15. FOR_1990(K:L) % PARTS ! ILLEGAL
16. FOR_1990(K:L) % PARTS(I:J) ! ILLEGALText goes here
```

References `11`, `12`, `15`, and `16` are illegal because only one component may be of rank greater than zero. References `3` and `7` are compact (contiguous) array objects and are classified as array-valued structure components. References `9`, `10`, `13`, and `14` are noncontiguous array objects and are classified as sections. These distinctions are important when such objects are actual arguments in procedure references.

6.4.5.1 Subscripts

In an array element reference, each subscript must be within the bounds for that dimension. A subscript can appear in an array section reference. Whenever this occurs, it decreases the rank of the section by one less than the rank of the parent array. A subscript used in this way must be within the bounds for the dimension. The compiler allowss overindexing, which may produce incorrect results because of optimization. For information on overindexing and optimization, see the *MIPSpro Fortran Language Reference Manual, Volume 3*.

ANSI/ISO: The Fortran standard does not address overindexing.

6.4.5.2 Subscript Triplets

The first subscript in a subscript triplet is the lower bound; the second is the upper bound. If the lower bound is omitted, the declared lower bound is used. If the upper

bound is omitted, the declared upper bound is used. The *stride* is the increment between successive subscripts in the sequence. If it is omitted, it is assumed to be 1. The stride must not be 0. If the subscripts and stride are omitted and only the colon (`:`) appears, the entire declared range for the dimension is used.

When the stride is positive, an increasing sequence of integer values is specified from the first subscript in increments of the stride, up to the last value that is not greater than the second subscript. The sequence is empty if the first subscript is greater than the second. If any subscript sequence is empty, the array section is a zero-sized array, because the size of the array is the product of its extents. For example, given the array declared `A(5,4,3)` and the section `A(3:5,2,1:2)`, the array section is of rank `2` with shape `(3,2)` and size `6`. The elements are as follows:

```
A(3,2,1)     A(3,2,2)
A(4,2,1)     A(4,2,2)
A(5,2,1)     A(5,2,2)
```

When the stride is negative, a decreasing sequence of integer values is specified from the first subscript, in increments of the stride, down to the last value that is not less than the second subscript. The sequence is empty if the second subscript is greater than the first, and the array section is a zero-sized array. For example, given the array declared `B(10)` and the section `B (9:4:-2)`, the array section is of rank `1` with shape `(3)` and size `3`. The elements are as follows:

```
B(9) B(7) B(5)
```

However, array section `B(9:4)` is a zero-sized array.

A subscript in a subscript triplet is not required to be within the declared bounds for the dimension as long as all subscript values selected by the triplet are within the declared bounds. For example, given an array declared `B(10)`, section `B(3:11:7)` is permitted. It has rank `1` with shape `(2)` and size `2`. The elements are as follows:

```
B(3) B(10)
```

6.4.5.3 Vector Subscripts

While subscript triplets specify values in increasing or decreasing order with a specified stride to form a regular pattern, vector subscripts specify values in arbitrary order. The values must be within the declared bounds for the dimension. A vector subscript is a rank-one array of integer values used as a section subscript to select elements from a parent array.

```
INTEGER J(3)
REAL A(30)
   . . .
J = (/ 8, 4, 7 /)
A(J) = 1.0
```

The last assignment statement assigns the value `1.0` to `A(4)`, `A(7)`, and `A(8)`. The section `A(J)` is a rank-one array with shape `(3)` and size `3`.

If `J` were assigned `(/ 4, 7, 4 /)` instead, the element `A(4)` would be accessed in two ways: as `A(J(1))` and as `A(J(3))`. Such an array section is called a many-to-one array section. A many-to-one section must not appear on the left of the equal sign in an assignment statement or as an input item in a `READ` statement. The reason is that the result will depend on the order of evaluation of the subscripts, which is not specified by the language. The results would not be predictable and the program containing such a statement would not be portable.

Array sections with vector subscripts are array expressions, so there are places, such as the following, where array sections with vector subscripts must not appear:

- As internal files
- As pointer targets
- As actual arguments for `INTENT(OUT)` or `INTENT(INOUT)` dummy arguments

6.4.6 Using Array Elements and Array Sections

Subscripts, subscript triplets, and vector subscripts can be mixed in a single section subscript list used to specify an array section. A triplet section can specify an empty sequence (for example `1:0`), in which case the resulting section is a zero-sized array.

Example 1: Assume that `B` is declared as follows:

```
REAL B(10, 10, 5)
```

Section `B(1:4:3, 6:8:2, 3)` consists of the following four elements:

```
B(1,6,3)     B(1,8,3)
B(4,6,3)     B(4,8,3)
```

The stride along the first dimension is `3`, resulting in a subscript-value list of `1` and `4`. The stride along the second dimension is `2`, resulting in a subscript-value list of `6` and

8. In the third position there is a subscript that reduces the rank of the section by 1. The section has shape (2, 2) and size 4.

Example 2: Assume IV is declared as follows:

```
INTEGER, DIMENSION(3) :: IV = (/ 4, 5, 4 /)
```

Then the section B(8:9, 5, IV) is a 2 x 3 array consisting of the following six elements:

```
B(8,5,4)     B(8,5,5)     B(8,5,4)
B(9,5,4)     B(9,5,5)     B(9,5,4)
```

B(8:9, 5:4, IV) is a zero-sized array of rank 3.

6.4.7 Array Element Order

When whole arrays are used as operands in an executable statement, the indicated operation is performed element-by-element, but no order is implied for these elemental operations. They can be executed in any order or simultaneously. Although there is no order of evaluation when whole array operations are performed, there is an ordering of the elements in an array itself. An ordering is required for the input and output of arrays and for certain intrinsic functions such as MAXLOC(3i). The elements of an array form a sequence whose ordering is called *array element order*. This is the sequence that occurs when the subscripts along the first dimension vary most rapidly, and the subscripts along the last dimension vary most slowly. Thus, for an array declared as REAL A(3, 2), the elements in array element order are: A(1, 1), A(2, 1), A(3, 1), A(1, 2), A(2, 2), A(3, 2).

The position of an array element in this sequence is its *subscript order value*. Element A(1, 1) has a subscript order value of 1. Element A(1, 2) has a subscript order value of 4. Figure 6-1, page 200, shows how to compute the subscript order value for any element in arrays of rank 1 through 7.

Rank	Explicit shape specifier	Subscript list	Subscript order 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	$1 + (s_1 - j_1) + (s_2 - j_2) \times d_1$
3	$j_1{:}k_1, j_2{:}k_2 , j_3{:}k_3$	s_1, s_2, s_3	$1 + (s_1 - j_1) + (s_2 - j_2) \times d_1 + (s_3 - j_3) \times d_2 \times d_1$
.	.	.	.
.	.	.	.
.	.	.	.
7	$j_1{:}k_1, \ldots, j_7{:}k_7$	$s_1, \ldots, s_7$	$1 + (s_1 - j_1) + (s_2 - j_2) \times d_1 + (s_3 - j_3) \times d_2 \times d_1 + \ldots + (s_7 - j_7) \times d_6 \times d_5 \times \ldots \times d_1$

a10879

Figure 6-1 Computation of subscript order value

The subscript order of the elements of an array section is that of the array object that the section represents. That is, given the array `A(10)` and the section `A(2:9:2)` consisting of the elements `A(2)`, `A(4)`, `A(6)`, and `A(8)`, the subscript order value of `A(2)` in the array section `A(2:9:2)` is `1`; the subscript order value of `A(4)` in the section is `2` and `A(8)` is `4`.

6.5 Pointers and Allocatable Arrays

Fortran provides several dynamic data objects. Automatic objects (arrays and character strings) are discussed in Section 5.8, page 167. In addition, there are two data attributes that can be used to specify dynamic data objects: `ALLOCATABLE` and `POINTER`. Arrays of any type can have the `ALLOCATABLE` attribute; scalars or arrays of any type can have the `POINTER` attribute. Chapter 5, page 115, described how such objects are declared. This section describes how space is created for these objects with the `ALLOCATE` statement, how it can be released with the `DEALLOCATE` statement, and how pointers can be disassociated from any target with the `NULLIFY` statement. The association status of a pointer can be defined or undefined; initially (when a pointer is declared), it is undefined. If it is defined, the pointer can be associated with a target or disassociated from any target. The target is referenced by the name of the pointer and is like any other variable in that it is defined when it acquires a value. Figure 6-2 shows the various states that a pointer may assume.

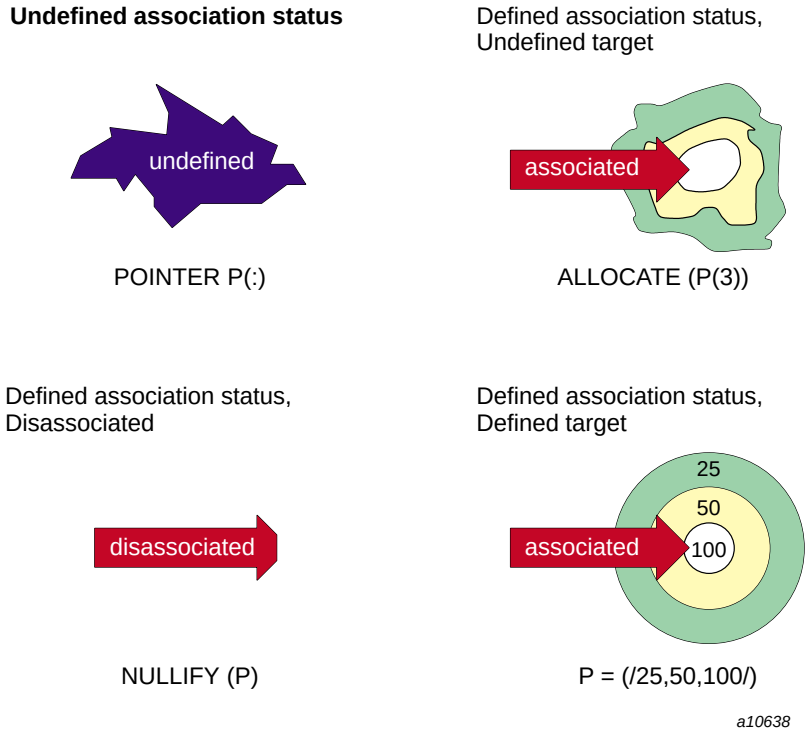

Figure 6-2 States in the lifetime of a pointer

Section 7.5.3, page 280, describes how pointers can be associated with existing space and how dynamic objects can acquire values.

6.5.1 ALLOCATE Statement

The ALLOCATE statement creates space for the following:

- Arrays with the ALLOCATABLE attribute
- Variables with the POINTER attribute

The ALLOCATE statement is defined as follows:

allocate_stmt	**is**	ALLOCATE (*allocation_list* [, STAT = *stat_variable*])
stat_variable	**is**	*scalar_int_variable*
allocation	**is**	*allocate_object* [(*allocate_shape_spec_list*)]
allocate_object	**is**	*variable_name*
	or	*structure_component*
allocate_shape_spec	**is**	[*allocate_lower_bound* :] *allocate_upper_bound*
allocate_lower_bound	**is**	*scalar_int_expr*
allocate_upper_bound	**is**	*scalar_int_expr*

The *stat_variable*, *allocate_lower_bound*, and *allocate_upper_bound* must each be scalar integer expressions.

Each *allocate_object* must be a pointer or an allocatable array.

An attempt to allocate space for an allocatable array that is currently allocated results in an error condition.

An *allocate_object* or a subobject of an *allocate_object* cannot appear in a bound in the same allocate statement.

If a STAT= variable appears, it must not be allocated in the same ALLOCATE statement. A STAT= variable cannot depend on the value, bounds, allocation status, or association status of any *allocate_object* or subobject of an *allocate_object* allocated in the same statement. It is set to zero if the allocation is successful and is set to a

positive value if there is an error condition; when an error is detected, subsequent items in the *allocation_list* are not allocated. If there is no `STAT=` variable, the program terminates when an error condition occurs.

You can obtain an online explanation of an error identified by the `STAT=` return value. To do this, join the returned value with its group name, shown in the following list, and use the resulting string as an argument to the `explain`(1) command. For example:

```
explain lib-5000
explain 90476
```

Table 6-1 Message number identifiers

Message number	Group name	Source of message
1 through 899	`sys`	Operating system
4000 through 4999	`lib`	Fortran library (IRIX systems)
5000 through 5999	`lib`	Flexible File I/O (FFIO) library
90000 through 90500	None	Tape system

An argument to an inquiry function in an `ALLOCATE` statement must not appear as an allocate object in that statement. For example, the use of the intrinsic inquiry function `SIZE` in the following code is not permitted.

```
REAL, ALLOCATABLE :: A(:), B(:)
ALLOCATE (A(10), B(SIZE(A)))
```

The number of allocate shape specifications must agree with the rank of the array.

If the lower bound is omitted, the default is 1. If the upper bound is less than the lower bound, the extent in that dimension is 0 and the array has zero size.

An allocate object can be of type character and it can have a length of 0, in which case no memory is allocated.

The values of the bounds expressions at the time an array is allocated determine the shape of the array. If an entity in a bounds expression is subsequently redefined, the shape of the allocated array is not changed.

6.5.1.1 Allocation of Allocatable Arrays

You must declare the rank of an allocatable array, but the bounds, extents, shape, and size are determined when the array is allocated. After allocation the array can be defined and redefined. The array then is said to be *currently allocated*. It is an error to allocate an allocatable array that is already allocated. The intrinsic function ALLOCATED(3i) can be used to query the allocation status of an allocatable array if the allocation status is defined, for example:

```
REAL, ALLOCATABLE :: X(:, :, :)
   . . .
IF (.NOT. ALLOCATED(X)) ALLOCATE (X(-6:2, 10, 3))
```

X is not available for use in the program until it has been allocated space by an ALLOCATE statement. X must be declared with a deferred-shape array specification and the ALLOCATABLE attribute.

6.5.1.2 Allocation of Pointers

When an object with the POINTER attribute is allocated, space is created, and the pointer is associated with that space, which becomes the pointer target. A reference to the pointer name can be used to define or access its target. The target can be an array or a scalar. Additional pointers can become associated with the same target by pointer assignment (described in Section 7.5.3, page 280). A pointer target can be an array with the ALLOCATABLE attribute if the array also has the TARGET attribute. Allocation of a pointer creates an object that implicitly has the TARGET attribute.

It is not an error to allocate a pointer that is currently associated with a target. In this case, a new pointer target is created and the previous association of the pointer is lost. If there was no other way to access the previous target, it becomes inaccessible.

The ASSOCIATED(3i) intrinsic function can be used to query the association status of a pointer if the association status of the pointer is defined. The ASSOCIATED(3i) intrinsic function also can be used to inquire whether a pointer is associated with a target or whether two pointers are associated with the same target.

Pointers can be used in many ways; an important usage is creating linked lists. For example:

```
TYPE NODE
   INTEGER :: VALUE
   TYPE(NODE), POINTER :: NEXT
END TYPE NODE
```

```
TYPE(NODE), POINTER :: LIST
    . . .
ALLOCATE(LIST)
LIST%VALUE = 17
ALLOCATE(LIST%NEXT)
```

The first two executable statements create a node pointed to by LIST and put the value 17 in the VALUE component of the node. The third statement creates a second node pointed to by the NEXT component of the first node.

6.5.2 NULLIFY Statement

The NULLIFY statement causes a pointer to be disassociated from any target. Pointers have an initial association status that is undefined. To initialize a pointer to point to no target, it is necessary to execute a NULLIFY statement for the pointer.

The NULLIFY statement is defined as follows:

nullify_stmt	**is**	NULLIFY (*pointer_object_list*)
pointer_object	**is**	*variable_name*
	or	*structure_component*

Each *pointer_object* must have the POINTER attribute.

A *pointer_object* cannot depend on the value, bounds, or association status of another *pointer_object* or subobject of another *pointer_object* in the same NULLIFY statement.

6.5.3 DEALLOCATE Statement

The DEALLOCATE statement releases the space allocated for an allocatable array or a pointer target and nullifies the pointer. After an allocatable array or pointer has been deallocated, it cannot be accessed or defined until it is allocated again or, in the case of a pointer, assigned to an existing target.

In some cases, the execution of a RETURN statement in a subprogram may cause the association status of a pointer to become undefined. This can be avoided if the pointer is given the SAVE attribute or if it is declared in a subprogram that remains

active. (The main program is always active. Variables declared in modules accessed by the main program and named common blocks specified in the main program do not need to be given the SAVE attribute; these entities have the attribute automatically. If the main program calls subroutine A and subroutine A calls function B, then the main program, subroutine A, and function B are active until a return from function B is executed, at which time only the main program and subroutine A are active. If a recursive subprogram becomes active, it remains active until the return from its first invocation is executed.)

The DEALLOCATE statement is defined as follows:

deallocate_stmt **is** DEALLOCATE (*allocate_object_list* [, STAT = *stat_variable*])

The *stat_variable* must be a scalar integer variable.

Each *allocate_object* must be a pointer or an allocatable array.

An *allocate_object* must not depend on the value, bounds, allocation status, or association status of another *allocate_object* or subobject of another *allocate_object* in the same DEALLOCATE statement, nor can it depend on the value of the *stat_variable* in the same DEALLOCATE statement.

If there is a STAT= variable and it is a pointer, it must not be deallocated in the same DEALLOCATE statement, nor can it depend on the value, bounds, allocation status, or association status of any *allocate_object* or subobject of an *allocate_object* in the same DEALLOCATE statement. The *stat_variable* is set to zero if the deallocation is successful and is set to a positive value if there is an error condition. If STAT=*stat_variable* is not specified, the program terminates when an error condition occurs.

You can obtain an online explanation of an error identified by the STAT= return value. See Section 6.5.1, page 202 for information about obtaining online explanations.

Table 6-2 Message number identifiers

Message number	Group name	Source of message
1 through 899	sys	Operating system
4000 through 4999	lib	Fortran library (IRIX systems)

Message number	Group name	Source of message
5000 through 5999	lib	Flexible File I/O (FFIO) library
90000 through 90500	None	Tape system

6.5.3.1 Deallocation of Allocatable Arrays

To be deallocated, an allocatable array must be currently allocated; otherwise, an error condition will occur. You can use the inquiry intrinsic function ALLOCATED(3i) to determine if an array is currently allocated.

An allocatable array can have the TARGET attribute. If such an array is deallocated, the association status of any pointer associated with the array will become undefined. Such an array must be deallocated by the appearance of its name in a DEALLOCATE statement. It must not be deallocated by the appearance of the pointer name in a DEALLOCATE statement.

When a RETURN or END statement is executed in a subprogram, local allocatable arrays are deallocated unless any of the following conditions exist:

- The array has the SAVE attribute
- The array is specified in a module that is accessed by an active subprogram
- The array is accessed by host association

The following is an example of the allocation and deallocation of an allocatable array:

```
REAL, ALLOCATABLE :: X(:, :)
   . . .
ALLOCATE (X(10, 2), STAT = IERR)
IF (IERR .GT. 0) CALL HANDLER
X = 0.0
   . . .
DEALLOCATE (X)
   . . .
ALLOCATE (X(-10:10, 5), STAT = JERR)
```

X is declared to be a deferred-shape, two-dimensional, real array with the ALLOCATABLE attribute. Space is allocated for it and it is given bounds, extents, shape, and size and then initialized to have zero values in all elements. Later X is deallocated, and still later, it is again allocated with different bounds, extents, shape, and size, but its rank remains as declared.

6.5.3.2 Deallocation of Pointers

Only a pointer with defined association status can be deallocated. Deallocating a pointer with an undefined association status or a pointer associated with a target that was not created by allocation causes an error condition in the DEALLOCATE statement. A pointer associated with an allocatable array must not be deallocated. (Of course, the array itself can be deallocated.)

It is possible (by pointer assignment) to associate a pointer with a portion of an object such as an array section, an array element, or a substring. A pointer associated with only a portion of an object cannot be deallocated. If more than one pointer is associated with an object, deallocating one of the pointers causes the association status of the others to become undefined. Such pointers must not be arguments to the ASSOCIATED(3i) inquiry function.

When a RETURN or END statement is executed in a procedure, the association status of a pointer declared or accessed in the procedure becomes undefined unless one of the following conditions is true:

- The pointer has the SAVE attribute
- The pointer is specified in a module that is accessed by an active subprogram
- The pointer is accessed by host association
- The pointer is in blank common
- The pointer is in a named common block that is specified in an active subprogram or has the SAVE attribute
- The pointer is the return value of a function declared to have the POINTER attribute

If the association status of a pointer becomes undefined, the pointer can no longer be referenced, defined, or deallocated. It can be allocated, nullified, or pointer-assigned to a new target.

The following is an example of the allocation and deallocation of a pointer:

```
REAL, POINTER :: X(:, :)
   . . .
ALLOCATE (X (10, 2), STAT = IERR)
IF (IERR .GT. 0) CALL HANDLER
X = 0.0
   . . .
```

```
DEALLOCATE (X)
   . . .
ALLOCATE (X(-10:10, 5), STAT = JERR)
```

X is declared to be a deferred-shape, two-dimensional, real array with the POINTER attribute. Space is allocated for it and it is given bounds, extents, shape, and size and then initialized to have zero values in all elements. Later X is deallocated, and still later, it is again allocated with different bounds, extents, shape, and size. This example is quite similar to the previous example for allocatable arrays, except that, in the case of pointers, it is not necessary to deallocate X before allocating it again.

Expressions and Assignments

In Fortran, calculations are specified by writing expressions. Expressions look much like algebraic formulas in mathematics, particularly when the expressions involve calculations on numerical values.

Expressions often involve nonnumeric values, such as character strings, logical values, or structures; these also can be considered to be formulas that involve nonnumeric quantities rather than numeric ones.

This chapter describes how valid expressions can be formed, how they are interpreted, and how they are evaluated. One of the major uses of expressions is in assignment statements where the value of an expression is assigned to a variable. The assignment statement appears in four forms: intrinsic assignment, defined assignment, masked array assignment, and pointer assignment. In the first three forms, a value is computed by performing the computation specified in an expression and the value is assigned to a variable. In the fourth form, a pointer, the object on the left side, is made to point to the object or target on the right side. The four forms of the assignment statement are also described in this chapter.

Note: The Fortran statement syntax in this manual is defined using the same terms used in the Fortran standard. In this chapter, however, certain terms from the standard have been changed to improve clarity.

7.1 Introduction to Expressions

Fortran allows you to define data types, operators for these types, and operators for intrinsic types. These capabilities are provided within the general framework for expressions, which consists of three sets of rules:

- The rules for forming a valid expression
- The rules for interpreting the expression (giving it a meaning)
- The rules for evaluating the expression (how the computation may be carried out)

An expression is formed from operators and operands. There is no change from FORTRAN 77 in the rules for forming expressions, except that a new class of operators has been defined. These are user-defined operators, which are either unary

or binary operators. They have the form of a sequence of letters surrounded by periods; `.INVERSE.` and `.PLUS.` are examples of possible user-defined operators.

The formal (BNF) rules for forming expressions imply an order for combining operands with operators. These rules specify that expressions enclosed in parentheses are combined first and that, for example, the multiply operator `*` is combined with its operands before the addition operator `+` is combined with its operands. This order for operators in the absence of specific parentheses is called the *operator precedence* and is summarized in Section 7.2.5, page 236. New operators, such as `==` and `>=`, have the same precedence and meaning as `.EQ.` and `.GE.`, respectively.

The formation rules for expressions imply that the defined unary operators have highest precedence of all operators, and defined binary operators have the lowest precedence of all operators. When they appear in a context where two or more of these operators of the same precedence are adjacent, the operands are combined with their operators in a left-to-right manner, as is the case for the familiar `+` and `-` operators, or in a right-to-left manner for the exponentiation operator (`**`).

Intrinsic operators are generic in the sense that they can operate on operands of different types. For example, the plus operator `+` operates on operands of type integer as well as real and complex. Intrinsic operators can be extended further by the programmer to operate on operands of types for which there are no intrinsic operations. Similarly, you can use defined unary and defined binary operators to operate on operands of types for which there are no previous definitions. The *MIPSpro Fortran Language Reference Manual, Volume 2*, describes how any operator can be made generic by the programmer using a generic specifier on an interface block.

The rules for interpretation of an expression are provided by the interpretation of each operator in the expression. When the operator is an intrinsic operator such as `+`, `*`, or `.NOT.`, and the operands are of intrinsic types allowed for the intrinsic operator, the interpretation is provided by the usual mathematical or symbolic meaning of the operation. Thus, `+` with two numeric operands means that the two operands are added together. For the user-defined operators, the interpretation is provided by a user-supplied function subprogram with a designation that this subprogram is to be used to define the operation. Fortran allows the intrinsic operator symbols to be extended to cases in which the operands are not of the usual intrinsic types defined by the standard. For example, the `+` operator can be defined for operands of type `RATIONAL` (a user-defined type) or for operands of type logical with the interpretation provided by a user-supplied function subprogram. The rules for construction of expressions (the syntax rules) are the same for user-defined operators as for intrinsic operators.

The general rule for evaluation of a Fortran expression states that any method that is mathematically equivalent to that provided by the construction and interpretation rules for the expression is permitted, provided the order of evaluation indicated by explicit parentheses in the expression is followed. Thus, a compiler has a great deal of freedom to rearrange or optimize the computation, provided the rearranged expression has the same mathematical meaning.

Arrays and pointers as objects can appear in expressions and assignment statements. This chapter describes using arrays and pointers in the following contexts:

- As operands of intrinsic and user-defined operations
- As the variables being assigned in intrinsic assignment statements
- As the variables in pointer assignment statements and masked array assignment statements

7.1.1 Assignment

The result obtained from the evaluation of an expression can be used in many ways. For example, it can be printed or passed to a subprogram. In many cases, however, the value is assigned to a variable and that value can be used later in the program by referencing the variable.

Execution of the assignment statement causes the expression to be evaluated (by performing the computation indicated), and then the value of the expression is assigned to the variable on the left of the equal sign.

The following example shows an assignment statement:

```
REAL_AGE = REPORTED_AGE + 3.0
```

`REPORTED_AGE + 3.0` is the expression that indicates how to compute a value, which is assigned to the variable `REAL_AGE`.

The following example involves subscripts:

```
A(I+3) = PI + A(I-3)
```

The value of the subscript expression `I-3` is determined and the value of the `I-3` element of `A` is added to the value of `PI` to produce a sum. Before the result of this expression is assigned, the value of the subscript expression `I+3` is determined and the value of the sum is assigned to the element `I+3` of `A`.

The previous examples are arithmetic. Fortran has expressions of other types, such as logical, character, and derived type. Values of expressions of these other types can be assigned to variables of these other types. As with operators, the programmer can extend the meaning of assignment to types not defined intrinsically and can redefine assignment for two objects of the same derived type. Such assignments are called *defined assignments*. In addition, arrays and pointers each have special forms of assignment statements called *masked array assignment* and *pointer assignment*, respectively. These two assignment statement forms are described later in this chapter.

7.1.2 Expressions

An assignment statement is only one of the Fortran statements in which expressions may occur. Expressions also can appear in subscripts, actual arguments, `IF` statements, `PRINT` statements, `WHERE` statements, declaration statements, and many other statements.

An expression represents a computation that results in a value and can be as simple as a constant or variable. The value of an expression has a type and can have zero, one, or two type parameter values. If the value is of a derived type, it has no type parameter. If it is of an intrinsic type, it has a kind type parameter, and if, in addition, it is of the type character, it has a character length parameter. In addition, the value is a scalar (including a structure) or an array.

A complex value or a structure value is a scalar, even though it can consist of more than one value (for example, a complex value consists of two real values).

Arrays and pointers can be used as operands of intrinsic and defined operators. For intrinsic operators, when an array is an operand, the operation is performed element-wise on the elements of the array. For intrinsic operators, when a pointer is an operand, the value of the target pointed to by (associated with) the pointer is used as the operand. For defined operators, the array or pointer is used in a manner determined by the procedure defining the operation.

As indicated in the introduction to this chapter, the presentation of expressions is described in terms of the following three basic sets of rules:

- The rules for forming expressions (syntax)
- The rules for interpreting expressions (semantics)
- The rules for evaluating expressions (optimization)

The syntax rules indicate which forms of expressions are valid. The semantics indicate how each expression is to be interpreted. After an expression has been given an interpretation, a compiler can evaluate another completely different expression, provided the expression evaluated is mathematically equivalent to the one written.

To see how this works, consider the expression `2 * A + 2 * B` in the following `PRINT` statement:

```
PRINT *, 2 * A + 2 * B
```

The syntax rules described later in this chapter indicate that the expression is valid and suggest an order of evaluation. The semantic rules specify the operations to be performed, which in this case, are the multiplication of the values of `A` and `B` by `2` and the addition of the two results. That is, the semantic rules indicate that the expression is to be interpreted as if it were the following:

```
((2 * A) + (2 * B))
```

After the correct interpretation has been determined, the Fortran rules of evaluation allow a different expression to be used to evaluate the expression, provided the different expression is mathematically equivalent to the one written. For example, a processor can first add `A` and `B` and then multiply the result by `2`, because the following expression is mathematically equivalent to the one written:

```
2 * (A + B)
```

Although alternative evaluations are allowed, three properties should be noted:

- Parentheses must not be violated. Consider the following expression:

  ```
  (2 * A) + (2 * B)
  ```

  This expression must **not** be evaluated as follows:

  ```
  2 * (A + B)
  ```

  This gives the programmer some control over the method of evaluation.

- Integer division is not mathematically equivalent to real division. The value of `3/2` is `1` and so cannot be evaluated as `3*0.5`, which is `1.5`.

- Mathematically equivalent expressions can produce computationally different results because of the implementation of arithmetic and rounding on computer systems. For example, the expression `X/2.0` could be evaluated as `0.5*X`, even though the results may be slightly different. Also, for example, the expression `2 * A + 2 * B` could be evaluated as `2*(A+B)`; when `A` and `B` are of type real, the

two mathematically equivalent expressions may yield different values because of different rounding errors and different arithmetic exceptions in the two expressions.

7.2 Formation of Expressions

An expression is formed from operands, operators, and parentheses. The simplest form of an expression is a constant or a variable.

Expression	Meaning
`3.1416`	A real constant
`.TRUE.`	A logical constant
`X`	A scalar variable
`Y`	An array variable
`Y(K)`	A variable that is an array element of `Y`
`Y(2:10:2)`	A variable that is an array subsection of `Y`
`M%N`	A variable that is a component of a structure `M`
`Y(K)(I:I+3)`	A variable that is a substring of array element `Y(K)`

The values of these simple expressions are the constant value `3.1416`, the constant value `.TRUE.`, the value of the variable `X`, the value of the array `Y`, the value of the array element `Y(K)`, the value of the array subsection `Y(2:10:2)`, the value of the component `N` of structure `M`, and the value of a substring of an array element `Y(K)`, respectively.

7.2.1 Operands

An operand in an expression can be one of the following items:

- A constant or subobject of a constant
- A variable (for example, a scalar, an array, a substring, or a pointer)
- An array constructor
- A structure constructor

- A function reference (returning, for example, a scalar, an array, a character variable, or a pointer)
- Another expression in parentheses

The following examples show operands:

```
A              ! scalar or an array
B(1)           ! array element or function
C(3:5)         ! array section or a substring
(A + COS(X))   ! expression in parentheses
(/ 1.2, 2.41 /)! array constructor
RATIONAL(1,2)  ! structure constructor or function
I_PTR          ! pointer to an integer target
```

7.2.2 Binary and Unary Operations

There are two forms that operations can take in an expression. One is an operation involving two operands, such as multiplying two numbers together. The other is an operation on one operand, such as making a number negative. These forms are called *binary* and *unary* operations, respectively.

Table 7-1 lists the intrinsic operators. You can use function subprograms to define additional operators. User-defined operators are either binary or unary operators.

Table 7-1 Intrinsic operators and the allowed types of their operands

Operator category	Intrinsic operator	Operand types
Arithmetic	`**`, `*`, `/`, `+`, `-`, unary `+`, unary `-`	Numeric of any combination of numeric types and kind type parameters or Cray pointer. Cray pointers are only allowed with the `+` or `-` operators.
Character	`//`	Character of any length.
Relational	`.EQ.`, `.NE.`, `==`, `/=`	Both of any numeric type and any kind type parameter or Cray pointer, or both of type character with any character length parameter.

Operator category	Intrinsic operator	Operand types
Relational	`.GT.`, `.GE.`, `.LT.`, `.LE.`, `>`, `>=`, `<`, `<=`	Both of any numeric type (except complex) and any kind type parameter or Cray pointer, or both of type character with any character length parameter.
Logical	`.NOT.`, `.AND.`, `.OR.`, `.XOR.`, `.EQV.`, `.NEQV.`	Both of type logical with any combination of kind type parameters.
Bitwise masking (Boolean) expressions (EXT)	`.NOT.`, `.AND.`, `.OR.`, `.XOR.`, `.EQV.`, `.NEQV.`	Integer, real, typeless, or Cray pointer.

ANSI/ISO: The Fortran standard does not specify the bitwise masking (Boolean) expressions, nor does it specify the `.XOR.` operator as a logical operator.

A binary operator combines two operands, as in the following:

x_1 *operator* x_2

Examples:

```
A + B
2 * C
```

The examples show an addition between two operands `A` and `B`, and a multiplication of two operands, the constant `2` and the operand `C`.

A unary operation acts on one operand, as in the following:

operator x_1

Examples:

```
- C
.NOT. L
```

The first example results in the value minus `C`. The second example produces a value that is the logical complement of `L`; the operator `.NOT.` is the only intrinsic operator that is a unary operator and is never a binary operator.

7.2.3 Intrinsic and Defined Operations

Intrinsic operations are those whose definitions are known to the compiler. They are built into Fortran and are always available for use in expressions. Table 7-1, page 217, lists the operators built into Fortran as specified by the standard.

The relational operator symbols ==, /=, >, >=, <, and <= are synonyms for the operators .EQ., .NE., .GT., .GE., .LT., and .LE., respectively.

The *less than or greater than* operation is represented by the <> operator and the .LG. keyword. This operation is suggested by the IEEE standard for floating-point arithmetic, and the compiler supports this operator. Only values of type real can appear on either side of the <> or .LG. operators. If the operands are not of the same kind type value, the compiler converts them to equivalent kind types. This operator's functionality differs slightly, depending on your platform. The <> and .LG. operators perform a less-than-or-greater-than operation as specified in the IEEE standard for floating-point arithmetic.

ANSI/ISO: The Fortran standard does not specify the <> or .LG. operators.

The compiler allows abbreviations for the logical and masking operators. The abbreviations .A., .O., .N., and .X. are synonyms for .AND., .OR., .NOT., and .XOR., respectively. If you define the abbreviated operator for any type, the abbreviated form of the intrinsic operator cannot be used in any scope in which the defined operator is accessible.

ANSI/ISO: The Fortran standard does not specify abbreviations for the logical and masking operators.

In addition to the Fortran operators that are intrinsic (built in), there may be user-defined operators in expressions.

Defined operations are those that you define in the Fortran program and are made available to each program unit that uses them. The computation performed by a defined operation is described explicitly in a function that must appear as a subprogram in the Fortran program where it is used. The operator used in a defined operation is called a *defined operator*. In this way, you extend the repertoire of operations so that computations can be expressed in a natural way using operator notation. Function subprograms that define operators are explained in detail in the *MIPSpro Fortran Language Reference Manual, Volume 2*.

A defined operator uses a symbol that is either the symbol for an intrinsic operator or is a new operator symbol. The synonyms described above for the relational operators remain synonyms in all contexts, even when there are defined operators. For example, if the operator < is defined for a new type, say `STRING`, the same definition applies to the operator `.LT.` for the type `STRING`; if the operator `.LT.` is specified as private, the operator < is also private.

A distinction is made between a defined (or new) operator and an extended intrinsic operator. An *extended intrinsic operator* is one that uses the same symbol as an intrinsically defined Fortran operator, like plus + or multiply *. It also causes the operations to be combined in the same order as is specified for the intrinsic operator. A *defined operator* is one where the operator symbol is not the same as an intrinsic operator but is new, such as the `.INVERSE.` operator. Defined operators, however, have a fixed precedence; defined unary operators have the highest precedence of all operators, and defined binary operators have the lowest precedence of all operators. The precedences of all operators are described in more detail in Section 7.2.5, page 236.

A *defined elemental operation* is a defined operation for which the function is elemental.

ANSI/ISO: The masking or Boolean operators and their abbreviations, which are extensions to Fortran, can be redefined as defined operators. If you redefine a masking operator, your definition overrides the intrinsic masking operator definition. See Table 7-1, page 217, for a list of the operators.

7.2.4 Rules for Forming Expressions

Expressions are formed by combining operands. Operands can be constants, variables (scalars, array elements, arrays, array sections, structures, structure components, and pointers), array constructors, structure constructors, functions, and parenthesized expressions with intrinsic and defined operators.

The method used to specify the expression formation rules is a collection of syntax rules that determine the forms of expressions. The order of evaluation of the operations in an expression is determined by the usual semantics for the operations, and the syntax rules are designed to be consistent with these semantics. In fact, the order of evaluation defines a precedence order for operators that is summarized in Table 7-2.

Table 7-2 The hierarchy of expressions through forms

Term	Definition
expression	[*expression defined_operator*] *equivalence_expression*
equivalence_expression	[*equivalence_expression* `.EQV.`] *disjunct_expression* *equivalence_expression* `.NEQV.` *disjunct_expression*
Exclusive OR (extension)	[*disjunct_expression* `.XOR.`] *conjunct_expression*
disjunct_expression	[*disjunct_expression* `.OR.`] *conjunct_expression*
conjunct_expression	[*conjunct_expression* `.AND.`] *not_expression*
not_expression	[`.NOT.`] *comparison_expression*
comparison_expression	[*concatenation_expression relational_operator*] *concatenation_expression*
concatenation_expression	[*concatenation_expression* `//`] *summation_expression*
summation_expression	[*summation_expression* `+`] *multiplication_expression* *summation_expression* `-` *multiplication_expression* `+` *multiplication_expression* `-` *multiplication_expression*
multiplication_expression	[*multiplication_expression* `*`] *exponentiation_expression* *multiplication_expression* `/` *exponentiation_expression*
exponentiation_expression	*defined_unary_expression* [`**` *exponentiation_expression*]
defined_unary_expression	[*defined_operator*] *primary*
primary	*constant* *constant_subobject* *variable* *array_constructor* *structure_constructor* *function_reference* (*expression*)

The set of syntax rules defines an expression at the highest level in terms of operators and operands, which are themselves expressions. As a result, the formal set of rules is recursive. The basic or lowest level of an expression is a primary, which, for example, can be a variable, a constant, or a function, or recursively an expression enclosed in parentheses. The rules for forming expressions are described from the lowest or most

primitive level to the highest or most complex level; that is, the rules are stated from a primary up to an expression.

7.2.4.1 Primary

A primary is defined as follows:

primary	**is**	*constant*
	or	*constant_subobject*
	or	*variable*
	or	*array_constructor*
	or	*structure_constructor*
	or	*function_reference*
	or	(*expr*)
constant_subobject	**is**	*subobject*

A variable that is a primary must not be a whole assumed-size array or a section of an assumed-size array name, unless the last subscript position of the array is specified with a scalar subscript or a section subscript in which the upper bound is specified.

The following examples show primaries:

Primary	Meaning
`3.2`	A real constant
`ONE`	A named constant
`'ABCS'(I:I)`	A constant subobject
`A`	A variable (scalar, array, structure, or pointer)
`B(:,1:N)`	An assumed-size array with an upper bound in the last dimension
`C(I)`	An array element
`CH(I:J)`	A substring
`(/ 1, J, 7 /)`	An array constructor

`RATIONAL(I, J)`	A structure constructor
`FCN(A)`	A function reference
`(A * B)`	A parenthesized expression

In the previous examples, `ONE` is a named constant if it has the `PARAMETER` attribute or appears in a `PARAMETER` statement. `'ABCS'(I:I)` is a constant subobject even though `I` may be a variable because its parent (`'ABCS'`) is a constant; the reference `'ABCS'(I:I)` is a constant subobject because it cannot be defined like a variable can be defined. `RATIONAL` is a derived type and `FCN` is a user-defined function.

When an array variable is a primary, the whole array is used, except in a masked assignment statement. In a masked assignment statement, only that part of the array specified by the mask is used.

When a pointer is a primary, the target associated with (pointed to by) the pointer is used, except possibly when the pointer is an actual argument of a procedure, or is an operand of a defined operation or a defined assignment. Whether the pointer or the target is used in these exceptional cases is determined by the procedure invoked by the reference.

Recall that an assumed-size array is a dummy argument whose shape is not completely specified in the subprogram in that the extent in the last dimension is determined by its corresponding actual argument. The implementation model is that the extent in the last dimension is never known to the subprogram but is specified by the use of a subscript, section subscript, or vector subscript expression that defines an upper bound in the last dimension. Unless the extent is specified in this way, such an object must not be used as a primary in an expression. On the other hand, if a subscript, section subscript with an extent for the upper bound, or a vector subscript is specified for the last dimension, the array value has a well-defined shape and hence can be used as a primary in any expression. For example, if `A` is declared as `REAL A(3,*)`, array `A(:,3)` has a well-defined shape and can be used as a primary in an expression.

Expressions can be used as actual arguments in procedure references (function references or subroutine calls). Because actual arguments can be expressions involving operations, actual arguments must not contain assumed-size arrays, unless their shape is well-defined, as described above. An actual argument, however, can be just a variable, which then allows the actual argument to be the name of an assumed-size array. This implies that such actual arguments can be assumed-size arrays, unless the procedure requires the shape of the argument to be specified by the actual argument. Most of the intrinsic procedures that allow array arguments require the shape to be specified for the actual array arguments, and therefore assumed-size

arrays cannot be used as actual arguments for most intrinsic functions. The exceptions are all references to the intrinsic function LBOUND(3i), and certain references to the intrinsic functions UBOUND(3i) and SIZE(3i).

7.2.4.2 Defined Unary Expression

Defined unary expressions have the highest operator precedence. A *defined unary expression* is a defined operator followed by a primary. These are defined as follows:

defined_unary_expr	**is** [*defined_operator*] *primary*
defined_operator	**is** . *letter* [*letter*] ... .

A defined operator must not contain more than 31 letters.

A defined operator must not be the same as any intrinsic operator (.NOT., .AND., .OR., .EQV., .NEQV., .EQ., .NE., .GT., .GE., .LT., .LE., or .LG.) or any logical literal constant (.FALSE. or .TRUE.).

ANSI/ISO: The Fortran standard does not describe the .LG. operator.

A defined operator can be the same as one of the masking or Boolean operators supported by the compiler as extensions to the Fortran standard. The corresponding operator loses its intrinsic properties. Note that the abbreviations .T., .F., .A., .O., .N., and .X. are synonyms for .TRUE., .FALSE., .AND., .OR., .NOT., and .XOR., respectively. If you define the abbreviated operator for any type, the abbreviated form of the intrinsic operator also cannot be used in any scope in which the defined operator is accessible, and the redefined abbreviated logical constants can no longer be used as logical constants.

The following examples show defined unary expressions:

Expression	Meaning
.INVERSE. B	A defined unary expression (where .INVERSE. is a defined operator)

`A`	A primary is also a defined unary expression

7.2.4.3 Exponentiation Expression

An *exponentiation expression* is an expression in which the operator is the exponentiation operator `**`. This is defined as follows:

exponentiation_expr	**is**	*defined_unary_expr* [`**` *exponentiation_expr*]

Note that the definition is right recursive (that is, the defined term appears to the right of the operator `**`) which indicates that the precedence of the `**` operator in contexts of equal precedence is right-to-left. Thus, the interpretation of the expression `A ** B ** C` is `A ** ( B ** C )`.

The following examples show exponentiation expressions:

Expression	**Meaning**
`A ** B`	An exponentiation expression
`A ** B ** C`	An exponentiation expression with right-to-left precedence
`.INVERSE. B`	A defined unary expression is also an exponentiation expression
`A`	A primary is also an exponentiation expression

7.2.4.4 Multiplication Expression

A multiplication expression is an expression in which the operator is either `*` or `/`. It is defined as follows:

multiplication_expr	**is**	[*multiplication_expr* `*`] *exponentiation_expr*
	or	[*multiplication_expr* `/`] *exponentiation_expr*

Note that the definition is left recursive (that is, the defined term appears to the left of the operator `*` or `/`) which indicates that the precedence of the `*` and `/` operators in contexts of equal precedence is left-to-right. Thus, the interpretation of the expression

`A * B * C` is `(A * B) * C`, or `A / B * C` is `(A / B) * C`. This left-to-right precedence rule applies to the remaining binary operators except the relational operators.

The following examples show multiplication expressions:

Expression	Meaning
`A * B`	A multiplication expression
`A * B * C`	A multiplication expression with left-to-right precedence
`A / B`	A multiplication expression
`A / B / C`	A multiplication expression with left-to-right precedence
`A ** B`	An exponentiation expression is also a multiplication expression
`.INVERSE. B`	A defined unary expression is also a multiplication expression
`A`	A primary is also a multiplication expression

7.2.4.5 Summation Expression

A *summation expression* is an expression in which the operator is either `+` or `-`. It is defined as follows:

summation_expr	**is**	[[*summation_expr*] `+`] *multiplication_expr*
	or	[[*summation_expr*] `-`] *multiplication_expr*

The following examples show summation expressions:

Expression	Meaning
`A + B`	A summation expression
`A + B - C`	A summation expression with left-to-right precedence
`- A - B - C`	A summation expression with left-to-right precedence
`+ A`	A summation expression using unary +

`- A`	A summation expression using unary `-`
`A * B`	A multiplication expression is also a summation expression
`A ** B`	An exponentiation expression is also a summation expression
`.INVERSE. B`	A defined unary expression is also a summation expression
`A`	A primary is also a summation expression

7.2.4.6 Concatenation Expression

A *concatenation expression* is an expression in which the operator is `//`. It is defined as follows:

concatenation_expr	**is**	[*concatenation_expr* `//`] *summation_expr*

The following examples show concatenation expressions:

Expression	**Meaning**
`A // B`	A concatenation expression
`A // B // C`	A concatenation expression with left-to-right precedence
`A - B`	A summation expression is also a concatenation expression
`- A`	A summation expression is also a concatenation expression
`A * B`	A multiplication expression is also a concatenation expression
`A ** B`	An exponentiation expression is also a concatenation expression
`.INVERSE. B`	A defined unary expression is also a concatenation expression

	A	A primary is also a concatenation expression

7.2.4.7 Comparison Expression

A *comparison expression* is an expression in which the operator is a relational operator. It is defined as follows:

	comparison_expr	**is**	[*concatenation_expr rel_op*] *concatenation_expr*
	rel_op	**is**	`.EQ.`
		or	`.NE.`
		or	`.LT.`
		or	`.LE.`
		or	`.GT.`
		or	`.GE.`
EXT		**or**	`.LG.`
		or	`==`
		or	`/=`
		or	`<`
		or	`<=`
		or	`>`
		or	`>=`
EXT		**or**	`<>`

The operators `==`, `/=`, `<`, `<=`, `>`, `>=`, and `<>` are synonyms in all contexts for the operators `.EQ.`, `.NE.`, `.LT.`, `.LE.`, `.GT.`, `.GE.`, and `.LG.`, respectively.

ANSI/ISO: The Fortran standard does not describe the `.LG.` or `<>` operators.

Note that the definition of a comparison expression is not recursive, and therefore comparison expressions cannot contain relational operators in contexts of equal precedence.

The following examples show comparison expressions:

Expression	Meaning
`A .EQ. B`	A comparison expression
`A < B`	A comparison expression
`A // B`	A concatenation expression is also a comparison expression
`A - B`	A summation expression is also a comparison expression
`- A`	A summation expression is also a comparison expression
`A * B`	A multiplication expression is also a comparison expression
`A ** B`	An exponentiation expression is also a comparison expression
`.INVERSE. B`	A defined unary expression is also a comparison expression
`A`	A primary is also a comparison expression

7.2.4.8 Not Expression

A *not expression* is an expression in which the operator is `.NOT.`. It is defined as follows:

not_expr	**is**	[`.NOT.`] *comparison_expr*

Note that the definition of a not expression is not recursive, and therefore not expressions cannot contain adjacent `.NOT.` operators.

The following examples show not expressions:

Expression	Meaning
`.NOT. A`	A not expression
`A .EQ. B`	A comparison expression is also a not expression

`A // B`	A concatenation expression is also a not expression
`A - B`	A summation expression is also a not expression
`- A`	A summation expression is also a not expression
`A * B`	A multiplication expression is also a not expression
`A ** B`	An exponentiation expression is also a not expression
`.INVERSE. B`	A defined unary expression is also a not expression
`A`	A primary is also a not expression

7.2.4.9 Conjunct Expression

A *conjunct expression* is an expression in which the operator is `.AND.`. It is defined as follows:

conjunct_expr	**is**	[*conjunct_expr* `.AND.`] *not_expr*

Note that the definition of a conjunct expression is left recursive, and therefore the precedence of the `.AND.` operator in contexts of equal precedence is left-to-right. Thus, the interpretation of the expression `A .AND. B .AND. C` is `(A .AND. B) .AND. C`.

The following examples show conjunct expressions:

Expression	**Meaning**
`A .AND. B`	A conjunct expression
`A .AND. B .AND. C`	A conjunct expression with left-to-right precedence
`.NOT. A`	A not expression is also a conjunct expression
`A .EQ. B`	A comparison expression is also a conjunct expression
`A // B`	A concatenation expression is also a conjunct expression
`A - B`	A summation expression is also a conjunct expression
`- A`	A summation expression is also a conjunct expression
`A * B`	A multiplication expression is also a conjunct expression
`A ** B`	An exponentiation expression is also a conjunct expression

`.INVERSE. B`	A defined unary expression is also a conjunct expression
`A`	A primary is also a conjunct expression

7.2.4.10 Inclusive Disjunct Expression

An *inclusive disjunct expression* is an expression in which the operator is `.OR.`. It is defined as follows:

inclusive_disjunct_expr	**is**	[*inclusive_disjunct_expr* `.OR.`] *conjunct_expr*

Note that the definition of an inclusive disjunct expression is left recursive, and therefore the precedence of the `.OR.` operator in contexts of equal precedence is left-to-right. Thus, the interpretation of the expression `A. OR. B .OR. C` is `(A .OR. B) .OR. C`.

The following examples show inclusive disjunct expressions:

Expression	**Meaning**
`A .OR. B`	An inclusive disjunct expression
`A .OR. B .OR. C`	An inclusive disjunct expression with left-to-right precedence
`A .AND. B`	A conjunct expression is also an inclusive disjunct expression
`.NOT. A`	A not expression is also an inclusive disjunct expression
`A .EQ. B`	A comparison expression is also an inclusive disjunct expression
`A // B`	A concatenation expression is also an inclusive disjunct expression
`A - B`	A summation expression is also an inclusive disjunct expression
`- A`	A summation expression is also an inclusive disjunct expression
`A * B`	A multiplication expression is also an inclusive disjunct expression

`A ** B`	An exponentiation expression is also an inclusive disjunct expression
`.INVERSE. B`	A defined unary expression is also an inclusive disjunct expression
`A`	A primary is also an inclusive disjunct expression

7.2.4.11 Equivalence Expressions and Exclusive Disjunct Expressions

An *equivalence expression* is an expression in which the operator is either `.EQV.` or `.NEQV.`. It is defined as follows:

equivalence_expr	**is**	[*equivalence_expr* `.EQV.`] *inclusive_disjunct_expr*
	or	[*equivalence_expr* `.NEQV.`] *inclusive_disjunct_expr*

An *exclusive disjunct expression* is an expression in which the operator is `.XOR.`. It is defined as follows:

EXT	*exclusive_disjunct_expr*	**is**	[*exclusive_disjunct_expr* `.XOR.`] *inclusive_disjunct_expr*

ANSI/ISO: The Fortran standard does not specify the `.XOR.` operator.

Note the following:

- In the following discussion, *equivalence expression* means either *equivalence expression* or *exclusive disjunct expression*.
- The definition of an equivalence expression is left recursive, and therefore the precedence of the `.EQV.` or `.NEQV.` operators in contexts of equal precedence is left-to-right. Thus, the interpretation of the expression `A .EQV. B .NEQV. C` is `(A .EQV. B) .NEQV. C`.

The following examples show equivalence expressions:

Expression	**Meaning**
`A .EQV. B`	An equivalence expression

`A .XOR. B`	An equivalence exclusive disjunct expression
`A .NEQV. B .XOR. C`	An equivalence expression with left-to-right precedence
`A .OR. B`	An inclusive disjunct expression is also an equivalence expression
`A .AND. B`	A conjunct expression is also an equivalence expression
`.NOT. A`	A not expression is also an equivalence expression
`A .EQ. B`	A comparison expression is also an equivalence expression
`A // B`	A concatenation expression is also an equivalence expression
`A - B`	A summation expression is also an equivalence expression
`- A`	A summation expression is also an equivalence expression
`A * B`	A multiplication expression is also an equivalence expression
`A ** B`	An exponentiation expression is also an equivalence expression
`.INVERSE. B`	A defined unary expression is also an equivalence expression
`A`	A primary is also an equivalence expression

7.2.4.12 Expression

The most general form of an expression is defined as follows:

expr	**is**	[*expr defined_binary_op*] *equivalence_expr*
defined_binary_op	**is**	**.***letter* [*letter*] **....**

Note that the definition of an expression is left recursive, and therefore the precedence of the binary defined operator in contexts of equal precedence is

left-to-right. The interpretation of the expression `A .PLUS. B .MINUS. C` is thus `(A .PLUS. B) .MINUS. C` (where `.MINUS.` is a defined operator).

The following examples show expressions:

Expression	Meaning
`A .PLUS. B`	An expression (where `.PLUS.` is a defined operator)
`A .CROSS. B .CROSS. C`	An expression with left-to-right precedence (where `.CROSS.` is a defined operator)
`A .EQV. B`	An equivalence expression is also an expression
`A .OR. B`	An inclusive disjunct expression is also an expression
`A .AND. B`	A conjunct expression is also an expression
`.NOT. A`	A not expression is also an expression
`A .EQ. B`	A comparison expression is also an expression
`A // B`	A concatenation expression is also an expression
`A - B`	A summation expression is also an expression
`- A`	A summation expression is also an expression
`A * B`	A multiplication expression is also an expression
`A ** B`	An exponentiation expression is also an expression
`.INVERSE. B`	A defined unary expression is also an expression
`A`	A primary is also an expression

7.2.4.13 Summary of the Forms and Hierarchy for Expressions

The previous sections have described in detail the sorts of expressions that can be formed. These expressions form a hierarchy that can best be illustrated by a figure. Figure 7-1, page 235, describes the hierarchy by placing the simplest form of an expression, a variable, at the center of a set of nested rectangles. The more general forms of an expression are the enclosing rectangles, from a primary to an exponentiation expression, to a summation expression, and finally to a general expression using a defined binary operator `.CROSS.`. Figure 7-1 demonstrates that an expression is all of these special case forms, including the simplest form, a primary.

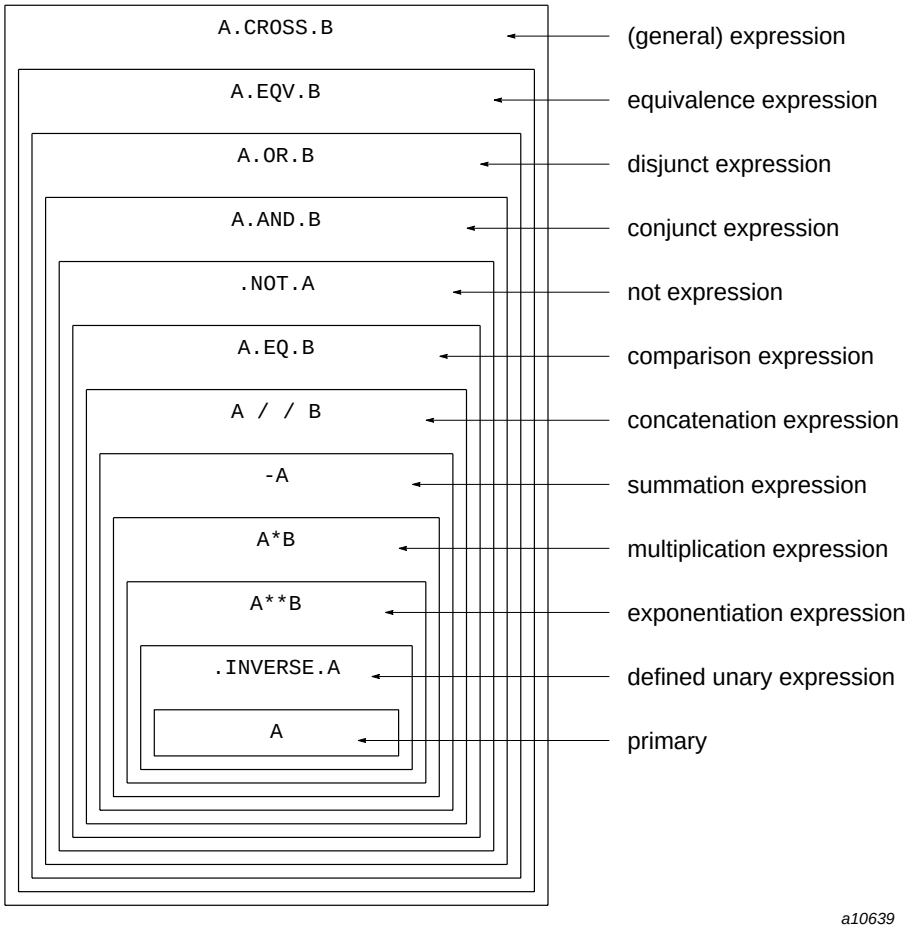

Figure 7-1 The hierarchy of expressions by examples

Table 7-2, page 221, illustrates the relationship between the different sorts of expressions by summarizing the definitional forms in one table. The simplest form of an expression is at the bottom and is the primary, as in Figure 7-1. The next, more general, form is second from the bottom and is the defined unary expression; it uses the primary in its definition. At the top of the table is the most general form of an expression.

7.2.5 Precedence of Operators

Table 7-3 summarizes the relative precedence of operators, including the precedence when operators of equal precedence are adjacent. An entry of N/A in the rightmost column indicates that the operator cannot appear in such contexts. The leftmost column classifies the operators as defined, numeric, character, relational, and logical operators. Note that these operators are not intrinsic operators unless the types of the operands are those specified in Table 7-4, page 238.

Table 7-3 Categories of operations and relative precedences

Category of operator	Operator	Precedence	In context of equal precedence
Defined	Unary defined-operator	Highest	N/A
Numeric	**	.	Right-to-left
Numeric	* or /	.	Left-to-right
Numeric	Unary + or -	.	N/A
Numeric	Binary + or -	.	Left-to-right
Character	//	.	Left-to-right
Relational	.EQ., .NE., .LT., .LE., .GT., .GE., .LG., ==, /=, <, <=, >, >=, <>	.	N/A
Logical or Boolean	.NOT.	.	N/A
Logical or Boolean	.AND.	.	Left-to-right
Logical or Boolean	.OR.	.	Left-to-right
Logical	.EQV. or .NEQV.	.	Left-to-right
Logical or Boolean	.XOR.	.	Left-to-right
Defined	Binary defined-operator	Lowest	Left-to-right

For example, consider the following expression:

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

Table 7-2, page 221, indicates that the `.AND.` operator is of higher precedence than the `.OR.` operator, and the `.AND.` operators are combined left-to-right when in contexts of equal precedence; thus, `A` and `B` are combined by the `.AND.` operator, the result `A .AND. B` is combined with `C` using the `.AND.` operator, and that result is combined with `D` using the `.OR.` operator. Thus, this expression is interpreted the same way as the following fully parenthesized expression:

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

Notice that the defined operators have fixed precedences; defined unary operators have the highest precedence of all operators and are all of equal precedence; defined binary operators have the lowest precedence, are all of equal precedence, and are combined left-to-right when in contexts of equal precedence. Both kinds of defined operators may have multiple definitions in the program unit and therefore may be generic just as intrinsic operators and intrinsic procedures are generic.

As a consequence of the expression formation rules, unary operators cannot appear in a context of equal precedence; the precedence must be specified by parentheses. There is thus no left-to-right or right-to-left rule for any unary operators. Similarly, the relational operators cannot appear in a context of equal precedence; consequently, there is no left-to-right or right-to-left rule for the relational operators. Use of some of the operators as Boolean or masking operators is an extension to the Fortran standard that is supported by the compiler.

ANSI/ISO: The Fortran standard does not specify the use of operators as Boolean or masking operators.

7.2.6 Intrinsic Operations

Intrinsic operations are those known to the processor. For an operation to be intrinsic, an intrinsic operator symbol must be used, and the operands must be of the intrinsic types specified in Table 7-4, page 238.

Note: In the following table, the symbols I, R, Z, C, L, B, and P stand for the types integer, real, complex, character, logical, Boolean, and Cray pointer, respectively. Where more than one type for x_2 is given, the type of the result of the operation is given in the same relative position in the next column. Boolean and Cray pointer types are compiler extensions.

Table 7-4 Operand types and results for intrinsic operations

Intrinsic operator	Type of x_1	Type of x_2	Type of result
Unary +, -		I, R, Z, B, P	I, R, Z, I, P
Binary +, -, *, /, **	I	I, R, Z, B, P	I, R, Z, I, P
	R	I, R, Z, B	R, R, Z, R
	Z	I, R, Z	Z, Z, Z
	B	I, R, B, P	I, R, B, P
	P	I, B, P	P, P, P
	(For Cray pointer, only + and - are allowed.)		
//	C	C	C
.EQ., ==, .NE., /=	I	I, R, Z, B, P	L, L, L, L, L
	R	I, R, Z, B, P	L, L, L, L, L
	Z	I, R, Z, B, P	L, L, L, L, L
	B	I, R, Z, B, P	L, L, L, L, L
	P	I, R, Z, B, P	L, L, L, L, L
	C	C	L
.GT., >, .GE., >=, .LT., <, .LE., <=	I	I, R, B, P	L, L, L, L
	R	I, R, B	L, L, L
	C	C	L
	P	I, P	L, L
.LG., <>	R	R	L
.NOT.		L	L
		I, R, B	B
.AND., .OR., .EQV., .NEQV., .XOR.	L	L	L
	I, R, B	I, R, B	B

ANSI/ISO: The Fortran standard does not specify the use of type Boolean or the use of type Boolean in masking expressions, nor does it describe the .LG. and <> operators.

The intrinsic operations are either binary or unary. The binary operations use the binary intrinsic operator symbols +, -, *, /, **, //, .EQ., .NE., .LT., .GT., .LE., .GE., and .LG.(and their synonyms ==, /=, <>, <, >=, and <>), .AND., .OR., .XOR., .EQV., and .NEQV.. The unary operations use the unary intrinsic operator symbols +, -, and .NOT..

Note that the intrinsic operators .AND., .OR., .NOT., and .XOR. can be abbreviated as .A., .O., .N., or .X.. If a user-defined operator with the same name as the abbreviated name is accessible in a scope, the abbreviated forms of these operators may not be used as synonyms for .AND., .OR., .NOT., or .XOR..

The intrinsic operations are divided into five classes with different rules and restrictions for the types of the operands. The five classes are numeric intrinsic, character intrinsic, logical intrinsic, relational intrinsic operations, and bitwise masking expressions.

The numeric intrinsic operations use the intrinsic operators +, -, *, /, and **. The operands can be of any numeric type and with any kind type parameters. The result of the operation is of a type specified by Table 7-4, page 238, and has type parameters as specified in Section 7.2.8.2, page 244.

For example, the following expressions, in which I, R, D, and Z are declared to be of types integer, real, double-precision real, and complex, have the types and type parameters of the variables R, I, D, and Z, respectively:

```
I + R
I * I
I - D
I / Z
```

There is only one character intrinsic operation; it uses the intrinsic operator //. The result of a character intrinsic operation is type character.

The logical intrinsic operations use the intrinsic operators .AND., .OR., .NOT., .EQV., and .NEQV., respectively. The result of a logical intrinsic operation is type logical and has type parameters as specified in Section 7.2.8.2, page 244.

The relational intrinsic operations use the intrinsic operators .EQ., .NE., .GT., .GE., .LT., .LE., or .LG. or their symbolic synonyms. A relational intrinsic operation is a

numeric relational intrinsic operation if its operands are of type integer, real, or complex. A relational intrinsic operation is a character relational intrinsic operation if its operands are of type character. The result of either kind of relational operation is type logical and has type parameters as specified in Section 7.2.8.2, page 244.

ANSI/ISO: The Fortran standard does not describe the `.LG.` operator.

The operators `.NOT.`, `.AND.`, `.OR.`, `.EQV.`, and `.XOR.` can also be used in the compiler's bitwise masking expressions; these are extensions to the Fortran standard. The result is type integer.

7.2.7 Defined Operations

A *defined operation* is any nonintrinsic operation that is interpreted and evaluated by a function subprogram specified by an interface block with a generic specifier of the following form:

```
OPERATOR (defined_operator)
```

A *defined elemental operation* is a defined operation for which the function is elemental.

A defined operation uses either a defined operator or an intrinsic operator symbol, and it is either unary or binary. Its forms are as follows:

intrinsic_unary_op x_2

defined_operator x_2

x_1 *intrinsic_binary_op* x_2

x_1 *defined_operator* x_2

The terms *intrinsic_unary_op* and *intrinsic_binary_op* include all intrinsically defined operators; these terms are not specifically defined in any syntax rules.

x_1 and x_2 are operands. When an intrinsic operator symbol is used, the type of x_2 (for a unary operator) and types of x_1 and x_2 (for a binary operator) must not be the same

as the types of the operands specified in Table 7-4, page 238, for the particular intrinsic operator symbol. Thus, you cannot redefine intrinsic operations on intrinsic types.

When a defined operation uses an intrinsic operator symbol, the generic properties of that operator are extended to the new types specified by the interface block. When a defined operation uses a defined operator, the defined operation is called an *extension operation*, and the operator is called an *extension operator*. An extension operator can have generic properties by specifying more than one function subprogram in an interface block with a generic specifier of the form `OPERATOR (`*defined_operator*`)`.

7.2.8 Data Type, Type Parameters, and Shape of an Expression

The data type, type parameters, and shape of a complete expression are determined by the data type, type parameters, and shape of each constant, variable, constructor, and function reference appearing in the expression. The determination is inside-out in the sense that the properties are determined first for the primaries. These properties are then determined repeatedly for the operations in precedence order, resulting eventually in the properties for the expression.

For example, consider the expression `A + B * C`, where `A`, `B`, and `C` are of numeric type. First, the data types, type parameter values, and shapes of the three variables `A`, `B`, and `C` are determined. Because `*` has a higher precedence than `+`, the operation `B * C` is performed first. The type, type parameters, and shape of the expression `B * C` are determined next, and then these properties for the entire expression are determined from those of `A` and `B * C`.

A *defined elemental operation* is a defined operation for which the function is elemental.

7.2.8.1 Data Type and Type Parameters of a Primary

The type, type parameters, and shape of a primary that is a nonpointer variable or constant are straightforward because these properties are determined by specification statements for the variable or named constant, or by the form of the constant. For example, if `A` is a variable, its declaration in a specification statement such as the following determines it as an explicit-shaped array of type real with a default kind parameter:

```
REAL A(10, 10)
```

For a constant such as the following, the form of the constant indicates that it is a scalar constant of type complex and of default kind:

```
(1.3, 2.9)
```

For a pointer variable, the type, type parameters, and rank are determined by the declaration of the pointer variable. However, if the pointer is of deferred shape, the shape (in particular, the extents in each dimension) is determined by the target of the pointer. Consider the following declarations and assume that pointer A is associated with the target B:

```
REAL, POINTER :: A(:, :)
REAL, TARGET  :: B(10, 20)
```

The shape of A is (10, 20).

The type and type parameters of an array constructor are determined by the contents of the constructor. Unless the element is of type Boolean (typeless), its type and type parameters are those of any element of the constructor because they must all be of the same type and type parameters. If the element is of type Boolean, the type and kind type of the array constructor are the same as the default integer type. Therefore, the type and type parameters of the following array constructor are integer and kind value 1:

```
(/ 1_1, 123_1, -10_1 /)
```

Its shape is always of rank one and of size equal to the number of elements.

The type of a structure constructor is the derived type used as the name of the constructor. A structure has no type parameters. So, the type of the following structure constructor is the derived type PERSON:

```
PERSON(56, 'Father')
```

(See Section 4.6, page 110, for the type definition PERSON.)

A structure constructor is always a scalar.

The type, type parameters, and shape of a function are determined by one of the following:

- An implicit type declaration for the function within the program unit referencing the function
- An explicit type declaration for the function within the program unit referencing the function (just like a variable)
- An explicit interface to the function. (When the interface is not explicit, the function is either an external function or a statement function.)

If the interface is explicit, the type, type parameter, and shape are determined by one of the following:

- The type and other specification statements for the function in an interface block within the program unit referencing the function
- The type and other specification statements for the internal or module procedure specifying the function
- The description of the particular intrinsic function being referenced

Note, however, that because intrinsic functions and functions with interface blocks can be generic, these properties are determined by the type, type parameters, and shapes of the actual arguments of the particular function reference.

For example, consider the following statements as part of the program unit specifying an internal function FCN:

```
REAL FUNCTION FCN(X)
DIMENSION FCN(10, 15)
```

A reference to FCN(3.3) is of type default real with shape (10,15). As a second example, consider the following:

```
REAL(SINGLE)  X(10, 10, 10)
   . . .
   . . .  SIN(X)   . . .
```

The interface to SIN(3i) is specified by the definition of the sine intrinsic function. In this case, the function reference SIN(X) is of type real with kind parameter value SINGLE and of shape (10,10,10).

The interface is implicit if the function is external (and no interface block is provided) or is a statement function. In these cases, the shape is always that of a scalar, and the type and type parameters are determined by the implicit type declaration rules in effect, or by an explicit type declaration for the function name. In the following example, FCN(X) is a scalar of type integer with kind type parameter value SHORT:

```
IMPLICIT INTEGER(SHORT) (A-F)
   . . .
   . . .  FCN(X)   . . .
```

The one case for variables and functions that is not straightforward is the determination of the shape of a variable when it is of deferred shape or of assumed shape. For a deferred-shape array, the rank is known from the declaration but the size

of each dimension is determined as the result of executing an ALLOCATE statement or a pointer assignment statement. For an assumed-shape array, the rank is also known from the declaration but the size is determined by information passed into the subprogram. In the case of pointers, the shape of the object is that of the target associated with (pointed to by) the pointer. The shape of deferred-shape and assumed-shape arrays thus cannot be determined in general until execution time.

7.2.8.2 Type and Type Parameters of the Result of an Operation

The type of the result of an intrinsic operation is determined by the type of the operands and the intrinsic operation and is specified by Table 7-4, page 238.

For nonnumeric operations, the type parameters of the result of an operation are determined as follows:

- For the relational intrinsic operations, the kind type parameter is that for the default logical type.
- For logical intrinsic operations, the result kind type parameter depends on that of the operands. If the operands have the same kind type parameter (that is the same *value* for KIND=*value*), the kind type parameter is that of the operands. If logical operands have different kind type parameter values, the value of the result is that of the greater kind type parameter value.
- For the character intrinsic operation (//), the operands must have the same kind type parameter, so the result has that kind type parameter.
- The character length parameter value for the result is the sum of the character length parameters of the operands.

For numeric intrinsic operations, the kind type parameter value of the result is determined as follows:

- For unary operations, the kind type parameter value of the result is that of the operand.
- Floating-point operands are those of type real or type complex. If an operation includes a floating-point operand and an integer, the result type and kind type parameter value is determined as follows:
  - The result type is that of the floating-point operand.
  - The result kind type parameter value is the same as the kind type parameter value of the floating-point operand.

See Table 7-4, page 238, for more information.

- If an operation includes operands of type complex and type real, the result type and kind type parameter value is determined as follows:
  - The result type is type complex.
  - The result kind type parameter value is that of the greater value. That is, the kind type parameter value of each operand is examined, and whichever kind type parameter value is greater is assigned to the result.

    See Table 7-4, page 238, for more information.
- For binary operations, if the operands are of the same type and kind type parameters, the kind type parameter value of the result is the kind type parameter of the operands.
- For binary operations, if the operands are both of type integer but with different kind type parameters, the kind type parameter value of the result is the kind type parameter of the operand with the larger decimal exponent range. If the decimal exponent ranges of the two kinds are the same, the kind type parameter value of the result is the same as each operand.
- For binary operations, if the operands are both of type real or complex but with different kind type parameters, the kind type parameter of the result is the kind type parameter of the operand with the larger decimal precision. If the decimal precisions are the same, the kind type parameter value is that of the operands.

For numeric intrinsic operations, an easy way to remember the result type and type parameter rules is to consider that the numeric types (integer; real; complex; and compiler Boolean) are ordered by the increasing generality of numbers. Integers are contained in the set of real numbers and real numbers are contained in the set of complex numbers. Within the integer type, the kinds are ordered by increasing value ranges. Within the real and complex types, the kinds for each type are ordered by increasing decimal precision.

Using this model, the result type of a numeric intrinsic operation is the same type as the operand of the greater generality. For the result type parameter, the rule is complicated: if one or both of the operands is of type real or complex, the type parameter is that of the set of numbers of the more general type described above and with a precision as large as the precision of the operands; if both are of type integer, the result type parameter is of a set of numbers that has a range as large as the range of the operands.

The type and type parameter values of a defined operation are determined by the interface block for the referenced operation and are the type and type parameters of the name of the function specified by the interface block. Note that the operator can be generic and therefore the type and type parameters can be determined by the operands. For example, consider the following interface:

```
INTERFACE OPERATOR (.PLUS.)

   TYPE(SET) FCN_SET_PLUS(X, Y)
      USE DEFINITIONS
      TYPE (SET)  X, Y
      INTENT(IN) X, Y
   END FUNCTION FCN_SET_PLUS

   TYPE(RATIONAL) FCN_RAT_PLUS(X, Y)
      USE DEFINITIONS
      TYPE(RATIONAL)  X, Y
      INTENT(IN) X, Y
   END FUNCTION FCN_RAT_PLUS

END INTERFACE
```

The operation `A .PLUS. B`, where `A` and `B` are of type `RATIONAL`, is an expression of type `RATIONAL` with no type parameters. The operation `C .PLUS. D`, where `C` and `D` are of type `SET` is an expression of type `SET` with no type parameters.

7.2.8.3 Shape of an Expression

The shape of an expression is determined by the shape of each operand in the expression in the same recursive manner as for the type and type parameters for an expression. That is, the shape of an expression is the shape of the result of the last operation determined by the interpretation of the expression.

However, the shape rules are simplified considerably by the requirement that the operands of binary intrinsic operations must be in shape conformance; two operands are in *shape conformance* if both are arrays of the same shape, or one or both operands are scalars. The operands of a defined operation have no such requirement but must match the shape of the corresponding dummy arguments of the defining function.

For primaries that are constants, variables, constructors, or functions, the shape is that of the constant, variable, constructor, or function name. Recall that structure constructors are always scalar, and array constructors are always rank-one arrays of

size equal to the number of elements in the constructor. For unary intrinsic operations, the shape of the result is that of the operand. For binary intrinsic operations, the shape is that of the array operand if there is one and is scalar otherwise. For defined operations, the shape is that of the function name specifying the operation.

For example, consider the intrinsic operation `A + B` where `A` and `B` are of type default integer and default real respectively; assume `A` is a scalar and `B` is an array of shape `(3, 5)`. Then, the result is of type default real with shape `(3, 5)`.

7.2.8.4 The Extents of an Expression

For most contexts, the lower and upper bounds of an array expression are not needed; only the sizes of each dimension are needed to satisfy array conformance requirements for expressions. The bounds of an array expression when it is the `ARRAY` argument (first positional argument) of the `LBOUND`(3i) and `UBOUND`(3i) intrinsic functions are needed, however.

The functions `LBOUND`(3i) and `UBOUND`(3i) have two keyword arguments `ARRAY` and `DIM`. `ARRAY` is an array expression and `DIM`, which is optional, is an integer. If the `DIM` argument is present, `LBOUND`(3i) and `UBOUND`(3i) return the lower and upper bounds, respectively, of the dimension specified by the `DIM` argument. If `DIM` is absent, they return a rank-one array of the lower and upper bounds, respectively, of all dimensions of the `ARRAY` argument. As described below, these functions distinguish the special cases when the array argument is a name or structure component with no section subscript list from the general case when the array argument is a more general expression. Note that if `A` is a structure with an array component `B`, `A%B` is treated as if it were an array name and not an expression.

When the `ARRAY` argument is an array expression that is not a name or a structure component, the function `LBOUND`(3i) returns `1` if the `DIM` argument is specified and returns a rank-one array of 1s if the `DIM` argument is absent. For the same conditions, the function `UBOUND`(3i) returns as the upper bound the size of the requested dimension or the size of all dimensions in a rank-one array.

When the `ARRAY` argument is an array name or a structure component with no section subscript list, there are four cases to distinguish depending on the array specifier for the name. The following sections describe these four cases.

7.2.8.4.1 Explicit-shape Specifier

The `LBOUND`(3i) and `UBOUND`(3i) functions return the declared lower and upper bounds of the array name or the structure component with no section subscript list.

```
INTEGER  A(2:10, 11:12)
. . .
TYPE PASSENGER_INFO
   INTEGER NUMBER
   INTEGER TICKET_IDS(2:500)
END TYPE PASSENGER_INFO
. . .
TYPE(PASSENGER_INFO) PAL, MANY(3:10)
```

In this example, `LBOUND(A)` has the value `(/ 2, 11 /)`, and `UBOUND(A, 1)` has the value `10`. `LBOUND(PAL%TICKET_IDS)` has the value `(/ 2 /)` and `UBOUND(MANY%TICKET_IDS(2), 1)` has the value `10`.

7.2.8.4.2 Assumed-shape Specifier

The name is a dummy argument whose extents are determined by the corresponding actual argument. The dummy argument may have its lower bound in a particular dimension specified but if not, the lower bound is defined to be 1. The `LBOUND`(3i) function returns these lower bounds. The upper bound for a particular dimension is the extent of the actual argument in that dimension, if no lower bound is specified for the dummy argument. It is the extent minus 1 plus the lower bound if a lower bound is specified. The `UBOUND`(3i) function returns these upper bounds.

```
REAL  C(2:10, 11:12)
   . . .
CALL  S(C(4:8, 7:9))
CONTAINS
  SUBROUTINE S(A)
    REAL  A(:, 2:)
       . . .
    ! Reference to LBOUND(A)  and  UBOUND(A)
       . . .
```

Inside the body of subroutine `S`, `LBOUND(A)` has the value `(/ 1, 2 /)`, because the array starts at subscript position `1` by default in the first dimension and starts at subscript position `2` by declaration in the second dimension. `UBOUND(A)` has the value `(/ 5, 4 /)`, because there are five subscript positions (4 to 8) in the first dimension of the actual argument corresponding to `A`, and three subscript positions (7 to 9) in the second dimension of the same actual argument and the subscripts are specified to start at `2` by the declaration of the dummy argument `A`.

7.2.8.4.3 Assumed-size Specifier

The name is a dummy argument whose upper and lower bounds in all but the last dimension are declared for the dummy argument. The lower bound for the last dimension may be specified in the assumed-size specifier but, if absent, the lower bound is 1. The LBOUND(3i) function returns these lower bounds. The upper bound for all dimensions except the last one is known to the subprogram but the upper bound in the last dimension is not known. The UBOUND(3i) function, therefore, must not be referenced with the first argument being the name of an assumed-size array and no second argument, or the first argument being the name of an assumed-size array and the second argument specifying the last dimension of the array. Otherwise, the UBOUND(3i) function returns the upper bounds as declared for all but the last dimension.

```
REAL  C(2:10, 11:12)
   . . .
CALL  S (C(4:8, 7:9))
CONTAINS
  SUBROUTINE  S (A)
    REAL  A(-2:2, *)
       . . .
 ! Reference to LBOUND(A, 1)  and  UBOUND(A(:, 2))
 ! A reference to UBOUND(A) would be illegal.
 ! A reference to UBOUND(A, 2) would be illegal.
    . . .
```

Inside the body of subroutine S, LBOUND(A, 1) has the value -2. UBOUND(A(:, 2)) has the value 5 because A(:,2) is an expression, which is an array section, not an array name, and has five elements in the first dimension.

7.2.8.4.4 Deferred-shape Specifier

The name is the name of an allocatable array, an array pointer, or a structure component with one of its part references being a pointer array. As such, if the array or a part reference has not been allocated or associated with a target, the LBOUND(3i) and UBOUND(3i) functions must not be invoked with the ARRAY argument equal to such an array name. If it is an array pointer, either its target has been allocated by an ALLOCATE statement or its target has become associated with the pointer using a pointer assignment statement. In the former case, the LBOUND(3i) and UBOUND(3i) functions return the lower and upper bounds specified in the ALLOCATE statement. In the latter case, LBOUND is always 1 in pointer assignment, and UBOUND is the

extent of the dimension. Note that for zero-sized dimensions, LBOUND is always 1 and UBOUND is always 0.

```
REAL, ALLOCATABLE ::  A(:, :)
   . . .
ALLOCATE  ( A(5, 7:9) )
   . . .! Reference to LBOUND(A)  and  UBOUND(A)
   . . .
```

After the ALLOCATE statement above is executed, LBOUND(A) has the value (/ 1, 7 /), and UBOUND(A) has the value (/ 5, 9 /).

7.2.9 Special Expressions

Expressions can appear in statements other than assignment statements, in particular in specification statements. In many cases, such expressions are restricted in some way; for example, the operands in expressions in a PARAMETER statement are essentially restricted to constants. The standard and this manual use specific terms for the various categories of expressions allowed in specific syntactic contexts. For example, the expressions that can be used in PARAMETER statements are called *initialization expressions* and can be evaluated at the time the program is compiled. Initialization expressions are restricted forms of constant expressions.

The expressions that can be used as array bounds and character lengths in specification statements are called *specification expressions*. These expressions are scalar and of type integer and can be evaluated on entry to the program unit at the time of execution. The remainder of this subsection describes and defines such limited expressions and summarizes where they can be used.

7.2.9.1 Constant Expressions

A *constant expression* is one of the following constant values or is an expression consisting of intrinsic operators whose operands are any of the following constant values:

1. A literal or named constant, or a subobject of a constant where each subscript, section subscript, or starting and ending point of a substring range is a constant expression.
2. An array constructor where every subexpression has primaries that are constant expressions or are implied-DO variables of the array constructor.

3. A structure constructor where each component is a constant expression.
4. An elemental intrinsic function reference that can be evaluated at compile time.
5. A transformational intrinsic function reference that can be evaluated at compile time.
6. A reference to NULL(3i).
7. A reference to an intrinsic function that is one of the following:
   - An array inquiry function other than ALLOCATED(3i)
   - The bit inquiry function BIT_SIZE(3i)
   - The character inquiry function LEN(3i)
   - The KIND(3i) inquiry function
   - A numeric inquiry function

   Each argument of the function must be a constant expression or must be a variable whose type parameters or bounds inquired about are not assumed, defined by an expression that is not a constant expression, or definable by an ALLOCATE or pointer assignment statement.
8. An implied-DO variable within an array constructor in which the bounds and strides of the corresponding implied-DO are constant expressions.
9. A constant expression enclosed in parentheses.

The restriction in item 4 to intrinsic functions that can be evaluated at compile-time eliminates the use of the intrinsic functions PRESENT(3i), ALLOCATED(3i), and ASSOCIATED(3i). It also requires that each argument of the intrinsic function reference be a constant expression or a variable whose type parameters or bounds are known at compile time. This restriction excludes, for example, named variables that are assumed-shape arrays, assumed-size arrays for inquiries requiring the size of the last dimension, and variables that are pointer arrays or allocatable arrays. For example, if an array X has explicit bounds in all dimensions, an inquiry such as SIZE(X) can be computed at compile-time, and SIZE(X) + 10 is considered a constant expression.

Constant expressions can be used in any executable statement where general expressions (that is, unrestricted expressions) are permitted.

The following examples show constant expressions:

Expression	Meaning
`2`	An integer literal constant
`-7.5_LARGE`	A real literal constant where `LARGE` is a named integer constant
`(/ 7, (I, I = 1, 10) /)`	An array constructor
`RATIONAL(1, 2+J)`	A structure constructor where `RATIONAL` is a derived type and `J` is a named integer constant
`LBOUND(A,1)+3`	A reference to an inquiry intrinsic function where `A` is an explicit-shape array
`INT(N)`	An intrinsic function reference where `N` is a named constant
`KIND(X)`	An intrinsic function reference where `X` is a real variable with known type parameter
`REAL(10+I)`	An intrinsic function reference where `I` is a named integer constant
`COUNT(A)`	An intrinsic function where `A` is a named logical constant
`I/3.3 + J**3.3`	A numeric expression where `I` and `J` are named integer constants
`SUM(A)`	A reference to a transformational intrinsic function where `A` is a named integer array constant

7.2.9.2 Initialization Expressions

An *initialization expression* is a constant expression restricted as follows:

- The exponentiation operator (`**`) is allowed only when the power (second operand) is of type integer; that is, `X ** Y` is allowed only if `Y` is of type integer.
- Subscripts, section subscripts, starting and ending points of substring ranges, components of structure constructors, and arguments of intrinsic functions must be initialization expressions.

- The elements of array constructors must be initialization expressions or implied-DOs for which the array constructor values and implied-DO parameters are expressions whose primaries are initialization expressions or implied-DO variables.
- An elemental intrinsic function in an initialization expression must have arguments that are initialization expressions and are of type integer or character. These elemental intrinsic functions must return a result of type integer or character.
- A transformational intrinsic function in an initialization expression must be one of the intrinsic functions NULL(3i), REPEAT(3i), RESHAPE(3i), SELECTED_INT_KIND(3i), SELECTED_REAL_KIND(3i), TRANSFER(3i), and TRIM(3i), and must have initialization expressions as arguments; this excludes the use of the transformational functions ALL(3i), ANY(3i), COUNT(3i), CSHIFT(3i), DOT_PRODUCT(3i), EOSHIFT(3i), MATMUL(3i), MAXLOC(3i), MAXVAL(3i), MINLOC(3i), MINVAL(3i), PACK(3i), PRODUCT(3i), SPREAD(3i), SUM(3i), TRANSPOSE(3i), and UNPACK(3i).
- An inquiry intrinsic function is allowed, except that the arguments must either be initialization expressions or variables whose type parameters or bounds inquired about are not assumed, not defined by an ALLOCATE statement, or not defined by pointer assignment.
- Any subexpression enclosed in parentheses must be an initialization expression.

All but the last examples in Section 7.2.9.1, page 250, are initialization expressions. The last are not because initialization expressions cannot contain functions that return results of type real (REAL(3i), LOG(3i)), must not reference certain transformational functions (COUNT(3i), SUM(3i)), or cannot use the exponentiation operator when the second operand is of type real.

The following are examples of initialization expressions:

Expression	Meaning
`SIZE(A, 1) * 4`	An integer expression where A is an array with an explicit shape
`KIND(0.0D0)`	An inquiry function with a constant argument
`SELECTED_REAL_KIND(6, 30)`	An inquiry function with constant arguments
`SELECTED_INT_KIND(2 * R)`	An inquiry function with an argument that is an initialization

expression, where R is a previously declared named constant of type integer

Initialization expressions must be used in the following contexts:

- As initial values following the equal signs in PARAMETER statements and in type declaration statements with the PARAMETER attribute.
- As initial values following the equal signs in type declaration statements for variables.
- As expressions in structure constructors in DATA statement value lists.
- As expressions in default initializers.
- As kind type parameter values in type declaration statements; in this case, they also must be scalar and of type integer.
- As actual arguments for the KIND(3i) dummy argument of the conversion intrinsic functions AINT(3i), ANINT(3i), CHAR(3i), INT(3i), LOGICAL(3i), NINT(3i), REAL(3i), CMPLX(3i); in this case, they also must be scalar and of type integer.
- As case values in the CASE statement; in this situation, they must be scalar and of type integer, logical, or character.
- As subscript or substring range expressions of equivalence objects in an EQUIVALENCE statement; in this case, they must be scalar and of type integer.

Initialization expressions must be used for situations where the value of the expression is needed at compile time. Note that the initialization expressions do not include intrinsic functions that return values of type real, logical, or complex, or have arguments of type real, logical, or complex.

7.2.9.3 Specification Expressions

A *specification expression* is a restricted expression that has a scalar value and is of type integer. Specification expressions are used as bounds for arrays and length parameter values for character entities in type declarations, attribute specifications, dimension declarations, and other specification statements (see Table 7-5, page 259). A *constant specification expression* is a specification expression that is also a constant.

Specification expressions are forms of restricted expressions (defined below), limited in type and rank. Briefly, a restricted expression is limited to constants and certain variables accessible to the scoping unit whose values can be determined on entry to

the program unit before any executable statement is executed. For example, variables that are dummy arguments, are in a common block, are in a host program unit, or are in a module made accessible to the program unit can be evaluated on entry to a program unit. Array constructors, structure constructors, intrinsic function references, and parenthesized expressions made up of these primaries must depend only on restricted expressions as building blocks for operands in a restricted expression.

A *restricted expression* is an expression in which each operation is intrinsic and each primary is limited to one of the following:

- A constant or constant subobject.
- A variable that is a dummy argument with neither the OPTIONAL nor the INTENT(OUT) attribute.
- A variable that is in a common block.
- A variable made accessible from a module.
- A variable from the host program unit.
- A variable accessible through USE association.
- An array constructor in which every expression has primaries that are restricted expressions or are implied-DO variables of the array constructor.
- A structure constructor in which each component is a restricted expression.
- An elemental intrinsic function whose result is of type integer or character and whose arguments are all restricted expressions of type integer or character.
- One of the transformational intrinsic functions (REPEAT(3i), RESHAPE(3i), SELECTED_INT_KIND(3i), SELECTED_REAL_KIND(3i), TRANSFER(3i), or TRIM(3i)), in which each argument is a restricted expression of type integer or character (this excludes the use of the transformational functions ALL(3i), ANY(3i), COUNT(3i), CSHIFT(3i), DOT_PRODUCT(3i), EOSHIFT(3i), MATMUL(3i), MAXLOC(3i), MAXVAL(3i), MINLOC(3i), MINVAL(3i), PACK(3i), PRODUCT(3i), SPREAD(3i), SUM(3i), TRANSPOSE(3i), and UNPACK(3i)).
- An inquiry intrinsic function (except for PRESENT(3i), ALLOCATED(3i), and ASSOCIATED(3i)), in which each argument is one of the following:
  - A restricted expression. Any subscript, section subscript, and starting or ending point of a substring range is a *restricted expression*.

  - A variable whose bounds or type parameters inquired about are not assumed, not defined by an ALLOCATE statement, and not defined by a pointer assignment statement

- A reference to any other intrinsic function in which each argument is a restricted expression.
- A reference to an external function whose result is a nonpointer scalar intrinsic type.
- A reference to a specification function in which each argument is a restricted expression.

A function is a *specification function* if it is a pure function, is not an intrinsic function, is not an internal function, is not a statement function, does not have a dummy procedure argument, and is not RECURSIVE.

ANSI/ISO: The Fortran standard does not specify restricted expressions in which a primary can be a reference to an external function that is not a specification function and with a result that is a nonpointer scalar intrinsic type.

7.2.9.4 Initialization and Specification Expressions in Declarations

The following rules and restrictions apply to the use of initialization and specification expressions in specification statements.

The type and type parameters of a variable or named constant in one of these expressions must be specified in a prior specification in the same scoping unit, in a host scoping unit, in a module scoping unit made accessible to the current scoping unit, or by the implicit typing rules in effect. If the variable or named constant is explicitly given these attributes in a subsequent type declaration statement, it must confirm the implicit type and type parameters.

If an element of an array is referenced in one of these expressions, the array bounds must be specified in a prior specification.

If a specification expression includes a variable that provides a value within the expression, the expression must appear within the specification part of a subprogram. For example, consider variable N in the following program segment:

```
INTEGER  N
COMMON   N
REAL  A (N)
```

N is providing a value that determines the size of the array A. This program segment must not appear in a main program but may appear in the specification part of a subprogram.

A prior specification in the above cases may be in the same specification statement, but to the left of the reference. For example, the following declarations are valid:

```
INTEGER, DIMENSION(4), PARAMETER :: A = (/4, 3, 2, 1 /)
REAL, DIMENSION(A (2)) :: B, C(SIZE(B))
```

B and C are of size 3 (the second element of the array A). The following declaration, however, is invalid because SIZE(E) precedes E:

```
REAL, DIMENSION(2) :: D(SIZE(E)), E
```

7.2.9.5 Uses of the Various Kinds of Expressions

The various kinds of expressions may be somewhat confusing, and it can be difficult to remember where they can be used. To summarize the differences, Section 7.2.4, page 220, specifies the most general kind of expression; the other kinds of expressions are restrictions of the most general kind. The classification of expressions forms two orderings, each from most general to least general, as follows:

- Expression, restricted expression, and specification expression
- Expression, constant expression, and initialization expression

The relationship between the various kinds of expression can be seen in the diagram in Figure 7-2.

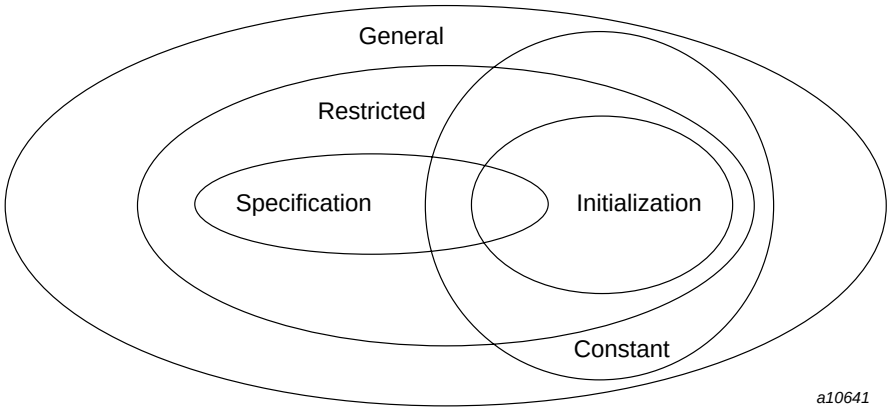

Figure 7-2 Relationships between the kinds of expressions

Initialization expressions are not a subset of specification expressions because the result of an initialization expression can be of any type, whereas the result of a specification expression must be of type integer and scalar. Also, specification expressions are not a subset of initialization expressions because specification expressions allow certain variables (such as dummy arguments and variables in common blocks) to be primaries, where as initialization expressions do not allow such variables.

Table 7-5, page 259, describes the differences between initialization and specification expressions. Table 7-6, page 260, summarizes where each kind of expression is used and gives the restrictions as to their type and rank when used in the various contexts. For example, Table 7-5, page 259, indicates that initialization and specification expressions are different in that initialization expressions can be array valued, whereas specification expressions are scalar. A consequence of this difference, as indicated in Table 7-6, page 260, is that an initialization expression is used in a type declaration statement or a PARAMETER statement to specify the value of a named constant array, whereas a specification expression is used to specify the bounds of an array in a declaration statement.

Table 7-5 Differences and similarities between initialization and specification expressions

	Kind of expression:	
Property:	Initialization	Specification
Character result	Yes	No[1]
Integer result	Yes	Yes
Scalar result	Yes	Yes
Array result	Yes	No
Variables as primaries(limited to dummy arguments, common block objects, host objects, module objects)	No	Yes
Elemental intrinsic functions of type integer and character as primaries	Yes	Yes
Elemental intrinsic functions of type real, complex, logical, and derived type as primaries	No	No
Only constants as primaries	Yes	No
Only constant subscripts, strides, character lengths	Yes	No
One of the transformational intrinsic functions `REPEAT`, `RESHAPE`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND`, `TRANSFER`, or `TRIM` as primaries	Yes	Yes
Inquiry intrinsic functions (not including `ALLOCATED`, `ASSOCIATED`, or `PRESENT`) as primaries	Yes	Yes
Reference to specific functions	No	Yes
Reference to scalar external function with intrinsic type result that is not a specification function [2]	No	Yes

1 Expression results of type character are allowed if they are arguments of an intrinsic function.

2 This is an extension to the Fortran standard.

Table 7-6 Kinds of expressions and their uses

Context	Arb. expr.	Init. expr.	Spec. expr.	Type [3]	Rank [4]
Default integer expression					
Bounds in declaration statement [5]	No	No	Yes	I	Scalar
Lengths in declaration statement [6]	No	No	Yes	I	Scalar
Subscripts and substring ranges in `EQUIVALENCE` statement	No	Yes	No	I	Scalar
Values in `CASE` statement	No	Yes	No	I,L,C	Scalar
Kind parameters in declaration statement	No	Yes	No	I	Scalar
Kind arguments in intrinsics	No	Yes	No	I	Scalar
Initial value in `PARAMETER` and type declaration statement	No	Yes	No	Any	Any
`DATA` implied-`DO` parameters	No	[7]	No	I	Scalar
Assignment	Yes	Yes	Yes	Any	Any
Subscripts in executable statement	Yes	Yes	Yes	I	≤1
Strides in executable statement	Yes	Yes	Yes	I	Scalar
Substring ranges in executable statement	Yes	Yes	Yes	I	Scalar
Expression in `SELECT CASE`	Yes	Yes	Yes	I,L,C	Scalar
`IF` statement	Yes	Yes	Yes	L	Scalar
Arithmetic `IF` statement	Yes	Yes	Yes	I,R	Scalar
`DO` statement	Yes	Yes	Yes	I,R	Scalar

3 "Any" in this column means any intrinsic or derived type.

4 "Any" in this column means that the result may be a scalar or an array of any rank.

5 The relevant declaration statements are type declaration, component definition, `DIMENSION` , `TARGET` , and `COMMON` statements.

6 The relevant declaration statements are type declaration, component definition, `IMPLICIT`, and `FUNCTION` statements.

7 A `DATA` implied-`DO` parameter may be an expression involving intrinsic operations with constants and `DATA` implied-`DO` variables as operands.

Context	Arb. expr.	Init. expr.	Spec. expr.	Type [3]	Rank [4]
Mask in `WHERE` statement	Yes	Yes	Yes	L	Array
Mask in `WHERE` construct	Yes	Yes	Yes	L	Array
`IF-THEN` statement	Yes	Yes	Yes	L	Scalar
`ELSE-IF` statement	Yes	Yes	Yes	L	Scalar
Output item list	Yes	Yes	Yes	Any	Any
I/O specifier values except character `FMT=` specifier [8]	Yes	Yes	Yes	I,C	Scalar
I/O `FMT=` specifier value	Yes	Yes	Yes	C	Any
`RETURN` statement	Yes	Yes	Yes	I	Scalar
Computed `GO TO` statement	Yes	Yes	Yes	I	Scalar
Array constructor implied-`DO` parameters	Yes	Yes	Yes	I	Scalar
Actual arguments	Yes	Yes	Yes	Any	Any
I/O implied-`DO` parameters	Yes	Yes	Yes	I,R	Scalar
`FORALL` *triplet_spec_list*	Yes	Yes	Yes	I	Scalar
`FORALL` *scalar_mask*	Yes	Yes	Yes	L	Scalar
Expressions in statement function definitions	Yes	Yes	Yes	Any	Scalar

Example 1: The expressions `I*3`, `1+2*J`, `5*J/3`, `1`, and `10` in the following statement, are all expressions allowed in subscripts and implied-`DO` parameter expressions in an implied-`DO` list in a `DATA` statement:

```
DATA ((A(I*3), I = 1+2*J, 5*J/3), J = 1, 10) /.../
```

Example 2: An expression such as `RADIX(I)` is not allowed as a data implied-`DO` parameter or subscript of a `DATA` statement object.

An expression such as `N`, where `N` is a variable in the scoping unit that contains the `DATA` statement, is not allowed because `N` is neither a named constant nor an implied-`DO` variable in a containing implied-`DO` list.

8 If the I/O `FMT=` specifier is not of type character, it must be a default integer variable.

Such special expressions in DATA statements are restricted forms of initialization expressions in the sense that the primaries must not include references to any intrinsic function. On the other hand, they are extended forms of initialization expressions in the sense that they permit the use of implied-DO variables that have the scope of the implied-DO list.

7.3 Interpretation of Expressions

The interpretation of an expression specifies the value of the expression when it is evaluated. As with the rules for forming an expression, the rules for interpreting an expression are described from the bottom up, from the interpretation of constants, variables, constructors, and functions to the interpretation of each subexpression to the interpretation of the entire expression.

When an expression is interpreted, the value of each constant and variable is determined. After these are determined, the operations for which the variables and constants are interpreted in precedence order, and a value for the operation is determined by the interpretation rules for each operator. This repeats recursively until the entire expression is interpreted and a value is determined.

The interpretation rules for operations are of two sorts: rules for the intrinsic operations (intrinsic operators with operands of the intrinsic types specified by Table 7-4, page 238) and rules for the defined operations (provided by the programmer using function subprograms). Except for integer division, the intrinsic operations are interpreted by the usual mathematical method, subject to representation limitations imposed by a computer (for example, a finite range of integers, or finite precision of real numbers). The defined operations are interpreted by a function program that is specified in an interface block with a generic specifier of the form OPERATOR(*defined_operator*).

The interpretation rules for an intrinsic or a defined operation are independent of the context in which the expression occurs. That is, the type, type parameters, and interpretation of any expression do not depend on any part of a larger expression in which it occurs.

7.3.1 Interpretation of the Intrinsic Operations

When the arguments of the intrinsic operators satisfy the requirements of Table 7-4, page 238, the operations are intrinsic and are interpreted in the usual mathematical way as described in Table 7-7, page 263, except for integer division. For example, the

binary operator * is interpreted as the mathematical operation multiplication and the unary operator - is interpreted as negation.

Table 7-7 Interpretation of the intrinsic operations

Use of operator	Interpretation
x_1 ** x_2	Raise x_1 to the power x_2
x_1 / x_2	Divide x_1 by x_2
x_1 * x_2	Multiply x_1 by 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
x_1 // x_2	Concatenate x_1 with x_2
x_1 .LT. x_2	x_1 less than x_2
x_1 < x_2	x_1 less than x_2
x_1 .LE. x_2	x_1 less than or equal to x_2
x_1 <= x_2	x_1 less than or equal to x_2
x_1 .GT. x_2	x_1 greater than x_2
x_1 > x_2	x_1 greater than x_2
x_1 .GE. x_2	x_1 greater than or equal to x_2
x_1 >= x_2	x_1 greater than or equal to x_2
x_1 .EQ. x_2	x_1 equal to x_2
x_1 == x_2	x_1 equal to x_2
x_1 .NE. x_2	x_1 not equal to x_2
x_1 /= x_2	x_1 not equal to x_2
x_1 .LG. x_2	x_1 less than or greater than x_2
x_1 <> x_2	x_1 less than or greater than x_2
.NOT. $x2$	True if x_2 is false

Use of operator	Interpretation
x_1 `.AND.` x_2	True if x_1 and x_2 are both true
x_1 `.OR.` x_2	True if x_1 and/or x_2 is true
x_1 `.NEQV.` x_2	True if either x_1 or x_2 is true, but not both
x_1 `.XOR.` x_2	1 if corresponding bits differ; 0 otherwise (bitwise exclusive `.OR.`)
x_1 `.EQV.` x_2	True if both x_1 or x_2 are true or both are false

7.3.1.1 Interpretation of Numeric Intrinsic Operations

Except for exponentiation to an integer power, when an operand for a numeric intrinsic operation does not have the same type or type parameters as the result of the operation, the operand is converted to the type, type parameter, and shape of the result and the operation is then performed. For exponentiation to an integer power, the operation can be performed without the conversion of the integer power, say, by developing binary powers of the first operand and multiplying them together to obtain an efficient computation of the result.

For integer division, when both operands are of type integer, the result is of type integer, but the mathematical quotient is often not an integer. In this case, the result is specified to be the integer value closest to the quotient and between zero and the quotient inclusively.

For exponentiation, there are three cases that need to be further described. When both operands are of type integer, the result is of type integer; when x_2 is negative, the operation x_1 `**` x_2 is interpreted as the quotient `1/(`x_1`**ABS(`x_2`))`. Note that it is subject to the rules for integer division. For example, `4**(-2)` is `0`.

The second case occurs when the first operand is a negative value of type integer or real and the second operand is of type real. A program is invalid if it causes a reference to the exponentiation operator with such operands. For example, a program that contains the expression `(-1.0)**0.5` is an invalid program.

The third case occurs when the second operand is of type real or of type complex. In this case, the result returned is the principal value of the mathematical power function.

7.3.1.2 Interpretation of Standard Nonnumeric Intrinsic Operations

There is only one intrinsic character operation: concatenation. For this operation, the operands must be of type character. The length parameter values can be different. The result is of type character with a character length parameter value equal to the sum of the lengths of the operands. The result consists of the characters of the first operand in order followed by those of the second operand in order. For example, `'Fortran' // '95'` yields the result `'Fortran 95'`.

The intrinsic relational operations perform comparison operations for character and most numeric operands. For these operations, the operands must both be of numeric type or both be of character type. The kind type parameter values of the operands of the numeric types can be different and the lengths of character operands can be different. Complex operands must only be compared for equality and inequality; the reason is that complex numbers are not totally ordered. The result in all cases is of type default logical.

When the operands of an intrinsic relational operation are both numeric, but of different types or type parameters, each operand is converted to the type and type parameters they would have if the two operands were being added. Then, the operands are compared according to the usual mathematical interpretation of the particular relational operator.

When the operands are both of type character, the shorter one is padded on the right with blank padding characters until the operands are of equal length. Then, the operands are compared one character at a time in order, starting from the leftmost character of each operand until the corresponding characters differ. The first operand is less than or greater than the second operand according to whether the characters in the first position where they differ are less than or greater than each other. The operands are equal if both are of zero length or all corresponding characters are equal, including the padding characters.

There is no ordering defined for logical values. However, logical values can be compared for equality and inequality by using the logical equivalence and not equivalence operators `.EQV.` and `.NEQV.`. That is, `L1 .EQV. L2` is true when `L1` and `L2` are both true or both false and is false otherwise. `L1.NEQV. L2` is true if either `L1` or `L2` is true (but not both true) and is false otherwise.

The intrinsic logical operations perform many of the common operations for logical computation. For these operations, the operands must both be of logical type but can have different kind type parameters. If the kind type parameter values are the same, the kind type parameter value of the result is the kind type parameter value of the operands. If the kind type parameter values are different, the kind type parameter

value is the larger of the two kind type parameter values. The values of the result in all cases are specified in Table 7-8, page 266.

Table 7-8 The values of operations involving logical operators

x_1	x_2	.NOT.x_1	x_1.AND.x_2	x_1.OR.x_2	x_1.EQV.x_2	x_1.NEQV.x_2
True	True	False	True	True	True	False
True	False	False	False	True	False	True
False	True	True	False	True	False	True
False	False	True	False	False	True	False

7.3.1.3 Interpretation of Intrinsic Operations with Array Operands

Each of the intrinsic operations can have array operands; however, for the binary intrinsic operations, the operands must both be of the same shape, if both are arrays. When one operand is an array and the other is a scalar, the operation behaves as if the scalar operand were broadcast to an array of the result shape and the operation performed.

For both the unary and binary intrinsic operators, the operation is interpreted element-by-element; that is, the scalar operation is performed on each element of the operand or operands. For example, if A and B are arrays of the same shape, the expression A * B is interpreted by taking each element of A and the corresponding element of B and multiplying them together using the scalar intrinsic operation * to determine the corresponding element of the result. Note that this is not the same as matrix multiplication. As a second example, the expression -A is interpreted by taking each element of A and negating it to determine the corresponding element of the result.

For intrinsic operations that appear in masked assignment statements (in WHERE blocks, ELSEWHERE blocks, or in a WHERE statement), the scalar operation is performed only for those elements selected by the logical mask expression.

Note that there is no order specified for the interpretation of the scalar operations. A processor is allowed to perform them in any order, including all at the same time.

7.3.1.4 Interpretation of Intrinsic Operations with Pointer Operands

The intrinsic operations can have operands with the POINTER attribute. In such cases, each pointer must be associated with a target that is defined, and the value of the target is used as the operand. The target can be scalar or array-valued; the rules for interpretation of the operation are those appropriate for the operand being a scalar or an array, respectively.

Recall that an operand can be a structure component that is the component of a structure variable that is itself a pointer. In this case, the value used for the operand is the named component of the target structure associated with the structure variable. For example, consider the following declarations and assume that the pointer PTR is associated with T:

```
TYPE RATIONAL
   INTEGER :: N, D
END TYPE

TYPE(RATIONAL), POINTER :: PTR
TYPE(RATIONAL), TARGET  :: T
```

If PTR%N appears as an operand, its value is the component N of the target T, namely T%N.

7.3.2 Interpretation of Defined Operations

The interpretation of a defined operation is provided by a function subprogram with an OPERATOR interface (see the *MIPSpro Fortran Language Reference Manual, Volume 2*). When there is more than one function with the same OPERATOR interface, the function giving the interpretation of the operation is the one whose dummy arguments match the operands in order, types, kind type parameters, and rank. For example, for the operation A .PLUS. B, where A and B are structures of the derived type RATIONAL, the following interface specifies that the function RATIONAL_PLUS provides the interpretation of this operation.

Example:

```
INTERFACE OPERATOR(.PLUS.)

   FUNCTION RATIONAL_PLUS(L, R)
      USE RATIONAL_MODULE
      TYPE(RATIONAL), INTENT(IN) :: L, R
      TYPE(RATIONAL)          :: RATIONAL_PLUS
```

```
   END FUNCTION RATIONAL_PLUS

   FUNCTION LOGICAL_PLUS(L, R)
      LOGICAL, INTENT(IN) :: L, R
      LOGICAL             :: LOGICAL_PLUS
   END FUNCTION LOGICAL_PLUS

END INTERFACE
```

A defined operation is declared by using a function with one or two dummy arguments. (Note that the function can be an entry in an external or module function.)

The dummy arguments to the function represent the operands of the operation. If there is only one, the operation is a unary operation; otherwise it is a binary operation. For a binary operation, the first argument is the left operand and the second is the right operand.

There must be an interface block for the function with the generic specifier of the form `OPERATOR (`*defined_operator*`)`.

The types and kind type parameters of the operands in the expression must be the same as those of the dummy arguments of the function.

The function is elemental or the rank of the operands in the expression must match the ranks of the corresponding dummy arguments of the function.

One of the following conditions must be true:

- One of the dummy arguments must be of a derived type
- Both of the dummy arguments are of intrinsic type but do not match the types and kind type parameters for intrinsic operations as specified in Table 7-4, page 238.

As with the intrinsic operations, the type, type parameters, and interpretation of a defined operation are independent of the context of the larger expression in which the defined operation appears. The interpretation of the same defined operation in different contexts is the same, but the results can be different because the results of the procedure being invoked may depend on values that are not operands and that are different for each invocation.

The relational operators `==`, `/=`, `>`, `>=`, `<`, `<`, and `<>` are synonyms for the operators `.EQ.`, `.NE.`, `.GT.`, `.GE.`, `.LT.`, `.LE.`, and `.LG.`even when they are defined operators. It is invalid, therefore, to have an interface block for both `==` and `.EQ.`, for

example, for which the order, types, type parameters, and rank of the dummy arguments of two functions are the same.

ANSI/ISO: The Fortran standard does not describe the <> or .LG. operators.

Defined operations are either unary or binary. An existing unary operator (that is, one that has the same name as an intrinsic operator) cannot be defined as a binary operator unless it is also a binary operator. Similarly, an existing binary operator cannot be defined as a unary operator unless it is also a unary operator. However, a defined operator, .PLUS. say, (that is, one that does not have a name that is the same as an intrinsic operator) can be defined as both a unary and binary operator.

7.4 Evaluation of Expressions

The form of the expression and the meaning of the operations establish the interpretation; once established, a compiler can evaluate the expression in any way that provides the same interpretation with one exception; parentheses specify an order of evaluation that cannot be modified. This applies to both intrinsic operations and defined operations.

Essentially, two sorts of alternative evaluations are allowed:

- The rearrangement of an expression that yields an equivalent expression; for example, A + B + C can be evaluated equivalently as A + (B + C) and would improve the efficiency of the compiled program if B + C were a subexpression whose value had already been computed.
- The partial evaluation of an expression because the value of the unevaluated part can be proven not to affect the value of the entire expression. For example, when one operand of a disjunction (.OR. operator) is known to be true, the other operand need not be evaluated to determine the result of the operation. To be specific, the operand A * B < C need not be evaluated in the expression A < B .OR. A * B < C if A < B is true. This freedom for a compiler to use alternative equivalent evaluations permits the compiler to produce code that is more optimal in some sense (for example, fewer operations, array operations rather than scalar operations, or a reduction in the use of registers or work space), and thereby produce more efficient executable code.

7.4.1 Possible Alternative Evaluations

Before describing in more detail the possible evaluation orders, four basic issues need to be addressed, namely, definition of operands, well-defined operations, functions (and defined operations) with side effects, and equivalent interpretations.

Definition status is described in detail in the *MIPSpro Fortran Language Reference Manual, Volume 2*. For the purpose of evaluation of expressions, it is required that each operand is defined, including all of its parts, if the operand is an aggregate (an array, a structure, or a string). If the operand is a subobject (part of an array, structure, or string), only the selected part is required to be defined. If the operand is a pointer, it must be associated with a target that is defined. An integer operand must be defined with an integer value rather than a statement label.

For the numeric intrinsic operations, the operands must have values for which the operation is well-defined. For example, the divisor for the division operation must be nonzero, and the result of any of the numeric operations must be within the exponent range for the result data type; otherwise, the program is not standard conforming. Other cases include limitations on the operands of the exponentiation operation `**`: for example, a zero-valued first operand must not be raised to a nonpositive second operand; and a negative-valued first operand of type real cannot be raised to a real power.

The third issue is functions with side effects. In Fortran, functions are allowed to have side effects; that is, they are allowed to modify the state of the program so that the state is different after the function is invoked than before it is invoked. This possibility potentially affects the equivalence of two schemes for evaluating an expression, particularly if the function modifies objects appearing in other parts of the expression. However, Fortran prohibits the formation of statements with these kinds of side effects. That is, a function (or defined operation) within a statement is not permitted to change any entity in the same statement. Exceptions are those statements that have statements within them, for example, an `IF` statement or a `WHERE` statement. In these cases, the evaluation of functions in the logical expressions in parentheses after the `IF` keyword or `WHERE` keyword are allowed to affect objects in the statement following the closing right parenthesis. For example, if `F` and `G` are functions that change their actual argument `I`, the following statements are valid, even though `I` is changed when the functions are evaluated:

```
IF (F(I))  A = I
WHERE (G(I))  B = I
```

The following statements are examples of statements that are not valid because `F` and `G` change `I`, which is used elsewhere in the same statement:

```
A(I) = F(I)
Y = G(I) + I
```

It is also not valid for there to be two function references in a statement, if each causes a side effect and the order in which the functions are invoked yields a different final status, even though nothing in the statement is changed.

The fourth issue is equivalent interpretation. For the numeric intrinsic operations, the definition of equivalent interpretation is defined as being mathematical equivalence of the expression, not computational equivalence. Mathematical equivalence assumes exact arithmetic (no rounding errors and infinite exponent range) and thus assumes the rules of commutativity, associativity, and distributivity as well as other rules that can be used to determine equivalence (except that the order of operations specified by parentheses must be honored). Under these assumptions, two evaluations are mathematically equivalent if they yield the same values for all possible values of the operands. `A + B + C` and `A + (B + C)` are thus mathematically equivalent but are not necessarily computationally equivalent because of possible different rounding errors. On the other hand, `I/2` and `0.5 * I` (where `I` is an integer) is a mathematical difference because of the special Fortran definition of integer division.

Table 7-9, page 271, gives examples of equivalent evaluations of expressions where `A`, `B`, and `C` are operands of type real or complex, and `X`, `Y`, and `Z` are of any numeric type. All of the variables are assumed to be defined and have values that make all of the operations in this table well-defined.

Table 7-9 Equivalent evaluations for numeric intrinsic operations

Expression	Equivalent evaluations
`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*A/Z`	`X*(A/Z)`
`X*Y-X*Z`	`X*(Y-Z)`

Expression	Equivalent evaluations
A/B/C	A/(B*C)
A/5.0	0.2*A

Table 7-10 provides examples of alternative evaluations that are not valid and are not mathematically equivalent to the original expression. In addition to the operands of the same names used in Table 7-9, Table 7-10 uses I and J as operands of type integer. Recall that when both operands of the division operator are of type integer, a Fortran integer division truncates the result toward zero to obtain the nearest integer quotient.

Table 7-10 Nonequivalent evaluations of numeric expressions

Expression	Prohibited evaluations
I/2	0.5*I
X*I/J	X*(I/J)
I/J/A	I/(J*A)
(X+Y)+Z	X+(Y+Z)
(X*Y)-(X*Z)	X*(Y-Z)
X*(Y-Z)	X*Y-X*Z

7.4.2 Partial Evaluations

For character, relational, and logical intrinsic operations, the definition of the equivalence of two evaluations is that, given the same values for their operands, each evaluation produces the same result. The definition for equivalence of two evaluations of the same defined operation also requires the results to be the same; note that this definition is more restrictive than for the numeric intrinsic operations, because only mathematical equivalence need be preserved for numeric operations. As described for numeric intrinsic operations, a compiler can choose any evaluation scheme equivalent to that provided by the interpretation. Table 7-11 gives some equivalent schemes for evaluating a few example expressions. For these examples, I and J are of type integer; L1, L2, and L3 are of type logical; and C1, C2, and C3 are of type character of the same length. All of the variables are assumed to be defined.

Table 7-11 Equivalent evaluations of other expressions

Expression	Equivalent evaluations
`I .GT. J`	`(I-J) .GT. 0`
`L1 .OR. L2 .OR. L3`	`L1 .OR. (L2 .OR. L3)`
`L1 .AND. L1`	`L1`
`C3 = C1//C2`	`C3=C1` (`C1`, `C2`, `C3` all of the same length)

These rules for equivalent evaluation schemes allow the compiler to not evaluate any part of an expression that has no effect on the resulting value of the expression. Consider the expression `X*F(Y)`, where `F` is a function and `X` has the value 0. The result will be the same regardless of the value of `F(Y)`; therefore, `F(Y)` need not be evaluated. This shortened evaluation is allowed in all cases, even if `F(Y)` has side effects. In this case every data object that `F` could affect is considered to be undefined after the expression is evaluated (that is, it does not have a predictable value).

The appearance of an array element, an array section, or a character substring reference requires, in most cases, the evaluation of the expressions that are the subscripts, strides, or substring ranges. The type or type parameters of the containing expression are not affected by the evaluation of such subscript, stride, or substring range expressions. It is not necessary for these expressions to be evaluated, if the array section can be shown to be zero-sized or the substring can be shown to be of a zero-length by other means. For example, in the expression `A(1:0) + B(` $expr_1$: $expr_2$), $expr_1$ and $expr_2$ need not be evaluated because the conformance rules for intrinsic operations require that the section of `B` be zero-sized.

The type and type parameters, if any, of a constructor are not affected by the evaluation of any expressions within the constructor.

Parentheses within the expression must be honored. This is particularly important for computations involving numeric values in which rounding errors or range errors may occur or for computations involving functions with side effects.

7.5 Assignment

The most common use of the result of an expression is to give a value to a variable. This is done with an assignment statement. For example:

```
RUG = BROWN + 2.34 / TINT
```

The forms of intrinsic assignment, defined assignment, and masked array assignment are the same, and they are defined as follows:

assignment_stmt	**is** *variable* = *expr*

Assignment establishes a value for the variable on the left of the assignment symbol in an assignment statement. Execution of the assignment statement causes the expression to be evaluated (by performing the computation indicated), and then the value of the expression is assigned to the variable. If the variable has subscripts, section subscripts, or a substring range, the execution of the assignment statement must behave as if they were evaluated before any part of the value is assigned.

There are four forms of the assignment statement: intrinsic assignment, defined assignment, pointer assignment, and masked array assignment.

An assignment statement is a defined assignment if the following conditions are true:

- There is a subroutine subprogram with an assignment interface of the form `ASSIGNMENT (=)`.
- The types, kind type parameters, and ranks (if arrays) of the variable and expression match in order the dummy arguments of the subroutine with the assignment interface.

An assignment statement is a masked array assignment if it appears in a `WHERE` construct, `WHERE` statement, `FORALL` statement, or `FORALL` construct; otherwise, it is an intrinsic or defined assignment.

The form of the pointer assignment statement is similar to the assignment statement except that the assignment operator is `=>` instead of `=`.

The rules and restrictions for each of these forms of assignment are different and are described in the sections below for each form of assignment.

The following examples show the forms of assignment:

Expression	**Meaning**
`X = X + 1.0`	Intrinsic assignment for reals
`CHAR(1:4) = "A123"`	Intrinsic assignment for characters

`STUDENT = B_JONES`	Intrinsic assignment for structures if `STUDENT` and `B_JONES` are of the same derived type
`STRING = "Brown"`	Defined assignment for structure if `STRING` is of derived type and an interface exists that defines the operator = for the types `STRING` and character
`X=Y`	Defined elemental assignment in which `X` and `Y` are both arrays of derived type. In addition, an interface exists that defines the operator = for the type and shape of the array, and the specific is elemental.

`WHERE (Z /= 0.0)`
`A = B / Z`
`END WHERE`
	Masked array assignment

`PTR => X`	Pointer assignment

7.5.1 Intrinsic Assignment

Intrinsic assignment can be used to assign a value to a nonpointer variable of any type or to the target associated with a pointer variable. The assignment statement defines or redefines the value of the variable or the target, as appropriate. The value is determined by the evaluation of the expression on the right side of the equal sign.

The types and kind parameters of the variable and expression in an intrinsic assignment statement must be of the types given in Table 7-12, page 275.

Table 7-12 Types of the variable and expression in an intrinsic assignment

Type of the variable	Type of the expression
Integer	Integer, real, complex, Boolean, Cray pointer
Real	Integer, real, complex, Boolean
Complex	Integer, real, complex
Character	Character
Logical	Logical
Cray pointer	Cray pointer, integer, Boolean
Derived type	Same derived type as the variable

ANSI/ISO: The Fortran standard does not define Boolean or Cray pointer types.

If the variable is an array, the expression must either be a scalar or an array of the same shape as the variable. If the variable is a scalar, the expression must be a scalar. The shape of the variable can be specified in specification statements if it is an explicit-shape array. The shape of the variable can be determined by the section subscripts in the variable, by an actual argument if it is a assumed-shape array, or by an `ALLOCATE` statement or a pointer assignment statement if it is a deferred-shape array. It must not be an assumed-size array unless there is a vector subscript, a scalar subscript, or a section subscript containing an upper bound in the last dimension of the array. The shape of the expression is determined by the shape of the operands, the operators in the expression, and the functions referenced in the expression. A complete description of the shape of an expression appears in Section 7.2.8.3, page 246.

If the variable is a pointer, it must be associated with a target; the assignment statement assigns the value of the expression to the target of the pointer. The pointer can be associated with a target that is an array; the pointer determines the rank of the array, but the extents in each dimension are that of the target.

The evaluation of the expression on the right-hand side of the equal sign, including subscript and section subscript expressions that are part of the expression and part of the variable, must be performed before any portion of the assignment is performed. Before the assignment begins, any necessary type conversions are completed if the variable has a different numeric type or type parameter from the expression. The conversion is the same as that performed by the conversion intrinsic functions `INT`(3i), `REAL`(3i), `CMPLX`(3i), and `LOGICAL`(3i), as specified in Table 7-13. The result of a Boolean expression is of type integer; no conversion is done for BOZ or Hollerith constants.

ANSI/ISO: The Fortran standard does not specify type Boolean or `BOZ` constants in assignment statements.

Table 7-13 Conversion performed on an expression before assignment

Type of the variable	Value assigned
Integer	`INT (`*expr*`, KIND (`*variable*`))`
Real	`REAL (`*expr*`, KIND (`*variable*`))`
Complex	`CMPLX (`*expr*`, KIND (`*variable*`))`
Logical	`LOGICAL (`*expr*`, KIND (`*variable*`))`

An expression can use parts of the variable that appear on the left side of an assignment statement. For example, in evaluating a character string expression on the right-hand side of an assignment, the values in the variable on the left-hand side can be used, as in the following example:

```
DATE(2:5) = DATE(1:4)
```

If the variable and expression are of character type with different lengths, the assignment occurs as follows:

- If the length of the variable is less than that of the expression, the value of the expression is truncated from the right.
- If the length of the variable is greater than the expression, the value of the expression is filled with blanks on the right.

The evaluation of expressions in the variable on the left-hand side, such as subscript expressions, has no effect on, nor is affected by, the evaluation of the expression on the right-hand side, which is evaluated completely first. As usual, this requirement that the expression on the right be evaluated first is specifying the semantics of the statement and does not imply that an implementation must perform the computation in this way if there is an equivalent order that computes the same result.

When a scalar is assigned to an array, the assignment behaves as if the scalar is broadcast to an array of the shape of the variable; it is then in shape conformance with the variable. In the following example, all ten elements of the array `A` are assigned the value `1.0`:

```
REAL A(10)
A = 1.0
```

Array assignment is element-by-element, but the order is not specified. Assume that `A` and `B` are real arrays of size `10`, and the whole array assignment is as follows:

```
A = B
```

The first element of B would be assigned to the first element of A, the second element of B would be assigned to the second element of A, and this would continue element-by-element for 10 elements. The assignment of elements, however, may be performed in any order.

For derived-type intrinsic assignment, the derived types of the variable and the expression must be the same. Derived-type intrinsic assignment is performed component-by-component following the above rules, except when a component is a pointer. For pointer components, pointer assignment between corresponding components is used.

7.5.2 Defined Assignment

A *defined assignment* is an assignment operation provided by a subroutine with an assignment interface `ASSIGNMENT (=)`. When the variable and expression in the assignment statement are of intrinsic types and do not satisfy the type matching rules described in Table 7-12, page 275, or are of derived type, a defined assignment operation will be used, provided the assignment interface and subroutine are accessible. For example, a defined assignment may apply when an integer object is to be assigned to a logical variable, provided a subroutine with a generic assignment interface is accessible. Assignment thus can be extended to types other than the intrinsic types or can replace the intrinsic assignment operation for derived types, if the programmer defines the rules for this assignment in a subroutine. For more information on the assignment interface, see the *MIPSpro Fortran Language Reference Manual, Volume 2*.

An assignment operation is declared by using a subroutine with two dummy arguments.

The dummy arguments to the subroutine represent the variable and the expression, in that order.

There must be an interface block for the subroutine with the generic specifier of the form `ASSIGNMENT(=)`.

The types and kind type parameters of the variable and expression in the assignment statement must be the same as those of the dummy arguments.

The rank of the variable and the expression in the assignment must match the ranks of the corresponding dummy arguments.

One of the following conditions must be true:

- One of the dummy arguments must be of a derived type
- Both of the dummy arguments are of intrinsic type but do not match the types and kind type parameters for intrinsic operations as specified in Table 7-12, page 275.
- The subroutine is elemental, and either both dummy arguments have the same shape or one is scalar.

Example:

```
INTERFACE  ASSIGNMENT (=)

   SUBROUTINE  RATIONAL_TO_REAL(L, R)
      USE  RATIONAL_MODULE
      TYPE(RATIONAL), INTENT(IN)   :: R
      REAL, INTENT(OUT)            :: L
   END SUBROUTINE  RATIONAL_TO_REAL
   SUBROUTINE  REAL_TO_RATIONAL(L, R)
      USE  RATIONAL_MODULE
      REAL, INTENT(IN)              :: R
      TYPE(RATIONAL), INTENT(OUT)   :: L
   END SUBROUTINE  REAL_TO_RATIONAL

END INTERFACE
```

The preceding interface block specifies two defined assignments for two assignment operations in terms of two external subroutines, one for assignment of objects of type RATIONAL to objects of type real, and the other for assignment of objects of type real to objects of type RATIONAL. With this interface block, the following assignment statements are defined:

```
REAL  R_VALUE
TYPE(RATIONAL)  RAT_VALUE

R_VALUE = RATIONAL(1, 2)
RAT_VALUE = 3.7
```

The effect of the defined assignment on variables in the program is determined by the referenced subroutine.

7.5.3 Pointer Assignment

A *pointer* is a variable with the POINTER attribute that points to another object. The term *pointer association* is used for the concept of "pointing to" and the term *target* is used for the object associated with a pointer.

A pointer assignment associates a pointer with a target. If the target is disassociated or undefined, the pointer becomes disassociated or undefined according to the status of the target.

Once a pointer assignment has been executed, the association status of the pointer remains unchanged until one of the following events occurs:

- Another pointer assignment statement is executed that redefines the pointer.
- An ALLOCATE, DEALLOCATE, or NULLIFY statement is executed that redefines the pointer.
- A RETURN statement is executed. This changes the association status only if the pointer is local to the subprogram containing the return and the pointer does not have the SAVE attribute.

The pointer assignment statement is defined as follows:

pointer_assignment_stmt	**is**	*pointer_object* => *target*
pointer_object	**is**	*variable_name*
	or	*structure_component*
target	**is**	*variable*
	or	*expr*

If the pointer object is a variable name, the name must have the POINTER attribute. If the pointer object is a structure component, the component must have the POINTER attribute.

The form of the expression permitted as a target is severely limited.

If the target is a variable, then it must have one of the following characteristics:

- It must have the TARGET attribute.

- It must be the component of a structure, the element of an array variable, or the substring of a character variable that has the TARGET attribute.
- It must have the POINTER attribute.

The type, kind type parameters (including length, if character), and rank of the target must be the same as the pointer object.

If the variable on the right of => has the TARGET attribute, the pointer object on the left of => becomes associated with this target.

If the variable on the right of => has the POINTER attribute and is associated, the pointer object on the left of => points to the same target data that the right hand pointer points to after the pointer assignment statement is executed.

If the variable on the right of => has the POINTER attribute and is disassociated, or if it is a reference to the NULL(3i) intrinsic, the pointer object on the left of => becomes disassociated.

If the variable on the right of => has the POINTER attribute and has an undefined association status, the association status of the pointer object on the left of => becomes undefined.

A pointer assignment statement terminates any previous association for that pointer and creates a new association.

If the pointer object is a deferred-shape array, the pointer assignment statement establishes the extents for each dimension of the array, unless the target is a disassociated or undefined pointer. Except for the case of a disassociated or undefined pointer, the extents are those of the target. For example, if the following statements have been processed, the extents of P1 are those of T, namely 11 and 20, but those of P2 are 1 and 10, because T(:) has a section subscript list:

```
INTEGER, TARGET :: T(11:20)
INTEGER, POINTER :: P1(:), P2(:)
P1 => T
P2 => T(:)
```

The target must not be a variable that is an assumed-size array. If it is an array section of an assumed-size array, the upper bound for the last dimension must be specified.

If the target is an array section, it must not have a vector subscript.

If the target is an expression, it must deliver a pointer result. This implies that the expression must be a user-defined function reference or defined operation that returns

a pointer (there are no intrinsic operations or functions that return results with the POINTER attribute). This also implies that a pointer can never point at a constant because constants cannot have the TARGET attribute.

If the target of a pointer cannot be referenced or defined, the pointer must not be referenced or defined.

If a structure has a component with the POINTER attribute and the structure is assigned a value using an intrinsic derived-type assignment, pointer assignment is used for each component with the POINTER attribute. Also, defined assignment may cause pointer assignment between some components of a structure.

Note that when a pointer appears on the right side of => in a pointer assignment, the pointer on the left side of => is defined or redefined to be associated with the target on the right side of the =>; neither the pointer on the right nor its target are changed in any way.

General examples:

```
MONTH => DAYS(1:30)
PTR => X(:, 5)
NUMBER => JONES % SOCSEC
```

Example 1: In this example, the target is another pointer:

```
REAL, POINTER :: PTR, P
REAL, TARGET :: A
REAL  B
A = 1.0
P => A
PTR => P
B = PTR + 2.0
```

The previous program segment defines A with the value 1.0, associates P with A; then PTR is associated with A as well (through P). The value assigned to B in the regular assignment statement is 3.0, because the reference to PTR in the expression yields the value of the target A which is the value 1.0.

Example 2: In this example, the target is an expression:

```
INTERFACE
   FUNCTION POINTER_FCN(X)
      REAL  X
      REAL, POINTER :: POINTER_FCN
```

```
   END FUNCTION
END INTERFACE

REAL, POINTER :: P
REAL  A

P => POINTER_FCN(A)
```

In this example, the function POINTER_FCN takes a real argument and returns a pointer to a real target. After execution of the pointer assignment statement, the pointer P points to this real target.

Pointers can become associated by using the ALLOCATE statement instead of a pointer assignment statement. Pointers can become disassociated by using the DEALLOCATE or NULLIFY statements, as well as with the pointer assignment statement.

A pointer can be used in an expression (see Section 7.3.1.4, page 267, for the details). Briefly, any reference to a pointer in an expression, other than in a pointer assignment statement, or in certain procedure references, yields the value of the target associated with the pointer. When a pointer appears as an actual argument corresponding to a dummy argument that has the POINTER attribute, the reference is to the pointer and not the value. Note that a procedure must have an explicit interface if it has a dummy argument with the POINTER attribute. For information on explicit interfaces, see the *MIPSpro Fortran Language Reference Manual, Volume 2*.

7.5.4 Masked Array Assignment

Sometimes it is desirable to assign only certain elements of one array to another array. To invert the elements of an array element-by-element, for example, you have to avoid elements that are 0. The masked array assignment is ideal for such selective assignment, as the following example using a WHERE construct illustrates:

```
REAL  A(10,10)
  ...
WHERE (A /= 0.0)
    RECIP_A = 1.0 / A   ! Assign only where the
                        !    elements are nonzero
ELSEWHERE
    RECIP_A = 1.0       ! Use the value 1.0 for
                        !    the zero elements
END WHERE
```

The first array assignment statement is executed for only those elements where the mask `A /= 0.0` is true. Next, the second assignment statement (after the `ELSEWHERE` statement) is executed for only those elements where the same mask is false. If the values of `RECIP_A` where `A` is `0` are never used, this example can be simply written by using the `WHERE` statement, rather than the `WHERE` construct, as follows:

```
WHERE (A /= 0.0) RECIP_A = 1.0 / A
```

A *masked array assignment* is an intrinsic assignment statement in a `WHERE` block, an `ELSEWHERE` block, or a `WHERE` statement for which the variable being assigned is an array. The `WHERE` statement and `WHERE` construct appear to have the characteristics of a control statement or construct such as the `IF` statement and `IF` construct. But there is a major difference: every assignment statement in a `WHERE` construct is executed, whereas at most one block in the `IF` construct is executed. Similarly, the assignment statement following a `WHERE` statement is always executed. For this reason, `WHERE` statements and constructs are discussed here under assignment rather than under control constructs.

In a masked array assignment, the assignment is made to certain elements of an array based on the value of a logical array expression serving as a mask for picking out the array elements. The logical array expression acts as an array-valued condition on the elemental intrinsic operations, functions, and assignment for each array assignment statement in the `WHERE` statement or `WHERE` construct.

As in an intrinsic array assignment, a pointer to an array can be used as the variable, and a pointer to a scalar or an array can be used as a primary in the expression. If the target of the pointer is an array, the target array is masked in the same manner as a nonpointer array used in a masked array assignment.

7.5.4.1 `WHERE` Statement and Construct

The `WHERE` construct is defined as follows:

where_stmt	**is**	`WHERE (`*mask_expr*`)` *where_assignment_stmt*
where_construct	**is**	*where_construct_stmt* [*where_body_construct*] ... [*masked_elsewhere_stmt* [*where_body_construct*] ...] ... [*elsewhere_stmt* [*where_body_construct*] ...] *end_where_stmt*
where_construct_stmt	**is**	[*where_construct_name*`:`] `WHERE (`*mask_expr*`)`
where_body_construct	**is**	*where_assignment_stmt*
	or	*where_stmt*
	or	*where_construct*
where_assignment_stmt	**is**	*assignment_stmt*
mask_expr	**is**	*logical_expr*
masked_elsewhere_stmt	**is**	`ELSEWHERE (`*mask_expr*`)` [*where_construct_name*]
elsewhere_stmt	**is**	`ELSEWHERE` [*where_construct_name*]
end_where_stmt	**is**	`END WHERE` [*where_construct_name*]

The definition of the `WHERE` construct can be simplified to the following general format:

```
WHERE (condition_1) ! STATEMENT_1
   ...
   ELSEWHERE (condition_2)     ! STATEMENT_2
   ...
   ELSEWHERE            ! STATEMENT_3
   ...
   END WHERE
```

The following information applies to the preceding general format:

- Following execution of `STATEMENT_1`, the control mask has the value *condition_1* and the pending control mask has the value `.NOT.`*condition_1*.

- Following execution of STATEMENT_2, the control mask has the value (.NOT.*condition_1*).AND.*condition_2* and the pending control mask has the value (.NOT.*condition_1*).AND.(.NOT.*condition_2*).
- Following execution of STATEMENT_3, the control mask has the value (.NOT.*condition_1*).AND.(.NOT.*condition_2*).
- The false condition values are propogated through the execution of the masked ELSEWHERE statement.

If an array constructor appears in a *where_assignment_stmt* or in a *mask_expr*, the array constructor is evaluated without any masked control. After that, the *where_assignment_stmt* is executed or the *mask_expr* is evaluated.

When a *where_assignment_stmt* is executed, the values of *expr* that correspond to true values of the control mask are assigned to the corresponding elements of *variable*.

A statement that is part of a *where_body_construct* must not be a branch target statement. The value of the control mask is established by the execution of a WHERE statement, a WHERE construct statement, an ELSEWHERE statement, a masked ELSEWHERE statement, or an ENDWHERE statement. Subsequent changes to the value of entities in a *mask_expr* have no effect on the value of the control mask. The execution of a function reference in the mask expression of a WHERE statement is permitted to affect entities in the assignment statement. Execution of an END WHERE has no effect.

If the *where_construct_stmt* has a *where_construct_name*, then the corresponding *end_where_stmt* must specify the same name. If the construct also has an *elsewhere_stmt* or *masked_ elsewhere_stmt*, it must have the same *where_construct_name*. If no *where_construct_name* is specified for the *where_construct*, then the *end_where_stmt* and any *elsewhere_stmt* or *masked_elsewhere_stmt* must have the *where_construct_name*.

In a WHERE construct, only the WHERE construct statement can be labeled as a branch target statement.

The WHERE *block* is the set of assignments between the WHERE construct statement and the ELSEWHERE statement (or END WHERE statement, if the ELSEWHERE statement is not present). The ELSEWHERE *block* is the set of assignment statements between the ELSEWHERE and the END WHERE statements.

Each assignment in the ELSEWHERE block assigns a value to each array element that corresponds with a mask array element that is false.

The ELSEWHERE block is optional; when it is not present, no assignment is made to elements corresponding to mask array elements that are false.

All of the assignment statements are executed in sequence as they appear in the construct (in both the WHERE and ELSEWHERE blocks).

Any elemental intrinsic operation or function within an array assignment statement is evaluated only for the selected elements. In the following example, the square roots are taken only of the elements of A that are positive:

```
REAL  A(10, 20)
  ...
WHERE (A > 0.0)
  SQRT_A = SQRT(A)
END WHERE
```

An elemental function reference is evaluated independently for each element, and only those elements needed in the array assignment are referenced. A *where_assignment_stmt* that is a defined assignment must be elemental.

The expression in the array assignment statement can contain nonelemental function references. *Nonelemental function references* are references to any function or operation defined by a subprogram, without the ELEMENTAL keyword. All elements of the arguments of such functions and returned results (if arrays) are evaluated in full. If the result of the nonelemental function is an array and is an operand of an elemental operation or function, then only the selected elements are used in evaluating the remainder of the expression.

Example 1:

```
REAL  A(2, 3), B(3, 10), C(2, 10), D(2, 10)
INTRINSIC  MATMUL
  ...
WHERE (D < 0.0)
  C = MATMUL(A, B)
END WHERE
```

The matrix product A multiplied by B is performed, yielding all elements of the product, and only for those elements of D that are negative are the assignments to the corresponding elements of C made.

Example 2:

```
WHERE (TEMPERATURES > 90.0)  HOT_TEMPS = TEMPERATURES
WHERE (TEMPERATURES < 32.0)  COLD_TEMPS = TEMPERATURES
```

Example 3:

```
WHERE (TEMPERATURES > 90.0)
   NUMBER_OF_SWEATERS = 0
ELSEWHERE (TEMPERATURES < 0.0)
   NUMBER_OF_SWEATERS = 3
ELSEWHERE (TEMPERATURES < 40)
   NUMBER_OF_SWEATERS = 2
ELSEWHERE
   NUMBER_OF_SWEATERS = 1
ENDWHERE
```

7.5.4.2 Differences between the WHERE Construct and Control Constructs

One major difference between the WHERE construct and control constructs has been described in Section 7.5.4, page 283. Another difference is that no transfers out of WHERE or ELSEWHERE blocks are possible (except by a function reference) because only intrinsic assignment statements are permitted within these blocks. Note that the execution of statements in the WHERE block can affect variables referenced in the ELSEWHERE block (because the statements in both blocks are executed).

7.5.5 FORALL Statement and Construct

FORALL statements and constructs control the execution of assignment and pointer assignment statements with selection by using index values and an optional mask expression.

7.5.5.1 FORALL Construct

The FORALL construct allows multiple assignments, masked array (WHERE) assignments, and nested FORALL constructs and statements to be controlled by a single *forall_triplet_spec_list* and *scalar_mask*.

The format of the FORALL construct is as follows:

forall_construct	**is**	*forall_construct_stmt* [*forall_body_construct*] ... *end_forall_stmt*
forall_construct_stmt	**is**	[*forall_construct_name*:] FORALL *forall_header*
forall_header	**is**	(*forall_triplet_spec_list* [, *scalar_mask_expr*])
forall_triplet_spec	**is**	*index_name* = *subscript* : *subscript* [: *stride*]
subscript	**is**	*scalar_int_expr*
stride	**is**	*scalar_int_expr*
forall_body_construct	**is**	*forall_assignment_stmt*
	or	*where_stmt*
	or	*where_construct*
	or	*forall_construct*
	or	*forall_stmt*
forall_assignment_stmt	**is**	*assignment_stmt*
	or	*pointer_assignment_stmt*
end_forall_stmt	**is**	END FORALL [*forall_construct_name*]

If the *forall_construct_stmt* has a *forall_construct_name*, the *end_forall_stmt* must have the same *forall_construct_name*. If the *end_forall_stmt* has a *forall_construct_name*, the *forall_construct_stmt* must have the same *forall_construct_name*.

The *scalar_mask_expr* must be scalar and of type logical.

A procedure that is referenced in the *scalar_mask_expr*, including one referenced by a defined operation, must be a pure procedure.

A procedure that is referenced in a *forall_body_construct*, including one referenced by a defined operation or assignment, must be a pure procedure.

The *index_name* must be a named scalar variable of type integer.

A *subscript* or *stride* in a *forall_triplet_spec* must not contain a reference to any *index_name* in the *forall_triplet_spec_list* in which it appears.

A statement in a *forall_body_construct* must not define an *index_name* of the *forall_construct*.

A *forall_body_construct* must not be a branch target.

Example:

```
REAL :: A(10, 10), B(10, 10) = 1.0
...
FORALL (I = 1:10, J = 1:10, B(I, J) /= 0.0)
   A(I, J) = REAL (I + J - 2)
   B(I, J) = A(I, J) + B(I, J) * REAL (I * J)
END FORALL
```

Each *forall_body_construct* is executed in the order in which it appears. Each construct is executed for all active combinations of the *index_name* values.

Execution of a *forall_assignment_stmt* that is an *assignment_stmt* causes the evaluation of *expr* and all expressions within *variable* for all active combinations of *index_name* values. After all evaluations have been performed, each *expr* value is assigned to the corresponding *variable*.

Execution of a *forall_assignment_stmt* that is a *pointer_assignment_stmt* causes the evaluation of all expressions within *target* and *pointer_object*, the determination of any pointers within *pointer_object*, and the determination of the target for all active combinations of *index_name* values. After these evaluations have been performed, each *pointer_object* is associated with the corresponding *target*.

In a *forall_assignment_stmt*, a defined assignment subroutine must not reference any variable that becomes defined or a *pointer_object* that becomes associated by the statement.

The following code fragment shows a FORALL construct with two assignment statements. The assignment to array B uses the values of array A computed in the previous statement, not the values A had prior to execution of the FORALL:

```
FORALL (I = 2:N-1, J = 2:N-1 )
   A (I, J) = A(I, J-1) + A(I,J+1) + A(I-1,J) + A(I+1, J)
   B (I, J) = 1.0 / A(I, J)
END FORALL
```

The following code fragment shows how to avoid an error condition by using an appropriate *scalar_mask_expr* that limits the active combinations of the *index_name* values:

```
FORALL (I = 1:N, Y(I) .NE. 0.0)
   X(I) = 1.0 / Y(I)
```

```
END FORALL
```

Each statement in a *where_construct* within a *forall_construct* is executed in sequence. When a *where_stmt*, *where_construct_stmt*, or *masked_elsewhere_stmt* is executed, the statement's *mask_expr* is evaluated for all active combinations of *index_name* values as determined by the outer *forall_constructs*, masked by any control mask corresponding to outer *where_constructs*. Any *where_assignment_stmt* is executed for all active combinations of *index_name* values, masked by the control mask in effect for the *where_assignment_stmt*. The following FORALL construct contains a WHERE statement and an assignment statement:

```
INTEGER A(5,4), B(5,4)
FORALL ( I = 1:5 )
   WHERE ( A(I,:) .EQ. 0 ) A(I,:) = I
   B (I,:) = I / A(I,:)
END FORALL
```

The preceding code is executed with array A as follows::

```
    0 0 0 0
    1 1 1 0
A = 2 2 0 2
    1 0 2 3
    0 0 0 0
```

The result is as follows:

```
    1 1 1 1         1 1 1 1
    1 1 1 2         2 2 2 1
A = 2 2 3 2     B = 1 1 1 1
    1 4 2 3         4 1 2 1
    5 5 5 5         1 1 1 1
```

When a *forall_stmt* or *forall_construct* is executed, the compiler evaluates the *subscript* and *stride* expressions in the *forall_triplet_spec_list* for all active combinations of the *index_name* values of the outer FORALL construct. The set of combinations of *index_name* values for the inner FORALL is the union of the sets defined by these bounds and strides for each active combination of the outer *index_name* values; it also includes the outer *index_name* values. The *scalar_mask_expr* is then evaluated for all combinations of the *index_name* values of the inner construct to produce a set of active combinations for the inner construct. If no *scalar_mask_expr* is specified, the compiler uses .TRUE. as its value. Each statement in the inner FORALL is then executed for each active combination of the *index_name* values. The following FORALL construct

contains a nested FORALL construct. It assigns the transpose of the lower triangle of array A, which is the section below the main diagonal, to the upper triangle of A. The code fragment is as follows:

```
INTEGER A (3, 3)
FORALL (I = 1:N-1 )
   FORALL ( J=I+1:N )
      A(I,J) = A(J,I)
   END FORALL
END FORALL
```

Prior to execution of the preceding code, N=3 and array A is as follows:

```
    0 3 6
A = 1 4 7
    2 5 8
```

After the preceding code is executed, array A is as follows:

```
    0 1 2
A = 1 4 5
    2 5 8
```

You could also use the following FORALL statement to obtain identical results:

```
FORALL ( I = 1:N-1, J=1:N, J > I ) A(I,J) = A(J,I)
```

For more information on the FORALL statement, see Section 7.5.5.2, page 292.

7.5.5.2 FORALL Statement

The FORALL statement allows a single assignment statement or pointer assignment to be controlled by a set of values and an optional mask expression. The format for this statement is as follows:

forall_stmt	**is** FORALL *forall_header forall_assignment_stmt*

Execution of a FORALL statement starts with the evaluation of the *forall_triplet_spec_list* for each *index_name* variable. All possible combinations of the values of the *index_name* variables are considered for execution of the FORALL body. The mask expression, if present, is then evaluated for each combination of *index_name*

values and each combination that has a .TRUE. outcome is in the active combination of *index_name* values. This set of active combinations is then used in executing the FORALL body.

A FORALL statement is equivalent to a FORALL construct that contains a single *forall_body_construct* that is a *forall_assignment_stmt*.

The scope of an *index_name* in a *forall_stmt* is the statement itself.

The following FORALL statement assigns the elements of vector X to the elements of the main diagonal of matrix A:

```
FORALL (I=1:N) A(I,I) = X(I)
```

In the following FORALL statement, array element X(I,J) is assigned the value (1.0 / REAL (I+J-1)) for values of I and J between 1 and N, inclusive:

```
FORALL (I = 1:N, J = 1:N) X(I,J) = 1.0 / REAL (I+J-1)
```

The following statement takes the reciprocal of each nonzero off-diagonal element of array Y(1:N, 1:N) and assigns it to the corresponding element of array X. Elements of Y that are zero or are on the diagonal do not participate, and no assignments are made to the corresponding elements of X:

```
FORALL (I=1:N, J=1:N, Y(I,J) /= 0 .AND. I /= J) X(I,J) = 1.0 / Y(I,J)
```

7.5.5.3 Restrictions on FORALL Constructs and Statements

A *many-to-one assignment* is more than one assignment to the same object or subobject, or association of more than one target with the same pointer, whether the object is referenced directly or indirectly through a pointer. A many-to-one assignment must not occur within a single statement in a FORALL construct or statement. It is possible to assign, or pointer assign, to the same object in different assignment statements in a FORALL construct.

The appearance of each *index_name* in the identification of the left-hand side of an assignment statement is helpful in eliminating many-to-one assignments, but it is not sufficient to guarantee that there will be none. The following code fragment is permitted only if INDEX(1:10) contains no repeated values:

```
FORALL (I = 1:10)
   A (INDEX (I)) = B(I)
END FORALL
```

Within the scope of a FORALL construct, a nested FORALL statement or FORALL construct cannot have the same *index_name*. The *forall_header* expressions within a nested FORALL can depend on the values of outer *index_name* variables.

Controlling Execution

A program performs its computation by executing the statements in sequence from beginning to end. Control constructs and branching statements modify this normal sequential execution of a program. The modification can select blocks of statements and constructs for execution or repetition, or can transfer control to another statement in the program.

As outlined in Chapter 2, page 7, the statements and constructs making up a program are of two sorts, nonexecutable and executable. The *nonexecutable statements* establish the environment under which the program runs. The *executable statements* and *executable constructs*, some of which are *action statements,* perform computations, assign values, perform input/output (I/O) operations, or control the sequence in which the other executable statements and constructs are executed. This chapter describes the latter group of executable statements, the control statements and control constructs.

Control constructs and control statements alter the usual sequential execution order of statements and constructs in a program. This execution order is called the *normal execution sequence*. The control constructs are block constructs and consist of the `IF` construct, the `DO` construct, and the `CASE` construct. A nonblock form of the `DO` construct is also available. Individual statements that alter the normal execution sequence include the `CYCLE` and `EXIT` statements which are special statements for `DO` constructs, branch statements such as arithmetic `IF` statements, various forms of `GO TO` statements, and the statements that cause execution to cease such as the `STOP` and `PAUSE` statements.

With any of the block constructs, a construct name can be used to identify the construct and to identify which `DO` construct, particularly in a nest of `DO` constructs, is being terminated or cycled when using the `EXIT` or `CYCLE` statements.

8.1 The Execution Sequence

There is an established execution sequence for action statements in a Fortran program. Normally, a program or subprogram begins with the first executable statement in that program or subprogram and continues with the next executable statement in the order in which these statements appear. However, there are executable constructs and statements that cause statements to be executed in an order that is different from the order in which they appear in the program. These are either control constructs or branching statements.

There are two basic ways to affect the execution sequence. One is to use an executable construct that selects a block of statements and constructs for execution. The second is to execute a statement that branches to a specific statement in the program. In almost all cases, the use of constructs will result in programs that are more readable and maintainable, so constructs are discussed first, followed by branching statements.

8.2 Blocks and Executable Constructs

A *control construct* consists of one or more blocks of statements and constructs and the control logic that explicitly or implicitly encloses these blocks. Based on a control condition, a block of statements and constructs is selected for execution. A block is a sequence of zero or more statements and constructs, and it is defined as follows:

block	**is**	[*execution_part_construct* . . .]

A block of statements and constructs is treated as a whole. Either the block as a whole is executed or it is not executed. Whether or not the block is executed is determined by expressions in the control logic of the construct. Note that not every statement or construct in the block need be executed; for example, a branch statement early in the block can prevent subsequent statements in the block from being executed. This is still considered a complete execution of the block.

An executable construct consists of one or more blocks of statements surrounded by control statements. The construct usually contains an initial statement before a block and a terminal statement after the block. The construct includes conditions that determine which block in the construct is executed.

There are three executable constructs that contain blocks:

- `IF` construct
- `CASE` construct
- `DO` construct

There is also a construct called the `WHERE` construct that controls array assignment for individual elements (masked array assignment) as opposed to controlling flow of the program statements. Even though it looks like a control construct, it really is a construct for unconditional but masked array assignment. For information on masked array assignment, see Section 7.5.4, page 283.

Construct names are described in the introductory material for blocks and also with each construct and statement that uses them. The name, if used, must appear on the same line as the initial statement of the construct and a matching name must appear on the terminal statement of the construct.

Some of the general rules and restrictions that apply to blocks and control of blocks are as follows:

- The statements of a block are executed in order unless there is a control construct or statement within the block that changes the sequential order.
- A block, as an integral unit, must be completely contained within a construct.
- A block can be empty; that is, it can contain no statements or constructs at all.
- A branching or control construct within a block that transfers to a statement or construct within the same block is permitted.
- Branching to a statement or construct within a block from outside the block is prohibited. (Even branching to the first executable statement within a block from outside the block is prohibited.)
- Exiting from a block can be done from anywhere within the block.
- References to procedures are permitted within a block.

8.3 IF Construct and IF Statement

An IF construct selects at most one block of statements and constructs within the construct for execution. The IF statement controls the execution of only one statement; in previous Fortran standards it was called the logical IF statement. The arithmetic IF statement is not the same as the IF statement; it is a branching statement that is designated as obsolescent.

8.3.1 The IF Construct

The IF construct contains one or more executable blocks. At most one block is executed. It is possible for no block to be executed when there is no ELSE statement.

8.3.1.1 Form of the IF Construct

The IF construct is defined as follows:

if_construct	**is**	*if_then_stmt* *block* [*else_if_stmt* *block*] ... [*else_stmt* *block*] *end_if_stmt*
if_then_stmt	**is**	[*if_construct_name* :] IF (*scalar_logical_expr*) THEN
else_if_stmt	**is**	ELSE IF (*scalar_logical_expr*) THEN [*if_construct_name*]
else_stmt	**is**	ELSE [*if_construct_name*]
end_if_stmt	**is**	END IF [*if_construct_name*]

Branching to an ELSE IF or an ELSE statement is prohibited.

Branching to an END IF statement is allowed from any block within the IF construct.

If a construct name appears on the IF-THEN statement, the same name must appear on the corresponding END IF statement.

The construct names on the ELSE IF and ELSE statements are optional, but if present must be the same name as the one on the IF-THEN statement. If one such ELSE IF or ELSE statement has a construct name, the others are not required to have a construct name.

The same construct name must not be used for different named constructs in the same scoping unit; thus, two IF constructs must not be both named INNER in the same executable part, for example.

8.3.1.2 Execution of the IF Construct

The logical expressions are evaluated in order until one is found to be true. The block following the first true condition is executed, and the execution of the IF construct terminates. Subsequent true conditions in the construct have no effect. There may be no logical expressions found to be true in the construct. In this case, the block following the ELSE statement is executed if there is one; otherwise, no block in the construct is executed.

Figure 8-1 indicates the execution flow for an IF construct.

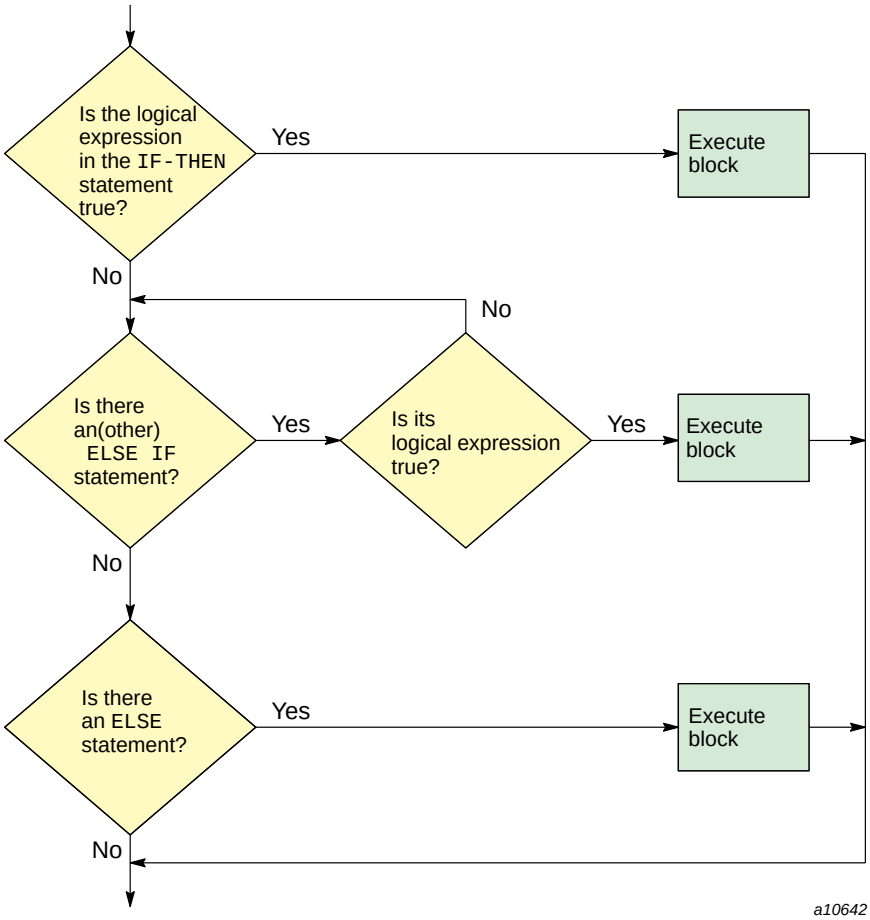

Figure 8-1 Execution flow for an `IF` construct

Example:

```
IF (I < J) THEN
   X = Y + 5.0
ELSE IF (I > 100) THEN
   X = 0.0
   Y = -1.0
ELSE
   X = -1.0
```

```
   Y = 0.0
END IF
```

If `I` is less than `J`, the statement `X = Y + 5.0` is executed and execution proceeds following the `END IF` statement. If `I .GE. J` and if `I` > 100, the two statements following the `ELSE IF` statement are executed and execution proceeds following the `END IF` statement. If neither of these conditions is true, the block after the `ELSE` statement is executed.

8.3.2 `IF` Statement

The `IF` statement controls a single action statement, as opposed to a block of statements.

8.3.2.1 Form of the `IF` Statement

The `IF` statement is defined as follows:

if_stmt	**is**	`IF` (*scalar_logical_expr*) *action_stmt*

Example:

```
IF (S < T) S = 0.0
```

8.3.2.2 Execution of the `IF` Statement

The scalar logical expression is evaluated. If true, the action statement is executed. If false, the action statement is not executed, and control passes to the next statement in the program.

The action statement must not be an `IF` statement or an `END` statement for a program, function, or subroutine. Note that the action statement cannot be any of the other `END` statements, such as `END DO` because they are not action statements.

If the logical expression contains a function reference, its evaluation may have side effects that modify the action statement.

A complete list of the action statements can be found in Section 2.5, page 23. Action statements change the definition state of variables or the condition of the I/O system, or are control statements. Specification statements such as type declaration

statements, FORMAT statements, and ENTRY statements are not action statements. Note that constructs are not action statements.

8.4 CASE Construct

The CASE construct, like the IF construct, consists of a number of blocks, of which at most one is selected for execution. The selection is based on the value of the scalar expression in the SELECT CASE statement at the beginning of the construct; the value of this expression is called the *case index*. The case selected is the one for which the case index matches a case selector value in a CASE statement. There is an optional default case that, in effect, matches all values not matched by any other CASE statement in the construct.

8.4.1 Form of the CASE Construct

The general form of the CASE construct is as follows:

```
[ case_construct_name : ] SELECT CASE (case_expression)
      [ CASE (case_value_range_list) [ case_construct_name ]
         block ] ...
      [ CASE DEFAULT [ case_construct_name ]
         block ]
   END SELECT [ case_construct_name ]
```

The case construct is defined as follows:

case_construct	**is**	*select_case_stmt* [*case_stmt* *block*] ... *end_select_stmt*
select_case_stmt	**is**	[*case_construct_name* :] SELECT CASE (*case_expr*)
case_stmt	**is**	CASE *case_selector* [*case_construct_name*]
end_select_stmt	**is**	END SELECT [*case_construct_name*]
case_expr	**is**	*scalar_int_expr*

	or	*scalar_char_expr*
	or	*scalar_logical_expr*
case_selector	**is**	(*case_value_range_list*)
	or	DEFAULT
case_value_range	**is**	*case_value*
	or	*case_value* :
	or	: *case_value*
	or	*case_value* : *case_value*
case_value	**is**	*scalar_int_initialization_expr*
	or	*scalar_char_initialization_expr*
	or	*scalar_logical_initialization_expr*

The statement containing the keywords SELECT CASE is called the SELECT CASE statement. The statement beginning with the keyword CASE is called the CASE statement. The statement beginning with the keywords END SELECT is called the END SELECT statement. A case value range list enclosed in parenthesis or the DEFAULT keyword is called a *case selector*.

If a construct name is present on a SELECT CASE statement, it must also appear on the END SELECT statement.

Any of the case selector statements may or may not have a construct name. If one does, it must be the same name as the construct name on the SELECT CASE statement.

A CASE statement with the case selector DEFAULT is optional. If it is present, it is not required to be the last CASE statement.

The *case_expr* must be a scalar expression of type integer, character, or logical.

Within a particular CASE construct, the case expression and all case values must be of the same type and must have the same kind type parameter values. If the character type is used, different character lengths are allowed.

Each *case_value* must be a scalar initialization expression of the same type as the case expression. An *initialization expression* is an expression that can be evaluated at compile time; that is, essentially, a constant expression.

The colon forms of the case values expressing a range can be used for expressions in the construct of type integer and character but not type logical. For example, the following CASE statement would select all character strings that collate between BOOK and DOG, inclusive:

```
CASE ('BOOK':'DOG')
```

After expression evaluation, there must be no more than one case selector that matches the case index. In other words, overlapping case values and case ranges are prohibited.

Branching to the END SELECT statement is allowed only from within the construct. Branching to a CASE statement is prohibited; branching to the SELECT CASE statement is allowed, however.

The following example shows the CASE construct:

```
! Compute the area with a formula  appropriate for
! the shape of the object
FIND_AREA: &
   SELECT CASE (OBJECT)
     CASE (CIRCLE)
         AREA = PI * RADIUS ** 2
     CASE (SQUARE)
         AREA = SIDE * SIDE
     CASE (RECTANGLE)
         AREA = LENGTH * WIDTH
     CASE DEFAULT
        PRINT*, "Unable to compute area."
   END SELECT  FIND_AREA
```

8.4.2 Execution of the CASE Construct

The case index (the scalar expression) in the SELECT CASE statement is evaluated in anticipation of matching one of the case values preceding the blocks. The case index must match at most one of the selector values. The block following the case matched is executed, the CASE construct terminates, and control passes to the next executable statement or construct following the END SELECT statement of the construct. If no match occurs and the CASE DEFAULT statement is present, the block after the CASE DEFAULT statement is selected. If there is no CASE DEFAULT statement, the CASE construct terminates, and the next executable statement or construct following the END SELECT statement of the construct is executed. If the case value is a single value, a

match occurs if the index is equal to the case value (determined by the rules used in evaluating the equality or equivalence operator). If the case value is a range of values, there are three possibilities to determine a match depending on the form of the range:

Case value range	Condition for a match
$case\_value_1$: $case\_value_2$	$case\_value_1 \leq case\_index \leq case\_value_2$
case_value :	*case_value* ≤ *case_index*
: *case_value*	*case_value* ≥ *case_index*

Figure 8-2, page 304, illustrates the execution of a CASE construct.

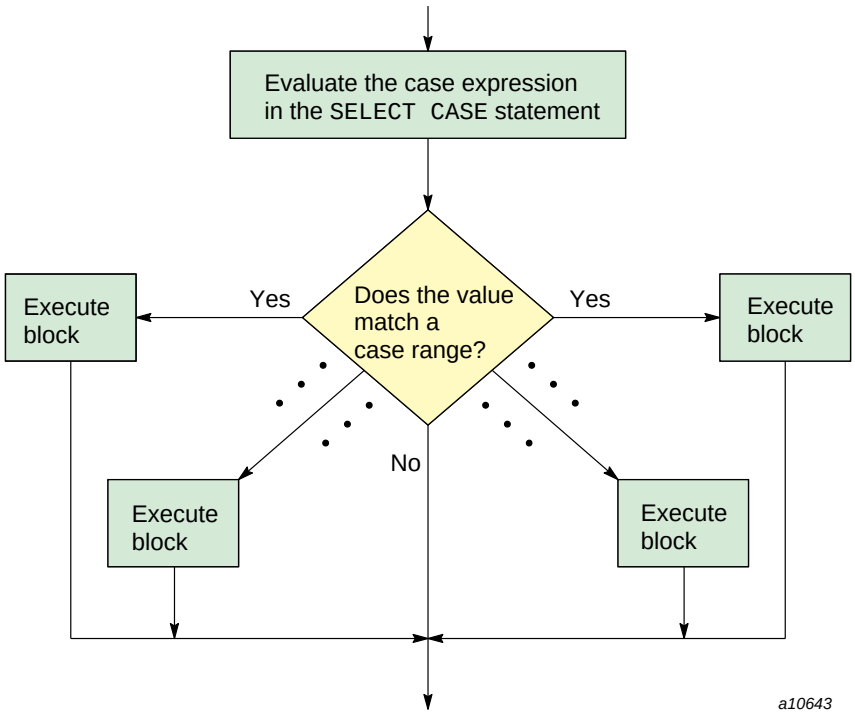

Figure 8-2 Execution flow for a CASE construct

Example 1:

```
INDEX = 2
SELECT CASE (INDEX)
  CASE (1)
     X = 1.0
  CASE (2)
     X = 2.0
  CASE DEFAULT
     X = 99.0
END SELECT
```

The case expression `INDEX` has the value `2`. The block following the case value of `2` is executed; that is, the statement `X = 2.0` is executed, and execution of the `CASE` construct terminates.

Example 2:

```
COLOR = 'GREEN'
SELECT CASE (COLOR)
  CASE ('RED')
     STOP
  CASE ('YELLOW')
     CALL  PROCEED_IF_YOU_CAN_SAFELY
  CASE ('GREEN')
     CALL  GO_AHEAD
END SELECT
```

This example uses selectors of type character. The expression `COLOR` has the value `GREEN`, and therefore the procedure `GO_AHEAD` is executed. When it returns, the execution of the `CASE` statement terminates, and the executable statement after the `END SELECT` statement executes next.

8.5 DO Construct

The `DO` construct controls the number of times a sequence of statements and constructs within the range of a loop is executed. There are three steps in the execution of a `DO` construct:

1. If execution of the DO construct is controlled by a DO variable, the expressions representing the parameters that determine the number of times the range is to be executed are evaluated (step 1 of Figure 8-3).
2. A decision is made as to whether the range of the loop is to be executed (step 2 of Figure 8-3).
3. If appropriate, the range of the loop is executed (step 3a of Figure 8-3); the DO variable, if present, is updated (step 3b of Figure 8-3, page 306); and step 2 is repeated.

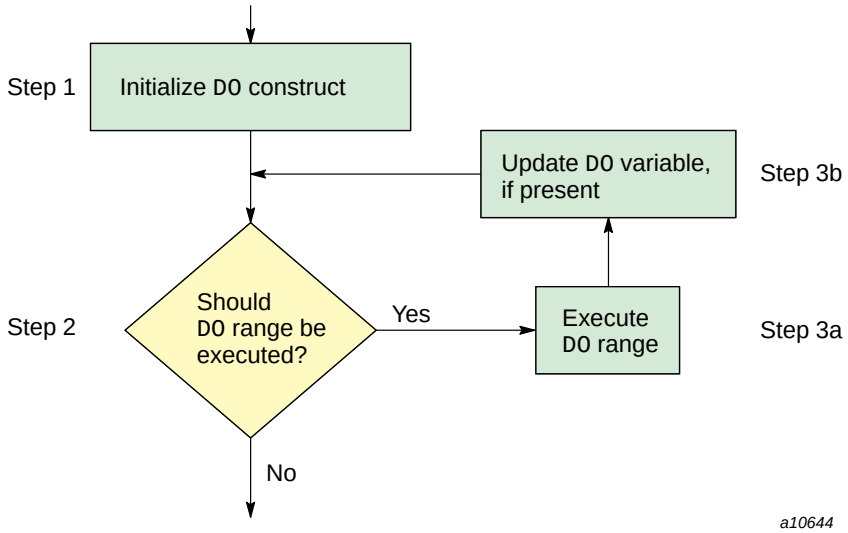

Figure 8-3 Execution flow for a DO construct

DO loop execution can be controlled by a DO variable that is incremented a certain number of times as prescribed in the initial DO statement, a DO WHILE construct, or a simple DO.

There are two basic forms of the DO construct, the block DO and the nonblock DO.

Modern programming practice favors the block DO form, so the block DO form is the recommended construct. The nonblock DO form is obsolescent. The block DO contains all of the functionality of the nonblock DO and vice versa. Indeed, both forms of DO construct permit the DO WHILE and DO forever forms of loops. The feature

distinguishing the two forms is that the block DO construct is always terminated by an END DO or CONTINUE statement whereas the nonblock DO construct either terminates with an action statement or shares a termination statement with another nonblock DO construct.

The following example shows a block DO construct:

```
DO I = 1, N
  SUM = SUM + A(I)
END DO
```

8.5.1 Form of the Block DO Construct

The block DO construct is a DO construct that terminates with an END DO statement or a CONTINUE statement that is not shared with another DO construct.

The block DO construct is defined as follows:

block_do_construct	**is**	*do_stmt* *do_block* *end_do*
do_stmt	**is**	*label_do_stmt*
	or	*nonlabel_do_stmt*
label_do_stmt	**is**	[*do_construct_name* :] DO *label* [*loop_control*]
nonlabel_do_stmt	**is**	[*do_construct_name* :] DO [*loop_control*]
loop_control	**is**	[,] *do_variable* = *scalar_int_expr* *scalar_int_expr* [, *scalar_int_expr*]
	or	[,] WHILE (*scalar_logical_expr*)
do_variable	**is**	*scalar_int_variable*
do_block	**is**	*block*
end_do	**is**	*end_do_stmt*
	or	*continue_stmt*
end_do_stmt	**is**	END DO [*do_construct_name*]

The DO variable must be a scalar named variable of type integer. (This excludes variables that are array elements, structures, and components of structures.)

Each scalar numeric expression in the loop control must be of type integer.

If the DO statement of a block DO construct has a construct name, the corresponding *end_do* must be an END DO statement that has the same construct name. If the DO statement of a block DO construct does not have a construct name, the corresponding *end_do* must not have a construct name.

If the DO statement does not contain a label, the corresponding *end_do* must be an END DO statement. If the DO statement does contain a label, the corresponding *end_do* must be identified with the same label. By definition, a block DO construct cannot share its terminal statement with another DO construct, even if it is a labeled statement. If a DO construct does share its terminal statement with another DO construct, it is a nonblock DO construct. Refer to the following examples:

```
SUM = 0.0
DO I = 1, N
  SUM = SUM + X(I) ** 2
END DO

FOUND = .FALSE.
I = 0
DO WHILE (.NOT. FOUND .AND. I < LIMIT )
  IF (KEY == X(I))  THEN
    FOUND = .TRUE.
  ELSE
    I = I + 1
  END IF
END DO

NUM_ITERS = 0
DO
  ! F and F_PRIME are functions
  X1 = X0 - F(X0) / F_PRIME(X0)
  IF (ABS(X1-X0) < SPACING(X0) .OR. &
      NUM_ITERS > MAX_ITERS)  EXIT
  X0 = X1
  NUM_ITERS = NUM_ITERS + 1
END DO
```

```
INNER_PROD = 0.0
DO 10 I = 1, 10
   INNER_PROD = INNER_PROD + X(I) * Y(I)
10 CONTINUE

LOOP: DO I = 1, N
        Y(I) = A * X(I) + Y(I)
      END DO LOOP
```

Although a DO construct can have both a label and a construct name, use of both is not in the spirit of modern programming practice where the use of labels is minimized.

8.5.2 Form of the Nonblock DO Construct

The nonblock DO construct is a DO construct that either shares a terminal statement with another DO construct, or the terminal statement is an action statement. The nonblock DO construct always uses a label to specify the terminal statement of the construct.

Note: The Fortran standard has declared the nonblock DO construct to be obsolescent.

The nonblock DO construct that ends with an action statement is defined as follows:

nonblock_do_construct	**is**	*action_term_do_construct*
	or	*outer_shared_do_construct*
action_term_do_construct	**is**	*label_do_stmt* *do_body* *do_term_action_stmt*
do_body	**is**	[*execution_part_construct*] ...
do_term_action_stmt	**is**	*action_stmt*

The nonblock DO construct that shares a termination statement is defined as follows:

nonblock_do_construct	**is**	*action_term_do_construct*
	or	*outer_shared_do_construct*
outer_shared_do_construct	**is**	*label_do_stmt* *do_body* *shared_term_do_construct*
shared_term_do_construct	**is**	*outer_shared_do_construct*
	or	*inner_shared_do_construct*
inner_shared_do_construct	**is**	*label_do_stmt* *do_body* *do_term_shared_stmt*
do_term_shared_stmt	**is**	*action_stmt*

The last statement in a nonblock DO construct (the statement in which the loop label is defined), is called the DO *termination* or *terminal statement* of that construct.

An *action_term_do_construct* is a nonblock DO construct that does not share its DO termination with any other nonblock DO construct. An *outer_shared_do_construct* is a nonblock DO construct that shares its DO termination with at least one inner nonblock DO construct.

The DO termination of an *action_term_do_construct* must not be one of the following:

- A GO TO statement
- A RETURN statement
- A STOP statement
- An EXIT statement
- A CYCLE statement
- An END statement for a program or subprogram
- An arithmetic IF statement

Note that a *do_term_action_stmt* is an *action_stmt*. A CONTINUE statement is an *action_stmt*, but by definition, if a DO construct ends with a CONTINUE statement, it is a block DO construct. Also note that a *do_term_action_stmt* cannot be any kind of END statement; END statements other than program or subprogram END statements are not specifically named in the preceding list because they are not *action_stmt*s.

The DO termination must be identified with a label and the corresponding DO statement must refer to the same label.

The DO termination of an *outer_shared_do_construct* must not be a GO TO statement, a RETURN statement, a STOP statement, an EXIT statement, a CYCLE statement, an END statement for a program or subprogram, an arithmetic IF statement, or an assigned GO TO statement. Note that DO termination cannot be any other END statement because the other END statements are not *action_stmt*s.

The DO termination must be identified with a label and all DO statements of the shared termination DO construct must refer to the same label.

The following are examples of DO constructs that are nonblock DO constructs because the DO terminations are action statements.

```
   PROD = 1.0
   DO 10 I = 1, N
10   PROD = PROD * P(I)

   FOUND = .FALSE.
   I = 0
   DO 10 WHILE (.NOT. FOUND .AND. I < LIMIT)
     I = I + 1
10   FOUND = KEY == X(I)
```

The following are examples of DO constructs that are nonblock DO constructs because the DO terminations are shared.

```
   DO 10 I = 1, N
     DO 10 J = 1, N
10     HILBERT(I, J) = 1.0 / REAL(I + J)

   DO 20 I = 1, N
     DO 20 J = I+1, N
       T = A(I, J); A(I, J) = A(J, I); A(J, I) = T
20 CONTINUE
```

8.5.3 Range of a DO Construct

The range of a DO construct consists of all statements and constructs following the DO statement, bounded by and including the terminal statement. The DO range can contain constructs, such as an IF construct, a CASE construct, or another DO construct, but the inner construct or constructs must be entirely enclosed within the nearest

outer construct. If the range of a DO construct contains another construct, the constructs are said to be *nested*.

A branch to a statement within the range of a DO construct is diagnosed by the compiler as being unsafe.

ANSI/ISO: The Fortran standard prohibits a branch into the range of a DO construct.

8.5.4 Active and Inactive DO Constructs

A DO construct is either active or inactive. A DO construct becomes *active* when the DO statement is executed. A DO construct becomes *inactive* when any one of the following situations occurs:

- The iteration count is zero at the time it is tested.
- The WHILE condition is false at the time it is tested.
- An EXIT statement is executed that causes an exit from the DO construct or any DO construct containing the DO construct.
- A CYCLE statement is executed that causes cycling of any DO construct containing the DO construct.
- There is a transfer of control out of the DO construct.
- A RETURN statement in the DO construct is executed.
- The program terminates for any reason.

8.5.5 Execution of DO Constructs

There are three forms of DO constructs, each with its own rules for execution: a DO construct with an iteration count, a DO WHILE construct, and a simple DO construct.

8.5.5.1 DO Construct with an Iteration Count

In this case, an iteration count controls the number of times the range of the loop is executed.

The general form of a DO statement using an iteration count is as follows:

```
DO [ label ] [ , ] do_variable = start_expr, end_expr [, inc_expr ]
```

The DO variable and the expressions may be of type integer. The following are examples of the iterative DO statement:

```
DO 10 I = 1, N
DO, J = -N, N
DO K = N, 1, -1
```

8.5.5.1.1 The Iteration Count

An iteration count is established for controlling the number of times the program executes the range of the DO construct. This is done by evaluating the expressions *start_expr*, *end_expr*, and *inc_expr*, and converting these values to the type of the DO variable. For example, let m_1, m_2, and m_3 be the values obtained:

- m_1 is the initial value of the DO variable
- m_2 is the terminal value the DO variable may assume
- m_3 is an optional parameter, specifying the DO variable increment

The value of m_3 must not be zero. If $expression_3$ is not present, m_3 is given the value 1. The iteration count is calculated from the following formula:

MAX (INT ((m_2 - m_1 + m_3) / m_3), 0)

Note that the iteration count is 0 if one of the following conditions is true:

- $m_1 > m_2$ and $m_3 > 0$
- $m_1 < m_2$ and $m_3 < 0$

8.5.5.1.2 Controlling Execution of the Range of the DO Construct

The steps that control the execution of the range of the DO construct are as follows:

1. The DO variable is set to m_1, the initial parameter (step 1 of Figure 8-3, page 306).
2. The iteration count is tested (step 2 of Figure 8-3, page 306). If it is 0, the DO construct terminates.
3. If the iteration count is not 0, the range of the DO construct is executed (step 3a of Figure 8-3, page 306). The iteration count is decremented by 1, and the DO

variable is incremented by m_3 (step 3b of Figure 8-3, page 306). Steps 2 and 3 are repeated until the iteration count is 0.

After termination, the DO variable retains its last value, the one that it had when the iteration count was tested and found to be 0.

The DO variable must not be redefined or become undefined during the execution of the range of the DO construct. Note that changing the variables used in the expressions for the loop parameters during the execution of the DO construct does not change the iteration count; it is fixed when execution of the DO construct starts.

```
  N = 10
  SUM = 0.0
  DO 2 I = 1, N
     SUM = SUM + X(I)
     N = N + 1
2 CONTINUE
```

The loop is executed 10 times; after execution I=11 and N=20.

```
X = 20.
DO I = 1, 2
   DO J = 1, 5
   X = X + 1.0
END DO
END DO
```

The inner loop is executed 10 times. After completion of the outer DO construct, J=6, I=3, and X=30.

If the second DO statement had been the following, the inner DO construct would not have executed at all; X would remain equal to 20; J would equal 5, its initial value; and I would be equal to 3:

```
DO J = 5, 1
```

8.5.5.2 DO WHILE Construct

The DO WHILE form of the DO construct provides the ability to repeat the DO range while a specified condition remains true.

The general form of the DO WHILE statement is as follows:

```
DO [ label ] [ , ] WHILE (expression)
```

The following examples show the DO WHILE statement:

```
DO WHILE( K >= 4 )
DO 20 WHILE( .NOT. FOUND )
```

The DO range is executed repeatedly. Prior to each execution of the DO range, the logical expression is evaluated. If it is true, the range is executed; if it is false, the DO WHILE construct terminates.

```
SUM = 0.0
I = 0
DO WHILE (I < 5)
   I = I + 1
   SUM = SUM + I
END DO
```

The loop would execute five times, after which SUM = 15.0 and I = 5.

8.5.5.3 Simple DO Construct

A DO construct without any loop control provides the ability to repeat statements in the DO range until the DO construct is terminated explicitly by some statement within the range. When the end of the DO range is reached, the first executable statement of the DO range is executed next.

The form of the simple DO statement is as follows:

```
DO [ label ]
```

Example:

```
DO
   READ *, DATA
   IF (DATA < 0) STOP
   CALL PROCESS (DATA)
END DO
```

The DO range executes repeatedly until a negative value of DATA is read, at which time the DO construct (and the program, in this case) terminates. The previous example, rewritten using a label, appears as follows:

```
    DO 100
       READ *, DATA
       IF (DATA <0) STOP
       CALL PROCESS(DATA)
100 CONTINUE
```

8.5.6 Altering the Execution Sequence Within the Range of a DO Construct

There are two statements that can appear only in the range of a DO construct that alter the execution sequence of the DO construct. One is the EXIT statement; the other is the CYCLE statement. Other statements, such as branch statements, the RETURN statement, and the STOP statement, also alter the execution sequence but are not restricted to DO constructs as are the EXIT and CYCLE statements.

8.5.6.1 EXIT Statement

The EXIT statement immediately causes termination of the DO construct. No further action statements within the range of the DO construct are executed. It can appear in either the block or nonblock form of the DO construct, except that it must not be the DO termination of the nonblock form.

The EXIT statement is defined as follows:

exit_stmt	**is**	EXIT [*do_construct_name*]

If the EXIT statement has a construct name, it must be within the DO construct with the same name; when it is executed, the named DO construct is terminated as well as any DO constructs containing the EXIT statement and contained within the named DO construct.

If the EXIT statement does not have a construct name, the innermost DO construct in which the EXIT statement appears is terminated.

Example 1: In the following example, the DO construct has a construct name, LOOP_8; the DO range is executed repeatedly until the condition in the IF statement is met, when the DO construct terminates:

```
LOOP_8 : DO
   ...
   IF (TEMP == INDEX) EXIT LOOP_8
```

```
   ...
END DO LOOP_8
```

Example 2: In the following example, when the EXIT statement in the IF statement is executed, both the inner loop and the outer loop are terminated:

```
OUTER_LOOP:  DO I = 1, 10
INNER_LOOP:  DO J = 1, 10
               ...
               IF (TEMP == INDEX) EXIT OUTER_LOOP
               ...
               END DO INNER_LOOP
               ...
             END DO OUTER_LOOP
```

8.5.6.1.1 CYCLE Statement

In contrast to the EXIT statement, which terminates execution of the DO construct entirely, the CYCLE statement interrupts the execution of the DO range and begins a new cycle of the DO construct, with appropriate adjustments made to the iteration count and DO variable, if present. It can appear in either the block or nonblock form of the DO construct, except it must not be the DO termination of the nonblock form. When the CYCLE statement is executed in the nonblock form, the DO termination is not executed.

The CYCLE statement is defined as follows:

cycle_stmt	**is** CYCLE [*do_construct_name*]

If the CYCLE statement has a construct name, it must be within the DO construct with the same name; when it is executed, the execution of the named DO construct is interrupted, and any DO construct containing the CYCLE statement and contained within the named DO construct is terminated.

If the CYCLE statement does not have a construct name, the innermost DO construct in which the CYCLE statement appears is interrupted.

The CYCLE statement can be used with any form of the DO statement and causes the next iteration of the DO range to begin, if permitted by the condition controlling the loop.

Upon interruption of the DO construct, if there is a DO variable, it is updated and the iteration count is decremented by 1. Then, in all cases, the processing of the next iteration begins.

In the following example, the loop is executed as long as INDEX is nonnegative. If INDEX is negative, the loop is terminated. If INDEX is 0, the latter part of the loop is skipped.

```
DO
   . . .
   INDEX = . . .
   . . .
   IF (INDEX < 0) EXIT
   IF (INDEX == 0) CYCLE
   . . .
END DO
```

8.6 Branching

Branching is a transfer of control from the current statement to another statement or construct in the program unit. A branch alters the execution sequence. This means that the statement or construct immediately following the branch is usually not executed. Instead, some other statement or construct is executed, and the execution sequence proceeds from that point. The terms *branch statement* and *branch target statement* are used to distinguish between the transfer statement and the statement to which the transfer is made.

An example of branching is provided by the GO TO statement. It is used to transfer control to a statement in the execution sequence that is usually not the next statement in the program, although this is not prohibited.

The statements that can be branch target statements are those classified as action statements plus the IF-THEN statement, SELECT CASE statement, DO statement, WHERE statement, and a few additional statements in limited situations.

The additional statements that can be branch targets in limited contexts are as follows:

- An END SELECT statement, provided the branch is taken from within the CASE construct.
- A DO termination, provided the branch is taken from within the DO construct.

- An `END IF` statement, provided that the branch is taken from within the `IF` construct.

The standard does not permit a branch to a statement within a block from outside the block. The compiler, however, permits these branches; such branches are diagnosed as being unsafe.

ANSI/ISO: The Fortran standard does not permit branches into executable blocks.

8.6.1 Use of Labels in Branching

A statement label is a means of identifying the branch target statement. Any statement in a Fortran program can have a label. However, if a branch statement refers to a statement label, some statement in the program unit must have that label, and the statement label must be on an allowed branch target statement.

As described in Section 3.2.5, page 52, a label is a string of from one to five decimal digits; leading zeros are not significant. Note that labels can be used in both free and fixed source forms.

8.6.2 `GO TO` Statement

The `GO TO` statement is an unconditional branching statement that alters the execution sequence.

8.6.2.1 Form of the `GO TO` Statement

The `GO TO` statement is defined as follows:

goto_stmt	**is** `GO TO` *label*

The label must be a branch target statement in the same scoping unit as the `GO TO` statement (that is, in the same program unit, excluding labels on statements in internal procedures, derived-type definitions, and interface blocks).

8.6.2.2 Execution of the GO TO Statement

When the GO TO statement is executed, the next statement that is executed is the branch target statement identified with the label specified. Execution proceeds from that point.

```
GO TO 200   ! This is an unconditional branch and
            ! always goes to 200.
            !
X = 1.0     ! Because this statement follows a GO
            ! TO statement and is unlabeled, it is
            ! not reachable.
GO TO 10
GO TO 010   ! 10 and 010 are the same label.
```

8.6.3 Computed GO TO Statement

The computed GO TO statement transfers to one of a set of branch target statements based on the value of an integer expression, selecting the branch target from a list of labels. The CASE construct provides a similar functionality in a more structured form.

Note: The Fortran standard has declared the computed GO TO statement to be obsolescent.

The computed GO TO statement is defined as follows:

computed_goto_stmt	**is**	GO TO (*label_list*) [,] *scalar_int_expr*

If there are *n* labels in the list and the expression has one of the values from 1 to *n*, the value identifies a statement label in the list: the first, second, ... , or *n*th label. A branch to the statement with that label is executed.

If the value of the expression is less than 1 or greater than *n*, no branching occurs and execution continues with the next executable statement or construct following the computed GO TO statement.

Each label in the list must be the label of a branch target statement in the same scoping unit as the computed GO TO statement.

A label can appear more than once in the list of target labels.

```
GO TO ( 10, 20 ), SWITCH
GO TO ( 100, 200, 3, 33 ),  2*I-J
```

In the following example, if SWITCH has the value 1 or 3, the assignment statement labeled 10 is executed; if it has the value 2, the assignment statement labeled 11 is executed. If it has a value less than 1 or greater than 3, the assignment statement Y = Z is executed, because it is the next statement after the computed GO TO statement, and the statement with label 10 is executed next.

```
   SWITCH = . . .
   GO TO (10, 11, 10) SWITCH
   Y = Z
10 X = Y + 2.
   . . .
11 X = Y
```

8.6.4 CONTINUE Statement

The CONTINUE statement is defined as follows:

continue_stmt	**is** CONTINUE

Typically, the statement has a label and is used for DO termination; however, it can serve as some other place holder in the program or as a branch target statement. It can appear without a label. The statement by itself does nothing and has no effect on the execution sequence or on program results. The following are examples of CONTINUE statements:

```
100 CONTINUE
CONTINUE
```

8.6.5 STOP Statement

The STOP statement terminates the program whenever and wherever it is executed. The STOP statement is defined as follows:

	stop_stmt	**is**	`STOP` [*stop_code*]
	stop_code	**is**	*scalar_char_constant*
EXT		**or**	*digit* `...`

The character constant or list of digits identifying the `STOP` statement is optional and is called a *stop code.*

When the *stop_code* is a string of digits, leading zeros are not significant; `10` and `010` are the same *stop_code*. You can specify from 1 to 80 *digit*s.

The stop code is accessible following program termination. The compiler sends it to the standard error file (`stderr`). The following are examples of `STOP` statements:

```
STOP
STOP 'Error #823'
STOP 20
```

ANSI/ISO: The Fortran standard specifies from 1 to 5 *digit*s in the *stop_code*.

8.7 Arithmetic `IF` Statement (Obsolescent)

The arithmetic `IF` statement is a three-way branching statement based on an arithmetic expression.

The arithmetic `IF` statement is defined as follows:

R840	*arithmetic_if_stmt*	**is**	`IF` (*scalar_numeric_expr*) *label*`,` *label*`,` *label*

The same label can appear more than once in an arithmetic `IF` statement.

The numeric expression must not be of type complex.

Each statement label must be the label of a branch target statement in the same scoping unit as the arithmetic `IF` statement itself.

The execution begins with the evaluation of the expression. If the expression is negative, the branch is to the first label; if zero, to the second label; and if positive, to the third label.

The following example shows an arithmetic `IF` statement:

```
     READ *, DATA
     IF(DATA) 10, 20, 30
10   PRINT *, 'NEGATIVE VALUE'
     ...
20   PRINT *, 'ZERO VALUE'
     ...
30   PRINT *, 'POSTIVE VALUE'
     ...
```

Glossary

argument keyword

The name of a dummy (or formal) argument. This name is used in the subprogram definition; it also may be used when the subprogram is invoked to associate an actual argument with a dummy argument. Using argument keywords allows the actual arguments to appear in any order. The Fortran 90 standard specifies argument keywords for all intrinsic procedures. Argument keywords for user-supplied external procedures may be specified in a procedure interface block.

array

(1) A data structure that contains a series of related data items arranged in rows and columns for convenient access. (2) In Fortran 90, an object with the `DIMENSION` attribute. It is a set of scalar data, all of the same type and type parameters. The rank of an array is at least 1, and at most 7. Arrays may be used as expression operands, procedure arguments, and function results, and they may appear in input/output (I/O) lists.

association

An association permits an entity to be referenced by different names in a scoping unit or by the same or different names in different scoping units. Several kinds of association exist. The principal kinds of association are pointer association, argument association, host association, use association, and storage association.

automatic variable

A variable that is not a dummy argument but whose declaration depends on a nonconstant expression (array bounds and/or character length).

bottom loading

An optimization technique used on some scalar loops in which operands are prefetched during each loop iteration for use in the next iteration. The operand is available as soon as the first loop instruction executes. A prefetch is performed even during the final loop iteration, before the loop's final jump test has been performed.

cache

In a processing unit, a high-speed buffer storage that is continually updated to contain recently accessed contents of main storage. Its purpose is to reduce access time. In disk subsystems, a method the channel buffers use to buffer disk data during transfer between the devices and memory.

CIV

A constant increment variable is a variable that is incremented only by a loop invariant value (for example, in a loop with index J, the statement J = J + K, in which K can be equal to 0, J is a CIV).

constant

A data object whose value cannot be changed. A named entity with the `PARAMETER` attribute is called a named constant. A constant without a name is called a literal constant.

construct

A sequence of statements that starts with a `SELECT CASE`, `DO`, `IF`, or `WHERE` statement and ends with the corresponding terminal statement.

control construct

An action statement that can change the normal execution sequence (such as a `GO TO`, `STOP`, or `RETURN` statement) or a `CASE`, `DO`, or `IF` construct.

data entity

A data object, the result of the evaluation of an expression, or the result of the execution of a function reference (also called the function result). A data entity always has a type.

data object

A constant, a variable, or a part of a constant or variable.

declaration

A nonexecutable statement that specifies the attributes of a data object (for example, it may be used to specify the type of a variable or function result or the shape of an array).

definition

This term is used in two ways. (1) A data object is said to be defined when it has a valid or predictable value; otherwise, it is undefined. It may be given a valid value by execution of statements such as assignment or input. Under certain circumstances, it may subsequently become undefined. (2) Procedures and derived types are said to be defined when their descriptions have been supplied by the programmer and are available in a program unit.

derived type

A type that is not intrinsic (a user-defined type); it requires a type definition to name the type and specify its components. The components may be of intrinsic or user-defined types. An object of derived type is called a structure. For each derived type, a structure constructor is available to specify values. Operations on objects of derived type must be defined by a function with an interface and the generic specifier `OPERATOR`. Assignment for derived type objects is defined intrinsically, but it may be redefined by a subroutine with the `ASSIGNMENT` generic specifier. Data objects of derived type may be used as procedure arguments and function results, and they may appear in input/output (I/O) lists.

designator

Sometimes it is convenient to reference only part of an object, such as an element or section of an array, a substring of a character string, or a component of a structure. This requires the use of the name of the object followed by a selector that selects a part of the object. A name followed by a selector is called a **`designator`**.

executable construct

A statement (such as a `GO TO` statement) or a construct (such as a `DO` or `CASE` construct).

expression

A set of operands, which may be function invocations, and operators that produce a value.

extent

A structure that defines a starting block and number of blocks for an element of file data.

function

Usually a type of operating-system-related function written outside a program and called in to do a specific function. Smaller and more limited in capability than a utility. In a programming language, a function is usually defined as a closed subroutine that performs some defined task and returns with an answer, or identifiable return value.

The word "function" has a more specific meaning in Fortran than it has in C. In C, it is refers to any called code; in Fortran, it refers to a subprogram that returns a value.

generic function

In FORTRAN 77 and Fortran 90, a **`generic function`** is one whose output value data type is determined by the data type of its input arguments. In FORTRAN 77, the only generic functions allowed are those that the standard defines. In Fortran 90, programmers may construct their own generic function by creating "generic interface," which is like a regular procedure interface, except that it has a "generic specifier" (the name of the generic function) after the keyword `INTERFACE`.

generic specifier

An optional component of the `INTERFACE` statement. It can take the form of an identifier, an `OPERATOR (`defined_operator`)` clause, or an `ASSIGNMENT (=)` clause.

heap

A section of memory within the user job area that provides a capability for dynamic allocation. See the `HEAP` directive in SR-0066.

inlining

The process of replacing a user subroutine or function call with the definition itself. This saves subprogram call overhead and may allow better optimization of the inlined code. If all calls within a loop are inlined, the loop becomes a candidate for vectorization and/or tasking.

intrinsic

Anything that the language defines is intrinsic. There are intrinsic data types, procedures, and operators. You may use these freely in any scoping unit. Fortran programmers may define types, procedures, and operators; these entities are not intrinsic.

local

(1) A type of scope in which variables are accessible only to a particular part of a program (usually one module). (2) The system initiating the request for service. This term is relative to the perspective of the user.

multitasking

(1) The parallel execution of two or more parts of a program on different CPUs; these parts share an area of memory. (2) A method in multiuser systems that incorporates multiple interconnected CPUs; these CPUs run their programs simultaneously (in parallel) and shares resources such as memory, storage devices, and printers. This term can often be used interchangeably with **`parallel processing`**.

name

A term that identifies many different entities of a program such as a program unit, a variable, a common block, a construct, a formal argument of a subprogram (dummy argument), or a user-defined type (derived type). A name may be associated with a specific constant (named constant).

operator

(1) A symbolic expression that indicates the action to be performed in an expression; operator types include arithmetic, relational, and logical. (2) In Fortran 90, an operator indicates a computation that involves one or two operands. Fortran 90 defines several intrinsic operators (for example, +, -, *, /, ** are numeric operators, and .NOT., .AND., and .OR. are logical operators). Users also may define operators for use with operands of intrinsic or derived types.

overindexing

The nonstandard practice of referencing an array with a subscript not contained between the declared lower and upper bounds of the corresponding dimension for that array. This practice sometimes, but not necessarily, leads to referencing a storage location outside of the entire array.

parallel processing

Processing in which multiple processors work on a single application simultaneously.

pointer

A data item that consists of the address of a desired item.

procedure

In Fortran 90, procedure is defined by a sequence of statements that expresses a computation that may be invoked as a subroutine or function during program execution. It may be an intrinsic procedure, an external procedure, an internal procedure, a module procedure, a dummy procedure, or a statement function. If a subprogram contains an `ENTRY` statement, it defines more than one procedure.

procedure interface

In Fortran 90, a sequence of statements that specifies the name and characteristics of one or more procedures, the name and attributes of each dummy argument, and the generic specifier by which it may be referenced if any. See **`generic specifier`**.

In FORTRAN 77 and Fortran 90, a **`generic function`** is one whose output value data type is determined by the data type of its input arguments. In FORTRAN 77, the only generic functions allowed are those that the standard defines. In Fortran 90, programmers may construct their own generic function by creating "generic interface," which is like a regular procedure interface, except that it has a "generic specifier" (the name of the generic function) after the keyword `INTERFACE`.

reduction loop

A loop that contains at least one statement that reduces an array to a scalar value by doing a cumulative operation on many of the array elements. This involves including the result of the previous iteration in the expression of the current iteration.

reference

A data object reference is the appearance of a name, designator, or associated pointer in an executable statement that requires the value of the object. A procedure reference is the appearance of the procedure name, operator symbol, or assignment symbol in an executable program that requires execution of the procedure. A module reference is the appearance of the module name in a `USE` statement.

scalar

(1) In Fortran 90, a single object of any intrinsic or derived type. A structure is scalar even if it has a component that is an array. The rank of a scalar is 0. (2) A

nonvectorized, single numerical value that represents one aspect of a physical quantity and may be represented on a scale as a point. This term often refers to a floating-point or integer computation that is not vectorized; more generally, it also refers to logical and conditional (jump) computation.

scope

The region of a program in which a variable is defined and can be referenced.

scoping unit

Part of a program in which a name has a fixed meaning. A program unit or subprogram generally defines a scoping unit. Type definitions and procedure interface bodies also constitute scoping units. Scoping units do not overlap, although one scoping unit may contain another in the sense that it surrounds it. If a scoping unit contains another scoping unit, the outer scoping unit is referred to as the host scoping unit of the inner scoping unit.

search loop

A loop that can be exited by means of an IF statement.

sequence

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.

shared

Accessible by multiple parts of a program. Shared is a type of scope.

shell variable

A name representing a string value. Variables that are usually set only on a command line are called **`parameters`** (positional parameters and keyword parameters). Other variables are simply names to which a user (user-defined variables) or the shell itself may assign string values. The shell has predefined shell variables (for example, `HOME`). Variables are referenced by prefixing the variable name by a `$` (for example, `$HOME`).

software pipelining

Software pipelining is a compiler code generation technique in which operations from various loop iterations are overlapped in order to exploit instruction-level parallelism, increase instruction issue rate, and better hide memory and instruction latency. As an optimization technique, software pipelining is similar to bottom loading, but it includes additional, and more efficient, scheduling optimizations.

SGI compilers perform safe bottom loading by default. Under these conditions, code generated for a loop contains operations and stores associated with the present loop iteration and contains loads associated with the next loop iteration. Loads for the first iteration are generated in the loop preamble.

When software pipelining is performed, code generated for the loop contains loads, operations, and stores associated with various iterations of the loop. Loads and operations for first iterations are generated in the preamble to the loop. Operations and stores for last iterations of loop are generated in the postamble to the loop.

statement keyword

A keyword that is part of the syntax of a statement. Each statement, other than an assignment statement and a statement function definition, begins with a statement keyword. Examples of these keywords are `IF`, `READ`, and `INTEGER`. Statement keywords are not reserved words; you may use them as names to identify program elements.

stripmining

A single-processor optimization technique in which arrays, and the program loops that reference them, are split into optimally-sized blocks, termed strips. The original loop is transformed into two nested loops. The inner loop references all data elements within a single strip, and the outer loop selects the strip to be addressed in the inner loop. This technique is often performed by the compiler to maximize the usage of cache memory or as part of vector code generation.

structure

A language construct that declares a collection of one or more variables grouped together under one name for convenient handling. In C and C++, a structure is defined with the `struct` keyword. In Fortran 90, a derived type is defined first and various structures of that type are subsequently declared.

subobject

Parts of a data object may be referenced and defined separately from other parts of the object. Portions of arrays are array elements and array sections. Portions of character strings are substrings. Portions of structures are structure components. Subobjects are referenced by designators and are considered to be data objects themselves.

subroutine

A series of instructions that accomplishes a specific task for many other routines. (A subsection of a user-written program of varying size and, therefore, function. It is written within the program. It is not a subsection of a routine.)

TKR

An acronym that represents attributes for argument association. It represents the data type, kind type parameter, and rank of the argument.

type parameter

Two type parameters exist for intrinsic types: kind and length. The kind type parameter `KIND` indicates the decimal range for the integer type, the decimal precision and exponent range for the real and complex types, and the machine representation method for the character and logical types. The length type parameter `LEN` indicates the length of a character string.

variable

(1) A name that represents a string value. Variables that usually are set only on a command line are called parameters. Other variables are simply names to which the user or the shell may assign string values. (2) In Fortran 90, data object whose value can be defined and redefined. A variable may be a scalar or an array. (3) In the shell command language, a named parameter. See also **`shell variable`**.

Index

B

C

D

N

O

P

R

S

Z