Fortran 90/95 Explained

Second Edition

MICHAEL METCALF

Formerly of the
Information Technology Division
CERN, Geneva, Switzerland

JOHN REID

JKR Associates, Oxfordshire

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS
Great Clarendon Street, Oxford OX2 6DP
Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide in
Oxford New York
Athens Auckland Bangkok Bogotá Buenos Aires Calcutta
CapeTown Chennai DaresSalaam Delhi Florence HongKong Istanbul
Karachi KualaLumpur Madrid Melbourne MexicoCity Mumbai
Nairobi Paris SãoPaulo Singapore Taipei Tokyo Toronto Warsaw
with associated companies in Berlin Ibadan
Oxford is a registered trade mark of Oxford University Press
in the UK and in certain other countries
Published in the United States
by Oxford University Press Inc., New York

First published 1996
Second edition 1999

Library of Congress Cataloging in Publication Data
(Data available)
ISBN 0 19 850558 2
Typeset by the authors
Printed in Great Britain
on acid-free paper by
Bookcraft (Bath) Ltd
Midsomer Norton, Avon

Preface

Fortran has always been the principal language used in the fields of scientific, numerical, and engineering programming, and a series of revisions to the standard defining successive versions of the language has progressively enhanced its power and kept it competitive with several generations of rivals.

Beginning in 1978, the technical committee responsible for the development of Fortran standards, X3J3 (now called J3), laboured to produce a new, much-needed modern version of the language, Fortran 90. Its purpose is to "promote portability, reliability, maintainability, and efficient execution... on a variety of computing systems". The standard was published in 1991, and work began in 1993 on a minor revision, known informally as Fortran 95. Now this revised standard is in use, it seems appropriate to prepare a definitive informal description of the language it defines. This continues the series of editions of this book – the two editions of *Fortran 8x Explained* that described the two drafts of the standard (1987 and 1989), and *Fortran 90 Explained* that described the Fortran 90 standard (1990).

The whole of Fortran 77 is contained in Fortran 90, but certain of its features are labelled 'obsolescent' in the standard, and their use is not recommended. The obsolescent features of Fortran 90 have replacements in Fortran 77 and some have been removed from the Fortran 95 standard. Other Fortran 77 features, with replacements in Fortran 90, are labelled obsolescent in Fortran 95. We have relegated the description of all these Fortran 77 features to Appendix C. They are falling into disuse and an understanding of them is required only when dealing with old programs.

In this book, an initial chapter sets out the background to the work on the new standards, and the ten following chapters describe Fortran 90/95 less the obsolescent features in a manner suitable both for grasping the implications of the new features, and for writing programs. Features that are available in Fortran 95 only are labelled as such. Where the word 'Fortran' is used, it means 'common to both Fortran 90 and 95'. Some knowledge of programming concepts, although not necessarily of Fortran 77, is assumed. In order to reduce the number of forward references and also to enable, as quickly as possible, useful programs to be written based on material already absorbed, the order of presentation does not always follow that of the standard. In particular, we have chosen to defer to the final chapter the description of features that are redundant in Fortran 90 and whose

use we deprecate. It would impair the flow of the exposition if we were to describe them in the main body of the text. They may be encountered in old programs, but are not needed in new ones.

This edition differs from the first edition in that descriptions of Fortran 95 features are integrated into the body instead of being confined to a separate chapter. This book is thus suitable for the reader who wishes to learn only Fortran 95 as well as for one who wishes to learn just Fortran 90 or even both Fortran 90 and Fortran 95.

We have chosen to use lower-case letters instead of upper-case letters for all the Fortran keywords and names, since this is the preferred style of many Fortran programmers and there are very few processors still in use that support only upper case. Also, we have switched to a different typesetting system (now LaTeX). These two changes give the book a very different (and in our opinion, improved) look.

This edition also differs from the first edition in that we have added two new chapters, 12 and 13, on official extensions of Fortran 95. Each is specified by an ISO Technical Report and WG5 has promised that the features of both will be included in the next revision of the language, apart from correcting defects found in the field. We expect Fortran 95 compilers increasingly to offer these features as extensions.

In order to make the book a complete reference work, it concludes with six appendices. They contain, successively, a list of the intrinsic procedures, a summary of Fortran statements, a description of the obsolescent and deleted features, an extended example illustrating the use of pointers and recursion, a glossary of Fortran terms, and solutions to most of the exercises.

It must be remembered that although the obsolescent features appear in an appendix, they were nevertheless an integral part of the Fortran 90 language, and some remain part of Fortran 95. However, the appendix includes advice on how to avoid their use, thereby enhancing the upwards compatibility of programs with respect to possible future standards. The same is true for the features whose use we deprecate, and which are described in Chapter 11.

It is our hope that this book, by providing a complete description of Fortran 90 and Fortran 95, will continue the helpful role that earlier editions played for the Fortran 90 standard, and will serve as a long-term reference work for Fortran 95 into the next decade.

Acknowledgements

The development of the Fortran 90 standard was a long procedure involving several hundred people in many countries. The main burden fell on the principal members of X3J3 (now called J3), and especially on its then chairman, Jeanne Adams, and on the then Convenor of WG5, Jeanne Martin. The work continued for Fortran 95 with Jerry Wagener as chairman of X3J3. We extend our thanks to them and all our colleagues on X3J3 and elsewhere for their devotion to this important but thankless task, and for creating such a friendly working atmosphere. We will long remember the week-long meetings held over such an extended period, as well as the personal contacts we made and valued. We have taken great pains to ensure that this book is a true and accurate representation of the final documents produced by the committee, and clearly any omissions or other errors or misrepresentations are entirely our responsibility.

We gratefully acknowledge the actual and former management of CERN and Harwell, and especially P. Zanella and D.O. Williams of CERN and the late A.E. Taylor of Harwell, for encouraging us to undertake this work, and for providing the necessary resources for its realization. JKR is also indebted to the Rutherford Appleton Laboratory for its subsequent support.

Conventions used in this book

Fortran displayed text is set in typewriter font:

```
integer :: i, j
```

and a line consisting of a colon indicates omitted lines:

```
subroutine sort
   :
end subroutine sort
```

Informal BNF terms are in italics:

`if` (*scalar-logical-expr*) *action-stmt*

Square brackets indicate optional items:

`end if` [*name*]

and an ellipsis represents an arbitrary number of repeated items:

[`case` *selector* [*name*]
block] ...

The italic letter *b* signifies a blank character.

Contents

1 Whither Fortran? 1
- 1.1 Fortran history . . . 2
- 1.2 The drive for the Fortran 90 standard . . . 3
- 1.3 Language evolution . . . 4
- 1.4 Fortran 95 . . . 5
- 1.5 Beyond Fortran 95 . . . 6
- 1.6 Conformance . . . 7

2 Language elements 9
- 2.1 Introduction . . . 9
- 2.2 Fortran character set . . . 9
- 2.3 Tokens . . . 10
- 2.4 Source form . . . 11
- 2.5 Concept of type . . . 13
- 2.6 Literal constants of intrinsic type . . . 13
  - 2.6.1 Integer literal constants . . . 14
  - 2.6.2 Real literal constants . . . 15
  - 2.6.3 Complex literal constants . . . 17
  - 2.6.4 Character literal constants . . . 17
  - 2.6.5 Logical literal constants . . . 20
- 2.7 Names . . . 20
- 2.8 Scalar variables of intrinsic type . . . 21
- 2.9 Derived data types . . . 21
- 2.10 Arrays of intrinsic type . . . 23
- 2.11 Character substrings . . . 26
- 2.12 Objects and subobjects . . . 27
- 2.13 Pointers . . . 28
- 2.14 Summary . . . 29
- 2.15 Exercises . . . 30

3 Expressions and assignments 33
- 3.1 Introduction . . . 33
- 3.2 Scalar numeric expressions . . . 34
- 3.3 Defined and undefined variables . . . 37

3.4 Scalar numeric assignment 38
3.5 Scalar relational operators 38
3.6 Scalar logical expressions and assignments 39
3.7 Scalar character expressions and assignments 41
3.8 Structure constructors and scalar defined operators 42
3.9 Scalar defined assignments 45
3.10 Array expressions 47
3.11 Array assignment 48
3.12 Pointers in expressions and assignments 49
3.13 Summary 51
3.14 Exercises 52

4 Control constructs **55**
4.1 Introduction 55
4.2 The go to statement 55
4.3 The if statement and construct 56
4.3.1 The if statement 56
4.3.2 The if construct 57
4.4 The case construct 59
4.5 The do construct 61
4.6 Summary 66
4.7 Exercises 68

5 Program units and procedures **69**
5.1 Introduction 69
5.2 Main program 70
5.3 The stop statement 71
5.4 External subprograms 72
5.5 Modules 73
5.6 Internal subprograms 75
5.7 Arguments of procedures 76
5.7.1 Pointer arguments 78
5.7.2 Restrictions on actual arguments 78
5.7.3 Arguments with the target attribute 79
5.8 The return statement 80
5.9 Argument intent 80
5.10 Functions 81
5.10.1 Prohibited side-effects 83
5.11 Explicit and implicit interfaces 83
5.12 Procedures as arguments 85
5.13 Keyword and optional arguments 87
5.14 Scope of labels 88
5.15 Scope of names 89
5.16 Direct recursion 91
5.17 Indirect recursion 92

5.18 Overloading and generic interfaces 93
5.19 Assumed character length 98
5.20 The subroutine and function statements 99
5.21 Summary 99
5.22 Exercises 100

6 Array features **103**
6.1 Introduction 103
6.2 Zero-sized arrays 103
6.3 Assumed-shape arrays 104
6.4 Automatic objects 105
6.5 Heap storage 106
6.5.1 Allocatable arrays 106
6.5.2 The allocate statement 107
6.5.3 The deallocate statement 109
6.5.4 The nullify statement 110
6.6 Elemental operations and assignments 111
6.7 Array-valued functions 111
6.8 The where statement and construct 112
6.8.1 Some where construct extensions (Fortran 95 only) . . . 114
6.9 The forall statement and construct (Fortran 95 only) 116
6.10 Pure procedures (Fortran 95 only) 118
6.11 Elemental procedures (Fortran 95 only) 120
6.12 Array elements 122
6.13 Array subobjects 123
6.14 Arrays of pointers 126
6.15 Pointers as aliases 127
6.16 Array constructors 128
6.17 Mask arrays 129
6.18 Summary 130
6.19 Exercises 133

7 Specification statements **135**
7.1 Introduction 135
7.2 Implicit typing 136
7.3 Declaring entities of differing shapes 138
7.4 Named constants and constant expressions 139
7.5 Initial values for variables 141
7.5.1 Initialization in type declaration statements 141
7.5.2 The data statement 142
7.5.3 Pointer initialization and the function null (Fortran 95 only) 144
7.5.4 Default initialization of components (Fortran 95 only) . 145
7.6 The public and private attributes 146
7.7 The pointer, target, and allocatable statements 148
7.8 The intent and optional statements 148

7.9 The save attribute 149
7.10 The use statement 150
7.11 Derived-type definitions 153
7.12 The type declaration statement 155
7.13 Type and type parameter specification 156
7.14 Specification expressions 157
7.14.1 Specification expression restrictions (Fortran 90 only) . 158
7.14.2 Specification functions (Fortran 95 only) 159
7.15 The namelist statement 160
7.16 Summary 162
7.17 Exercises 163

8 Intrinsic procedures **165**
8.1 Introduction 165
8.1.1 Keyword calls 165
8.1.2 Categories of intrinsic procedures 166
8.1.3 The intrinsic statement 166
8.1.4 Argument intents 167
8.2 Inquiry functions for any type 167
8.3 Elemental numeric functions 167
8.3.1 Elemental functions that may convert 168
8.3.2 Elemental functions that do not convert 169
8.4 Elemental mathematical functions 169
8.5 Elemental character and logical functions 170
8.5.1 Character-integer conversions 170
8.5.2 Lexical comparison functions 171
8.5.3 String-handling elemental functions 171
8.5.4 Logical conversion 172
8.6 Non-elemental string-handling functions 172
8.6.1 String-handling inquiry function 172
8.6.2 String-handling transformational functions 172
8.7 Numeric inquiry and manipulation functions 173
8.7.1 Models for integer and real data 173
8.7.2 Numeric inquiry functions 174
8.7.3 Elemental functions to manipulate reals 175
8.7.4 Transformational functions for kind values 175
8.8 Bit manipulation procedures 176
8.8.1 Inquiry function 176
8.8.2 Elemental functions 176
8.8.3 Elemental subroutine 178
8.9 Transfer function 178
8.10 Vector and matrix multiplication functions 179
8.11 Transformational functions that reduce arrays 179
8.11.1 Single argument case 179
8.11.2 Optional argument dim 180

8.11.3 Optional argument mask 180
8.12 Array inquiry functions 181
8.12.1 Allocation status 181
8.12.2 Bounds, shape, and size 181
8.13 Array construction and manipulation functions 181
8.13.1 The merge elemental function 181
8.13.2 Packing and unpacking arrays 182
8.13.3 Reshaping an array 182
8.13.4 Transformational function for replication 183
8.13.5 Array shifting functions 183
8.13.6 Matrix transpose 183
8.14 Transformational functions for geometric location 184
8.15 Transformational function for pointer disassociation (Fortran 95) 184
8.16 Non-elemental intrinsic subroutines 184
8.16.1 Real-time clock 185
8.16.2 CPU time (Fortran 95 only) 185
8.16.3 Random numbers 186
8.17 Summary 187
8.18 Exercises 187

9 Data transfer **189**
9.1 Introduction 189
9.2 Number conversion 189
9.3 I/O lists 190
9.4 Format definition 192
9.5 Unit numbers 194
9.6 Internal files 196
9.7 Formatted input 197
9.8 Formatted output 198
9.9 List-directed I/O 199
9.10 Namelist I/O 201
9.10.1 Comments in namelist input (Fortran 95 only) 203
9.11 Carriage control 203
9.12 Non-advancing I/O 204
9.13 Edit descriptors 205
9.13.1 Repeat counts 206
9.13.2 Data edit descriptors 207
9.13.3 Minimal field width editing (Fortran 95 only) 210
9.13.4 Character string edit descriptor 210
9.13.5 Control edit descriptors 210
9.14 Unformatted I/O 213
9.15 Direct-access files 214
9.16 Execution of a data transfer statement 216
9.17 Summary 217
9.18 Exercises 217

10 Operations on external files **219**
10.1 Introduction 219
10.2 File positioning statements 220
10.2.1 The backspace statement 220
10.2.2 The rewind statement 221
10.2.3 The endfile statement 221
10.2.4 Data transfer statements 222
10.3 The open statement 222
10.4 The close statement 225
10.5 The inquire statement 225
10.6 Summary 229
10.7 Exercises 229

11 Other features **231**
11.1 Introduction 231
11.2 Storage association 231
11.2.1 Storage units 231
11.2.2 The equivalence statement 233
11.2.3 The common block 234
11.2.4 The block data program unit 237
11.2.5 Shape and character length disagreement 238
11.2.6 The entry statement 239
11.3 New redundant features 241
11.3.1 The include line 241
11.3.2 The do while form of loop control 242
11.4 Old redundant features 242
11.4.1 Double precision real 242
11.4.2 The dimension and parameter statements 243
11.4.3 Specific names of intrinsic procedures 244

12 Floating-point exception handling **249**
12.1 Introduction 249
12.1.1 Abandoned alternative 250
12.2 Intrinsic modules 251
12.3 The IEEE standard 251
12.4 Access to the features 253
12.5 The Fortran flags 255
12.6 Halting 256
12.7 The rounding modes 256
12.8 The module ieee_exceptions 257
12.8.1 Derived types 257
12.8.2 Inquiry functions for IEEE exceptions 258
12.8.3 Elemental subroutines 258
12.8.4 Non-elemental subroutines 259
12.9 The module ieee_arithmetic 260

12.9.1 Derived types 260
12.9.2 Inquiry functions for IEEE arithmetic 261
12.9.3 Elemental functions 262
12.9.4 Non-elemental subroutines 264
12.9.5 Transformational function for kind value 264
12.10 Examples 265
12.10.1 Dot product 265
12.10.2 Calling alternative procedures 266
12.10.3 Calling alternative in-line code 266
12.10.4 Reliable hypotenuse function 267

13 Allocatable array extensions **269**
13.1 Introduction 269
13.2 Allocatable dummy arguments 270
13.3 Allocatable functions 270
13.4 Allocatable components 272

A Intrinsic procedures **277**

B Fortran 90/95 statements **283**

C Obsolescent features **287**
C.1 Obsolescent in Fortran 95 only 287
C.1.1 Fixed source form 287
C.1.2 Computed go to 288
C.1.3 Character length specification character* 289
C.1.4 Data statements among executables 289
C.1.5 Statement functions 289
C.1.6 Assumed character length of function results 290
C.2 Obsolescent in Fortran 90 and 95 291
C.2.1 Arithmetic if statement 291
C.2.2 Shared do loop termination 292
C.2.3 Alternate return 292
C.3 Obsolescent in Fortran 90, deleted in Fortran 95 294
C.3.1 Non-integer do indices 294
C.3.2 Assigned go to and assigned formats 294
C.3.3 Branching to an end if statement 296
C.3.4 The pause statement 296
C.3.5 H edit descriptor 296

D Pointer example **297**

E Fortran terms **307**

F Solutions to exercises **319**

Index **331**

1. Whither Fortran?

This book is concerned with the Fortran programming language (Fortran 90 and Fortran 95), setting out a reasonably concise description of the whole language. The form chosen for its presentation is that of a textbook intended for use in teaching or learning the language. Its description occupies Chapters 2 to 11, which are written in such a way that simple programs can already be coded after the first three of these chapters (on language elements, expressions and assignments, and control) have been read. Successively more complex programs can be written as the information in each subsequent chapter is absorbed. Chapter 5 describes the important concept of the module and the many aspects of procedures, Chapter 6 completes the description of the powerful array features, Chapter 7 considers the details of specifying data objects and derived types, and Chapter 8 details the intrinsic procedures. Chapters 9 and 10 cover the whole of the input/output features in a manner such that the reader can also approach this more difficult area feature by feature, but always with a useful subset already covered. Finally, Chapter 11 describes those features that are redundant in the language, and whose use we choose to deprecate. Here we emphasize that this deprecation represents our own opinion, and is completely unofficial. In a concluding section of each of Chapters 2 to 10, we summarize the differences from Fortran 77. Chapters 12 and 13 describe official extensions to Fortran 95 that we expect increasingly to be present in Fortran 95 compilers.

Fortran 95 is a minor revision of Fortran 90, so most of this book applies to both. Features of Fortran 95 that are not part of Fortran 90 are usually described in separate subsections, but sometimes we use separate paragraphs. Every example that is not applicable to Fortran 90 is labelled as such. Features of Fortran 90 that are not part of Fortran 95 are obsolescent in Fortan 90 and are described only in Appendix C.

This introductory chapter has the task of setting the scene for those that follow. The first section presents the Fortran language and its considerable evolution since it was first introduced over thirty years ago. The second continues with a justification for preparing the Fortran 90 standard, summarizes the important new features, and outlines how standards are developed; the third looks at the mechanism that has been proposed to permit the language to evolve. The fourth section considers the development of Fortran 95 and the fifth some related issues.

The sixth concludes by considering the requirements on programs and processors for conformance with the standard.

1.1 Fortran history

Programming in the early days of computing was tedious in the extreme. Programmers required a detailed knowledge of the instructions, registers, and other aspects of the central processing unit (CPU) of the computer for which they were writing code. The *source code* itself was written in a numerical notation, so-called *octal code*. In the course of time mnemonic codes were introduced, a form of coding known as *machine* or *assembly code*. These codes were translated into the instruction words by programs known as *assemblers*. In the 1950s it became increasingly apparent that this form of programming was highly inconvenient, although it did enable the CPU to be used in a very efficient way.

These difficulties spurred a team led by John Backus of IBM to develop one of the earliest high-level languages, Fortran. Their aim was to produce a language which would be simple to understand but almost as efficient in execution as assembly language. In this they succeeded beyond their wildest dreams. The language was indeed simple to learn, as it was possible to write mathematical formulae almost as they are usually written in mathematical texts. (In fact, the name Fortran is a contraction of Formula Translation.) This enabled working programs to be written faster than before, for only a small loss in efficiency, as a great deal of care was devoted to the construction of the compiler.

But Fortran was revolutionary as well as innovatory. Programmers were relieved of the tedious burden of using assembler language, and were able to concentrate more on the problem in hand. Perhaps more important, however, was the fact that computers became accessible to any scientist or engineer willing to devote a little effort to acquiring a working knowledge of Fortran; no longer was it necessary to be an expert on computers to be able to write application programs.

Fortran spread rapidly as it fulfilled a real need. Inevitably dialects of the language developed, which led to problems in exchanging programs between computers, and so, in 1966 the then American Standards Association (later the American National Standards Institute, ANSI) brought out the first ever standard for a programming language, now known as Fortran 66.

Fortran brought with it several other advances, apart from its ease of learning combined with a stress on efficient execution of code. It was, for instance, a language which remained close to, and exploited, the available hardware rather than being an abstract concept. It also brought with it the possibility for programmers to control storage allocation in a simple way, a feature which was very necessary in those early days of small memories, even if it is now regarded as being potentially dangerous.

The proliferation of dialects remained a problem after the publication of the 1966 standard. There was a widespread implementation in compilers of features

which were essential for large-scale programs, but which were ignored by the standard. Different compilers implemented such facilities in different ways.

These difficulties were partially resolved by the publication of a new standard, in 1978, known as Fortran 77. It included several new features that were based on vendor extensions or pre-processors and it was, therefore, not simply a common subset of existing dialects. By the mid-1980s, the changeover to Fortran 77 was in full swing. It was a relatively simple matter to write new code under the new standard, and converting old standard-conforming code was usually easy as there is a large measure of compatibility between the two standards.

1.2 The drive for the Fortran 90 standard

After thirty years' existence, Fortran was far from being the only programming language available on most computers. In the course of time new languages had been developed, and where they were demonstrably more suitable for a particular type of application they had been adopted in preference to Fortran for that purpose. Fortran's superiority had always been in the area of numerical, scientific, engineering, and technical applications and, in order that it be brought properly up-to-date, the ANSI-accredited technical committee X3J3 (now known as J3) working as a development body for the ISO committee ISO/IEC JTC1/SC22/WG5 (which we abbreviate to WG5), once again prepared a new standard, formerly known as Fortran 8x and now as Fortran 90.

X3J3 itself is a body composed of representatives of computer hardware and software vendors, users, and academia. It is accredited to ANSI, the body that publishes final American standards, but reports directly to its parent committee, X3 (computer systems), which is responsible for actually adopting, or rejecting, the proposed draft standards presented to it. In these decisions, it tries to ensure that the proposals really do represent a consensus of those concerned. X3J3 acts as the development body for the corresponding international group, WG5, consisting of international experts responsible for recommending that a draft standard become an international standard. X3J3 maintains other close contacts with the international community by welcoming foreign members, including both the present authors.

What were the justifications for continuing to revise the definition of the Fortran language? As well as standardizing vendor extensions, there was a need to modernize it in response to the developments in language design which had been exploited in other languages, such as APL, Algol 68, Pascal, Ada, C and C++. Here, X3J3 could draw on the obvious benefits of concepts like data hiding. In the same vein was the need to begin to provide an alternative to dangerous storage association, to abolish the rigidity of the outmoded source form, and to improve further on the regularity of the language, as well as to increase the safety of programming in the language and to tighten the conformance requirements. To preserve the vast investment in Fortran 77 codes, the whole of Fortran 77 was retained as a subset. However, unlike the previous standard, which resulted almost

entirely from an effort to standardize *existing practices*, the Fortran 90 standard is much more a *development* of the language, introducing features which are new to Fortran, but are based on experience in other languages.

The main features of Fortran 90 are, first and foremost, the array language and abstract data types. The former is built on whole array operations and assignments, array sections, intrinsic procedures for arrays, and dynamic storage. It was designed with optimization in mind. The latter is built on modules and module procedures, derived data types, operator overloading and generic interfaces, together with pointers. Also important are the new facilities for numerical computation including a set of numeric inquiry functions, the parametrization of the intrinsic types, new control constructs – `select case` and new forms of `do`, internal and recursive procedures and optional and keyword arguments, improved I/O facilities, and many new intrinsic procedures. Last but not least are the new free source form, an improved style of attribute-oriented specifications, the `implicit none` statement, and a mechanism for identifying redundant features for subsequent removal from the language. The requirement on compilers to be able to identify, for example, syntax vextensions, and to report why a program has been rejected, are also significant. The resulting language is not only a far more powerful tool than its successor, but a safer and more reliable one too. Storage association, with its attendant dangers, is not abolished, but rendered unnecessary. Indeed, experience shows that compilers detect errors far more frequently than before, resulting in a faster development cycle. The array syntax and recursion also allow quite compact code to be written, a further aid to safe programming.

1.3 Language evolution

The procedures under which X3J3 works require that a period of notice be given before any existing feature is removed from the language. This means, in practice, a minimum of one revision cycle, which for Fortran means about five years. The need to remove features is evident: if the only action of the committee is to add new features, the language will become grotesquely large, with many overlapping and redundant items. The solution finally adopted by X3J3 was to publish as an appendix to a standard a set of two lists showing which items have been removed or are candidates for eventual removal.

One list contains the *deleted features*, those that have been removed. Since Fortran 90 contains the whole of Fortran 77, this list is empty for Fortran 90 but is not for Fortran 95 (see Appendix C).

The second list contains the *obsolescent features*, those considered to be outmoded and redundant, and which are candidates for deletion in the next revision. The Fortran 90 and Fortran 95 obsolescent features are described in Appendix C.

1.4 Fortran 95

Following the publication of the Fortran 90 standard in 1991, two further significant developments concerning the Fortran language occurred. The first was the continued operation of the two Fortran standards committees, X3J3 and WG5, and the second the founding of the High Performance Fortran Forum (HPFF).

Early on in their deliberations, the standards committees decided on a strategy whereby a minor revision of Fortran 90 would be prepared by the mid-1990s and a further revision by about the year 2000. The first revision, Fortran 95, is a subject of this book.

The HPFF was set up in an effort to define a set of extensions to Fortran, such that it would be possible to write portable code when using parallel computers for handling problems involving large sets of data that can be represented by regular grids. This version of Fortran was to be known as High Performance Fortran (HPF), and it was quickly decided, given the array features of Fortran 90, that it, and not Fortran 77, should be its base language. The final form of HPF[1] is of a superset of Fortran 90, the main extensions being in the form of directives that take the form of Fortran 90 comment lines, and are thus recognized as directives only by an HPF processor. However, it did become necessary also to add some additional syntax, as not all the desired features could be accommodated in the form of such directives.

The work of J3 (as X3J3 became known) and WG5 went on at the same time as that of HPFF, and the bodies liaised closely. It was evident that, in order to avoid the development of divergent dialects of Fortran, it would be desirable to include the new syntax defined by HPFF in Fortran 95 and, indeed, the HPF features are the most significant new features that Fortran 95 introduces. The other changes consist mainly of what are known as corrections, clarifications and interpretations. These came about as it was quickly discovered, as Fortran 90 compilers were written and used, that the text of the Fortran 90 standard contained a number of errors that required correction, some obscure wording that required further textual clarification, and ambiguous statements that required interpretation. (J3 and WG5 processed about 200 requests for interpretation.) All the resulting changes have been included in the Fortran 95 standard and, where appropriate, they have been incorporated at the relevant places in this book. Apart from the HPF syntax and the corrections, only a small number of other pressing but minor language changes were made and these too are described.

Fortran 95 is backwards compatible with Fortran 90, apart from a minor change in the definition of `sign` (Section 8.3.2) and the deletion of some Fortran 77 features declared obsolete in Fortran 90 (as described in Appendix C). However, there are two new intrinsic procedures, `null` and `cpu_time`, which might also be names of external procedures in an existing Fortran 90 program.

[1] *The High Performance Fortran Handbook*, C. Koebel et al., MIT Press, Cambridge, MA, 1994.

The details of Fortran 95 were finalized in November 1995, and the new ISO standard, replacing Fortran 90, was adopted in 1997, following successful ballots, as ISO/IEC 1539-1 : 1997.

1.5 Beyond Fortran 95

About the time of the publication of Fortran 95, another interesting development occurred. This was the specification of two (similar) subset versions of Fortran 90 that retain its modern features while casting aside its outmoded ones. The present authors were involved in the development of one of these, known as F, and that language's description can be found in *"The F programming language"* (OUP, 1996). These subsets can be regarded as important vehicles for the teaching of a safe and reliable style of modern programming.

At the same time, a formal standard, ISO/IEC 1539-2 : 1994, was developed for varying length strings. It defines the interface and semantics for a module that provides facilities for the manipulation of character strings of arbitrary and dynamically variable length. An annex contains a possible implementation in Fortran 90, which demonstrates its feasibility, but the intention was that vendors provide equivalent features that execute more efficiently. Unfortunately, none has done so. At the time of writing, this standard is being revised to take advantage of the Fortran 95 enhancements. In particular, this will allow a better implentation within the standard language.

Further, in 1995, WG5 decided that these three features:

i) handling floating point exceptions,

ii) permitting allocatable arrays as structure components, dummy arguments, and function results, and

iii) interoperability with C,

were so urgently needed in Fortran that it established development bodies to develop 'Technical Reports of Type 2'. The intent was that the material of these technical reports be integrated into the next revision of the Fortran standard, apart from any defects found in the field. It is essentially a beta-test facility for a language feature. In the event, the first two have been completed and are the subjects of Chapters 12 and 13. Difficulties were encountered with the third, so the report mechanism has therefore been abandoned for interoperability with C, but the intent of inclusion in the next standard remains.

Another auxiliary standard, ISO/IEC 1539-3 : 1998, has been developed to meet the need of programmers to maintain several versions of code to allow for different systems and different applications. Keeping several copies of the source code is error prone. It is far better to maintain a master code from which any of the versions may be selected. This standard is for a very simple form of conditional compilation, which selects some of the Fortran lines from the source and omits the rest or converts them to comments. The process is controlled by 'coco lines'

in the source that are also omitted or converted to comments. It is hoped that the facilities will be built into compilers, but they may also be implemented by a preprocessor.

The next full language revision is planned for 2001 and is already being referenced as Fortran 2000, and the main features have been chosen:

- handling floating point exceptions, as in TR15580,
- permitting allocatable arrays as structure components, dummy arguments, and function results, as in TR15581,
- interoperability with C,
- parameterized data types,
- object-orientation: constructors/destructors, inheritance, and polymorphism,
- derived type I/O,
- asynchronous I/O,
- procedure variables, and
- various minor enhancements.

All this activity provides a means of ensuring that Fortran remains a powerful and well-honed tool for numerical and scientific applications for the next decade and beyond.

1.6 Conformance

The standard is almost exclusively concerned with the rules for programs rather than processors. A processor is required to accept a standard-conforming program and to interpret it according to the standard, subject to limits that the processor may impose on the size and complexity of the program. The processor is allowed to accept further syntax and to interpret relationships that are not specified in the standard, provided they do not conflict with the standard. Of course, the programmer must avoid such syntax extensions if portability is desired.

The interpretation of some of the standard syntax is *processor dependent*, that is, may vary from processor to processor. For example, the set of characters allowed in character strings is processor dependent. Care must be taken whenever a processor-dependent feature is used in case it leads to the program not being portable to a desired processor.

A drawback of the Fortran 77 standard was that it made no statement about requiring processors to provide a means to detect any departure from the allowed syntax by a program, as long as that departure did not conflict with the syntax rules defined by the standard. The new standards are written in a different style to the old one. The syntax rules are expressed in a form of BNF with associated

constraints, and the semantics are described by the text. This semi-formal style is not used in this book, so an example is perhaps helpful:

R609 *substring* **is** *parent-string* (*substring-range*)
R610 *parent-string* **is** *scalar-variable-name*
or *array-element*
or *scalar-structure-component*
or *scalar-constant*
R611 *substring-range* **is** [*scalar-int-expr*] : [*scalar-int-expr*]

Constraint: *parent-string* must be of type character.
The first *scalar-int-expr* in *substring-range* is called the **starting point** and the second one is called the **ending point**. The length of a substring is the number of characters in the substring and is MAX($\ell - f + 1, 0$), where f and ℓ are the starting and ending points, respectively.

Here, the three production rules and the associated constraint for a character substring are defined, and the meaning of the length of such a substring explained.

The standard is written in such a way that a processor, at compile-time, may check that the program satisfies all the constraints. In particular, the processor must provide a capability to detect and report the use of any

- obsolescent feature,
- additional syntax,
- kind type parameter (Section 2.5) that it does not support,
- non-standard source form or character,
- name that is inconsistent with the scoping rules, or
- non-standard intrinsic procedure.

Furthermore, it must be able to report the reason for rejecting a program. These capabilities are of great value in producing correct and portable code. They were not required for Fortran 77 programs.

2. Language elements

2.1 Introduction

Written prose in a natural language, such as an English text, is composed firstly of basic elements – the letters of the alphabet. These are combined into larger entities, words, which convey the basic concepts of objects, actions, and qualifications. The words of the language can be further combined into larger units, phrases and sentences, according to certain rules. One set of rules defines the grammar. This tells us whether a certain combination of words is correct in that it conforms to the *syntax* of the language, that is those acknowledged forms which are regarded as correct renderings of the meanings we wish to express. Sentences can in turn be joined together into paragraphs, which conventionally contain the composite meaning of their constituent sentences, each paragraph expressing a larger unit of information. In a novel, sequences of paragraphs become chapters and the chapters together form a book, which usually is a self-contained work, largely independent of all other books.

2.2 Fortran character set

Analogies to these concepts are found in a programming language. In Fortran (Fortran 90 and Fortran 95), the basic elements, or character set, are the 26 letters of the English alphabet, the 10 Arabic numerals, 0 to 9, the underscore, _, and the so-called special characters listed in Table 2.1. The standard does not require the support of lower-case letters, but almost all computers nowadays support them. Within the Fortran syntax, the lower-case letters are equivalent to the corresponding upper-case letters; they are distinguished only when they form part of character sequences. In this book, syntactically significant characters will always be written in lower case. The letters, numerals, and underscore are known as *alphanumeric* characters.

Except for the currency symbol, whose graphic may vary (for example, to be £ in the United Kingdom), the graphics are fixed, though their styles are not fixed. The special characters $ and ? have no specific meaning within the Fortran language.

In the course of this and the following chapters, we shall see how further analogies with natural language may be drawn. The unit of Fortran information

Table 2.1. The special characters of the Fortran language

Character	Name	Character	Name
=	Equals sign	:	Colon
+	Plus sign		Blank
-	Minus sign	!	Exclamation mark
*	Asterisk	"	Quotation mark
/	Slash	%	Percent
(	Left parenthesis	&	Ampersand
)	Right parenthesis	;	Semicolon
,	Comma	<	Less than
.	Decimal point	>	Greater than
$	Currency symbol	?	Question mark
'	Apostrophe		

is the *lexical token*, which corresponds to a word or punctuation mark. Adjacent tokens are usually separated by spaces or the end of a line, but sensible exceptions are allowed just as for a punctuation mark in prose. Sequences of tokens form *statements*, corresponding to sentences. Statements, like sentences, may be joined to form larger units like paragraphs. In Fortran these are known as *program units*, and out of these may be built a *program*. A program forms a complete set of instructions to a computer to carry out a defined sequence of operations. The simplest program may consist of only a few statements, but programs of more than 100,000 statements are now quite common.

2.3 Tokens

Within the context of Fortran, alphanumeric characters (the letters, the underscore, and the numerals) may be combined into sequences that have one or more meanings. For instance, one of the meanings of the sequence 999 is a constant in the mathematical sense. Similarly, the sequence `date` might represent, as one possible interpretation, a variable quantity to which we assign the calendar date.

The special characters are used to separate such sequences and also have various meanings. We shall see how the asterisk is used to specify the operation of multiplication, as in `x*y`, and has also a number of other interpretations.

Basic significant sequences of alphanumeric characters or of special characters are referred to as *tokens;* they are labels, keywords, names, constants (other than complex literal constants), operators (listed in Table 3.4, Section 3.8), and *separators*, which are

```
/   (   )   (/   /)   ,   =   =>   :   ::   ;   %
```

For example, the expression `x*y` contains the three tokens `x`, `*`, and `y`.

Apart from within a character string or within a token, blanks may be used freely to improve the layout. Thus, whereas the variable `date` may not be written as `d a t e`, the sequence `x * y` is syntactically equivalent to `x*y`. In this context, multiple blanks are syntactically equivalent to a single blank.

A name, constant, or label must be separated from an adjacent keyword, name, constant or label by one or more blanks or by the end of a line. For instance, in

```
   real x
   read 10
30 do k=1,3
```

the blanks are required after `real`, `read`, `30`, and `do`. Likewise, adjacent keywords must normally be separated, but some pairs of keywords, such as `else if`, are not required to be separated. Similarly, some keywords may be split; for example `inout` may be written `in out`. We do not use these alternatives in the main text, but the exact rules are given in the statement summaries in Appendix B.

2.4 Source form

Fortran 90/95 brings with it a new source form, well adapted to use at a terminal.[1] The statements of which a source program is composed are written on *lines*. Each line may contain up to 132 characters,[2] and usually contains a single statement. Since leading spaces are not significant, it is possible to start all such statements in the first character position, or in any other position consistent with the user's chosen layout. A statement may thus be written as

```
x = (-y + root_of_discriminant)/(2.0*a)
```

In order to be able to mingle suitable comments with the code to which they refer, Fortran allows any line to carry a trailing comment field, following an exclamation mark (!). An example is

```
x = y/a - b   ! Solve the linear equation
```

Any comment always extends to the end of the source line and may include processor-dependent characters (it is not restricted to the Fortran character set, Section 2.2). Any line whose first non-blank character is an exclamation mark, or contains only blanks, or which is empty, is purely commentary, and is ignored by the compiler. Such comment lines may appear anywhere in a program unit, including ahead of the first statement (but not after the final program unit). A *character context* (those contexts defined in Sections 2.6.4, 9.13.4, and C.3.5) is allowed to contain !, so the ! does not initiate a comment in this case; in all other cases it does.

[1] Some hints on maintaining compatibility between the old and the new source forms are given at the end of Section C.1.1.

[2] Lines containing characters of non-default kind (Sections 2.6.4) are subject to a processor-dependent limit.

Since it is possible that a long statement might not be accommodated in the 132 positions allowed in a single line, up to 39 additional continuation lines are allowed. The so-called *continuation mark* is the ampersand (&) character, and this is appended to each line that is followed by a continuation line. Thus, the first statement of this section (considerably spaced out) could be written as

```
x =                                                    &
   (-y + root_of_discriminant)                         &
   /(2.0*a)
```

In this book, the ampersands will normally be aligned to improve readability. On a non-comment line, if & is the last non-blank character or the last non-blank character ahead of the comment symbol !, the statement continues from the character immediately preceding the &. Normally, continuation is to the first character of the next non-comment line, but if the first non-blank character of the next non-comment line is &, continuation is to the character following the &. For instance, the above statement may be written

```
x =                                                    &
   &(-y + root_of_discriminant)/(2.0*a)
```

In particular, if a token cannot be contained at the end of a line, the first non-blank character on the next noncomment line must be an & followed immediately by the remainder of the token.

Comments are allowed to contain any characters, including &, so they cannot be continued since a trailing & is taken as part of the comment. However, comment lines may be freely interspersed among continuation lines and do not count towards the limit of 39 lines.

In a character context, continuation must be from a line without a trailing comment and to a line with a leading ampersand. This is because both ! and & are permitted both in character contexts and in comments.

No line is permitted to have & as its only non-blank character, or as its only non-blank character ahead of !. Such a line is really a comment and becomes a comment if & is removed.

When writing short statements one after the other, it can be convenient to write several of them on one line. The semi-colon (;) character is used as a *statement separator* in these circumstances, for example:

```
a = 0; b = 0; c = 0
```

Since commentary always extends to the end of the line, it is not possible to insert commentary between statements on a single line. In principle, it is possible to write even long statements one after the other in a solid block of lines, each 132 characters long and with the appropriate semi-colons separating the individual statements. In practice, such code is unreadable, and the use of multiple-statement lines should be reserved for trivial cases such as the one shown in this example.

Any Fortran statement (that is not part of a compound statement) may be labelled, in order to be able to identify it. For some statements a label is mandatory.

A statement *label* precedes the statement, and is regarded as a token. The label consists of from one to five digits, one of which must be nonzero. An example of a labelled statement is

```
100 continue
```

Leading zeros are not significant in distinguishing between labels. For example, `10` and `010` are equivalent.

2.5 Concept of type

In Fortran, it is possible to define and manipulate various types of data. For instance, we may have available the value 10 in a program, and assign that value to an integer scalar variable denoted by `i`. Both 10 and `i` are of type integer; 10 is a fixed or *constant* value, whereas `i` is a *variable* which may be assigned other values. Integer expressions, such as `i+10`, are available too.

A *data type* consists of a set of data values, a means of denoting those values, and a set of operations that are allowed on them. For the integer data type, the values are $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$ between some limits depending on the kind of integer and computer system being used. Such tokens as these are *literal constants*, and each data type has its own form for expressing them. Named scalar variables, such as `i`, may be established. During the execution of a program, the value of `i` may change to any valid value, or may become *undefined*, that is have no predictable value. The operations which may be performed on integers are those of usual arithmetic; we can write `1+10` or `i-3` and obtain the expected results. Named constants may be established too; these have values that do not change during a given execution of the program.

Properties like those just mentioned are associated with all the data types of Fortran, and will be described in detail in this and the following chapters. The language itself contains five data types whose existence may always be assumed. These are known as the *intrinsic data types*, whose literal constants form the subject of the next section. Of each intrinsic type there is a default kind and a processor-dependent number of other kinds. Each kind is associated with a non-negative integer value known as the *kind type parameter*. This is used as a means of identifying and distinguishing the various kinds available.

In addition, it is possible to define other data types based on collections of data of the intrinsic types, and these are known as *derived data types*. The ability to define data types of interest to the programmer – matrices, geometrical shapes, lists, interval numbers – is a powerful feature of the language, one which permits a high level of *data abstraction*, that is the ability to define and manipulate data objects without being concerned about their actual representation in a computer.

2.6 Literal constants of intrinsic type

The intrinsic data types are divided into two classes. The first class contains three *numeric* types which are used mainly for numerical calculations – integer,

real, and complex. The second class contains the two *non-numeric* types which are used for such applications as text-processing and control – character and logical. The numerical types are used in conjunction with the usual operators of arithmetic, such as + and -, which will be described in Chapter 3. Each includes a zero and the value of a signed zero is the same as that of an unsigned zero[3]. The non-numeric types are used with sets of operators specific to each type; for instance, character data may be concatenated. These too will be described in Chapter 3.

2.6.1 Integer literal constants

The first type of literal constant is the *integer literal constant.* The default kind is simply a signed or unsigned integer value, for example

```
1
0
-999
32767
+10
```

The *range* of the default integers is not specified in the language, but on a computer with a word size of n bits, is often from -2^{n-1} to $+2^{n-1} - 1$. Thus on a 32-bit computer the range is often from −2147483648 to +2147483647.

To be sure that the range will be adequate on any computer requires the specification of the kind of integer by giving a value for the kind type parameter. This is best done through a named integer constant. For example, if the range −999999 to 999999 is desired, k6 may be established as a constant with an appropriate value by the statement, fully explained later,

```
integer, parameter :: k6=selected_int_kind(6)
```

and used in constants thus:

```
-123456_k6
+1_k6
~2_k6
```

Here, `selected_int_kind(6)` is an intrinsic inquiry function call, and it returns a kind parameter value that yields the range −999999 to 999999 with the least margin (see Section 8.7.4).

On a given processor, it might be known that the kind value needed is 3. In this case, the first of our constants can be written

```
-123456_3
```

[3]Although the representation of data is processor dependent, for the numeric data types the standard defines model representations and means to inquire about the properties of those models. The details are deferred to Section 8.7.

but this form is less portable. If we move the code to another processor, this particular value may be unsupported, or might correspond to a different range.

Many implementations use kind values that indicate the number of bytes of storage occupied by a value, but the standard allows greater flexibility. For example, a processor might have hardware only for 4-byte integers, and yet support kind values 1, 2, and 4 with this hardware (to ease portability from processors that have hardware for 1-, 2-, and 4-byte integers). However, the standard makes no statement about kind values or their order, except that the kind value is never negative.

The value of the kind type parameter for a given data type on a given processor can be obtained from the `kind` intrinsic function (Section 8.2):

```
kind(1)         for the default value
kind(2_k6)      for the example
```

and the decimal exponent range (number of decimal digits supported) of a given entity may be obtained from another function (Section 8.7.2), as in

```
range(2_k6)
```

which in this case would return a value of at least 6.

In addition to the usual integers of the decimal number system, for some applications it is very convenient to be able to represent positive whole numbers in binary, octal, or hexadecimal form. Unsigned constants of these forms exist in Fortran, and are represented as illustrated in these examples:

```
binary (base 2):        b'01100110'
octal (base 8):         o'076543'
hexadecimal (base 16):  z'10fa'
```

In the hexadecimal form, the letters `a` to `f` represent the values beyond 9; they may be used also in upper case. The delimiters may be quotation marks or apostrophes. The use of these forms of constants is limited to their appearance as implicit integers in the `data` statement (Section 7.5.2). A binary, octal, or hexadecimal constant may also appear in an internal or external file as a digit string, without the leading letter and the delimiters (see Section 9.13.2).

Bits stored as an integer representation may be manipulated by the intrinsic procedures described in Section 8.8.

2.6.2 Real literal constants

The second type of literal constant is the *real literal constant.* The default kind is a floating-point form built of some or all of: a signed or unsigned integer part, a decimal point, a fractional part, and a signed or unsigned exponent part. One or both of the integer part and fractional part must be present. The exponent part is either absent or consists of the letter `e` followed by a signed or unsigned integer. One or both of the decimal point and the exponent part must be present. An example is

```
-10.6e-11
```

meaning -10.6×10^{-11}, and other legal forms are

```
1.
-0.1
1e-1
3.141592653
```

The default real literal constants are representations of a subset of the real numbers of mathematics, and the standard specifies neither the allowed range of the exponent nor the number of significant digits represented by the processor. Many processors conform to the IEEE standard for floating-point arithmetic and have values of 10^{-37} to 10^{+37} for the range, and a precision of six decimal digits.

To be sure to obtain a desired range and significance requires the specification of a kind parameter value. For example,

```
integer, parameter :: long = selected_real_kind(9, 99)
```

ensures that the constants

```
1.7_long
12.3456789e30_long
```

have a precision of at least nine significant decimals, and an exponent range of at least 10^{-99} to 10^{+99}. The number of digits specified in the significand has no effect on the kind. In particular, it is permitted to write more digits than the processor can in fact use.

As for integers, many implementations use kind values that indicate the number of bytes of storage occupied by a value, but the standard allows greater flexibility. It specifies only that the kind value is never negative. If the desired kind value is known it may be used directly, as in the case

```
1.7_4
```

but the resulting code is then less portable.

The processor must provide at least one representation with more precision than the default, and this second representation may also be specified as `double precision`. We defer the description of this alternative but outmoded syntax to Section 11.4.1.

The `kind` function is valid also for real values:

```
kind(1.0)          for the default value
kind(1.0_long)     for the example
```

In addition, there are two inquiry functions available which return the actual precision and range, respectively, of a given real entity (see Section 8.7.2). Thus, the value of

```
precision(1.7_long)
```

would be at least 9, and the value of

```
range(1.7_long)
```

would be at least 99.

2.6.3 Complex literal constants

Fortran, as a language intended for scientific and engineering calculations, has the advantage of having as third literal constant type the *complex literal constant.* This is designated by a pair of literal constants, which are either integer or real, separated by a comma and enclosed in parentheses. Examples are

```
(1., 3.2)
(1, .99e-2)
(1.0, 3.7_8)
```

where the first constant of each pair is the real part of the complex number, and the second constant is the imaginary part. If one of the parts is integer, the kind of the complex constant is that of the other part. If both parts are integer, the kind of the constant is that of the default real type. If both parts are real and of the same kind, this is the kind of the constant. If both parts are real and of different kinds, the kind of the constant is that of one of the parts: the part with the greater decimal precision, or the part chosen by the processor if the decimal precisions are identical.

A default complex constant is one whose kind value is that of default real.

The `kind`, `precision`, and `range` functions are equally valid for complex entities.

Note that if an implementation uses the number of bytes needed to store a real as its kind value, the number of bytes needed to store a complex value of the corresponding kind is twice the kind value. For example, if the default real type has kind 4 and needs four bytes of storage, the default complex type has kind 4 but needs eight bytes of storage.

2.6.4 Character literal constants

The fourth type of literal constant is the *character literal constant.* The default kind consists of a string of characters enclosed in a pair of either apostrophes or quotation marks, for example

```
'Anything goes'

"Nuts & bolts"
```

The characters are not restricted to the Fortran set (Section 2.2). Any graphic character supported by the processor is permitted, but not control characters such as "newline". The apostrophes and quotation marks serve as *delimiters,* and are not part of the value of the constant. The value of the constant

```
'STRING'
```

is STRING. We note that in character constants the blank character is significant. For example

```
'a string'
```

is not the same as

```
'astring'
```

A problem arises with the representation of an apostrophe or a quotation mark in a character constant. Delimiter characters of one sort may be embedded in a string delimited by the other, as in the examples

```
'He said "Hello"'
"This contains an ' "
```

Alternatively, a doubled delimiter without any embedded intervening blanks is regarded as a single character of the constant. For example

```
'Isn''t it a nice day'
```

has the value `Isn't it a nice day`.

The number of characters in a string is called its *length*, and may be zero. For instance, `''` and `""` are character constants of length zero.

We mention here the particular rule for the source form concerning character constants that are written on more than one line (needed because constants may include the characters ! and &): not only must each line that is continued be without a trailing comment, but each continuation line must *begin* with a continuation mark. Any blanks following a trailing ampersand or preceding a leading ampersand are not part of the constant, nor are the ampersands themselves part of the constant. Everything else, including blanks, is part of the constant. An example is

```
long_string =                                                  &
     'Were I with her, the night would post too soon;          &
    & But now are minutes added to the hours;                  &
    & To spite me now, each minute seems a moon;               &
    & Yet not for me, shine sun to succour flowers!            &
    &   Pack night, peep day; good day, of night now borrow:   &
    &   Short, night, to-night, and length thyself tomorrow.'
```

On any computer, the characters have a property known as their *collating sequence*. One may ask the question whether one character occurs before or after another in the sequence. This question is posed in a natural form such as 'Does C precede M?', and we shall see later how this may be expressed in Fortran terms. Fortran requires the computer's collating sequence to satisfy the following conditions:

- A is less than B is less than C ... is less than Y is less than Z;
- 0 is less than 1 is less than 2 ... is less than 8 is less than 9;
- blank is less than A and Z is less than 0, or blank is less than 0 and 9 is less than A;

and, if the lower-case letters are available,

- a is less than b is less than c ... is less than y is less than z;
- blank is less than a and z is less than 0, or blank is less than 0 and 9 is less than a.

Thus we see that there is no rule about whether the numerals precede or succeed the letters, nor about the position of any of the special characters or the underscore, apart from the rule that blank precedes both partial sequences. Any given computer system has a complete collating sequence, and most computers nowadays use the collating sequence of the ASCII standard (also known as ISO/IEC 646:1991). However, Fortran is designed to accommodate other sequences, notably EBCDIC, so for portability, no program should ever depend on any ordering beyond that stated above. Alternatively, Fortran provides access to the ASCII collating sequence on any computer through intrinsic functions (Section 8.5.1), but this access is not so convenient and is less efficient on some computers.

A processor is required to provide access to the default kind of character constant just described. In addition, it may support other kinds of character constants, in particular those of non-European languages, which may have more characters than can be provided in a single byte. For example, a processor might support Kanji with the kind parameter value 2; in this case, a Kanji character constant may be written

```
2_国内
```

or

```
kanji_標準
```

where kanji is an integer named constant with the value 2. We note that, in this case, the kind type parameter exceptionally *precedes* the constant. This is necessary in order to enable compilers to parse statements simply.

There is no requirement on a processor to provide more than one kind of character, and the standard does not require any particular relationship between the kind parameter values and the character sets and the number of bytes needed to represent them. In fact, all that is required is that each kind of character set includes a blank character. As for the other data types, the kind function gives the actual value of the kind type parameter, as in

```
kind('ASCII')
```

Non-default characters are permitted in comments.

2.6.5 Logical literal constants

The fifth type of literal constant is the *logical literal constant.* The default kind has one of two values, `.true.` and `.false.`. These logical constants are normally used only to initialize logical variables to their required values, as we shall see in Section 3.6.

The default kind has a kind parameter value which is processor dependent. The actual value is available as `kind(.true.)`. As for the other intrinsic types, the kind parameter may be specified by an integer constant following an underscore, as in

```
.false._1
.true._long
```

Non-default logical kinds are useful for storing logical arrays compactly; we defer further discussion until Section 6.17.

2.7 Names

A Fortran program references many different entities by name. Such names must consist of between 1 and 31 alphanumeric characters (letters, underscores, and numerals) of which the first must be a letter. There are no other restrictions on the names; in particular there are no reserved words in Fortran. We thus see that valid names are, for example,

```
a
a_thing
x1
mass
q123
real
time_of_flight
```

and invalid names are

```
1a              First character is not alphabetic
a thing         Contains a blank
$sign           Contains a non-alphanumeric character
```

Within the constraints of the syntax, it is important for program clarity to choose names that have a clear significance – these are known as *mnemonic names.* Examples are `day`, `month`, and `year`, for variables to store the calendar date.

The use of names to refer to constants, already met in Section 2.6.1, will be fully described in Section 7.4.

2.8 Scalar variables of intrinsic type

We have seen in the section on literal constants that there exist five different intrinsic data types. Each of these types may have variables too. The simplest way by which a variable may be declared to be of a particular type is by specifying its name in a *type declaration statement* such as

```
integer   :: i
real      :: a
complex   :: current
logical   :: pravda
character :: letter
```

Here all the variables have default kind, and `letter` has default length, which is 1. Explicit requirements may also be specified through *type parameters*, as in the examples

```
integer(kind=4)              :: i
real(kind=long)              :: a
character(len=20, kind=1)    :: english_word
character(len=20, kind=kanji) :: kanji_word
```

Character is the only type to have two parameters, and here the two character variables each have length 20. Where appropriate, just one of the parameters may be specified, leaving the other to take its default value, as in the cases

```
character(kind=kanji) :: kanji_letter
character(len=20)     :: english_word
```

The shorter forms

```
integer(4)           :: i
real(long)           :: a
character(20, 1)     :: english_word
character(20, kanji) :: kanji_word
character(20)        :: english_word
```

are available, but note that

```
character(kanji) :: kanji_letter        ! Beware
```

is not an abbreviation for

```
character(kind=kanji) :: kanji_letter
```

because a single unnamed parameter is taken as the length parameter.

2.9 Derived data types

When programming, it is often useful to be able to manipulate objects that are more sophisticated than those of the intrinsic types. Imagine, for instance, that

we wished to specify objects representing persons. Each person in our application is distinguished by a name, an age, and an identification number. Fortran allows us to define a corresponding data type in the following fashion:

```
type person
   character(len=10) :: name
   real              :: age
   integer           :: id
end type person
```

This is the *definition* of the type. A scalar object of such a type is called a *structure*. In order to create a structure of that type, we write an appropriate type declaration statement, such as

```
type(person) :: you
```

The scalar variable you is then a composite object of type person containing three separate components, one corresponding to the name, another to the age, and a third to the identification number. As will be described in Sections 3.8 and 3.9, a variable such as you may appear in expressions and assignments involving other variables or constants of the same or different types. In addition, each of the components of the variable may be referenced individually using the *component selector* character percent (%). The identification number of you would, for instance, be accessed as

```
you%id
```

and this quantity is an integer variable which could appear in an expression such as

```
you%id + 9
```

Similarly, if there were a second object of the same type:

```
type(person) :: me
```

the differences in ages could be established by writing

```
you%age - me%age
```

It will be shown in Section 3.8 how a meaning can be given to an expression such as

```
you - me
```

Just as the intrinsic data types have associated literal constants, so too may literal constants of derived type be specified. Their form is the name of the type followed by the constant values of the components, in order and enclosed in parentheses. Thus, the constant

```
person( 'Smith', 23.5, 2541)
```

may be written assuming the derived type defined at the beginning of this section, and could be *assigned* to a variable of the same type:

```
you = person( 'Smith', 23.5, 2541)
```

Any such *structure constructor* can appear only after the definition of the type.

A derived type may have a component that is of a previously defined derived type. This is illustrated in Figure 2.1. A variable of type `triangle` may be declared thus

```
type(triangle) :: t
```

and `t` has components `t%a`, `t%b`, and `t%c` all of type `point`, and `t%a` has components `t%a%x` and `t%a%y` of type real.

Figure 2.1

```
type point
   real :: x, y
end type point
type triangle
   type(point) :: a, b, c
end type triangle
```

2.10 Arrays of intrinsic type

Another compound object supported by Fortran is the *array*. An array consists of a rectangular set of elements, all of the same type and type parameters. There are a number of ways in which arrays may be declared; for the moment we shall consider only the declaration of arrays of fixed sizes. To declare an array named `a` of 10 real elements, we add the `dimension` attribute to the type declaration statement thus:

```
real, dimension(10) :: a
```

The successive elements of the array are `a(1)`, `a(2)`, `a(3)`, ..., `a(10)`. The number of elements of an array is called its *size*. Each array element is a scalar.

Many problems require a more elaborate declaration than one in which the first element is designated 1, and it is possible in Fortran to declare a lower as well as an upper *bound*:

```
real, dimension(-10:5) :: vector
```

This is a vector of 16 elements, `vector(-10)`, `vector(-9)`, ..., `vector(5)`. We thus see that whereas we always need to specify the upper bound, the lower bound is optional, and by default has the value 1.

An array may extend in more than one dimension, and Fortran allows up to seven dimensions to be specified. For instance

```
real, dimension(5,4) :: b
```

declares an array with two dimensions, and

```
real, dimension(-10:5, -20:-1, 0:1, -1:0, 2, 2, 2) :: grid
```

declares seven dimensions, the first four with explicit lower bounds. It may be seen that the size of this second array is

$$16 \times 20 \times 2 \times 2 \times 2 \times 2 \times 2 = 10240,$$

and that arrays of many dimensions can thus place large demands on the memory of a computer. The number of dimensions of an array is known as its *rank*. Thus, `grid` has a rank of seven. Scalars are regarded as having rank zero. The number of elements along a dimension of an array is known as the *extent* in that dimension. Thus, `grid` has extents 16, 20, ...

The sequence of extents is known as the *shape*. For example, `grid` has the shape (16, 20, 2, 2, 2, 2, 2).

A derived type may contain an array component. For example, the following type

```
type triplet
   real                  :: u
   real, dimension(3)    :: du
   real, dimension(3,3)  :: d2u
end type triplet
```

might be used to hold the value of a variable in three dimensions and the values of its first and second derivatives. If `t` is of type `triplet`, `t%du` and `t%d2u` are arrays of type real.

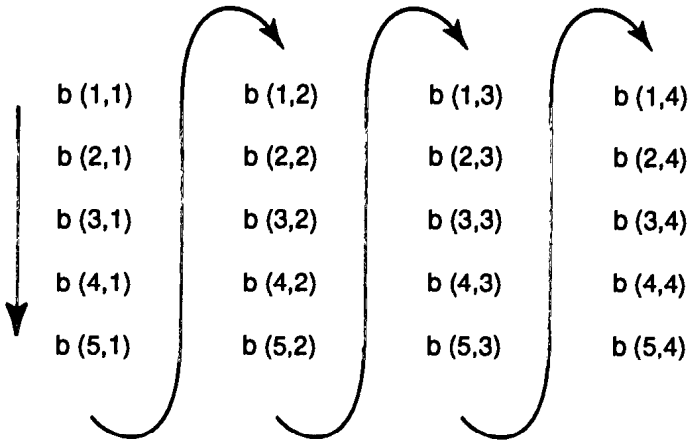
Some statements treat the elements of an array one-by-one in a special order which we call the *array element order*. It is obtained by counting most rapidly in the early dimensions. Thus, the elements of `grid` in array element order are

```
grid(-10, -20, 0, -1, 1, 1, 1)
grid( -9, -20, 0, -1, 1, 1, 1)
    :
grid(  5,  -1, 1,  0, 2, 2, 2).
```

This is illustrated for an array of two dimensions in Figure 2.2. Most implementations actually store arrays in contiguous storage in array element order, but we emphasize that the standard does not require this.

We reference an individual element of an array by specifying, as in the examples above, its *subscript* values. In the examples we used integer constants, but in general each subscript may be formed of a *scalar integer expression*, that is, any arithmetic expression whose value is scalar and of type integer. Each subscript must be within the corresponding ranges defined in the array declaration and the number of subscripts must equal the rank. Examples are

Figure 2.2 The ordering of elements in the array `b(5,4)`.




```
a(1)
a(i*j)            ! i and j are of type integer
a(nint(x+3.))     ! x is of type real
t%d2u(i+1,j+2)    ! t is of derived type triplet
```

where `nint` is an intrinsic function to convert a real value to the nearest integer (see Section 8.3.1). In addition subarrays, called *sections*, may be referenced by specifying a range for one or more subscripts. The following are examples of array sections:

```
a(i:j)            ! Rank-one array of size j-i+1
b(k, 1:n)         ! Rank-one array of size n
c(1:i, 1:j, k)    ! Rank-two array with extents i and j
```

We describe array sections in more detail in Section 6.13. An array section is itself an array, but its individual elements must not be accessed through the section designator. Thus, `b(k, 1:n)(1)` cannot be written; it must be expressed as `b(k, 1)`.

A further form of subscript is shown in

```
a(ipoint)         ! ipoint is an integer array
```

where `ipoint` is an array of indices, pointing to array elements. It may thus be seen that `a(ipoint)`, which identifies as many elements of `a` as `ipoint` has elements, is an example of another *array-valued object*, and `ipoint` is referred to as a *vector subscript*. This will be met in greater detail in Section 6.13.

It is often convenient to be able to define an array constant. In Fortran, a rank-one array may be constructed as a list of elements enclosed between the tokens (/ and /). A simple example is

```
(/ 1, 2, 3, 5, 10 /)
```

which is an array of rank one and size five. To obtain a series of values, the individual values may be defined by an expression that depends on an integer variable having values in a range, with an optional stride. Thus, the constructor

```
(/1, 2, 3, 4, 5/)
```

can be written as

```
(/ (i, i = 1,5) /)
```

and

```
(/2, 4, 6, 8/)
```

as

```
(/ (i, i = 2,8,2) /)
```

and

```
(/ 1.1, 1.2, 1.3, 1.4, 1.5 /)
```

as

```
(/ (i*0.1, i=11,15) /)
```

An array constant of rank greater than one may be constructed by using the function reshape (see Section 8.13.3) to reshape a rank-one array constant.

A full description of array constructors is reserved for Section 6.16.

2.11 Character substrings

It is possible to build arrays of characters, just as it is possible to build arrays of any other type:

```
character, dimension(80) :: line
```

declares an array, called `line`, of 80 elements, each one character in length. Each character may be addressed by the usual reference, `line(i)` for example. In this case, however, a more appropriate declaration might be

```
character(len=80) :: line
```

which declares a scalar data object of 80 characters. These may be referenced individually or in groups using a *substring* notation

```
line(i:j)    ! i and j are of type integer
```

which references all the characters from `i` to `j` in `line`. The colon is used to separate the two substring subscripts, which may be any scalar integer expressions. The colon is obligatory in substring references, so that referencing a single character requires `line(i:i)`. There are default values for the substring subscripts. If the lower one is omitted, the value 1 is assumed; if the upper one is omitted, a value corresponding to the character length is assumed. Thus,

```
line(:i)   is equivalent to line(1:i)
line(i:)   is equivalent to line(i:80)
line(:)    is equivalent to line or line(1:80)
```

If `i` is greater than `j` in `line(i:j)`, the value is a zero-sized string.

We may now combine the length declaration with the array declaration to build arrays of character objects of specified length, as in

```
character(len=80), dimension(60) :: page
```

which might be used to define storage for the characters of a whole page, with 60 elements of an array, each of length 80. To reference the line `j` on a page we may write `page(j)`, and to reference character `i` on that line we could combine the array subscript and character substring notations into

```
page(j)(i:i)
```

A substring of a character constant or of a structure component may also be formed:

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'(j:j)
 you%name(1:2)
```

At this point we must note a limitation associated with character variables, namely that character variables must have a declared maximum length, making it impossible to manipulate character variables of variable length, unless they are defined appropriately as of a derived data type.[4] Nevertheless, this data type is adequate for most character manipulation applications.

2.12 Objects and subobjects

We have seen that derived types may have components that are arrays, as in

```
type triplet
   real, dimension(3) :: vertex
end type triplet
```

and arrays may be of derived type as in the example

```
type(triplet), dimension(10) :: t
```

[4]But see Section 1.5.

A single structure (for example, `t(2)`) is always regarded as a scalar, but it may have a component (for example, `t(2)%vertex`) that is an array. Derived types may have components of other derived types.

An object referenced by an unqualified name (up to 31 alphanumeric characters) is called a *named object* and is not part of a bigger object. Its subobjects have *designators* that consist of the name of the object followed by one or more qualifiers (for example, `t(1:7)` and `t(1)%vertex`). Each successive qualifier specifies a part of the object specified by the name or designator that precedes it.

Because of these possibilities, the terms *array* and *variable* are now used with a more general meaning than in Fortran 77. The term 'array' is used for any object that is not scalar, including an array section or an array-valued component of a structure. The term 'variable' is used for any named object that is not specified to be a constant and for any part of such an object, including array elements, array sections, structure components, and substrings.

2.13 Pointers

In everyday language, nouns are often used in a way that makes their meaning precise only because of the context. 'The chairman said that ...' will be understood precisely by the reader who knows that the context is the Fortran Committee developing Fortran 90 and that its chairman was then Jeanne Adams.

Similarly, in a computer program it can be very useful to be able to use a name that can be made to refer to different objects during execution of the program. One example is the multiplication of a vector by a sequence of square matrices. We might write code that calculates

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1, 2, \ldots, n$$

from the vector x_j, $j = 1, 2, \ldots, n$. In order to use this to calculate

$$BCz$$

we might first make x refer to z and A refer to C, thereby using our code to calculate $y = Cz$, then make x refer to y and A refer to B so that our code calculates the result vector we finally want.

An object that can be made to refer to other objects in this way is called a *pointer*, and must be declared with the pointer attribute, for example

```
real, pointer                  :: son
real, pointer, dimension(:)    :: x, y
real, pointer, dimension(:,:) :: a
```

In the case of an array, only the rank (number of dimensions) is declared, and the bounds (and hence shape) are taken from that of the object to which it points. Given such a declaration, the compiler arranges storage for a descriptor that is

initially undefined but will later hold the address of the actual object (known as the *target*) and holds, if it is an array, its bounds and strides.

Besides pointing to existing variables, a pointer may be made explicitly to point at nothing:

```
nullify (son, x, y, a)   ! Further details are in Section 6.5.4
```

or may be given fresh storage by an `allocate` statement such as

```
allocate (son, x(10), y(-10:10), a(n, n))
```

In the case of arrays, the lower and upper bounds are specified just as for the `dimension` attribute (Section 2.10) except that any scalar integer expression is permitted. Permitting such expressions remedies one of the major deficiencies of Fortran 77, namely that areas of only static storage may be defined, and we will discuss this use further in Section 6.5.

Components of derived types are permitted to have the pointer attribute. This enables a major application of pointers: the construction of linked lists. As a simple example, we might decide to hold a sparse vector as a chain of variables of the type shown in Figure 2.3, which allows us to access the entries one by one; given

```
type(entry), pointer :: chain
```

where `chain` is a scalar of this type and holds a chain that is of length two, its entries are `chain%index` and `chain%next%index`, and `chain%next%next` will have been nullified. Additional entries may be created when necessary by an appropriate allocate statement. We defer the details to Section 3.12.

Figure 2.3

```
type entry
   real                  :: value
   integer               :: index
   type(entry), pointer :: next
end type entry
```

A subobject is not a pointer unless it has a final component selector for the name of a pointer component, for example, `chain%next`.

2.14 Summary

In this chapter, we have introduced the elements of the Fortran language. The character set has been listed, and the manner in which sequences of characters form literal constants and names explained. In this context, we have encountered the five intrinsic data types defined in Fortran, and seen how each data type has corresponding literal constants and named objects. We have seen how derived

types may be constructed from the intrinsic types. We have introduced one method by which arrays may be declared, and seen how their elements may be referenced by subscript expressions. The concepts of the array section, character substring, and pointer have been presented, and some important terms defined. In the following chapter we shall see how these elements may be combined into expressions and statements, Fortran's equivalents of 'phrases' and 'sentences'.

With respect to Fortran 77, there are many changes in this area: the larger character set; a new source form; the significance of blanks; the parameterization of the intrinsic types; the binary, octal, and hexadecimal constants; the ability to obtain a desired precision and range; quotes as well as apostrophes as character constant delimiters; longer names; derived data types; new array subscript notations; array constructors; and last, but not least, pointers. Together they represent a substantial improvement in the ease of use and power of the language.

2.15 Exercises

1. For each of the following assertions, state whether it is true, false or not determined, according to the Fortran collating sequences:

```
b is less than m
8 is less than 2
* is greater than T
$ is less than /
blank is greater than A
blank is less than 6
```

2. Which of the Fortran lines in Figure 2.4 are correctly written according to the requirements of the Fortran source form? Which ones contain commentary? Which lines are initial lines and which are continuation lines?

Figure 2.4

```
      x = y
  3   a = b+c ! add
      word = 'string'
      a = 1.0; b = 2.0
      a = 15. ! initialize a; b = 22. ! and b
      song = "Life is just&
         & a bowl of cherries"
      chide = 'Waste not,
         want not!'
  0   c(3:4) = 'up"
```

3. Classify the following literal constants according to the five intrinsic data types of Fortran. Which are not legal literal constants?

```
-43                'word'
4.39               1.9-4
0.0001e+20         'stuff & nonsense'
4 9                (0.,1.)
(1.e3,2)           'I can''t'
'(4.3e9, 6.2)'     .true._1
e5                 'shouldn' 't'
1_2                "O.K."
z10                z'10'
```

4. Which of the following names are legal Fortran names?

```
name        name32
quotient    123
a182c3      no-go
stop!       burn_
no_go       long__name
```

5. What are the first, tenth, eleventh and last elements of the following arrays?

```
real, dimension(11)       :: a
real, dimension(0:11)     :: b
real, dimension(-11:0)    :: c
real, dimension(10,10)    :: d
real, dimension(5,9)      :: e
real, dimension(5,0:1,4)  :: f
```

Write an array constructor of eleven integer elements.

6. Given the array declaration

```
character(len=10), dimension(0:5,3) :: c
```

which of the following subobject designators are legal?

```
c(2,3)          c(4:3)(2,1)
c(6,2)          c(5,3)(9:9)
c(0,3)          c(2,1)(4:8)
c(4,3)(:)       c(3,2)(0:9)
c(5)(2:3)       c(5:6)
c(5,3)(9)       c(,)
```

7. Write derived type definitions appropriate for:

a) a vehicle registration;

b) a circle;

c) a book (title, author, and number of pages).

Give an example of a derived type constant for each one.

8. Given the declaration for t in Section 2.12, which of the following objects and subobjects are arrays?

```
t                t(4)%vertex(1)
t(10)            t(5:6)
t(1)%vertex      t(5:5)
```

9. Write specifications for these entities:

a) an integer variable inside the range -10^{20} to 10^{20};

b) a real variable with a minimum of 12 decimal digits of precision and a range of 10^{-100} to 10^{100};

c) a Kanji character variable on a processor that supports Kanji with `kind=2`.

3. Expressions and assignments

3.1 Introduction

We have seen in the previous chapter how we are able to build the 'words' of Fortran — the constants, keywords, and names — from the basic elements of the character set. In this chapter we shall discover how these entities may be further combined into 'phrases' or *expressions,* and how these, in turn, may be combined into 'sentences', or *statements.*

In an expression, we describe a computation that is to be carried out by the computer. The result of the computation may then be assigned to a variable. A sequence of assignments is the way in which we specify, step-by-step, the series of individual computations to be carried out, in order to arrive at the desired result. There are separate sets of rules for expressions and assignments, depending on whether the operands in question are numeric, logical, character, or derived in type, and whether they are scalars or arrays. There are also separate rules for pointer assignments. We shall discuss each set of rules in turn, including a description of the relational expressions which produce a result of type logical and are needed in control statements (see next chapter). To simplify the initial discussion, we commence by considering expressions and assignments that are intrinsically defined and involve neither arrays nor entities of derived data types.

An expression in Fortran is formed of operands and operators, combined in a way which follows the rules of Fortran syntax. A simple expression involving a *dyadic* (or binary) operator has the form

operand *operator* operand

an example being

```
x+y
```

and a unary or *monadic* operator has the form

operator operand

an example being

```
-y
```

The operands may be constants, variables, or functions (see Chapter 5), and an expression may itself be used as an operand. In this way, we can build up more complicated expressions such as

operand *operator* operand *operator* operand

where consecutive operands are separated by a single operator. Each operand must have a defined value and the result must be mathematically defined; for example, raising a negative real value to a real power is not permitted.

The rules of Fortran state that the parts of expressions without parentheses are evaluated successively from left to right for operators of equal precedence, with the exception of ** (see Section 3.2). If it is necessary to evaluate part of an expression, or *subexpression*, before another, parentheses may be used to indicate which subexpression should be evaluated first. In

operand *operator* (operand *operator* operand)

the subexpression in parentheses will be evaluated, and the result used as an operand to the first operator.

If an expression or subexpression has no parentheses, the processor is permitted to evaluate an equivalent expression, that is an expression that always has the same value apart, possibly, from the effects of numerical round-off. For example, if a, b, and c are real variables, the expression

```
a/b/c
```

might be evaluated as

```
a/(b*c)
```

on a processor that can multiply much faster than it can divide. Usually, such changes are welcome to the programmer since the program runs faster, but when they are not (for instance because they would lead to more round-off) parentheses should be inserted because the processor is required to respect them.

If two operators immediately follow each other, as in

operand *operator operator* operand

the only possible interpretation is that the second operator is unary. Thus, there is a general rule that a binary operator must not follow immmediately after another operator.

3.2 Scalar numeric expressions

A *numeric expression* is an expression whose operands are one of the three numeric types — integer, real, and complex — and whose operators are

**	exponentiation
* /	multiplication, division
+ -	addition, subtraction

These operators are known as *numeric intrinsic* operators, and are listed here in their order of precedence. In the absence of parentheses, exponentiations will be carried out before multiplications and divisions, and these before additions and subtractions.

We note that the minus sign (-) and the plus sign (+) can be used as a unary operators, as in

```
-tax
```

Because it is not permitted in ordinary mathematical notation, a unary minus or plus must not follow immediatedly after another operator. When this is needed, as for x^{-y}, parentheses must be placed around the operator and its operand:

```
x**(-y)
```

The type and kind type parameter of the result of a unary operation are those of the operand.

The exception to the left-to-right rule noted in Section 3.1 concerns exponentiations. Whereas the expression

```
-a+b+c
```

will be evaluated from left to right as

```
((-a)+b)+c
```

the expression

```
a**b**c
```

will be evaluated as

```
a**(b**c)
```

For integer data, the result of any division will be truncated towards zero, that is to the integer value whose magnitude is equal to or just less than the magnitude of the exact result. Thus, the result of

```
 6/3   is  2
 8/3   is  2
-8/3   is -2
```

This fact must always be borne in mind whenever integer divisions are written. Similarly, the result of

```
2**3   is 8
```

whereas the result of

```
2**(-3)    is 1/(2**3)
```

which is zero.

The rules of Fortran allow a numeric expression to contain numeric operands of differing types or kind type parameters. This is known as a *mixed-mode expression.* Except when raising a real or complex value to an integer power, the object of the weaker (or simpler) of the two data types will be converted, or *coerced,* into the type of the stronger one. The result will also be that of the stronger type. If, for example, we write

```
a*i
```

when `a` is of type real and `i` is of type integer, then `i` will be converted to a real data type before the multiplication is performed, and the result of the computation will also be of type real. The rules are summarized for each possible combination for the operations +, -, * and / in Table 3.1, and for the operation ** in Table 3.2. The functions `real` and `cmplx` that they reference are defined in Section 8.3.1. In both Tables, I stands for integer, R for real, and C for complex.

Table 3.1. Type of result of *a* .op. *b*, where .op. is +, -, * or /.

Type of *a*	**Type of *b***	**Value of *a* used**	**Value of *b* used**	**Type of result**
I	I	*a*	*b*	I
I	R	`real(`*a*`,kind(`*b*`))`	*b*	R
I	C	`cmplx(`*a*`,0,kind(`*b*`))`	*b*	C
R	I	*a*	`real(`*b*`,kind(`*a*`))`	R
R	R	*a*	*b*	R
R	C	`cmplx(`*a*`,0,kind(`*b*`))`	*b*	C
C	I	*a*	`cmplx(`*b*`,0,kind(`*a*`))`	C
C	R	*a*	`cmplx(`*b*`,0,kind(`*a*`))`	C
C	C	*a*	*b*	C

If both operands are of type integer, the kind type parameter of the result is that of the operand with the greater decimal exponent range, or is processor dependent if the kinds differ but the decimal exponent ranges are the same. If both operands are of type real or complex, the kind type parameter of the result is that of the operand with the greater decimal precision, or is processor dependent if the kinds differ but the decimal precisions are the same. If one operand is of type integer and the other is of real or complex, the type parameter of the result is that of the real or complex operand.

In the case of raising a complex value to a complex power, the principal value[1] is taken.

[1] The principal value of a^b is $\exp(b(\log|a| + i \arg a))$, with $-\pi < \arg a \leq \pi$.

Table 3.2. Type of result of *a****b*.

Type of *a*	Type of *b*	Value of *a* used	Value of *b* used	Type of result
I	I	*a*	*b*	I
I	R	`real(`*a*`,kind(`*b*`))`	*b*	R
I	C	`cmplx(`*a*`,0,kind(`*b*`))`	*b*	C
R	I	*a*	*b*	R
R	R	*a*	*b*	R
R	C	`cmplx(`*a*`,0,kind(`*b*`))`	*b*	C
C	I	*a*	*b*	C
C	R	*a*	`cmplx(`*b*`,0,kind(`*a*`))`	C
C	C	*a*	*b*	C

3.3 Defined and undefined variables

In the course of the explanations in this and the following chapters, we shall often refer to a variable becoming *defined* or *undefined*. In the previous chapter, we showed how a scalar variable may be called into existence by a statement like

```
real :: speed
```

In this simple case, the variable speed has, at the beginning of the execution of the program, no defined value. It is undefined. No attempt must be made to reference its value since it has none. A common way in which it might become defined is for it to be assigned a value:

```
speed = 2.997
```

After the execution of such an *assignment statement* it has a value, and that value may be referenced, for instance in an expression:

```
speed*0.5
```

For a compound object, it is necessary for all its subobjects to be individually defined before the object as a whole is regarded as defined. Thus, an array is said to be defined only when each of its elements is defined, an object of a derived data type is defined only when each of its components is defined, and a character variable is defined only when each of its characters is defined.

A variable that is defined does not necessarily retain its state of definition throughout the execution of a program. As we shall see in Chapter 5, a variable that is local to a single subprogram usually becomes undefined when control is returned from that subprogram. In certain circumstances, it is even possible that a single array element becomes undefined: this causes the array considered as a whole to become undefined; a similar rule holds for entities of derived data type and for character variables.

A means to specify the initial value of a variable will be explained in Section 7.5.

In the case of a pointer, the pointer association status may be *undefined* (its initial state), *associated* with a target, or *disassociated*, which means that it is not associated with a target but has a definite status that may be tested by the function `associated` (Section 8.2). Even though a pointer is associated with a target, the target itself may be defined or undefined. A means to specify the initial status of disassociated is provided by Fortran 95 (see Section 7.5.3).

3.4 Scalar numeric assignment

The general form of a scalar numeric assignment is

variable = *expr*

where *variable* is a scalar numeric variable and *expr* is a scalar numeric expression. If *expr* is not of the same type or kind as *variable*, it will be converted to that type and kind before the assignment is carried out, according to the set of rules given in Table 3.3 (the function `int` is defined in Section 8.3.1).

Table 3.3. Numeric conversion for assignment statement *variable* = *expr*

Type of *variable*	**Value assigned**
integer	`int`(*expr*, `kind`(*variable*))
real	`real`(*expr*, `kind`(*variable*))
complex	`cmplx`(*expr*, `kind=kind`(*variable*))

We note that if the type of *variable* is integer but *expr* is not, then the assignment will result in a loss of precision unless *expr* happens to have an integral value. Similarly, assigning a real expression to a real variable of a kind with less precision will also cause a loss of precision to occur, and the assignment of a complex quantity to a non-complex variable involves the loss of the imaginary part. Thus, the values in `i` and `a` following the assignments

```
i = 7.3                    ! i of type default integer
a = (4.01935, 2.12372)     ! a of type default real
```

are 7 and 4.01935, respectively.

3.5 Scalar relational operators

It is possible in Fortran to test whether the value of one numeric expression bears a certain relation to that of another, and similarly for character expressions. The relational operators are

`.lt.` or `<`	less than
`.le.` or `<=`	less than or equal
`.eq.` or `==`	equal
`.ne.` or `/=`	not equal
`.gt.` or `>`	greater than
`.ge.` or `>=`	greater than or equal

If either or both of the expressions are complex, only the operators == and /= (or `.eq.` and `.ne.`) are available.

The result of such a comparison is one of the default logical values `.true.` or `.false.`, and we shall see in the next chapter how such tests are of great importance in controlling the flow of a program. Examples of relational expressions (for `i` and `j` of type integer, `a` and `b` of type real, and `char1` of type default character) are

`i < 0`	integer relational expression
`a < b`	real relational expression
`a+b > i-j`	mixed-mode relational expression
`char1 == 'Z'`	character relational expression

In the third expression above, we note that the two components are of different numeric types. In this case, and whenever either or both of the two components consist of numeric expressions, the rules state that the components are to be evaluated separately, and converted to the type and kind of their sum before the comparison is made. Thus, a relational expression such as

```
a+b .le. i-j
```

will be evaluated by converting the result of `(i-j)` to type real.

For character comparisons, the kinds must be the same and the letters are compared from the left until a difference is found or the strings are found to be identical. If the lengths differ, the shorter one is regarded as being padded with blanks[2] on the right. Two zero-sized strings are considered to be identical.

No other form of mixed mode relational operator is intrinsically available, though such an operator may be defined (Section 3.8). The numeric operators take precedence over the relational operators.

3.6 Scalar logical expressions and assignments

Logical constants, variables, and functions may appear as operands in logical expressions. The logical operators, in decreasing order of precedence, are:

[2]Here and elsewhere, the blank padding character used for a non-default type is processor dependent.

unary operator:

`.not.`	logical negation

binary operators:

`.and.`	logical intersection
`.or.`	logical union
`.eqv.` and `.neqv.`	logical equivalence and non-equivalence

If we assume a logical declaration of the form

```
logical i,j,k,l
```

then the following are valid logical expressions:

```
.not.j
j .and. k
i .or. l .and.  .not.j
( .not.k .and. j .neqv. .not.l) .or. i
```

In the first expression we note the use of `.not.` as a unary operator. In the third expression, the rules of precedence imply that the subexpression `l.and..not.j` will be evaluated first, and the result combined with `i`. In the last expression, the two subexpressions `.not.k.and.j` and `.not.l` will be evaluated and compared for non-equivalence. The result of the comparison, `.true.` or `.false.`, will be combined with `i`.

The kind type parameter of the result is that of the operand for `.not.`, and for the others is that of the operands if they have the same kind or processor dependent otherwise.

We note that the `.or.` operator is an inclusive operator; the `.neqv.` operator provides an exclusive logical or (`a.and..not.b .or. .not.a.and.b`).

The result of any logical expression is the value true or false, and this value may then be assigned to a logical variable such as element 3 of the logical array `flag` in the example

```
flag(3) = ( .not. k .eqv. l) .or. j
```

The kind type parameter values of the variable and expression need not be identical.

A logical variable may be set to a predetermined value by an assignment statement:

```
flag(1) = .true.
flag(2) = .false.
```

In the foregoing examples, all the operands and results were of type logical – no other data type is allowed to participate in an intrinsic logical operation or assignment.

The results of several relational expressions may be combined into a logical expression, and assigned, as in

```
real    :: a, b, x, y
logical :: cond
:
cond = a>b .or. x<0.0 .and. y>1.0
```

where we note the precedence of the relational operators over the logical operators. If the value of such a logical expression can be determined without evaluating a subexpression, a processor is permitted not to evaluate the subexpression. An example is

```
i<=10 .and. ary(i)==0
```

when i has the value 11. However, the programmer must not rely on such behaviour. For example, when ary has size 10, an out-of-bounds subscript might be referenced if the processor chooses to evaluate the right-hand subexpression before the left-hand one. We return to this topic in Section 5.10.1.

3.7 Scalar character expressions and assignments

The only intrinsic operator for character expressions is the concatenation operator //, which has the effect of combining two character operands into a single character result. For example, the result of concatenating the two character constants AB and CD, written as

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

is the character string ABCD. The operands must have the same kind parameter values, but may be character variables, constants, or functions. For instance, if word1 and word2 are both of default kind and length 4, and contain the character strings LOOP and HOLE respectively, the result of

```
word1(4:4)//word2(2:4)
```

is the string POLE.

The length of the result of a concatenation is the sum of the lengths of the operands. Thus, the length of the result of

```
word1//word2//'S'
```

is 9, which is the length of the string LOOPHOLES.

The result of a character expression may be assigned to a character variable of the same kind. Assuming the declarations

```
character(len=4) :: char1, char2
character(len=8) :: result
```

we may write

```
char1 = 'any '
char2 = 'book'
result = char1//char2
```

In this case, `result` will now contain the string `any book`. We note in these examples that the lengths of the left- and right-hand sides of the three assignments are in each case equal. If, however, the length of the result of the right-hand side is shorter than the length of the left-hand side, then the result is placed in the left-most part of the left-hand side and the rest is filled with blank characters. Thus, in

```
character(len=5) :: fill
fill(1:4) = 'AB'
```

`fill(1:4)` will have the value AB*bb* (where *b* stands for a blank character). The value of `fill(5:5)` remains undefined, that is, it contains no specific value and should not be used in an expression. As a consequence, `fill` is also undefined. On the other hand, when the left-hand side is shorter than the result of the right-hand side, the right-hand end of the result is truncated. The result of

```
character(len=5) :: trunc8
trunc8 = 'TRUNCATE'
```

is to place in `trunc8` the character string `TRUNC`. If a left-hand side is of zero length, no assignment takes place.

The left-hand and right-hand sides of an assignment may overlap. In such a case, it is always the old values that are used in the right-hand side expression. For example, the assignment

```
result(3:5) = result(1:3)
```

is valid and if `result` began with the value `ABCDEFGH`, it would be left with the value `ABABCFGH`.

Other means of manipulating characters and strings of characters, via intrinsic functions, are described in Sections 8.5 and 8.6.

3.8 Structure constructors and scalar defined operators

No operators for derived types are automatically available, but a structure may be constructed from expressions for its components, just as a constant structure may be constructed from constants (Section 2.9). The general form of a *structure constructor* is

type-name (*expr-list*)

where the *expr-list* specifies the values of the components. For example, given the type

```
type string
   integer           :: length
   character(len=10) :: value
end type string
```

and the variables

```
character(len=4) :: char1, char2
```

the following is a value of type `string`

```
string(8, char1//char2)
```

Each expression in *expr-list* corresponds to a component of the structure; if it is not a pointer component, the value is assigned to the component under the rules of intrinsic assignment; if it is a pointer component, the expression must be a valid target for it,[3] as in a pointer assignment statement (Section 3.12).

When a programmer defines a derived type and wishes operators to be available, he or she must define the operators, too. For a binary operator this is done by writing a function, with two intent `in` arguments, that specifies how the result depends on the operands, and an interface block that associates the function with the operator token (functions, intent, and interface blocks will be explained fully in Chapter 5). For example, given the type

```
type interval
   real :: lower, upper
end type interval
```

that represents intervals of numbers between a lower and an upper bound, we may define addition by a module (Section 5.5) containing the procedure

```
function add_interval(a,b)
   type(interval)             :: add_interval
   type(interval), intent(in) :: a, b
   add_interval%lower = a%lower + b%lower ! Production code
   add_interval%upper = a%upper + b%upper ! would allow for
end function add_interval                 ! roundoff.
```

and the interface block (Section 5.18)

```
interface operator(+)
   module procedure add_interval
end interface
```

This function would be invoked in an expression such as

```
y + z
```

to perform this programmer-defined add operation for scalar variables y and z of type `interval`. A unary operator is defined by an interface block and a function with one intent `in` argument.

The operator token may be any of the tokens used for the intrinsic operators or may be a sequence of up to 31 letters enclosed in decimal points other than `.true.` or `.false.` . An example is

[3] In particular, it must not be a constant.

```
.sum.
```

In this case, the header line of the interface block would be written as

```
interface operator(.sum.)
```

and the expression as

```
y.sum.z
```

If an intrinsic token is used, the number of arguments must be the same as for the intrinsic operation, the priority of the operation is as for the intrinsic operation, and a unary minus or plus must not follow immediately after another operator. Otherwise, it is of highest priority for defined unary operators and lowest priority for defined binary operators. The complete set of priorities is given in Table 3.4. Where another priority is required within an expression, parentheses must be used.

Table 3.4. Relative precedence of operators (in decreasing order)

Type of operation when intrinsic	**Operator**
—	monadic (unary) defined operator
Numeric	**
Numeric	* or /
Numeric	monadic + or -
Numeric	dyadic + or -
Character	//
Relational	.eq. .ne. .lt. .le. .gt. .ge. == /= < <= > >=
Logical	.not.
Logical	.and.
Logical	.or.
Logical	.eqv. or .neqv.
—	dyadic (binary) defined operator

Retaining the intrinsic priorities is helpful both to the readability of expressions and to the efficiency with which a compiler can interpret them. For example, if + is used for set union and * for set intersection, we can interpret the expression

```
i*j + k
```

for sets `i`, `j`, and `k` without difficulty.

If either of the intrinsic tokens `.eq.` and == is used, the definition applies to both tokens so that they are always equivalent. The same is true for the other equivalent pairs of relational operators.

Note that a defined unary operator not using an intrinsic token may follow immediately after another operator as in

```
y .sum. .inverse. x
```

Operators may be defined for any types of operands, except where there is an intrinsic operation for the operator and types. For example, we might wish to be able to add an interval number to an ordinary real, which can be done by adding the procedure

```
function add_interval_real(a,b)
   type(interval)                add_interval_real
   type(interval), intent(in) :: a
   real, intent(in)           :: b
   add_interval_real%lower = a%lower + b ! Production code would
   add_interval_real%upper = a%upper + b ! allow for roundoff.
end function add_interval_real
```

and changing the interface block to

```
interface operator(+)
   module procedure add_interval, add_interval_real
end interface
```

The result of a defined operation may have any type. The type of the result, as well as its value, must be specified by the function.

Note that an operation that is defined intrinsically cannot be redefined; thus in

```
real :: a, b, c
:
c = a + b
```

the meaning of the operation is always unambiguous.

3.9 Scalar defined assignments

Assignment of an expression of derived type to a variable of the same type is automatically available and takes place component by component. For example, if a is of the type `interval` defined at the start of Section 3.8, we may write

```
a = interval(0.0, 1.0)
```

(structure constructors were met in Section 3.8, too).

In other circumstances, however, we might wish to define a different action for an assignment involving an object of derived type, and indeed this is possible. An assignment may be redefined or another assignment may be defined by a subroutine with two arguments, the first having intent `out` or intent `inout` and corresponding to the variable and the second having intent `in` and corresponding

to the expression (subroutines will also be dealt with fully in Chapter 5). In the case of an assignment involving an object of derived type and an object of a different type, such a definition must be provided. For example, assignment of reals to intervals and vice versa might be defined by a module containing the subroutines

```
subroutine real_from_interval(a,b)
   real, intent(out)          :: a
   type(interval), intent(in) :: b
   a = (b%lower + b%upper)/2
end subroutine real_from_interval
```

and

```
subroutine interval_from_real(a,b)
   type(interval), intent(out) :: a
   real, intent(in)            :: b
   a%lower = b
   a%upper = b
end subroutine interval_from_real
```

and the interface block

```
interface assignment(=)
   module procedure real_from_interval, interval_from_real
end interface
```

Given this, we may write

```
type(interval) :: a
a = 0.0
```

A defined assignment must not redefine the meaning of an intrinsic assignment for intrinsic types, that is an assignment between two objects of numeric type, of logical type, or of character type with the same kind parameter, but may redefine the meaning of an intrinsic assignment for two objects of the same derived type. For instance, for an assignment between two variables of the type `string` (Section 3.8) that copies only the relevant part of the character component, we might write

```
subroutine assign_string (left, right)
   type(string), intent(out) :: left
   type(string), intent(in)  :: right
   left%length = right%length
   left%value(1:left%length) = right%value(1:right%length)
end subroutine assign_string
```

Intrinsic assignment for a derived-type object always involves intrinsic assignment for all its non-pointer components, even if a component is of a derived type for which assignment has been redefined.

3.10 Array expressions

So far in this chapter, we have assumed that all the entities in an expression are scalar. However, any of the unary intrinsic operations may also be applied to an array to produce another array of the same shape (identical rank and extents, see Section 2.10) and having each element value equal to that of the operation applied to the corresponding element of the operand. Similarly, binary intrinsic operations may be applied to a pair of arrays of the same shape to produce an array of that shape, with each element value equal to that of the operation applied to corresponding elements of the operands. One of the operands to a binary operation may be a scalar, in which case the result is as if the scalar had been broadcast to an array of the same shape as the array operand. Given the array declarations

```
real, dimension(10,20) :: a,b
real, dimension(5)     :: v
```

the following are examples of array expressions:

```
a/b       ! Array of shape (10,20), with elements a(i,j)/b(i,j)
v+1.      ! Array of shape (5), with elements v(i)+1.0
5/v+a(1:5,5) ! Array of shape (5), with elements 5/v(i)+a(i,5)
a.eq.b    ! Logical array of shape (10,20), with elements
          ! .true. if a(i,j).eq.b(i,j), and .false. otherwise
```

Two arrays of the same shape are said to be *conformable* and a scalar is conformable with any array.

Note that the correspondence is by position in the extent and not by subscript value. For example,

```
a(2:9,5:10) + b(1:8,15:20)
```

has element values

```
a(i+1,j+4) + b(i,j+14), i=1,2,...,8, j=1,2,...,6.
```

This may be represented pictorially as in Figure 3.1.

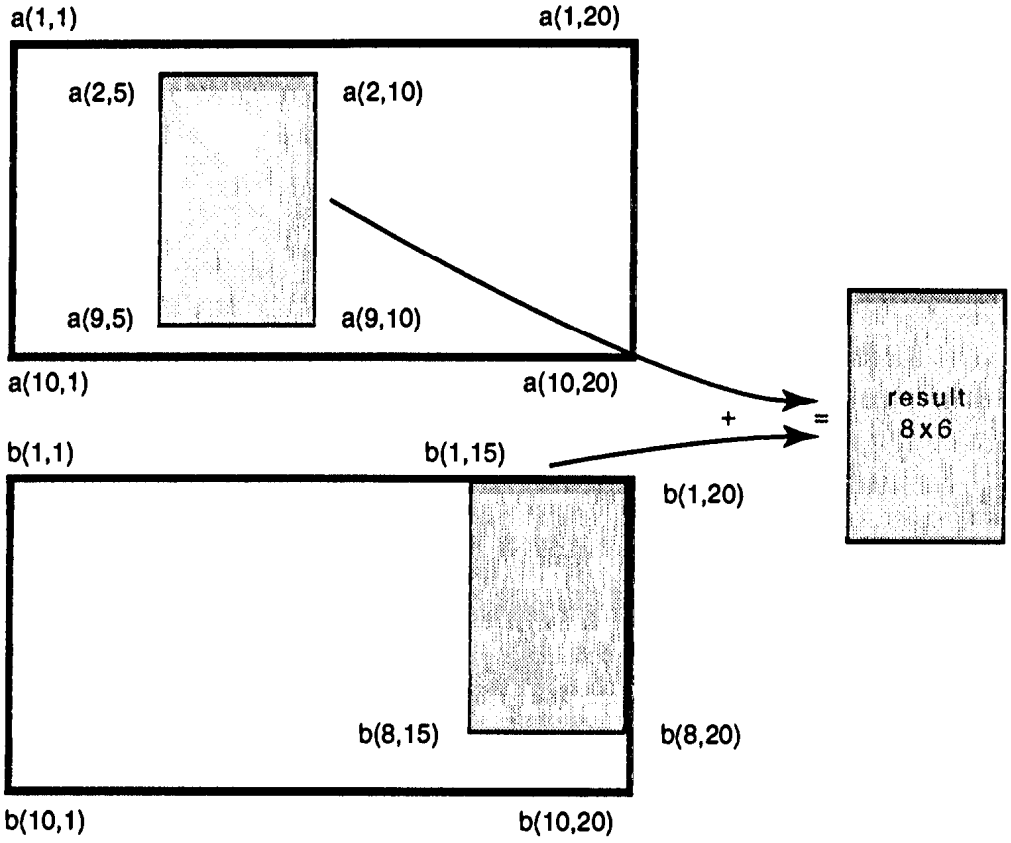
The order in which the scalar operations in any array expression are executed is not specified in the standard, thus enabling a compiler to arrange efficient execution on a vector or parallel computer.

Any scalar intrinsic operator may be applied in this way to arrays and array-scalar pairs. For derived operators, the programmer must define operators directly for array operands, for each rank or pair of ranks involved. For example, the type

```
type matrix
   real :: element
end type matrix
```

might be defined to have scalar operations that are identical to the operations for reals, but for arrays of ranks one and two the operator * defined to mean

Figure 3.1 The sum of two array sections.



matrix multiplication. The type `matrix` would therefore be suitable for matrix arithmetic, whereas reals are not suitable because multiplication for real arrays is done element by element. This is further discussed in Section 6.7.

3.11 Array assignment

By intrinsic assignment, an array expression may be assigned to an array variable of the same shape, which is interpreted as if each element of the expression were assigned to the corresponding element of the variable. For example, with the declarations of the beginning of the last section, the assignment

```
a = a + 1.0
```

replaces `a(i,j)` by `a(i,j)+1.0` for `i` = 1, 2, . . . , 10 and `j` = 1, 2, . . . , 20. Note that, as for expressions, the element correspondence is by position within the extent rather than by subscript value. This is illustrated by the example

```
a(1,11:15) = v      ! a(1,j+10) is assigned from
                    ! v(j), j=1,2,...,5
```

A scalar expression may be assigned to an array, in which case the scalar value is broadcast to all the array elements.

If the expression includes a reference to the array variable or to a part of it, the expression is interpreted as being fully evaluated before the assignment commences. For example, the statement

```
v(2:5) = v(1:4)
```

results in each element `v(i)` for `i` = 2, 3, 4, 5 having the value that `v(i-1)` had prior to the commencement of the assignment. This rule exactly parallels the rule for substrings that was explained in Section 3.7. The order in which the array elements are assigned is not specified by the standard, to allow optimizations.

Sets of numeric and mathematical intrinsic functions, whose results may be used as operands in scalar or array expressions and in assignments, are described in Sections 8.3 and 8.4.

For a defined assignment (Section 3.9), a separate subroutine must be provided for each combination of ranks for which it is required. Intrinsic assignment is overridden only for those combinations of ranks for which a corresponding defined assignment is accessible.

A form of array assignment expressed with the help of indices is provided by Fortran 95 (Section 6.9). Also, elemental defined assignments are available in Fortran 95 (Section 6.11).

3.12 Pointers in expressions and assignments

A pointer may appear as a variable in the expressions and assignments that we have considered so far in this chapter, provided it has a valid association with a target. The target is accessed without any need for an explicit dereferencing symbol. In particular, if two pointers appear on opposite sides of an assignment statement, data are copied from one target to the other target.

Sometimes the need arises for another sort of assignment. We may want the left-hand pointer to point to another target, rather than that its current target acquire fresh data. That is, we want the descriptor to be altered. This is called *pointer assignment* and takes place in a pointer assignment statement:

pointer `=>` *target*

where *pointer* is the name of a pointer or the designator of a structure component that is a pointer, and *target* is usually a variable but may also be a reference to a pointer-valued function (see Section 5.10). For example, the statements

```
x => z
a => c
```

have variables as targets and are needed for the first matrix multiplication of Section 2.13, in order to make `x` refer to `z` and `a` to refer to `c`. In Fortran 95, the statement

```
x => null()     ! Fortran 95 only
```

nullifies x. Pointer assignment also takes place for a pointer component of a structure when the structure appears on the left-hand side of an ordinary assignment. For example, suppose we have used the type entry of Section 2.13 to construct a chain of entries and wish to add a fresh entry at the front. If first points to the first entry and current is a scalar pointer of type entry, the statements

```
allocate (current)
current = entry(new_value, new_index, first)
first => current
```

allocate a new entry and link it into the top of the chain. The assignment statement has the effect

```
current%next => first
```

and establishes the link. The pointer assignment statement gives first the new entry as its target without altering the old first entry. An ordinary assignment would be incorrect because the target would be copied, destroying the old first entry, corresponding to the component assignments

```
first%value = current%value
first%index = current%index
first%next => current%next
```

In the case where the chain began with length two and consisted of

```
first :      (1.0, 10, associated)
first%next : (2.0, 15, null)
```

following the execution of the first set of statements it would have length 3 and consist of

```
first :           (4.0, 16, associated)
first%next :      (1.0, 10, associated)
first%next%next : (2.0, 15, null)
```

If the *target* in a pointer assignment statement is a variable that is not itself a pointer or a subobject of a pointer, it must have the target attribute. For example, the statement

```
real, dimension(10), target :: y
```

declares y to have the target attribute. Any subobject of an object with the target attribute also has the target attribute. The target attribute is required for the purpose of code optimization by the compiler. It is very helpful to the compiler to know that a variable that is not a pointer or a target may not be accessed by a pointer.

The target in a pointer assignment statement may be a subobject of a pointer. For example, given the declaration

```
character(len=80), dimension(:), pointer :: page
```

and an appropriate association, the following are all permitted targets:

```
page, page(10), page(2:4), page(2)(3:15)
```

Note that it is sufficient for any part of the subobject to have the `pointer` attribute. For example, given the declaration

```
type(entry) :: node ! This has a pointer component next,
                    ! see Section 2.13.
```

and an appropriate association, `node%next%value` is a permitted target.

If the *target* in a pointer assignment statement is itself a pointer, then a straightforward copy of the descriptor takes place. If the pointer association status is undefined or disassociated, this state is copied.

If the *target* is a pointer or a subobject of a pointer, the new association is with that pointer's target and is not affected by any subsequent changes to its pointer association status. This is illustrated by the following example. The sequence

```
b => c    ! c has the target attribute
a => b
nullify (b)
```

will leave `a` still pointing to `c`.

The type, type parameters, and rank of the *pointer* and *target* in a pointer assignment statement must each be the same. If the *pointer* is an array, it takes its shape and bounds from the *target*. The bounds are as would be returned by the functions `lbound` and `ubound` (Section 8.12.2) for the target, which means that an array section or array expression is always taken to have the value 1 for a lower bound and the extent for the corresponding upper bound.

Fortran is unusual in not requiring a special character for a reference to a pointer target, but requiring one for distinguishing pointer assignment from ordinary assignment. The reason for this choice was the expectation that most engineering and scientific programs will refer to target data far more often than they change targets.

3.13 Summary

In this chapter, we have seen how scalar and array expressions of numeric, logical, character, and derived types may be formed, and how the corresponding assignments of the results may be made. The relational expressions and the use of pointers have also been presented. We now have the information required to write short sections of code forming a sequence of statements to be performed one after the other. In the following chapter we shall see how more complicated sequences, involving branching and iteration, may be built up.

Features described in this chapter which are new since Fortran 77 are the use of the alternative representations <, <=, ... for the relational operators; the ability

of the two sides of a character assignment to overlap; structure constructors; defined operators and assignment; array expressions and assignment; and the use of pointers in expressions and assignment.

3.14 Exercises

1. If all the variables are numeric scalars, which of the following are valid numeric expressions?

```
a+b              -c
a+-c             d+(-f)
(a+c)**(p+q)     (a+c)(p+q)
-(x+y)**i        4.((a-d)-(a+4.*x)+1)
```

2. In the following expressions, add the parentheses which correspond to Fortran's rules of precedence (assuming `a`, `c-f` are real scalars, `i-n` are logical scalars, and `b` is a logical array), for example

```
a+d**2/c      becomes      a+((d**2)/c)
```

```
c+4.*f
4.*g-a+d/2.
a**e**c**d
a*e-c**d/a+e
i .and. j .or. k
.not. l .or. .not. i .and. m .neqv. n
b(3).and.b(1).or.b(6).or..not.b(2)
```

3. What are the results of the following expressions?

```
3+4/2        6/4/2
3.*4**2      3.**3/2
-1.**2       (-1.)**3
```

4. A scalar character variable `r` has length 8. What are the contents of `r` after each of the following assignments?

```
r = 'ABCDEFGH'
r = 'ABCD'//'01234'
r(:7) = 'ABCDEFGH'
r(:6) = 'ABCD'
```

5. Which of the following logical expressions are valid if `b` is a logical array?

```
.not.b(1).and.b(2)       .or.b(1)
b(1).or..not.b(4)        b(2)(.and.b(3).or.b(4))
```

6. If all the variables are real scalars, which of the following relational expressions are valid?

```
d .le. c            p .lt. t > 0
x-1 /= y            x+y < 3 .or. > 4.
d.lt.c.and.3.0      q.eq.r .and. s>t
```

7. Write expressions to compute:

a) the perimeter of a square of side l;

b) the area of a triangle of base b and height h;

c) the volume of a sphere of radius r.

8. An item costs n cents. Write a declaration statement for suitable variables and assignment statements which compute the change to be given from a $1 bill for any value of n from 1 to 99, using coins of denomination 1, 5, 10, and 25 cents.

9. Given the type declaration for `interval` in Section 3.8, the definitions of + given in Section 3.8, the definitions of assignment given in Section 3.9, and the declarations

```
type(interval) :: a,b,c,d
real           :: r
```

which of the following statements are valid?

```
a = b + c
c = b + 1.0
d = b + 1
r = b + c
a = r + 2
```

10. Given the type declarations

```
real, dimension(5,6) :: a, b
real, dimension(5)   :: c
```

which of the following statements are valid?

```
a = b               c = a(:,2) + b(5,:5)
a = c+1.0           c = a(2,:) + b(:,5)
a(:,3) = c          b(2:,3) = c + b(:5,3)
```

4. Control constructs

4.1 Introduction

We have learnt in the previous chapter how assignment statements may be written, and how these may be ordered one after the other to form a sequence of code which is executed step-by-step. In most computations, however, this simple sequence of statements is by itself inadequate for the formulation of the problem. For instance, we may wish to follow one of two possible paths through a section of code, depending on whether a calculated value is positive or negative. We may wish to sum 1000 elements of an array, and to do this by writing 1000 additions and assignments is clearly tedious; the ability to iterate over a single addition is required instead. We may wish to pass control from one part of a program to another, or even stop processing altogether.

For all these purposes, we have available in Fortran various facilities to enable the logical flow through the program statements to be controlled. The facilities contained in Fortran correspond to those now widely regarded as being the most appropriate for a modern programming language. Their general form is that of a *block* construct, that is a construct which begins with an initial keyword statement, may have intermediate keyword statements, and ends with a matching terminal statement, and that may be entered only at the initial statement. Each sequence of statements between keywords is called a *block*. A block may be empty, though such cases are rare.

Block constructs may be *nested*, that is a block may contain another block construct. In such a case, the block must contain the whole of the inner construct. Execution of a block always begins with its first statement.

However, we begin by describing the simple `go to` statement.

4.2 The go to statement

In this section, we consider the most disputed statement in programming languages – the `go to` statement. It is generally accepted that it is difficult to understand a program which is interrupted by many branches, especially if there is a large number of backward branches – those returning control to a statement

preceding the branch itself. At the same time there are certain occasions, especially when dealing with error conditions, when go to statements are required in even the most advanced languages.

The form of the unconditional go to statement is

```
go to label
```

where *label* is a statement label. This statement label must be present on an *executable statement* (a statement which can be executed, as opposed to one of an informative nature, like a declaration). An example is

```
   x = y+3.0
   go to 4
3  x = x+2.0
4  z = x+y
```

in which we note that after execution of the first statement, a branch is taken to the last statement, labelled 4. This is a *branch target statement.* The statement labelled 3 is jumped over, and can be executed only if there is a branch to the label 3 somewhere else. If the statement following an unconditional go to is unlabelled it can never be reached and executed, creating *dead code,* normally a sign of incorrect coding.

A go to statement must never specify a branch into a block, though it may specify a branch

- from within a block to another statement in the block,
- to the terminal statement of its construct, or
- to a statement outside its construct.

4.3 The if statement and construct

The if statement and if construct provide a mechanism for branching depending on a condition. They are powerful tools, the if construct being a generalized form of the if statement.

4.3.1 The if statement

In the if statement, the value of a scalar logical expression is tested, and a single statement executed if its value is true. The general form is

```
if (scalar-logical-expr) action-stmt
```

where *scalar-logical-expr* is any scalar logical expression, and *action-stmt* is any executable statement other than one that marks the beginning or end of a block (for instance, if, else if, else, end if, see next subsection), another if statement, or an end statement (see Chapter 5). Examples are

```
if (flag) go to 6
if (x-y > 0.0) x = 0.0
if (cond .or. p<q .and. r<=1.0) s(i,j) = t(j,i)
```

The `if` statement is normally used either to perform a single assignment depending on a condition, or to branch depending on a condition. The *action-stmt* may not be labelled separately.

4.3.2 The if construct

The `if` construct allows either the execution of a sequence of statements (a block) to depend on a condition, or the execution of alternative sequences of statements (blocks) to depend on alternative conditions. The simplest of its three forms is

```
[name:] if (scalar-logical-expr) then
          block
        end if [name]
```

where *scalar-logical-expr* is any scalar logical expression and *block* is any sequence of executable statements (except an `end` statement or an incomplete construct). The *block* is executed if *scalar-logical-expr* evaluates to the value true, and is not executed if it evaluates to the value false. The `if` construct may be optionally named: the first and last statements may bear the same name, which may be any valid and distinct Fortran name (see Section 5.15 for a discussion on the scope of names). The fact that the name is optional is indicated here by the square brackets, a convention that will be followed throughout the book.

We notice that the `if` construct is a compound statement, the beginning being marked by the `if...then`, and the end by the `end if`. An example is

```
swap: if (x < y) then
        temp = x
        x = y
        y = temp
      end if swap
```

in which we notice also that the block inside the `if` construct is indented with respect to its beginning and end. This is not obligatory, but makes the logic easier to understand, especially in nested `if` constructs as we shall see at the end of this section.

In the second form of the `if` construct, an alternative block of statements is executable, for the case where the condition is false. The general form is

```
[name:]  if (scalar-logical-expr) then
           block1
         else [name]
           block2
         end if [name]
```

in which the first block of statements (*block1*) is executed if the condition is true and the second block (*block2*), following the `else` statement, is executed if the condition is false. An example is

```
if (x < y) then
   x = -x
else
   y = -y
end if
```

in which the sign of `x` is changed if `x` is less than `y`, and the sign of `y` is changed if `x` is greater than or equal to `y`.

The third and most general type of `if` construct uses the `else if` statement to make a succession of tests, each of which has its associated block of statements. The tests are made one after the other until one is fulfilled, and the associated statements of the relevant `if` or `else if` block are executed. Control then passes to the end of the `if` construct. If no test is fulfilled, no block is executed, unless there is a final 'catch-all' `else` clause. The general form is shown in Figure 4.1. Here, and later in the book, we use the notation [ ] to indicate an optional item and [ ]... to indicate an item that may occur any number of times (including zero). There can be any number (including zero) of `else if` statements, and zero or one `else` statements. An `else` or `else if` statement may be named only if the corresponding `if` and `end if` statements are named, and must be given the same name.

Figure 4.1

```
[name:] if (scalar-logical-expr) then
            block
        [else if (scalar-logical-expr) then [name]
            block]...
        [else [name]
            block]
        end if [name]
```

The statements within an `if` construct may be labelled, but the labels must never be referenced in such a fashion as to pass control into the range of an `if` construct from outside it, to an `else if` or `else` statement, or into a block of the construct from outside the block. For example, the following `if` construct is illegal:

```
   if (temp > 100.0) then
      go to 1                          ! illegal branch
      boil = .true.
      steam = .true.
   else
1     boil = .false.
      liquid = .true.
   end if
```

It is permitted to pass control to an end `if` statement from within its construct. execution of an end `if` statement has no effect.

It is permitted to nest `if` constructs within one another to an arbitrary depth, as shown to two levels in Figure 4.2, in which we see again the necessity to indent the code in order to be able to understand the logic easily. For even deeper nesting, naming is to be recommended. The constructs must be properly nested, that is each construct must be wholly contained in a block of the next outer construct.

Figure 4.2

```
if (i < 0) then
   if (j < 0) then
      x = 0.0
      y = 0.0
   else
      z = 0.0
   end if
else if (k < 0) then
   z = 1.0
else
   x = 1.0
   y = 1.0
end if
```

4.4 The case construct

Fortran provides another means of selecting one of several options, rather similar to that of the `if` construct. The principal differences between the two constructs are that, for the `case` construct, only *one* expression is evaluated for testing, and the evaluated expression may belong to no more than one of a series of pre-defined sets of values. The form of the `case` construct is shown by:

```
[name:]  select case (expr)
           [case selector [name]
              block]...
           end select [name]
```

As for the `if` construct, the leading and trailing statements must either both be unnamed or both bear the same name; an intermediate statement may be named only if the leading statement is named and bears the same name. The expression *expr* must be scalar and of type character, logical, or integer, and the specified values in each *selector* must be of this type. In the character case, the lengths are permitted to differ, but not the kinds. In the logical and integer cases, the kinds

may differ. The simplest form of *selector* is a scalar initialization expression[1] in parentheses, such as in the statement

```
case(1)
```

For character or integer *expr*, a range may be specified by a lower and an upper scalar initialization expression separated by a colon:

```
case (low:high)
```

Either *low* or *high*, but not both, may be absent; this is equivalent to specifying that the case is selected whenever *expr* evaluates to a value that is less than or equal to *high*, or greater than or equal to *low*, respectively. An example is shown in Figure 4.3.

Figure 4.3

```
select case (number)        ! number is of type integer
case (:-1)                  ! all values below 0
   n_sign = -1
case (0)                    ! only 0
   n_sign = 0
case (1:)                   ! all values above 0
   n_sign = 1
end select
```

The general form of *selector* is a list of non-overlapping values and ranges, all of the same type as *expr*, enclosed in parentheses, such as

```
case (1, 2, 7, 10:17, 23)
```

The form

```
case default
```

is equivalent to a list of all the possible values of *expr* that are not included in the other selectors of the construct. Though we recommend that the values be in order, as in this example, this is not required. Overlapping values are not permitted within one *selector*, nor between different ones in the same construct.

There may be only a single `case default` *selector* in a given `case` construct as shown in Figure 4.4. The `case default` clause does not necessarily have to be the last clause of the `case` construct.

Since the values of the selectors are not permitted to overlap, at most one selector may be satisfied; if none is satisfied, control passes to the next executable statement following the `end select` statement.

[1] An initialization expression is a restricted form of constant expression (the restrictions being chosen for ease of implementation). The details are tedious and are deferred to Section 7.4. In this section, all examples employ the simplest form of initialization expression: the literal constant.

Figure 4.4

```
select case (ch)          ! ch of type character
case ('c', 'd', 'r':)
   ch_type = .true.
case ('i':'n')
   int_type = .true.
case default
   real_type = .true.
end select
```

Like the `if` construct, `case` constructs may be nested inside one another. Branching to a statement in a `case` block is permitted only from another statement in the block, it is not permitted to branch to a `case` statement, and any branch to an `end select` statement must be from within the `case` construct which it terminates.

4.5 The do construct

Many problems in mathematics require, for their representation in a programming language, the ability to *iterate*. If we wish to sum the elements of an array `a` of length 10, we could write

```
sum = a(1)
sum = sum+a(2)
:
sum = sum+a(10)
```

which is clearly laborious. Fortran provides a facility known as the `do` construct which allows us to reduce these ten lines of code to

```
sum = 0.0
do  i = 1,10 ! i is of type integer
   sum = sum+a(i)
end do
```

In this fragment of code we first set `sum` to zero, and then require that the statement between the `do` statement and the `end do` statement shall be executed ten times. For each iteration there is an associated value of an index, kept in `i`, which assumes the value 1 for the first iteration through the loop, 2 for the second, and so on up to 10. The variable `i` is a normal integer variable, but is subject to the rule that it must not be explicitly modified within the `do` construct.

The `do` statement has more general forms. If we wished to sum the fourth to ninth elements we would write

```
do  i = 4, 9
```

thereby specifying the required first and last values of `i`. If, alternatively, we wished to sum all the odd elements, we would write

```
do  i = 1, 9, 2
```

where the third of the three loop *parameters*, namely the 2, specifies that `i` is to be incremented in steps of 2, rather than by the default value of 1, which is assumed if no third parameter is given. In fact, we can go further still, as the parameters need not be constants at all, but integer expressions, as in

```
do  i = j+4, m, -k(j)**2
```

in which the first value of `i` is `j+4`, and subsequent values are decremented by `k(j)**2` until the value of `m` is reached. Thus, `do` constructs may run 'backwards' as well as 'forwards'. If any of the three parameters is a variable or is an expression that involves a variable, the value of the variable may be modified within the loop without affecting the number of iterations, as the *initial* values of the parameters are used for the control of the loop.

The general form of this type of bounded `do` construct control clause is

```
[name:]  do [,] variable = expr1, expr2 [,expr3]
             block
         end do [name]
```

where *variable* is a named scalar integer variable, *expr1*, *expr2*, and *expr3* (*expr3* is optional but must be nonzero when present) are any valid scalar integer expressions, and *name* is the optional construct name. The `do` and `end do` statements must either both bear the same *name*, or both be unnamed.

The number of iterations of a `do` construct is given by the formula

```
max((expr2-expr1+expr3)/expr3, 0)
```

where `max` is a function which we shall meet in Section 8.3.2 and which returns either the value of the expression or zero, whichever is the larger. There is a consequence following from this definition, namely that if a loop begins with the statement

```
do  i = 1, n
```

then its body will not be executed at all if the value of `n` on entering the loop is zero or less. This is an example of the *zero-trip loop*, and results from the application of the `max` function.

A very simple form of the `do` statement is the unbounded

```
[name:] do
```

which specifies an endless loop. In practice, a means to exit from an endless loop is required, and this is provided in the form of the `exit` statement:

```
exit [name]
```

where *name* is optional and is used to specify from which construct the exit should be taken in the case of nested constructs. Execution of an `exit` statement causes control to be transferred to the next executable statement after the end do statement to which it refers. If no name is specified, it terminates execution of the innermost do construct in which it is enclosed. As an example of this form of the do, suppose we have used the type `entry` of Section 2.13 to construct a chain of entries in a sparse vector, and we wish to find the entry with index 10, known to be present. If `first` points to the first entry, the code in Figure 4.5 is suitable.

Figure 4.5

```
type(entry), pointer :: first, current
:
current => first
do
   if (current%index == 10) exit
   current => current%next
end do
```

The `exit` statement is also useful in a bounded loop when all iterations are not always needed.

A related statement is the `cycle` statement

```
cycle [name]
```

which transfers control to the end do statement of the corresponding construct. Thus, if further iterations are still to be carried out, the next one is initiated.

The value of a do construct index (if present) is incremented at the end of every loop iteration for use in the subsequent iteration. As the value of this index is available outside the loop after its execution, we have three possible situations, each illustrated by the following loop:

```
do  i = 1, n
   :
   if (i==j) exit
   :
end do
l = i
```

The situations are:

i) If, at execution time, `n` has the value zero or less, `i` is set to 1 but the loop is not executed, and control passes to the statement following the end do statement.

ii) If `n` has a value which is greater than or equal to `j`, an exit will be taken at the `if` statement, and `l` will acquire the last value of `i`, which is of course `j`.

iii) If the value of n is greater than zero but less than j, the loop will be executed n times, with the successive values of i being 1, 2, ... *etc.* up to n. When reaching the end of the loop for the n*th* time, i will be incremented a final time, acquiring the value n+1, which will then be assigned to l.

We see how important it is to make careful use of loop indices outside the do block, especially when there is the possibility of the number of iterations taking on the boundary value of the maximum for the loop.

The do block, just mentioned, is the sequence of statements between the do statement and the end do statement. From anywhere outside a do block, it is prohibited to jump into the block or to its end do statement. The following sequence is thus illegal:

```
      go to 2          ! illegal branch
      :
      do i = 1, n
         :
   2     a = b + c
         :
      end do
```

It is similarly illegal for the block of a do construct (or an if, case, or where construct, see Section 6.8), to be only partially contained in a block of another construct. The construct must be completely contained in the block. The following two sequences are thus legal:

```
      if (scalar-logical-expr) then
         do i = 1, n
            :
         end do
      else
         :
      end if
```

and

```
      do i = 1, n
         if (scalar-logical-expr) then
            :
         end if
      end do
```

but this third sequence is not:

```
      if (scalar-logical-expr) then
         do  i = 1, 10
            :
         end if  ! illegal position of if construct termination
         :
      end do
```

Any number of do constructs may be nested provided that the range of each nested loop is completely contained within the range of another. We may thus write a matrix multiplication as shown in Figure 4.6.

Figure 4.6

```
do  i = 1, n
   do  j = 1, m
      a(i,j) = 0.0
      do  l = 1, k
         a(i,j) = a(i,j)+b(i,l)*c(l,j)
      end do
   end do
end do
```

Another example is the summation loop in Figure 4.7.

Figure 4.7

```
do  i = 1, n
   sum = 0.0
   do  j = 1,i
      sum = sum+b(j,i)
   end do
   a(i) = sum
end do
```

A final form of the do construct makes use of a statement label to identify the end of the construct. In this case, the terminating statement may be either a labelled end do statement or a labelled `continue` ('do nothing') statement[2]. The label is, in each case, the same as that on the do statement itself. Simple examples are

```
do 10 i = 1, n
   :
10 end do
```

and

```
do 20 i = 1, j
   do 10 k = 1, l
      :
10    continue
20 continue
```

[2]The `continue` statement is not limited to being the last statement of a do construct; it may appear anywhere among the executable statements.

As shown in the second example, each loop must have a separate label. Additional, but redundant, do syntax is described in Section 11.3.2 (and Appendix C.2.2 and C.3.1). The full do construct syntax is given in Appendix B.

Finally, it should be noted that many short do-loops can be expressed alternatively in the form of array expressions and assignments. However, this is not always possible, and a particular danger to watch for is where one iteration of the loop depends upon a previous one. Thus, the loop

```
do i = 2, n
   a(i) = a(i-1) + b(i)
end do
```

cannot be replaced by the statement

```
a(2:n) = a(1:n-1) + b(2:n)        ! Beware
```

4.6 Summary

In this chapter we have introduced the four main features by which the control in Fortran code may be programmed – the go to statement, the if statement and construct, the case construct and the do construct. The effective use of these features is the key to sound code. Of these features, the case construct is new to Fortran, and the do construct was formerly limited to the labelled form ending on a continue statement.

We have touched upon the concept of a *program unit* as being like the chapter of a book. Just as a book may have just one chapter, so a complete program may consist of just one program unit, which is known as a *main program.* In its simplest form it consists of a series of statements of the kinds we have been dealing with so far, and terminates with an end statement, which acts as a signal to the computer to stop processing the current program. In order to test whether a program unit of this type works correctly, we need to be able to output, to a terminal or printer, the values of the computed quantities. This topic will be fully explained in Chapter 9, and for the moment we need to know only that this can be achieved by a statement of the form

```
print * , ' var1 = ', var1 , ' var2 = ', var2
```

which will output a line such as

```
var1 = 1.0  var2 = 2.0
```

Similarly, input data can be read by statements like

```
read *, val1, val2
```

Figure 4.8

```
!   Print a conversion table of the Fahrenheit and Celsius
!   temperature scales between specified limits.
!
   real      :: celsius, fahrenheit
   integer   :: low_temp, high_temp, temperature
   character :: scale
!
read_loop: do
!
!   Read scale and limits
      read *, scale, low_temp, high_temp
!
!   Check for valid data
      if (scale /= 'C' .and. scale /= 'F') exit read_loop
!
!   Loop over the limits
      do  temperature = low_temp, high_temp
!
!   Choose conversion formula
         select case (scale)
         case ('C')
            celsius = temperature
            fahrenheit = 9.0/5.0*celsius + 32.0
         case ('F')
            fahrenheit = temperature
            celsius = 5.0/9.0*(fahrenheit-32.0)
         end select
!
!   Print table
         print *, celsius, ' degrees C correspond to',  &
                  fahrenheit, ' degrees F'
      end do
   end do read_loop
!
!   Termination
print *, ' End of valid data'
   end
C  90   100
F  20   32
*  0   0
```

This is sufficient to allow us to write simple programs like that in Figure 4.8, which outputs the converted values of a temperature scale between specified limits. Valid inputs are shown at the end of the example.

4.7 Exercises

1. Write a program which

a) defines an array to have 100 elements;

b) assigns to the elements the values 1, 2, 3, . . . , 100;

c) reads two integer values in the range 1 to 100;

d) reverses the order of the elements of the array in the range specified by the two values.

2. The first two terms of the Fibonacci series are both 1, and all subsequent terms are defined as the sum of the preceding two terms. Write a program which reads an integer value `limit` and which computes and prints the coefficients of the first `limit` terms of the series.

3. The coefficients of successive orders of the binomial expansion are shown in the normal Pascal triangle form as

```
      1
    1 1
  1 2 1
1 3 3 1
1 4 6 4 1
  etc.
```

Write a program which reads an integer value `limit` and prints the coefficients of the first `limit` lines of this Pascal triangle.

4. Define a character variable of length 80. Write a program which reads a value for this variable. Assuming that each character in the variable is alphabetic, write code which sorts them into alphabetic order, and prints out the frequency of occurrence of each letter.

5. Write a program to read an integer value `limit` and print the first `limit` prime numbers, by any method.

6. Write a program which reads a value `x`, and calculates and prints the corresponding value `x/(1.+x)`. The case `x=-1.` should produce an error message and be followed by an attempt to read a new value of `x`.

7. Given a chain of entries of the type `entry` of Section 2.13, modify the code in Figure 4.5 (Section 4.5) so that it removes the entry with index 10, and makes the previous entry point to the following entry.

5. Program units and procedures

5.1 Introduction

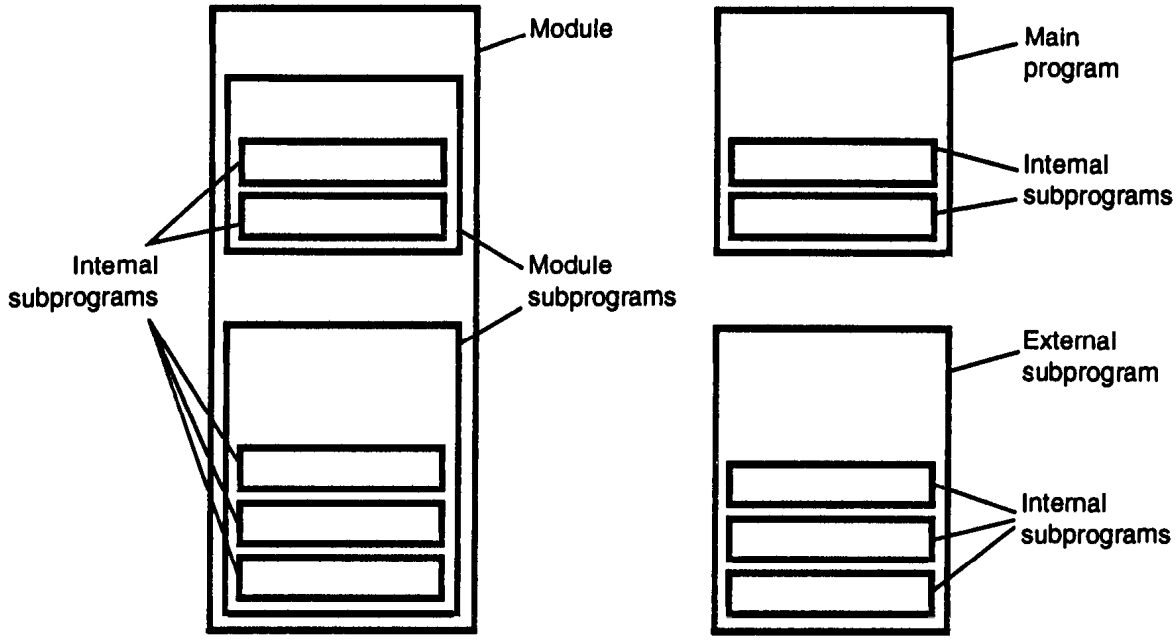
As we saw in the previous chapter, it is possible to write a complete Fortran program as a single unit, but it is preferable to break the program down into manageable units. Each such *program unit* corresponds to a program task that can be readily understood and, ideally, can be written, compiled, and tested in isolation. We will discuss the three kinds of program unit, the main program, external subprogram, and module.

A complete program must, as a minimum, include one *main program.* This may contain statements of the kinds that we have met so far in examples, but normally its most important statements are invocations or *calls* to subsidiary programs known as *subprograms.* A subprogram defines a *function* or a *subroutine*[1]. They differ in that a function returns a single object and usually does not alter the values of its arguments (so that it represents a function in the mathematical sense), whereas a subroutine usually performs a more complicated task, returning several results through its arguments and by other means. Functions and subroutines are known collectively as *procedures.*

There are various kinds of subprograms. A subprogram may be a program unit in its own right, in which case it is called an *external subprogram* and defines an *external procedure.* External procedures may also be defined by means other than Fortran (usually assembly language). A subprogram may be a member of a collection in a program unit called a *module*, in which case we call it a *module subprogram* and it defines a *module procedure.* A subprogram may be placed inside a module subprogram, an external subprogram, or a main program, in which case we call it an *internal subprogram* and it defines an *internal procedure.* Internal subprograms may not be nested, that is they may not contain further subprograms, and we expect them normally to be short sequences of code, say up to about twenty lines. We illustrate the nesting of subprograms in program units in Figure 5.1. If a program unit or subprogram contains a subprogram, it is called the *host* of that subprogram.

[1]It is possible to write a subprogram that defines more than one function or more than one subroutine (see Section 11.2.6), but we do not recommend this practice.

Figure 5.1 Nesting of subprograms in program units.



Besides containing a collection of subprograms, a module may contain data definitions, derived type definitions, interface blocks (Section 5.11), and namelist groups (Section 7.15). This collection may be expected to provide facilities associated with some particular task, such as providing matrix arithmetic, a library facility, or a data base. It may sometimes be large.

In this chapter, we will describe program units and the statements that are associated with them. Within a complete program, they may appear in any order, but many compilers require a module to precede other program units that use it.

5.2 Main program

Every complete program must have one, and only one, main program. Optionally, it may contain calls to subprograms. A main program has the following form:

```
[program program-name]
    [specification-stmts]
    [executable-stmts]
[contains
    internal-subprograms]
end [program [program-name]]
```

The `program` statement is optional, but we recommend its use. The *program-name* may be any valid Fortran name such as `model`. The only non-optional statement is the `end` statement which has two purposes. It acts as a signal to the compiler that it has reached the end of the program unit and, when executed, it causes the complete program to stop. If it includes *program-name*, this must be the name on

the program statement. We recommend using the full form so that it is clear both to the reader and to the compiler exactly what is terminated by the end statement.

A main program without calls to subprograms is usually used only for short tests, as in

```
program test
   print *, 'Hello world!'
end program test
```

The specification statements define the environment for the executable statements. So far, we have met the type declaration statement (integer, real, complex, logical, character, and type(*type-name*)) that specifies the type and other properties of the entities that it lists, and the type definition block (bounded by type *type-name* and end type statements). We will meet other specification statements in this and the next two chapters.

The executable statements specify the actions that are to be performed. So far, we have met the assignment statement, the pointer assignment statement, the go to statement, the if statement and construct, the do and case constructs, and the read and print statements. We will meet other executable statements in this and later chapters. Execution of a program always commences with the first executable statement of the main program.

The contains statement flags the presence of one or more internal subprograms. We will describe internal subprograms in Section 5.6. If the execution of the last statement ahead of the contains statement does not result in a branch, control passes over the internal subprograms to the end statement and the program stops. The end statement may be labelled and may be the target of a branch from one of the executable statements. If such a branch is taken, again the program stops.

5.3 The stop statement

Another way to stop program execution is to execute a stop statement. This statement may be labelled, may be part of an if statement, and is an executable statement that may appear in the main program or any subprogram. A well-designed program normally returns control to the main program for program termination, so the stop statement should appear there. However, in applications where several stop statements appear in various places in a complete program, it is possible to distinguish which of the stop statements has caused the termination by adding to each one an *access code* consisting of a default character constant or a string of up to five digits whose leading zeros are not significant. This might be used by a given processor to indicate the origin of the stop in a message. Examples are

```
stop
stop 'Incomplete data. Program terminated.'
stop 12345
```

5.4 External subprograms

External subprograms are called from a main program or elsewhere, usually to perform a well-defined task within the framework of a complete program. Apart from the leading statement, they have a form that is very like that of a main program:

```
 subroutine-stmt
    [specification-stmts]
    [executable-stmts]
 [contains
     internal-subprograms]
 end [subroutine [subroutine-name]]
```

or

```
 function-stmt
    [specification-stmts]
    [executable-stmts]
 [contains
     internal-subprograms]
 end [function [function-name]]
```

The `contains` statement plays exactly the same role as within a main program (see Section 5.2). The effect of executing an `end` statement in a subprogram is to return control to the caller, rather than to stop execution. As for the `end program` statement, we recommend using the full form for the `end` statement so that it is clear both to the reader and to the compiler exactly what it terminates.

The simplest form of external subprogram defines a subroutine without any arguments and has a *subroutine-stmt* of the form

```
subroutine subroutine-name
```

Such a subprogram is useful when a program consists of a sequence of distinct phases, in which case the main program consists of a sequence of `call` statements that invoke the subroutines as in the example

```
program game          ! Main program to control a card game
   call shuffle       ! First shuffle the cards.
   call deal          ! Now deal them.
   call play          ! Play the game.
   call display       ! Display the result.
end program game      ! Cease execution.
```

But how do we handle the flow of information between the subroutines? How does `play` know which cards `deal` has dealt? There are, in fact, two methods by which information may be passed. The first is via data held in a module (Section 5.5) and accessed by the subprograms, and the second is via arguments (Section 5.7) in the procedure calls.

5.5 Modules

The third type of program unit, the module, provides a means of packaging global data, derived types and their associated operations, interface blocks (Section 5.11), and namelist groups (Section 7.15). Everything associated with some task (such as interval arithmetic, see later in this section) may be collected into a module and accessed whenever it is needed. Those parts that are associated with the internal working and are of no interest to the user may be made 'invisible' to the user, which allows the internal design to be altered without the need to alter the program that uses it and prevents accidental alteration of internal data. We expect Fortran 90/95 libraries to consist of sets of modules.

The module has the form

```
module module-name
    [specification-stmts]
[contains
    module-subprograms]
end [module [module-name]]
```

As for the `end program`, `end subroutine`, and `end function` statements, we recommend using the full form for the `end` statement.

In its simplest form, the body consists only of data specifications. For example

```
module state
   integer, dimension(52) :: cards
end module state
```

might hold the state of play of the game of Section 5.4. It is accessed by the statement

```
use state
```

appearing at the beginnings of the main program `game` and subprograms `shuffle`, `deal`, `play`, and `display`. The array `cards` is set by `shuffle` to contain the integer values 1 to 52 in a random order, where each integer value corresponds to a predefined playing card. For instance, 1 might stand for the ace of clubs, 2 for the two of clubs, etc. up to 52 for the king of spades. The array `cards` is changed by the subroutines `deal` and `play`, and finally accessed by subroutine `display`.

A further example of global data in a module would be the definitions of the values of the kind type parameters that might be required throughout a program (Sections 2.6.1 and 2.6.2). They can be placed in a module and used wherever they are required. On a processor that supports all the kinds listed, an example might be:

```
module numeric_kinds
   ! named constants for 4, 2, and 1 byte integers:
   integer, parameter ::                                  &
        i4b = selected_int_kind(9),                       &
        i2b = selected_int_kind(4),                       &
        i1b = selected_int_kind(2)
   ! and for single, double and quadruple precision reals:
   integer, parameter ::                                  &
        sp = kind(1.0),                                   &
        dp = selected_real_kind(2*precision(1.0_sp)), &
        qp = selected_real_kind(2*precision(1.0_dp))
end module numeric_kinds
```

A very useful role for modules is to contain definitions of types and their associated operators. For example, a module might contain the type `interval` of Section 3.8, as shown in Figure 5.2. Given this module, any program unit needing this type and its operators need only include the statement

```
use interval_arithmetic
```

at the head of its specification statements.

Figure 5.2

```
module interval_arithmetic
   type interval
      real :: lower, upper
   end type interval
   interface operator(+)
      module procedure add_intervals
   end interface
   :
contains
   function add_intervals(a,b)
      type(interval)             :: add_intervals
      type(interval), intent(in) :: a, b
      add_intervals%lower = a%lower + b%lower
      add_intervals%upper = a%upper + b%upper
   end function add_intervals
   :
end module interval_arithmetic
```

A module subprogram has exactly the same form as an external subprogram, except that `function` or `subroutine` *must* be present on the end statement, so there is no need for a separate description. It always has access to other entities of the module, including the ability to call other subprograms of the module, rather as if it contained a `use` statement for its module.

A module may contain use statements that access other modules. It must not access itself directly or indirectly through a chain of use statements, for example a accessing b and b accessing a. No ordering of modules is required by the standard, but normal practice is to require each module to precede its use. We recommend this practice, which will make it impossible for a module to access itself through other modules. It is required by many compilers.

It is possible within a module to specify that some of the entities are private to it and cannot be accessed from other program units. Also there are forms of the use statement that allow access to only part of a module and forms that allow renaming of the entities accessed. These features will be explained in Sections 7.6 and 7.10. For the present, we assume that the whole module is accessed without any renaming of the entities in it.

Besides data definitions, type definitions, subprograms, and interface blocks, a module may contain namelist groups (Section 7.15). The ability to make single definitions of interface blocks will be seen to be important in the context of constructing large libraries of reusable software.

5.6 Internal subprograms

We have seen that internal subprograms may be defined inside main programs and external subprograms, and within module subprograms. They have the form

```
subroutine-stmt
    [specification-stmts]
    [executable-stmts]
end subroutine [subroutine-name]
```

or

```
function-stmt
    [specification-stmts]
    [executable-stmts]
end function [function-name]
```

that is, the same form as a module subprogram, except that they may not contain further internal subprograms. Note that function or subroutine must be present on the end statement. An internal subprogram automatically has access to all the host's entities, including the ability to call its other internal subprograms. Internal subprograms must be preceded by a contains statement in the host.

In the rest of this chapter, we describe several properties of subprograms that apply to external, module, and internal subprograms. We therefore do not need to describe internal subprograms separately. An example is given in Figure 5.10 (Section 5.15).

5.7 Arguments of procedures

Procedure arguments provide an alternative means for two program units to access the same data. Returning to our card game example, instead of placing the array cards in a module, we might declare it in the main program and pass it as an actual argument to each subprogram, as shown in Figure 5.3.

Figure 5.3

```
program game        ! Main program to control a card game
   integer, dimension(52) :: cards
   call shuffle(cards)       ! First shuffle the cards.
   call deal(cards)          ! Now deal them.
   call play(cards)          ! Play the game.
   call display(cards)       ! Display the result.
end program game             ! Cease execution.
```

Each subroutine receives cards as a dummy argument. For instance, shuffle has the form shown in Figure 5.4.

Figure 5.4

```
subroutine shuffle(cards)
   ! Subroutine that places the values 1 to 52 in cards
   ! in random order.
   integer, dimension(52) :: cards
   ! Statements that fill cards
   :
end subroutine shuffle   ! Return to caller.
```

We can, of course, imagine a card game in which deal is going to deal only three cards to each of four players. In this case, it would be a waste of time for shuffle to prepare a deck of 52 cards when only the first 12 cards are needed. This can be achieved by requesting shuffle to limit itself to a number of cards that is transmitted in the calling sequence thus:

```
call shuffle(3*4, cards(1:12))
```

Inside shuffle, we would define the array to be of the given length and the algorithm to fill cards would be contained in a do construct with this number of iterations, as shown in Figure 5.5.

We have seen how it is possible to pass an array and a constant expression between two program units. An actual argument may be any variable or expression (or a procedure name, see Section 5.12). Each dummy argument of the called procedure must agree with the corresponding actual argument in type, type parameters, and shape (the requirements on character length and shape agreement

Figure 5.5

```
subroutine shuffle(ncards, cards)
   integer                    :: ncards, icard
   integer, dimension(ncards) :: cards
   do icard = 1, ncards
      :
      cards(icard) = ...
   end do
end subroutine shuffle
```

are relaxed in Chapter 12). However, the names do not have to be the same. For instance, if two decks had been needed, we might have written the code thus:

```
program game
   integer, dimension(52) :: acards, bcards
   call shuffle(acards)       ! First shuffle the a deck.
   call shuffle(bcards)       ! Next shuffle the b deck.
   :
end program game
```

The important point is that subprograms can be written independently of one another, the association of the dummy arguments with the actual arguments occurring each time the call is executed. We can imagine `shuffle` being used in other programs which use other names. In this manner, libraries of subprograms may be built up.

Being able to have different names for actual and dummy arguments provides a useful flexibility, but it should only be used when it is actually needed. When the same name can be used, the code is more readable.

As the type of an actual argument and its corresponding dummy argument must agree, care must be taken when using component selection within an actual argument. Thus, supposing the type definitions `point` and `triangle` of Figure 2.1 (Section 2.9) are available in a module `def`, we might write

```
   use def
   type(triangle) :: t
   :
   call sub(t%a)
   :
contains
   subroutine sub(p)
      type(point) :: p
```

5.7.1 Pointer arguments

A dummy argument is permitted to have the attribute `pointer`. In this case, the actual argument must also have the attribute `pointer`. When the subprogram is invoked, the rank of the actual argument must match that of the dummy argument, and its pointer association status is passed to the dummy argument. On return, the actual argument normally takes its pointer association status from that of the dummy argument, but it becomes undefined if the dummy argument is associated with a target that becomes undefined when the return is executed (for example, if the target is a local variable that does not have the `save` attribute, Section 7.9). The `intent` attribute (Section 5.9) would be ambiguous in this context, since it might refer to the pointer association status alone or to both the pointer association status and the value of its target; it is not allowed to be specified.

In the case of a module or internal procedure, the compiler knows when the dummy argument is a pointer. In the case of an external or dummy procedure, the compiler assumes that the dummy argument is not a pointer unless it is told otherwise in an interface block (Section 5.11).

A pointer actual argument is also permitted to correspond to a non-pointer dummy argument. In this case, the pointer must have a target and the target is associated with the dummy argument, as in

```
   real, pointer :: a(:,:)
   :
   allocate ( a(80,80) )
   call find (a)
   :
subroutine find (c)
   real :: c(:,:)
```

5.7.2 Restrictions on actual arguments

There are two important restrictions on actual arguments, which are designed to allow the compiler to optimize on the assumption that the dummy arguments are distinct from each other and from other entities that are accessible within the procedure. For example, a compiler may arrange for an array to be copied to a local variable on entry, and copied back on return. While an actual argument is associated with a dummy argument:

i) Action that affects the allocation status or pointer association status of the argument or any part of it (any pointer assignment, allocation, deallocation, or nullification) must be taken through the dummy argument. If this is done, then throughout the execution of the procedure, the argument may be referenced only through the dummy argument.

ii) Action that affects the value of the argument or any part of it must be taken through the dummy argument unless

a. the dummy argument has the `pointer` attribute,

b. the part is all or part of a pointer subobject, or

c. the dummy argument has the `target` attribute, the dummy argument does not have intent `in`, the dummy argument is scalar or an assumed-shape array, and the actual argument is a target other than an array section with a vector subscript.

If the value of the argument or any part of it is affected through a dummy argument for which neither a., b., or c. holds, then throughout the execution of the procedure, the argument may be referenced only through that dummy argument.

An example of i) is a pointer that is nullified (Section 6.5.4) while still associated with the dummy argument. As an example of ii), consider

```
call modify(a(1:5), a(3:9))
```

Here, `a(3:5)` may not be changed through either dummy argument since this would violate the rule for the other argument. However, `a(1:2)` may be changed through the first argument and `a(6:9)` may be changed through the second. Another example is an actual argument that is an object being accessed from a module; here, the same object must not be accessed from the module by the procedure and redefined. As a third example, suppose an internal procedure call associates a host variable `h` with a dummy argument `d`. If `d` is defined during the call, then at no time during the call may `h` be referenced directly.

5.7.3 Arguments with the target attribute

In most circumstances, an implementation is permitted to make a copy of an actual argument on entry to a procedure and copy it back on return. This may be desirable on efficiency grounds, particularly when the actual argument is not held in contiguous storage. In any case, if a dummy argument has neither the `target` nor `pointer` attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument but remain associated with the actual argument.

However, copy in / copy out is not allowed when

i) a dummy argument has the `target` attribute and is either scalar or is an assumed-shaped array, and

ii) the actual argument is a target other than an array section with a vector subscript.

In this case, the dummy and actual arguments must have the same shape, any pointer associated with the actual argument becomes associated with the dummy argument on invocation, and any pointer associated with the dummy argument on return remains associated with the actual argument.

When a dummy argument has the `target` attribute, but the actual argument is not a target or is an array section with a vector subscript, any pointer associated with the dummy argument obviously becomes undefined on return.

In other cases where the dummy argument has the `target` attribute, whether copy in / copy out occurs is processor dependent. No reliance should be placed on the pointer associations with such an argument after the invocation.

5.8 The return statement

We saw in Section 5.2 that if the last executable statement in a main program is executed and does not cause a branch, the `end` statement is executed and the program stops. Similarly, if the last executable statement in a subprogram is executed and does not cause a branch, the `end` statement is executed and control returns to the point of invocation. Just as the `return` statement is an executable statement that provides an alternative means of stopping execution, so the `return` statement provides an alternative means of returning control from a subprogram. It has the form

```
return
```

Like the `stop` statement, this statement may be labelled, may be part of an `if` statement, and is an executable statement. It must not appear among the executable statements of a main program.

5.9 Argument intent

In Figure 5.5, the dummy argument `cards` was used to pass information out from `shuffle` and the dummy argument `ncards` was used to pass information in. A third possibility is for a dummy argument to be used for both. We can specify the intent on the type declaration statement for the argument, for example:

```
subroutine shuffle(ncards, cards)
   integer, intent(in)                    :: ncards
   integer, intent(out), dimension(ncards) :: cards
```

For input-output arguments, intent `inout` may be specified.

If a dummy argument is specified with intent `in`, it must not be redefined by the procedure, say by appearing on the left-hand side of an assignment or by being passed on as an actual argument to a procedure that redefines it. For the specification intent `inout`, the corresponding actual argument must be a variable because the expectation is that it will be redefined by the procedure. For the specification intent `out`, the corresponding actual argument must again be a variable; in this case, it becomes undefined on entry to the procedure because the intention is that it be used only to pass information out.

If a function specifies a defined operator (Section 3.8), the dummy arguments must have intent `in`. If a subroutine specifies defined assignment (Section 3.9),

the first argument must have intent `out` or `inout`, and the second argument must have intent `in`.

If a dummy argument has no intent, the actual argument may be a variable or an expression, but the actual argument must be a variable if the dummy argument is redefined. It has been traditional for Fortran compilers not to check this rule, since they usually compile each program unit separately. Breaching the rule can lead to program errors at execution time that are very difficult to find. We recommend that all dummy arguments be given a declared intent. Not only is this good documentation, but it allows compilers to make more checks at compile time.

If a dummy argument has the `pointer` attribute, its `intent` is not allowed to be specified. This is because of the ambiguity of whether the intent applies to the target data object or to the pointer association.

If a dummy argument is of a derived type with pointer components, its `intent` attribute refers to the pointer association status of those components. For example, if the intent is `in`, no pointer assignment, allocation, or deallocation is permitted. The `intent` attribute has no bearing on the values of the targets of the pointer components.

5.10 Functions

Functions are similar to subroutines in many respects, but they are invoked within an expression and return a value that is used within the expression. For example, the subprogram in Figure 5.6 returns the distance between two points in space and the statement

```
if (distance(a, c) > distance(b, c) ) then
```

invokes the function twice in the logical expression that it contains.

Figure 5.6

```
function distance(p, q)
   real                              :: distance
   real, intent(in), dimension(3) :: p, q
   distance = sqrt( (p(1)-q(1))**2 + (p(2)-q(2))**2 +    &
                    (p(3)-q(3))**2 )
   ! The intrinsic function sqrt is defined in Section 8.4.
end function distance
```

Note the type declaration for the function result. The result behaves just like a dummy argument with intent out. It is initially undefined, but once defined it may appear in an expression and it may be redefined. The type may also be defined on the `function` statement thus:

```
real function distance(p, q)
```

It is permissible to write functions that change the values of their arguments, modify values in modules, rely on saved local data (Section 7.9), or perform input-output operations. However, these are known as *side-effects* and conflict with good programming practice. Where they are needed, a subroutine should be used. It is reassuring to know that when a function is called, nothing else goes on 'behind the scenes', and it may be very helpful to an optimizing compiler, particularly for internal and module subprograms. A formal mechanism for avoiding side-effects is provided by Fortran 95, but we defer its description to Section 6.10.

A function result may be an array, in which case it must be declared as such. It may also be a pointer,[2] which is very useful when the size of the result depends on a calculation in the function itself. The result is initially undefined. Within the function, it must become associated or defined as disassociated. We expect the function reference usually to be such that a pointer assignment takes place for the result, that is, the reference occurs as the right-hand side of a pointer assignment (Section 3.12) or as a pointer component of a structure constructor. For example, the statements

```
use data_handler
real          :: x(100)
real, pointer :: y(:)
:
y => compact(x)
```

might be used to reference the pointer function

```
function compact(x) ! a procedure to remove duplicates from
                    ! the array x
   real, pointer :: compact(:)
   real          :: x(:)
   integer       :: n
   :               ! find the number of distinct values, n
   allocate(compact(n))
   :               ! copy the distinct values into compact
end function compact
```

in the module data_handler. The reference may also occur as a primary of an expression or as the right-hand side of an ordinary assignment, in which case the result must be associated with a target that is defined and the value of the target is used. We do not recommend this practice, however, since it is likely to lead to memory leakage, discussed at the end of Section 6.5.3.

The value returned by a non-pointer function must always be defined.

As well as being a scalar or array value of intrinsic type, a function result may also be a scalar or array value of a derived type, as we have seen already in Section 3.8. When the function is invoked, the function value must be used as a whole, that

[2]However, it is not possible for a pointer to have a function as its target. In other words, *dynamic binding*, or association of a pointer with a function at run time, is not available.

is, it is not permitted to be qualified by substring, array-subscript, array-section, or structure-component selection.

Although this is not very useful, a function is permitted to have an empty argument list. In this case, the brackets are obligatory both within the function statement and at every invocation.

5.10.1 Prohibited side-effects

In order to assist an optimizing compiler, the standard prohibits reliance on certain side-effects. It specifies that it is not necessary for a processor to evaluate all the operands of an expression, or to evaluate entirely each operand, if the value of the expression can be determined otherwise. For example, in evaluating

```
x>y .or. l(z)  ! x, y, and z are real; l is a logical function
```

the function reference need not be made if `x` is greater than `y`. Since some processors will make the call and others will not, any variable (for example `z`) that is redefined by the function becomes undefined following such expression evaluation. Similarly, it is not necessary for a processor to evaluate any subscript or substring expressions for an array of zero size or character object of zero character length.

Another prohibition is that a function reference must not redefine the value of a variable that appears in the same statement or affect the value of another function reference in the same statement. For example, in

```
d = max(distance(p,q), distance(q,r))
```

`distance` is required not to redefine its arguments. This rule allows any expressions that are arguments of a single procedure call to be evaluated in any order. With respect to this rule, an `if` statement,

```
if (lexpr) stmt
```

is treated as the equivalent `if` construct

```
if (lexpr) then
   stmt
end if
```

and the same is true for the `where` statement (Section 6.8).

5.11 Explicit and implicit interfaces

A call to an internal subprogram must be from a statement within the same program unit. It may be assumed that the compiler will process the program unit as a whole and will therefore know all about any internal subprogram. In particular, it will know about its *interface*, that is whether it defines a function or

a subroutine, the names and properties of the arguments, and the properties of the result if it defines a function. This, for example, permits the compiler to check whether the actual and dummy arguments match in the way that they should. We say that the interface is *explicit.*

A call to a module subprogram must either be from another statement in the module or from a statement following a `use` statement for the module. In both cases, the compiler will know all about the subprogram, and again we say that the interface is explicit. Similarly, intrinsic procedures (Chapter 8) always have explicit interfaces.

When compiling a call to an external or dummy procedure (Section 5.12), the compiler normally does not have a mechanism to access its code. We say that the interface is *implicit.* To specify that a name is that of an external or dummy procedure, the `external` statement is available. It has the form

`external` *external-name-list*

and appears with other specification statements, after any `use` or `implicit` statements (Section 7.2) and before any executable statements. The type and type parameters of a function with an implicit interface are usually specified by a type declaration statement for the function name; an alternative is by the rules of implicit typing (Section 7.2) applied to the name, but this is not available in a module unless the function has the `private` attribute (see Section 7.6).

The `external` statement merely specifies that each *external-name* is the name of an external or dummy procedure. It does not specify the interface, which remains implicit. However, a mechanism is provided for the interface to be specified. It may be done through an interface block of the form

```
interface
    interface-body
end interface
```

Normally, the *interface-body* is an exact copy of the subprogram's header, the specifications of its arguments and function result, and its `end` statement. However,

- the names of the arguments may be changed;
- other specifications may be included (for example, for a local variable), but not internal procedures, `data` or `format` statements;
- the information may be given by a different combination of statements[3]; and
- in the case of an array argument or function result, the bounds may differ as long as the shape does not.

[3]A practice that is permitted by the standard, but which we do not recommend, is for a dummy argument to be declared implicitly as a procedure by invoking it in an executable statement. If the subprogram has such a dummy procedure, the interface will need an `external` statement for that dummy procedure.

An *interface-body* may be provided for a call to an external procedure defined by means other than Fortran (usually assembly language).

Naming a procedure in an `external` statement or giving it an interface body (doing both is not permitted) ensures that it is an external or dummy procedure. We strongly recommend the practice for external procedures, since otherwise the processor is permitted to interpret the name as that of an intrinsic procedure. It is needed for portability since processors are permitted to provide additional intrinsics. Naming a procedure in an `external` statement makes all versions of an intrinsic procedure having the same name unavailable. The same is true for giving it an interface body in the way described in the next section (but not when the interface is generic, Section 5.18).

The interface block is placed in a sequence of specification statements and this suffices to make the interface explicit. Perhaps the most convenient way to do this is to place the interface block among the specification statements of a module and then `use` the module. We imagine subprogram libraries being written as sets of external subprograms which are precompiled and whose interfaces are collected into modules. This keeps the modules of modest size. Note that if a procedure is accessible in a scoping unit, its interface is either explicit or implicit there. An external procedure may have an explicit interface in some scoping units and an implicit interface in others.

Interface blocks may also be used to allow procedures to be called as defined operators (Section 3.8), as defined assignments (Section 3.9), or under a single generic name. We therefore defer description of the full generality of the interface block until Section 5.18, where overloading is discussed.

An explicit interface is required to invoke a procedure with a pointer or target dummy argument or a pointer function result, and is required for several useful features that we will meet later in this and the next chapter. It is needed so that the processor can make the appropriate linkage. Even when not strictly required, it gives the compiler an opportunity to examine data dependencies and thereby improve optimization. Explicit interfaces are also desirable because of the additional security that they provide. It is straightforward to ensure that all interfaces are explicit and we recommend the practice.

5.12 Procedures as arguments

So far, we have taken the actual arguments of a procedure invocation to be variables and expressions, but another possibility is for them to be procedures. Let us consider the case of a library subprogram to perform function minimization. It needs to receive the user's function, just as the subroutine `shuffle` in Figure 5.5 needs to receive the required number of cards. The library code might look like the code in Figure 5.7. Notice the way the procedure argument is declared by an interface block playing a similar role to that of the type declaration statement for a data object. Although such an interface block is not required, we recommend its use.

Figure 5.7

```
real function minimum(a, b, func) ! Returns the minimum
        ! value of the function func(x) in the interval (a,b)
   real, intent(in) :: a, b
   interface
      real function func(x)
         real, intent(in) :: x
      end function func
   end interface
   real :: f,x
   :
   f = func(x)    ! invocation of the user function.
   :
end function minimum
```

Just as the type and shape of actual and dummy data objects must agree, so must the properties of the actual and dummy procedures. The agreement is exactly as for a procedure and an interface body for that procedure (see Section 5.11). It would make no sense to specify an `intent` attribute (Section 5.9) for a dummy procedure, and this is not permitted.

On the user side, the code may look like that in Figure 5.8. Notice that the structure is rather like a sandwich: user-written code invokes the library code which in turn invokes user-written code. Again, we recommend the use of an interface block. As a minimum, the procedure name must be declared in an `external` statement.

Figure 5.8

```
program main
   real :: a, b, f
   interface
      real function fun(x)
         real, intent(in) :: x
      end function fun
   end interface
   f = minimum(1.0, 2.0, fun)
   :
end program main
real function fun(x)
   :
end function fun
```

The procedure that is passed must be an external or module procedure and its specific name must be passed when it also has a generic name (Section 5.18). Internal procedures are not permitted because it is anticipated that they may be implemented quite differently (for example, by in-line code), and because of the need to identify the depth of recursion when the host is recursive (Section 5.16) and the procedure involves host variables.

5.13 Keyword and optional arguments

In practical applications, argument lists can get long and many of the arguments may often not be needed. For example, a subroutine for constrained minimization might have the form

```
subroutine mincon(n, f, x, upper, lower,                    &
          equalities, inequalities, convex, xstart)
```

On many calls, there may be no upper bounds, or no lower bounds, or no equalities, or no inequalities, or it may not be known whether the function is convex, or a sensible starting point may not be known. All the corresponding dummy arguments may be declared `optional` (see also Section 7.8). For instance, the bounds might be declared by the statement

```
real, optional, dimension(n) :: upper,lower
```

If the first four arguments are the only wanted ones, we may use the statement

```
call mincon(n, f, x, upper)
```

but usually the wanted arguments are scattered. In this case, we may follow a (possibly empty) ordinary positional argument list for leading arguments by a keyword argument list, as in the statement

```
call mincon(n, f, x, equalities=q, xstart=x0)
```

The keywords are the dummy argument names and there must be no further positional arguments after the first keyword argument.

This example also illustrates the merits of both positional and keyword arguments as far as readability is concerned. A small number of leading positional arguments (for example, `n`, `f`, `x`) are easily linked in the reader's mind to the corresponding dummy arguments. Beyond this, the keywords are very helpful to the reader in making these links. We recommend their use for long argument lists even when there are no gaps caused by optional arguments that are not present.

A non-optional argument must appear exactly once, either in the positional list or in the keyword list. An optional argument may appear at most once, either in the positional list or in the keyword list. An argument must not appear in both lists.

The called subprogram needs some way to detect whether an argument is present so that it can take appropriate action when it is not. This is provided by the intrinsic function `present` (see Section 8.2). For example

```
present(xstart)
```

returns the value `.true.` if the current call has provided a starting point and `.false.` otherwise. When it is absent, the subprogram might use a random number generator to provide a starting point.

A slight complication occurs if an optional dummy argument is used within the subprogram as an actual argument in a procedure invocation. For example, our minimization subroutine might start by calling a subroutine that handles the corresponding equality problem by the call

```
call mineq(n, f, x, equalities, convex, xstart)
```

In such a case, an absent optional argument is also regarded as absent in the second-level subprogram. For instance, when `convex` is absent in the call of `mincon`, it is regarded as absent in `mineq` too. Such absent arguments may be propagated through any number of calls, provided the dummy argument is optional in each case. An absent argument further supplied as an actual argument must be specified as a whole, and not as a subobject. Furthermore, an absent pointer is not permitted to be associated with a non-pointer dummy argument (the target is doubly absent).

Since the compiler will not be able to make the appropriate associations unless it knows the keywords (dummy argument names), the interface must be explicit (Section 5.11) if any of the dummy arguments are optional or keyword arguments are in use. Note that an interface block may be provided for an external procedure to make the interface explicit. In all cases where an interface block is provided, it is the names of the dummy arguments in the block that are used to resolve the associations.

5.14 Scope of labels

Execution of the main program or a subprogram always starts at its first executable statement and any branching always takes place from one of its executable statements to another. Indeed, each subprogram has its own independent set of labels. This includes the case of a host subprogram with several internal subprograms. The same label may be used in the host and the internal subprograms without ambiguity.

This is our first encounter with *scope*. The scope of a label is a main program or a subprogram, excluding any internal subprograms that it contains. The label may be used unambiguously anywhere among the executable statements of its scope. Notice that the host end statement may be labelled and be a branch target from a host statement, that is the internal subprograms leave a hole in the scope of the host (see Figure 5.9).

5.15 Scope of names

In the case of a named entity, there is a similar set of statements within which the name may always be used to refer to the entity. Here, type definitions and interface blocks as well as subprograms can knock holes in scopes. This leads us to regard each program unit as consisting of a set of non-overlapping scoping units. A *scoping unit* is one of the following:

- a derived-type definition,
- a procedure interface body, excluding any derived-type definitions and interface bodies contained within it, or
- a program unit or subprogram, excluding derived-type definitions, interface bodies, and subprograms contained within it.

An example containing five scoping units is shown in Figure 5.9.

Figure 5.9

```
module scope1                ! scope 1
   :                         ! scope 1
contains                     ! scope 1
   subroutine scope2         ! scope 2
      type scope3            ! scope 3
         :                   ! scope 3
      end type scope3        ! scope 3
      interface              ! scope 2
         :                   ! scope 4
      end interface          ! scope 2
      :                      ! scope 2
   contains                  ! scope 2
      function scope5(...)   ! scope 5
         :                   ! scope 5
      end function scope5    ! scope 5
   end subroutine scope2     ! scope 2
end module scope1            ! scope 1
```

Once an entity has been declared in a scoping unit, its name may be used to refer to it in that scoping unit. An entity declared in another scoping unit is always a different entity even if it has the same name and exactly the same properties[4]. Each is known as a *local* entity. This is very helpful to the programmer, who does not have to be concerned about the possibility of accidental name clashes. Note that this is true for derived types, too. Even if two derived types have the same

[4]Apart from the effect of storage association, which is not discussed until Chapter 11 and whose use we strongly discourage.

name and the same components, entities declared with them are treated as being of different types[5].

A use statement of the form

use *module-name*

is regarded as a re-declaration of all the module entities inside the local scoping unit, with exactly the same names and properties. The module entities are said to be accessible by *use association*. Names of entities in the module may not be used to declare local entities (but see Section 7.10 for a description of further facilities provided by the use statement when greater flexibility is required).

In the case of a derived-type definition, a module subprogram, or an internal subprogram, the name of an entity in the host (including an entity accessed by use association) is similarly treated as being automatically re-declared with the same properties, provided no entity with this name is declared locally, is a local dummy argument or function result, or is accessed by use association. The host entity is said to be accessible by *host association*. For example, in the subroutine inner of Figure 5.10, x is accessible by host association, but y is a separate local variable and the y of the host is inaccessible. We note that inner calls another internal procedure that is a function, f; it must not contain a type specification for that function, as the interface is already explicit. Such a specification would, in fact, declare a different, *external* function of that name. The same remark applies to a module procedure calling a function in the same module.

Figure 5.10

```
subroutine outer
   real :: x, y
   :
contains
   subroutine inner
      real :: y
      y = f(x) + 1.
      :
   end subroutine inner
   function f(z)
      real             :: f
      real, intent(in) :: z
      :
   end function f
end subroutine outer
```

Note that the host does not have access to the local entities of any subroutine that it contains.

[5] Apart from storage association effects (Chapter 11)

Host association does not extend to interface blocks. This allows an interface body to be constructed mechanically from the specification statements of an external procedure. Note, however, that if a derived type needed for the interface is accessed from a module, the interface block constructed from the procedure cannot be placed in the module that defines the type since a module is not permitted to access itself. For example, the following is not permitted:

```
module m
   type t
      integer :: i, j, k
   end type t
   interface g
      subroutine s(a)
         use m        ! Illegal module access.
         type(t) :: a
      end subroutine s
   end interface
end module m
```

Within a scoping unit, each named data object, procedure, derived type, named construct, and namelist group (Section 7.15) must have a distinct name, with the one exception of generic names of procedures (to be described in Section 5.18). Note that this means that any appearance of the name of an intrinsic procedure in another rôle makes the intrinsic procedure inaccessible by its name (the renaming facility described in Section 7.10 allows an intrinsic procedure to be accessed from a module and renamed). Within a type definition, each component of the type, each intrinsic procedure referenced, and each derived type or named constant accessed by host association, must have a distinct name. Apart from these rules, names may be re-used. For instance, a name may be used for the components of two types, or the arguments of two procedures referenced with keyword calls.

The names of program units and external procedures are *global*, that is available anywhere in a complete program. Each must be distinct from the others and from any of the local entities of the program unit.

At the other extreme, the `do` variable of an implied-do in a `data` statement (Section 7.5.2) or an array constructor (Section 6.16) has a scope that is just the implied-do. It is different from any other entity with the same name.

5.16 Direct recursion

Normally, a subprogram may not invoke itself, either directly or indirectly through a sequence of other invocations. However, if the leading statement is prefixed `recursive`, this is allowed. Where the subprogram is a function that calls itself directly in this fashion, the function name cannot be used for the function result and another name is needed. This is done by adding a further clause to the

function statement as in Figure 5.11, which illustrates the use of a recursive function to calculate $n! = n(n-1)\ldots(1)$.

Figure 5.11

```
recursive function factorial(n) result(res)
   integer, intent(in) :: n
   integer             :: res
   if(n==1) then
      res = 1
   else
      res = n*factorial(n-1) ! Beware - few computers check for
   end if                    ! integer overflow.
end function factorial
```

The type of the function (and its result) may be specified on the function statement, either before or after the token `recursive`:

```
integer recursive function factorial(n) result(res)
```

or

```
recursive integer function factorial(n) result(res)
```

or in a type declaration statement for the result name (as in Figure 5.11). In fact, the result name, rather than the function name, must be used in any specification statement. In the executable statements, a reference to the function name is a recursive invocation of the function and the result name must be used for the result variable. If there is no `result` clause, the function name is used for the result, and is not available for a recursive function call.

The `result` clause may also be used in a non-recursive function.

Just as in Figure 5.11, any recursive procedure that calls itself directly must contain a conditional test that terminates the sequence of calls at some point, otherwise it will call itself indefinitely.

Each time a recursive procedure is invoked, a fresh set of local data objects is created, which ceases to exist on return. They consist of all data objects declared in its specification statements or declared implicitly (see Section 7.2), but excepting those with the `data` or `save` attribute (see Sections 7.5 and 7.9) and any dummy arguments. The interface is explicit within the procedure.

5.17 Indirect recursion

A procedure may also be invoked by indirect recursion, that is, it may call itself through calls to other procedures. To illustrate that this may be useful, suppose we wish to perform a two-dimensional integration but have only the procedure for one-dimensional integration shown in Figure 5.12.

Figure 5.12

```
recursive function integrate(f, bounds)
   ! Integrate f(x) from bounds(1) to bounds(2)
   real :: integrate
   interface
      function f(x)
         real             :: f
         real, intent(in) :: x
      end function f
   end interface
   real, dimension(2), intent(in) :: bounds
   :
end function integrate
```

For example, suppose that it is desired to integrate a function f of x and y over a rectangle. We might write a Fortran function in a module to receive the value of x as an argument and the value of y from the module itself by host association, as shown in Figure 5.13. We can then integrate over x for a particular value of

Figure 5.13

```
module func
   real               :: yval
   real, dimension(2) :: xbounds, ybounds
contains
   function f(xval)
      real             :: f
      real, intent(in) :: xval
      r = ...      ! Expression involving xval and yval
   end function f
end module func
```

y, as shown in Figure 5.14, where `integrate` might be as shown in Figure 5.12. We may now integrate over the whole rectangle thus

```
volume = integrate(fy, ybounds)
```

Note that `integrate` calls `fy`, which in turn calls `integrate`.

5.18 Overloading and generic interfaces

We saw in Section 5.11 how to use a simple interface block to provide an explicit interface to an external or dummy procedure. Another use is for overloading, that is being able to call several procedures by the same generic name. Here

Figure 5.14

```
function fy(y)
   use func
   real             :: fy
   real, intent(in) :: y
   yval = y
   r = integrate(f, xbounds)
end function fy
```

the interface block contains several interface bodies and the `interface` statement specifies the generic name. For example,

```
interface gamma
   function sgamma(x)
      real (selected_real_kind( 6))             :: sgamma
      real (selected_real_kind( 6)), intent(in) :: x
   end function sgamma
   function dgamma(x)
      real (selected_real_kind(12))             :: dgamma
      real (selected_real_kind(12)), intent(in) :: x
   end function dgamma
end interface
```

permits both the functions `sgamma` and `dgamma` to be invoked using the generic name `gamma`.

A specific name for a procedure may be the same as its generic name. For example, the procedure `sgamma` could be renamed `gamma` without invalidating the interface block.

Furthermore, a generic name may be the same as another accessible generic name. In such a case, all the procedures that have this generic name may be invoked through it. This capability is important, since a module may need to extend the intrinsic functions such as `sin` to a new type such as `interval` (Section 3.8).

If it is desired to overload a module procedure, the interface is already explicit so it is inappropriate to specify an interface body. Instead, the statement module procedure *procedure-name-list* is included in the interface block in order to name the module procedures for overloading: if the functions `sgamma` and `dgamma` above were defined in a module, the interface block becomes

```
interface gamma
   module procedure sgamma, dgamma
end interface
```

It is probably most convenient to place such a block in the module itself.

Fortran 95 allows any generic specification on an `interface` statement to be repeated on the corresponding `end interface` statement, for example,

```
end interface gamma              ! Fortran 95 only
```

As for other end statements, we recommend use of this fuller form.

Another form of overloading occurs when an interface block specifies a defined operation (Section 3.8) or a defined assignment (Section 3.9) to *extend* an intrinsic operation or assignment. The scope of the defined operation or assignment is the scoping unit that contains the interface block, but it may be accessed elsewhere by use or host association. If an intrinsic operator is extended, the number of arguments must be consistent with the intrinsic form (for example, it is not possible to define a unary *).

The general form of the interface block is

```
interface [generic-spec]
   [interface-body ]...
   [module procedure procedure-name-list]...
      ! In Fortran 95, interface bodies and
      ! module procedure statements may appear in any order.
end interface [generic-spec] ! Only in Fortran 95 is
                             ! generic-spec allowed here
```

where *generic-spec* is

generic-name, `operator(`*defined-operator*`)`, or `assignment(=)`.

A `module procedure` statement is permitted only when a *generic-spec* is present, and all the procedures must be accessible module procedures (as shown in the complete module in Figure 5.16 below). No procedure name may be given a particular *generic-spec* more than once in the interface blocks within a scoping unit. An interface body must be provided for an external or dummy procedure.

If `operator` is specified on the interface statement, all the procedures in the block must be functions with one or two non-optional arguments having intent `in`[6]. If `assignment` is specified, all the procedures must be subroutines with two non-optional arguments, the first having intent `out` or `inout` and the second intent `in`. In order that invocations are always unambiguous, if two procedures have the same generic operator and the same number of arguments or both define assignment, one must have a dummy argument that corresponds by position in the argument list to a dummy argument of the other that has a different type, different kind type parameter, or different rank.

All procedures that have a given generic name must be subroutines or all must be functions, including the intrinsic ones when an intrinsic procedure is extended. Any two non-intrinsic procedures with the same generic name must have arguments that differ sufficiently for any invocation to be unambiguous. The rule is that either

[6]Since intent must not be specified for a pointer dummy argument (Section 5.7.1), this implies that if an operand of derived data type also has the pointer attribute, it is the value of its target that is passed to the function defining the operator, and not the pointer itself. The pointer status is inaccessible within the function.

i) one of them has more non-optional dummy arguments of a particular data type, kind type parameter, and rank than the other has dummy arguments (including optional dummy arguments) of that data type, kind type parameter, and rank; or

ii) at least one of them must have a non-optional dummy argument that both

- corresponds by position in the argument list to a dummy argument that is not present in the other, is present with a different type or kind type parameter, or is present with a different rank, and
- corresponds by name to a dummy argument that is not present in the other, is present with a different type or kind type parameter, or is present with a different rank.

For case (ii), both rules are needed in order to cater for both keyword and positional dummy argument lists. For instance, the interface in Figure 5.15 is invalid because the two functions are always distinguishable in a positional call, but not on a keyword call such as `f(i=int, x=posn)`. If a generic invocation is ambiguous between a non-intrinsic and an intrinsic procedure, the non-intrinsic procedure is invoked.

Figure 5.15

```
!  Example of a broken overloading rule
   interface f
      function fxi(x,i)
         real             :: fxi
         real, intent(in) :: x
         integer          :: i
      end function fxi
      function fix(i,x)
         real             :: fix
         real, intent(in) :: x
         integer          :: i
      end function fix
   end interface
```

Note that the presence or absence of the pointer attribute is insufficient to ensure an unambiguous invocation since a pointer actual argument may be associated with a non-pointer dummy argument, see Section 5.7.1.

There are many scientific applications in which it is useful to keep a check on the sorts of quantities involved in a calculation. For instance, in dimensional analysis, whereas it might be sensible to divide length by time to obtain velocity, it is not sensible to add time to velocity. There is no intrinsic way to do this, but we conclude this section with an outline example, Figures 5.16 & 5.17, of how it might be achieved using derived types.

Figure 5.16

```
module sorts
   type time
      real :: seconds
   end type time
   type velocity
      real :: metres_per_second
   end type velocity
   type length
      real :: metres
   end type length
   type length_squared
      real :: metres_squared
   end type length_squared
   interface operator(/)
      module procedure length_by_time
   end interface
   interface operator(+)
      module procedure time_plus_time
   end interface
   interface sqrt
      module procedure sqrt_metres_squared
   end interface
contains
   function length_by_time(s, t)
      type(length), intent(in) :: s
      type(time), intent(in)   :: t
      type(velocity)           :: length_by_time
      length_by_time%metres_per_second = s%metres / t%seconds
   end function length_by_time
   function time_plus_time(t1, t2)
      type(time), intent(in)   :: t1, t2
      type(time)               :: time_plus_time
      time_plus_time%seconds = t1%seconds + t2%seconds
   end function time_plus_time
   function sqrt_metres_squared(l2)
      type(length_squared), intent(in) :: l2
      type(length)                     :: sqrt_metres_squared
      sqrt_metres_squared%metres = sqrt(l2%metres_squared)
   end function sqrt_metres_squared
end module sorts
```

Figure 5.17

```
program test
   use sorts
   type(length)          :: s  = length(10.0), l
   type(length_squared)  :: s2 = length_squared(10.0)
   type(velocity)        :: v
   type(time)            :: t  = time(3.0)
   v = s / t
! Note: v = s + t    or    v = s * t  would be illegal
   t = t + time(1.0)
   l = sqrt(s2)
   print *, v, t, l
end program test
```

Note that definitions for operations between like entities are also required, as shown by `time_plus_time`. Similarly, any intrinsic function that might be required, here `sqrt`, must be overloaded appropriately. Of course, this can be avoided if the components of the variables are referenced directly, as in

```
t%seconds = t%seconds + 1.0
```

5.19 Assumed character length

A character dummy argument may be declared with an asterisk for the value of the length type parameter, in which case it automatically takes the value from the actual argument. For example, a subroutine to sort the elements of a character array might be written thus

```
subroutine sort(n,chars)
   integer, intent(in)                          :: n
   character(len=*), dimension(n), intent(in) :: chars
   :
end subroutine sort
```

If the length of the associated actual argument is needed within the procedure, the intrinsic function `len` (Section 8.6) may be invoked, as in Figure 5.18.

An asterisk must not be used for a kind type parameter value. This is because a change of character length is analogous to a change of an array size and can easily be accommodated in the object code, whereas a change of kind probably requires a different machine instruction for every operation involving the dummy argument. A different version of the procedure would need to be generated for each possible kind value of each argument. The overloading feature (previous section) gives the programmer an equivalent functionality with explicit control over which versions are generated.

Figure 5.18

```
integer function count (letter, string)
   character (1), intent(in) :: letter
   character (*), intent(in) :: string
!     Count the number of occurrences of letter in string
   count = 0
   do i = 1, len(string)
      if (string(i:i) == letter) count = count + 1
   end do
end function count
```

5.20 The subroutine and function statements

We finish this chapter by giving the full syntax of the Fortran 90 `subroutine` and `function` statements, which have so far been explained through examples. It is

```
[recursive]                                                    &
subroutine subroutine-name [([dummy-argument-list])]
```

and

```
[prefix] function function-name ([dummy-argument-list]) &
[result(result-name) ]
```

where *prefix* is

```
type [recursive]
```

or

```
recursive [type]
```

(for details of *type* see Section 7.13).

Each feature has been explained separately and the meanings are the same in the combinations allowed by the syntax. The syntax has been extended in Fortran 95 to allow pure and elemental procedures to be specified (Sections 6.10 and 6.11).

5.21 Summary

A program consists of a sequence of program units. It must contain exactly one main program but may contain any number of modules and external subprograms. We have described each kind of program unit. Modules contain data definitions, type definitions, namelist groups, interface blocks, and module subprograms, all of which may be accessed in other program units with the `use` statement. The program units may be in any order, but many compilers require modules to precede their use.

Subprograms define procedures, which may be functions or subroutines. They may also be defined intrinsically (Chapter 8) and external procedures may be defined by means other than Fortran. We have explained how information is passed between program units and to procedures through argument lists and through the use of modules. Procedures may be called recursively provided they are correspondingly specified.

The interface to a procedure may be explicit or implicit. If it is explicit, keyword calls may be made, and the procedure may have optional arguments. Interface blocks permit procedures to be invoked as operations or assignments, or by a generic name. The character lengths of dummy arguments may be assumed.

We have also explained about the scope of labels and Fortran names, and introduced the concept of a scoping unit.

Many of the features are new since Fortran 77: the internal subprogram, modules, the interface block, optional and keyword arguments, argument intent, pointer dummy arguments and function results, recursion, and overloading. These are powerful additions to the language, particularly in the construction of large programs and libraries.

5.22 Exercises

1. A subroutine receives as arguments an array of values, x, and the number of elements in x, n. If the mean and variance of the values in x are estimated by

$$\text{mean} = \frac{1}{n}\sum_{i=1}^{n} x(i)$$

and

$$\text{variance} = \frac{1}{n-1}\sum_{i=1}^{n} (x(i) - \text{mean})^2$$

write a subroutine which returns these calculated values as arguments. The subroutine should check for invalid values of n (≤ 1).

2. A subroutine `matrix_mult` multiplies together two matrices A and B, whose dimensions are $i \times j$ and $j \times k$, respectively, returning the result in a matrix C dimensioned $i \times k$. Write `matrix_mult`, given that each element of C is defined by

$$C(m,n) = \sum_{\ell=1}^{J} (A(m,\ell) \times B(\ell,n))$$

The matrices should appear as arguments to `matrix_mult`.

3. the subroutine `random_number` (Section 8.16.3) returns a random number in the range 0.0 to 1.0, that is

```
call random_number(r)   ! 0≤r<1
```

Using this function, write the subroutine `shuffle` of Figure 5.4.

4. A character string consists of a sequence of letters. Write a function to return that letter of the string which occurs earliest in the alphabet, for example, the result of applying the function to 'DGUMVETLOIC' is 'C'.

5. Write an internal procedure to calculate the volume of a cylinder of radius r and length ℓ, $\pi r^2 \ell$, using as the value of π the result of `acos(-1.0)`, and reference it in a host procedure.

6. Choosing a simple card game of your own choice, and using the random number procedure (Section 8.16.3), write subroutines `deal` and `play` of Section 5.4, using data in a module to communicate between them.

7. Objects of the intrinsic type `character` are of a fixed length. Write a module containing a definition of a variable length character string type, of maximum length 80, and also the procedures necessary to:

i) assign a character variable to a string;

ii) assign a string to a character variable;

iii) return the length of a string;

iv) concatenate two strings.

6. Array features

6.1 Introduction

In an era when many computers have the hardware capability for efficient processing of array operands, it is self-evident that a numerically based language such as Fortran should have matching notational facilities. Such facilities provide not only a notational convenience for the programmer, but provide an opportunity to extend the power of the language. However, new optimization techniques are required, for instance the ability to recognize that two or more consecutive array statements may, in some cases, be processed in a single loop at the object code level. These techniques are being progressively introduced.[1]

Arrays were introduced in Sections 2.10 to 2.13, their use in simple expressions and in assignments was explained in Sections 3.10 and 3.11, and they were used as procedure arguments in Chapter 5. These descriptions were deliberately restricted because Fortran contains a very full set of array features whose complete description would have unbalanced those chapters. The purpose of this chapter is to describe the array features in detail, but without anticipating the descriptions of the array intrinsic procedures of Chapter 8; the rich set of intrinsic procedures should be regarded as an integral part of the array features.

6.2 Zero-sized arrays

It might be thought that an array would always have at least one element. However, such a requirement would force programs to contain extra code to deal with certain natural situations. For example, the code in Figure 6.1 solves a lower-triangular set of linear equations. When i has the value n, the sections have size zero, which is just what is required.

Fortran allows arrays to have zero size in all contexts. Whenever a lower bound exceeds the corresponding upper bound, the array has size zero.

There are few special rules for zero-sized arrays because they follow the usual rules, though some care may be needed in their interpretation. For example, two zero-sized arrays of the same rank may have different shapes. One might have shape (0,2) and the other (0,3) or (2,0). Such arrays of differing shape are not

[1] A fuller discussion of this topic can be found in *Optimizing Supercompilers for Supercomputers*, M. Wolfe (Pitman, 1989).

Figure 6.1

```
do i = 1,n
   x(i) = b(i) / a(i, i)
   b(i+1:n) = b(i+1:n) - a(i+1:n, i) * x(i)
end do
```

conformable and therefore may not be used together as the operands of a binary operation. However, an array is always conformable with a scalar so the statement

zero-sized-array = scalar

is valid and the scalar is 'broadcast to all the array elements', making this a 'do nothing' statement.

A zero-sized array is regarded as being defined always, because it has no values that can be undefined.

6.3 Assumed-shape arrays

Outside Chapter 11, we require that the shapes of actual and dummy arguments agree, and so far we have achieved this by passing the extents of the array arguments as additional arguments. However, it is possible to require that the shape of the dummy array be taken automatically to be that of the corresponding actual array argument. Such an array is said to be an *assumed-shape* array. When the shape is declared by the `dimension` clause, each dimension has the form

[*lower-bound*]:

where *lower-bound* is an integer expression that may depend on module data or the other arguments (see Section 7.14 for the exact rules). If *lower-bound* is omitted, the default value is 1. Note that it is the shape that is passed, and not the upper and lower bounds. For example, if the actual array is `a`, declared thus:

```
real, dimension(0:10, 0:20) :: a
```

and the dummy array is `da`, declared thus:

```
real, dimension(:, :) :: da
```

then `a(i,j)` corresponds to `da(i+1,j+1)`; to get the natural correspondence, the lower bound must be declared:

```
real, dimension(0:, 0:) :: da
```

In order that the compiler knows that additional information is to be supplied, the interface must be explicit (Section 5.11) at the point of call. A dummy array with the `pointer` attribute is not regarded as an assumed-shape array because its shape is not necessarily assumed.

6.4 Automatic objects

A procedure with dummy arguments that are arrays whose size varies from call to call may also need local arrays whose size varies. A simple example is the array work in the subroutine to interchange two arrays that is shown in Figure 6.2.

Figure 6.2

```
subroutine swap(a, b)
   real, dimension(:), intent(inout) :: a, b
   real, dimension(size(a))          :: work
            ! size provides the size of an array,
            ! and is defined in Section 8.12.2.
   work = a
   a = b
   b = work
end subroutine swap
```

Arrays whose extents vary in this way are called *automatic arrays*, and are examples of *automatic data objects*. These are data objects whose declarations depend on the value of non-constant expressions, and are not dummy arguments. Implementations are likely to bring them into existence when the procedure is called and destroy them on return, maintaining them on a stack[2]. The non-constant expressions are limited to be specification expressions (Section 7.14).

The other way that automatic objects arise is through varying character length. The variable word2 in

```
subroutine example(word1)
   character(len = *), intent(inout) :: word1
   character(len = len(word1))       :: word2
```

is an example. If a function result has varying character length, the interface must be explicit at the point of call because the compiler needs to know this, as shown in Figure 6.3.

An array bound or the character length of an automatic object is fixed for the duration of each execution of the procedure and does not vary if the value of the specification expression varies or becomes undefined.

Some small restrictions on the use of automatic data objects appear in Sections 7.5, 7.9, and 7.15.

[2]A stack is a memory management mechanism whereby fresh storage is established and old storage is discarded on a 'last in, first out' basis within contiguous memory.

Figure 6.3

```
program loren
   character (len = *), parameter :: a = 'just a simple test'
   print *, double(a)
contains
   function double(a)
      character (len = *), intent(in) :: a
      character (len = 2*len(a))      :: double
      double = a//a
   end function double
end program loren
```

6.5 Heap storage

There is an underlying assumption in Fortran that the processor supplies a mechanism for managing heap [3] storage. The statements described in this section are the user interface to that mechanism.

6.5.1 Allocatable arrays

Sometimes an array is required to be of a size that is known only after some data have been read or some calculations performed. An array with the `pointer` attribute might be used for this purpose, but this is really not appropriate if the other properties of pointers are not needed. Instead, an array that is not a dummy argument or function result may be given the `allocatable` attribute by a statement such as

```
real, dimension(:, :), allocatable :: a
```

Such an array is called *allocatable*. Its rank is specified when it is declared, but the bounds are undefined until an `allocate` statement such as

```
allocate(a(n, 0:n+1))     ! n of type integer
```

has been executed for it. Its initial allocation status is 'not currently allocated' and it becomes allocated following successful execution of an `allocate` statement.

An important example is shown in Figure 6.4. The array `work` is placed in a module and is allocated at the beginning of the main program to a size that depends on input data. The array is then available throughout program execution in any subprogram that has a `use` statement for `work_array`.

When an allocatable array `a` is no longer needed, it may be deallocated by execution of the statement

[3] A heap is a memory management mechanism whereby fresh storage may be established and old storage may be discarded in any order. Mechanisms to deal with the progressive fragmentation of the memory are usually required.

Figure 6.4

```
module work_array
   integer                                  :: n
   real, dimension(:,:,:), allocatable :: work
end module work_array
program main
   use work_array
   read *, n
   allocate(work(n, 2*n, 3*n))
   :
```

```
deallocate (a)
```

following which the array is 'not currently allocated'. The `deallocate` statement is described in more detail in Section 6.5.3.

If it is required to make any change to the bounds of an allocatable array, the array must be deallocated and then allocated afresh. Allocating an allocatable array that is already allocated, or deallocating an allocatable array that is not currently allocated, is an error.

If a variable-sized array component of a structure is required, unfortunately, an array pointer must be used (see Section 6.14). The prohibition on allocatable arrays here was made to keep the feature simple, but this is now recognized as a mistake that will be corrected in Fortran 2000 (see Sections 1.5 and 13.4).

In Fortran 90, an allocatable array that does not have the `save` attribute (Section 7.9) may have a third allocation state: undefined. Since an undefined allocatable array may not be referenced in any way, not even an enquiry about its status using the `allocated` intrinsic function, we recommend avoiding this state. It occurs on return from a subprogram if the array is local to the subprogram or local to a module that is currently accessed only by the subprogram, and the array is allocated. To avoid this situation, such an allocatable array must be explicitly deallocated before such a return.

In Fortran 95, the undefined allocation status cannot occur. On return from a subprogram, an allocated allocatable array without the `save` attribute is automatically deallocated if it is local to the subprogram, and it is processor dependent as to whether it remains allocated or is deallocated if it is local to a module and is accessed only by the subprogram. This automatic deallocation not only avoids inadvertent memory leakage, but prevents the very undesirable undefined allocation status.

6.5.2 The allocate statement

We mentioned in Section 2.13 that the `allocate` statement can also be used to give fresh storage for a pointer target directly. A pointer becomes associated

(Section 3.3) following successful execution of the statement. The general form of the `allocate` statement is

```
allocate( allocation-list [,stat=stat] )
```

where *allocation-list* is a list of allocations of the form

```
allocate-object [( array-bounds-list )]
```

each *array-bound* has the form

```
[lower-bound:] upper-bound
```

and *stat* is a scalar integer variable that must not be part of an object being allocated.

If the `stat=` specifier is present, *stat* is given either the value zero after a successful allocation or a positive value after an unsuccessful allocation (for example, if insufficient storage is available). After an unsuccessful execution, each array that was not successfully allocated retains its previous allocation or pointer association status. If `stat=` is absent and the allocation is unsuccessful, program execution stops.

Each *allocate-object* is an allocatable array or a pointer. It may have zero character length and in the case of a pointer may be a structure component.

Each *lower-bound* and each *upper-bound* is a scalar integer expression. The default value for the lower bound is 1. The number of *array-bounds* in a list must equal the rank of the *allocate-object*. They determine the array bounds, which do not alter if the value of a variable in one of the expressions changes subsequently. An array may be allocated to be of size zero.

The bounds of all the arrays being allocated are regarded as undefined during the execution of the `allocate` statement, so none of the expressions that specify the bounds may depend on any of the bounds. For example,

```
allocate (a(size(b)), b(size(a)))   ! illegal
```

or even

```
allocate (a(n), b(size(a)))         ! illegal
```

is not permitted, but

```
allocate (a(n))
allocate (b(size(a)))
```

is valid. This restriction allows the processor to perform the allocations in a single `allocate` statement in any order.

In contrast to the case with an allocatable array, a pointer may be allocated a new target even if it is currently associated with a target. In this case, the previous association is broken. If the previous target was created by allocation, it becomes inaccessible unless another pointer is associated with it. We expect linked lists

normally to be created by using a single pointer in an `allocate` statement for each node of the list, using pointer components of the allocated object at the node to hold the links from the node. We illustrate this by the addition of an extra nonzero element to the sparse vector held as a chain of entries of the type

```
type entry
   real                  :: value
   integer               :: index
   type(entry), pointer :: next
end type entry
```

of Section 2.13. The code in Figure 6.5 adds the new entry at the front of the chain. Note the importance of the last statement being a pointer assignment: the assignment

```
first = current
```

would overwrite the old leading entry by the new one.

Figure 6.5

```
type(entry), pointer :: first, current
real    :: fill
integer :: fill_index
:
allocate (current)
current = entry (fill, fill_index, first)
first => current
```

6.5.3 The deallocate statement

When an allocatable array or pointer target is no longer needed, its storage may be recovered by using the `deallocate` statement. Its general form is

`deallocate (` *allocate-object-list* `[,stat=`*stat*`] )`

where each *allocate-object* is an allocatable array that is allocated or a pointer that is associated with the whole of a target that was allocated through a pointer in an `allocate` statement. Here *stat* is a scalar integer variable that must not be deallocated by the statement nor depend on an object that is deallocated by the statement. If `stat=` is present, *stat* is given either the value zero after a successful execution or a positive value after an unsuccessful execution (for example, if a pointer is disassociated). After an unsuccessful execution, each array that was not successfully deallocated retains its previous allocation or pointer association status. If `stat=` is absent and the deallocation is unsuccessful, program execution stops.

A pointer becomes disassociated (Section 3.3) following successful execution of the statement. If there is more than one object in the list, there must be no dependencies among them, to allow the processor to deallocate the objects one by one in any order.

A danger in using the `deallocate` statement is that storage may be deallocated while pointers are still associated with the targets it held. Such pointers are left 'dangling' in an undefined state, and must not be reused until they are again associated with an actual target.

In order to avoid an accumulation of unused and unusable storage, all explicitly allocated storage should be explicitly deallocated when it is no longer required (although, as noted at the end of Section 6.5.1, in Fortran 95, for allocatable arrays, there are circumstances in which this is automatic). This explicit management is required in order to avoid a potentially significant overhead on the part of the processor in handling arbitrarily complex allocation and reference patterns.

Note also that the standard does not specify whether the processor recovers storage allocated through a pointer but no longer accessible through this or any other pointer. This might be important where, for example, a pointer function is referenced within an expression — the programmer cannot rely on the compiler to arrange for deallocation. To ensure that there is no memory leakage, it is necessary to use functions on the right-hand side of pointer assignments, as in the example `compact` in Section 5.10, or as pointer component values in structure constructors, and to deallocate the pointer (`y` in Section 5.10) when it is no longer needed (but see also Section 13.3).

6.5.4 The nullify statement

A pointer may be explicitly disassociated from its target by executing a `nullify` statement. Its general form is

`nullify(`*pointer-object-list*`)`

There must be no dependencies among the objects, in order to allow the processor to nullify the objects one by one in any order. The statement is also useful for giving the disassociated status to an undefined pointer. An advantage of nullifying pointers rather than leaving them undefined is that they may then be tested by the intrinsic function `associated` (Section 8.2). For example, the end of the chain of Figure 6.5 will be flagged as a disassociated pointer if the statement

```
nullify(first)
```

is executed initially to create a zero-length chain. Because often there are other ways to access a target (for example, through another pointer), the `nullify` statement does not deallocate the targets. If deallocation is also required, a `deallocate` statement should be executed instead.

6.6 Elemental operations and assignments

We saw in Section 3.10 that an intrinsic operator can be applied to conformable operands, to produce an array result whose element values are the values of the operation applied to the corresponding elements of the operands. Such an operation is called *elemental.*

It is not essential to use operator notation to obtain this effect. Many of the intrinsic procedures (Chapter 8) are elemental and have scalar dummy arguments that may be called with array actual arguments provided all the array arguments have the same shape. For a function, the shape of the result is the shape of the array arguments. For example, we may find the square roots of all the elements of a real array thus:

```
a = sqrt(a)
```

For a subroutine, if any argument is array-valued, all the arguments with intent `out` or `inout` must be arrays. If a procedure that references an elemental function has an optional array-valued dummy argument that is absent, that dummy argument must not be used in the elemental reference unless another array of the same rank is associated with an non-optional argument of the elemental procedure (to ensure that the rank does not vary from call to call).

Similarly, an intrinsic assignment may be used to assign a scalar to all the elements of an array, or to assign each element of an array to the corresponding element of an array of the same shape (Section 3.11). Such an assignment is also called *elemental.*

If a similar effect is desired for a defined operator, a function must be provided for each rank or pair of ranks for which it is needed (but this is not necessary in Fortran 95, see Section 6.11). For example, the module in Figure 6.6 provides summation for scalars and rank-one arrays of intervals (Section 3.8). We leave it as an exercise for the reader to add definitions for mixing scalars and rank-one arrays.

Similarly, elemental versions of defined assignments must be provided explicitly (but, again, this is not necessary in Fortran 95, see Section 6.11).

6.7 Array-valued functions

We mentioned in Section 5.10 that a function may have an array-valued result, and have used this language feature in Figure 6.6 where the interpretation is obvious.

In order that the compiler should know the shape of the result, the interface must be explicit (Section 5.11) whenever such a function is referenced. The shape is specified within the function definition by the `dimension` attribute for the function name. Unless the function result is a pointer, the bounds must be explicit expressions and they are evaluated on entry to the function. For another example, see the declaration of the function result in Figure 6.7.

An array-valued function is not necessarily elemental. For example, at the end of Section 3.10 we considered the type

Figure 6.6

```
module interval_addition
   type interval
      real :: lower, upper
   end type interval
   interface operator(+)
      module procedure add00, add11
   end interface
contains
   function add00 (a, b)
      type (interval)             :: add00
      type (interval), intent(in) :: a, b
      add00%lower = a%lower + b%lower  ! Production code would
      add00%upper = a%upper + b%upper  ! allow for roundoff.
   end function add00
   function add11 (a, b)
      type (interval), dimension(:), intent(in)       :: a
      type (interval), dimension(size(a))             :: add11
      type (interval), dimension(size(a)), intent(in) :: b
      add11%lower = a%lower + b%lower  ! Production code would
      add11%upper = a%upper + b%upper  ! allow for roundoff.
   end function add11
end module interval_addition
```

```
type matrix
   real :: element
end type matrix
```

Its scalar and rank-one operations might be as for reals, but for multiplying a rank-two array by a rank-one array, we might use the module function shown in Figure 6.7 to provide matrix by vector multiplication.

6.8 The where statement and construct

It is often desired to perform an array operation only for certain elements, say those whose values are positive. The where statement provides this facility. A simple example is

```
where ( a > 0.0 ) a = 1.0/a      ! a is a real array
```

which reciprocates the positive elements of a and leaves the rest unaltered. The general form is

where (*logical-array-expr*) *array-variable* = *expr*

Figure 6.7

```
function mult(a, b)
!
   type(matrix), dimension(:, :)       :: a
   type(matrix), dimension(size(a, 2)) :: b
                 ! size is defined in Section 8.12
   type(matrix), dimension(size(a, 1)) :: mult
   integer                             :: j, n
!
   mult = 0.0     ! A defined assignment from a real
                  ! scalar to a rank-one matrix.
   n = size(a, 1)
   do j = 1, size(a, 2)
      mult = mult + a(1:n, j) * b(j)
            ! Uses defined operations for addition of
            ! two rank-one matrices and multiplication
            ! of a rank-one matrix by a scalar matrix.
   end do
end function mult
```

The logical array expression *logical-array-expr* must have the same shape as *array-variable*. It is evaluated first and then just those elements of *expr* that correspond to elements of *logical-array-expr* that have the value true are evaluated and are assigned to the corresponding elements of *array-variable*. All other elements of *array-variable* are left unaltered. **In Fortran 90**, the assignment must not be a defined assignment (this restriction is relaxed in Fortran 95, see Section 6.8.1).

A single logical array expression may be used for a sequence of array assignments all of the same shape. The general form of this construct is

```
where (logical-array-expr)
   array-assignments
end where
```

The logical array expression *logical-array-expr* is first evaluated and then each array assignment is performed in turn, under the control of this mask. If any of these assignments affect entities in *logical-array-expr*, it is always the value obtained when the where statement is executed that is used as the mask.

Finally, the where construct may take the form

```
where (logical-array-expr)
   array-assignments
elsewhere
   array-assignments
end where
```

Here, the assignments in the first block of assignments are performed in turn under the control of *logical-array-expr* and then the assignments in the second block are performed in turn under the control of .not.*logical-array-expr*. Again, if any of these assignments affect entities in *logical-array-expr*, it is always the value obtained when the where statement is executed that is used as the mask. No array assignment in a where construct may be a branch target statement.

A simple example of a where construct is

```
where (pressure <= 1.0)
   pressure = pressure + inc_pressure
   temp = temp + 5.0
elsewhere
   raining = .true.
end where
```

where pressure, inc_pressure, temp, and raining are arrays of the same shape.

If a where statement or construct masks an elemental function reference, the function is called only for the wanted elements. For example,

```
where ( a > 0 ) a = log(a)      ! log is defined in Section 8.4
```

would not lead to erroneous calls of log for negative arguments.

This masking applies to all elemental function references except any that are within an argument of a non-elemental function reference. The masking does not extend to array arguments of such a function. In general, such arguments have a different shape so that masking would not be possible, but the rule applies in such a case as

```
where (a > 0) a = a/sum(log(a)) ! sum is defined in Section 8.11
```

Here the logarithms of each of the elements of a are summed, and the statement will fail if they are not all positive.

If a non-elemental function reference or an array constructor is masked, it is fully evaluated before the masking is applied.

6.8.1 Some where construct extensions (Fortran 95 only)

In Fortran 95, it is permitted to mask not only the where statement of the where construct (Section 6.8), but also any elsewhere statement that it contains. The masking expressions involved must be of the same shape. A where construct may

contain any number of masked `elsewhere` statements but at most one `elsewhere` statement without a mask, and that must be the final one. In addition, `where` constructs may be nested within one another; the masking expressions of the nested constructs must be of the same shape, as must be the array variables on the left-hand sides of the assignments.

A `where` assignment statement is permitted to be a defined assignment, provided that it is elemental (Section 6.11).

Finally, a `where` construct may be named in the same way as other constructs.

These extensions allow sequences like those in Figure 6.8.

Figure 6.8

```
assign_1: where (cond_1)      ! Fortran 95
             :                ! masked by cond_1
          elsewhere (cond_2)
             :                ! masked by
             :                ! cond_2.and..not.cond_1
assign_2:    where (cond_4)
                :             ! masked by
                :             ! cond_2.and..not.cond_1.and.cond_4
             elsewhere
                :       ! masked by
                :       ! cond_2.and..not.cond_1.and..not.cond_4
             end where assign_2
             :
          elsewhere (cond_3) assign_1
             :          ! masked by
             :          ! cond_3.and..not.cond_1.and.not.cond_2
          elsewhere assign_1
             :       ! masked by
             :       ! not.cond_1.and..not.cond_2.and..not.cond_3
          end where assign_1
```

All the statements of a `where` construct are executed one by one in sequence, including the `where` and `elsewhere` statements. The logical array expressions in the `where` and `elsewhere` statements are evaluated once and control of subsequent assignments is not affected by changes to the values of these expressions. Throughout a `where` construct there is a control mask and a pending mask which change after the evaluation of each `where`, `elsewhere`, and `end where` statement, as illustrated in Figure 6.8.

6.9 The forall statement and construct (Fortran 95 only)

When a do construct such as

```
do i = 1, n
   a(i, i) = x(i)     ! a is rank-2 and x rank-1
end do
```

is executed, the processor is required to perform each successive iteration in order and one after the other. This represents a potentially severe impediment to optimization on a parallel processor so, for this purpose, Fortran 95 has the `forall` statement. The above loop can be written as

```
forall(i = 1:n) a(i, i) = x(i)       ! Fortran 95
```

which specifies that the individual assignments may be carried out in any order, and even simultaneously. The `forall` statement may be considered to be an array assignment expressed with the help of indices. In this particular example, we note also that this operation could not otherwise be represented as a simple array assignment. Other examples of the `forall` statement are

```
          ! Fortran 95
forall(i = 1:n, j = 1:m)                    a(i, j) = i + j
forall(i = 1:n, j = 1:n, y(i, j) /= 0.) x(j, i) = 1.0/y(i, j)
```

where, in the second statement, we note the masking condition — the assignment is not carried out for zero elements of y.

The `forall` construct also exists. It allows several assignment statements to be executed in order. The `forall` equivalent of the array assignments

```
a(2:n-1, 2:n-1) = a(2:n-1, 1:n-2) + a(2:n-1, 3:n)   &
                + a(1:n-2, 2:n-1) + a(3:n, 2:n-1)
b(2:n-1, 2:n-1) = a(2:n-1, 2:n-1)
```

is

```
forall(i = 2:n-1, j = 2:n-1)          ! Fortran 95
   a(i, j) = a(i, j-1) + a(i, j+1) + a(i-1, j) + a(i+1, j)
   b(i, j) = a(i, j)
end forall
```

This sets each internal element of a equal to the sum of its four nearest neighbours and copies the result to b. The `forall` version is more readable. Note that each assignment in a `forall` is like an array assignment; the effect is as if all the expressions were evaluated in any order, held in temporary storage, then all the assignments performed in any order. The first statement must fully complete before the second can begin.

A `forall` statement or construct may contain pointer assignments. An example is

```
type element
   character(32), pointer :: name
end type element
type(element)          :: chart(200)
character(32), target :: names(200)
:                       ! define names
forall(i =1:200)                    ! Fortran 95
   chart(i)%name => names(i)
end forall
```

Note that there is no array syntax for performing, as in this example, an array of pointer assignments.

As with all constructs, `forall` constructs may be nested. The sequence

```
forall (i = 1:n-1)                  ! Fortran 95
   forall (j = i+1:n)
      a(i, j) = a(j, i)             ! a is a rank-2 array
   end forall
end forall
```

assigns the transpose of the lower triangle of a to the upper triangle of a.

A `forall` construct can include a `where` statement or construct. Each statement of a `where` construct is executed in sequence. An example with a `where` statement is

```
forall (i = 1:n)                    ! Fortran 95
   where ( a(i, :) == 0) a(i, :) = i
   b(i, :) = i / a(i, :)
end forall
```

Here, each zero element of a is replaced by the value of the row index and, following this complete operation, the elements of the rows of b are assigned the reciprocals of the corresponding elements of a multiplied by the corresponding row index.

The complete syntax of the `forall` construct is

```
[name:] forall(index = lower: upper [:stride] & ! Fortran 95
          [, index = lower: upper [:stride]]... [,scalar-logical-expr] )
              [body]
          end forall [name]
```

where *index* is a named integer scalar variable. Its scope is that of the construct; that is, other variables may have the name but are separate and not accessible in the `forall`. The *index* may not be redefined within the construct. Within a nested construct, each *index* must have a distinct name. The expressions *lower*, *upper*, and *stride* (*stride* is optional but must be nonzero when present) are scalar integer expressions and form a sequence of values as for a section subscript (Section 6.13); they may not reference any *index* of the same statement but may reference

an *index* of an outer `forall`. Once these expressions have been evaluated, the *scalar-logical-expr*, if present, is evaluated for each combination of index values. Those for which it has the value `.true.` are active in each statement of the construct. The *name* is the optional construct name; if present, it must appear on both the `forall` and the `end forall` statements. The blank between the keywords `end` and `forall` is optional.

The *body* itself consists of one or more: assignment statements, pointer assignment statements, `where` statements or constructs, and further `forall` statements or constructs. The subobject on the left-hand side of each assignment in the *body* should reference each *index* of the constructs it is contained in as part of the identification of the subobject, whether it be a non-pointer variable or a pointer object.[4] None of the statements in the *body* may be a branch target, for instance for a `go to` statement.

In the case of a defined assignment statement, the subroutine that is invoked must not reference any variable that becomes defined by the statement, nor any pointer object that becomes associated.

A `forall` construct whose body is a single assignment or pointer assignment statement may be written as a single `forall` statement.

Procedures may be referenced within the scope of a `forall`, both in the logical scalar expression that forms the optional mask or, directly or indirectly (for instance as a defined operation or assignment), in the body of the construct. *All such procedures must be pure* (see Section 6.10)

As in assignments to array sections (Section 6.13), it is not allowed to make a many-to-one assignment. The construct

```
forall (i = 1:10)          ! Fortran 95
   a(index(i)) = b(i)      ! a, b and index are arrays
end forall
```

is valid if and only if `index(1:10)` contains no repeated values. Similarly, it is not permitted to associate more than one target with the same pointer.

6.10 Pure procedures (Fortran 95 only)

In the description of functions in Section 5.10, we noted the fact that, although it is permissible to write functions with side-effects, this is regarded as undesirable. In fact, used within `forall` statements or constructs (Section 6.9), the possibility that a function or subroutine reference might have side-effects is a severe impediment to optimization on a parallel processor – the order of execution of the assignments could affect the results. In order to control this situation, it is possible for the

[4]This is not actually a requirement, but any missing *index* would need to be restricted to a single value to satisfy the requirements of the final paragraph of this section. For example, the statement

```
forall (i = i1:i2, j = j1:j2) a(j) = a(j) + b(i, j)
```

is valid only if `i1` and `i2` have the same value.

programmer to assert that a procedure has no side-effects by adding the pure keyword to the subroutine or function statement. In practical terms, this is an assertion that the procedure

i) if a function, does not alter any dummy argument;

ii) does not alter any part of a variable accessed by host or use association;

iii) contains no local variable with the `save` attribute;

iv) performs no operation on an external file (Chapters 9 and 10); and

v) contains no `stop` statement.

To ensure that these requirements are met and that a compiler can easily check that this is so, there are the following further rules:

i) any dummy argument that is a procedure and any procedure referenced must be pure and have an explicit interface;

ii) the intent of a dummy argument must be declared unless it is a procedure or a pointer, and this intent must be `in` in the case of a function;

iii) any procedure internal to a pure procedure must be pure; and

iv) a variable that is accessed by host or use association or is an intent `in` dummy argument or any part of such a variable must not be the target of a pointer assignment statement; if it is of derived type with a pointer component, it must not be the right-hand side of an assignment; and it must not be associated as an actual argument with a dummy argument that is a pointer or has intent `out` or `inout`.

This last rule ensures that a local pointer cannot cause a side effect but unfortunately prevents benign uses such as aliasing (Section 6.15) for use within an expression.

The function in Figure 5.6 (Section 5.10) is pure, and this could be specified explicitly:

```
pure function distance(p, q)      ! Fortran 95
```

An external or dummy procedure that is used as a pure procedure must have an interface block that specifies it as pure. However, the procedure may be used in other contexts without the use of an interface block or with an interface block that does not specify it as pure. For example, this allows library procedures to be specified as pure without limiting them to be used as such.

The main reason for allowing pure subroutines is to be able to use a defined assignment in a `forall` statement or construct and so, unlike pure functions, they may have dummy arguments that have intent `out` or `inout` or the pointer attribute. Their existence also gives the possibility of making subroutine calls from within pure functions.

All the intrinsic functions (Chapter 8) are pure, and can thus be referenced freely within pure procedures. In addition, the elemental intrinsic subroutine `mvbits` (Section 8.8.3) is pure.

The pure attribute is given automatically to any procedure that has the `elemental` attribute (next section).

The complete set of options for the *prefix* of a function statement (Section 5.20) is

```
prefix-spec [ prefix-spec ] ...     ! Fortran 95
```

where *prefix-spec* is *type*, `recursive`, `pure`, or `elemental`. A *prefix-spec* must not be repeated. A `subroutine` statement is permitted a similar prefix, except of course that *type* must not be present.

6.11 Elemental procedures (Fortran 95 only)

We have met already the notion of elemental intrinsic procedures (Section 6.6 and, later, Chapter 8) — those with scalar dummy arguments that may be called with array actual arguments provided that the array arguments have the same shape (that is, provided all the arguments are conformable). For a function, the shape of the result is the shape of the array arguments. Fortran 95 extends this to non-intrinsic procedures. This requires the `elemental` prefix on the `function` or `subroutine` statement. For example, we could make the function `add_intervals` of Section 3.8 elemental, as shown in Figure 6.9. This is enormously useful to the programmer who can get the same effect in Fortran 90 only by writing 22 versions, for ranks 0-0, 0-1, 1-0, 1-1, 0-2, 2-0, 2-2, ... 0-7, 7-0, 7-7, and is an aid to optimization on parallel processors.

Figure 6.9

```
elemental function add_intervals(a,b)      ! Fortran 95
   type(interval)              :: add_intervals
   type(interval), intent(in) :: a, b
   add_intervals%lower = a%lower + b%lower ! Production code
   add_intervals%upper = a%upper + b%upper ! would allow for
end function add_intervals                 ! roundoff.
```

A procedure is not permitted to be both elemental and recursive.

An elemental procedure must satisfy all the requirements of a pure procedure (previous section); in fact, it automatically has the pure attribute. In addition, all dummy arguments and function results must be scalar variables without the pointer attribute. A dummy argument or its subobject may be used in a specification expression only as an argument to the intrinsic functions `bit_size`, `kind`, `len` or numeric inquiry functions of Section 8.7.2. An example is

```
elemental real function f(a)     ! Fortran 95
   real, intent(in)                              :: a
   real(selected_real_kind(precision(a)*2)) :: work
   :
end function f
```

This restriction prevents character functions yielding an array result with elements of varying character lengths and permits implementations to create array-valued versions that employ ordinary arrays internally. A simple example that would break the rule is

```
elemental function c(n)              ! Fortran 95
   character (len=n)   :: c          ! Invalid
   integer, intent(in) :: n
   real                :: work(n) ! Invalid
   :
end function c
```

If this were allowed, a rank-one version would need to hold `work` as a ragged-edge array of rank two.

An interface block for an external or dummy procedure is required if the procedure itself is non-intrinsic and elemental. The interface must specify it as elemental. This is because the compiler may use a different calling mechanism in order to accommodate the array case efficiently. It contrasts with the case of pure procedures, where more freedom is permitted (see previous section).

For an elemental subroutine, if any argument is array valued, all the arguments with intent `inout` or `out` must be arrays. For example, we can make the subroutine `swap` of Figure 6.2 (Section 6.4) perform its task on arrays of any shape or size, as shown in Figure 6.10. Calling `swap` with an array and a scalar argument is obviously erroneous and is not permitted.

Figure 6.10

```
elemental subroutine swap(a, b)     ! Fortran 95
   real, intent(inout)  :: a, b
   real                 :: work
   work = a
   a = b
   b = work
end subroutine swap
```

If a generic procedure reference (Section 5.18) is consistent with both an elemental and a non-elemental procedure, the non-elemental procedure is invoked. For example, we might write versions of `add_intervals` (Figure 6.9) for arrays of rank one and rely on the elemental function for other ranks. In general, one must expect the elemental version to execute more slowly for a specific rank than the corresponding non-elemental version.

We note that an elemental procedure may not be used as an actual argument.

6.12 Array elements

In Section 2.10, we restricted the description of array elements to simple cases. In general, an array element is a scalar of the form

part-ref [*%part-ref*] ...

where *part-ref* is

part-name[(*subscript-list*)]

The last *part-ref* must have a *subscript-list.* The number of subscripts in each list must be equal to the rank of the array or array component, and each subscript must be a scalar integer expression whose value is within the bounds of its dimension of the array or array component. To illustrate this, take the type

```
type triplet
   real                  :: u
   real, dimension(3)    :: du
   real, dimension(3,3)  :: d2u
end type triplet
```

which was considered in Section 2.10. An array may be declared of this type:

```
type(triplet), dimension(10,20,30) :: tar
```

and

```
tar(n,2,n*n)          ! n of type integer
```

is an array element. It is a scalar of type `triplet` and

```
tar(n, 2, n*n)%du
```

is a real array with

```
tar(n, 2, n*n)%du(2)
```

as one of its elements.

If an array element is of type character, it may be followed by a substring reference:

(*substring-range*)

for example,

```
page (k*k) (i+1:j-5) ! i, j, k of type integer.
```

By convention, such an object is called a substring rather than an array element.

Notice that it is the array *part-name* that the subscript list qualifies. It is not permitted to apply such a subscript list to an array designator unless the designator terminates with an array *part-name.* An array section, a function reference, or an array expression in parentheses must not be qualified by a subscript list.

6.13 Array subobjects

Array sections were introduced in Section 2.10 and provide a convenient way to access a regular subarray such as a row or a column of a rank-two array:

```
a(i, 1:n)   ! Elements 1 to n of row i
a(1:m, j)   ! Elements 1 to m of column j
```

For simplicity of description, we did not explain that one or both bounds may be omitted when the corresponding bound of the array itself is wanted, and that a stride other than one may be specified:

```
a(i, :)       ! The whole of row i
a(i, 1:n:3)   ! Elements 1, 4, ... of row i
```

Another form of section subscript is a rank-one integer expression. All the elements of the expression must be defined with values that lie within the bounds of the parent array's subscript. For example,

```
v( (/ 1, 7, 3, 2 /) )
```

is a section with elements `v(1)`, `v(7)`, `v(3)`, and `v(2)`, in this order. Such a subscript is called a *vector subscript.* If there are any repetitions in the values of the elements of a vector subscript, the section is called a *many-one section* because more than one element of the section is mapped onto a single array element. For example

```
v( (/ 1, 7, 3, 7 /) )
```

has elements 2 and 4 mapped onto `v(7)`. A many-one section must not appear on the left of an assignment statement because there would be several possible values for a single element. For instance, the statement

```
v( (/ 1, 7, 3, 7 /) ) = (/ 1, 2, 3, 4 /)      ! Illegal
```

is not allowed because the values 2 and 4 cannot both be stored in `v(7)`. The extent is zero if the vector subscript has zero size.

When an array section with a vector subscript is an actual argument, it is regarded as an expression and the corresponding dummy argument must not be defined or redefined and must not have intent `out` or `inout`. We expect compilers to make a copy as a temporary regular array on entry but to perform no copy back on return. Also, an array section with a vector subscript is not permitted to be a pointer target, since allowing them would seriously complicate the mechanism that compilers would otherwise have to establish for pointers. For similar reasons, such an array section is not permitted to be an internal file (Section 9.6).

In addition to the regular and irregular subscripting patterns just described, the intrinsic circular shift function `cshift` (Section 8.13.5) provides a mechanism that manipulates array sections in a 'wrap-round' fashion. This is useful in handling the boundaries of certain types of periodic grid problems, although it is subject to

similar restrictions to those on vector subscripts. If an array `v(5)` has the value `[1,2,3,4,5]`, then `cshift(v, 2)` has the value `[3,4,5,1,2]`.

The general form of a subobject is

part-ref[%*part-ref*] ... [(*substring-range*)]

where *part-ref* now has the form

part-name [(*section-subscript-list*)]

where the number of section subscripts in each list must be equal to the rank of the array or array component. Each *section-subscript* is either a *subscript* (Section 6.12), a rank-one integer expression (vector subscript), or a *subscript-triplet* of the form

[*lower*] : [*upper*] [: *stride*]

where *lower*, *upper*, and *stride* are scalar integer expressions. If *lower* is omitted, the default value is the lower bound for this subscript of the array. If *upper* is omitted, the default value is the upper bound for this subscript of the array. If *stride* is omitted, the default value is one. The stride may be negative so that it is possible to take, for example, the elements of a row in reverse order by specifying a section such as

```
a(i, 10:1:-1)
```

The extent is zero if *stride*>0 and *lower*>*upper*, or if *stride*<0 and *lower*<*upper*. The value of *stride* must not be zero.

Normally, we expect the values of both *lower* and *upper* to be within the bounds of the corresponding array subscript. However, all that is required is that each value actually used to select an element is within the bounds. Thus,

```
a(1, 2:11:2)
```

is legal even if the upper bound of the second dimension of `a` is only 10.

The *subscript-triplet* specifies a sequence of subscript values,

lower, lower + *stride*, *lower* + 2**stride*, ...

going as far as possible without going beyond *upper* (above it when *stride* > 0 or below it when *stride*< 0). The length of the sequence for the *i*-th *subscript-triplet* determines the *i*-th extent of the array that is formed.

The rank of a *part-ref* with a *section-subscript-list* is the number of vector subscripts and subscript triplets that it contains. So far in this section, all the examples have been of rank one; by contrast, the ordinary array element

```
a(1,7)
```

is an example of a *part-ref* of rank zero, and the section

```
a(:,1:7)
```

is an example of a *part-ref* of rank two. The rank of a *part-ref* without a *section-subscript-list* is the rank of the object or component. A *part-ref* may be an array; for example,

```
tar%du(2)
```

for the array `tar` of Section 6.12 is an array section with elements `tar(1)%du(2)`, `tar(2)%du(2)`, `tar(3)%du(2)`, ... Being able to form sections in this way from arrays of derived type, as well as by selecting sets of elements, is a very useful feature of the language. A more prosaic example, given the specification

```
type(person), dimension(1:50) :: my_group
```

for the type `person` of Section 2.9, is the subobject `my_group%id` which is an integer array section of size 50.

Unfortunately, it is not permissible for more than one *part-ref* to be an array; for example, it is not permitted to write

```
tar%du   ! Illegal
```

for the array `tar` of Section 6.12. The reason for this is that if `tar%du` were considered to be an array, its element (1,2,3,4) would correspond to

```
tar(2,3,4)%du(1)
```

which would be too confusing a notation.

The *part-ref* with nonzero rank determines the rank and shape of the subobject. If any of its extents is zero, the subobject itself has size zero. It is called an array section if the final *part-ref* has a *section-subscript-list* or another *part-ref* has a nonzero rank.

A *substring-range* may be present only if the last *part-ref* is of type character and is either a scalar or has a *section-subscript-list*. By convention, the resulting object is called a section rather than a substring. It is formed from the unqualified section by taking the specified substring of each element. Note that, if `c` is a rank-one character array,

```
c(i:j)
```

is the section formed from elements `i` to `j`; if substrings of all the array elements are wanted, we may write the section

```
c(:)(k:l)
```

An array section that ends with a component name is also called a *structure component*. Note that if the component is scalar, the section cannot be qualified by a trailing subscript list or section subscript list. Thus, using the example of Section 6.12,

```
tar%u
```

is such an array section and

```
tar(1, 2, 3)%u
```

is a component of a valid element of `tar`. The form

```
tar%u(1, 2, 3)  ! not permitted
```

is not allowed.

Additionally, a *part-name* to the right of a *part-ref* with nonzero rank must not have the `pointer` attribute. This is because such an object would represent an array of pointers and require a very different implementation mechanism from that needed for an ordinary array. For example, consider the array

```
type(entry), dimension(n) :: rows  ! n of type integer
```

for the type `entry` defined near the end of Section 6.5.2. If we were allowed to write the object `rows%next`, it would be interpreted as another array of size `n` and type `entry`, but its elements are likely to be stored without any regular pattern (each having been separately given storage by an `allocate` statement) and indeed some will be null if any of the pointers are disassociated. Note that there is no problem over accessing individual pointers such as `rows(i)%next`.

6.14 Arrays of pointers

Although arrays of pointers as such are not allowed in Fortran, the equivalent effect can be achieved by creating a type containing a pointer component. For example, a lower-triangular matrix may be held by using a pointer for each row. Consider the type

```
type row
   real, dimension(:), pointer :: r
end type row
```

and the arrays

```
type(row), dimension(n) :: s, t     ! n of type integer
```

Storage for the rows can be allocated thus

```
do i = 1, n                ! i of type integer.
   allocate (t(i)%r(1:i)) ! Allocate row i of length i.
end do
```

The array assignment

```
s = t
```

would then be equivalent to the pointer assignments

```
s(i)%r => t(i)%r
```

for all the components.

A type containing just a pointer component is useful also when constructing a linked list that is more complicated than the chain described in Section 2.13. For instance, if a variable number of links are needed at each entry, the recursive type entry of Figure 2.3 might be expanded to the pair of types:

```
type ptr
   type(entry), pointer :: point
end type ptr
type entry
   real                 :: value
   integer              :: index
   type(ptr), pointer   :: children(:)
end type entry
```

After appropriate allocations and pointer associations, it is then possible to refer to the index of child j of node as

```
node%children(j)%point%index
```

This extra level of indirection is necessary because the individual elements of children do not, themselves, have the pointer attribute – this is a property only of the whole array. For example, we can take two existing nodes, say a and b, each of which is a tree root, and make a big tree thus

```
tree%children(1)%point => a
tree%children(2)%point => b
```

which would not be possible with the original type entry.

6.15 Pointers as aliases

If an array section without vector subscripts, such as

```
table(m:n, p:q)
```

is wanted frequently while the integer variables m, n, p, and q do not change their values, it is convenient to be able to refer to the section as a named array such as

```
window
```

Such a facility is provided in Fortran by pointers and the pointer assignment statement. Here window would be declared thus

```
real, dimension(:, :), pointer :: window
```

and associated with table, which must of course have the target or pointer attribute, by the execution of the statement

```
window  => table(m:n, p:q)
```

If, later on, the size of `window` needs to be changed, all that is needed is another pointer assignment statement. Note, however, that the subscript bounds for `window` in this example are `(1:n-m+1, 1:q-p+1)` since they are as provided by the functions `lbound` and `ubound` (Section 8.12.2).

The facility provides a mechanism for subscripting or sectioning arrays such as

```
tar%u
```

where `tar` is an array and `u` is a scalar component, discussed in Section 6.13. Here we may perform the pointer association

```
taru => tar%u
```

if `taru` is a rank-three pointer of the appropriate type. Subscripting as in

```
taru(1, 2, 3)
```

is then permissible. Here the subscript bounds for `taru` will be those of `tar`.

6.16 Array constructors

The syntax that we introduced in Section 2.10 for array constants may be used to construct more general rank-one arrays. The general form of an *array-constructor* is

(/ *array-constructor-value-list* /)

where each *array-constructor-value* is one of *expr* or *constructor-implied-do*. The array thus constructed is of rank one with its sequence of elements formed from the sequence of scalar expressions and elements of the array expressions in array element order. A *constructor-implied-do* has the form

(*array-constructor-value-list, variable* = *expr1, expr2* [,*expr3*])

where *variable* is a named integer scalar variable, and *expr1*, *expr2*, and *expr3* are scalar integer expressions. Its interpretation is as if the *array-constructor-value-list* had been written

```
max ( (expr2 - expr1 + expr3)/expr3, 0 )
```

times, with *variable* replaced by *expr1*, *expr1*+*expr3*, . . ., as for the do construct (Section 4.5). A simple example is

```
(/ (i,i=1,10) /)
```

which is equal to

```
(/ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 /)
```

Note that the syntax permits nesting of one *constructor-implied-do* inside another, as in the example

```
(/ ((i,i=1,3), j=1,3) /)
```

which is equal to

```
(/ 1, 2, 3, 1, 2, 3, 1, 2, 3 /)
```

and the nesting of structure constructors within array constructors (and vice versa), for instance, for the type in Section 6.7,

```
(/ (matrix(0.0), i = 1, 100) /)
```

The sequence may be empty, in which case a zero-sized array is constructed. The scope of the *variable* is the *constructor-implied-do*. Other statements, or even other parts of the array constructor, may refer to another variable having the same name. The value of the other variable is unaffected by execution of the array constructor and is available except within the *constructor-implied-do*.

The type and type parameters of an array constructor are those of the first *expr*, and each *expr* must have the same type and type parameters. If every *expr*, *expr1*, *expr2*, and *expr3* is a constant expression, the array constructor is a constant expression.

An array of rank greater than one may be constructed from an array constructor by using the intrinsic function `reshape` (Section 8.13.3). For example,

```
reshape( source = (/ 1,2,3,4,5,6 /), shape = (/ 2,3 /) )
```

has the value

```
1   3   5
2   4   6
```

6.17 Mask arrays

Logical arrays are needed for masking in `where` statements and constructs (Section 6.8), and they play a similar role in many of the array intrinsic functions (Chapter 8). Often, such arrays are large, and there may be a worthwhile storage gain from using non-default logical types, if available. For example, some processors may use bytes to store elements of `logical(kind=1)` arrays, and bits to store elements of `logical(kind=0)` arrays. Unfortunately, there is no *portable* facility to specify such arrays, since there is no intrinsic function comparable to `selected_int_kind` and `selected_real_kind`.

Logical arrays are formed implicitly in certain expressions, usually as compiler-generated temporary variables. In

```
where (a > 0.0) a = 2.0 * a
```

or

```
if (any(a > 0.0)) then    ! any is described in Section 8.11.1.
```

the expression `a > 0.0` is a logical array. In such a case, an optimizing compiler can be expected to choose a suitable kind type parameter for the temporary array.

6.18 Summary

We have explained that arrays may have zero size and that no special rules are needed for them. A dummy array may assume its shape from the corresponding actual argument. Storage for an array may be allocated automatically on entry to a procedure and automatically deallocated on return, or the allocation may be controlled in detail by the program. Functions may be array-valued either through the mechanism of an elemental reference that performs the same calculation for each array element (in Fortran 90, for intrinsic functions only), or through the truly array-valued function. Array assignments may be masked through the use of the `where` statement and construct. Structure components may be arrays if the parent is an array or the component is an array, but not both. A subarray may either be formulated directly as an array section, or indirectly by using pointer assignment to associate it with a pointer. An array may be constructed from a sequence of expressions. A logical array may be used as a mask.

Basically, the whole of the contents of this chapter represents features new since Fortran 77, and is a hallmark of Fortran 90/95. The intrinsic functions are an important part of the array features and will be described in Chapter 8.

We conclude this chapter with a complete program, Figures 6.12 & 6.13, that illustrates the use of array expressions, array assignments, allocatable arrays, automatic arrays, and array sections. The module `linear` contains a subroutine for solving a set of linear equations, and this is called from a main program that prompts the user for the problem and then solves it.

Figure 6.12

```
module linear
   integer, parameter, public :: kind=selected_real_kind(10)
   public :: solve

contains
   subroutine solve(a, piv_tol, b, ok)
   ! arguments
      real(kind), intent(inout), dimension(:,:) :: a
                        ! The matrix a.
      real(kind), intent(in) :: piv_tol
                        ! Smallest acceptable pivot.
      real(kind), intent(inout), dimension(:) :: b
                        ! The right-hand side vector on
                        ! entry. Overwritten by the solution.
      logical, intent(out)      :: ok
                        ! True after a successful entry
                        ! and false otherwise.

   ! Local variables
      integer :: i      ! Row index.
      integer :: j      ! Column index.
      integer :: n      ! Matrix order.
      real(kind), dimension(size(b)) :: row
                        ! Automatic array needed for workspace;
                        ! size is described in Section 8.12.2.
      real(kind) :: element ! Workspace variable.

      n = size(b)
      ok = size(a, 1) == n .and. size(a, 2) == n
      if (.not.ok) then
         return
      end if

      do j = 1, n

!     Update elements in column j.
         do i = 1, j - 1
             a(i+1:n, j) = a(i+1:n, j) - a(i,j) * a(i+1:n, i)
         end do

!     Find pivot and check its size (using maxval just to
!     obtain a scalar).
         i = maxval(maxloc(abs(a(j:n, j)))) + j - 1
            ! maxval and maxloc are in Sections 8.11.1 and 8.14.
         if (abs(a(i, j)) < piv_tol) then
            ok = .false.
            return
         end if
```

Figure 6.13

```
!     If necessary, apply row interchange
         if (i/=j) then
            row = a(j, :); a(j, :) = a(i, :); a(i, :) = row
            element = b(j); b(j) = b(i); b(i) = element
         end if

!     Compute elements j+1 : n of j-th column.
         a(j+1:n, j) = a(j+1:n, j)/a(j, j)
      end do

!  Forward substitution
      do i = 1, n-1
          b(i+1:n) = b(i+1:n) - b(i)*a(i+1:n, i)
      end do

!   Back-substitution
      do j = n, 1, -1
         b(j) = b(j)/a(j, j)
         b(1:j-1) =  b(1:j-1) - b(j)*a(1:j-1, j)
      end do
   end subroutine solve
end module linear

program main
   use linear
   integer :: i, n
   real(kind), allocatable :: a(:, :), b(:)
   logical :: ok

   print *, ' Matrix order?'
   read *,  n
   allocate ( a(n, n), b(n) )
   do i = 1, n
      write(*,'(a, i2, a)') ' Elements of row ', i, ' of a?'
           ! Edit descriptors are described in Section 9.13
      read *, a(i,:)
      write(*,'(a, i2, a)') ' Component ', i, ' of b?'
      read *, b(i)
   end do

   call solve(a, maxval(abs(a))*1.0e-10, b, ok)
   if (ok) then
      write(*, '(/,a,/,(5f12.4))') ' Solution is', b
   else
      print *, ' The matrix is singular'
   end if
end program main
```

6.19 Exercises

1. Given the array declaration

```
real, dimension(50,20) :: a
```

write array sections representing

i) the first row of a;

ii) the last column of a;

iii) every second element in each row and column;

iv) as for (iii) in reverse order in both dimensions;

v) a zero-sized array.

2. Write a where statement to double the value of all the positive elements of an array z.

3. Write an array declaration for an array j which is to be completely defined by the statement

```
j = (/ (3, 5, i=1,5), 5,5,5, (i, i =  5,3,-1 ) /)
```

4. Classify the following arrays:

```
subroutine example(n, a, b)
   real, dimension(n, 10) :: w
   real                   :: a(:), b(0:)
   real, pointer          :: d(:, :)
```

5. Write a declaration and a pointer assignment statement suitable to reference as an array all the third elements of component du in the elements of the array tar having all three subscript values even (Section 6.12).

6. Given the array declarations

```
integer, dimension(100, 100), target :: l, m, n
integer, dimension(:, :), pointer    :: ll, mm, nn
```

rewrite the statements

```
l(j:k+1, j-1:k) = l(j:k+1, j-1:k) + l(j:k+1, j-1:k)
l(j:k+1, j-1:k) = m(j:k+1, j-1:k) + n(j:k+1, j-1:k) + n(j:k+1, j:k+1)
```

as they could appear following execution of the statements

```
ll => l(j:k+1, j-1:k)
mm => m(j:k+1, j-1:k)
nn => n(j:k+1, j-1:k)
```

7. Complete Exercise 1 of Chapter 4 using array syntax instead of do constructs.

8. Write a module to maintain a data structure consisting of a linked list of integers, with the ability to add and delete members of the list, efficiently.

9. Write a module that contains the example in Figure 6.7 (Section 6.7) as a module procedure and supports the defined operations and assignments that it contains.

7. Specification statements

7.1 Introduction

In the preceding chapters we have learnt the elements of the Fortran language, how they may be combined into expressions and assignments, how we may control the logic flow of a program, how to divide a program into manageable parts, and have considered how arrays may be processed. We have seen that this knowledge is sufficient to write programs, when combined with a rudimentary `print` statement and with the `end` statement.

Already in Chapters 2 to 6, we met some specification statements when declaring the type and other properties of data objects, but to ease the reader's task we did not always explain all the available options. In this chapter we fill this gap. To begin with, however, it is necessary to recall the place of specification statements in a programming language. A program is processed by a computer in (usually) three stages. In the first stage, *compilation*, the source code (text) of the program is read and processed by a program known as a *compiler* which analyses it, and generates a file containing *object code*. Each program unit of the complete program is usually processed separately. The object code is a translation of the source code into a form which can be understood by the computer hardware, and contains the precise instructions as to what operations the computer is to perform. In the second stage of processing, the object code is placed in the relevant part of the computer's storage system by a program often known as a loader which prepares it for the next stage; during this second stage, the separate program units are linked to one another, that is joined to form a complete executable program. The third stage consists of the *execution*, whereby the coded instructions are performed and the results of the computations made available.

During the first stage, the compiler requires information about the entities involved. This information is provided at the beginning of each program unit or subprogram by specification statements. The description of most of these is the subject of this chapter. The specification statements associated with procedure interfaces, including interface blocks and the `interface` statement and also the `external` statement, were explained in Chapter 5. The `intrinsic` statement is explained in Chapter 8. The statements connected with storage association (`common`, `equivalence`, and `sequence`) are deferred to Chapter 11.

7.2 Implicit typing

Many programming languages require that all typed entities have their types specified explicitly. Any data entity that is encountered in an executable statement without its type having been declared will cause the compiler to indicate an error. This, and a prohibition on mixing types, is known as *strong typing*. In the case of Fortran, an entity appearing in the code without having been explicitly typed is normally *implicitly* typed, being assigned a type according to its initial letter. The default in a program unit or an interface block is that entities whose names begin with one of the letters `i,j,...,n` are of type default integer, and variables beginning with the letters `a,b,...,h` or `o,p,...,z` are of type default real. This absence of strong typing can lead to program errors; for instance, if a variable name is misspelt, the misspelt name will give rise to a separate variable. For this reason, we recommend that implicit typing be avoided.

Implicit typing does not apply to an entity accessed by use or host association because its type is the same as in the module or the host.

If a different rule for implicit typing is desired in a given scoping unit, the `implicit` statement may be employed. For no implicit typing whatsoever, the statement

```
implicit none
```

is available (our recommendation), and for changing the mapping between the letters and the types, statements such as

```
implicit integer (a-h)
implicit real(selected_real_kind(10)) (r,s)
implicit type(entry) (u,x-z)
```

are available. The letters are specified as a list in which a set of adjacent letters in the alphabet may be abbreviated, as in `a-h`. No letter may appear twice in the implicit statements of a scoping unit and if there is an `implicit none` statement, there must be no other implicit statement in the scoping unit. For a letter not included in the implicit statements, the mapping between the letter and a type is the default mapping.

In the case of a scoping unit other than a program unit or an interface block, for example a module subprogram, the default mapping for each letter in an inner scoping unit is the mapping for the letter in the immediate host. If the host contains an `implicit none` statement, the default mapping is null and the effect may be that implicit typing is available for some letters, because of an additional `implicit` statement in the inner scope, but not for all of them. The mapping may be to a derived type even when that type is not otherwise accessible in the inner scoping unit because of a declaration there of another type with the same name.

Figure 7.1 provides a comprehensive illustration of the rules of implicit typing.

Figure 7.1

```
module example_mod
   implicit none
   :
   interface
      function fun(i)      ! i is implicitly
         integer :: fun    ! declared integer.
      end function fun
   end interface
contains
   function jfun(j)        ! All data entities must
      integer :: jfun, j   ! be declared explicitly.
      :
   end function jfun
end module example_mod
subroutine sub
   implicit complex (c)
   c = (3.0,2.0)           ! c is implicitly declared complex.
   :
contains
   subroutine sub1
      implicit integer (a,c)
      c = (0.0,0.0) ! c is host associated and of type complex
      z = 1.0       ! z is implicitly declared real.
      a = 2         ! a is implicitly declared integer.
      cc = 1.0      ! cc is implicitly declared integer.
      :
   end subroutine sub1
   subroutine sub2
      z = 2.0       ! z is implicitly declared real and is
                    ! different from the variable z in sub1.
      :
   end subroutine sub2
   subroutine sub3
   use example_mod          ! Access the integer function fun.
      q = fun(k)            ! q is implicitly declared real and
                            ! k is implicitly declared integer.
      :
   end subroutine sub3
end subroutine sub
```

The general form of the `implicit` statement is

```
implicit none
```

or

```
implicit type (letter-spec-list) [,type (letter-spec-list)]...
```

where *type* specifies the type and type parameters (Section 7.13) and each *letter-spec* is *letter* [- *letter*].

The `implicit` statement may be used for a derived type. For example, given access to the type

```
type posn
   real    :: x, y
   integer :: z
end type posn
```

and given the statement

```
implicit type(posn) (a,b), integer (c-z)
```

variables beginning with the letters a and b are implicitly typed posn and variables beginning with the letters c,d,...,z are implicitly typed integer.

An `implicit none` statement may be preceded within a scoping unit only by `use` (and `format`) statements, and other implicit statements may be preceded only by use, parameter, and format statements. We recommend that all `implicit` statements be at the start of the specifications, immediately following any `use` statements.

7.3 Declaring entities of differing shapes

So far, we have used separate type declaration statements such as

```
integer                  :: a, b
integer, dimension(10)   :: c, d
integer, dimension(8,7)  :: e
```

to declare several entities of the same type but differing shapes. In fact, Fortran permits the convenience of using a single statement. Whether or not there is a `dimension` attribute present, arrays may be declared by placing the shape information after the name of the array:

```
integer :: a, b, c(10), d(10), e(8, 7)
```

If the `dimension` attribute is present, it provides a default shape for the entities that are not followed by their own shape information, and is ignored for those that are:

```
integer, dimension(10) :: c, d, e(8, 7)
```

7.4 Named constants and constant expressions

Inside a program, we often need to define a constant or set of constants. For instance, in a program requiring repeated use of the speed of light, we might use a real variable c that is given its value by the statement

```
c = 2.99792458
```

A danger in this practice is that the value of c may be overwritten inadvertently, for instance because another programmer re-uses c as a variable to contain a different quantity, failing to notice that the name is already in use.

Another situation which can arise is that the program contains specifications such as

```
real    :: x(10), y(10), z(10)
integer :: mesh(10, 10), ipoint(100)
```

where all the dimensions are 10 or 10^2. Such specifications may be used extensively, and 10 may even appear as an explicit constant, say as a parameter in a do-construct which processes these arrays:

```
do i = 1, 10
```

Later, it may be realised that the value 20 rather than 10 is required, and the new value must be substituted everywhere the old one occurs, an error-prone undertaking.

Yet another case was met in Section 2.6, where named constants were needed for kind type parameter values.

In order to deal with all of these situations, Fortran contains what are known as *named constants*. These may never appear on the left-hand side of an assignment statement, but may be used in expressions in any way in which a literal constant may be used. A type declaration statement may be used to specify such a constant:

```
real, parameter :: c = 2.99792458
```

The value is protected, as c is now the name of a constant and may not be used as a variable name in the same scoping unit. Similarly, we may write

```
integer, parameter :: length = 10
real               :: x(length), y(length), z(length)
integer            :: mesh(length, length), ipoint(length**2)
:
do i = 1, length
```

which has the clear advantage that in order to change the value of 10 to 20 only a single line need be modified, and the new value is then correctly propagated.

In this example, the expression `length**2` appeared in one of the array bound specifications. This is a particular example of a constant expression. A *constant expression* is an expression in which each operation is intrinsic, and each primary is

i) a constant or a subobject of a constant,

ii) an array constructor whose expressions (including bounds and strides) have primaries that are constant expressions,

iii) a structure constructor whose components are constant expressions,

iv) an elemental intrinsic function reference whose arguments are constant expressions,

v) a transformational intrinsic function reference whose arguments are constant expressions,

vi) a reference to an inquiry function (Section 8.1.2) other than `present`, `associated`, or `allocated`, where each argument is either a constant expression or a variable whose type parameters or bounds inquired about are neither assumed, defined by an expression that is not constant, defined by an `allocate` statement, nor defined by a pointer assignment,

vii) an implied-do variable whose bounds and strides are constant expressions, or

viii) a constant expression enclosed in parentheses,

and where each subscript, section subscript, and substring bound is a constant expression. In Fortran 95, v) includes `null`.

Because the values of named constants are expected to be evaluated at compile time, the expressions permitted for their definition are restricted in their form. An *initialization expression* is a constant expression in which

i) the exponentiation operator must have an integer power,

ii) an elemental intrinsic function must have arguments and results of type integer or character, and

iii) of the transformational functions, only `repeat`, `reshape`, `selected_int_kind`, `selected_real_kind`, `transfer`, and `trim` are permitted.

If an initialization expression invokes an inquiry function for a type parameter or an array bound of an object, the type parameter or array bound must be specified in a prior specification statement or to the left in the same specification statement.

In the definition of a named constant we may use any initialization expression, and the constant becomes defined with the value of the expression according to the rules of intrinsic assignment. This is illustrated by the example:

```
integer, parameter :: length=10, long=selected_real_kind(12)
real, parameter    :: lsq = length**2
```

Note from this example that it is possible in one statement to define several named constants, in this case two, separated by commas.

A named constant may be an array, as in the case

```
real, dimension(3), parameter :: field = (/ 0.0, 10.0, 20.0 /)
```

For an array of rank greater than one, the `reshape` function described in Section 8.13.3 must be applied. A named constant may be of derived type, as in the case

```
type(posn), parameter :: a = posn(1.0,2.0,0)
```

for the type defined at the end of Section 7.2. Note that a subobject of a constant need not necessarily have a constant value. For example, if `i` is an integer variable, `field(i)` may have the value 0.0, 10.0, or 20.0. Note also that a constant may not be a pointer, allocatable array, dummy argument, or function result, since these are always variables.

Any named constant used in an initialization expression must either be accessed from the host, be accessed from a module, be declared in a preceding statement, or be declared to the left of its use in the same statement. An example using a constant expression including a named constant that is defined in the same statement is

```
integer, parameter :: apple = 3, pear = apple**2
```

Finally, there is an important point concerning the definition of a scalar named constant of type `character`. Its length may be specified as an asterisk and taken directly from its value, which obviates the need to count the length of a character string, and makes modifications to its definition much easier. An example of this is

```
character(len=*), parameter :: string = 'No need to count'
```

Unfortunately, there *is* a need to count when a character array is defined using an array constructor, since all the elements must be of the same length:

```
character(len=7), parameter, dimension(2) ::          &
                          c=(/'Metcalf', 'Reid   '/)
```

would not be correct if the three blanks in `'Reid   '` were removed.

The `parameter` attribute is an important means whereby constants may be protected from overwriting, and programs modified in a safe way. It should be used for these purposes on every possible occasion.

7.5 Initial values for variables

7.5.1 Initialization in type declaration statements

A variable may be assigned an initial value in a type declaration statement, simply by following the name of the variable by an initialization expression (Section 7.4), as in the examples:

```
real                :: a = 0.0
real, dimension(3) :: b = (/ 0.0, 1.2, 4.5 /)
```

The initial value is defined by the value of the corresponding expression according to the rules of intrinsic assignment. The variable automatically acquires the save attribute (Section 7.9). It must not be a dummy argument, a pointer, an allocatable array, an automatic object, or a function result.

7.5.2 The data statement

An alternative way to specify an initial value for a variable is by the data statement. It has the general form

data *object-list /value-list/* [[,] *object-list /value-list/*]...

where *object-list* is a list of variables and *implied*-do *loops*; and *value-list* is a list of scalar constants and structure constructors. A simple example is

```
real    :: a, b, c
integer :: i, j, k
data       a,b,c/1.,2.,3./, i,j,k/1,2,3/
```

in which a variable a acquires the initial value 1., b the value 2., etc.

If any part of a variable is initialized in this way, the variable automatically acquires the save atribute. The variable must not be a dummy argument, an allocatable array, an automatic object, or a function result. It may be a pointer only in Fortran 95, and the correspnding value must be null().

After any array or array section in *object-list* has been expanded into a sequence of scalar elements in array element order, there must be as many constants in each *value-list* as scalar elements in the corresponding *object-list*. Each scalar element is assigned the corresponding scalar constant.

Constants which repeat may be written once and combined with a scalar integer *repeat count* which may be a named or literal constant:

```
data i,j,k/3*0/
```

The value of the repeat count must be positive or zero. As an example consider the statement

```
data r(1:length)/length*0./
```

where r is a real array and length is a named constant which might take the value zero.

Arrays may be initialized in three different ways: as a whole, by element, or by an implied-do loop. These three ways are shown below for an array declared by

```
real :: a(5, 5)
```

Firstly, for the whole array, the statement

```
data a/25*1.0/
```

sets each element of a to 1.0.

Secondly, individual elements and sections of a may be initialized, as in

```
data a(1,1), a(3,1), a(1,2), a(3,3) /2*1.0, 2*2.0/
data a(2:5,4) /4*1.0/
```

in each of which only the four specified elements and the section are initialized. Each array subscript must be an initialization expression, as must any character substring subscript.

When the elements to be selected fall into a pattern which can be represented by do-loop indices, it is possible to write data statements a third way, like

```
data ((a(i,j), i=1,5,2), j=1,5) /15*0./
```

The general form of an implied-do loop is

(*dlist, do-var* = *expr, expr*[, *expr*])

where *dlist* is a list of array elements, scalar structure components, and implied-do loops, *do-var* is a named integer scalar variable, and each *expr* is a scalar integer expression. It is interpreted as for a do construct (Section 4.5) except that the do variable has the scope of the implied-do as in an array constructor (Section 6.16). A variable in an *expr* must be a *do-var* of an outer implied-do:

```
integer             :: j, k
integer, parameter :: l=5, l2=((l+1)/2)**2
real                :: a(l,l)
data ((a(j,k), k=1,j), j=1,l,2) / l2 * 1.0 /
```

This example sets to 1.0 the first element of the first row of a, the first three elements of the third row, and all the elements of the last row, as shown in Figure 7.2.

Figure 7.2 Result of an implied-do loop in a data statement.

1.0	.	.	.	.
.	.	.	.	.
1.0	1.0	1.0	.	.
.	.	.	.	.
1.0	1.0	1.0	1.0	1.0

The only variables permitted in subscript expressions in data statements are do indices of the same or an outer level loop, and all operations must be intrinsic.

An object of derived type may appear in a data statement. In this case, the corresponding value must be a structure constructor having an initialization expression for each component. Using the type definition of posn in Section 7.2, we can write

```
type(posn) :: position1, position2
data position1 /posn(2., 3., 0)/, position2%z /4/
```

In the examples given so far, the types and type parameters of the constants in a *value-list* have always been the same as the type of the variables in the *object-list*. This need not be the case, but they must be compatible for intrinsic assignment since the entity is initialized following the rules for intrinsic assignment. It is thus possible to write statements such as

```
data q/1/, i/3.1/, b/(0.,1.)/
```

(where b and q are real and i is integer). Integer values may be binary, octal, or hexadecimal constants (Section 2.6.1).

Each variable must either have been typed in a previous type declaration statement in the scoping unit, or its type is that associated with the first letter of its name according to the implicit typing rules of the scoping unit. In the case of implicit typing, the appearance of the name of the variable in a subsequent type declaration statement in the scoping unit must confirm the type and type parameters. Similarly, any array variable must have previously been declared as such.

No variable or part of a variable may be initialized more than once in a scoping unit.

We recommend using the type declaration statement rather than the data statement, but the data statement *must* be employed when only part of a variable is to be initialized.

7.5.3 Pointer initialization and the function null (Fortran 95 only)

In Fortran 90, the initial status of a pointer is always undefined. This is a most undesirable status since such a pointer cannot even be tested by the intrinsic function associated (Section 8.2). Fortran 95 allows pointers to be given the initial status of disassociated in a type declaration statement such as

```
real, pointer, dimension(:) :: vector => null()  ! Fortran 95
```

or a data statement

```
real, pointer, dimension(:) :: vector
data vector/ null() /                            ! Fortran 95
```

This, of course, implies the save attribute, which applies to the pointer association status. The pointer must not be a dummy argument or function result.

Our recommendation is that all pointers be so initialized to reduce the risk of bizarre effects from the accidental use of undefined pointers. This is an aid too in writing code that avoids memory leaks.

The function null is a new intrinsic function (Section 8.15), whose simple form null(), as used in the above example, is almost always suitable since the attributes are immediately apparent from the context. For example, given the type entry of Section 6.5.2, the structure constructor

```
entry (0.0, 0, null())                                    ! Fortran 95
```

is available. Also, for a pointer `vector`, the statement

```
vector => null()                                          ! Fortran 95
```

is equivalent to

```
nullify(vector)
```

The form with the argument is needed when `null` is an actual argument in a reference to a generic procedure and the type, type parameter, or rank is needed to resolve the reference (Section 5.18).

As in Fortran 90, there is no mechanism to initialize a pointer as associated.

7.5.4 Default initialization of components (Fortran 95 only)

Means are available in Fortran 95 to specify that any object of a derived type is given a default initial value for a component. The value must be specified when the component is declared as part of the type definition (Section 2.9). If the component is not a pointer, this is done in the usual way (Section 7.5.1) with the equal sign followed by an initialization expression and the rules of intrinsic assignment apply (including specifying a scalar value for all the elements of an array component). If the component is a pointer, the only initialization allowed is the pointer assignment symbol followed by `null()`.

Initialization does not have to apply to all components of a given derived type. An example for the type defined in Section 6.5.2 is

```
type entry
   real                  :: value = 2.0
   integer               :: index
   type(entry), pointer :: next => null()   ! Fortran 95
end type entry
```

Given an array declaration such as

```
type(entry), dimension(100) :: matrix
```

subobjects such as `matrix(3)%value` will have the initial value 2.0, and the reference `associated(matrix(3)%next)` will return the value false.

For an object of a nested derived type, the initializations associated with components at all levels are recognized. For example, given the specifications

```
type node
   integer     :: counter
   type(entry) :: element
end type node
type (node) :: n
```

the component n%element%value will have the initial value 2.0.

Unlike explicit initialization in a type declaration or data statement, default initialization does not imply that the objects have the save attribute. However, an object of such a type that is declared in a module is required to have the save attribute unless it is a pointer or an allocatable array. This is because of the difficulty that some implementations would have with determining when a non-saved object would need to be re-initialized.

Objects may still be explicitly initialized in a type declaration statement, as in

```
type(entry), dimension(100) :: matrix=entry(huge(0.0),   &
                                huge(0),null()) ! Fortran 95
```

in which case the default initialization is ignored. Similarly, default initialization may be overridden in a nested type definition such as

```
type node
   integer     :: counter
   type(entry) :: element=entry(0.0, 0 , null()) ! Fortran 95
 end type node
```

However, no part of a non-pointer object with default initialization is permitted in a data statement (subsection 7.5.2).

As well as applying to the initial values of static data, default initialization also applies to any data that is dynamically created during program execution. This includes allocation with the allocate statement. For example, the statement

```
allocate (matrix(1)%next)
```

creates a partially initialized object of type entry. It also applies to automatic objects and to dummy arguments with intent out. It applies even if the type definition is private or the components are private.

7.6 The public and private attributes

Modules (Section 5.5) permit specifications to be 'packaged' into a form that allows them to be accessed elsewhere in the program. So far, we have assumed that all the entities in the module are to be accessible, that is have the public attribute, but sometimes it is desirable to limit the access. For example, several procedures in a module may need access to a work array containing the results of calculations that they have performed. If access is limited to only the procedures of the module, there is no possibility of an accidental corruption of these data by another procedure and design changes can be made within the module without affecting the rest of the program. In cases where entities are not to be accessible outside their own module, they may be given the private attribute.

These two attributes may be specified with the public and private attributes on type declaration statements in the module, as in

```
real, public     :: x, y, z
integer, private :: u, v, w
```

or in public and private statements, as in

```
public  :: x, y, z, operator(.add.)
private :: u, v, w, assignment(=), operator(*)
```

which have the general forms

```
public [ [ :: ] access-id-list]
private [ [ :: ] access-id-list]
```

where *access-id* is a name or a *generic-spec* (Section 5.18).

Note that if a procedure has a generic identifier, the accessibility of its specific name is independent of the accessibility of its generic identifier. One may be public while the other is private, which means that it is accessible only by its specific name or only by its generic identifier.

If a public or private statement has no list of entities, it confirms or resets the default. Thus the statement

```
public
```

confirms public as the default value, and the statement

```
private
```

sets the default value for the module to private accessibility. For example,

```
private
public :: means
```

gives the entity means the public attribute whilst all others are private. There may be at most one accessibility statement without a list in a scoping unit.

The entities that may be specified by name in public or private lists are named variables, procedures (including generic procedures), derived types, named constants, and namelist groups. Thus, to make a generic procedure name accessible but the corresponding specific names inaccessible, we might write

```
module example
   private specific_int, specific_real
   interface generic_name
      module procedure specific_int, specific_real
   end interface
contains
   subroutine specific_int(i)
      :
   subroutine specific_real(a)
      :
end module example
```

An entity with a type must not have the `public` attribute if its type is declared with the `private` attribute (because there is virtually nothing one can do with such an entity without access to its type). As we shall see in Section 7.10, the `public` and `private` attributes may be redefined for an entity accessed by use association but, here, we refer to the original declaration in such a context. This restriction is not needed when such a type is given the `private` attribute in another module that uses the first, since a subsequent `use` statement referencing an object of that type may be accompanied by an additional `use` statement for the original module that defines the type.

For similar reasons, if a module procedure has a dummy argument or function result of a type declared with the `private` attribute, the procedure must be given the attribute `private` and must not have a generic identifier that is `public`.

The use of the `private` statement for components of derived types in the context of defining an entity's access within a module will be described in Section 7.11.

The `public` and `private` attributes may appear only in the specifications of a module.

7.7 The pointer, target, and allocatable statements

For the sake of regularity in the language, there are statements for specifying the `pointer`, `target`, and `allocatable` attributes of entities. They take the forms:

```
pointer [::] object-name[(array-spec)]
                  [,object-name [(array-spec)]]...
target [::] object-name[(array-spec)]
                  [,object-name [(array-spec)]]...
```

and

```
allocatable [::] array-name[(array-spec)]
                           [,array-name [(array-spec)]]...
```

as in

```
real        ::a, son, y
allocatable :: a(:,:)
pointer     :: son
target      :: a, y(10)
```

We believe that it is much clearer to specify these attributes on the type declaration statements, and therefore do not use these forms.

7.8 The intent and optional statements

The `intent` attribute (Section 5.9) for a dummy argument that is not a dummy procedure or pointer may be specified in a type declaration statement or in an `intent` statement of the form

`intent(` *inout* `) [::]` *dummy-argument-name-list*

where *inout* is `in`, `out`, or `inout`. Examples are

```
subroutine solve (a, b, c, x, y, z)
   real        :: a, b, c, x, y, z
   intent(in)  :: a, b, c
   intent(out) :: x, y, z
```

The `optional` attribute (Section 5.13) for a dummy argument may be specified in a type declaration statement or in an `optional` statement of the form

`optional [::]` *dummy-argument-name-list*

An example is

```
optional :: a, b, c
```

The `optional` attribute is the only attribute which may be specified for a dummy argument that is a procedure.

Note that the `intent` and `optional` attributes may be specified only for dummy arguments.

7.9 The save attribute

Let us suppose that we wish to retain the value of a local variable in a subprogram, for example to count the number of times the subprogram is entered. We might write a section of code as in Figure 7.3. In this example, the local variables, `a` and `counter`, are initialized to zero, and it is assumed that their current values are available each time the subroutine is called. This is not necessarily the case. Fortran allows the computer system being used to 'forget' a new value, the variable becoming undefined on each return unless it has the `save` attribute. In Figure 7.3, it is sufficient to change the declaration of `a` to

```
real, save :: a
```

to be sure that its value is always retained between calls. This may be done for `counter`, too, but is not necessary as all variables with initial values acquire the `save` attribute automatically (Section 7.5).

A similar situation arises with the use of variables in modules (Section 5.5). On return from a subprogram that accesses a variable in a module, the variable becomes undefined unless the main program accesses the module, another subprogram in execution accesses the module, or the variable has the `save` attribute.

If a variable that becomes undefined has a pointer associated with it, the pointer's association status becomes undefined.

The `save` attribute must not be specified for a dummy argument, a function result, or an automatic object (Section 6.4). It may be specified for a pointer, in which case the pointer association status is saved. It may be specified for an

Figure 7.3

```
subroutine any(x)
   real    :: a, x
   integer :: counter = 0 ! Initialize the counter
   :
   counter = counter + 1
   if (counter==1) then
      a = 0.0
   else
      a = a + x
   end if
   :
```

allocatable array, in which case the allocation status and value are saved. A saved variable in a recursive subprogram is shared by all instances of the subprogram.

An alternative to specifying the save attribute on a type declaration statement is the save statement:

save [[::] *variable-name-list*]

A save statement with no list is equivalent to a list containing all possible names, and in this case the scoping unit must contain no other save statements and no save attributes in type declaration statements. Our recomendation is against this form of save. If a programmer tries to give the save attribute explicitly to an automatic object, a diagnostic will result. On the other hand, he or she might think that save without a list would do this too, and not get the behaviour intended. Also, there is a loss of efficency associated with save on some processors, so it is best to restrict it to those objects for which it is really needed.

The save statement or save attribute may appear in the declaration statements in a main program but has no effect.

7.10 The use statement

In Section 5.5, we introduced the use statement in its simplest form

use *module-name*

which provides access to all the public named data objects, derived types, interface blocks, procedures, generic identifiers, and namelist groups in the module named. Any use statements must precede other specification statements in a scoping unit. The only attribute of an accessed entity that may be specified afresh is public or private (and this only in a module), but the entity may be included in one or more namelist groups (Section 7.15).

If access is needed to two or more modules that have been written independently, the same name may be used in more than one module. This is the main

reason for permitting accessed entities to be renamed by the use statement. Renaming is also available to resolve a name clash between a local entity and an entity accessed from a module, though our preference is to use a text editor or other tool to change the local name. With renaming, the use statement has the form

use *module-name*, *rename-list*

where each *rename* has the form

local-name => *use-name*

and refers to a public entity in the module that is to be accessed by a different local name.

As an example,

```
use stats_lib, sprod => prod
use maths_lib
```

makes all the public entities in both stats_lib and maths_lib accessible. If maths_lib contains an entity called prod, it is accessible by its own name while the entity prod of stats_lib is accessible as sprod.

Renaming is not needed if there is a name clash between two entities that are not required. A name clash is permitted if there is no reference to the name in the scoping unit.

A name clash is also permissible for a generic name that is required. Here, all generic interfaces accessed by the name are treated as a single concatenated interface block. This is true also for defined operators and assignments, where no renaming facility is available. In all these cases, any two procedures having the same generic identifier must differ as explained in Section 5.18. We imagine that this will usually be exactly what is needed. For example, we might access modules for interval arithmetic and matrix arithmetic, both needing the functions sqrt, sin, etc., the operators +, -, etc., and assignment, but for different types.

For cases where only a subset of the names of a module is needed, the only option is available, having the form

use *module-name*, only : [*only-list*]

where each *only* has the form

access-id

or

[*local-name* =>] *use-name*

where each *access-id* is a public entity in the module, and is either a *use-name* or a *generic-spec* (Section 5.18). This provides access to an entity in a module only if the entity is public and is specified as a *use-name* or *access-id*. Where a *use-name* is preceded by a *local-name*, the entity is known locally by the *local-name*. An example of such a statement is

```
use stats_lib, only : sprod => prod, mult
```

which provides access to prod by the local name sprod and to mult by its own name.

We would recommend that only one use statement for a given module be placed in a scoping unit, but more are allowed. If there is a use statement without an only qualifier, all public entities in the module are accessible and the *rename-list*s and *only-list*s are interpreted as if concatenated into a single *rename-list* (with the form *use-name* in an only list being treated as the rename *use-name* => *use-name*). If all the statements have the only qualification, only those entities named in one or more of the *only-list*s are accessible, that is all the *only-list*s are interpreted as if concatenated into a single *only-list*.

The form

```
use module-name, only :
```

might appear redundant. It is provided for the situation where a scoping unit calls a set of procedures that communicate with each other through shared data in a module. It ensures that the data are available throughout the execution of the scoping unit.

An only list will be rather clumsy if almost all of a module is wanted. The effect of an 'except' clause can be obtained by renaming unwanted entities. For example, if a large program (such as one written in Fortran 77) contains many external procedures, a good practice is to collect interface blocks for them all into a module that is referenced in each program unit for complete mutual checking. In an external procedure, we might then write:

```
use all_interfaces, except_this_one => name
```

to avoid having two explicit interfaces for itself (where all_interfaces is the module name and name is the procedure name).

When a module contains use statements, the entities accessed are treated as entities in the module. They may be given the private or public attribute explicitly or through the default rule in effect in the module.

An entity may be accessed by more than one local name. This is illustrated in Figure 7.4, where module b accesses s of module a by the local name bs; if a subprogram such as c accesses both a and b, it will access s by both its original name and by the name bs. Figure 7.4 also illustrates that an entity may be accessed by the same name by more than one route (see variable t).

A more direct way for an entity to be accessed by more than one local name is for it to appear more than once as a *use-name*. This is not a practice that we recommend.

Of course, all the local names of entities accessed from modules must differ from each other and from names of local entities. If a local entity is accidentally given the same name as an accessible entity from a module, this will be noticed at compile time if the local entity is declared explicitly (since no accessed entity may given any attribute locally, other than private or public, and that only in a

Figure 7.4

```
module a
   real :: s, t
   :
end module a
module b
   use a, bs => s
   :
end module b
subroutine c
   use a
   use b
   :
end subroutine c
```

module). However, if the local entity is intended to be implicitly typed (Section 7.2) and appears in no specification statements, then each appearance of the name will be taken, incorrectly, as a reference to the accessed variable. To avoid this, we recommend the use of

```
implicit none
```

in a scoping unit containing one or more use statements. For greater safety, the only option may be employed on a use statement to ensure that all accesses are intentional.

7.11 Derived-type definitions

When derived types were introduced in Section 2.9, some simple example definitions were given, but the full generality was not included. An example illustrating more features is

```
type, public :: lock
   private
   integer, pointer :: key(:)
   logical          :: state
end type lock
```

The general form (apart from redundant features, see Sections 11.2 and C.1.3) is

```
type [[,access]:: ] type-name
    [ private ]
    component-def-stmt
    [component-def-stmt] ...
end type [ type-name ]
```

Each *component-def-stmt* has the form

type [[,*component-attr-list*] ::]*component-decl-list*

where *type* specifies the type and type parameters (Section 7.13), each *component-attr* is either `pointer` or `dimension`(*bounds-list*), and each *component-decl* is

component-name [(*bounds-list*)] [**char-len*]

or (Fortran 95 only)

component-name [(*bounds-list*)] [**char-len*] [*comp-int*]

The meaning of **char-len* is explained in Section 7.13. If the *type* is a derived type and the `pointer` attribute is not specified, the type must be previously defined in the host scoping unit or accessible there by use or host association. If the `pointer` attribute is specified, the type may also be the one being defined (for example, the type `entry` of Section 2.13), or one defined elsewhere in the scoping unit.

A *type-name* must not be the same as the name of any intrinsic type or a derived type accessed from a module.

The bounds of an array component are declared by a *bounds-list* where each *bounds* is

:

for a pointer component (see example in Section 6.14) or

[*lower-bound*:] *upper-bound*

for a non-pointer component, and *lower-bound* and *upper-bound* are constant expressions that are restricted to specification expressions (Section 7.14). Similarly, the character length of a component of type character must be a constant specification expression. If there is a *bounds-list* attached to the *component-name*, this defines the bounds. If a `dimension` attribute is present in the statement, its *bounds-list* applies to any component in the statement without its own *bounds-list.*

Only if the host scoping unit is a module may the *access* qualifier or `private` statement appear. The *access* qualifier on a type statement may be `public` or `private` and specifies the accessibility of the type. If it is `private`, then the type name, the structure constructor for the type, any entity of the type, and any procedure with a dummy argument or function result of the type are all inaccessible outside the host module. The accessibility may also be specified in a `private` or `public` statement in the host. In the absence of both of these, the type takes the default accessibility of the host module. If a `private` statement appears for a type with `public` accessibility, the components of the type are inaccessible in any scoping unit accessing the host module, so that neither component selection nor structure construction are available there. Also, if any component is of a derived type that is `private`, the type being defined must be `private` or have `private` components.

We can thus distinguish three levels of access:

i) all `public`, where the type and all its components are accessible;

ii) a `public` type with `private` components, where the type is accessible but its components are hidden;

iii) all `private`, where both the type and its components are used only within the host module, and are hidden to an accessing procedure.

Case ii) has, where appropriate, the advantage of enabling changes to be made to the type without in any way affecting the code in the accessing procedure. Case iii) offers this advantage and has the additional merit of not cluttering the name space of the accessing procedure. The use of `private` accessibility for the components or for the whole type is thus recommended whenever possible.

We note that even if two derived-type definitions are identical in every respect except their names, that entities of those two types are *not* equivalent and are regarded as being of different types. Even if the names, too, are identical, the types are different (unless they have the `sequence` attribute, a feature that we do not recommend and whose description is left to Section 11.2.1). If a type is needed in more than one program unit, the definition should be placed in a module and accessed by a `use` statement wherever it is needed. Having a single definition is far less prone to errors.

7.12 The type declaration statement

We have already met many simple examples of the declarations of named entities by `integer`, `real`, `complex`, `logical`, `character`, and `type(`*type-name*`)` statements. The general form is

type [[, *attribute*]... ::] *entity-list*

where *type* specifies the type and type parameters (Section 7.13), *attribute* is one of the following

`parameter`	`dimension(`*bounds-list*`)`
`public`	`intent(`*inout*`)`
`private`	`optional`
`pointer`	`save`
`target`	`external`
`allocatable`	`intrinsic`

and each *entity* is

object-name [(*bounds-list*)] [**char-len*] [=*initialization-expr*]

or

function-name [**char-len*]

or (Fortran 95 only)

pointer-name [(*bounds-list*)] [**char-len*] [=>null()]

The meaning of **char-len* is explained in Section 7.13; a *bounds-list* specifies the rank and possibly bounds of array-valued entities.

No attribute may appear more than once in a given type declaration statement. The double colon :: need not appear in the simple case without any *attributes* and without any =*initialization-expr*; for example

```
real a, b, c(10)
```

If the statement specifies a `parameter` attribute, =*initialization-expr* must appear.

If a pointer attribute is specified, the `target`, `intent`, `external`, and `intrinsic` attributes must not be specified. The `target` and `parameter` attributes may not be specified for the same entity, and the `pointer` and `allocatable` attributes may not be specified for the same array. If the `target` attribute is specified, neither the `external` nor the `intrinsic` attribute may also be specified.

If an object is specified with the `intent` or `parameter` attribute, this is shared by all its subobjects. The `pointer` attribute is not shared in this manner, but note that a derived-data type component may itself be a pointer. However, the `target` attribute is shared by all its subobjects, except for any that are pointer components.

The `allocatable`, `parameter`, or `save` attribute must not be specified for a dummy argument or function result.

The `intent` and `optional` attributes may be specified only for dummy arguments.

For a function result, specifying the `external` attribute is an alternative to the external statement (Section 5.11) for declaring the function to be external, and specifying the `intrinsic` attribute is an alternative to the intrinsic statement (Section 8.1.3) for declaring the function to be intrinsic. These two attributes are mutually exclusive.

Each of the attributes may also be specified in statements (such as `save`) that list entities having the attribute. This leads to the possibility of an attribute being specified explicitly more than once for a given entity, but this is not permitted. Our recommendation is to avoid such statements because it is much clearer to have all the attributes for an entity collected in one place.

7.13 Type and type parameter specification

We have used *type* to represent one of the following

```
integer [( [kind=] kind-value)]
real    [( [kind=] kind-value)]
complex [( [kind=] kind-value)]
character [(actual-parameter-list)]
logical [([kind=] kind-value) ]
type ( type-name )
```

in the `function` statement (Section 5.20), the `implicit` statement (Section 7.2), the component definition statement (Section 7.11), and the type declaration statement (Section 7.12). A *kind-value* must be an initialization expression (Section 7.4) and must have a value that is valid on the processor being used.

For `character`, each *actual-parameter* has the form

`[len=]` *len-value*

or

`[kind=]` *kind-value*

and provides a value for one of the parameters. It is permissible to omit `kind=` from a kind *actual-parameter* only when `len=` is omitted and *len-value* is both present and comes first, just as for an actual argument list (Section 5.13). Neither parameter may be specified more than once.

For a scalar named constant or for a dummy argument of a subprogram, a *len-value* may be specified as an asterisk, in which case the value is assumed from that of the constant itself or the associated actual argument. In both cases, the `len` intrinsic function (Section 8.6.1) is available if the actual length is required directly, for instance as a do construct iteration count. A combined example is

```
character(len=len(char_arg)) function line(char_arg)
   character(len=*)              :: char_arg
   character(len=*), parameter :: char_const = 'page'
   if ( len(char_arg) < len(char_const) ) then
   :
```

A *len-value* that is not an asterisk must be a specification expression (Section 7.14). Negative values declare character entities to be of zero length.

In addition, it is possible to attach an alternative form of *len-value* to individual entities in a type declaration statement using the syntax *entity*char-len*, where *char-len* is either (*len-value*) or *len* and *len* is a scalar integer literal constant which specifies a length for the entity. The constant *len* must not have a kind type parameter specified for it. An illustration of this form is

```
character(len=8) :: word(4), point*1, text(20)*4
```

here, `word`, `point` and `text` have character length 8, 1, and 4, respectively. Similarly, the alternative form may be used for individual components in a component definition statement.

7.14 Specification expressions

Non-constant scalar integer expressions may be used to specify the array bounds (examples in Section 6.4) and character lengths of data objects in a subprogram, and of function results. Such an expression may depend only on data values that are defined on entry to the subprogram. Any variable referenced must not have

its type and type parameters specified later in the same sequence of specification statements, unless they are those implied by the implicit typing rules.

Array constructors and derived-type constructors are permitted. The expression may reference an inquiry function for an array bound or for a type parameter of an entity which either is accessed by use or host association, or is specified earlier in the same specification sequence, but not later in the sequence[1]. An element of an array specified in the same specification sequence can be referenced only if the bounds of the array are specified earlier in the sequence[2]. Such an expression is called a *specification expression.*

An array whose bounds are declared using specification expressions is called an *explicit-shape array.*

A variety of possibilities are shown in Figure 7.5.

Figure 7.5

```
subroutine sample(arr, value, string)
   use definitions  ! Contains the real a
   real, dimension(:,:), intent(out) :: arr ! Assumed-shape array
   integer, intent(in)               :: value
   character(len=*), intent(in)      :: string ! Assumed length
   real, dimension(ubound(arr, 1)+5) :: x      ! Automatic array
   character(len=value+len(string))  :: cc     ! Automatic object
   integer, parameter :: pa2 =   &
                            selected_real_kind(2*precision(a))
   real(kind=pa2) :: z   ! Precision of z is at least twice
                         ! the precision of a
   :
```

The bounds and character lengths are not affected by any redefinitions or undefinitions of variables in the expressions during execution of the procedure.

7.14.1 Specification expression restrictions (Fortran 90 only)

In Fortran 90, references to non-intrinsic procedures are not permitted and intrinsic function references are limited to:

- an elemental function reference for which the arguments and result are of type integer or character,

[1]This avoids such a case as

```
character (len=len(a)) ::  fun
character (len=len(fun)) ::  a
```

[2]This avoids such a case as

```
integer, parameter, dimension (j(1):j(1)+1) ::  i = (/0,1/)
integer, parameter, dimension (i(1):i(1)+1) ::  j = (/1,2/)
```

- a reference to `repeat`, `trim`, `transfer`, or `reshape` for which the arguments are of type integer or character,
- a reference to `selected_int_kind` or `selected_real_kind`,
- a reference to an inquiry function other than `present`, `associated`, or `allocated`, provided the quantity inquired about does not depend on an allocation or on a pointer assignment.

7.14.2 Specification functions (Fortran 95 only)

In Fortran 95, any of the intrinsic functions defined by the standard may be used in a specification expression. In addition, a non-intrinsic pure function may be used provided that such a function is neither an internal function nor recursive, it does not have a dummy procedure argument, and the interface is explicit. Functions that fulfil these conditions are termed *specification functions*. The arguments of a specification function when used in a specification expression are subject to the same restrictions as those on specification expressions themselves, except that they do not necessarily have to be scalar.

As the interfaces of specification functions must be explicit yet they cannot be internal functions,[3] such functions are probably most conveniently written as module procedures.

This feature will be a great convenience for specification expressions that cannot be written as simple expressions. Here is an example,

```
function solve (a, ...              ! Fortran 95
   use matrix_ops
   type(matrix), intent(in) :: a
   :
   real                     :: work(wsize(a))
   :
```

where `matrix` is a type defined in the module `matrix_ops` and intended to hold a sparse matrix and its LU factorization:

```
type matrix       ! Fortran 95
   integer :: n  ! Matrix order.
   integer :: nz ! Number of nonzero entries.
   logical :: new = .true. ! Whether this is a new,
                           ! unfactorized matrix.
      :
end type matrix
```

and `wsize` is a module procedure that calculates the required size of the array `work`:

[3]This prevents them enquiring, via host association, about objects being specified in the set of statements in which the specification function itself is referenced.

```
pure integer function wsize(a)  ! Fortran 95
   type(matrix), intent(in) :: a
   wsize = 2*a%n + 2
   if(a%new) wsize = a%nz + wsize
end function wsize
```

7.15 The namelist statement

It is sometimes convenient to gather a set of variables into a single group, in order to facilitate input/output (I/O) operations on the group as a whole. The actual use of such groups is explained in Section 9.10. The method by which a group is declared is via the `namelist` statement which in its simple form has the syntax

`namelist` *namelist-spec*

where *namelist-spec* is

/namelist-group-name/ variable-name-list

The *namelist-group-name* is the name given to the group for subsequent use in the I/O statements. A variable named in the list must not be a dummy array with a non-constant bound, a variable with non-constant character length, an automatic object, an allocatable array, a pointer, or have a component at any depth of component selection that is a pointer or is inaccessible. An example is

```
real :: carpet, tv, brushes(10)
namelist /household_items/ carpet, tv, brushes
```

It is possible to declare several namelist groups in one statement, with the syntax

`namelist` *namelist-spec* `[[,]`*namelist-spec*`]...`

as in the example

```
namelist /list1/ a, b, c /list2/ x, y, z
```

It is possible to continue a list within the same scoping unit by repeating the namelist name on more than one statement. Thus,

```
namelist /list/ a, b, c
namelist /list/ d, e, f
```

has the same effect as a single statement containing all the variable names in the same order. A namelist group object may belong to more than one namelist group.

If the type, type parameters, or shape of a namelist variable is specified in a specification statement in the same scoping unit, the specification statement must either appear before the `namelist` statement, or be a type declaration statement that confirms the implicit typing rule in force in the scoping unit for the initial letter of the variable. Also, if the namelist group has the `public` attribute, no variable in the list may have the `private` attribute or have private components.

Figure 7.6

```
module sort              ! To sort postal addresses by zip code.
   implicit none
   private
   public :: selection_sort
   integer, parameter :: string_length = 30
   type, public :: address
      character(len = string_length) :: name, street, town, &
                                        state*2
      integer                        :: zip_code
   end type address
contains
   recursive subroutine selection_sort (array_arg)
      type (address), dimension (:), intent (inout)          &
                                          :: array_arg
      integer                             :: current_size
      integer, dimension (1)              :: big
           ! Result of maxloc (Section 8.14) is array valued
      current_size = size (array_arg)
      if (current_size > 0) then
         big = maxloc (array_arg(:)%zip_code)
         call swap (big(1), current_size)
         call selection_sort (array_arg(1: current_size - 1))
      end if
   contains
      subroutine swap (i, j)
         integer, intent (in) :: i, j
         type (address)       :: temp
         temp = array_arg(i)
         array_arg(i) = array_arg(j)
         array_arg(j) = temp
      end subroutine swap
   end subroutine selection_sort
end module sort
```

Figure 7.7

```
program zippy
   use sort
   implicit none
   integer, parameter                          :: array_size = 100
   type (address), dimension (array_size) :: data_array
   integer                                          :: i, n
   do i = 1, array_size
      read  (*, '(/a/a/a/a2,i8)', end=10) data_array(i)
                                 ! For end= see Section 9.7;
      write (*, '(/a/a/a/a2,i8)')          data_array(i)
   end do                        ! for editing see Section 9.13.
10 n = i - 1
   call selection_sort (data_array(1: n))
   write(*, '(//a)') 'after sorting:'
   do i = 1, n
      write (*, '(/a/a/a/a2,i8)') data_array(i)
   end do
end program zippy
```

7.16 Summary

In this chapter most of the specification statements of Fortran have been described. The following concepts have been introduced: implicit typing and its attendant dangers, named constants, constant expressions, data initialization, control of the accessibility of entities in modules, saving data between procedure calls, selective access of entities in a module, renaming entities accessed from a module, specification expressions that may be used when specifying data objects and function results, and the formation of variables into namelist groups. We have also explained alternative ways of specifying attributes.

The features described here that are new since Fortran 77 are `implicit none`; initialization and specification expressions; a much extended type declaration statement; `data` statement extended to include derived types, subobjects, and binary, octal, and hexadecimal constants; new attributes and statements: `public`, `private`, `pointer`, `allocatable`, `target`, `intent`, and `optional`; and the `use` and `namelist` statements.

We conclude this chapter with a complete program, Figures 7.6 and 7.7, that uses a module to sort US-style addresses (name, street, town, and state with a numerical zip code) by order of zip code. It illustrates the interplay between many of the features described so far, but note that it is not a production code since the sort routine is not very efficient and the full range of US addresses is not handled. Suitable test data are:

```
Prof. James Bush,
206 Church St. SE,
Minneapolis,
MN 55455

J. E. Dougal,
Rice University,
Houston,
TX 77251

Jack Finch,
104 Ayres Hall,
Knoxville,
TN 37996
```

7.17 Exercises

1. Write suitable type statements for the following quantities:

i) an array to hold the number of counts in each of the 100 bins of a histogram numbered from 1 to 100;

ii) an array to hold the temperature to two significant decimal places at points, on a sheet of iron, equally spaced at 1cm intervals on a rectangular grid 20cm square, with points in each corner (the melting point of iron is 1530° C);

iii) an array to describe the state of 20 on/off switches;

iv) an array to contain the information destined for a printed page of 44 lines each of 70 letters or digits.

2. Explain the difference between the following pair of declarations

```
real :: i = 3.1
```

and

```
real, parameter :: i = 3.1
```

What is the value of `i` in each case?

3. Write type declaration statements which initialize:

i) all the elements of an integer array of length 100 to the value zero.

ii) all the odd elements of the same array to 0 and the even elements to 1.

iii) the elements of a real 10×10 square array to 1.0 .

iv) a character string to the digits 0 to 9.