INTRODUÇÃO AO FORTRAN MODERNO

RUDI GAELZER (INSTITUTO DE FÍSICA - UFRGS)

Apostila preparada para as disciplinas de Métodos Computacionais da Física, ministradas para os Cursos de Bacharelado em Física e Engenharia Física do Instituto de Física da Universidade Federal do Rio Grande do Sul, Porto Alegre - RS.

Início: JANEIRO DE 2005

Impresso: 24 de outubro de 2019

Apostila escrita usando:
Processador de Documentos LyX
http://www.lyx.org/
http://wiki.lyx.org/LyX/LyX

Sumário

1 Introdução **1**
1.1 As origens da Linguagem Fortran . . . 1
1.2 O padrão Fortran 90 . . . 2
1.2.1 Procedimentos adotados para a evolução da linguagem . . . 3
1.2.2 Recursos novos do Fortran 90 . . . 3
1.2.3 Recursos em obsolescência no Fortran 90 . . . 4
1.2.4 Recursos removidos do Fortran 90 . . . 4
1.3 Uma revisão menor: Fortran 95 . . . 5
1.3.1 Recursos novos do Fortran 95 . . . 5
1.3.2 Recursos em obsolescência no Fortran 95 . . . 6
1.3.3 Recursos removidos do Fortran 95 . . . 6
1.4 O Fortran no Século XXI: Fortran 2003 . . . 6
1.4.1 Recursos novos do Fortran 2003 . . . 7
1.4.2 Recursos em obsolescência no Fortran 2003 . . . 7
1.4.3 Recursos removidos do Fortran 2003 . . . 7
1.4.4 Status de suporte ao padrão Fortran 2003 . . . 8
1.5 O padrão Fortran 2008 . . . 8
1.5.1 Recursos novos do Fortran 2008 . . . 8
1.5.2 Recursos em obsolescência no Fortran 2008 . . . 9
1.5.3 Recursos removidos do Fortran 2008 . . . 10
1.5.4 Status do suporte ao Fortran 2008 . . . 10
1.6 O novo padrão: Fortran 2018 . . . 10
1.6.1 Recursos novos do Fortran 2018 . . . 10
1.6.2 Recursos em obsolescência no Fortran 2018 . . . 10
1.6.3 Recursos removidos do Fortran 2018 . . . 11
1.6.4 Status do suporte ao Fortran 2018 . . . 11
1.7 Recursos adicionais ao Fortran Moderno . . . 11
1.8 Comentários sobre a bibliografia . . . 12
1.9 Observações sobre a apostila e agradecimentos . . . 12

2 Elementos Básicos do Fortran **15**
2.1 A sintaxe da linguagem . . . 15
2.2 Vocábulos léxicos *(tokens)* . . . 16
2.3 Nomes válidos em Fortran . . . 17
2.4 Formatação do programa-fonte . . . 17
2.5 O programa “Alô Mamãe” . . . 18
2.6 Entrada e saída padrões . . . 19

3 Tipos de Objetos de Dados **21**
3.1 Tipos intrínsecos de dados . . . 22
3.1.1 Tipo `INTEGER` . . . 22
3.1.2 Tipo `REAL` . . . 22
3.1.3 Tipo `COMPLEX` . . . 23
3.1.4 Tipo `CHARACTER` . . . 24
3.1.5 Tipo `LOGICAL` . . . 25
3.2 Forma geral da declaração de tipo de objeto de dados . . . 26
3.3 O conceito de espécie *(kind)* . . . 27
3.3.1 Compilador `Intel® Fortran` . . . 27
3.3.2 Compilador `gfortran` . . . 29

3.3.3 Literais de diferentes espécies . . . 29
3.3.4 Recursos relacionados às espécies de tipo . . . 31
3.3.4.1 Funções intrínsecas associadas à espécie . . . 31
3.3.4.2 Nomes fixos para as espécies mais comuns . . . 34
3.3.4.3 Inquirição sobre o parâmetro de tipo . . . 35
3.4 Tipos derivados . . . 35
3.4.1 Uso básico . . . 37
3.4.2 Forma geral da definição de tipo derivado . . . 39
3.4.2.1 Configurações de acesso em tipos derivados . . . 40
3.4.2.2 Declarações dos componentes do tipo . . . 41
3.5 Matrizes (*Arrays*) . . . 41
3.6 Coarrays . . . 41
3.7 Ponteiros e estruturas alocáveis . . . 42

4 Expressões e Atribuições Escalares 43
4.1 Regras básicas para expressões escalares . . . 43
4.2 Expressões numéricas escalares . . . 44
4.3 Atribuições numéricas escalares . . . 45
4.4 Operadores relacionais . . . 46
4.5 Expressões e atribuições lógicas escalares . . . 47
4.6 Expressões e atribuições de caracteres . . . 48
4.6.1 Atribuição de caracteres . . . 49
4.6.2 Expressões: operador de concatenação . . . 49
4.6.3 *Substrings* de caracteres . . . 49
4.6.4 Uso de operadores relacionais com caracteres . . . 50
4.7 Hierarquia das diferentes operações . . . 51
4.8 Construtores de estruturas . . . 52

5 Comandos e Construtos de Controle de Fluxo 53
5.1 Comando e construto `IF` . . . 53
5.1.1 Comando `IF` . . . 53
5.1.2 Construto `IF` . . . 54
5.2 Construto `DO` . . . 56
5.2.1 Construto `DO` ilimitado . . . 58
5.2.2 Instrução `EXIT` . . . 58
5.2.3 Instrução `CYCLE` . . . 59
5.3 Construto `CASE` . . . 59

6 Processamento de Matrizes 63
6.1 Terminologia e especificações de matrizes . . . 63
6.2 Expressões e atribuições envolvendo matrizes . . . 67
6.3 Seções de matrizes . . . 69
6.3.1 Subscritos simples . . . 69
6.3.2 Tripleto de subscritos . . . 69
6.3.3 Vetores de subscritos . . . 70
6.4 Atribuições de matrizes e submatrizes . . . 71
6.5 Matrizes de tamanho zero . . . 72
6.6 Construtores de matrizes . . . 73
6.6.1 A função intrínseca `RESHAPE` . . . 74
6.6.2 A ordem dos elementos de matrizes . . . 75
6.7 Rotinas intrínsecas para matrizes . . . 75
6.7.1 Rotinas intrínsecas elementais aplicáveis a matrizes . . . 75
6.7.2 Funções inquiridoras . . . 76
6.7.3 Funções transformacionais . . . 76
6.8 Comando e construto `WHERE` . . . 76
6.8.1 Comando `WHERE` . . . 77
6.8.2 Construto `WHERE` . . . 77
6.9 Construto `DO CONCURRENT` . . . 79

7 Objetos e Estruturas Dinâmicas de Dados 83

7.1 Matrizes alocáveis . . . 83
7.1.1 Definição e uso básico . . . 84
7.1.2 Recursos avançados de alocação/realocação de matrizes . . . 86
7.1.2.1 A rotina intrínseca MOVE_ALLOC . . . 86
7.1.2.2 Alocação/realocação automática . . . 87
7.1.3 Alocação dinâmica de componentes de tipos derivados . . . 89
7.2 Ponteiros (*pointers*) . . . 90
7.2.1 Declarações básicas de ponteiros e alvos . . . 91
7.2.2 Atribuições de ponteiros e seus usos em expressões . . . 92
7.2.2.1 Status de associação . . . 92
7.2.2.2 Associação de ponteiros com objetos de dados . . . 93
7.2.2.3 Uso de ponteiros em expressões ou atribuições . . . 98
7.3 Objetos alocáveis . . . 102
7.3.1 Objetos com parâmetros de tipo deferido . . . 102
7.3.2 A forma geral da instrução ALLOCATE . . . 103
7.3.2.1 Alocação de tipo . . . 104
7.3.2.2 Alocação de fonte . . . 104
7.3.3 Tipos derivados com componentes alocáveis . . . 105
7.4 Estruturas dinâmicas de dados . . . 106
7.4.1 Listas encadeadas ou alocáveis . . . 106
7.4.1.1 Listas encadeadas . . . 107
7.4.1.2 Listas alocáveis . . . 110
7.4.1.3 Listas encadeadas circulares . . . 111
7.4.2 Listas duplamente encadeadas e árvores binárias . . . 112
7.4.3 Matrizes de ponteiros e listas multiplamente conectadas . . . 113
7.5 Recursos orientados à computação de alta performance . . . 115

8 Rotinas Intrínsecas **117**
8.1 Conceitos genéricos . . . 117
8.1.1 Uso de palavras-chave . . . 117
8.1.2 Categorias de rotinas intrínsecas . . . 117
8.1.3 Declaração e atributo INTRINSIC . . . 118
8.2 Funções inquiridoras de qualquer tipo . . . 118
8.3 Funções elementais numéricas . . . 119
8.3.1 Funções elementais que podem converter . . . 119
8.3.2 Funções elementais que não convertem . . . 120
8.4 Funções elementais matemáticas . . . 120
8.5 Funções transformacionais para as funções de Bessel . . . 122
8.6 Funções elementais lógicas e de caracteres . . . 122
8.6.1 Conversões caractere-inteiro . . . 122
8.6.2 Funções de comparação léxica . . . 123
8.6.3 Funções elementais para manipulações de strings . . . 123
8.6.4 Conversão lógica . . . 124
8.7 Funções não elementais para manipulação de strings . . . 124
8.7.1 Função inquiridora para manipulação de strings . . . 124
8.7.2 Funções transformacionais para manipulação de strings . . . 124
8.7.3 Função inquiridora de caractere . . . 124
8.8 Funções inquiridoras e de manipulações numéricas . . . 125
8.8.1 Modelos para dados inteiros e reais . . . 125
8.8.2 Funções numéricas inquiridoras . . . 125
8.8.3 Função elemental para tipos numéricos . . . 126
8.8.4 Funções elementais que manipulam quantidades reais . . . 126
8.8.5 Funções transformacionais para valores de espécie . . . 127
8.9 Rotinas de manipulação de bits . . . 128
8.9.1 Função inquiridora . . . 128
8.9.2 Funções elementais básicas . . . 128
8.9.3 Operações de deslocamento (*shift operations*) . . . 129
8.9.4 Subrotina elemental . . . 129
8.9.5 Comparação bit a bit (*bitwise unsigned comparison*) . . . 129

8.9.6 Deslocamento de dupla largura (*double-width shifting*) . . . 130
8.9.7 Reduções bit a bit (*bitwise reductions*) . . . 130
8.9.8 Contagem de bits . . . 130
8.9.9 Produzindo máscaras de bits (*bitmasks*) . . . 130
8.9.10 Fusão de bits . . . 130
8.10 Função de transferência . . . 131
8.11 Funções de multiplicação vetorial ou matricial . . . 131
8.12 Funções transformacionais que reduzem matrizes . . . 131
8.12.1 Caso de argumento único . . . 131
8.12.2 Argumento opcional `DIM` . . . 132
8.12.3 Argumento opcional `MASK` . . . 132
8.12.4 Redução generalizada de matriz . . . 133
8.13 Funções inquiridoras de matrizes . . . 133
8.13.1 Contiguidade . . . 133
8.13.2 Limites, forma e tamanho . . . 134
8.14 Funções de construção e manipulação de matrizes . . . 134
8.14.1 Função elemental `MERGE` . . . 134
8.14.2 Agrupando e desagrupando matrizes . . . 134
8.14.3 Alterando a forma de uma matriz . . . 135
8.14.4 Função transformacional para duplicação . . . 135
8.14.5 Funções de deslocamento matricial . . . 135
8.14.6 Transposta de uma matriz . . . 135
8.15 Funções transformacionais para localização geométrica . . . 136
8.16 Função transformacional para desassociação ou dealocação . . . 136
8.17 Subrotinas intrínsecas não-elementais . . . 137
8.17.1 Relógio de tempo real . . . 137
8.17.2 Tempo da CPU . . . 138
8.17.3 Números aleatórios . . . 138
8.17.4 Executando outros programas . . . 139
8.18 Acesso ao ambiente computacional . . . 139
8.18.1 Variáveis de ambiente . . . 139
8.18.2 Informações acerca da invocação do programa . . . 140
8.19 Funções elementais para teste de status de Entrada/Saída . . . 141
8.20 Espaço de memória ocupado por um objeto . . . 141
8.21 Rotinas adicionais . . . 141

9 Unidades de Programa do Fortran 143
9.1 Unidades de programa . . . 143
9.1.1 Programa principal . . . 143
9.1.2 Rotinas externas . . . 145
9.1.3 Módulos . . . 145
9.2 Subprogramas . . . 146
9.2.1 Funções e sub-rotinas externas ou de módulo . . . 146
9.2.1.1 Invocações de funções . . . 147
9.2.1.2 Invocações de sub-rotinas . . . 147
9.2.2 Rotinas internas . . . 147
9.3 Argumentos de subprogramas . . . 148
9.3.1 Instrução `RETURN` . . . 149
9.3.2 Controle de acesso aos argumentos mudos . . . 149
9.3.3 Argumentos com palavras-chave . . . 152
9.3.4 Argumentos opcionais . . . 153
9.3.5 Tipos derivados como argumentos de rotinas . . . 154
9.3.6 Matrizes como argumentos de rotinas . . . 154
9.3.6.1 Matrizes de forma explícita ou ajustáveis . . . 154
9.3.6.2 Matrizes de forma assumida . . . 155
9.3.6.3 Objetos automáticos . . . 157
9.3.7 Subprogramas como argumentos de rotinas . . . 159
9.3.8 Ponteiros como argumentos de rotinas . . . 161
9.3.9 Objetos alocáveis como argumentos de rotinas . . . 164

9.4 Rotinas externas e bibliotecas . . . 165
9.5 Interfaces implícitas e explícitas . . . 165
9.5.1 A instrução IMPORT . . . 169
9.6 Funções de valor matricial e funções alocáveis . . . 170
9.7 Funções com valor de ponteiro . . . 171
9.8 Recursividade e rotinas recursivas . . . 172
9.8.1 Recursividade direta . . . 172
9.8.2 Recursividade indireta . . . 173
9.9 Atributo e declaração SAVE . . . 174
9.10 Funções de efeito lateral e rotinas puras . . . 175
9.11 Rotinas elementais . . . 177
9.12 Interfaces abstratas e ponteiros de rotinas . . . 178
9.12.1 Interfaces abstratas e declaração PROCEDURE . . . 178
9.12.2 Ponteiros de rotinas . . . 180
9.12.2.1 Ponteiros de rotinas nomeados . . . 180
9.12.2.2 Ponteiros de rotinas como componentes de tipos derivados . . . 181
9.13 Módulos . . . 181
9.13.1 Dados globais . . . 182
9.13.2 Rotinas de módulos . . . 183
9.13.3 Controle de acesso aos objetos em módulos . . . 186
9.13.4 Interfaces e rotinas genéricas . . . 188
9.13.5 Estendendo rotinas intrínsecas *via* blocos de interface genéricos . . . 191
9.13.6 Módulos intrínsecos . . . 192
9.13.7 Submódulos . . . 194
9.13.7.1 Rotinas de módulo separadas . . . 195
9.13.7.2 Submódulos de submódulos . . . 196
9.13.7.3 Entidades de submódulos . . . 196
9.13.7.4 Submódulos e associação por uso . . . 196
9.14 Âmbito *(Scope)* . . . 196
9.14.1 Âmbito dos rótulos . . . 196
9.14.2 Âmbito dos nomes . . . 197
9.14.3 Controle de acesso ao hospedeiro . . . 198
9.15 Construtos BLOCK . . . 199

10 Comandos de Entrada/Saída de Dados **201**
10.1 Comandos de Entrada/Saída: introdução rápida . . . 201
10.2 Declaração NAMELIST . . . 209
10.3 Instrução INCLUDE . . . 211
10.4 Unidades lógicas . . . 212
10.5 Arquivos internos . . . 212
10.6 Comando OPEN . . . 214
10.7 Comando READ . . . 217
10.8 Comandos PRINT e WRITE . . . 220
10.9 Comando FORMAT e especificador FMT= . . . 221
10.10 Descritores de edição . . . 222
10.10.1 Contadores de repetição . . . 222
10.10.2 Descritores de edição de dados . . . 223
10.10.3 Descritores de controle de edição . . . 227
10.10.4 Descritores de edição de strings . . . 233
10.11 Comando CLOSE . . . 234
10.12 Comando INQUIRE . . . 235
10.13 Outros comandos que operam sobre arquivos . . . 238
10.13.1 Comando BACKSPACE . . . 238
10.13.2 Comando REWIND . . . 238
10.13.3 Comando ENDFILE . . . 239
10.13.4 Comando FLUSH . . . 239

Referências Bibliográficas **240**

1 INTRODUÇÃO

> "*I don't know what the technical characteristics of the standard language for scientific and engineering computation in the year 2000 will be... but I know it will be called Fortran.*"
>
> John Backus

Esta apostila destina-se ao aprendizado da **Linguagem de Programação Fortran Moderno.** Por *Fortran Moderno (Modern Fortran)* entende-se a linguagem de programação de acordo com o seu padrão mais recente, conforme será brevemente apresentado nas seções a seguir.

1.1 AS ORIGENS DA LINGUAGEM FORTRAN

Programação no período inicial do uso de computadores para a solução de problemas em física, química, engenharia, matemática e outras áreas da ciência era um processo complexo e tedioso ao extremo. Programadores necessitavam de um conhecimento detalhado das instruções, registradores, endereços de memória e outros constituintes da *Unidade Central de Processamento* (CPU[1]) do computador para o qual eles escreviam o código. O *Código-Fonte* era escrito em um notação numérica denominada *código de máquina.* Com o tempo, códigos mnemônicos foram introduzidos, uma forma de programação conhecida como *código numérico* ou *código Assembler*. Estes códigos eram traduzidos em instruções para a CPU por programas conhecidos como *Assemblers.* Durante os anos 50 ficou claro que esta forma de programação era de todo inconveniente, no mínimo devido ao tempo necessário para se escrever e testar um programa, embora esta forma de programação possibilitasse um uso otimizado dos recursos da CPU.[2]

Estas dificuldades motivaram que um time de programadores da IBM, liderados por John Warner Backus (1924–2007), desenvolvessem uma das primeiras *linguagem de alto-nível*, denominada FORTRAN (FORmula TRANslation). Seu objetivo era produzir uma linguagem que fosse simples de ser entendida e usada, mas que gerasse um código numérico quase tão eficiente quanto a linguagem Assembler. Desde o início, o Fortran era tão simples de ser usado que era possível programar fórmulas matemáticas quase como se estas fossem escritas de forma simbólica. Isto permitiu que programas fossem escritos mais rapidamente que antes, com somente uma pequena perda de eficiência no processamento, uma vez que todo cuidado era dedicado na construção do *compilador*, isto é, no programa que se encarrega de traduzir o código-fonte em Fortran para código Assembler ou octal.[3]

Mas o Fortran foi um passo revolucionário também porque o programadores foram aliviados da tarefa tediosa de usar Assembler, assim concentrando-se mais na solução do problema em questão. Mais importante ainda, computadores se tornaram acessíveis a qualquer cientista ou engenheiro disposto a devotar um pequeno esforço na aquisição de um conhecimento básico em Fortran; a tarefa da programação não estava mais restrita a um *corpus* pequeno de programadores especialistas.

O Fortran disseminou-se rapidamente, principalmente nas áreas da física, engenharia e matemática, uma vez que satisfazia uma necessidade premente dos cientistas. Inevitavelmente,

[1] Do inglês: *Central Processing Unit.*

[2] Em `http://asm.sourceforge.net/intro/hello.html` pode-se visualizar o código `hello.asm`, o qual implementa em assembler, em um total de 13 linhas, o mesmo tipo de programa que pode ser escrito em três ou quatro linhas em `Fortran`. Veja a listagem 2.1.

[3] Diversas publicações comemorando a história do Fortran e de John Backus são disponíveis. Ver, por exemplo, John Backus - the Father of Fortran, History of FORTRAN and FORTRAN II, History of Fortran ou From FORTRAN II to Modern Fortran.

dialetos da linguagem foram desenvolvidos, os quais levaram a problemas quando havia necessidade de se trocar programas entre diferentes computadores. O dialeto de Fortran otimizado para processadores fabricados pela IBM, por exemplo, facilmente gerava erro quando se tentava rodar o mesmo programa em um processador Burroughs, ou em outro qualquer. Assim, em 1966, após quatro anos de trabalho, a Associação Americana de Padrões (*American Standards Association*), posteriormente Instituto Americano Nacional de Padrões (*American National Standards Institute*, ou ANSI) originou o primeiro padrão para uma linguagem de programação, agora conhecido como Fortran 66. Essencialmente, era uma subconjunto comum de vários dialetos, de tal forma que cada dialeto poderia ser reconhecido como uma extensão do padrão. Aqueles usuários que desejavam escrever programas *portáveis* deveriam evitar as extensões e restringir-se ao padrão.

O Fortran trouxe consigo vários outros avanços, além de sua facilidade de aprendizagem combinada com um enfoque em execução eficiente de código. Era, por exemplo, uma linguagem que permanecia próxima (e explorava) o hardware disponível, ao invés de ser um conjunto de conceitos abstratos. Ela também introduziu, através das declarações `COMMON` e `EQUIVALENCE`, a possibilidade dos programadores controlarem a alocação da armazenagem de dados de uma forma simples, um recurso que era necessário nos primórdios da computação, quando havia pouco espaço de memória, mesmo que estas declarações sejam agora consideradas potencialmente perigosas e tenham o seu uso desencorajado. Finalmente, o código-fonte permitia espaços em branco na sua sintaxe, liberando o programador da tarefa de escrever código em colunas rigidamente definidas e permitindo que o corpo do programa fosse escrito da forma desejada e visualmente mais atrativa.

A proliferação de dialetos permaneceu um problema mesmo após a publicação do padrão Fortran 66. A primeira dificuldade foi que muitos compiladores não aderiram ao padrão. A segunda foi a implementação, em diversos compiladores, de recursos que eram essenciais para programas de grande escala, mas que eram ignorados pelo padrão. Diferentes compiladores implementavam estes recursos de formas distintas.

Esta situação, combinada com a existência de falhas evidentes na linguagem, tais como a falta de construções estruturadas de programação, resultaram na introdução de um grande número de *pré-processadores.* Estes eram programas que eram capazes de ler o código fonte de algum dialeto bem definido de Fortran e gerar um segundo arquivo com o texto no padrão, o qual então era apresentado ao compilador nesta forma. Este recurso provia uma maneira de estender o Fortran, ao mesmo tempo retendo a sua portabilidade. O problema era que embora os programas gerados com o uso de um pré-processador fossem portáveis, podendo ser compilados em diferentes computadores, o código gerado era muitas vezes de uma dificuldade proibitiva para a leitura direta.

Estas dificuldades foram parcialmente removidas pela publicação de um novo padrão, em 1978, conhecido como Fortran 77. Ele incluía diversos novos recursos que eram baseados em extensões comerciais ou pré-processadores e era, portanto, não um subconjunto comum de dialetos existentes, mas sim um novo dialeto por si só. O período de transição entre o Fortran 66 e o Fortran 77 foi muito mais longo que deveria, devido aos atrasos na elaboração de novas versões dos compiladores e os dois padrões coexistiram durante um intervalo de tempo considerável, que se estendeu até meados da década de 80. Eventualmente, os fabricantes de compiladores passaram a liberá-los somente com o novo padrão, o que não impediu o uso de programas escritos em Fortran 66, uma vez que o Fortran 77 permitia este código antigo por compatibilidade. Contudo, diversas extensões não foram mais permitidas, uma vez que o padrão não as incluiu na sua sintaxe.

1.2 O padrão Fortran 90

Após trinta anos de existência, Fortran não mais era a única linguagem de programação disponível para os programadores. Ao longo do tempo, novas linguagens foram desenvolvidas e, onde elas se mostraram mais adequadas para um tipo particular de aplicação, foram adotadas em seu lugar. A superioridade do Fortran sempre esteve na área de aplicações numéricas, científicas, técnicas e de engenharia. A comunidade de usuários do Fortran realizou um investimento gigantesco em códigos, com muitos programas em uso frequente, alguns com centenas de milhares ou milhões de linhas de código. Isto não significava, contudo, que a comunidade estivesse completamente satisfeita com a linguagem. Vários programadores passaram a migrar seus có-

digos para linguagens tais como Pascal, C e C++. Para levar a cabo mais uma modernização da linguagem, o comitê técnico X3J3 (agora PL22.3), constituído pela ANSI, trabalhando como o corpo de desenvolvimento do comitê da ISO (*International Standards Organization*, Organização Internacional de Padrões) ISO/IEC JTC1/SC22/WG5 (doravante conhecido como WG5), preparou um novo padrão, inicialmente conhecido como Fortran 8x, e agora como Fortran 90. Os comitês X3J3 e WG5 são formados por representantes dos vendedores e fabricantes de hardware e desenvolvedores de software, academia e usuários da linguagem.

Quais eram as justificativas para continuar com o processo de revisão do padrão da linguagem Fortran? Além de padronizar extensões comerciais, havia a necessidade de modernização, em resposta aos desenvolvimentos nos conceitos de linguagens de programação que eram explorados em outras linguagens tais como APL, Algol, Pascal, Ada, C e C++. Com base nestas, o X3J3 podia usar os óbvios benefícios de conceitos tais como ocultamento de dados. Na mesma linha, havia a necessidade de fornecer uma alternativa à perigosa associação de armazenagem de dados, de abolir a rigidez agora desnecessária do formato fixo de fonte, bem como de aumentar a segurança na programação. Para proteger o investimento em Fortran 77, todo o padrão anterior foi mantido como um subconjunto do Fortran 90.

Contudo, de forma distinta dos padrões prévios, os quais resultaram quase inteiramente de um esforço de padronizar práticas existentes, o Fortran 90 é muito mais um desenvolvimento da linguagem, na qual são introduzidos recursos que são novos em Fortran, mas baseados em experiências em outras linguagens. Os recursos novos mais significativos são a habilidade de manipular matrizes usando uma notação concisa mais poderosa e a habilidade de definir e manipular tipos de dados definidos pelo programador. O primeiro destes recursos leva a uma simplificação na escrita de muitos problemas matemáticos e também torna o Fortran uma linguagem mais eficiente em supercomputadores. O segundo possibilita aos programadores a expressão de seus problemas em termos de tipos de dados que reproduzem exatamente os conceitos utilizados nas sua elaborações.

O padrão Fortran 90 foi publicado em 1991 e está disponível em https://wg5-fortran.org/N001-N1100/N692.pdf. Um relato pessoal das atividades do X3J3, inclusive citando resistências para a elaboração e adoção de um novo padrão, pode ser lido em https://www.nag.co.uk/content/personal-history-nag-fortran-compiler.

1.2.1 PROCEDIMENTOS ADOTADOS PARA A EVOLUÇÃO DA LINGUAGEM

Os procedimentos de trabalho adotados pelo comitê X3J3 estabelecem um período de aviso prévio antes que qualquer recurso existente seja removido da linguagem. Isto implica, na prática, um ciclo de revisão, que para o Fortran é de cerca de cinco anos. A necessidade de remoção de alguns recursos é evidente; se a única ação adotada pelo X3J3 fosse de adicionar novos recursos, a linguagem se tornaria grotescamente ampla, com muitos recursos redundantes e sobrepostos. Assim, além de detalhar os novos recursos que passam a ser adicionados à linguagem, cada padrão também possui um apêndice contendo duas listas:

Recursos obsolescentes (*obsolescent features*): tratam-se daqueles recursos ainda suportados pelo padrão, mas que são considerados obsoletos e, assim, candidatos à remoção no próximo ciclo de revisão.

Recursos removidos (*deleted features*): recursos que não são mais suportados na linguagem.

1.2.2 RECURSOS NOVOS DO FORTRAN 90

Um resumo dos novos recursos é dado a seguir:

- Operações de matrizes.
- Ponteiros.
- Recursos avançados para computação numérica usando um conjunto de funções inquisidoras numéricas.

- Parametrização dos tipos intrínsecos, permitindo o suporte a inteiros curtos, conjuntos de caracteres muito grandes, mais de duas precisões para variáveis reais e complexas e variáveis lógicas agrupadas.
- Tipos de dados derivados, definidos pelo programador, compostos por estruturas de dados arbitrárias e de operações sobre estas estruturas.
- Facilidades na definição de coletâneas denominadas *módulos*, úteis para definições globais de dados e para bibliotecas de subprogramas.
- Exigência que o compilador detecte o uso de construções que não se conformam com a linguagem ou que estejam em obsolescência.
- Um novo formato de fonte, adequado para usar em um terminal.
- Novas estruturas de controle, tais como `SELECT CASE` e uma nova forma para os laços `DO`.
- A habilidade de escrever subprogramas internos e subprogramas recursivos e de chamar subprogramas com argumentos opcionais.
- Alocação dinâmica de dados (matrizes automáticas, matrizes alocáveis e ponteiros).
- Melhoramentos nos recursos de entrada/saída de dados.
- Vários novos subprogramas intrínsecos.

Todos juntos, estes novos recursos contidos em Fortran 90 irão assegurar que o padrão continue a ser bem sucedido e usado por um longo tempo. O Fortran 77 continua sendo suportado como um subconjunto durante um período de adaptação.

1.2.3 Recursos em obsolescência no Fortran 90

Os recursos obsolescentes no Fortran 90 são:

- `IF` aritmético;
- desvio para uma declaração `END IF` a partir de um ponto fora de seu bloco;
- variáveis reais e de dupla precisão nas expressões de controle de um comando `DO`;
- finalização compartilhada de blocos `DO`, bem como finalização por uma declaração ou comando distintos de um `CONTINUE` ou de um `END DO`;
- declaração `ASSIGN` e comando `GO TO` atribuído;
- `RETURN` alternativo;
- comando `PAUSE`;
- especificadores `FORMAT` atribuídos;
- descritor de edição `H`.

1.2.4 Recursos removidos do Fortran 90

Uma vez que Fortran 90 contém o Fortran 77 como subconjunto, esta lista permaneceu vazia para o Fortran 90.

1.3 UMA REVISÃO MENOR: FORTRAN 95

Seguindo a publicação do padrão Fortran 90 em 1991, dois significativos desenvolvimentos posteriores referentes à linguagem Fortran ocorreram. O primeiro foi a continuidade na operação dos dois comitês de regulamentação do padrão da linguagem: o X3J3 e o WG5; o segundo desenvolvimento foi a criação do Fórum Fortran de Alta Performance (*High Performance Fortran Forum,* ou HPFF).

Logo no inicio de suas deliberações, os comitês concordaram na estratégia de definir uma revisão menor no Fortran 90 para meados da década de 90, seguida por uma revisão de maior escala para o início dos anos 2000. Esta revisão menor passou a ser denominada Fortran 95.

O HPFF teve como objetivo a definição de um conjunto de extensões ao Fortran tais que permitissem a construção de códigos portáveis quando se fizesse uso de computadores paralelos para a solução de problemas envolvendo grandes conjuntos de dados que podem ser representados por matrizes regulares. Esta versão do Fortran ficou conhecida como o Fortran de Alta Performance (*High Performance Fortran*, ou HPF), tendo como linguagem de base o Fortran 90, não o Fortran 77. A versão final do HPF consiste em um conjunto de instruções que contém o Fortran 90 como subconjunto. As principais extensões estão na forma de diretivas, que são vistas pelo Fortran 90 como comentários, mas que são reconhecidas por um compilador HPF. Contudo, tornou-se necessária também a inclusão de elementos adicionais na sintaxe, uma vez que nem todos os recursos desejados puderam ser acomodados simplesmente na forma de diretivas.

À medida que os comitês X3J3 e WG5 trabalhavam, este comunicavam-se regularmente com o HPFF. Era evidente que, para evitar o surgimento de dialetos divergentes de Fortran, havia a necessidade de incorporar a sintaxe nova desenvolvida pelo HPFF no novo padrão da linguagem. De fato, os recursos do HPF constituem as novidades mais importantes do Fortran 95. As outras mudanças consistem em correções, clarificações e interpretações do novo padrão. Estas se tornaram prementes quando os novos compiladores de Fortran 90 foram lançados no mercado e utilizados; notou-se uma série de erros e detalhes obscuros que demandavam reparações. Todas estas mudanças foram incluídas no novo padrão Fortran 95, que teve a sua versão inicial lançada no próprio ano de 1995.

O Fortran 95 é compatível com o Fortran 90, exceto por uma pequena alteração na função intrínseca `SIGN` (seção 8.3.2) e a eliminação de recursos típicos do Fortran 77, declarados em obsolescência no Fortran 90. Os detalhes do Fortran 95 foram finalizados em novembro de 1995 e o novo padrão ISO foi finalmente publicado em outubro de 1996. O texto completo do padrão Fortran 95 pode ser obtido em https://wg5-fortran.org/N1151-N1200/N1176.pdf.

1.3.1 RECURSOS NOVOS DO FORTRAN 95

Em relação ao Fortran 90, foram introduzidos os seguintes recursos:

- Concordância aprimorada com o padrão de aritmética de ponto flutuante binária da IEEE (IEEE 754 ou IEC 559-1989).
- Rotinas *(procedures)* puras (seção 9.10).
- Rotinas *(procedures)* elementais (seção 9.11).
- Dealocação automática de matrizes alocáveis em rotinas.
- Comando e construto `FORALL`.
- Extensões do construto `WHERE` (página 78).
- Funções especificadoras.
- Inicialização de ponteiro e a função `NULL` (seção 8.16).
- Inicialização de componentes de tipo derivado (página 38).
- Funções elementares `CEILING`, `FLOOR` e `SIGN` (seções 8.3.1 e 8.3.2).
- Funções transformacionais (8.15).

- Subrotina intrínseca `CPU_TIME` (seção 8.17.2).
- Comentário na entrada de uma lista `NAMELIST` (página 210).
- Descritores de edição com largura de campo mínimo.
- Especificação genérica na cláusula `END INTERFACE`.

1.3.2 Recursos em obsolescência no Fortran 95

Os recursos abaixo entraram na lista em obsolescência no Fortran 95 e, portanto, pouco ou nada foram comentados ao longo desta Apostila.

- Formato fixo de fonte.
- Comando `GO TO` computado.
- Declaração de variável de caractere na forma `CHARACTER*`.
- Declarações `DATA` entre comandos executáveis.
- Funções definidas em uma linha *(statement functions)*.
- Extensão assumida de caracteres quando estas são resultados de funções.

1.3.3 Recursos removidos do Fortran 95

Cinco recursos foram removidos do padrão da linguagem Fortran 95 e, portanto, não serão mais aceitos por compiladores que respeitam o padrão Fortran 95.

- Índices de laços `DO` do tipo real (qualquer espécie).
- Declaração `ASSIGN` e comando `GO TO` atribuído e uso de um inteiro atribuído por `ASSIGN` em uma declaração `FORMAT`.
- Desvio para uma declaração `END IF` a partir de um ponto fora do bloco.
- Comando `PAUSE`.
- Descritor de edição `H`.

1.4 O Fortran no Século XXI: Fortran 2003

O Fortran 2003 é novamente uma revisão grande do padrão anterior: Fortran 95. As grandes novidades introduzidas foram: um direcionamento ainda maior para programação orientada a objeto, a qual oferece uma maneira efetiva de separar a programação de um código grande e complexo em tarefas independentes e que permite a construção de novo código baseado em rotinas já existentes e uma capacidade expandida de interface com a linguagem C, necessária para que os programadores em Fortran possam acessar rotinas escritas em C/C++ e para que programadores de C possam acessar rotinas escritas em Fortran.

O padrão completo foi publicado em novembro de 2004. O texto oficial somente é disponibilizado pela ISO mediante pagamento, mas uma cópia é disponibilizada livremente pelo WG5 em https://wg5-fortran.org/N1601-N1650/N1601.pdf.[4] Uma breve descrição dos novos recursos pode ser vista em https://wg5-fortran.org/N1601-N1650/N1648.pdf.

[4] Mais informações podem ser obtidas em https://wg5-fortran.org/f2003.html.

1.4.1 RECURSOS NOVOS DO FORTRAN 2003

Os principais recursos introduzidos pelo padrão são:

- Aprimoramentos dos tipos derivados: tipos derivados parametrizados, controle melhorado de acessibilidade, construtores de estrutura aperfeiçoados e finalizadores.
- Suporte para programação orientada a objeto: extensão de tipo e herança *(inheritance)*, polimorfismo *(polymorphism)*, alocação dinâmica de tipo e rotinas *(procedures)* ligadas a tipo.
- Aperfeiçoamentos na manipulação de dados: componentes alocáveis de estruturas, argumentos mudos alocáveis, parâmetros de tipo deferidos *(deferred type parameters)*, atributo `VOLATILE`, especificação explícita de tipo em construtores de matrizes e declarações de alocação, aperfeiçoamentos em ponteiros, expressões de inicialização estendidas e rotinas intrínsecas aperfeiçoadas.
- Aperfeiçoamentos em operações de Entrada/Saída (E/S) de dados: transferência assíncrona, acesso de fluxo *(stream access)*, operações de transferência especificadas pelo usuário para tipos derivados, controle especificado pelo usuário para o arredondamento em declarações `FORMAT`, constantes nomeadas para unidades pré-conectadas, o comando `FLUSH`, regularização de palavras-chave e acesso a mensagens de erro.
- Ponteiros de rotinas *(procedure pointers).*
- Suporte para as exceções do padrão de aritmética binária de ponto flutuante ISO/IEC 559, anteriormente conhecido como padrão IEEE 754.
- Interoperabilidade com a linguagem de programação C.
- Suporte aperfeiçoado para internacionalização: acesso ao conjunto de caracteres de 4 bytes ISO 10646 and escolha de vírgula ou ponto como separador de parte inteira e parte decimal em operações de E/S numéricas formatadas.
- Integração aperfeiçoada com o sistema operacional hospedeiro: acesso a argumentos de linha de comando, variáveis de ambiente e mensagens de erro do processador.

1.4.2 RECURSOS EM OBSOLESCÊNCIA NO FORTRAN 2003

Nesta lista permanecem itens que já estavam na lista de obsolescência no Fortran 95.[5] Os critérios para inclusão nesta lista continuam sendo: recursos redundantes e para os quais métodos melhores estavam disponíveis no Fortran 95. A lista adicional dos itens em obsolescência no Fortran 2003 é:

- `IF` aritmético.
- `END DO` compartilhado por dois ou mais laços e término de um laço em uma cláusula distinta de `END DO` ou `CONTINUE`.
- `RETURN` alternativo.

1.4.3 RECURSOS REMOVIDOS DO FORTRAN 2003

Um recurso é removido do novo padrão se este for considerado redundante e praticamente sem uso pela comunidade, devido a novos recursos muito mais úteis na construção do código. A lista de recursos removidos do Fortran 2003 repete os recursos removidos do Fortran 95 (seção 1.3.3). Portanto, nenhum outro recurso foi removido.

[5] Ver seção 1.3.2.

1.4.4 Status de suporte ao padrão Fortran 2003

Embora o Fortran 2003 tenha se tornado o novo padrão da linguagem após a sua publicação, até o presente momento nem todos os compiladores suportam completamente este padrão. Uma relação dos recursos do Fortran 2003 suportados por diferentes compiladores, tanto livres (ou gratuitos), quanto proprietários, pode ser obtida em http://fortranwiki.org/fortran/show/Fortran+2003+status ou em *Chivers & Sleightholme (2019)* [3].[6]

1.5 O padrão Fortran 2008

Embora nem todos os compiladores já suportem completamente o padrão estabelecido pelo Fortran 2003, os comitês X3J3/WG5 decidiram publicar, em outubro de 2010, um novo padrão da linguagem: o Fortran 2008. Uma versão gratuita do texto final do novo padrão pode ser obtida em http://j3-fortran.org/doc/year/10/10-007r1.pdf.[7]

1.5.1 Recursos novos do Fortran 2008

O Fortran 2008, embora pelo procedimento adotado pelos comitês J3 (antes X3J3) e WG5 devesse ser uma revisão menor do Fortran 2003, apresenta algumas novidades substanciais. Alguns dos novos recursos disponíveis na linguagem são:

- Submódulos.
- Comatrizes (*Coarrays*).
- Construto `DO CONCURRENT`.
- Atributo `CONTIGUOUS`.
- Construto `BLOCK`.
- Instrução `EXIT` em (quase) qualquer construto.
- Aperfeiçoamento na instrução `STOP`.
- Rotinas internas podem ser passadas como argumentos reais.
- Rotinas de ponteiro podem apontar para rotinas internas.
- Posto máximo de matrizes aumentado para 15.
- Palavra-chave NEWUNIT= na declaração OPEN.
- Descritor de edição `G0`.
- Lista ilimitada de itens em `FORMAT`.

As seguintes mudanças foram realizadas em rotinas intrínsecas:

- `acos`, `asin`, `atan`, `cosh`, `sinh`, `tan`, e `tanh` agora aceitam argumentos complexos.
- `atan2` pode ser agora chamada simplesmente como `atan`.
- `lge`, `lgt`, `lle`, e `llt` agora aceitam argumentos da espécie ASCII.
- `maxloc` e `minloc` possuem agora um argumento opcional `back=`.
- `selected_real_kind` possui agora um argumento `radix=`.

Novas rotinas intrínsecas:

- Funções especiais:

[6] É importante ressaltar que mesmo estas listas podem se tornar rapidamente desatualizadas. O status mais recente para cada compilador deve ser fornecido pelas respectivas páginas de suporte.

[7] Mais informações podem ser obtidas em https://wg5-fortran.org/f2008.html.

  - funções trigonométricas hiperbólicas inversas: `acosh`, `asinh`, e `atanh`.
  - Funções de Bessel de argumento real: `bessel_j0`, `bessel_j1`, `bessel_jn`, `bessel_y0`, `bessel_y1` e `bessel_yn`.
  - Função erro: `erf`, `erfc` e `erfc_scaled`.
  - Função gama: `gamma` e `log_gamma`.
  - Função distância Euclideana: `hypot` e norma L2: `norm2`.

- Operações bit-a-bit (*bitwise operations*):
  - Bit sequence comparisons: `bge`, `bgt`, `ble` e `blt`.
  - Combined shifting: `dshiftl` e `dshiftr`.
  - Counting bits: `leadz`, `trailz`, `popcnt` e `poppar`.
  - Masking bits: `maskl` e `maskr`.
  - Shifting bits: `shifta`, `shiftl` e `shiftr`.
  - Merging bits: `merge_bits`.
  - Bit transformational functions: `iall`, `iany` e `iparity`.

- Coarray intrinsics:
  - Convert a cosubscript to an image index: `image_index`.
  - Cobounds of a coarray: `lcobound` e `ucobound`.
  - Number of images: `num_images`.
  - Image index or cosubscripts: `this_image`.

- Outros:
  - Test for the contiguous attribute: `is_contiguous`.
  - Size of an element in bits: `storage_size`.
  - Tests for the number of true values being odd: `parity`.
  - Search for a value in an array: `findloc`.
  - Shell commands: `execute_command_line`.
  - Define and reference variables atomically: `atomic_define` e `atomic_ref`.

Adições a módulos intrínsecos:

- `ISO_FORTRAN_ENV`:
  - Information about the compiler: `compiler_version` e `compiler_options`.
  - Named constants for selecting kind values.

- `IEEE_ARITHMETIC`:
  - `ieee_selected_real_kind` now has a `radix=` argument.

- `ISO_C_BINDING`:
  - `c_sizeof` returns the size of an array element in bytes.

1.5.2 RECURSOS EM OBSOLESCÊNCIA NO FORTRAN 2008

A lista dos recursos obsolescentes no Fortran 2008 contém itens que já estavam presentes nas listas do Fortran 95 (seção 1.3.2) e do Fortran 2003 (seção 1.4.2). Adicionalmente, foi incluído:

- Instrução `ENTRY`.

1.5.3 Recursos removidos do Fortran 2008

Em adição aos recursos já removidos pelo Fortran 95 (seção 1.3.3), os seguintes recursos foram também excluídos do padrão:

- Formato de controle vertical.

1.5.4 Status do suporte ao Fortran 2008

Uma relação dos recursos do Fortran 2008 suportados por diferentes compiladores, tanto livres (ou gratuitos), quanto proprietários, pode ser obtida em http://fortranwiki.org/fortran/show/Fortran+2008+status ou em *Chivers & Sleightholme (2019)* [3].[8]

1.6 O novo padrão: Fortran 2018

O Fortran 2018 é uma extensão pequena ao Fortran 2008. Os principais detalhes já haviam sido definidos pelos comitês em 2015, mas o novo padrão foi lançado somente em novembro de 2018.[9] O texto completo do novo padrão pode ser obtido em https://j3-fortran.org/doc/year/18/18-007r1.pdf e uma descrição sucinta pode ser vista em The New Features of Fortran 2018, por John Reid.[10]

1.6.1 Recursos novos do Fortran 2018

O Fortran 2018 estende o Fortran 2008 em dois recursos básicos:

Interoperabilidade adicional entre as linguagens Fortran e C, com a introdução das estruturas denominadas `C descriptors`, as quais permitem que sejam passados os endereços de argumentos que não eram interoperáveis, ao invés dos objetos propriamente. O acesso aos objetos passados desta maneira às funções do C é possibilitado também pela definição do arquivo de cabeçalhos (*header file*) `ISO_Fortran_binding.h` [4].

Recursos adicionais de paralelismo no Fortran, com a definição de *Times* (*Teams*) dentro dos coarrays.

Conformidade com as normas ISO/IEC/IEEE 60559:2011, as quais definem o novo padrão para a artimética de ponto flutuante, adotado pela maioria dos fabricantes de hardware e desenvolvedores de software. Os novos recursos são adicionados aos módulos intrínsecos `IEEE_ARITHMETIC`, `IEEE_EXCEPTIONS` e `IEEE_FEATURES`. Maiores informações acerca da norma IEEE 60559:2011 podem ser obtidas em https://www.iso.org/standard/57469.html.

1.6.2 Recursos em obsolescência no Fortran 2018

A lista repete recursos já tornados obsoletos nas listas do Fortran 95 (seção 1.3.2), Fortran 2003 (seção 1.4.2) e Fortran 2008 (seção 1.5.2) e inclui:

- Declaração `DO` com rótulo.
- Declarações `COMMON` e `EQUIVALENCE` e a unidade de programa `BLOCK DATA`.
- Nomes específicos para funções intrínsecas.
- Declaração e construto `FORALL` (criados pelo Fortran 95).[11]

[8] Ver observação na nota de rodapé 6.
[9] Ver anúncio em Fortran 2018 has been published!.
[10] Mais informações podem ser obtidas em https://wg5-fortran.org/f2018.html.
[11] As descrições destes recursos podem ser visualizadas em Introdução ao Fortran 90/95, disponível em http://professor.ufrgs.br/rgaelzer/pages/disciplinas-ufpel.

1.6.3 RECURSOS REMOVIDOS DO FORTRAN 2018

Em adição aos recursos já removidos pelo Fortran 95 (seção 1.3.3) e pelo Fortran 2008 (seção 1.5.3), também foram eliminados do padrão:

- Comando `IF` aritmético.
- Construto `DO` que não está na forma de um bloco.

1.6.4 STATUS DO SUPORTE AO FORTRAN 2018

Devido ao seu recente lançamento, os novos recursos criados pelo Fortran 2018 são suportados (parcialmente) por poucos compiladores. Uma lista do suporte pode ser obtida em http://fortranwiki.org/fortran/show/Compiler+Support+for+Modern+Fortran.

1.7 RECURSOS ADICIONAIS AO FORTRAN MODERNO

Ao longo de seus mais de cinquenta anos, o Fortran acumulou uma quantidade impressionante de códigos e bibliotecas, muitos dos quais são acessíveis. Frisando que uma listagem completa de todos os recursos é impossível, pode-se mencionar os seguintes:

Listas de discussão. Salas virtuais de diálogos, onde é possível buscar a solução de dúvidas ou formular perguntas, as quais serão (tentarão) respondidas por outros programadores com os mais variados graus de conhecimento e experiência. Alguns dos assíduos participantes destas listas são autores de livros sobre Fortran e/ou desenvolvedores de compiladores ou bibliotecas. As principais listas de discussão são:

- Lista sobre Fortran no Google Groups.
- Listas do compilador Intel® Fortran para Linux e macOS e para Windows.

Bibliotecas e plataformas de desenvolvimento. Onde são acessíveis recursos tanto oriundos de décadas de experiência no desenvolvimento de métodos numéricos quanto de novos projetos desenvolvidos na linguagem. Pode-se destacar:

- Biblioteca NetLib em https://www.netlib.org.
- Biblioteca SLATEC em https://people.sc.fsu.edu/~jburkardt/f_src/slatec/slatec.html.
- Biblioteca fgsl: interface em Fortran para GSL (GNU Scientific Library): https://github.com/reinh-bader/fgsl.
- Plataforma de desenvolvimento GitHub. Projetos em Fortran: https://github.com/topics/fortran.
- Plataforma de desenvolvimento SourceForge. Projetos em Fortran: https://sourceforge.net/directory/os:linux/?q=fortran.

Compiladores. Compiladores de Fortran:

(Parcialmente) gratuitos:

- Gfortran: https://gcc.gnu.org/wiki/GFortran.
- Intel® Fortran (https://software.intel.com/en-us/fortran-compilers). Versão livre por 30 dias. Estudantes, professores ou desenvolvedores de software podem adquirir uma licença gratuita. Ver detalhes em https://software.intel.com/en-us/parallel-studio-xe/choose-download.
- PGI (Community Edition): https://www.pgroup.com/products/community.htm.

Comerciais:

- Absoft: https://www.absoft.com.
- NAG Fortran: https://www.nag.co.uk/nag-compiler.

O uso de um compilador para testar os exemplos contidos nesta Apostila é imprescindível para o aprendizagem de qualquer linguagem de programação.

Uma lista mais longa e detalhada de recursos pode ser obtida a partir do texto Fortran Resources, de Ian D. Chivers & Jane Sleghtholme, disponível em https://www.fortranplus.co.uk/app/download/30202494/fortran_resources.pdf.

1.8 Comentários sobre a bibliografia

Para escrever esta apostila, fiz uso de um número restrito de publicações, algumas das quais são de livre acesso através da internet.

- A principal fonte de informação sobre o Fortran Moderno foi o amplo livro de Michael Metcalf, John Reid e Malcolm Cohen: *Modern Fortran Explained. Incorporating Fortran 2018* [1].
- Outros livros relevantes:
  - Stephan J. Chapman. *Fortran for Scientists and Engineers* (4th. Ed.) [2].
  - Norman S. Clerman & Walter Spector. *Modern Fortran. Style and Usage* [5].
- O padrão do Fortran 2018, disponível em https://j3-fortran.org/doc/year/18/18-007r1.pdf, também foi consultado.
- O manual de referência à Linguagem Fortran que acompanha o compilador Intel também foi consultado [6].

1.9 Observações sobre a apostila e agradecimentos

Esta Apostila consiste em uma nova versão do texto Introdução ao Fortran 90/95, o qual foi empregado para ministrar disciplina de métodos computacionais aplicados à Física na Universidade Federal de Pelotas até 2012 e que ainda está acessível em http://professor.ufrgs.br/rgaelzer/pages/disciplinas-ufpel. A presente versão (Introdução ao Fortran Moderno) não faz menção a um padrão em particular da linguagem, abordando outrossim recursos suportados tanto pelo padrão mais recente quanto pelos compiladores nos quais os códigos foram testados.

Em particular, este texto não irá abordar recursos do Fortran 77 ou outros padrões posteriores que sejam hoje considerados obsolescentes. Programadores em Fortran 77 podem ainda consultar o texto Introdução ao Fortran 90/95 ou outros textos tais como: Fortran 90. A Conversion Course for Fortran 77 Programmers, Fortran90 for Fortran77 Programmers ou Conversion from Fortran 77 to Modern Fortran. Uma excelente revisão de Fortran 77 é também apresentada por Clive Page em Professional Programmer's Guide to Fortran 77.

Esta Apostila propõe-se a apresentar uma discussão abrangente de uma parte dos recursos oferecidos pelo padrão atual da linguagem. Contudo, as quatro grandes áreas a seguir não serão abordadas:

- Programação orientada a objetos.
- Coarrays.
- Interoperabilidade com a linguagem C.
- Controle de exceções de ponto flutuante e conformidade com o padrão IEEE 754-2008.

Embora sejam importantes, estes recursos estão além do nível de uma introdução à linguagem.

Todos os códigos ou partes de códigos apresentados neste texto estão de acordo com o padrão mais recente da linguagem e, portanto, devem ser aceitos por compiladores (também nas versões mais recentes) em qualquer arquitetura de hardware. Contudo, como foi ressaltado ao longo das seções 1.2 – 1.6, embora seja razoável esperar que todos os compiladores atuais suportem pelo menos o padrão Fortran 95, o nível de suporte a recursos criados a partir do Fortran 2003 é bastante variado. Os recursos apresentados nesta Apostila foram testados e são aceitos nos seguintes compiladores instalados no sistema operacional Linux:

Intel® Fortran Compiler. Versão empregada:

```
user@machine|dir>ifort –V
Intel(R) Fortran Intel(R) 64 Compiler for applications running on
Intel(R) 64,
Version 19.0.1.144 Build 20181018
Copyright (C) 1985–2018 Intel Corporation.  All rights reserved.
```

GFortran. Versão empregada:

```
user@machine|dir>gfortran --version
GNU Fortran (GCC) 9.1.1 20190503 (Red Hat 9.1.1-1)
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is
    NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
    PURPOSE.
```

Eventualmente, poderá ocorrer que um determinado recurso é suportado por somente um dos compiladores nas versões acima. Neste caso, será impressa uma imagem na margem lateral indicando qual compilador ainda não suporta o recurso em questão. As imagens são: para o compilador Intel® Fortran e para o GFortran.

Por se tratar de uma obra em constante revisão, esta apostila pode conter (e conterá, invariavelmente) uma série de erros de ortografia, pontuação, acentuação, *etc*. Certos pontos poderiam também ser melhor discutidos e podem conter até informações não completamente corretas.

Durante o desenvolvimento e divulgação desta apostila, algumas pessoas contribuiram com o seu aprimoramento ao apontar erros e inconsistências e ao oferecer sugestões quanto a sua estrutura e conteúdo. Qualquer contribuição é bem vinda e pode ser enviada ao endereço eletrônico: rudi.gaelzer@ufrgs.br.

Gostaria de agradecer publicamente as contribuições das seguintes pessoas: Leandro Tezani, Antonio Barbosa.

2 ELEMENTOS BÁSICOS DO FORTRAN

Neste capítulo estuda-se inicialmente a estruturação básica envolvida na criação de um código executável em Fortran, seguida da descrição dos elementos que compõe a linguagem.

2.1 A SINTAXE DA LINGUAGEM

Em Fortran há três formatos básicos de arquivos envolvidos no processo de criação de um código executável:

Programa-Fonte. Trata-se do programa e/ou dos subprogramas escritos pelo programador, usando algum tipo de editor de texto, de acordo com as regras definidas pela linguagem de programação de alto nível. O programa-fonte pode estar estruturado por diferentes *unidades de programa*, as quais serão discutidas em mais detalhes no capítulo 9.

O envolvimento ativo do programador ocorre somente nesta etapa do processo de criação do código executável. As etapas a seguir são usualmente realizadas de forma automática pelo sistema de compilação.

Programa-Objeto. Trata-se do programa-fonte compilado pelo compilador. Esta é a transcrição realizada pelo compilador a partir do programa-fonte fornecido pelo programador para uma linguagem de baixo nível, como Assembler ou outro código diretamente interpretável pela CPU. O programa-objeto não pode ser diretamente executado; é necessário ainda passar-se pela fase de *ligação*, *interconexão* ou *linkagem* (transliteração do termo *linking*).

Programa executável. Após a fase de compilação, onde os programas-objeto são criados, o agente de compilação aciona o *interconector* (*linker*), o qual consiste em um programa especial que agrupa estes programas-objeto com outros objetos contidos em bibliotecas do sistema, de forma a criar um arquivo final, o programa executável, o qual pode ser então executado pelo programador.

Este processo está representado na figura 2.1.

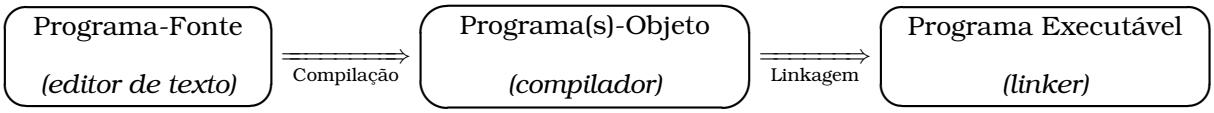


Figura 2.1: Processo de criação de um programa executável.

Nesta seção será apresentada a estrutura básica de um programa-fonte em Fortran. A estrutura de uma linguagem de programação assemelha-se a de uma língua indo-europeia, a qual é composta a partir de elementos básicos, as letras, as quais são combinadas para formar palavras, as quais são colocadas de forma adjacente separadas por espaços em branco ou por pontos, formando assim uma oração ou frase. Estas por sua vez compõe parágrafos que se juntam para formar as demais estruturas de um texto completo, destinado a transmitir ao leitor um pensamento, história ou raciocínio. Este texto é usualmente autoconsistente e em diferentes graduações independente de outros textos. O conjunto de regras que determina como todos estes elementos são compostos e empregados é denominado a **sintaxe** da linguagem.

A sintaxe do Fortran é estruturada de uma forma semelhante. Os elementos básicos da linguagem são:

caracteres alfanuméricos: compostos pelas 26 letras latinas maiúsculas ou minúsculas

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

a b c d e f g h i j k l m n o p q r s t u v w x y z
```

juntamente com os 10 numerais arábicos

```
0 1 2 3 4 5 6 7 8 9
```

e o caractere de grifo (*underscore*) "_".

caracteres especiais: símbolos que são reconhecidos pelo compilador e que podem ou não possuir função sintática. A tabela 2.1 lista os caracteres especiais suportados pela linguagem.

Tabela 2.1: Caracteres especiais do Fortran. C.F.S: com função sintática; S.F.S: sem função sintática.

	Nome (C.F.S.)		**Nome (C.F.S.)**		**Nome (S.F.S.)**
=	Igual	:	Dois pontos	\	Barra invertida
+	Soma		Espaço em branco	$	Cifrão
-	Subtração	!	Exclamação	?	Ponto de interrogação
*	Multiplicação	%	Porcentagem	{	Chave à esquerda
/	Divisão	&	E comercial (*ampersand*)	}	Chave à direita
(	Parênteses esquerdo	;	Ponto e vírgula	~	Til
)	Parênteses direito	<	Menor que	‘	Crase
[	Colchete esquerdo	>	Maior que	^	Acento circunflexo
]	Colchete direito	’	Apóstrofe	|	Linha vertical
,	Vírgula	“	Aspas	#	Cerquilha
.	Ponto decimal			@	Arroba

Os símbolos básicos listados acima combinam-se para formar os **vocábulos léxicos** (***lexical tokens*** ou simplesmente ***tokens***), os quais correspondem às palavras em uma linguagem natural e que fornecem unidade basica de informação do Fortran. Tokens adjacentes são usualmente separados por espaços em branco ou pelo final da linha. Sequências de tokens formam uma **declaração** (***statement***) ou **comando** (ou **instrução**). Estas declarações ou comandos correspondem às frases em uma linguagem natural. Da mesma forma, declarações ou comandos podem ser agrupados para formar o que correspondem aos parágrafos e que no Fortran são denominadas as **unidades de programa**, a partir das quais será composto o programa-fonte. Esta estruturação será discutida em maiores detalhes nas seções a seguir.

2.2 Vocábulos léxicos *(tokens)*

Os caracteres aceitos pelo Fortran podem ser combinados de diferentes maneiras em diferentes contextos para a formação de um determinado token. Um token pode ser um **rótulo**, uma **palavra-chave** (***keyword***), um **nome**, uma **constante**,[1] um **operador**,[2] ou um **separador**. Alguns dos tipos de tokes listado acima serão discutidos a seguir; em particular, os separadores são

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

os quais irão aparecer ao longo deste texto.

As palavras-chave (*keywords*) são nomes que possuem significados especiais no Fortran. Palavras-chave adjacentes devem ser separadas por um ou mais espaços em branco ou pelo final da linha. Contudo, como em qualquer linguagem natural, há exceções. As palavras-chave adjacentes listadas na tabela 2.2 não precisam ser separadas por espaços. Assim, "`end if`" tem o mesmo significado que "`endif`".

Por outro lado, com relação aos rótulos, nomes e constantes, o uso de espaços em branco entre tokens adjacentes irá depender do contexto. Para facilitar a compreensão, pode existir

[1] Discutidas no capítulo 3.
[2] Discutidos no capítulo 4.

Tabela 2.2: Palavras-chave adjacentes para as quais espaços em branco são opcionais.

block data	double precision	else if	else where
end associate	end block	end block data	end critical
end do	end enum	end file	end forall
end function	end if	end interface	end module
end procedure	end program	end select	end submodule
end subroutine	end type	end where	go to
in out	select case	select type	

mais do que um espaço em branco entre tokens adjacentes sem que o significado sintático seja alterado. Por exemplo, a expressão "`x*y`" é composta por três tokens: "`x`", "`*`" e "`y`" e significa o produto de `x` por `y`. O mesmo resultado será obtido se esta expressão for escrita "`x * y`" (com espaços em branco) ou "`x* y`". Por outro lado, "`real x`" é sintaticamente distinto de "`realx`".

2.3 NOMES VÁLIDOS EM FORTRAN

Um programa em Fortran faz referência a muitas entidades distintas, tais como programas, subprogramas, módulos, variáveis, etc. Os nomes destas entidades devem consistir em caracteres alfanuméricos e conter de 1 a 63 destes caracteres. Não há restrições na composição destes nomes, exceto que o primeiro caractere deve ser uma letra. Os seguintes exemplos são válidos:

```
A
A_COISA
X1
MASSA
Q123
TEMPO_DE_VOO
```

Já estes exemplos **não** são válidos:

```
1A        (começa com numeral)
A COISA   (contém espaço em branco)
$SINAL    (contém caractere não alfanumérico).
```

É importante enfatizar que o Fortran é *insensível a maiúsculas ou minúsculas*, isto é, para o compilador, o nome `TEMPO_DE_VOO` é idêntico a `tempo_de_voo` ou `Tempo_de_Voo`.

Sugestões de uso & estilo para programação

Existem convenções estilísticas para a definição de nomes de distintos tipos de objetos da linguagem. Essas convenções não fazem parte do padrão mas são outrossim sugestões que visam uma melhor compreensão do código. Uma regra universal consiste em empregar nomes mnemônicos. Por exemplo, `massa` ou `tempo_de_voo`. Algumas dessas convenções serão mencionadas ao longo do texto.

2.4 FORMATAÇÃO DO PROGRAMA-FONTE

As declarações ou comandos que irão compor um programa-fonte em Fotran são escritas ao longo de linhas em um editor de texto. O Fortran adota o **formato livre** para essas linhas:

- No formato livre não há uma coluna específica para iniciar a linha. Pode-se começar a escrever o código a partir da coluna 1 e a linha de código pode se estender até a coluna 132. Além disso, os caracteres em branco são irrelevantes em qualquer lugar do código, exceto quanto estiverem sendo utilizados entre apóstrofes. Neste caso, cada caractere em branco será incluído na composição final de uma *constante de caracteres*.[3]

- Cada linha de texto é encerrada pelos registros *end-of-line* (EOL, *final de linha*) ou *carriage-return/line-feed* (CR/LF, *retorno-de-carro/nova-linha*), automaticamente inseridos pelo editor de texto e que usualmente permanecem invisíveis. Ao contrário de outras linguagens,

[3] Discutidas na seção 3.1.4.

como o C, não existe no Fortran um caractere especial para marcar o final de uma instrução ou declaração.

- Mais de uma instrução pode ser colocada na mesma linha. O **separador de instruções ou declarações** é o ponto e vírgula (;). Múltiplos “;” seguidos em uma linha, com ou sem brancos, são considerados como um separador simples. Desta forma, a seguinte linha de texto é interpretada como 3 linhas de código em sequência:

```
A = 0; B = 0 ; C = 0
```

  O caractere “;” não pode iniciar uma linha, exceto quando esta for a continuação de uma linha anterior.

- O caractere *ampersand* “&” é a **marca de continuação**; isto é, ele indica que a linha com instruções imediatamente posterior é a continuação da linha onde o “&” foi digitado. A continuação de linhas pode ser empregada para facilitar a leitura do código ou porque as 132 colunas disponíveis para a linha não foram suficientes para escrever toda a instrução. São permitidas até 255 linhas adicionais de código.

  Como exemplo, a linha de código

```
X = (-Y + ROOT_OF_DISCRIMINANT)/(2.0*A)
```

  também pode ser escrita com linhas adicionais usando o “&”:

```
X =                                 &
      (-Y + ROOT_OF_DISCRIMINANT)   &
      /(2.0*A)
```

  O caractere de continuação é usualmente (mas não necessariamente) o último caractere sintaticamente significativo não nulo em uma linha, exceto se for seguido pelo caractere de comentário (a seguir). Em certas situações, pode ser necessário continuar um token na próxima linha. Neste caso, além de se incluir “&” ao final da linha, o primeiro caractere não nula da próxima linha também deverá ser “&”. Isto usualmente ocorre com constantes de caractere. Por outro lado, nenhuma linha pode conter somente “&” ou ter “&” como o único caractere não nulo antes da *marca de comentários*.

- A **marca de comentários** é o ponto de exclamação “!” Para entrar com comentários em qualquer ponto do código-fonte, o usuário deve digitar o ponto de exclamação “!” em qualquer coluna de uma linha. Todo o restante da linha será desprezado pelo compilador. Por exemplo:

```
X = Y/A - B ! Soluciona a equação linear.
```

  Como um comentário sempre se estende até o final da linha, não é possível inserir comentários entre instruções em uma mesma linha.

2.5 O PROGRAMA “ALÔ MAMÃE”

Como primeiro exemplo de um programa em Fortran, considere o seguinte código-fonte:

Listagem 2.1: Primeiro programa em *Fortran*.

```
program primeiro
implicit none
print *, "Alô Mamãe"
end program primeiro
```

A discussão mais detalhada a respeito das unidades de programa do Fortran será realizada no capítulo 9. O exemplo acima ilustra a forma mais simples de um programa em Fortran, composta somente pelo **programa principal**. A estrutura do programa principal é a seguinte:

```
PROGRAM <nome_do_programa>
<declarações de nomes de variáveis>
<comandos executáveis>
END PROGRAM <nome_do_programa>
```

Comparando com o exemplo do programa `primeiro`, a declaração

```
PROGRAM primeiro
```

é sempre a primeira instrução de um programa. Ela serve para identificar o nome do código ao compilador.

Em seguida vêm as `<declarações de nomes de variáveis>`, as quais podem conter mais de uma linha. Neste ponto, são definidos os nomes das variáveis a ser usadas pelo programa. Caso não seja necessário declarar variáveis, como no programa `primeiro`, é recomendável incluir, pelo menos, a instrução `implicit none`. Esta instrui o compilador a exigir que todas as variáveis usadas pelo programa tenham o seu tipo explicitamente definido.[4]

Sugestões de uso & estilo para programação

É fortemente recomendado que a declaração `implicit none` seja sempre incluída. Desta forma, se for incluído algum nome (de variável, função, *etc*) que não foi explicitamente declarado, o compilador encerra o processamento do código-fonte emitindo uma mensagem de erro. O uso deste recurso torna o código-fonte *fortemente tipado* (*strong typed*) e pode evitar erros de programação potencialmente sérios.

Após declaradas as variáveis, vêm os `comandos executáveis`. No caso do programa `primeiro`, o único comando executável empregado foi

```
print *, "Alô Mamãe"
```

o qual tem como consequência a impressão do texto "Alô Mamãe" na tela do monitor, o qual é a chamada *saída padrão.*

Finalmente, o programa é encerrado com a instrução

```
end program primeiro
```

Os recursos utilizados no programa `primeiro` serão explicados com mais detalhes neste e nos próximos capítulos da apostila.

2.6 ENTRADA E SAÍDA PADRÕES

O Fortran possui três comandos de entrada/saída de dados. A maneira mais direta de definir valores de variáveis ou de exibir os valores destas é através dos dispositivos de entrada/saída padrões.

O dispositivo padrão de entrada de dados é o teclado. O comando de leitura para o programa ler os valores das variáveis é:

```
READ *, <lista de nomes de variáveis>
READ(*,*) <lista de nomes de variáveis>
```

onde na `lista de nomes de variáveis` estão listados os nomes das variáveis que deverão receber seus valores via teclado. O usuário deve entrar com os valores das variáveis separando-as por vírgulas.

O dispositivo padrão de saída de dados é a tela do monitor. Há dois comandos de saída padrão de dados:

```
PRINT *, ['<mensagem>'[,]][<lista de nomes de variáveis>]
WRITE(*,*) ['<mensagem>'[,]][<lista de nomes de variáveis>]
```

O programa a seguir instrui o computador a ler o valor de uma variável real a partir do teclado e, então, imprimir o valor desta na tela do monitor:

[4] Os tipos de variáveis suportados pelo Fortran são discutidos no capítulo 3.

```
program le_valor
implicit none
real :: a
print *, "Informe o valor de a:"
read *, a
print *, "Valor lido:",a
end program le_valor
```

No programa acima, foi declarada a variável real a, cujo valor será lido pelo computador, a partir do teclado, e então impresso na tela do monitor. É importante salientar que a instrução `implicit none` deve ser sempre a segunda linha de um programa, sub-programa ou módulo, aparecendo antes do restante das declarações de variáveis.

Esta seção apresentou somente um uso muito simples dos recursos de Entrada/Saída de dados. Uma descrição mais completa destes recursos será realizada no capítulo 10.

3

Tipos de Objetos de Dados

No Fortran, assim como em qualquer outra linguagem de programação, existe o conceito de **objetos de dados** (*data objects*). Um objeto de dado consiste em uma certa área na memória do computador (ou seja, uma dada sequência de bits) cuja informação será interpretada como tendo um valor de um determinado **tipo de dado** (*data type*) e o qual pode ser manipulado por meio de expressões ou operadores específicos que atuam sobre esse tipo de dado.

No Fortran há cinco **tipos intrínsecos** (*intrinsic types*) de objetos de dados: três tipos numéricos: *inteiro, real* e *complexo* (`INTEGER`, `REAL` e `COMPLEX`) e dois tipos não numéricos: *caractere* e *lógico* (`CHARACTER` e `LOGICAL`). O primeiro grupo de tipos de dados é utilizado em operações matemáticas e é o mais utilizado. O segundo grupo é utilizado para operações que envolvem manipulações de texto ou operações lógicas.

Um exemplo de uma quantidade inteira é `10` e de uma quantidade real é `10.0`. Na sintaxe do Fortran, quando o valor constante de um tipo de dado é escrito diretamente (como nos exemplos acima), ao invés de ser referenciado por um nome, este é denominado uma **constante literal** (*literal constant*).

No Fortran, cada um dos cinco tipos intrínsecos possui um valor inteiro não negativo denominado *parâmetro de espécie do tipo* (*kind type parameter*). A norma da linguagem determina que qualquer processador deve suportar, para cada tipo intrínseco, uma espécie padrão e um número adicional de outras espécies (dependente do processador e do compilador).

Existe também o recurso de definir novos tipos de variáveis, baseados nos tipos intrínsecos. Estes são os chamados *tipos derivados* (*derived types*). Outras linguagens como C/C++ se referem a estes objetos por *estruturas* (*structures*). Os tipos derivados consistem em combinações de partes compostas por tipos intrínsecos e/ou por outros tipos derivados, permitindo gerar objetos de dados de complexidade crescente.

Neste capítulo, serão apresentadas as formas de declarações dos tipos intrínsecos de dados, seguida da definição das espécies mais comuns. As formas aqui discutidas servirão para definir objetos **escalares** de dados. O termo escalar significa que cada objeto armazena um único valor do tipo declarado, em contraste com objetos *compostos*, os quais podem armazenar mais de um valor. Os tipos derivados, cuja definição será apresentada na seção 3.4, são exemplo de objetos composto de dados. *Matrizes* (*arrays*) são outros exemplos de objetos compostos. Matrizes, as quais serão discutidas no capítulo 6, podem ser definidas tanto com tipos intrínsecos quanto com tipos derivados. Finalmente, no capítulo 7 serão discutidos *objetos dinâmicos de dados*, tais como matrizes alocáveis e ponteiros.

Sugestões de uso & estilo para programação

Como já foi colocado na seção 2.5, é uma boa praxe de programação iniciar o setor de declarações de variáveis com a instrução

```
IMPLICIT NONE
```

a qual impede a possibilidade de haver nomes de objetos de dados não declarados, os quais iriam desta maneira possuir o seu tipo implícito, uma prática corriqueira em Fortran 77.

3.1 TIPOS INTRÍNSECOS DE DADOS

Serão abordados agora os tipos intrínsecos no Fortran. Inicialmente serão abordadas as formas mais comuns de declarações dos tipos intrínsecos. A forma mais geral de declaração será apresentada na seção 3.2.

3.1.1 TIPO INTEGER

O tipo inteiro de dados é composto por objetos que possuem valores que pertencem ao conjunto dos números inteiros.

Exemplos de literais inteiros válidos são:

123, 89312, 5

A declaração básica de nomes de objetos de dados de tipo inteiro é:

```
INTEGER[, <lista-atributos>] :: <lista-nomes-objetos>
```

O programa exemplo a seguir ilustra diversas situações envolvendo objetos do tipo inteiro:

```
program inteiro
implicit none
integer, parameter :: dois= 2 ! Declaração de uma constante nomeada
integer :: v2= dois           ! Declaração de uma variável inicializada
integer :: x                  ! Declaração de uma variável não inicializada

print*, 'Valor da constante dois:', dois
print*, 'Valor da variável v2:', v2
! O valor de x será agora lido do teclado.
print*, 'Valor de x:'
read *, x ! Entre um número inteiro no cursor.
! O valor digitado não pode conter ponto (.) Caso isto
! aconteça, pode ocorrer um erro de execução do programa,
! abortando o mesmo.
print*, "Valor lido:",x
end program inteiro
```

3.1.2 TIPO REAL

Um objeto do tipo real é composto de até quatro partes, assim dispostas:

1. uma parte inteira, com ou sem sinal,
2. um ponto decimal,
3. uma parte fracionária e
4. um expoente, também com ou sem sinal.

Um ou ambos os itens 1 e 3 devem estar presentes. O item 4 ou está ausente ou consiste na letra `E` seguida por um inteiro com ou sem sinal. Um ou ambos os itens 2 e 4 devem estar presentes. Exemplos de literais reais são:

`-10.6E-11` (representando $-10,6 \times 10^{-11}$)
`1.`
`-0.1`
`1E-1` (representando 10^{-1} ou 0,1)
`3.141592653`

Os literais reais são representações do conjunto dos números reais e o padrão da linguagem não especifica o intervalo de valores aceitos para o tipo real e nem o número de dígitos significativos na parte fracionária (item 3) que o processador suporta, uma vez que estes valores dependem do tipo de processador em uso. Valores comuns são: intervalo de números entre 10^{-38} a 10^{+38}, com uma precisão de cerca de 7 (sete) dígitos decimais.

A declaração básica do tipo real é:

```
REAL[, <lista-atributos>] :: <lista-nomes-objetos>
```

PROGRAMA EXEMPLO:

```
program var_real
implicit none
real :: a, b= 10.5e-2           ! Variável b é inicializada a 10.5e-2.
real, parameter :: pi= 3.141593 ! Constante pi aproximada.
print *, 'Valor de a:'
read *, a
print *, 'Valor de a:', a
print *, 'Valor de b:', b
print *, 'Valor de pi:', pi
end program var_real
```

3.1.3 TIPO COMPLEX

O Fortran, como uma linguagem concebida para realização de cálculos científicos, tem a vantagem de possuir um terceiro tipo numérico intrínseco: números complexos. Este tipo é concebido como um par de literais, os quais são ou inteiros ou reais, separados por vírgula "," e contidos entre parênteses "(" e ")". Os literais complexos representam números contidos no conjunto dos números complexos, isto é, números do tipo $z = x + iy$, onde $i = \sqrt{-1}$, x é a parte real e y é a parte imaginária do número complexo z. Assim, um literal complexo deve ser escrito:

`(<parte real>,<parte imaginária>)`

Exemplos de literais complexos são:

`(1.,3.2)` (representando $1 + 3,2i$)
`(1.,.99E-2)` (representando $1 + 0,99 \times 10^{-2}i$)
`(1.0,-3.7)`

Uma outra grande vantagem do Fortran é que toda a álgebra de números complexos já está implementada no padrão. Assim, se for realizado o produto de dois números complexos `(x1,y1)` e `(x2,y2)`, o resultado será o literal complexo dado por `(x1*x2 - y1*y2,x1*y2 + x2*y1)`. O mesmo acontecendo com as outras operações algébricas.

A declaração básica do tipo complexo é:
`COMPLEX[, <lista-atributos>] :: <lista-nomes-objetos>`

As partes real ou imaginária de uma variável complexa podem ser acessadas individualmente de duas maneiras: ou por meio das funções `real()` e `imag()`, ou por meio dos sufixos `%re` ou `%im` agregados ao nome da variável complexa.

PROGRAMA EXEMPLO:

```
program var_complexa
implicit none
complex :: a= (5,-5),b,c ! Variável a é inicializada a (5,-5).
complex, parameter :: d= (6.0,-6.0)
print *, "Valor de b:"
! O valor de b deve ser entrado como um literal complexo.
! Exemplo: (-1.5,2.5)
```

```
read *, b
c= a*b
print *, "O valor de c= a*b:", c ! Verifique o resultado no papel.
! Acesso individual às partes real ou imaginária:
print*, 'Parte real:       Re(c)=', real(c)  !Ou: c%re (versão 9 GFortran)
print*, 'Parte imaginária: Im(c)=', aimag(c) !Ou: c%im (versão 9 GFortran)

c= b/d
print *, "O valor de c= b/d:", c ! Verifique o resultado no papel.
end program var_complexa
```

3.1.4 TIPO CHARACTER

Este tipo de objeto de dados consiste em um conjunto de caracteres em sequência, referidos como uma ***string***. Os caracteres da string não estão restritos ao conjunto de caracteres padrão definidos na seção 2.1; qualquer caractere que possa ser representado pelo processador é aceito, exceto os caracteres de controle tais como o *return*.

Os literais do tipo `CHARACTER` estão contidos entre um par de apóstrofes ou aspas. Os apóstrofes ou aspas servem como *delimitadores* dos literais de caractere e não são considerados parte integrante do conjunto. Ao contrário das normas usuais, um espaço em branco é diferente de dois ou mais. Exemplos de diferentes literais de caractere:

```
'bom Dia'
'bomDia'
'BRASIL'
“Fortran Moderno”
```

Os delimitadores podem fazer parte de uma string de duas maneiras:

1. Uma string que contenha um delimitador de um tipo deve ser delimitada pelo outro tipo. Por exemplo, em

   ```
   'Ele disse “Bom dia!”'
   “Eu respondi 'por quê?'”
   ```

2. Um delimitador duplo sem espaços em branco é considerado um único caractere do mesmo. Por exemplo,

   ```
   'Tromba d”água'
   ```

   é o literal `Tromba d'água`

A declaração mais utilizada para o tipo caractere é:

```
character[(len=<char-len>)][, <lista-atributos>] :: <lista-nomes-objetos>
```

onde `<char-len>` indica a quantidade de caracteres contidos nas strings, ou seja, o seu *comprimento*. Todas as strings definidas por esta declaração têm o mesmo número de caracteres. Se for informado um literal maior que `<char-len>`, este será truncado; se for informado um literal menor que o declarado, o processador irá preencher o espaço restante à direita com espaços em branco. Se a instrução `(len=<char-len>)` estiver ausente, as strings serão assumidas de comprimento 1.

Uma constante nomeada do tipo `CHARACTER` não precisa ter seu comprimento explicitamente declarado, uma vez que o mesmo será tomado (ou *assumido*) diretamente do valor da constante, o qual será conhecido no momento da compilação do código. Neste caso, pode-se usar o qualificador `LEN=*`, como é exemplificado no programa a seguir.

PROGRAMA EXEMPLO:

```
program caracteres
implicit none
character, parameter :: c1= 'v' ! Constante de caractere de comprimento 1.
character(len=*), parameter :: tda= 'Tromba d''água' ! Const. comp. assumido
character(len=10) :: str_read

print *, 'Imprimindo strings na tela:'
print *, 'As strings estão contidas entre os símbolos "->|" e "|<-"'
print *, 'Constante c1: ->|', c1, '|<-'
print *, 'Constante tda: ->|', tda, '|<-'
print *, "Entre com texto (sem aspas ou apóstrofes):"
read '(a)', str_read
print *, "Texto lido: ->|", str_read, "|<-"
end program caracteres
```

É importante mencionar aqui a regra particular para o formato de fonte dos literais de caractere que são escritos em mais de uma linha:

1. Cada linha deve ser encerrada com o caractere "&" e não pode ser seguida por comentário.
2. Cada linha de continuação deve ser precedida também pelo caractere "&".
3. Este par "&&" não faz parte do literal.
4. Quaisquer brancos seguindo um & em final de linha ou precedendo o mesmo caractere em início de linha não são partes do literal.
5. Todo o restante, incluindo brancos, fazem parte do literal.

Como exemplo temos:

```
poema_Fernando_Pessoa =                    &
        'O tempo que eu hei sonhado        &
        & Quantos anos foi de vida!        &
        & Ah, quanto do meu passado        &
        & Foi só a vida mentida            &
        & De um futuro imaginado!          &
        &                                  &
        & Aqui à beira do rio              &
        & Sossego sem ter razão.           &
        & Este seu correr vazio            &
        & Figura, anônimo e frio,          &
        & A vida vivida em vão.'
```

O padrão permite também que variáveis de caracteres com diferentes comprimentos sejam declarados na mesma declaração. Neste caso, cada nome na <lista-nomes-objetos> fica

```
<nome-var>[*<char-len>][ = <expr-const>]
```

Por exemplo,

```
character(len= 8) :: word, point*1, text*4
```

declara as variáveis word, point e text como strings com 8, 1 e 4 caracteres, respectivamente.

3.1.5 TIPO LOGICAL

O tipo lógico define variáveis lógicas. Uma variável lógica só pode assumir dois valores, *verdadeiro* ou *falso*. A representação dos dois valores possíveis de uma variável lógica são:

- .TRUE. ⇒ Verdadeiro
- .FALSE. ⇒Falso.

As declarações do tipo lógico são:
logical :: <lista-nomes-objetos>

PROGRAMA EXEMPLO:

```
program logico
implicit none
logical :: a= .true.
if (a) then ! Bloco IF, discutido na seção 5.1
   print*, "A variável é verdadeira."
end if
end program logico
```

3.2 FORMA GERAL DA DECLARAÇÃO DE TIPO DE OBJETO DE DADOS

A forma geral de uma declaração de tipo de dados é:

```
<tipo>[([KIND=]<parâmetro-espécie>)][, <lista-atributos>] ::  <lista-nomes-objetos>
```

onde `<tipo>` especifica o tipo de variável e `<parâmetro-espécie>` especifica a espécie da variável.

O campo `<lista-atributos>` é uma lista (opcional) de **atributos do tipo de dados**, ou simplesmente atributos, os quais são propriedades especiais atribuídas aos nomes contidos na `<lista-nomes-objetos>`. A `<lista-atributos>` pode conter os seguintes atributos, os quais serão abordados ao longo desta Apostila:

allocatable	dimension(<lista-ext>)	parameter	save
aynchronous	external	pointer	target
binc(c...)	intent(inout)	private	value
codimension(lista-cobounds)	intrinsic	protected	volatile
contiguous	optional	public	

Por sua vez, cada objeto na `<lista-nomes-objetos>` pode ser um dos seguintes:

```
<nome-obj>[(<lista-ext>)][(<lista-cobounds>)][*<char-len>][ = <expr-const>]
<nome-função>[*<char-len>]
<nome-ponteiro>[(<lista-bounds>)][*<char-len>][ => null-init]
```

onde `<nome-obj>` é o nome do objeto, seguindo a regra de nomes válidos definida na seção 2.3. A presença ou ausência do campo opcional `<expr-const>` irá definir o status do objeto.

Caso o campo `<expr-const>` esteja ausente, o status do objeto declarado é **indefinido**. O padrão não regula qual deve ser o valor inicial de um objeto indefinido e diferentes compiladores inicializam os mesmos de distintas maneiras. Por esta razão, o programador deve tomar cuidado para não fazer referência ao valor de um objeto indefinido antes de atribuir algum valor ao mesmo (o objeto não deve aparecer em uma expressão,[1] por exemplo).

Caso o campo `<expr-const>` exista, este irá realizar a **inicialização** do valor deste objeto, quando então este assumirá o status de **definido**. Esta inicialização pode ser realizada de três maneiras distintas:

1. Por meio de uma constante literal.
2. Por meio de uma *constante nomeada* (ver abaixo). Uma constante nomeada somente pode ser empregada na declaração de um outro objeto de dados se a sua declaração foi realizada em uma linha anterior do código-fonte.
3. Finalmente, `<expr-const>` pode ser uma expressão que resulta em um literal. Como expressões são desenvolvidas nos diversos tipos de objetos de dados do Fortran será discutido no capítulo 4.

Se o tipo do literal, constante nomeada ou expressão em `<expr-const>` for distinto do tipo do objeto sendo declarado, ocorrerá uma conversão de tipo, também discutida no capítulo 4.

Uma **constante nomeada** (*named constant*) é um valor constante com um nome próprio. A declaração de uma constante nomeada é realizada por intermédio do atributo `parameter`, o qual é obrigatório neste caso:

[1] Ver capítulo 4.

```
<tipo>[([KIND=]<parâmetro-espécie>)], parameter :: <lista-ctes-nomeadas>
```

Podendo conter outros atributos opcionais. Cada nome na <lista-ctes-nomeadas> é do tipo

```
<nome-obj>= <expr-const>
```

Portanto, a declaração de um constante nomeada deve ser necessariamente conter <expr-const>.

O valor de uma constante nomeada não pode ser alterado ao longo do programa. Por outro lado, se o atributo PARAMETER não for empregado, o objeto declarado (com ou sem inicialização) é uma **variável** e o seu valor pode ser atribuído ou modificado em qualquer ponto do código-fonte após a seção de declarações.

Os campos <parâmetro-espécie>, <lista-atributos>, <nome-função> e <nome-ponteiro> serão estudados ao longo deste e dos demais capítulos. A seguir, será discutido o conceito de espécie de um tipo de objeto de dados.

3.3 O CONCEITO DE ESPÉCIE *(kind)*

Além das representações apresentadas na seção 3.1 para os 5 tipos intrínsecos, é possível definir extensões a um determinado tipo, cujas declarações dependem do compilador que irá criar o executável e do processador no qual o programa será executado. Estas extensões são selecionadas determinando-se o qualificador KIND= <parâmetro de espécie> contido na forma geral de uma declaração de tipo, apresentada na seção 3.2.

Cada um dos cinco tipos intrínsecos, INTEGER, REAL, COMPLEX, CHARACTER e LOGICAL possui associado a si pelo menos um valor inteiro não negativo denominado *parâmetro de espécie do tipo* (*kind type parameter*), o qual será a espécie padrão para o tipo. Escolhendo-se outros valores (se estes forem suportados) deste parâmetro para um dado tipo de dado, a extensão da área de memória reservada para o mesmo irá mudar. Não serão abordados aqui os modelos para as representações dos tipos de dados.[2] Para o momento, basta mencionar que uma dada espécie de um tipo de dado ocupa um número inteiro de *bytes* (1 byte = 8 bits) na memória do computador. Alterando-se o parâmetro da espécie, o número reservado de bytes muda, mas não o tipo do dado; a informação contida nesses bytes será sempre interpretada e manipulada de acordo com os procedimentos reservados para esse tipo de dado em particular.

Os valores da espécie e da área de memória reservada são dependentes do processador e/ou do compilador empregado. O padrão exige que cada compilador suporte no mínimo duas espécies para os tipos REAL e COMPLEX e uma espécie para os tipos INTEGER, CHARACTER e LOGICAL. Contudo, há funções intrínsecas fornecidas pela linguagem que definem ou verificam as precisões suportadas pelo processador e que podem ser usadas para definir o valor do parâmetro KIND, possibilitando assim a portabilidade do código, isto é, a possibilidade deste rodar em diferentes arquiteturas usando uma precisão mínima especificada pelo programador.

Para demonstrar como diferentes compiladores implementam e usam o parâmetro de espécie, serão considerados os compiladores Intel® Fortran Compiler e gfortran, ambos nas versões apresentadas na seção 1.9.

3.3.1 COMPILADOR INTEL® FORTRAN

O compilador Intel® Fortran oferece os seguintes tipos intrínsecos, juntamente com as respectivas declarações de tipo e espécie:

Tipo Inteiro. Há 04 parâmetros de espécie para o tipo inteiro.

> **Declaração:** INTEGER([KIND=]<n>) [::] <lista-nomes-objetos>
>
> Sendo <n>= 1, 2, 4 ou 8. Se o parâmetro de espécie é explicitado, as variáveis na <lista-nomes-objetos> serão da espécie escolhida. Em caso contrário, a espécie será o **inteiro padrão: INTEGER(KIND=4)**; ou seja, a declaração INTEGER :: A equivale a INTEGER(KIND=4) :: A.

[2] Isto será realizado na seção 8.8.1 para os tipos numéricos.

Listagem 3.1: Testa distintas espécies suportadas pelo compilador Intel® Fortran.

```
program testa_kind_intel
implicit none
integer, parameter :: dp= 8, qp= 16
real :: r_simples
real(kind= dp) :: r_dupla
real(kind= qp) :: r_quad
!
!Calcula a raiz quadrada de 2 em diversas precisões.
r_simples= sqrt(2.0)    ! Precisão simples
r_dupla= sqrt(2.0_dp)   ! Precisão dupla
r_quad= sqrt(2.0_qp)    ! Precisão quádrupla ou estendida.
!
!Imprime resultados na tela.
print *, r_simples, precision(r_simples)
print *, r_dupla, precision(r_dupla)
write(*,'(2x, f29.26)') r_dupla
print *, r_quad, precision(r_quad)
write(*,'(2x, f46.43)')r_quad
!
end program testa_kind_intel
```

Tipo Real. Há 03 parâmetros de espécie para o tipo real.

Declaração: `REAL([KIND=]<n>) [::] <lista-nomes-objetos>`

Sendo `<n>= 4, 8` ou `16`. Caso o parâmetro de espécie não seja especificado, a espécie será o **real padrão: `REAL(KIND= 4)`**.

Tipo Complexo. Há 03 parâmetros de espécie para o tipo complexo.

Declaração: `COMPLEX([KIND=]<n>) [::] <lista-nomes-objetos>`

Sendo `<n>= 4, 8` ou `16`. Caso o parâmetro de espécie não seja explicitado, a espécie será o **complexo padrão: `COMPLEX(KIND= 4)`.**

Tipo Lógico. Há 04 parâmetros de espécie para o tipo lógico.

Declaração: `LOGICAL([KIND=]<n>) [::] <lista-nomes-objetos>`

Sendo `<n>= 1, 2, 4` ou `8`. **Lógico padrão: `LOGICAL(KIND= 4)`.**

Tipo Caractere. Há somente uma espécie do tipo caractere.

Declaração: `CHARACTER([KIND=1],[LEN=]<comprimento>) [::] <lista-nomes-objetos>`

Cada espécie distinta ocupa um determinado espaço de memória na CPU. Para o compilador `Intel® Fortran`, o espaço ocupado está descrito na tabela 3.1.

Tabela 3.1: Tabela de armazenamento de variáveis para o compilador *Intel® Fortran*.

Tipo e Espécie	Extensão (bytes)	Tipo e Espécie	Extensão (bytes)
INTEGER(KIND=1)	1=8 bits	LOGICAL(KIND=1)	1
INTEGER(KIND=2)	2	LOGICAL(KIND=2)	2
INTEGER(KIND=4)	4	LOGICAL(KIND=4)	4
INTEGER(KIND=8)	8	LOGICAL(KIND=8)	8
REAL(KIND=4)	4	COMPLEX(KIND=4)	8
REAL(KIND=8)	8	COMPLEX(KIND=8)	16
REAL(KIND=16)	16	COMPLEX(KIND=16)	32

O programa 3.1 a seguir ilustra do uso e as diferenças de algumas opções de espécies de tipos de variáveis. O mesmo exemplo também ilustra o uso da função implícita `SQRT(X)`:[3]

[3]Definida na seção 8.4.

```
R_SIMPLES= SQRT(2.0)
```

a qual calculou a raiz quadrada da constante `2.0` e atribuiu o resultado à variável `R_SIMPLES`.

3.3.2 COMPILADOR GFORTRAN

`Gfortran` é o compilador Fortran da GNU (Fundação *Gnu is Not Unix*), inicialmente desenvolvido como alternativa ao compilador `f95` distribuído pelas versões comerciais do Unix. Atualmente, o `gfortran` é parte integrante da plataforma de desenvolvimento de software `GCC` (`GNU Compiler Collection`), que é composta por compiladores de diversas linguagens distintas, tais como Fortran, C/C++, Java, Ada, entre outras. O comando `gfortran` consiste simplesmente em um *script* que invoca o programa `f951`, o qual traduz o código-fonte para assembler, invocando em seguida o linkador e as bibliotecas comuns do pacote `GCC`.

No `gfortran`, os parâmetros de espécie são determinados de forma semelhante ao compilador `Intel® Fortran`, discutido na seção 3.3.1, ou seja, o parâmetro indica o número de bytes necessários para armazenar cada variável da respectiva espécie de tipo. As espécies suportadas pelo `gfortran` são descritas na tabela 3.2. Pode-se notar que o `gfortran` suporta mais espécies que o compilador da Intel.

Tabela 3.2: Tabela de armazenamento de variáveis para o compilador *gfortran* da fundação *GNU*.

Tipo e Espécie	Extensão (bytes)	Tipo e Espécie	Extensão (bytes)
INTEGER(KIND=1)	1=8 bits	LOGICAL(KIND=1)	1
INTEGER(KIND=2)	2	LOGICAL(KIND=2)	2
INTEGER(KIND=4)	4	LOGICAL(KIND=4)	4
INTEGER(KIND=8)	8	LOGICAL(KIND=8)	8
INTEGER(KIND=16)[4]	16	LOGICAL(KIND=16)	16
REAL(KIND=4)	4	COMPLEX(KIND=4)	8
REAL(KIND=8)	8	COMPLEX(KIND=8)	16
REAL(KIND=10)	10	COMPLEX(KIND=10)	20
REAL(KIND=16)	16	COMPLEX(KIND=16)	32
CHARACTER(KIND=1)	1	CHARACTER(KIND=4)	4

O programa `testa_kind_gfortran` a seguir (programa 3.2) ilustra o uso e as diferenças entre as diversas espécies de tipos de dados no compilador `gfortran`. No mesmo programa, observa-se como é possível realizar a inicialização de variáveis com valores resultantes do uso de funções intrínsecas que serão definidas no capítulo 8.

3.3.3 LITERAIS DE DIFERENTES ESPÉCIES

Os programas nas listagens 3.1 e 3.2 mostram que para se distinguir as espécies dos literais (ou constantes) dentre diferentes parâmetros fornecidos ao compilador, utiliza-se o sufixo _<k>, sendo <k> o parâmetro da espécie do tipo:

Literais inteiros. Constante inteiras, incluindo a espécie, são especificadas por:
`[<s>]<nnn...>[_<k>]`
onde: <s> é um sinal (+ ou −); obrigatório se negativo, opcional se positivo. <nnn...> é um conjunto de dígitos (0 a 9); quaisquer zeros no início são ignorados. Já _<k> é um dos parâmetros de espécie do tipo; esta opção explicita a espécie do tipo à qual o literal pertence.

Literais reais. Constantes reais são especificadas de diferentes maneiras, dependendo se possuem ou não parte exponencial. A regra básica para a especificação de um literal real já foi definida na seção 3.1.2. Para explicitar a espécie do tipo real à qual o literal pertence, deve-se incluir o sufixo _<k>, onde <k> é um dos parâmetros de espécie do tipo. Por exemplo, `2.0_8`: para indicar literal real, espécie 8.

Literais complexos. A regra básica para a especificação de um literal complexo já foi definida na seção 3.1.3. Para explicitar a espécie do tipo à qual o literal pertence, deve-se incluir

[4] Em plataformas de 64 bits, tais como a família de processadores Intel®.

Listagem 3.2: Testa distintas espécies suportadas pelo compilador *gfortran*.

```
program testa_kind_gfortran
implicit none
integer, parameter :: i1= 1, i2= 2, i4= 4, i8= 8, i16= 16
integer, parameter :: dp= 8, lp= 10, qp= 16
!Inicializa as variáveis inteiras.
!A função intrínseca huge() é definida na seção 8.8.2.
integer(kind= i1) ::  vi1= huge(1_i1)
integer(kind= i2) ::  vi2= huge(1_i2)
integer(kind= i4) ::  vi4= huge(1_i4)
integer(kind= i8) ::  vi8= huge(1_i8)
integer(kind= i16) :: vi16= huge(1_i16)
!Inicializa as variáveis reais.
real ::           r_simples= huge(1.0)     !Precisão simples
real(kind= dp) :: r_dupla=   huge(1.0_dp)  !Precisão dupla
real(kind= lp) :: r_long=    huge(1.0_lp)  !Precisão longa (long double em C)
real(kind= qp) :: r_quad=    huge(1.0_qp)  !Precisão estendida
!Inicializa as variáveis complexas,
!calculando a raiz quadrada de z = 2 + 2i em diversas precisões.
complex ::           c_simples= sqrt((2.0,2.0))
complex(kind= dp) :: c_dupla=   sqrt((2.0_dp,2.0_dp))
complex(kind= lp) :: c_long=    sqrt((2.0_lp,2.0_lp))
complex(kind= qp) :: c_quad=    sqrt((2.0_qp,2.0_qp))

!Mostra os maiores números representáveis do tipo inteiro.
print*, 'Espécies Inteiras, valores máximos:'
print*, vi1, vi2, vi4
print*, vi8
print*, vi16

!Mostra os maiores números representáveis do tipo real.
print*, ''
print*, 'Espécies Reais, valores máximos:'
print *, r_simples
print *, r_dupla
print *, r_long
print *, r_quad

!Mostra os valores de z = sqrt(2 + 2i) nas diversas precisões.
print*, ''
print*, 'Espécies Complexas, representações de sqrt(2,2):'
print*, c_simples
print*, c_dupla
print*, c_long
print*, c_quad
!
end program testa_kind_gfortran
```

o sufixo _<k>, onde <k> é um dos parâmetros de espécie do tipo, em cada uma das partes real e imaginária do literal complexo. Por exemplo, `(1.0_8,3.5345_8)`: para indicar tipo complexo, espécie 8.

Literais lógicos. Uma constante lógica pode tomar uma das seguintes formas:
`.TRUE.[_<k>]`
`.FALSE.[_<k>]`
onde <k> é um dos parâmetros de espécie do tipo.

Já para literais do tipo `CHARACTER` a sintaxe é a oposta. Se <k> é o parâmetro de espécie de um determinado conjunto de caracteres suportados pelo processador, e o literal 'a' é suportado pelas espécies `k=1` e `k=4`, então `1_'a'` ou `2_'a'` seleciona este caractere nos dois conjuntos ou espécies.

3.3.4 RECURSOS RELACIONADOS ÀS ESPÉCIES DE TIPO

O padrão oferece diversos recursos que fornecem informações ou que auxiliam na determinação das espécies suportadas por uma dada instalação.

3.3.4.1 FUNÇÕES INTRÍNSECAS ASSOCIADAS À ESPÉCIE

Embora as declarações de espécie do tipo possam variar para diferentes compiladores, o padrão da linguagem estabelece um conjunto de funções intrínsecas que facilitam a determinação e a declaração destas espécies de uma forma totalmente portável, isto é, independente de compilador e/ou arquitetura.

As funções intrínsecas descritas nesta e nas subsequentes seções serão novamente abordadas no capítulo 8, onde serão apresentadas todas as rotinas intrínsecas fornecidas pelo padrão da linguagem Fortran.

KIND(X)

A função intrínseca `KIND(X)`,[5] a qual tem como argumento uma variável ou constante de qualquer tipo intrínseco, retorna o valor inteiro que identifica a espécie da variável `X`. Por exemplo,

```
program tes_fun_kind
implicit none
integer :: i,j
integer, parameter :: dp= 2
real :: y
real(kind= dp) :: x
!
i= kind(x) ; j= kind(y)
print*, i
print*, j
end program tes_fun_kind
```

Outros exemplos:

```
KIND(0)        ! Retorna a espécie padrão do tipo inteiro.
               ! (Dependente do processador).
KIND(0.0)      ! Retorna a espécie padrão do tipo real.
               ! (Depende do processador).
KIND(.FALSE.)  ! Retorna a espécie padrão do tipo lógico.
               ! (Depende do processador).
KIND('A')      ! Fornece a espécie padrão de caractere.
               ! (Sempre igual a 1).
KIND(0.0D0)    ! Usualmente retorna a espécie do tipo real de precisão dupla.
               ! (Pode não ser aceito por todos compiladores).
```

[5] Definida na seção 8.2.

SELECTED_REAL_KIND(P,R)

A função intrínseca SELECTED_REAL_KIND(P,R)[6] tem dois argumentos: P e R. A variável P especifica a precisão (número de dígitos decimais) mínima requerida e R especifica o intervalo de variação mínimo da parte exponencial da variável, também no sistema decimal.

A função SELECTED_REAL_KIND(P,R) retorna o valor da espécie que satisfaz, ou excede minimamente, os requerimentos especificados por P e R. Se mais de uma espécie satisfizer estes requerimentos, o valor retornado é aquele com a menor precisão decimal. Se a precisão requerida não for disponível, a função retorna o valor -1; se o intervalo da exponencial não for disponível, a função retorna -2 e se nenhum dos requerimentos for disponível, o valor -3 é retornado.

Esta função, usada em conjunto com a declaração de espécie, garante uma completa portabilidade ao programa, desde que o processador tenha disponíveis os recursos solicitados. O exemplo a seguir ilustra o uso desta função intrínseca e outros recursos.

```
program tes_selected
integer, parameter :: i10= selected_real_kind(10,200)
integer, parameter :: dp= 8
real(kind= i10) :: a,b,c
real(kind= dp) :: d
print*, i10
a= 2.0_i10
b= sqrt(5.0_i10)
c= 3.0e10_i10 ,
d= 1.0e201_dp
print*, a
print*, b
print*, c
print*, d
end program tes_selected
```

Pode-se ver que a precisão requerida na variável I10 é disponível na espécie correspondente à precisão dupla de uma variável real para os compiladores mencionados (ver, por exemplo, tabela 3.2). Um outro recurso disponível é a possibilidade de especificar constantes de uma determinada espécie, como na atribuição

```
A= 2.0_I10
```

A constante A foi explicitamente especificada como pertencente à espécie I10 seguindo-se o valor numérico com um *underscore* e com o parâmetro de espécie do tipo (I10). Deve-se notar também que a definição da espécie I10, seguida da declaração das variáveis A, B e C como sendo desta espécie, determina o intervalo *mínimo* de variação da parte exponencial destas variáveis. Se o parâmetro da espécie associada à constante I10 for distinto do parâmetro DP, a variável D não poderia ter sido declarada também da espécie I10, pois a atribuição

```
D= 1.0E201_I10
```

iria gerar uma mensagem de erro no momento da compilação, uma vez que a parte exponencial excede o intervalo definido para a espécie I10 (200). Contudo, para alguns compiladores, como o gfortran, a compilação irá resultar em I10 = DP, tornando desnecessária uma das declarações acima. Este exemplo demonstra a flexibilidade e a portabilidade propiciada pelo uso da função intrínseca SELECTED_REAL_KIND.

SELECTED_INT_KIND(R)

A função intrínseca SELECTED_INT_KIND(R)[7] é usada de maneira similar à função intrínseca SELECTED_REAL_KIND. Agora, a função tem um único argumento R, o qual especifica o intervalo de números inteiros requerido. Assim, SELECTED_INT_KIND(r) retorna o valor da espécie que representa, no mínimo, valores inteiros no intervalo -10^r a $+10^r$. Se mais de uma espécie

[6] Definida na seção 8.8.5.
[7] Também definida na seção 8.8.5.

satisfizer o requerimento, o valor retornado é aquele com o menor intervalo no expoente r. Se o intervalo não for disponível, o valor -1 é retornado.

O exemplo a seguir mostra a declaração de um inteiro de uma maneira independente do sistema:

```
INTEGER, PARAMETER :: I8= SELECTED_INT_KIND(8)
INTEGER(KIND= I8) :: IA, IB, IC
```

As variáveis inteiras `IA`, `IB` e `IC` podem ter valores entre -10^8 a $+10^8$ *no mínimo*, se disponível pelo processador.

SELECTED_CHAR_KIND(NOME)

A função intrínseca `SELECTED_CHAR_KIND(NOME)`,[8] por sua vez, fornece o parâmetro da espécie correspondente ao conjunto de caracteres identificado por `NOME`. O padrão exige o suporte a somente um conjunto de caracteres, o qual será então acessado por `nome` igual a `'default'`. Os dois conjuntos de caracteres com o maior suporte são o ASCII e o ISO 10646, os quais serão brevemente discutidos.

CONJUNTO DE CARACTERES ASCII. Usualmente os compiladores fornecem acesso à sequência de intercalação original da tabela ASCII[9] através de funções intrínsecas (seção 8.6.1), muito embora este não seja o obrigatoriamente o conjunto de caracteres padrão. Neste caso, o parâmetro da espécie também é obtido com `NOME` igual a `'ascii'`.

O ASCII consiste em uma padronização para um *encodeamento de caracteres* (*Character encoding*) de 7 bits (originalmente), baseado no alfabeto inglês. Ou seja o conjunto ASCII contém 128 caracteres. Códigos ASCII representam textos em computadores, equipamentos de comunicação e outros dispositivos eletrônicos que trabalham com texto. A tabela 3.3 mostra os caracteres ASCII originais, juntamente com os seus identificadores nos sistemas numéricos decimal, hexadecimal e octal, bem como o código html correspondente. Os primeiros 32 (0 – 31) caracteres mais o caractere 127 são caracteres de controle, destinados à comunicação com periféricos; estes não geram caracteres que podem ser impressos. Já os caracteres restantes (32 – 126) compõe-se de caracteres alfanuméricos mais caracteres especiais.

CONJUNTO DE CARACTERES ISO 10646. A norma internacional IOS/IEC 10646 UCS-4[10] estabeleceu um conjunto de caracteres ocupando 4 bytes, com o ambicioso objetivo de suportar todos os caracteres em todas as linguages no mundo mais todos os caracteres especiais em outros conjuntos. O padrão estabelece o valor de `NOME` igual a `'iso_10646'` para que a função `SELECTED_CHAR_KIND(NOME)` retorne o parâmetro de espécie deste conjunto; ou -1 se não for suportado, como é o caso do compilador Intel.

O conjunto ASCII corresponde ao subconjunto ocupando os primeiros 128 caracteres da tabela ISO 10646. Por esta razão, atribuindo um caractere ASCII a uma variável da espécie ISO 10646 é aceito e o caractere é convertido corretamente. Por outro lado, atribuindo-se um caractere ISO 10646 a uma variável ASCII ou de outra espécie também é aceito, mas o resultado será dependente do processador, caso o caractere não seja suportado.

O programa 3.3 testa o suporte dos conjuntos de caracteres ASCII e ISO 10646. O compilador Intel Fortran irá abortar na linha 7, gerando uma mensagem de erro, porque o conjunto ISO 10646 não é suportado.

Listagem 3.3: Testa o suporte dos conjuntos de caracteres ASCII e ISO 10646.

```
1 program testa_selected_char_kind
2 implicit none
3 integer, parameter :: ascii= selected_char_kind('ascii')
4 integer, parameter :: iso10646= selected_char_kind('ISO_10646')
5 character :: ce
6 character(kind= ascii) :: ca, x= ascii_'Y'
7 character(kind= iso10646) :: y
```

[8] Também definida na seção 8.8.5.
[9] *American Standard Code for Information Interchange* (ASCII).
[10] https://pt.wikipedia.org/wiki/ISO/IEC_10646.

Tabela 3.3: Tabela de códigos ASCII de 7 bits.

Dec	Hex	Char		Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NUL	(null)	32	20	Space	64	40	@	96	60	‘
1	1	SOH	(start of heading)	33	21	!	65	41	A	97	61	a
2	2	STX	(start of text)	34	22	“	66	42	B	98	62	b
3	3	ETX	(end of text)	35	23	#	67	43	C	99	63	c
4	4	EOT	(end of transmission)	36	24	$	68	44	D	100	64	d
5	5	ENQ	(enquiry)	37	25	%	69	45	E	101	65	e
6	6	ACK	(acknowledge)	38	26	&	70	46	F	102	66	f
7	7	BEL	(bell)	39	27	’	71	47	G	103	67	g
8	8	BS	(backspace)	40	28	(	72	48	H	104	68	h
9	9	TAB	(horizontal tab)	41	29	)	73	49	I	105	69	i
10	A	LF	(line feed, new line)	42	2A	*	74	4A	J	106	6A	j
11	B	VT	(vertical tab)	43	2B	+	75	4B	K	107	6B	k
12	C	FF	(form feed, new page)	44	2C	,	76	4C	L	108	6C	l
13	D	CR	(carriage return)	45	2D	-	77	4D	M	109	6D	m
14	E	SO	(shift out)	46	2E	.	78	4E	N	110	6E	n
15	F	SI	(shift in)	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	(data link escape)	48	30	0	80	50	P	112	70	p
17	11	DC1	(device control 1)	49	31	1	81	51	Q	113	71	q
18	12	DC2	(device control 2)	50	32	2	82	52	R	114	72	r
19	13	DC3	(device control 3)	51	33	3	83	53	S	115	73	s
20	14	DC4	(device control 4)	52	34	4	84	54	T	116	74	t
21	15	NAK	(negative acknowledge)	53	35	5	85	55	U	117	75	u
22	16	SYN	(synchronous idle)	54	36	6	86	56	V	118	76	v
23	17	ETB	(end of trans. block)	55	37	7	87	57	W	119	77	w
24	18	CAN	(cancel)	56	38	8	88	58	X	120	78	x
25	19	EM	(end of medium)	57	39	9	89	59	Y	121	79	y
26	1A	SUB	(substitute)	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	(escape)	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	(file separator)	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	(group separator)	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	(record separator)	62	3E	>	94	5E	^	126	7E	~
31	1F	US	(unit separator)	63	3F	?	95	5F	_	127	7F	DEL

```
9  print*, selected_char_kind('default')
10 print*, selected_char_kind('ascii')
11 print*, selected_char_kind('iso_10646')
12 ce= ascii_'X'
13 ca= 'X'
14 print'(/,2a)', 'ca= ', ca
15 print'(/,2a)', 'ce= ', ce
16 y= x
17 print'(/,4a)', 'x= ', x, '  y= ', y
18 end program testa_selected_char_kind
```

3.3.4.2 Nomes fixos para as espécies mais comuns

Existe uma alternativa ao emprego das funções intrínsecas `SELECTED_REAL_KIND(P,R)` ou `SELECTED_INT_KIND(R)`, a qual serve para padronizar os nomes das variáveis inteiras que armazenam os parâmetros das espécies mais comuns e para aumentar a portabilidade do código. Esta alternativa consiste no uso do módulo intrínseco `iso_fortran_env`.[11]

O módulo `iso_fortran_env` contém as definições (dependentes do processador e/ou compilador) dos seguintes escalares inteiros padrões:

[11] Módulos intrínsecos são definidos na seção 9.13.6.

int8	inteiro de 8 bits	real32	real de 32 bits (precisão simples)
int16	inteiro de 16 bits	real64	real de 64 bits (precisão dupla)
int32	inteiro de 32 bits	real128	real de 128 bits (precisão estendida ou quádrupla)
int64	inteiro de 64 bits		

Se o compilador suporta mais do que uma espécie com um tamanho em particular, o padrão não especifica qual destes será empregado para a constante. Se o compilador não suporta um tipo de um tamanho em particular, a variável terá o valor igual a -2 se o valor suportado for maior ou -1 se nenhum tamanho maior for suportado.

Para o caso de os valores acima não ser ainda satisfatórios, o módulo iso_fortran_env suporta também matrizes constantes contendo todos os valores de espécies para os tipos intrínsecos que são suportados pelo processador.[12] Estas matrizes constantes são: character_kinds, integer_kinds, logical_kinds e real_kinds.

Sugere-se que estes recursos do módulo iso_fortran_env sejam empregados para as diversas definições de espécies de tipo. O programa 3.4 ilustra o uso destes recursos.

3.3.4.3 INQUIRIÇÃO SOBRE O PARÂMETRO DE TIPO

O recurso de *inquirição sobre o parâmetro do tipo* (*type parameter inquiry*) corresponde à ação inversa daquela obtida com as funções intrínsecas discutidas na seção 3.3.4.1. O valor do parâmetro de espécie de um tipo numérico pode ser obtido em qualquer ponto do programa com o emprego do sufixo %KIND. Por exemplo, se A é objeto do tipo real, a instrução

```
PA= A%KIND ! PA é variável inteira
```

deposita em PA o valor do parâmetro de espécie de A. O uso do sufixo %KIND é equivalente à instrução PA = KIND(A).

Por sua vez, se for necessário conhecer o comprimento da string C, pode-se empregar o sufixo %LEN, como em

```
LC= C%LEN ! LC é variável inteira
```

Neste caso, o uso do sufixo %LEN é equivalente à instrução LC= LEN(C).[13]

O programa 3.5 ilustra o uso da inquirição sobre o parâmetro do tipo.

Listagem 3.5: Exemplo de uso da inquirição sobre o parâmetro do tipo.

```
program tes_type_parameter_inquiry
use, intrinsic :: iso_fortran_env
implicit none
character(len= 50) :: ch= compiler_version() ! <- Não funciona com ifort.
real(kind= real32) :: x
real(kind= real64) :: d
real(kind= real128) :: q
print'(2a)', 'Compiler: ', trim(ch) ! Função trim: seção 8.7.2
print'(a,g0)', 'kind parameter value: real: ', x%kind
print'(a,g0)', 'kind parameter value: double precision: ', d%kind
print'(a,g0)', 'kind parameter value: quad precision: ', q%kind
print'(a,g0)', 'len parameter value: character: ', ch%len
end program tes_type_parameter_inquiry
```

3.4 TIPOS DERIVADOS

Em muitas situações, a natureza do problema a ser resolvido por códigos numéricos demanda o uso de objetos de dados mais sofisticados do que os tipos intrínsecos discutidos nas seções anteriores. Na física, tal situação surge quando se faz necessário armazenar diferentes

[12] Matrizes são discutidas no capítulo 6.
[13] A função intrínseca LEN é discutida na seção 8.7.1.

Listagem 3.4: Nomes fixos para as espécies mais comuns.

```
!Mostra valores e rotinas do módulo intrínseco ISO_FORTRAN_ENV.
program tes_iso_fortran_env_kinds
use, intrinsic :: iso_fortran_env
implicit none
character(len= 1) :: chdef ! Default character
logical :: ldef ! Default logical
logical(kind= 1) :: l1
logical(kind= 2) :: l2
logical(kind= 4) :: l4
logical(kind= 8) :: l8
integer :: idef ! Default integer
integer(int8) :: i8
integer(int16) :: i16
integer(int32) :: i32
integer(int64) :: i64
real :: rdef    ! Default real
complex :: cdef ! Default complex
real(real32) :: r32
real(real64) :: r64
real(real128) :: r128

print'(/,a)', '------------------------> Character kinds: <------------------------'
print*, 'character_kinds:', character_kinds
print*, '# of kinds:', size(character_kinds)
print*, 'Default character:', kind(chdef)
print'(2(a,g0))', 'cdef: # bytes: ', storage_size(chdef)/8

print'(/,a)', '--------------------------> Logical kinds: <--------------------------'
print*, 'logical_kinds:', logical_kinds
print*, '# of kinds:', size(logical_kinds)
print*, 'Default logical:', kind(ldef)
print'(2(a,g0))', 'ldef: # bytes: ', storage_size(ldef)/8
print'(2(a,g0))', 'l1:   # bytes: ', storage_size(l1)/8
print'(2(a,g0))', 'l2:   # bytes: ', storage_size(l2)/8
print'(2(a,g0))', 'l4:   # bytes: ', storage_size(l4)/8
print'(2(a,g0))', 'l8:   # bytes: ', storage_size(l8)/8

print'(/,a)', '--------------------------> Integer kinds: <--------------------------'
print*, 'integer_kinds:', integer_kinds
print*, '# of kinds:', size(integer_kinds)
print*, 'Default integer:', kind(idef)
print*, 'int8, int16, int32, int64:', int8, int16, int32, int64
print'(2(a,g0))', 'int8:  # digits: ', digits(i8), '   # bytes: ', (digits(i8)+1)/8
print'(2(a,g0))', 'int16: # digits: ', digits(i16), '  # bytes: ', (digits(i16)+1)/8
print'(2(a,g0))', 'int32: # digits: ', digits(i32), '  # bytes: ', (digits(i32)+1)/8
print'(2(a,g0),/)', 'int64: # digits: ', digits(i64), '  # bytes: ', (digits(i64)+1)/8

print'(/,a)', '---------------------------> Real kinds: <---------------------------'
print*, 'real_kinds:', real_kinds
print*, '# of kinds:', size(real_kinds)
print*, 'Default real:   ', kind(rdef)
print*, 'Default complex:', kind(cdef)
print*, 'real32, real64, real128:', real32, real64, real128
print'(a)', '--> For real32 kind:'
print'(4(a,g0))', '# digits: ', digits(r32), '  precision= ', precision(r32), &
                  '  range= ', range(r32),    '  bit size= ', storage_size(r32)
print'(a)', '--> For real64 kind:'
print'(4(a,g0))', '# digits: ', digits(r64), '  precision= ', precision(r64), &
                  '  range= ', range(r64),    '  bit size= ', storage_size(r64)
print'(a)', '--> For real128 kind:'
print'(4(a,g0))', '# digits: ', digits(r128), '  precision= ', precision(r128), &
                  '  range= ', range(r128),    '  bit size= ', storage_size(r128)
end program tes_iso_fortran_env_kinds
```

informações a respeito de um objeto em particular, como uma partícula elementar, por exemplo. Partículas elementares como elétrons ou quarks possuem diversas propriedades físicas próprias: massa de repouso, carga elétrica, estado de spin, além de outras informações tais como a classificação (hádron ou lépton). Ou seja, um objeto físico em particular é caracterizado por mais de uma propriedade, sendo que essas propriedades podem ser armazenadas por meio de diferentes tipos intrínsecos: inteiro, real, caractere, *etc.* Em tal situação, não é adequado armazenar essas diferentes informações a respeito de um mesmo objeto em diversas variáveis distintas; é muito mais prático centralizar essas informações em um único objeto de dados.

O Fortran, assim como diversas outras linguagens modernas, resolve esse problema por meio de objetos de dados compostos: os *tipos derivados de dados* (*derived data types*), ou simplesmente *tipos derivados*, os quais são também conhecidos como *estruturas* (*structures*) em outras linguagens.

3.4.1 USO BÁSICO

Um objeto de tipo derivado possui um ou mais **componentes**, os quais podem ser dos tipos intrínsecos ou de outros tipos derivados previamente definidos. Uma forma resumida da declaração de um tipo derivado é:

```
TYPE [::] <nome do tipo>
   <declarações de componentes>
END TYPE [<nome do tipo>]
```

onde cada <*declaração de componentes*> tem a forma

```
<tipo>[[, <atributos>] ::] <lista de componentes>
```

onde aqui o <*tipo*> pode ser um tipo intrínseco ou outro tipo derivado, previamente definido.

De maneira simplificada, a declaração acima cria um novo tipo de objeto de dados, o qual é um objeto composto. Uma vez definido o novo tipo derivado, pode-se declarar uma lista de nomes como sendo objetos de dados deste tipo derivado. A declaração de uma lista de variáveis do tipo derivado <*nome do tipo*> é feita através da linha:

```
TYPE (<nome do tipo>) [::] <lista de nomes>
```

Para tentar entender a definição e o uso de uma variável de tipo derivado, vamos definir um novo tipo: `PONTO`, o qual será construído a partir de três valores reais, o quais armazenam os valores das coordenadas *x*, *y* e *z* de um ponto no espaço Cartesiano:

```
program def_tipo_der
implicit none
type :: ponto
   real :: x, y, z
end type ponto
!
type (ponto) :: centro, apice
!
apice%x= 0.0
apice%y= 1.0
apice%z= 0.0
centro = ponto(0.0,0.0,0.0)
!
print*, apice
print*, centro
!
end program def_tipo_der
```

No exemplo acima, definiu-se o tipo derivado `PONTO`, composto por três componentes reais *x*, *y* e *z*. Em seguida, declarou-se duas variáveis como sendo do tipo `PONTO`: `CENTRO` e `APICE`. A seguir, atribuiu-se os valores para estas variáveis. Finalmente, mandou-se imprimir na tela o valor das variáveis.

O exemplo já mostra duas das três maneiras possíveis de atribuição de valores aos componentes de um tipo derivado; quais sejam, via:

Seletor de componente (%): por exemplo, ao componente x da variável `apice` é atríbuído o valor `0.0` através da instrução

```
apice%x= 0.0
```

o mesmo se aplica aos outros componentes.

Construtor de estrutura: a variável `centro` teve os valores de seus três componentes simultaneamente atribuídos pela instrução

```
centro= ponto(0.0,0.0,0.0)
```

Conforme empregado neste exemplo, um construtor de estrutura consiste no nome do tipo derivado seguido pela lista de valores de seus componentes entre parênteses, na ordem em que os componentes foram declarados na definição do tipo. Este tipo de construção pode rapidamente se tornar confuso; por esta razão é recomendado o emprego de *palavras-chave* (*keywords*) que irão identificar de forma inequívoca os componentes. A forma geral de um construtor de estrutura é apresentada na seção 4.8.

Inicialização padrão de componentes: é possível também realizar-se uma inicialização de qualquer componente de um tipo derivado na sua definição. Por exemplo, o tipo `ponto` no exemplo acima poderia ter seus componentes inicializados a zero assim:

```
type :: ponto
   real :: x= 0.0, y= 0.0, z= 0.0
end type ponto
```

Qualquer variável do tipo `ponto` terá seus componentes inicializados aos valores acima. Posteriormente, qualquer destes componentes pode ter seu valor alterado, como ocorre na instrução `apice%y= 1.0`.

É possível construir estruturas progressivamente mais complicadas definindo-se novos tipos derivados que englobam aqueles previamente definidos. Por exemplo,

```
TYPE :: RETA
   TYPE (PONTO) :: P1,P2
END TYPE RETA
TYPE (RETA) :: DIAGONAL_PRINCIPAL
!
DIAGONAL_PRINCIPAL%P1%X= 0.0
DIAGONAL_PRINCIPAL%P1%Y= 0.0
DIAGONAL_PRINCIPAL%P1%Z= 0.0
!
DIAGONAL_PRINCIPAL%P2= PONTO(1.0,1.0,1.0)
```

Aqui foi definido o tipo `RETA` no espaço Cartesiano, a qual é totalmente caracterizada por dois pontos, `P1` e `P2`, os quais, por sua vez, são ternas de números do tipo (x, y, z). Definiu-se então a variável `DIAGONAL_PRINCIPAL` como sendo do tipo `RETA` e definiu-se os valores dos dois pontos no espaço `P1` e `P2` que caracterizam a diagonal principal. Note o uso de dois seletores de componente para definir o valor da coordenada x do ponto `P1` da `DIAGONAL_PRINCIPAL`. Note, por outro lado, o uso do construtor de estrutura para definir a posição do ponto `P2`, como componente da diagonal principal.

Além de permitir a atribuição de valores aos componentes de um tipo derivado, o seletor de componente (%) também permite que o valor armazenado neste componente seja empregado em expressões, conforme as regras próprias de manipulação do tipo de dado, regras essas que serão discutidas no capítulo 4. Por exemplo, a instrução

```
apice%y + 2
```

resultará no literal inteiro 3.

O exemplo a seguir define o tipo `ALUNO_T`, caracterizado por `NOME`, `CODIGO` de matrícula, notas parciais `N1`, `N2` e `N3` e média final `MF`. O programa lê as notas parciais e calcula e imprime a média final do aluno.

Listagem 3.6: Tipo derivado para armazenar os dados acadêmicos de um aluno.

```
program aluno
implicit none
type:: aluno_t
   character(len= 20):: nome
   integer:: codigo
   real:: n1,n2,n3,mf
end type aluno_t
type(aluno_t):: discente
!
   print*,'Nome:'
   read'(a)', discente%nome
   print*,'codigo:'
   read*,discente%codigo
   print*,'Notas: N1,N2,N3:'
   read*,discente%n1,discente%n2,discente%n3
   discente%mf= (discente%n1 + discente%n2 + discente%n3)/3.0
   print*,' '
   print*,'------------> ',discente%nome,' (',discente%codigo,') <------------'
   print*,'          Media final: ',discente%mf
end program aluno
```

Sugestões de uso & estilo para programação

Em muitas situações, é recomendável tornar claro que um determinado objeto de dado é uma estrutura, ao invés de ser um tipo intrínseco, por exemplo. Uma maneira de fazer isso é incluindo o sufixo "_t" ao nome do tipo derivado e (possivelmente) também aos nomes de variáveis deste tipo. Esta recomendação foi adotada na definição do tipo `aluno_t` no programa acima.

Esta prática é útil também para distinguir um construtor de estrutura, tal como `discente= aluno_t(...)`, de uma chamada de função (discutida no capítulo 9).

3.4.2 FORMA GERAL DA DEFINIÇÃO DE TIPO DERIVADO

Os exemplos acima ilustram somente uma parte dos recursos disponíveis com o emprego de tipos derivados. Na sua definição mais geral, os tipos derivados são empregados também em *programação orientada a objeto* e para realizar *interoperações* (*i. e.*, troca de informações) com funções escritas em outras linguagens. Até o presente momento, o padrão oferece recursos de interoperação com a linguagem C.

A forma geral de uma definição de tipo derivado é

```
type [[, <atrib-tipo>]... ::] <nome-tipo> [(<lista-nome-param-tipo>)]
   [<decl-param-tipo>]...
   [private]
   [<decl-componentes>]...
   [<sec-rotina-vinc-tipo>]
end type [<nome-tipo>]
```

O campo `<atrib-tipo>` contém a lista de atributos do tipo derivado. Os possíveis atributos são

`abstract`	(empregado em programação orientada a objeto)
`<tipo-accesso>`	`public` ou `private` (descritos na seção 3.4.2.1)
`bind(c)`	(empregado para realizar interoperações com C)
`extends (<nome-tipo-pai>)`	(empregado em programação orientada a objeto)

O `<nome-tipo>` é qualquer nome de objeto de dados aceitável em Fortran.[14] Contudo, `<nome-tipo>` não pode ser igual ao nome de um tipo intrínseco ou de algum outro tipo derivado

[14] Ver seção 2.3.

definido previamente na mesma unidade de programa ou em outra unidade acessível por meio de algum dos mecanismos discutidos no capítulo 9.

Os campos <lista-nome-param-tipo> e <decl-param-tipo> são empregados quando se está definindo um **tipo derivado parameterizado** (*parameterized derived type*). Este recurso não será discutida nesta Apostila.

O setor <decl-param-tipo> contém declarações de tipos especiais de rotinas, tais como *ponteiros de rotinas* (*procedure pointers*, seção 9.12), ou contém as declarações dos componentes do tipo derivado. A forma genérica destas declarações será apresentada na seção 3.4.2.2 abaixo.

O efeito da declaração `private` é discutido na seção 3.4.2.1.

O setor <sec-rotina-vinc-tipo> contém as **rotinas vinculadas ao tipo** (*type-bound procedures*), as quais também são recursos empregados na programação orientada a objetos. Esta forma de programação não será discutida nesta Apostila.

3.4.2.1 Configurações de acesso em tipos derivados

As opções do campo <tipo-acesso> são: `public` ou `private`. Estas opções determinam se toda a estrutrura definida pelo tipo derivado é acessível ao programador ou não, ou se somente uma parte da estrutura é acessível. O uso destas configurações somente faz sentido quando o tipo derivado é definido em um *módulo*. Módulos são discutidos na seção 9.13) e o conteúdo da mesma é suposto conhecido no restante desta seção.

Sugestões de uso & estilo para programação

Quando os tipos derivados forem empregados para nomear variáveis em diversas unidades de programa distintas, é recomendável que suas definições sejam realizadas em módulos, os quais serão por sua vez usados por essas unidades de programa. Desta forma, as definições dos tipos somente serão realizadas um vez, reduzindo a possibilidade de ocorrência de erros de programação.

O atributo ou declaração `private` somente poderá ser empregado em qualquer ponto na definição do tipo derivado se esta for realizada em um módulo. Neste caso, as seguintes possibilidades ocorrem:

- Se o tipo <nome-tipo> for definido com o atributo `private`, então o <nome-tipo>, o construtor de estrutura do tipo, qualquer entidade do tipo (componente ou rotina vinculada), qualquer rotina onde o tipo é argumento mudo ou função com resultado do tipo são inacessíveis a quaisquer unidades de programa que usem o módulo. A acessibilidade ao tipo também pode ser determinada com uma declaração `private` ou `public` em outro ponto do módulo. Caso estas configurações não ocorram, o tipo é acessível para uso fora do módulo.

- Se a declaração `private` ocorrer na definição do tipo, então qualquer componente declarado no setor <decl-componentes> que não tenha na sua lista <atrib-componentes> (seção 3.4.2.2) o atributo `public` será privado. Isto significa que mesmo que o tipo seja público, um componente privado não poderá ser acessado em outras unidades de programa que usem o módulo nem o seu construtor de estrutura poderá atribuir valor ao mesmo.

- Por fim, diferentes componentes do mesmo tipo poderão ser públicos ou privados, dependendo da presença dos respectivos atributos na <lista-componentes>.

Estas regras tornam a acessibilidade do tipo e dos seus componentes totalmente independentes, de tal forma que todas as combinações são possíveis: um componente público de um tipo público, um componente público de um tipo privado, um componente privado de um tipo público e um componente privado de um tipo privado.

Um exemplo dessa acessibilidade mista é:

```
module meutipo_mod
   type :: meutipo_t
      private
      character(len= 20), public :: debug_tag= “”
      ⋮ ! outros componentes privados omitidos
   end type meutipo_t
```

```
    ⋮
end module meutipo_mod
```

embora o tipo `meutipo_t` tenha alguns de seus componentes privados (protegendo assim a integridade dos mesmos), o string `debug_tag` é público e pode ser acessado em unidades de programa que usem o módulo `meutipo_mod`.

3.4.2.2 DECLARAÇÕES DOS COMPONENTES DO TIPO

A forma genérica das declarações dos componentes é

```
<tipo>[[, <atrib-componentes>]... ::] <lista-componentes>
```

onde <*tipo*> pode ser um tipo intrínseco ou um tipo derivado, o qual foi previamente definido ou é o próprio tipo em definição. Este último caso está sujeito às restrições discutidas ao final desta seção.

O campo <atrib-componentes> contém os atributos dos componentes, declarados na <lista-componentes>. Os possíveis atributos são

<acesso>	`public` ou `private` (seção 3.4.2.1)
`allocatable`	(componentes são *alocáveis*; ver seção 7.1.3)
`codimension[<lista-cobounds>]`	(empregado com **coarrays**)
`contiguous`	(para computação de alta performance; ver seção 7.5)
`dimension(<lista-ext>)`	(opção para matrizes; ver capítulo 6)
`pointer`	(componentes são *ponteiros*; ver seção 7.2)

Cada nome na <lista-componentes> tem a forma genérica

```
<nome-comp>[(<lista-ext>)][*<char-len>][[<lista-cobounds>]][= <init-comp>]
```

onde <char-len> foi discutido na seção 3.1.4 e <init-comp> contém a inicialização do valor do componente.

Se <tipo> for um tipo derivado, então considerações de acesso ao mesmo também devem ser feitas com relação aos atributos `allocatable` ou `pointer`, contidos na lista de <atrib-componentes>. Se nenhum destes atributos for empregado, então o tipo derivado deve ter sido definido previamente na mesma unidade de programa ou ser acessível por meio de algum dos mecanismos discutidos no capítulo 9. Se, por outro lado, `allocatable` ou `pointer` forem incluídos, então <tipo> pode ser o tipo derivado sendo definido.

3.5 MATRIZES (*Arrays*)

Um outro tipo de objeto de dados composto suportado pelo Fortran é uma *matriz* (*array*). Uma matriz é uma coleção de objetos, todos do mesmo tipo e espécie (em princípio), os quais estão relacionados entre si e que são acessados empregando-se um nome comum e índices inteiros.

O Fortran oferece diversos recursos para manipulação de matrizes e para o processamento rápido de operações envolvendo as mesmas. A discussão a respeito de matrizes será realizada no capítulo 6.

3.6 COARRAYS

Já há alguns anos que computadores em geral, em todas as escalas de produção, são comercializados com diversos **núcleos** (*cores*) de processamento. Mesmo aparelhos telefônicos atualmente são vendidos com 4 ou 8 núcleos. A existência desses diversos núcleos permitem que mais de uma aplicação seja executada simultaneamente.

Originalmente, a motivação para o desenvolvimento de processadores com diversos núcleos estava na possibilidade de se executar diferentes porções do mesmo código simultaneamente, aumentando assim o desempenho global do mesmo. Para tanto, foram desenvolvidas diversas estratégias para a implementação da chamada *computação paralela*. Os projetos MPI (*Message Passing Interface*),[15] OpenVMS (*Open Virtual Memory System*)[16] e HPF (*High Performance For-*

[15] https://pt.wikipedia.org/wiki/Message_Passing_Interface
[16] https://pt.wikipedia.org/wiki/OpenVMS

tran)[17] são apenas alguns exemplos de diferentes implementações dessas estratégias.

Usualmente, a comunicação entre a linguagem de alto nível (Fortran, C, C++, *etc*) e o sistema de computação paralela é realizada via *diretivas de compilação*, isto é, instruções que para o compilador da linguagem são apenas comentários, mas que atuam um programa específico, o qual é invocado pelo sistema de compilação. Em outras palavras, as diretivas de processamento paralelo usualmente não pertencem ao padrão da linguagem, mas são extensões ao mesmo.

No Fortran Moderno, uma estratégia de processamento paralelo, os **coarrays**, fazem parte do padrão da linguagem, o que elimina a necessidade de se inserir diretivas específicas de compilação e possibilita a integração dessas estratégias ao código de forma harmônica.

Coarray não serão discutidos nesta Apostila. Maiores informações podem ser obtidas nos textos citados na bibliografia ou no projeto OpenCoarrays.

3.7 Ponteiros e estruturas alocáveis

Ao longo das seções 3.2 – 3.4 discutiu-se os tipos intrínsecos e os tipos derivados de dados. Todos estes objetos têm duas características em comum: todos armazenam algum tipo de objeto de dados (mesmo que em uma estrutura), ou seja, todos armazenam literais de algum tipo, e todos são **estáticos**, no sentido de que os tipos e o número de variáveis no programa foram declarados antes da execução do programa e permanecem constantes durante a sua execução.

Contudo, em muitas aplicações, principalmente quando o programador está desenvolvendo projetos de larga escala, uma estruturação estática de objetos de dados se torna contraprodutiva. Para resolver este problema diversas linguagens de programação contêm *objetos e estruturas dinâmicas de dados*, com os quais se torna possível reservar, durante a execução do programa, espaços específicos na memória que serão utilizados quando for necessário.

No Fortran, existem dois tipos de objetos dinâmicos que cumprem esta função: *ponteiros* e *objetos alocáveis*, os quais serão abordados em maiores detalhes no capítulo 7.

[17] https://en.wikipedia.org/wiki/High_Performance_Fortran

4 Expressões e Atribuições Escalares

Em uma *expressão*, o programa descreve as operações que devem ser executadas pelo computador. O resultado desta expressão pode então ser *atribuído* a uma variável. Há diferentes conjuntos de regras para expressões e atribuições, dependendo se as variáveis em questão são numéricas, lógicas, de caracteres ou de tipo derivado; e também se as expressões são escalares ou matriciais.

Cada um dos conjuntos de regras para expressões escalares será agora discutido.

4.1 Regras básicas para expressões escalares

Uma expressão em Fortran é formada de *operandos* e *operadores*, combinados de tal forma que estes seguem as regras de sintaxe. Os operandos podem ser constantes, variáveis ou funções e uma expressão, por si mesma, também pode ser usada como operando. A sentença

$$\underbrace{\texttt{A} =}_{\text{atribuição}} \quad \underbrace{\texttt{X} + \texttt{Y}}_{\text{expressão}}$$

é composta por uma expressão no lado direito da mesma e de uma atribuição no lado esquerdo.

Uma expressão simples envolvendo um operador *unário* ou *monádico* tem a seguinte forma:

```
operador operando
```

um exemplo sendo

```
-Y
```

Já uma expressão simples envolvendo um operador binário (ou *diádico*) tem a forma:

```
operando operador operando
```

um exemplo sendo

```
X + Y
```

Uma expressão mais complicada seria

```
operando operador operando operador operando
```

onde operandos consecutivos são separados por um único operador.

Cada operando deve ter valor definido e o resultado da expressão deve ser matematicamente definido; por exemplo, divisão por zero não é permitido, gerando o que se denomina uma *exceção de ponto flutuante* (*floating point exception*).

A sintaxe do Fortran estabelece que as partes de expressões sem parênteses sejam desenvolvidas da esquerda para a direita para operadores de igual precedência, com a exceção do operador "**" (ver seção 4.2). Caso seja necessário desenvolver parte de uma expressão (ou *subexpressão*) antes de outra, parênteses podem ser usados para indicar qual subexpressão deve ser desenvolvida primeiramente. Na expressão

```
operando operador (operando operador operando)
```

a subexpressão entre parênteses será desenvolvida primeiro (no sentido usual) e o resultado usado como um operando para o primeiro operador, quando então a expressão completa será desenvolvida, novamente, no sentido usual, ou seja, da esquerda para a direita.

Se uma expressão ou subexpressão não contém parênteses e todos os operadores têm a mesma ordem de precedência, é permitido ao processador desenvolver uma expressão equivalente, a qual é uma expressão que resultará (em princípio) no mesmo valor, exceto por erros de arredondamento numérico. Por exemplo, se `A`, `B` e `C` são variáveis reais e o programador escrever a expressão

`A/B/C`, o compilador pode adotar a forma `A/(B*C)`,

caso o processador realize multiplicações mais rapidamente que divisões. Contudo, embora ambas as expressões sejam *matematicamente* equivalentes, elas não são *numericamente* equivalentes. Em particular, a segunda forma pode resultar em um erro maior de arredondamento. Neste tipo de situação cabe ao programador decidir qual estratégia deve ser adotada.

Se dois operadores seguem-se imediatamente, como em

```
operando operador operador operando
```

a única interpretação possível é que o segundo operador é monádico. Portanto, a sintaxe proíbe que um operador binário siga outro operador.

4.2 Expressões numéricas escalares

Uma *expressão numérica* é aquela cujos operandos são compostos pelos três tipos numéricos intrínsecos: `INTEGER`, `REAL` ou `COMPLEX`. A tabela 4.1 apresenta os operadores em ordem de precedência e o seu respectivo significado. Estes operadores são conhecidos como operadores *numéricos intrínsecos.*

Tabela 4.1: Operadores numéricos escalares

Operador	Operação
	Cálculo de função
**	Potenciação
*	Multiplicação
/	Divisão
+	Adição
-	Subtração

Na tabela 4.1, as linhas horizontais confinam os operadores de igual precedência e a ordem de precedência é dada de cima para baixo. O operador de potenciação "**" é o de maior precedência; os operadores de multiplicação "*" e divisão "/" têm a mesma precedência entre si e possuem precedência sobre os operadores de adição "+" e subtração "-", os quais têm a mesma precedência entre si.

Na ausência de parênteses, ou dentro destes no caso de subexpressões, a operação com a maior precedência é o cálculo de uma função, seguido de uma exponenciação que será realizada antes de multiplicações ou divisões e estas, por sua vez, antes de adições ou subtrações.

Uma expressão numérica especifica uma computação usando constantes, variáveis ou funções, seguindo o seguinte conjunto de regras:

- Os operadores de subtração "-" e de adição "+" podem ser usados como operadores unários, como em
  `-VELOCIDADE`
- Uma vez que não é permitido na notação matemática ordinária, uma subtração ou adição unária não pode seguir imediatamente após outro operador. Quando isto é necessário, deve-se fazer uso de parênteses. Por exemplo, nas operações matemáticas a seguir:

  x^{-y} deve ser digitada `X**(-Y)`;

  $x(-y)$ deve ser digitada `X*(-Y)`.

  Como já foi mencionado na seção 4.1, um operador binário também não pode seguir outro operador.

- Os parênteses devem indicar os agrupamentos, como se escreve em uma expressão matemática. Os parênteses indicam que as operações dentro deles devem ser executadas prioritariamente:

  $(a+b)\left[(x+y)^2+w^2\right]^3$ deve ser digitada `(A+B)*((X+Y)**2 + W**2)**3` ou `(A+B)*(((X + Y)**2 + W**2)**3)`

- Os parênteses não podem ser usados para indicar multiplicação, sendo sempre necessário o uso do operador "*". Qualquer expressão pode ser envolvida por parênteses exteriores, que não a afetam:

  `X + Y` é equivalente a `((X) + Y)` ou equivalente a `((X + Y))`

  Contudo, o número de parênteses à esquerda deve sempre ser igual ao número de parênteses à direita.

- Nenhum operador pode ser deixado implícito:

  `5*T` ou `T*5`: correto

  `5T, 5(T)` ou `T5`: incorreto.

- De acordo com a tabela 4.1, a qual fornece o ordenamento dos operadores, a operação matemática $2x^2+y$ pode ser expressa de, no mínimo, duas maneiras equivalentes: `2*X**2 + Y` ou `2*(X**2) + Y` ou ainda `(2*(x**2)) + Y.`

- A exceção à regra esquerda-para-direita para operadores de igual precedência ocorre apenas para a potenciação "**". A expressão

  `A**B**C`

  é válida e é desenvolvida da direita para a esquerda como

  `A**(B**C).`

Divisão inteira. Para dados inteiros, o resultado de qualquer divisão será truncado para zero, isto é, para o valor inteiro cuja magnitude é igual ou logo inferior à magnitude do valor exato. Assim, o resultado de:

6/3 é 2

8/3 é 2

-8/3 é -2.

Este fato deve sempre ser levado em conta quando operações com inteiros estão sendo realizadas. Por isto, o valor de `2**3` é `8`, enquanto que o valor de `2**(-3)` é `0`.

Expressão de modo misto. A regra padrão do Fortran também permite que uma expressão numérica contenha operandos numéricos de diferentes tipos ou espécies. Esta é uma *expressão de modo misto*. Exceto quando elevando um valor real ou complexo a uma potência inteira, o objeto do tipo mais fraco (ou mais simples) de variável dos dois tipos envolvidos em uma expressão será convertido, ou *coagido*, para o tipo mais forte. O resultado será também do tipo mais forte. Por exemplo, se `A` é real e `I` é inteiro, a expressão `A*I` tem, inicialmente, `I` sendo convertido a real antes que a multiplicação seja efetuada e o resultado da mesma é do tipo real. As tabelas 4.2 e 4.3 ilustram os resultados de diferentes operações numéricas escalares.

Com relação à última operação mencionada na tabela 4.3, no caso de um valor complexo ser elevado a uma potência também complexa, o resultado corresponderá ao valor principal, isto é, $a^b = \exp\left[b\left(\log|a| + i\arg a\right)\right]$, com $-\pi < \arg a < \pi$.

4.3 ATRIBUIÇÕES NUMÉRICAS ESCALARES

A forma geral de uma atribuição numérica escalar é

```
<nome variável> = <expressão>
```

Tabela 4.2: Tipos de resultados de *A .op. B*, onde *.op.* é *+, -, * ou /*. As funções intrínsecas *REAL()* e *CMPLX()* são discutidas na seção 8.3.1

Tipo de A	Tipo de B	Valor de A usado na .op.	Valor de B usado na .op.	Tipo do resultado
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

Tabela 4.3: Tipos de resultados de *A**B*.

Tipo de A	Tipo de B	Valor de A usado na .op.	Valor de B usado na .op.	Tipo do resultado
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

onde <nome variável> é o nome de uma variável numérica escalar e <expressão> é uma expressão numérica. Se <expressão> não é do mesmo tipo ou espécie da <variável>, a primeira será convertida ao tipo e espécie da última antes que a atribuição seja realizada, de acordo com as regras dadas na tabela 4.4.

Deve-se notar que se o tipo da variável for inteiro mas a expressão não, então a atribuição irá resultar em perda de acurácia, exceto se o resultado for exatamente inteiro. Da mesma forma, atribuindo uma expressão real a uma variável real de uma espécie com precisão menor também causará perda de acurácia. Similarmente, a atribuição de uma expressão complexa a uma variável não complexa resultará, no mínimo, na perda da parte imaginária. Por exemplo, os valores de I e A após as atribuições

```
I= 7.3                  ! I do tipo inteiro.
A= (4.01935, 2.12372)   ! A do tipo real.
```

são, respectivamente, 7 e 4.01935.

4.4 Operadores relacionais

A tabela 4.5 apresenta os *operadores relacionais escalares*, utilizados em operações que envolvem a comparação entre duas variáveis ou expressões. Na coluna 1 da tabela, é apresentado o significado do operador, na coluna 2 a sua forma escrita com caracteres alfanuméricos e, na coluna 3, a sua forma escrita com caracteres especiais, a qual é a mais moderna e recomendável, sendo a outra considerada obsoleta.

Se uma ou ambas as expressões forem complexas, somente os operadores == e /= (ou .EQ. e .NE.) são aplicáveis.

Tabela 4.4: Conversão numérica para o comando de atribuição *<nome variável> = <expressão>*.

Tipo de <variável>	Valor atribuído
I	INT(<expressão>,KIND(<nome variável>))
R	REAL(<expressão, KIND(<nome variável>))
C	CMPLX(<expressão>,KIND(<nome variável>))

Tabela 4.5: Operadores relacionais em Fortran.

Significado	Forma obsoleta	Forma moderna
Maior que	.GT.	>
Menor que	.LT.	<
Igual a	.EQ.	==
Maior ou igual	.GE.	>=
Menor ou igual	.LE.	<=
Diferente de	.NE.	/=

O resultado de uma comparação deste tipo tem como resultado um dos dois literais lógicos possíveis em uma álgebra booleana: `.TRUE.` ou `.FALSE.` e este tipo de teste é de crucial importância no controle do fluxo do programa. Exemplos de expressões relacionais são dados abaixo, sendo `I` e `J` do tipo inteiro, `A` e `B` do tipo real e `CHAR1` do tipo caractere padrão:

```
I .LT. 0          ! expressão relacional inteira
A < B             ! expressão relacional real
A + B > I - J     ! expressão relacional de modo misto
CHAR1 == 'Z'      ! expressão relacional de caractere
```

Os operadores numéricos têm precedência sobre os operadores relacionais. Assim, as expressões numéricas, caso existam, são desenvolvidas antes da comparação com os operadores relacionais. No terceiro exemplo acima, como as expressões envolvem dois tipos distintos, cada expressão numérica é desenvolvida separadamente e então ambas são convertidas ao tipo e espécie da soma dos resultados de cada expressão, de acordo com a tabela 4.2, antes que a comparação seja feita. Portanto, no exemplo, o resultado de `(I - J)` será convertido a real.

Para comparações de caracteres, as espécies devem ser as mesmas e as as letras (ou números ou caracteres especiais) são comparados da esquerda para a direita até que uma diferença seja encontrada ou ambos os caracteres sejam idênticos. Se os comprimentos diferirem, a variável mais curta é suposta preenchida por brancos à direita.

4.5 EXPRESSÕES E ATRIBUIÇÕES LÓGICAS ESCALARES

Constantes, variáveis e funções lógicas podem aparecer como operandos em expressões lógicas. Os operadores lógicos, em ordem decrescente de precedência, são apresentados na tabela 4.6.

Tabela 4.6: Operadores lógicos escalares.

Operador	Operação
Operador unário	
.NOT.	Negação lógica
Operadores binários	
.AND.	E lógico, intersecção lógica
.OR.	OU lógico, união lógica
.EQV.	Equivalência lógica
.NEQV.	Não-equivalência lógica

Dada então uma declaração de variáveis lógicas do tipo

```
LOGICAL :: I,J,K,L, FLAG
```

as seguintes expressões lógicas são válidas:

```
.NOT. J
J .AND. K
I .OR. L .AND. .NOT. J
(.NOT. K .AND. J .NEQV. .NOT. L) .OR. I
```

Na primeira expressão o `.NOT.` é usado como operador unário. Na terceira expressão, as regras de precedência implicam em que a subexpressão `L .AND. .NOT. J` é desenvolvida primeiro, e o resultado combinado com `I`. Na última expressão, as duas subexpressões `.NOT. K .AND. J` e `.NOT. L` serão desenvolvidas e comparadas para testar não-equivalência; o resultado da comparação será combinado com `I`.

O resultado de qualquer expressão lógica é, novamente, um dos literais lógicos `.TRUE.` ou `.FALSE.`, e este valor pode então ser atribuído a uma variável lógica, tal como no exemplo abaixo:

```
FLAG= (.NOT. K .EQV. L) .OR. J
```

O resultado de uma expressão lógica envolvendo duas variáveis lógicas, `A` e `B` por exemplo, pode ser inferido facilmente através da consulta às Tabelas-Verdade 4.7 e 4.8.

Tabela 4.7: Tabela-Verdade *.NOT.*

A	.NOT. A
T	F
F	T

Tabela 4.8: Tabelas-Verdade para as operações lógicas binárias.

A	B	A .AND. B	A .OR. B	A .EQV. B	A .NEQV. B
T	T	T	T	T	F
T	F	F	T	F	T
F	T	F	T	F	T
F	F	F	F	T	F

Uma variável lógica pode ter um valor pré-determinado por uma atribuição no corpo de comandos do programa:

```
FLAG= .TRUE.
```

ou no corpo de declarações de variáveis:

```
LOGICAL :: FLAG= .FALSE., BANNER= .TRUE., POLE
```

Nos exemplos acima, todos os operando e resultados foram do tipo lógico. Nenhum outro tipo de variável pode participar de uma operação lógica intrínseca, ou de uma atribuição.

Os resultados de diversas expressões relacionais podem ser combinados em uma expressão lógica, seguida de atribuição, como no caso:

```
REAL :: A, B, X, Y
LOGICAL :: COND
⋮
COND= A > B .OR. X < 0.0 .AND. Y > 1.0
```

onde os operadores relacionais têm precedência sobre os operadores lógicos. O uso mais frequente de expressões que envolvem operadores numéricos, relacionais e lógicos ocorre em testes destinados a determinar o fluxo do programa, como no caso do comando `IF`, exemplificado abaixo e que será discutido em mais detalhes no capítulo 5:

```
REAL :: A= 0.0, B= 1.0, X= 2.5, Y= 5.0
⋮
IF((A < B) .AND. (X - Y > 0.0))THEN
⋮
IF((B**2 < 10.0) .OR. (A > 0.0))THEN
⋮
```

No primeiro teste `IF` acima, o resultado, levando em conta a hierarquia das precedências nas diferentes operações, é `.FALSE.` e os comandos contidos após a cláusula `THEN` não serão executados. Já no segundo exemplo, o resultado é `.TRUE.` e os comandos após o `THEN` serão executados.

4.6 Expressões e atribuições de caracteres

Dados formados por caracteres podem ser manipulados usando **expressões de caracteres**, as quais podem ser seguidas por **atribuições de caracteres**, conforme será exemplificado nas seções abaixo.

4.6.1 ATRIBUIÇÃO DE CARACTERES

Uma expressão de caracteres pode ser atribuída a uma variável de caracteres (uma string). Essa expressão pode ser uma constante literal, como já foi exemplificado na seção 3.1.4 ou pode ser uma expressão mais complexa, conforme discutido na seção 4.6.2.

Se a string variável for declarada com um comprimento maior que a string resultante da expressão, então o comprimento restante da variável será complementado à direita por espaços em branco (indicados aqui por "␣"). Se a string resultante da expressão for mais longa que a variável à qual esta será atribuída, então a variável recebe somente os primeiros caracteres à esquerda da expressão suficientes para preencher o seu comprimento. Por exemplo, nas declarações e atribuições

```
CHARACTER(LEN= 3) :: STR1, STR2
STR1= 'F' ; STR2= 'FILE01'
```

resultam as strings `STR1= 'F␣␣'` e `STR2= 'FIL'`, sendo o literal `'E01'` descartado na atribuição de `STR2`.

4.6.2 EXPRESSÕES: OPERADOR DE CONCATENAÇÃO

O único operador intrínseco para expressões de caracteres é o **operador de concatenação** "//", o qual é um operador binário que tem o efeito de combinar dois operandos de caracteres em uma única string resultante, de comprimento igual à soma dos comprimentos dos operandos originais. Por exemplo, o resultado da concatenação dos literais `'AB'` e `'CD'`, escrita como

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

é o literal `'ABCD'`.

Uma outra operação possível com variáveis de caracteres é a extração de "pedaços" (*substrings*) das variáveis, os quais consistem em um determinado grupo de caracteres contidos na variável original.

4.6.3 *Substrings* DE CARACTERES

Consideremos a seguinte declaração da variável de caractere `HINO`, a qual tem o comprimento igual a 236 caracteres e cujo valor é atribuído no momento da declaração:

```
CHARACTER(LEN=236):: HINO = 'Como a aurora precursora     &
                          & do farol da divindade         &
                          & foi o 20 de setembro          &
                          & o precursor da liberdade.     &
                          & Mostremos valor, constância   &
                          & nesta ímpia, injusta guerra.  &
                          & Sirvam nossas façanhas        &
                          & de modelo a toda Terra (...) '
```

Pode-se isolar qualquer parte da variável `HINO` usando-se a notação de *substring*

```
HINO(I:J)
```

onde `I` e `J` são variáveis inteiras, os *índices da substring*. Neste exemplo, $1 \leqslant \texttt{I} \leqslant \texttt{J} \leqslant 236$, as quais localizam explicitamente os caracteres da posição `I` à posição `J` em `HINO`, formando assim uma substring. Os dois pontos ":" são usados para separar os dois índices da substring e são sempre obrigatórios, mesmo que se queira isolar somente um caractere da variável. Alguns exemplos de substrings da variável `HINO` são:

```
HINO(8:13)          !Correspondente a 'aurora'
HINO(61:80)         !Correspondente a 'foi o 20 de setembro'
HINO(143:143)       !Correspondente a 'â'
```

As constantes de caracteres resultantes das substrings podem ser então atribuídas a outras variáveis de caracteres.

Há valores padrão para os índices das substrings. Se o índice inferior é omitido, o valor 1 é assumido; se o valor superior é omitido, um valor correspondente ao comprimento da variável é assumido. Assim,

`HINO(:50)` é equivalente a `HINO(1:50)`

`HINO(100:)` é equivalente a `HINO(100:236)`.

Pode-se também fazer concatenações com substrings:

`HINO(8:13)//HINO(69:80)` gera a constante `'aurora de setembro'`

`TROCADILHO= HINO(158:163)//'s '//HINO(197:205)` atribui à `TROCADILHO` o valor `'ímpias façanhas'`.

Se o resultado da expressão no lado direito for menor que o tamanho da variável, o valor é atribuído à variável começando-se pela esquerda e o espaço restante é preenchido por brancos:

```
CHARACTER(LEN= 10) ::  PARTE1, PARTE2
PARTE1= HINO(183:188) !  Resulta em PARTE1= 'Sirvam␣␣␣␣'
```

onde "␣" é o símbolo que indica um espaço em branco. Ao passo que se o resultado da expressão foi maior que o tamanho da variável, esta será preenchida completamente e o restante do valor será truncado:

```
PARTE2= HINO(:22)        ! Resulta em PARTE2= 'Como a aur'
```

Finalmente, é possível substituir parte de uma variável de caractere usando-se uma substring da mesma em uma atribuição:

```
HINO(50:113)= 'AAAAAAAAAAAARRRRRRRRRRRRRGGGGGGGGGGGG&
              &GHHHHHHHHHHHH!!!!!$%#$%%&%#%$#'
```

de tal forma que resulta,

```
HINO = 'Como a aurora precursora     &
      & do farol da divindaAAAAAAAAAA&
      &ARRRRRRRRRRRRRGGGGGGGGGGGGGHHH&
      &HHHHHHHHH!!!!!$%#$%%&%#%$#e.  &
      & Mostremos valor, constância  &
      & nesta ímpia, injusta guerra. &
      & Sirvam nossas façanhas       &
      & de modelo a toda Terra (...) '
```

Os lados esquerdo e direito de uma atribuição podem ser sobrepor. Neste caso, serão sempre os valores antigos os usados no lado direito da expressão. Por exemplo,

```
PARTE2(3:5) = PARTE2(1:3)
```

resulta em `PARTE2= 'CoComa aur'`.

4.6.4 Uso de operadores relacionais com caracteres

Comparações e uso de operadores relacionais com variáveis ou constantes de caracteres são possíveis entre caracteres únicos, inclusive com os operadores > e <. Neste caso, não se trata de testar qual caractere é *"maior"* ou *"menor"* que outro. Em um sistema computacional, caracteres possuem uma propriedade denominada **sequência de intercalação** (*collating sequence*) a qual ordena o armazenamento destes caracteres pelo sistema.

O Fortran determina que a sequência de intercalação para qualquer arquitetura deve satisfazer as seguintes condições:

- `A` precede (é menor) que `B`, que precede `C` ... precede `Y`, que precede `Z`.
- `a` precede (é menor) que `b`, que precede `c` ... precede `y`, que precede `z`.

- `0` precede `1`, que precede `2` ... precede `8`, que precede `9`.
- Espaço em branco precede `A` e `Z` precede `0`; ou branco precede `0` e `9` precede `A`.
- Espaço em branco precede `a` e `z` precede `0`; ou branco precede `0` e `9` precede `a`.

Assim, não há regra que estabeleça que os números devem preceder ou não as letras, nem tampouco há regra de precedência para caracteres especiais.

De acordo com estas regras, as expressões relacionais

```
'A' < 'B'
'0' < '1'
```

fornecem ambas o resultado `.TRUE.`. Uma comparação definitiva entre distintos caracteres, portanto, deve ser realizada de acordo com a sua tabela de intercalação. Por exemplo, se a espécie de caractere é determinada pela tabela de códigos ASCII (tabela 3.3), então `'A'` e `'B'` aparecem, respectivamente, nas posições 65 e 66 da sequência; portanto, `A < B = .TRUE.`. Por sua vez, `'a'` está na posição 97; portanto, `A > a = .FALSE.`. O padrão também oferece funções intrínsecas que fornecem acesso aos caracteres de uma determinada tabela. Essas funções são discutidas na seção 8.6.1.

Para comparar strings com mais de um caractere, as seguintes regras são empregadas:

- Se as strings forem de comprimentos distintos, a mais curta é suposta preenchida por espaços em branco à direita até igualar o comprimento da outra.
- A comparação é então realizada da esquerda para a direita, sendo que cada caractere de uma string é comparado com o correspondente caractere da outra de acordo com a tabela de intercalação até que a primeira diferença entre as strings seja detectada. O resultado desta última comparação irá determinar o resultado da expressão relacional.

De acordo com estas regras, `'123' == '123␣' = .TRUE.`, `'123' < '1234' = .TRUE.`, `'AB' > 'AAAA' = .TRUE.`.

Os resultados da aplicação das regras acima ainda são dependentes do processador, o que diminui a portabilidade do programa. Para contornar esta deficiência, o padrão estabelece as *funções de comparação léxica*, descritas na seção 8.6.2. Se estas funções forem empregadas no lugar dos operadores relacionais, então as strings serão sempre comparadas de acordo com a sequência de intercalação da tabela ASCII.

Para que duas strings possam ser comparadas, ambos devem ser da mesma espécie. Além disso, não é possível comparar strings com números. As regras de atribuição e conversão entre caracteres de diferentes conjuntos de caracteres (se suportados) já foi discutida na seção 3.3.4.1.

Finalmente, outras funções intrínsecas para realizar diversas manipulações com strings são descritas nas seções 8.6.3 e 8.7.

4.7 HIERARQUIA DAS DIFERENTES OPERAÇÕES

Quando uma determinada expressão contém combinações das diferentes expressões discutidas acima (numéricas, relacionais ou lógicas), estas são realizadas de acordo com a seguinte hierarquia:

1. Os operadores numéricos são aplicados, de acordo com as regras da seção 4.2.
2. Os operadores relacionais são aplicados, de acordo com as regras da seção 4.4.
3. Os operadores lógicos são aplicados, de acordo com as regras da seção 4.5.

Cabe reforçar mais uma vez que a hierarquia acima sempre pode ser alterada com o emprego de pares de parênteses.

4.8 Construtores de estruturas

Na seção 3.4 foi mostrado como os componentes de um tipo derivado podem ter seus valores definidos por meio de seletores de componentes, por inicialização padrão ou por construtores de estrutura. Com relação a este último recurso, foi exemplificado o caso mais simples, onde os componentes têm seus valores atribuídos de acordo com a concordância na ordem de ocorrência dos mesmos na definição do tipo com o ordenamento de valores no argumento do construtor de estrutura. Dessa maneira, a sequência

```
character(len= 4) :: ch4= 'abra'
character(len= 7) :: ch7= 'kadabra'
type :: ch12_t
   integer :: comp
   character(len= 12) :: valor
end type ch12_t
type(ch12_t) :: ch12= ch12_t(12, ch4//' '//ch7)
```

resultará com `ch12%comp= 12` e `ch12%valor= 'abra kadabra'`.

Este tipo de uso de um construtor de estrutura é sujeito a erros de programação se o tipo derivado tiver sido definido em outra unidade de programa, por exemplo. Por esta razão, a forma geral de um construtor de estrutura prevê o uso de *palavras-chave* (*keywords*) para auxiliar na atribuição dos componentes.

A forma geral de um construtor de estrutura é

```
<spec-tipo-derivado> ([<lista-spec-componentes>])
```

onde `<spec-tipo-derivado>` é o nome do tipo derivado e cada `<spec-componente>` na lista é

```
[<keyword>= ] <expr>
```

ou

```
[<keyword>= ] <alvo>
```

sendo que a palavra-chave `<keyword>` é o nome de um componente do tipo derivado, `<expr>` é uma expressão cujo resultado é atribuído ao componente, caso este não seja um ponteiro e `<alvo>` é o *alvo* (*target*) de um componente ponteiro.[1]

No exemplo ilustrado acima, nenhuma palavra-chave foi empregada e, por isso, as expressões devem aparecer na ordem correta no argumento do construtor de estrutura. O programador deve se encarregar da correta redação, pois nem sempre o compilador conseguirá detectar erros de programação neste ponto.

Se alguma palavra-chave for empregada, esta atribuição deverá aparecer no argumento do construtor após todos os `<spec-componentes>` sem palavras-chave, mas, agora, as palavras-chave podem aparecer em qualquer ordem. Nenhum componente pode aparecer mais de uma vez na `<lista-spec-componentes>`.

Usando palavras-chave, a atribuição no exemplo acima pode ser escrita

```
ch12= ch12_t(valor= ch4//' '//ch7, comp= 12)
```

Finalmente, se algum componente tiver sido inicializado na definição do tipo, este somente precisará aparecer no construtor de estrutura caso seja necessário alterar o seu valor. Se o componente não foi inicializado, então ele deve necessariamente aparecer no argumento do construtor de estrutura.

Sugestões de uso & estilo para programação

O emprego de palavras-chave é sempre sugerido com construtores de estrutura, para evitar a ocorrência de erros de programação.

[1] Ponteiros são discutidos na seção 7.2.

5

COMANDOS E CONSTRUTOS DE CONTROLE DE FLUXO

Nos capítulos anteriores foi descrito como comandos de atribuição devem ser escritos e como estes podem ser ordenados um após o outro para formar uma sequência de código, a qual é executada passo-a-passo. Na maior parte das computações, contudo, esta sequência simples de comandos é, por si só, inadequada para a formulação do problema. Por exemplo, podemos desejar seguir um de dois possíveis caminhos em uma seção de código, dependendo se um valor calculado é positivo ou negativo. Como outro exemplo, podemos querer somar 1000 elementos de uma matriz; escrever 1000 adições e atribuições é uma tarefa claramente tediosa e não muito eficiente. Claramente, a habilidade de realizar uma iteração sobre uma única adição é necessária. Podemos querer passar o controle de uma parte do programa a outra ou ainda parar completamente o processamento.

Para estes propósitos, recursos são disponíveis em Fortran que possibilitam o controle do fluxo lógico através dos comandos no programa. Os recursos contidos em Fortran correspondem aos que agora são geralmente reconhecidos como os mais apropriados para uma linguagem de programação moderna. Sua forma geral é a de um *construto de bloco* (*block construct*), o qual é um construto (uma construção ou estrutura) que começa com uma palavra-chave inicial, pode ter palavras-chave intermediárias e que termina com uma palavra-chave final que identifica a palavra-chave inicial. Cada sequência de comandos entre as palavras-chave é chamada de um *bloco*. Um bloco pode ser vazio, embora tais casos sejam raros.

Construtos executáveis podem ser *aninhados* (*nested*) ou *encadeados*, isto é, um bloco pode conter um outro construto executável. Neste caso, o bloco deve conter o construto interno por inteiro. Execução de um bloco sempre inicia com o seu primeiro comando executável.

Neste capítulo discutiremos os comandos e construtos que regulam o fluxo de execução de um programa em Fortran.

5.1 COMANDO E CONSTRUTO IF

O comando/construto `IF` fornece um mecanismo para controle de desvio de fluxo, dependendo de uma condição lógica. Há duas formas: o comando `IF` e o construto `IF`, sendo o último uma forma geral do primeiro.

5.1.1 COMANDO IF

No comando `IF`, o valor de uma expressão lógica escalar é testado e um único comando é executado se e somente se o seu valor for verdadeiro. A forma geral é:

```
IF (<expressão relacional e/ou lógica>) <comando executável>
```

O `<comando executável>` é qualquer, exceto aqueles que marcam o início ou o final de um bloco, como por exemplo `IF, ELSE IF, ELSE, END IF`, outro comando `IF` ou uma declaração `END`. Temos os seguintes exemplos:

```
IF (FLAG) EXIT       !Teste para determinar a saída (exit) de um bloco
IF(X-Y > 0.0) X= 0.0 !Teste para determinar se ocorre a atribuição X= 0.0
IF(COND .OR. P < Q .AND. R <= 1.0)S(I,J)= T(J,I) !Manipulação de matrizes
```

5.1.2 CONSTRUTO IF

Um construto IF permite que a execução de uma sequência de comandos (ou seja, um bloco) seja realizada, dependendo de uma condição ou de um outro bloco, dependendo de outra condição. Há três formas usuais para um construto IF. A forma mais simples tem a seguinte estrutura geral:

```
[<nome>:] IF (<expr-rel-log>) THEN
   <bloco>
END IF [<nome>]
```

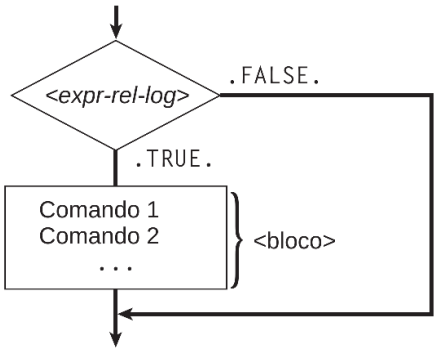


Figura 5.1: Fluxograma de um construto *IF* simples.

onde <bloco> denota uma sequência de linhas de comandos executáveis. O bloco é executado somente se o resultado da expressão relacional e/ou lógica <expr-rel-log> for verdadeiro. O construto IF pode ter um <nome>, o qual deve ser um nome válido em Fortran. O <nome> é opcional, mas se for definido no cabeçalho do construto, ele deve ser também empregado no final, denotado pela declaração END IF <nome>. A figura 5.1 mostra o fluxograma correspondente a esta forma básica do construto IF.

Como exemplo prático de um construto IF que segue o fluxograma da figura 5.1, temos:

```
SWAP: IF (X < Y) THEN
   TEMP= X
   X= Y
   Y= TEMP
END IF SWAP
```

As três linhas de texto entre "SWAP: IF ..." e "END IF SWAP" serão executadas somente se X < Y. Pode-se incluir outro construto IF ou outra estrutura de controle de fluxo no bloco de um construto IF.

A segunda forma usada para o construto IF é a seguinte:

```
[<nome>:] IF (<expr-rel-log>) THEN
   <bloco 1>
ELSE [<nome>]
   <bloco 2>
END IF [<nome>]
```

Na qual o <bloco 1> é executado se o resultado da <expr-rel-log> for verdadeira; caso contrário, o <bloco 2> será executado. Este construto permite que dois conjuntos distintos de instruções sejam executados, dependendo de um teste lógico. Um exemplo de aplicação desta forma intermediária seria:

```
IF (X < Y) THEN
   X= -Y
ELSE
   Y= -Y
END IF
```

Neste exemplo, se o resultado de X < Y for verdadeiro, então X= -Y, senão (ou seja, se X >= Y) a ação será Y= -Y.

A terceira e final versão usa a instrução ELSE IF para realizar uma série de testes independentes, cada um dos quais possui um bloco de comandos associado. Os testes são realizados um após o outro até que um deles seja satisfeito, em cujo caso o bloco associado é executado, enquanto que os outros são solenemente ignorados. Após, o controle do fluxo é transferido para a instrução END IF. Caso nenhuma condição seja satisfeita, nenhum bloco é executado, exceto se houver uma instrução ELSE final, que abarca quaisquer possibilidades não satisfeitas nos testes realizados no construto. A forma geral é:

```
[<nome>:] IF (<expr-rel-log>) THEN
   <bloco>
[ELSE IF (<expr-rel-log>) THEN [<nome>]
   <bloco>]
...
[ELSE [<nome>]
   <bloco>]
END IF [<nome>]
```

Pode haver qualquer número (inclusive zero) de instruções `ELSE IF` e, no máximo, uma instrução `ELSE`. Novamente, o `<nome>` é opcional, mas se for adotado no cabeçalho do construto, então deve ser mencionado em todas as circunstâncias ilustradas acima. A figura 5.2 mostra o fluxograma de um bloco `IF` aninhado com uma cláusula `ELSE IF` e a cláusula final `ELSE`.

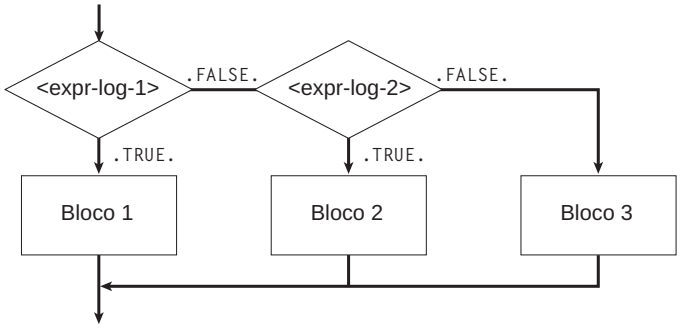


Figura 5.2: Fluxograma de um construto *IF* com uma cláusula *ELSE IF* e uma cláusula *ELSE*.

O exemplo a seguir ilustra um encadeamento de construtos `IF` que segue o fluxograma ilustrado na figura 5.2. Estruturas ainda mais complicadas são possíveis.

```
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
```

Sugestões de uso & estilo para programação

- Procure sempre avançar a margem esquerda do corpo de um bloco em relação ao cabeçalho por dois ou mais espaços para melhorar a visualização do código.
- Quando há um número grande de construtos aninhados, é recomendável adotar `<nomes>` para os construtos, como forma de facilitar a leitura e compreensão do programa.

O programa-exemplo a seguir faz uso de construtos `IF` para implementar o cálculo do fatorial de um número natural, já utilizando o construto `DO`, abordado na seção 5.2.

```
!Calcula o fatorial de um número natural.
program if_fat
implicit none
integer :: i, fat, j
!
print *, "Entre com valor:"
read *, i
```

```
if (i < 0) then
   print *, "Não é possível calcular o fatorial."
else if (i == 0) then
   print *, "fat(",i,")=",1
else
   fat= 1
   do j= 1, i
      fat= fat*j
   end do
   print *, "fat(",i,")=",fat
end if
end program if_fat
```

5.2 Construto DO

Um laço DO é usado quando for necessário calcular uma série de operações semelhantes, dependendo ou não de algum parâmetro que é atualizado em cada início da série. Por exemplo, para somar o valor de um polinômio de grau N em um dado ponto x:

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_N x^N = \sum_{i=0}^{N} a_i x^i.$$

Outro tipo de operações frequentemente necessárias são aquelas que envolvem operações matriciais, como o produto de matrizes, triangularização, *etc*. Para este tipo de operações repetidas, é conveniente usar-se um laço DO para implementá-las, ao invés de escrever o mesmo bloco de operações N vezes, como aconteceria se fosse tentado implementar a soma do polinômio acima através das seguintes expressões:

```
REAL :: POL,A0,A1,A2,...,AN
POL= A0
POL= POL + A1*X
POL= POL + A2*X**2
...
POL= POL + AN*X**N
```

A forma geral de um construto DO é a seguinte:

```
[<nome>:] DO [,] [<int-index> = <expressão 1>, <expressão 2> [, <expressão 3>]]
   <bloco>
END DO [<nome>]
```

onde <int-index> é uma variável inteira, denominada *índice do laço*, e as três expressões contidas no cabeçalho do construto devem resultar em liteirais inteiros. No cabeçalho acima, cada uma das expressões indica:

<expressão 1>: o valor inicial do <int-index>;

<expressão 2>: o valor máximo, ou limite, do <int-index> (não necessariamente deve ser o último valor assumido pela variável);

<expressão 3>: o passo (ou incremento) da variável em cada nova iteração. Se o passo for omitido, este será tomado igual a 1; se estiver presente, não pode ser nulo.

O número de iterações realizadas pelo laço é definido antes de se executar o primeiro comando do <bloco> e é dado pelo resultado da conta

```
MAX((<expressão 2> - <expressão 1> + <expressão 3>)/<expressão 3> , 0),
```

onde a função MAX() é definida na seção 8.3.2. Se o resultado do primeiro argumento da função MAX acima for negativo, o laço não será executado. Isto pode acontecer, por exemplo, se <expressão 2> for menor que <expressão 1> e o passo <expressão 3> for positivo. Este tipo de construto também é denominado um DO *iterativo.* A figura 5.3 mostra o fluxograma de um construto DO iterativo.

O valor do <int-index> (se presente) é incrementado pelo resultado da <expressão 3> ao final de cada laço para uso na iteração subsequente, se houver. Na saída do laço, o último valor adotado por essa variável permanece disponível para o programa, exceto se o laço estava inserido em um construto BLOCK (seção 9.15).

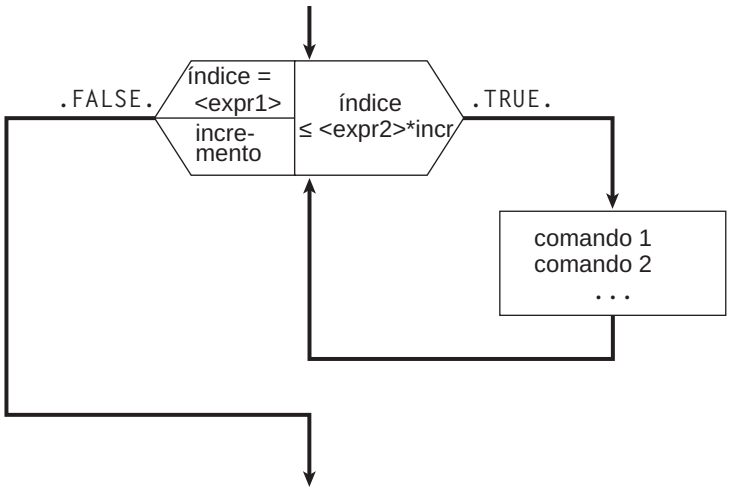


Figura 5.3: Fluxograma de um bloco *DO* iterativo.

O exemplo abaixo ilustra o uso de expressões no cabeçalho do construto:

```
DO I= J + 4, M, -K**2
   ...
END DO
```

Como se pode ver, o incremento pode ser negativo, o que significa, na prática, que os comandos do bloco somente serão executados se M <= J + 4. O exemplo a seguir ilustra um bloco DO onde a variável I somente assume valores ímpares:

```
DO I= 1, 11, 2
   ...
END DO
```

Caso alguma das expressões envolva o uso de variáveis, estas podem ser modificadas no bloco do laço, inclusive o passo das iterações. Entretanto, o número de iterações e o passo a ser realmente tomado não são modificados, uma vez que foram pré-determinados antes do início das iterações. O programa 5.1 exemplifica este fato.

Abaixo, temos um outro exemplo que ilustra o funcionamento do laço DO:

```
FAT= 1
DO I= 2, N
   FAT= FAT*I
END DO
```

Neste exemplo, o número de iterações será

$$\texttt{MAX(N-2+1,0)} = \texttt{MAX(N-1,0)}.$$

Caso N < 2, o laço não será executado, e o resultado será FAT= 1. Caso N >= 2, o valor inicial da variável inteira I é 2, esta é usada para calcular um novo valor para a variável FAT, em seguida a variável I será incrementada por 1 e o novo valor I= 3 será usado novamente para calcular o novo valor da variável FAT. Desta forma, a variável I será incrementada e o bloco executado até que I= N + 1, sendo este o último valor de I e o controle é transferido para o próximo comando após a declaração END DO. O exemplo ilustrado acima retornará, na variável FAT, o valor do fatorial de N.

Temos os seguintes casos particulares e instruções possíveis para um construto DO.

Listagem 5.1: Exemplo de uso do construto *DO*.

```
! Modifica o passo de um laço DO dentro do bloco.
program mod_passo
implicit none
integer :: i,j
! Bloco sem modificação de passo.
do i= 1, 10
   print *, i
end do
! Bloco com modificação do passo.
j= 1
do i= 1, 10, j
   if (i > 5)j= 3
   print *, i,j
end do
! Valor do índice após laço.
print*, 'Valor do índice i=', i
end program mod_passo
```

5.2.1 Construto DO ilimitado

Como caso particular de um construto DO, a seguinte instrução é possível:

```
[<nome>:] DO
   <bloco>
END DO [<nome>]
```

Neste caso, o conjunto de comandos contidos no <bloco> serão realizados sem limite de número de iterações, exceto se algum teste for incluído dentro do bloco, o que possibilita um desvio de fluxo para a primeira instrução após o END DO. Uma instrução que realiza este tipo de desvio é a instrução EXIT, descrita a seguir.

5.2.2 Instrução EXIT

Esta instrução permite a saída de qualquer um dos construtos de controle de fluxo discutidos neste capítulo, exceto dos construtos DO CONCURRENT[1] e CRITICAL.[2]

A forma geral desta instrução é:

```
EXIT [<nome>]
```

onde o <nome> é opcional.

Se o <nome> não for empregado, é assumido que a instrução EXIT ocorre em um construto DO e, neste caso, ele determina a saída do laço mais interno do construto. Este comportamento está ilustrado no fluxograma da figura 5.4

O uso do <nome> é opcional em construtos DO, mas é recomendado se houver diversos laços encadeados. Neste caso, a execução de um EXIT transfere controle ao primeiro comando executável após o END DO [<nome>] correspondente.

Se a instrução EXIT for empregada para forçar a saída de algum outro construto <const>, então o uso do <nome> é obrigatório e a execução da instrução transfere o controle de fluxo ao primeiro comando executável após a declaração end <const>.

Um exemplo que explora diversas possibilidades é o seguinte:

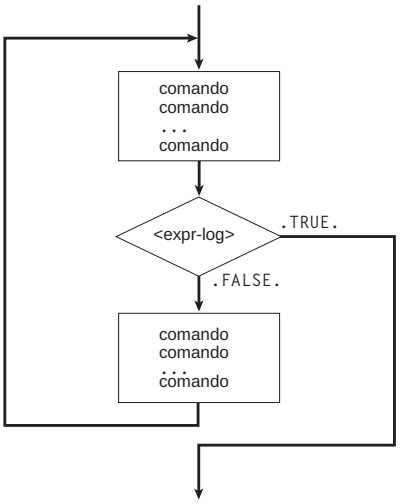


Figura 5.4: Fluxograma de um laço *DO* infinito com uma instrução *EXIT*.

[1] Discutido na seção 6.9.
[2] Empregado em coarrays.

```
DO I= 1, N
   ⋮
   IF(I == J) EXIT
   ⋮
END DO
K= I
```

Há três possibilidades:

1. Se no início do bloco `N` $\leqslant 0$, é realizada a atribuição `I= 1` mas o bloco não é executado; o controle passa para a instrução seguinte ao `END DO`, resultando com `K= 1`.
2. Se `N` $\geqslant$ `J`, a instrução EXIT será executada, resultando com `K= J`.
3. Se $0 <$ `N` $<$ `J`, o laço será executado `N` vezes; no final da última volta ocorrerá a atribuição `I= N + 1` e, ao sair do laço, resultará também `K= N + 1`.

5.2.3 INSTRUÇÃO CYCLE

A forma geral desta instrução é:

```
CYCLE [<nome>]
```

a qual transfere controle à declaração `END DO` do construto correspondente sem ocorrer a execução dos comandos posteriores à instrução. Assim, se outras iterações do laço restarem, estas são realizadas, incrementando-se o valor de <variável> (caso exista) pelo passo dado pela <expressão 3>. O programa a seguir ilustra o uso desta instrução.

```
program do_cycle
implicit none
integer :: indic= 0
do
   indic= indic + 1
   if(indic == 20) cycle ! Pula o valor índice = 20
   if(indic == 30) exit  ! Último valor impresso: índice = 29
   print *, "Valor do índice:",indic
end do
print *, "Saiu do laço.  Fim do programa."
end program do_cycle
```

5.3 CONSTRUTO CASE

O Fortran fornece uma outra alternativa para selecionar uma dentre diversas opções: trata-se do construto `CASE`. A principal diferença entre este construto e um bloco `IF` está no fato de somente uma expressão ser calculada para decidir o fluxo e esta pode ter uma série de resultados pré-definidos. A forma geral do construto `CASE` é:

```
[<nome>:] SELECT CASE (<expressão>)
[CASE (<seletor>) [<nome>]
   <bloco>]
...
[CASE DEFAULT
   <bloco>]
END SELECT [<nome>]
```

A <expressão> deve ser escalar e pode ser dos tipos inteiro, lógico ou de caractere e o valor especificado por cada <seletor> deve ser do mesmo tipo. No caso de variáveis de caracteres, os comprimentos podem diferir, mas não a espécie. Nos casos de variáveis inteiras ou lógicas, as espécies podem diferir. A forma mais simples do <seletor> é uma constante entre parênteses, como na declaração

```
CASE (1)
```

Para <expressões> inteiras ou de caracteres, um intervalo pode ser especificado separando os limites inferior e superior por dois pontos ": "

```
CASE (<inf> : <sup>)
```

Um dos limites pode estar ausente, mas não ambos. Caso um deles esteja ausente, significa que o bloco de comandos pertencente a esta declaração CASE é selecionado cada vez que a <expressão> calcula um valor que é menor ou igual a <sup>, ou maior ou igual a <inf>, respectivamente. Um exemplo é mostrado abaixo:

```
SELECT CASE (NUMERO)      ! NUMERO é do tipo inteiro.
CASE (:-1)                ! Todos os valores de NUMERO menores que 0.
   N_SINAL= -1
CASE (0)                  ! Somente NUMERO= 0.
   N_SINAL= 0
CASE (1:)                 ! Todos os valores de NUMERO > 0.
   N_SINAL= 1
END SELECT
```

A forma geral do <seletor> é uma lista de valores e de intervalos não sobrepostos, todos do mesmo tipo que <expressão>, tal como

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

Caso o valor calculado pela <expressão> não pertencer a nenhuma lista dos seletores, nenhum dos blocos é executado e o controle do fluxo passa ao primeiro comando após a declaração END SELECT. Já a declaração

```
CASE DEFAULT
```

é equivalente a uma lista de todos os valores possíveis de <expressão> que não foram incluídos nos outros seletores do construto. Portanto, somente pode haver um CASE DEFAULT em um dado construto CASE. O exemplo a seguir ilustra o uso desta declaração:

```
SELECT CASE (CH)       ! CH é do tipo de caractere.
CASE ('C', 'D', 'R':)
   CH_TYPE= .TRUE.
CASE ('I':'N')
   INT_TYPE= .TRUE.
CASE DEFAULT
   REAL_TYPE= .TRUE.
END SELECT
```

No exemplo acima, os caracteres 'C', 'D', 'R' e todos os caracteres após o último indicam nomes de variáveis do tipo de caractere. Os caracteres 'I' e 'N' indicam variáveis do tipo inteiro, e todos os outros caracteres alfabéticos indicam variáveis reais. Note que a sequência de caracteres determinada pelos intervalos 'R': ou 'I':'N' corresponde à sequência de intercalação do conjunto de caracteres empregado pela espécie.[3]

O programa-exemplo abaixo mostra o uso deste construto. Note que os seletores de caso somente testam o primeiro caractere da variável NOME, embora esta tenha um comprimento igual a 5.

```
program case_string
implicit none
character(len= 5) :: nome
print *, "Entre com o nome (5 caracteres):"
read "(a5)", nome      ! Lê string com 5 caracteres.
select case (nome)
case ("a":"z")         ! Seleciona nome que começa com letras minúsculas.
```

[3] Ver seção 4.6.

```
   print *, "Palavra inicia com Letra minúscula."
case ("A":"Z")          ! Seleciona nome que começa com letras maiúsculas.
   print *, "Palavra inicia com letras maiúscula."
case ("0":"9")          ! Seleciona números.
   print *, "Palavra inicia com números!!!"
case default            ! Outros tipos de caracteres.
   print *, "Palavra inicia com caractere especial!!!"
end select
end program case_string
```

Já o programa abaixo, testa o sinal de números inteiros:

```
program testa_case
implicit none
integer :: a
print *, "Entre com a (inteiro):"
read *, a
select case (a)
case (:-1)
   print *, "Menor que zero."
case (0)
   print *, "Igual a zero."
case (1:)
   print *, "Maior que zero."
end select
end program testa_case
```

Sugestões de uso & estilo para programação

Procure sempre incluir uma cláusula `DEFALT CASE` no construto para, no mínimo, gerar um aviso caso exista algum erro de lógica ou de execução no programa.

O programa abaixo ilustra o uso de alguns dos construtos discutidos neste capítulo.

```
! Imprime uma tabela de conversão das escalas Celsius e Fahrenheit
!   entre limites de temperatura especificados.
program conv_temp
implicit none
character(len= 1) :: escala
integer :: low_temp,high_temp,temp
real :: celsius,fahrenheit
!
read_loop: do ! Lê escala e limites.
   print *, "Escala de temperaturas (C/F):"
   read "(a)", escala
! Confere validade dos dados.
   if (escala /= "C" .and. escala /= "F")then
      print *, "Escala não válida!"
      exit read_loop
   end if
   print *, "Limites (temp. inferior , temp. superior):"
   read *, low_temp,high_temp
!
   do temp= low_temp, high_temp ! Laço sobre os limites de temperatura.
      select case (escala)      ! Escolhe fórmula de conversão
      case ("C")
         celsius= temp
         fahrenheit= 9*celsius/5.0 + 32.0
      case ("F")
         fahrenheit= temp
```

```
        celsius= 5*(fahrenheit - 32)/9.0
      end select
! Imprime tabela
      print *, celsius,"graus C correspondem a",fahrenheit,"graus F."
   end do
end do read_loop
!
! Finalização.
print *, "Final dos dados válidos."
end program conv_temp
```

6 Processamento de Matrizes

A definição e o processamento de matrizes e vetores sempre foi um recurso presente em todas as linguagens de programação, inclusive no Fortran 77. Uma novidade importante introduzida no Fortran 90 é a capacidade estendida de processamento das mesmas. A partir de então é possível trabalhar diretamente com a matriz completa, ou com seções da mesma, sem ser necessário o uso de laços `DO`. Novas funções intrínsecas agora atuam de forma *elemental* (em todos os elementos) em matrizes e funções podem retornar valores na forma de matrizes. Também estão disponíveis as possibilidades de matrizes alocáveis, matrizes de forma assumida e matrizes dinâmicas.

Estes e outros recursos serão abordados neste e nos próximos capítulos.

6.1 Terminologia e especificações de matrizes

Uma *matriz* ou *vetor*[1] é um outro tipo de objeto composto suportado pelo Fortran. Uma matriz consiste em um conjunto retangular de elementos, todos do mesmo tipo e espécie do tipo. Uma outra definição equivalente seria: uma matriz é um grupo de posições na memória do computador as quais são acessadas por intermédio de um único nome, fazendo-se uso dos **subscritos** da matriz. Este tipo de objeto é útil quando for necessário se fazer referência a um número grande, porém a princípio desconhecido, de variáveis do tipo intrínseco ou outras estruturas, sem que seja necessário definir um nome para cada variável.

O Fortran permite que uma matriz tenha até 15 (quinze) subscritos, cada um relacionado com uma dimensão da matriz. Os índices de cada subscrito da matriz são constantes ou variáveis inteiras e, por convenção, eles começam em 1, exceto quando um intervalo distinto de valores é especificado, através do fornecimento de um limite inferior e um limite superior.

Uma matriz pode ser declarada de diversas maneiras distintas. Neste capítulo, serão consideradas somente matrizes com um número fixo de elementos. O Fortran suporta também *matrizes alocáveis*, as quais são matrizes cujo número total de elementos pode ser definido dinamicamente durante a execução do programa. Este tipo de objeto dinamico de dados será abordado na seção 7.1.

Suponha que seja necessário declarar o vetor `A`, composto por 5 (cinco) elementos reais. Até o padrão Fortran 77, este vetor poderia ser declarado diretamente na declaração de tipo,

```
REAL A(5)
```

ou por meio da declaração `DIMENSION`,

```
REAL A
DIMENSION A(5)
```

Ambas as maneiras continuam sendo válidas e ambas criam um total de 10 variáveis reais com um nome em comum. Quando o nome `A` aparece sem os parênteses, este identifica a **matriz completa** (*whole array*), ao passo que as variáveis individuais armazenadas no vetor `A` são denominadas os **elementos da matriz** (*array elements*) e são acessadas por meio dos subscritos `A(1)`, `A(2)`, ..., `A(5)`, podendo aparecer tanto em expressões (lado direito da igualdade) quanto em atribuições (lado esquerdo). Na memória do computador, os elementos de uma matriz de tamanho fixo são armazenados em espaços contíguos de memória, sendo assim rapidamente acessados. Isto é ilustrado na figura 6.1.

[1] Nome usualmente empregado para uma matriz de uma dimensão.

Uma terceira forma de declaração do vetor `A` faz uso de uma constante inteira:

```
INTEGER, PARAMETER :: NMAX= 5
REAL A(NMAX)
```

Outros exemplos que fazem uso destas formas de declarações de matrizes são:

```
INTEGER NMAX
INTEGER POINTS(NMAX),MAT_I(50)
REAL R_POINTS(0:50), B
DIMENSION B(NMAX,50)
CHARACTER COLUMN(5)*25, ROW(10)*30
```

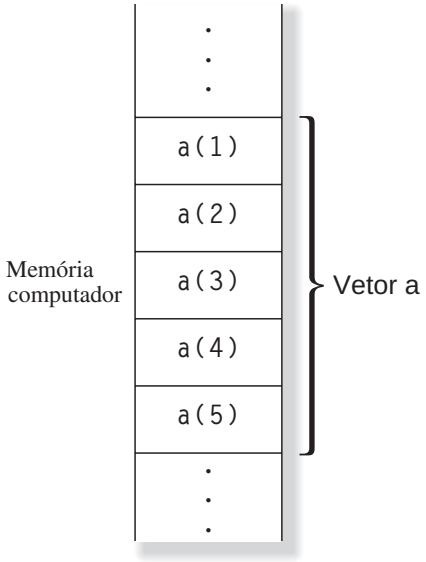


Figura 6.1: Os elementos do vetor *a* ocupam posições sucessivas na memória do computador.

No último exemplo acima, o vetor `COLUMN` possui 5 elementos, `COLUMN(1)`, `COLUMN(2)`, ..., `COLUMN(5)`, cada um destes sendo uma variável de caractere de comprimento 25. Já o vetor `ROW` possui 10 elementos, cada um sendo uma variável de caractere de comprimento 30. A matriz real `B` possui 2 dimensões, sendo `NMAX` linhas e 50 colunas. Todas as matrizes neste exemplo têm seus índices iniciando em 1, exceto pela matriz real `R_POINTS`, a qual inicia em 0: `R_POINTS(0)`, `R_POINTS(1)`, ..., `R_POINTS(50)`. Ou seja, este vetor possui 51 componentes.

A partir do Fortran 90, as dimensões de uma matriz podem ser especificadas também empregando-se o *atributo* `DIMENSION`. Por exemplo,

```
REAL, DIMENSION(5) :: A
REAL, DIMENSION(5:54) :: X ! Elementos de X: X(5), X(6), ..., X(54)
CHARACTER(LEN= 25), DIMENSION(5) :: COLUMN
CHARACTER(LEN= 30), DIMENSION(10) :: ROW
```

As formas de declarações de matrizes em Fortran 77 são aceitas no Fortran Moderno. Porém, é recomendável que estas sejam declaradas na forma de *atributos* de tipos de variáveis.

Antes de prosseguir, será introduzida a terminologia usada com relação a matrizes.

Posto. O *posto* (*rank*) de uma matriz é o número de dimensões da mesma. Assim, um escalar tem posto 0, um vetor tem posto 1 e uma matriz tem posto maior ou igual a 2.

Limites. Os *limites* (*bounds*) de uma matriz em uma dada dimensão são o menor e o maior valores dos índices naquela dimensão.

Extensão. A *extensão* (*extent*) de uma matriz também se refere a uma dimensão em particular e é o número de componentes naquela dimensão.

Forma. A *forma* (*shape*) de uma matriz é um vetor cujos componentes são a extensão de cada dimensão da matriz.

Tamanho. O *tamanho* (*size*) de um matriz é o número total de elementos que compõe a mesma. Este número pode ser zero, em cujo caso esta se denomina **matriz de tamanho zero** (*zero-sized array*).

Duas matrizes são ditas serem **conformáveis** (*conformable*) se elas têm a mesma forma. Todas as matrizes são conformáveis com um escalar, uma vez que o escalar é expandido em uma matriz com a mesma forma (ver seção 6.2).

Por exemplo, dadas as seguintes matrizes:

```
REAL, DIMENSION(-3:4,7) :: A
REAL, DIMENSION(8,2:8) :: B
REAL, DIMENSION(8,0:8) :: D
INTEGER :: C
```

A matriz `A` possui:

- posto 2;
- extensões 8 e 7;
- forma `[8,7]`, onde os símbolos "`[`" e "`]`" são os *construtores de matrizes*, isto é, eles definem ou inicializam os valores de um vetor ou matriz. Estes construtores de matrizes serão descritos com mais detalhes na seção 6.6.
- Tamanho $8 \times 7 = 56$.

Além disso, `A` é conformável com `B` e `C`, uma vez que a forma de `B` também é `[8,7]` e `C` é um escalar. Contudo, `A` não é conformável com `D`, uma vez que esta última tem forma `[8,9]`.

A forma geral da declaração de uma ou mais matrizes é como se segue:

```
<tipo-esp>[[, DIMENSION(<lista-exten>)] [, <atributos>] ::] <lista-nomes>
```

Entretanto, a forma recomendável da declaração é a seguinte:

```
<tipo-esp>, DIMENSION(<lista-exten>) [, <outros atributos>] :: <lista-nomes>
```

Onde `<tipo-esp>` pode ser um tipo e espécie intrínsecos de variável ou um tipo derivado (desde que a definição do tipo derivado esteja acessível). A `<lista-exten>` fornece as dimensões e extensões da matriz, através de:

- constantes inteiras;
- expressões inteiras usando variáveis **mudas** (*dummy*) ou constantes;
- somente o caractere "`:`" para indicar que a matriz é alocável (seção 7.1) ou de forma assumida (seção 9.3.6.2).

Os outros `<atributos>` podem ser quaisquer da seguinte lista:

```
PARAMETER   ALLOCATABLE   INTENT(INOUT)   OPTIONAL
SAVE        EXTERNAL      INTRINSIC       PUBLIC
PRIVATE     POINTER       TARGET
```

Os atributos contidos na lista acima serão abordados ao longo deste e dos próximos capítulos.

Finalmente, segue a `<lista-nomes>` válidos no Fortran, onde os mesmos são atribuídos às matrizes. Os seguintes exemplos mostram a forma da declaração de diversos tipos diferentes de matrizes.

1. Inicialização de vetores contendo 3 elementos:

```
INTEGER :: I
INTEGER, DIMENSION(3) :: IA= [1,2,3], IB= [(I, I=1,3)]
```

onde `IA` foi inicializada com um construtor de matrizes e `IB` foi inicializada via um *`DO` implícito*. Ambos os recursos serão discutidos na seção 6.6.

2. Declaração da matriz automática `LOGB`:

```
LOGICAL, DIMENSION(SIZE(LOGA)) :: LOGB
```

Aqui, a matriz `LOGA` é um argumento mudo de uma rotina (capítulo 9) e `SIZE` é uma função intrínseca que retorna um escalar inteiro correspondente ao tamanho do seu argumento.

3. Declaração das matrizes dinâmicas, ou alocáveis, de duas dimensões (posto 2) `A` e `B`:

```
REAL, DIMENSION(:,:), ALLOCATABLE :: A,B
```

A forma das matrizes será definida *a posteriori* por um comando `ALLOCATE`, discutido na seção 7.1.

4. Declaração das matrizes de *forma assumida* de três dimensões `A` e `B`:

```
REAL, DIMENSION(:,:,:) :: A,B
```

A forma das matrizes será assumida a partir das informações transferidas pela rotina que aciona o sub-programa onde esta declaração é feita. Este recurso também será discutido no capítulo 9.

MATRIZES DE TIPOS DERIVADOS. A capacidade de se misturar matrizes com definições de tipos derivados possibilita a construção de objetos de complexidade crescente. Alguns exemplos ilustram estas possibilidades.

Um tipo derivado pode conter um ou mais componentes que são matrizes:

```
TYPE :: TRIPLETO
   REAL :: U
   REAL, DIMENSION(3) :: DU
   REAL, DIMENSION(3,3) :: D2U
END TYPE TRIPLETO
TYPE(TRIPLETO) :: T
```

Este exemplo serve para definir, em uma única estrutura, um tipo de variável denominado `TRIPLETO`, cujos componentes correspondem ao valor de uma função de 3 variáveis (componente `U`), suas 3 derivadas parciais de primeira ordem (componente `DU`) e suas 9 derivadas parciais de segunda ordem (componente `D2U`). Se a variável `T` é do tipo `TRIPLETO`, `T%U` é um escalar real, mas `T%DU` e `T%D2U` são matrizes do tipo real.

É possível agora realizar-se combinações entre matrizes e o tipo derivado `TRIPLETO` para se obter objetos mais complexos. No exemplo abaixo, declara-se um vetor cujos elementos são `TRIPLETO`s de diversas funções distintas:

```
TYPE(TRIPLETO), DIMENSION(10) :: V
```

Assim, a referência ao objeto `V(2)%U` fornece o valor da função correspondente ao segundo elemento do vetor `V`; já a referência `V(5)%D2U(1,1)` fornece o valor da derivada segunda em relação à primeira variável da função correspondente ao elemento 5 do vetor `V`, e assim por diante.

O primeiro programa a seguir exemplifica um uso simples de matrizes:

```
!  Declara vetores, atribui valores aos elementos dos mesmos e imprime
!   estes valores na tela.
program ex1_array
use, intrinsic :: iso_fortran_env, only: dp => real64
implicit none
integer :: i
real, dimension(10) :: vr
real(kind= dp), dimension(10) :: vd

do i= 1,10
   vr(i)= sqrt(real(i))
   vd(i)= sqrt(real(i))
end do
print *, "Raiz quadrada dos 10 primeiros inteiros, em precisão simples:"
print *, vr      ! Imprime todos os componentes do vetor.
print *, " "
print *, "Raiz quadrada dos 10 primeiros inteiros, em precisão dupla:"
print *, vd
end program ex1_array
```

O segundo programa-exemplo, a seguir, está baseado no programa 3.6. Agora, será criado um vetor para armazenar os dados de um número fixo de alunos e os resultados somente serão impressos após a aquisição de todos os dados.

```
!Dados acadêmicos de alunos usando tipo derivado.
program alunos_vet
implicit none
integer :: i,ndisc= 5 !Mude este valor, caso seja maior.
type:: aluno
   character(len= 20):: nome
   integer:: codigo
   real:: n1,n2,n3,mf
end type aluno
```

```
type(aluno), dimension(5):: disc

do i= 1,ndisc
   print*, "Nome:"
   read "(a)", disc(i)%nome
   print*, "código:"
   read*,disc(i)%codigo
   print*, "Notas: N1,N2,N3:"
   read*,disc(i)%n1,disc(i)%n2,disc(i)%n3
   disc(i)%mf= (disc(i)%n1 + disc(i)%n2 + disc(i)%n3)/3.0
end do
do i= 1,ndisc
   print*, " "
   print*, "-------------> ",disc(i)%nome," (",disc(i)%codigo,") <-------------"
   print*, "          Média final: ",disc(i)%mf
end do
end program alunos_vet
```

6.2 EXPRESSÕES E ATRIBUIÇÕES ENVOLVENDO MATRIZES

Até o Fortran 77 não era possível desenvolver expressões envolvendo o conjunto de todos os elementos de uma matriz simultaneamente. Ao invés disso, cada elemento da matriz deveria ser envolvido na expressão separadamente, em um processo que com frequência demandava o uso de diversos laços `DO` encadeados. Quando a operação envolvia matrizes grandes, com 100×100 elementos ou mais, tais processos podiam ser extremamente dispendiosos do ponto de vista do tempo necessário para a realização de todas as operações desejadas, pois os elementos da(s) matriz(es) deveriam ser manipulado de forma sequencial. Além disso, o código tornava-se gradualmente mais complexo para ser lido e interpretado, à medida que o número de operações envolvidas aumentava. Este tipo de procedimento trabalhoso e suscetível a erros continua sendo necessário em algumas linguagens de programação modernas.

Um desenvolvimento novo, introduzido no Fortran 90, é a habilidade de realizar operações envolvendo a matriz na sua totalidade, possibilitando o tratamento de uma matriz como um objeto único, o que, no mínimo, facilita enormemente a construção, leitura e interpretação do código. Uma outra vantagem ainda mais importante que resulta dessa nova estratégia de manipulação de matrizes se aplica aos sistemas de *processamento distribuído* ou *processamento paralelo*. As normas definidas pelos comitês J3/WG5 para o padrão da linguagem Fortran supõe que compiladores usados em sistemas distribuídos devem se encarregar de distribuir automaticamente os processos numéricos envolvidos nas expressões com matrizes de forma equilibrada entre os diversos processadores que compõe a arquitetura. A evidente vantagem nesta estratégia consiste no fato de que as mesmas operações numéricas são realizadas de forma simultânea em diversos componentes distintos das matrizes, acelerando substancialmente a eficiência do processamento. Com a filosofia das operações sobre matrizes inteiras, a tarefa de implantar a paralelização do código numérico fica, essencialmente, a cargo do compilador e não do programador. Uma outra vantagem deste enfoque consiste na manutenção da portabilidade dos códigos numéricos.

Para que as operações envolvendo matrizes inteiras sejam possíveis, é necessário que as matrizes consideradas sejam *conformáveis*, ou seja, elas devem todas ter a mesma forma. Operações entre duas matrizes conformáveis são realizadas na maneira **elemental**, isto é, elemento a elemento e distribuindo as operações entre os diversos processadores, se existirem, e todos os operadores numéricos definidos para operações entre escalares também são definidos para operações entre matrizes.

Por exemplo, sejam `A` e `B` duas matrizes 2×3:

$$\mathtt{A} = \begin{pmatrix} 3 & 4 & 8 \\ 5 & 6 & 6 \end{pmatrix}, \; \mathtt{B} = \begin{pmatrix} 5 & 2 & 1 \\ 3 & 3 & 1 \end{pmatrix},$$

o resultado da adição de A por B é:

$$\mathtt{A}+\mathtt{B}=\begin{pmatrix}8&6&9\\8&9&7\end{pmatrix},$$

o resultado da multiplicação é:

$$\mathtt{A*B}=\begin{pmatrix}15&8&8\\15&18&6\end{pmatrix}$$

e o resultado da divisão é:

$$\mathtt{A/B}=\begin{pmatrix}3/5&2&8\\5/3&2&6\end{pmatrix}.$$

Se um dos operandos é um escalar, então este é distribuído (*broadcast*) em uma matriz conformável com o outro operando. Assim, o resultado de adicionar 5 a A é:

$$\mathtt{A}+5=\begin{pmatrix}3&4&8\\5&6&6\end{pmatrix}+\begin{pmatrix}5&5&5\\5&5&5\end{pmatrix}=\begin{pmatrix}8&9&13\\10&11&11\end{pmatrix}.$$

Esta distribuição de um escalar em uma matriz conformável também é empregada no momento da inicialização dos elementos de uma matriz.

Da mesma forma como acontece com expressões e atribuições escalares, em uma linha de programação como a seguir,

```
A= A + B
```

sendo A e B matrizes, a expressão do lado direito é desenvolvida antes da atribuição do resultado da expressão à matriz A. Este ordenamento é importante quando uma matriz aparece em ambos os lados de uma atribuição, como no caso do exemplo acima.

Além dos recursos algébricos para manipulações de matrizes, o padrão também estabelece uma abundância de rotinas intrínsecas (discutidas no capítulo 8) destinadas a realizar outras operações. As vantagens desta estratégia ficam evidentes nos exemplos a seguir:

1. Considere três vetores, A, B e C, todos do mesmo comprimento. Inicialize todos os elementos de A a zero e realize as atribuições `A(I)= A(I)/3.1 + B(I)*SQRT(C(I))` para todos os valores de $0 \leqslant \mathtt{I} \leqslant 20$:

   ```
   REAL, DIMENSION(20) :: A= 0.0, B, C
   A= A/3.1 + B*SQRT(C)
   ```

   Note como a função intrínseca SQRT opera de forma elemental sobre cada elemento do vetor C.

2. Considere uma matriz tri-dimensional. Encontre o maior valor menor que 1000 nesta matriz:

   ```
   REAL, DIMENSION(5,5,5) :: A
   REAL :: VAL_MAX
   VAL_MAX= MAXVAL(A, MASK=(A<1000.0))
   ```

   A função intrínseca MAXVAL devolve o maior valor entre os elementos de uma matriz. O argumento opcional MASK=(...) estabelece uma *máscara,* isto é, uma expressão lógica envolvendo a(s) matriz(es). Em MASK=(A<1000.0), somente aqueles elementos de A que satisfazem a condição de ser menores que 1000 são levados em consideração.

3. Encontre o valor médio dos elementos maiores que 3000 na matriz A do exemplo anterior:

   ```
   REAL, DIMENSION(5,5,5) :: A
   REAL :: MEDIA
   MEDIA= SUM(A,MASK=(A>3000.0))/COUNT(MASK=(A>3000.0))
   ```

   A função intrínseca SUM realiza a soma dos elementos da matriz que satisfazem a condição estabelecida pela máscara MASK, enquanto que a função COUNT conta quantos elementos da matriz satisfazem a mesma condição.

4. Dadas as matrizes `A(100,200)` e `B(200,300)`, obtenha a matriz `C(100,300)` a partir do produto matricial entre `A` e `B`:

```
REAL A(100,200), B(200,300), C(100,300)
C= MATMUL(A,B)
```

   Como o nome implica, a função intrínseca `MATMUL` realiza o produto matricial entre duas matrizes.

Todos as operações contidas nestes exemplos, as quais foram realizadas em apenas uma linha de código usando o Fortran, necessitariam de diversas linhas de programação em outras linguagens. Adicionalmente, o padrão exige que as operações realizadas por essas funções sejam sempre realizadas da maneira mais otimizada possível, tendo em vista o hardware disponível e as opções de compilação empregadas.

Uma matriz também pode ser uma constante nomeada e, neste caso, não é necessário declarar a sua forma, a qual será tomada diretamente do valor da matriz, empregando-se um asterisco no lugar do limite superior em uma dada dimensão da matriz. Isto caracteriza **matrizes de forma implícita** (*implied-shape array*s), exemplificadas em

```
REAL, DIMENSION(*), PARAMETER :: FIELD= [ 0.0, 10.0, 20.0 ]
CHARACTER, PARAMETER :: VOGAIS(*)= [ 'A', 'E', 'I', 'O', 'U' ]
```

6.3 SEÇÕES DE MATRIZES

Uma **submatriz**, também chamada **seção de matriz**, pode ser acessada através da especificação de um intervalo de valores de subscritos da matriz. Uma seção de matriz pode ser acessada e operada da mesma forma que a matriz completa, mas não é possível fazer-se referência direta a elementos individuais ou a subseções da seção. Neste caso, deve-se realizar a atribuição desta submatriz a uma outra matriz e então elementos individuais poderão ser acessados.

Seções de matrizes podem ser extraídas usando-se um dos seguintes artifícios:

- Um subscrito simples.
- Um tripleto de subscritos.
- Um vetor de subscritos.

Estes recursos serão descritos a seguir.

6.3.1 SUBSCRITOS SIMPLES

Um subscrito simples seleciona um único elemento da matriz. Considere a seguinte matriz 5×5, denominada `RA`. Então o elemento `X` pode ser selecionado através de `RA(2,2)`:

$$\texttt{RA} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & \texttt{X} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \texttt{RA}(2,2) = \texttt{X}.$$

6.3.2 TRIPLETO DE SUBSCRITOS

A forma de um tripleto de subscritos é a seguinte, sendo esta forma geral válida para todas as dimensões definidas para a matriz:

```
[<limite inferior>] : [<limite superior>] : [<passo>]
```

Se um dos limites, inferior ou superior (ou ambos) for omitido, então o limite ausente é assumido como o limite inferior ou superior, respectivamente, da correspondente dimensão da matriz da qual a seção está sendo extraída; se o `<passo>` for omitido, então assume-se `<passo>=1`.

Os exemplos a seguir ilustram várias seções de matrizes usando-se tripletos. Os elementos das matrizes marcados por `X` denotam a seção a ser extraída. Novamente, no exemplo será utilizada a matriz 5×5 denominada `RA`.

$$\texttt{RA} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & \texttt{X} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \texttt{RA(2:2,2:2) = RA(2,2) = X;}$$

Elemento simples, escalar.
Forma: $[1]$

$$\texttt{RA} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \texttt{X} & \texttt{X} & \texttt{X} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \texttt{RA(3,3:5);}$$

Seção de linha da matriz.
Forma: $[3]$

$$\texttt{RA} = \begin{bmatrix} 0 & 0 & \texttt{X} & 0 & 0 \\ 0 & 0 & \texttt{X} & 0 & 0 \\ 0 & 0 & \texttt{X} & 0 & 0 \\ 0 & 0 & \texttt{X} & 0 & 0 \\ 0 & 0 & \texttt{X} & 0 & 0 \end{bmatrix} \Longrightarrow \texttt{RA(:,3);}$$

3ª Coluna inteira.
Forma: $[5]$

$$\texttt{RA} = \begin{bmatrix} 0 & \texttt{X} & \texttt{X} & \texttt{X} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & \texttt{X} & \texttt{X} & \texttt{X} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & \texttt{X} & \texttt{X} & \texttt{X} & 0 \end{bmatrix} \Longrightarrow \texttt{RA(1::2,2:4);}$$

Seções de linhas com passo 2.
Forma: $[3, 3]$

6.3.3 Vetores de subscritos

Um vetor de subscritos é uma expressão inteira de posto 1, isto é, um vetor. Cada elemento desta expressão deve ser definido com valores que se encontrem dentro dos limites dos subscritos da matriz-mãe. Os elementos de um vetor de subscritos podem estar em qualquer ordem.

Um exemplo ilustrando o uso de um vetor de subscritos, denominado `IV`, é dado a seguir:

```
REAL, DIMENSION(6) :: RA= [ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 ]
REAL, DIMENSION(3) :: RB
INTEGER, DIMENSION(3) :: IV= [ 1, 3, 5 ] ! Expressão inteira de posto 1.
RB= RA(IV)                                ! IV é o vetor de subscritos.
! Resulta:
!RB= [ RA(1), RA(3), RA(5) ], ou
!RB= [ 1.2, 3.0, 1.0 ].
```

Um vetor de subscritos pode também estar do lado esquerdo de uma atribuição:

```
RA= [ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 ]
IV= [ 1, 3, 5 ]
RA(IV)= [ -1.2, 7.8, 5.6 ] !Atribuições dos elementos 1, 3 e 5 de RA.
! Resulta:
! RA(1)= -1.2; RA(3)= 7.8; RA(5)= 5.6
! Agora: RA= [ -1.2, 3.4, 7.8, 11.2, 5.6, 3.7 ]
```

Pode-se também repetir elementos em um vetor de subscritos, caso este seja empregado em expressões (*i.e.*, no lado direito). Por exemplo,

```
RA= [ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 ]
IV= [1, 3, 1]
RB= RA(IV)
! Resulta: RB= [ 1.2, 3.0, 1.2 ]
```

Neste caso, o que resulta é uma **seção de matriz muitos-de-um** (*many-one section*). De fato, com este recurso uma seção de matriz pode até ser mais extensa que a matriz original. Contudo, uma seção muitos-de-um não pode ser empregada em atribuições (*i.e.*, no lado esquerdo):

```
RA([ 1, 3, 1])= [ -1.2, 7.8, 5.6 ] ! Inválido.
```

Uma vez que o padrão não estabelece a ordem de atribuições neste caso.

6.4 ATRIBUIÇÕES DE MATRIZES E SUBMATRIZES

Tanto matrizes inteiras quanto seções de matrizes podem ser usadas como operandos (isto é, podem estar tanto no lado esquerdo quanto do lado direito de uma atribuição) desde que todos os operandos sejam conformáveis (página 64). Por exemplo,

```
REAL, DIMENSION(5,5) :: RA, RB, RC
INTEGER :: ID
...
RA= RB + RC*ID !Forma: [5,5].
...
RA(3:5,3:4)= RB(1::2,3:5:2) + RC(1:3,1:2)
!Forma [3,2]:
!RA(3,3)= RB(1,3) + RC(1,1)
!RA(4,3)= RB(3,3) + RC(2,1)
!RA(5,3)= RB(5,3) + RC(3,1)
!RA(3,4)= RB(1,5) + RC(1,2)
!etc.
...
RA(:,1)= RB(:,1) + RB(:,2) + RC(:3)
!Forma [5].
```

Um outro exemplo, acompanhado de figura, torna a operação com submatrizes conformáveis mais clara.

```
REAL, DIMENSION(10,20) :: A,B
REAL, DIMENSION(8,6) :: C
...
C= A(2:9,5:10) + B(1:8,15:20) !Forma: [8,6].
...
```

A figura 6.2 ilustra como as duas submatrizes são conformáveis neste caso e o resultado da soma das duas seções será atribuído à matriz `C`, a qual possui a mesma forma.

O programa-exemplo a seguir (programa 6.1) mostra algumas operações e atribuições básicas de matrizes. Nota-se que já se faz uso de instruções saída formatada de matrizes (seção 10.1) e de laços `DO` implícitos (seção 6.6).

Listagem 6.1: Primeiro programa envolvendo matrizes.

```
! Testa atribuições e seções de matrizes.
program testa_atr_matr
implicit none
real, dimension(3,3) :: a
real, dimension(2,2) :: b
integer :: i, j

do i= 1, 3
   do j= 1, 3
      a(i,j)= sin(real(i)) + cos(real(j))
   end do
end do
b= a(1:2, 1:3:2)
print*, "Matriz A:"
print"(3(f12.5))", ((a(i,j), j= 1,3), i= 1,3)
print*, "Matriz B:"
print"(2(f12.5))", ((b(i,j), j= 1,2), i= 1,2)
end program testa_atr_matr
```

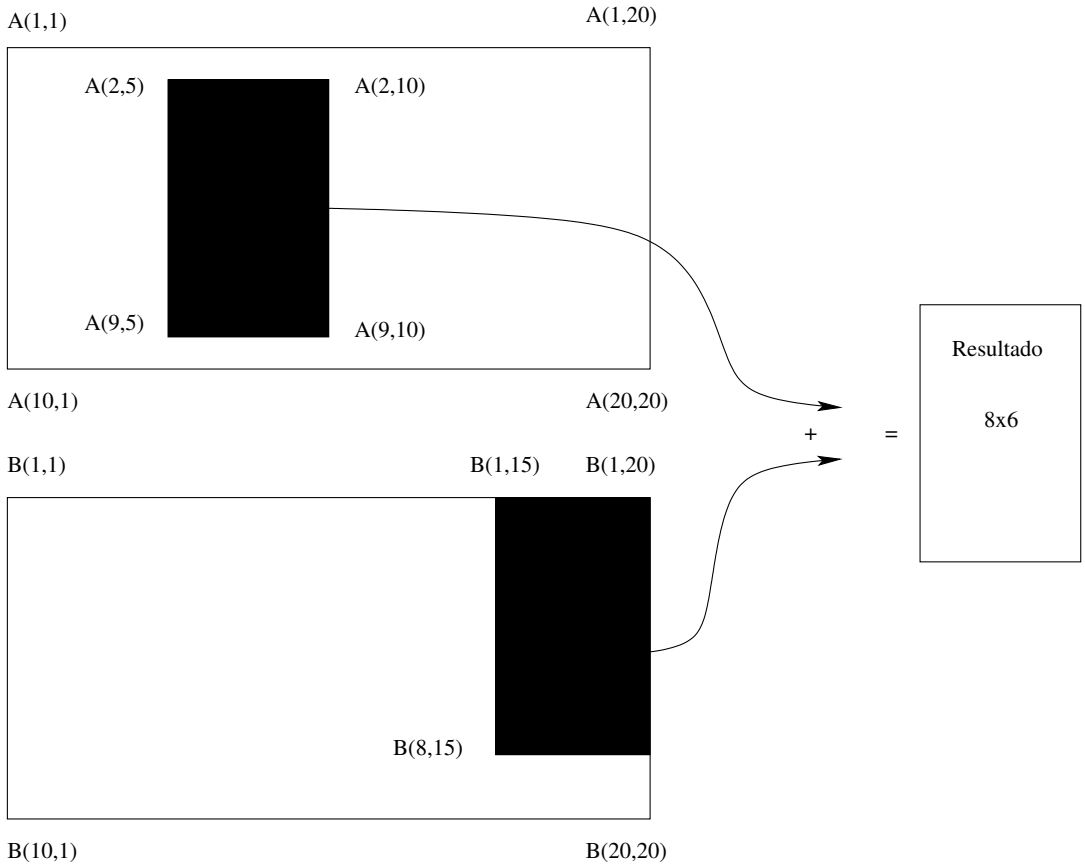


Figura 6.2: A soma de duas seções de matrizes conformáveis.

6.5 MATRIZES DE TAMANHO ZERO

Matrizes de tamanho zero (*zero-sized arrays*), ou *matrizes nulas* também são permitidas em Fortran. A noção de uma matriz nula é útil quando se quer contar com a possibilidade de existência de uma matriz sem nenhum elemento, o que pode simplificar a programação do código em certas situações. Uma matriz é nula quando o limite inferior de uma ou mais de suas dimensões é maior que o limite superior.

Por exemplo, o código abaixo resolve um sistema linear de equações que já estão na forma triangular:

```
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
```

Quando `I` assume o valor `N`, as submatrizes `B(N+1:N)` e `A(N+1:N,N)` se tornam nulas. Se esta possibilidade não existisse, seria necessária a inclusão de linhas adicionais de programação.

As matrizes nulas seguem as mesmas regras de operação e atribuição que as matrizes usuais, porém elas ainda devem ser conformáveis com as matrizes restantes. Por exemplo, duas matrizes nulas podem ter o mesmo posto mas formas distintas; as formas podem ser `[2,0]` e `[0,2]`. Neste caso, estas matrizes não são conformáveis. Contudo, uma matriz é sempre conformável com um escalar, assim a atribuição

```
<matriz nula>= <escalar>
```

é sempre válida.

6.6 CONSTRUTORES DE MATRIZES

Um *construtor de matrizes* (*array constructor*) cria um vetor (matriz de posto 1) contendo valores constantes. Estes construtores servem, por exemplo, para inicializar os elementos de um vetor, como foi exemplificado na página 65. Outro exemplo de uso dos construtores de matrizes está na definição de um vetor de subscritos, como foi abordado nas seções 6.3 e 6.4.

A forma geral de um construtor de matrizes é a seguinte:

```
(/ [<type-spec> :: ] <lista de valores do construtor> /)
```

ou

```
[ [<type-spec> :: ] <lista de valores do construtor> ]
```

sendo que a última forma (com os colchetes) é a preferida no padrão mais recente, pois diminui a quantidade de parênteses.

Nas definições acima, `<lista de valores do construtor>` pode ser tanto uma lista de constantes, como no exemplo

```
IV= [ 1, 3, 5 ]
```

como pode ser um conjunto de expressões numéricas e/ou com rotinas:

```
A= [ I+J, 2*I, 2*J, I**2, J**2, SIN(REAL(I)), COS(REAL(J)) ]
```

ou ainda uma instrução de **DO implícito** (*implied-do*) no construtor, cuja forma geral é a seguinte:

```
(<lista de valores do construtor>, <variável>= <exp1>, <exp2>, [<exp3>])
```

onde `<variável>` é uma variável escalar inteira e `<exp1>`, `<exp2>` e `<exp3>` são expressões escalares inteiras. A interpretação dada a este DO implícito é que a `<lista de valores dos construtor>` é escrita um número de vezes igual a

```
MAX((<exp2> - <exp1> + <exp3>)/<exp3>, 0)
```

sendo a `<variável>` substituída por `<exp1>`, `<exp1> + <exp3>`, ..., `<exp2>`, como acontece no construto DO (seção 5.2). Também é possível encadear-se DO's implícitos.

A seguir, alguns exemplos destes recursos são apresentados:

```
[ (i, i= 1,6) ] ! Resulta: [ 1, 2, 3, 4, 5, 6 ]


[ 7, (i, i= 1,4), 9 ] ! Resulta: [ 7, 1, 2, 3, 4, 9 ]


[ (1.0/REAL(I), I= 1,6) ]
! Resulta: [ 1.0/1.0, 1.0/2.0, 1.0/3.0, 1.0/4.0, 1.0/5.0, 1.0/6.0 ]


! DO's implícitos encadeados:
[ ((I, I= 1,3), J= 1,3) ]
! Resulta: [ 1, 2, 3, 1, 2, 3, 1, 2, 3 ]


[ ((I+J, I= 1,3), J= 1,2) ]
! = [ ((1+J, 2+J, 3+J), J= 1,2) ]
! Resulta: [ 2, 3, 4, 3, 4, 5 ]
```

Um construtor de matrizes pode também ser criado usando-se tripletos de subscritos:

```
[ A(I,2:4), A(1:5:2,I+3) ]
! Resulta: [ A(I,2), A(I,3), A(I,4), A(1,I+3), A(3,I+3), A(5,I+3) ]
```

Retornando à forma geral de um construtor de matrizes, <type-spec> se refere ao nome do tipo de objeto de dados (intrínseco ou derivado), seguido pelos parâmetros do tipo em parênteses. Se o construtor de matrizes inicia por <type-spec> ::, esta instrução determina o tipo e parâmetros do tipo das constantes que aparecem na <lista de valores do construtor> e os valores nesta lista são convertidos ao tipo e parâmetros do tipo adequados, dependendo do contexto em que o construtor é empregado. Um exemplo seria:

```
[ character(len= 33) :: 'the good', 'the bad', 'and', 'the appearance-challenged' ]
```

Todas as constantes de caracteres nesta lista têm comprimento 33 e serão submetidas às regras de conversão adequadas no momento da atribuição.

6.6.1 A FUNÇÃO INTRÍNSECA RESHAPE

Uma matriz de posto maior que um pode ser construída a partir de um construtor de matrizes através do uso da função intrínseca RESHAPE. Por exemplo,

```
RESHAPE( SOURCE= [ 1, 2, 3, 4, 5, 6 ], SHAPE= [ 2, 3 ] )
```

toma o vetor [1, 2, 3, 4, 5, 6] e gera com o mesmo a matriz de posto 2:

$$\begin{matrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{matrix}$$

a qual é uma matriz de forma [2, 3], isto é, 2 linhas e 3 colunas.

Um outro exemplo seria:

```
REAL, DIMENSION(3,2) :: RA
RA= RESHAPE( SOURCE= [ ((I+J, I= 1,3), J= 1,2) ], SHAPE= [ 3,2 ] )
```

O construtor em SOURCE gera o vetor [2, 3, 4, 3, 4, 5] e a instrução SHAPE organiza o vetor na forma [3,2], resultando em

$$\mathtt{RA} = \begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix}$$

A função RESHAPE também deve ser empregada para uma matriz de posto maior que um caso esta seja uma constante nomeada. Neste caso, novamente pode-se empregar asteriscos para os limites superiores em cada dimensão, como em

```
INTEGER, DIMENSION(0:*, *), PARAMETER :: POWERS= &
   RESHAPE( [ 0, 1, 2, 3, 0, 1, 4, 9, 0, 1, 8, 27 ], [ 4, 3 ])
```

A definição completa da função RESHAPE será apresentada na seção 8.14.3. Com a mesma, o programa-exemplo 6.1 apresenta uma forma mais concisa:

```
! Emprega a função intrínseca RESHAPE no programa 6.1
!   para definir a matriz a.
program testa_atr_matr
implicit none
real, dimension(3,3) :: a
real, dimension(2,2) :: b
integer :: i, j

a= reshape(source= [((sin(real(i))+cos(real(j)), i= 1,3), j= 1,3)], &
           shape= [3,3])
b= a(1:2,1:3:2)
print*, "Matriz A:"
print"(3(f12.5))", ((a(i,j), j= 1,3), i= 1,3)
Print*, "Matriz B:"
print"(2(f12.5))", ((b(i,j), j= 1,2), i= 1,2)
end program testa_atr_matr
```

6.6.2 A ORDEM DOS ELEMENTOS DE MATRIZES

A maneira como a função RESHAPE organizou os elementos das matrizes na seção 6.6.1 seguiu a denominada *ordem dos elementos de matriz*, a qual é a maneira como os compiladores de Fortran armazenam os elementos de matrizes em espaços contíguos de memória. Este ordenamento é obtido variando-se inicialmente o índice da primeira dimensão da matriz, depois variando-se o índice da segunda dimensão e assim por diante. Em uma matriz com 2 dimensões isto é obtido variando-se inicialmente as linhas e depois as colunas. A figura 6.3 ilustra este procedimento com a matriz B(5,4).

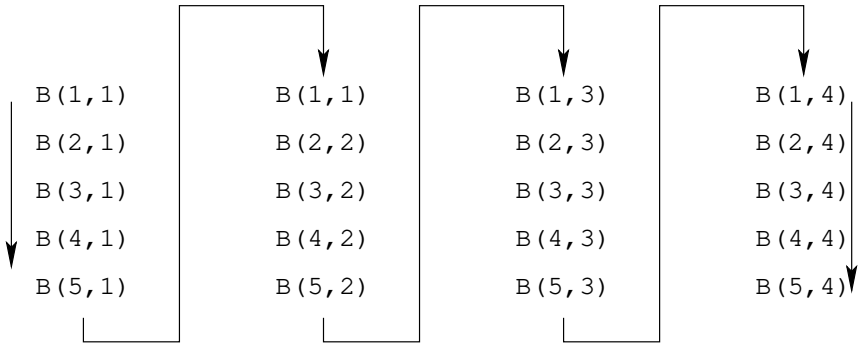


Figura 6.3: O ordenamento dos elementos da matriz *B(5,4)*.

Em uma matriz com mais de 2 dimensões, o ordenamento é realizado da mesma maneira. Assim, dada a matriz de posto 7

```
REAL, DIMENSION(-10:5, -20:1, 0:1, -1:0, 2, 2, 2) :: G
```

cujo tamanho é $16 \times 22 \times 2 \times 2 \times 2 \times 2 \times 2 = 11264$, o ordenamento segue a ordem:

$$
\begin{array}{l}
\mathtt{G}(-10,-20,0,-1,1,1,1) \to \mathtt{G}(-9,-20,0,-1,1,1,1) \to \ldots \to \mathtt{G}(5,-20,0,-1,1,1,1) \to \\
\mathtt{G}(-10,-19,0,-1,1,1,1) \to \mathtt{G}(-9,-19,0,-1,1,1,1) \to \ldots \to \mathtt{G}(5,-19,0,-1,1,1,1) \to \\
\mathtt{G}(-10,-18,0,-1,1,1,1) \to \mathtt{G}(-9,-18,0,-1,1,1,1) \to \ldots \to \mathtt{G}(5,-18,0,-1,1,1,1) \to \\
\qquad\qquad\qquad\qquad \vdots\!\vdots \qquad\qquad\qquad\qquad \vdots\!\vdots \qquad\qquad\qquad\qquad \vdots \\
\mathtt{G}(-10,-20,1,-1,1,1,1) \to \mathtt{G}(-9,-20,1,-1,1,1,1) \to \ldots \to \mathtt{G}(5,-20,1,-1,1,1,1) \to \\
\qquad\qquad\qquad\qquad \vdots\!\vdots \qquad\qquad\qquad\qquad \vdots\!\vdots \qquad\qquad\qquad\qquad \vdots \\
\mathtt{G}(-10,1,1,0,2,2,2) \to \mathtt{G}(-9,1,1,0,2,2,2) \qquad \to \ldots \to \mathtt{G}(5,1,1,0,2,2,2).
\end{array}
$$

Em muitas situações, o programa executável é mais eficiente se o código-fonte foi escrito levando em conta o ordenamento dos elementos de matrizes. As rotinas intrínsecas do Fortran que manipulam matrizes inteiras foram concebidas de forma a levar em conta este fato.

6.7 ROTINAS INTRÍNSECAS PARA MATRIZES

O padrão fornece uma biblioteca extensa, com diversas rotinas intrínsecas relacionadas com matrizes. Essas rotinas dividem-se em três categorias: *funções elementais*, *funções inquiridoras* ou *inquisidoras* e *funções transformacionais*. Nesta seção será apresentada somente uma descrição curta dessas rotinas; as definições completas serão apresentadas no capítulo 8.

6.7.1 ROTINAS INTRÍNSECAS ELEMENTAIS APLICÁVEIS A MATRIZES

O Fortran permite a existência de rotinas (funções ou subrotinas) intrínsecas *elementais*, isto é, rotinas que se aplicam a cada elemento de uma matriz. Matrizes podem, então, ser usadas como argumentos de rotinas intrínsecas, da mesma forma que escalares. As operações definidas

na rotina intrínseca serão aplicadas a cada elemento da matriz separadamente. Novamente, caso mais de uma matriz apareça no argumento da rotina, estas devem ser conformáveis.

A seguir, alguns exemplos de rotinas intrínsecas elementais. A lista completa destas rotinas pode ser obtida no capítulo 8.

1. Calcule as raízes quadradas de todos os elementos da matriz `A`. O resultado será atribuído à matriz `B`, a qual é conformável com `A`.

```
REAL, DIMENSION(10,10) :: A, B
B= SQRT(A)
```

2. Calcule a exponencial de todos os argumentos da matriz `A`. Novamente, o resultado será atribuído a `B`.

```
COMPLEX, DIMENSION(5,-5:15, 25:125) :: A, B
B= EXP(A)
```

3. Encontre o comprimento da string (variável de caractere) excluindo brancos no final da variável para todos os elementos da matriz `CH`.

```
CHARACTER(LEN= 80), DIMENSION(10) :: CH
INTEGER :: COMP
COMP= LEN_TRIM(CH)
```

6.7.2 Funções inquiridoras

São funções cujo valor depende das propriedades da matriz sendo investigada. A tabela 6.1 apresenta algumas funções inquisidoras de matrizes.

Tabela 6.1: Algumas funções inquisidoras de matrizes.

Nome da função e invocação	Propósito
`ALLOCATED(ARRAY)`	Determina o status de alocação de uma matriz alocável (seção 7.1)
`LBOUND(ARRAY, [DIM])`	Retorna os limites inferiores das extensões da matriz `ARRAY` em um vetor
`UBOUND(ARRAY, [DIM])`	Retorna os limites superioes das extensões da matriz `ARRAY` em um vetor
`SHAPE(ARRAY)`	Retorna a forma da matriz `ARRAY` em um vetor
`SIZE(ARRAY, [DIM])`	Retorna o tamanho da matriz ou a extensão da mesma ao longo da dimensão `DIM`

As definições das funções acima e de outras funções inquiridoras são dadas nas seções 8.2 e 8.13.

6.7.3 Funções transformacionais

São funções que têm como argumento uma ou mais matrizes. Ao contrário das funções elementais, as funções transformacionais atuam sobre a matriz como um todo. A tabela 6.2 apresenta algumas funções transformacionais de matrizes.

Várias das funções acima possuem opções adicionais. Suas definições e de muitas outras rotinas são dadas nas seções 8.12 e 8.14.

6.8 Comando e construto WHERE

Em muitas situações, é desejável realizar-se operações somente para alguns elementos de uma matriz, mediante a aplicação de alguma condição lógica aos seus elementos ou aos elementos de uma outra matriz conformável. O comando/construto `WHERE` oferece uma implementação para este tipo de problema.

Tabela 6.2: Algumas funções transformacionais de matrizes.

Nome da função e invocação	Propósito
ALL(MASK)	Função lógica que retorna .TRUE. se *todos* os elementos em MASK forem verdadeiros
ANY(MASK)	Função lógica que retorna .TRUE. se *algum* elemento em MASK for verdadeiro
COUNT(MASK)	Conta o número de elementos verdadeiros em MASK
MAXLOC(ARRAY)	Retorna em um vetor a *posição* do *maior* elemento de ARRAY
MAXVAL(ARRAY)	Retorna o *valor* do *maior* elemento de ARRAY
MINLOC(ARRAY)	Retorna em um vetor a *posição* do *menor* elemento de ARRAY
MINVAL(ARRAY)	Retorna o *valor* do *menor* elemento de ARRAY
FINDLOC(ARRAY, VALUE)	Retorna em um vetor as *posições* dos elementos de ARRAY que são iguais a VALUE
DOT_PRODUCT(VECTORA, VECTORB)	Retorna o produto escalar de dois vetores conformáveis
MATMUL(ARRAYA, ARRAYB)	Retorna o produto matricial de duas matrizes
PRODUCT(ARRAY)	Calcula o produto dos elementos de ARRAY
SUM(ARRAY)	Calcula a soma dos elementos de ARRAY
RESHAPE(ARRAY, SHAPE)	Muda a forma de ARRAY para a forma dada por SHAPE
TRANSPOSE(ARRAY)	Retorna a transporta de uma matriz de posto 2

6.8.1 COMANDO WHERE

O comando WHERE fornece esta possibilidade em uma única linha de instruções. A forma geral do comando é

```
WHERE (<expressão lógica matriz>) <variável matriz>= <expressão matriz>
```

A <expressão lógica matriz> deve ter a mesma forma que a <variável matriz>. A <expressão lógica matriz> é desenvolvida inicialmente, elemento a elemento. Em seguida, a <expressão matriz> será desenvolvida e os resultados atribuídos à <variável matriz>, novamente elemento a elemento, mas somente para aqueles subscritos nos quais a <expressão lógica matriz> teve resultado verdadeiro. Os elementos restantes permanecem inalterados.

Um exemplo simples é:

```
REAL, DIMENSION(10,10) :: A
WHERE (A > 0.0) A= 1.0/A
```

o qual fornece a recíproca (inversa) de todos os elementos positivos de A, deixando os demais inalterados.

6.8.2 CONSTRUTO WHERE

Uma única expressão lógica de matriz pode ser usada para determinar uma sequência de operações e atribuições em matrizes, todas com a mesma forma. A sintaxe deste construto é:

```
WHERE (<expressão matriz lógica>)
   <operações atribuições matrizes>
END WHERE
```

Inicialmente, a <expressão matriz lógica> é desenvolvida em cada elemento da matriz, resultando em uma matriz lógica temporária, cujos elementos são os resultados da <expressão matriz lógica>. Então, cada operação e atribuição de matriz no bloco do construto é executada sob o controle da *máscara* determinada pela matriz lógica temporária; isto é, as <operações atribuições matrizes> serão realizadas em cada elemento das matrizes do bloco que corresponda ao valor .TRUE. da matriz lógica temporária.

Existe também uma forma mais geral do construto WHERE que permite a execução de atribuições a elementos de matrizes que não satisfazem o teste lógico no cabeçalho:

```
WHERE (<expressão matriz lógica>)
   <operações atribuições matrizes 1>
ELSEWHERE
   <operações atribuições matrizes 2>
END WHERE
```

O bloco <operações atribuições matrizes 1> é executado novamente sob o controle da máscara definida na <expressão matriz lógica> e somente elementos que satisfazem esta máscara são afetados. Em seguida, as <operações atribuições matrizes 2> são executadas sob o controle da máscara definida por .NOT. <expressão matriz lógica>, isto é, novas atribuições são realizadas sobre elementos que não satisfazem o teste lógico definido no cabeçalho.

Um exemplo simples do construto WHERE é:

```
WHERE (PRESSURE <= 1.0)
   PRESSURE= PRESSURE + INC_PRESSURE
   TEMP= TEMP + 5.0
ELSEWHERE
   RAINING= .TRUE.
END WHERE
```

Neste exemplo, PRESSURE, INC_PRESSURE, TEMP e RAINING são todas matrizes com a mesma forma, embora não do mesmo tipo.

O programa-exemplo 6.2 mostra outra aplicação do construto WHERE.

Listagem 6.2: Exemplo do construto *WHERE*.

```
! Testa o construto WHERE
program testa_where
implicit none
real, dimension(3,3) :: a
integer :: i, j

a= reshape(source= [ ((sin(real(i)) + cos(real(j)), i= 1, 3), j= 1, 3) ], &
           shape= [ 3, 3 ])
print*, "Matriz A original:"
print*, a(1,:)
print*, a(2,:)
print*, a(3,:)

where (a >= 0.0)
   a= sqrt(a)
elsewhere
   a= a**2
end where
print*, "Matriz A modificada:"
print*, a(1,:)
print*, a(2,:)
print*, a(3,:)
end program testa_where
```

É possível também realizar-se o mascaramento na instrução ELSEWHERE, juntamente com a possibilidade de existir um número qualquer de instruções ELSEWHERE mascaradas mas com uma única instrução ELSEWHERE sem máscara, a qual deve ser a última. Todas as expressões lógicas que definem as máscaras devem ter a mesma forma. Adicionalmente, os construtos WHERE podem ser encadeados e nomeados; neste caso, as condições lógicas de todas as máscaras devem ter a mesma forma. O seguinte exemplo ilustra este recurso:

```
ATRIB1: WHERE (<cond 1>)
```

```
            <corpo 1>          !Mascarado por <cond 1>
         ELSEWHERE (<cond 2>) ATRIB1
            <corpo 2>          !Masc. por <cond 2> .AND. .NOT. <cond 1>
ATRIB2:     WHERE (<cond 4>)
               <corpo 4>       !Masc. por <cond 2> .AND. .NOT. <cond 1> .AND. <cond 4>
            ELSEWHERE ATRIB2
               <corpo 5>       !Masc. por <cond 2> .AND. .NOT. <cond 1>
               ...             !    .AND. .NOT. <cond 4>
            END WHERE ATRIB2
            ...
         ELSEWHERE (<cond 3>) ATRIB1
            <corpo 3>          !Masc. por <cond 3> .AND. .NOT. <cond 1>
            ...                !    .AND. .NOT. <cond 2>
         ELSEWHERE ATRIB1
            <corpo 6>          !Masc. por .NOT. <cond 1> .AND. .NOT. <cond 2>
            ...                !    .AND. .NOT. <cond 3>
         END WHERE ATRIB1
```

6.9 CONSTRUTO DO CONCURRENT

Uma forma aperfeiçoada do construto DO discutido na seção 5.2, o construto DO CONCURRENT, foi criada para possibilitar a execução em paralelo de iterações em laços. Para que este aperfeiçoamento possa ser empregado, o programador deve se certificar de que não existam interdependências entre as iterações do laço.

A lista de exigências para o uso do construto pode ser dividida em "limitações" a respeito do que pode ser colocado no bloco do construto e "garantias do programador" de que o código possui as propriedades que possibilitam a paralelização. Deve-se observar que se essas as exigências forem satisfeitas, não é necessário que o computador realmente disponha de múltiplos processadores ou que, caso os possua, que eles venham a ser realmente empregados. A implementação do construto também prevê que outros recursos existentes de optimização possam ser aplicados, tais como vetorização ou encadeamento de instruções (*pipelining*).

A forma geral do construto é:

```
DO [,] CONCURRENT ([<int-type-spec> ::] <index-spec-list> [, <scalar-mask-exp>])
   <do-conc-instrucs>
END DO
```

onde <int-type-spec> (caso presente) determina a espécie das variáveis inteiras dos índices, enquanto que <index-spec-list> é uma lista de especificações de índices, sendo que cada elemento da lista tem a forma

```
<index-var-name>= <valor-inicial> : <valor-final> [: <valor-passo>]
```

É importante ressaltar que cada <index-var-name> é local ao laço; isto é, terá sua ação limitada ao bloco do construto e qualquer variável com o mesmo nome que exista fora do construto não terá seu valor modificado. Se <int-type-spec> for omitido, a variável deve necessariamente ser do tipo inteiro padrão. Os campos <valor-inicial> , <valor-final> e <valor-passo> são expressões inteiras escalares e se <valor-passo> for omitido, é adotado <valor-passo>= 1.

Finalmente, a máscara opcional <scalar-mask-exp> é uma expressão lógica tal que somente aquelas iterações para as quais o resultado é verdadeiro serão realmente executadas.

Um exemplo simples do construto é:

```
DO CONCURRENT (I= 1:N)
   A(I,J)= A(I,J) + ALPHA*B(I,J)
END DO
```

durante cuja execução o índice J permanece constante.

Um outro exemplo usando uma máscara é :

```
DO CONCURRENT (I= 1:N, J= 1:M, I /= J)
   A(I,J)= A(I,J) + ALPHA*B(I,J)
END DO
```

Uma maneira de se escrever as manipulações realizadas neste DO CONCURRENT em termos de laço DO usuais é a seguinte:

```
DO I= 1, N
   DO J= 1, M
      IF(I /= J)A(I,J)= A(I,J) + ALPHA*B(I,J)
   END DO
END DO
```

Entretanto, existem diferenças fundamentais entre as duas estratégias:

1. Com os construtos DO usuais, os índices I e J não são locais; se estas variáveis forem usadas fora dos laços, elas terão seus valores alterados na saída.
2. Os laços DO usuais não são otimizados; para tanto, o programador deverá usar diretivas especiais de compilação e alguma estratégia de otimização.
3. O laço em J é previamente estabelecido como o laço interno, enquanto que o laço em I é o externo. Com o construto DO CONCURRENT o oposto também pode ocorrer; na verdade, os laços são executados em uma ordem qualquer, determinada pelo compilador.

As exigências impostas para o uso do construto são descritas a seguir. Os primeiros itens são os casos proibidos em um construto DO CONCURRENT:

- Alguma instrução que determina o encerramento não usual do construto ou de alguma iteração: um comando RETURN (seção 9.3.1), uma instrução EXIT ou uma instrução CYCLE com o nome de um construto DO externo.
- Uma instrução de controle de imagem.[2]
- Uma referência a uma rotina que não é *pura* (seção 9.10).
- Uma referência a uma das rotinas intrínsecas destinadas à manipulação de exceções de ponto flutuante.[3]
- Uma instrução de entrada/saída com um especificador ADVANCE= (seções 10.7 e 10.8).

O padrão exige que o compilador detecte estes casos, exceto o caso com o especificador ADVANCE.

Além das exigências acima, é necessário também que o programador escreva o código de tal forma que os resultados obtidos em uma iteração não afetem as outras iterações. Desta forma, as iterações podem ser realizadas em qualquer ordem, a qual será determinada pelo compilador. Em particular, é necessário que:

- qualquer variável referenciada em uma dada iteração, ou foi definida na mesma iteração ou seu valor não é afetado por qualquer outra iteração, a qual pode estar ocorrente simultaneamente;
- qualquer ponteiro referenciado em uma dada iteração, ou foi associado a um *alvo* na mesma iteração ou não tem o seu status de associação alterado por uma outra iteração;
- qualquer objeto alocável (seção 7.3) que é alocado ou dealocado em uma dada iteração não será usado por qualquer outra iteração, exceto se em cada iteração ocorrerem a alocação e a dealocação deste objeto;
- registros (ou posições) em um arquivo (capítulo 10) não serão ambos escritos por uma iteração e lidos por outra.

Registros podem ser escritos em um arquivos sequencial (seção 10.6) por mais de uma iteração, mas a ordem em que os registros são gravados no arquivo é indeterminada.

Se as condições acima forem satisfeitas e o contruto for executado, ao final da execução do mesmo,

[2] Aplica-se a coarrays, não discutidos nesta Apostila.
[3] Este conteúdo também não será abordado nesta Apostila.

- qualquer variável cujo valor é afetado por mais do que uma iteração se torna indefinida ao final do laço;
- qualquer ponteiro cujo status de associação (seção 7.2) é alterado por mais do que uma iteração passa a ter o status indefinido.

Também é importante enfatizar que mesmo que todos os requisitos sejam cumpridos e os laços sejam efetivamente paralelizados, isto não garante que o código executável será executado mais rapidamente que um código criado a partir de laços DO usuais ou por meio de alguma outra estratégia de otimização. Isto é particularmente verdade quando o número de iterações for pequeno, pois qualquer ganho em otimização será compensado com prejuízo devido ao custo adicional (*overhead*) em tempo de execução imposto pela inicialização de linhas (*threads*) paralelas de execução. Independente disso, o compilador pode ser esperto o suficiente para detectar que mesmo o laço DO é vetorizável/paralelizável e gerar o código incluindo as mesmas estratégias de otimização que o DO CONCURRENT. Em resumo, para verificar se existe uma real vantagem no uso do DO CONCURRENT, é recomendável que sejam realizados alguns testes comparativos. O programa abaixo implementa um destes testes.

Listagem 6.3: Compara desempenho de álgebra matricial com três abordagens distintas.

```
program compare_loop_times
use, intrinsic :: iso_fortran_env, only: dp => real64
implicit none
integer, parameter :: n= 10000000 ! n= 10^7
integer, parameter :: loop_count= 20
integer :: i, j
real :: time_begin, time_end, time, fconv= 1.0e6
real(kind= dp), dimension(n) :: x, y, z

! Atribui valores aos vetores x e y
call random_number(x) ! A rotina random_number() atribui valores
call random_number(y) !  aleatórios às matrizes (seção 8.17.3)

! Testa sintaxe de matriz inteira
call cpu_time(time_begin)
do j= 1, loop_count
   z= x + y
end do
call cpu_time(time_end)
time= fconv*(time_end - time_begin)
print'(a,g0)', 'Tempo para operação de matriz inteira (us): ', time
print'(a,g0)', 'Tempo médio (s):                             ',  time/loop_count

! Testa construto DO usual
call cpu_time(time_begin)
do j= 1, loop_count
   do i= 1, n
      z(i)= x(i) + y(i)
   end do
end do
call cpu_time(time_end)
time= fconv*(time_end - time_begin)
print'(/,a,g0)', 'Tempo para construto DO (us):                ', time
print'(a,g0)', 'Tempo médio (us):                            ',  time/loop_count

! Testa construto DO CONCURRENT
call cpu_time(time_begin)
do j= 1, loop_count
   do concurrent (i= 1:n)
      z(i)= x(i) + y(i)
   end do
end do
call cpu_time(time_end)
time= fconv*(time_end - time_begin)
print'(/,a,g0)', 'Tempo para construto DO CONCURRENT(s):       ', time
```

```
print '(a,g0)', 'Tempo médio (s):                                    ', time/loop_count

end program compare_loop_times
```

O programa `compare_loop_times.f90` foi testado com os compiladores Intel® Fortran e GFortran com diferentes opções de compilação em uma arquitetura contendo um núcleo com 8 processadores Intel® Core™ i7-4710HQ @ 2.50GHz. Os tempos médios de execução em cada caso são reportados na tabela 6.3. As opções de compilação foram as seguintes:

GFortran (SO)	`gfortran compare_loop_times.f90`	(opções-padrão)
GFortran (O1)	`gfortran -O3 compare_loop_times.f90`	(maior otimização)
Intel (SO)	`ifort compare_loop_times.f90`	(opções-padrão)
Intel (O1)	`ifort -fast -parallel compare_loop_times.f90`	(maior otimização)

Os resultados reportados na tabela 6.3 foram obtidos após repetidas execuções do programa (para cada compilação), uma vez que os tempos medidos podem variar substancialmente entre diferentes execuções. A tabela mostra que houve um ganho substancial de desempenho com o compilador GFortran quando foram empregadas as opções de máxima otimização. Já com o compilador da Intel® (`ifort`) não houve variação significativa entre as compilações com opções-padrão ou com otimização máxima. Observou-se também que para este teste em particular, ambos os compiladores geraram executáveis com performances equivalentes.

Tabela 6.3: Tempos médios de execução (em μs) das operações algébricas no programa 6.3 com diferentes compiladores e opções de compilação.

	GFortran (SO)	GFortran (O1)	Intel (SO)	Intel (O1)
Matriz inteira	$2,36 \times 10^4$	$0,150$	$0,100$	$0,200$
Laço `DO`	$3,19 \times 10^4$	$0,100$	$0,051$	$0,100$
Laço `DO CONCURRENT`	$2,39 \times 10^4$	$0,050$	$0,051$	$0,049$

7 Objetos e Estruturas Dinâmicas de Dados

Nos capítulos anteriores, em particular nos capítulos 3 e 6, foram abordados objetos de dados escalares ou matriciais, definidos como tipos intrínsecos ou derivados. Todos esses objetos possuem duas características em comum: eles armazenam algum tipo de valor de dado e todos são *estáticos*, no sentido de que o número e os tipos desses objetos são declarados antes da execução do programa. Neste caso, ao se rodar o programa executável, sabe-se de antemão o espaço de memória necessário para tanto.

A memória total destinada ao programa é dividida em diversos segmentos, de acordo com a natureza dos objetos de dados e do espaço de instruções contidas no programa. Especificamente, nas arquiteturas que são tipicamente empregadas atualmente, o espaço destinado aos dados estáticos fica em duas regiões com baixo endereçamento de memória adjacentes: o *data segment* (para dados inicializados) e o *bss* (*basic service set*) *segment* (para dados não inicializados).[1] Ambos os segmentos ocupam um valor fixo de memória.

Um outro segmento existente é o *stack*, o qual contém intervalos variáveis de memória destinados a armazenar as instruções das rotinas chamadas pelo programa e os dados passados e declarados estaticamente nas mesmas. Uma vez que o processamento é retornado ao ponto de onde a rotina foi invocada, esse espaço de memória é liberado.

Porém, existe um outro tipo de objeto de dados que é *dinâmico*, no sentido de que não se conhece de antemão o número e/ou a natureza desses objetos; essas informações somente serão conhecidas no decorrer da execução do programa. Os objetos que são criados dinamicamente durante a execução do programa são armazenados em um outro segmento de memória denominado *heap*, o qual também permite a destinação de intervalos variáveis de memória para os dados dinâmicos, com a posterior liberação desse intervalo quando os objetos forem destruídos.

Um exemplo imediato da necessidade de objetos dinâmicos de memória consiste na interface de um experimento. O equipamento de medida monitora constantemente um determinado processo físico, gerando dados experimentais em intervalos regulares de tempo. Se o tempo total de experimento não é conhecido de antemão, então o número total de amostras medidas pelo equipamento também não é conhecido. Nesta situação, é ideal que os dados obtidos do experimento sejam primeiro armazenados em uma estrutura dinâmica de dados.

O Fortran oferece três recursos básicos para armazenar objetos dinâmicos de dados: *matrizes alocáveis*, *ponteiros* (*pointers*) ou *objetos alocáveis*. Esses recursos serão apresentados neste capítulo.

7.1 Matrizes alocáveis

Matrizes alocáveis são objetos compostos, todos do mesmo tipo e espécie, criados dinâmicamente. O Fortran fornece tanto matrizes alocáveis quanto matrizes automáticas, ambos os tipos sendo matrizes dinâmicas. Usando matrizes alocáveis, é possível alocar e dealocar espaço de memória conforme o necessário. O recurso de matrizes automáticas permite que matrizes locais em uma função ou subrotina tenham forma e tamanho diferentes cada vez que a rotina é invocada. Matrizes automáticas são discutidas na seção 9.3.6.3.

[1] Maiores informações a respeito da organização da memória podem ser obtidos em https://medium.com/@shoheiyokoyama/understanding-memory-layout-4ef452c2e709, https://www.geeksforgeeks.org/memory-layout-of-c-program/ ou https://en.wikipedia.org/wiki/Data_segment.

7.1.1 DEFINIÇÃO E USO BÁSICO

Uma matriz alocável é declarada na linha de declaração de tipo de variável com o atributo ou a declaração ALLOCATABLE. O posto da matriz deve também ser declarado com a inclusão dos símbolos de dois pontos ":", um para cada dimensão da matriz. Ou seja, a matriz tem o posto fixo, mas a sua forma é arbitrária. Este tipo de objeto é denominado uma **matriz de forma deferida** (*deferred-shape array*). Por exemplo, a matriz real de duas dimensões A é declarada como alocável através da declaração:

```
REAL, DIMENSION(:,:), ALLOCATABLE :: A
```

a qual empregou o atributo ALLOCATABLE ou por meio da declaração ALLOCATABLE:

```
REAL :: A
ALLOCATABLE :: A(:,:)
```

Estas declarações não alocam espaço de memória à matriz imediatamente, como acontece com as declarações usuais de matrizes. O status da matriz nesta situação é *not currently allocated*, isto é, correntemente não alocada. Espaço de memória é dinamicamente alocado durante a execução do programa, logo antes da matriz ser utilizada, usando-se o comando ALLOCATE. Este comando especifica os limites da matriz, seguindo as mesmas regras definidas na seção 6.1. Um exemplo de aplicação deste comando, usando expressões inteiras na alocação é:

```
ALLOCATE (A(0:N+1,M))
```

sendo as variáveis inteiras escalares N e M previamente determinadas. Como se pode ver, esta estratégia confere uma flexibilidade grande na definição de matrizes ao programador.

O espaço alocado à matriz com o comando ALLOCATE pode, mais tarde, ser liberado com o comando DEALLOCATE. Este comando requer somente nome da matriz previamente alocada. Por exemplo, para liberar o espaço na memória reservado para a matriz A, o comando fica

```
DEALLOCATE (A)
```

Os comandos ALLOCATE e DEALLOCATE são empregados não somente para alocar espaço de memória para matrizes, mas também para objetos alocáveis em geral, e as suas formas mais gerais serão apresentadas na seção 7.3. Especificamente para matrizes alocáveis, as instruções têm as formas

```
ALLOCATE (<lista-alloc> [, STAT= <status>] [, ERRMSG= <err-mess>])
DEALLOCATE (<lista-alloc> [, STAT= <status>] [, ERRMSG= <err-mess>])
```

onde <lista-alloc> é uma lista de alocações, sendo que cada elemento da lista tem a forma

```
<obj-allocate>(<lista-ext-array>)
```

com <obj-allocate> sendo o nome da matriz e <lista-ext-array> a lista das extensões de cada dimensão da matriz na forma

```
[<limite-inf>:]<limite-sup>
```

sendo <limite-inf> e <limite-sup> expressões inteiras escalares. Como o usual, se <limite-inf> está ausente, é tomado <limite-inf>= 1.

Se a palavra-chave STAT= está presente no comando, <status> status deve ser uma variável inteira escalar, a qual recebe o valor zero se o procedimento do comando ALLOCATE/DEALLOCATE foi bem sucedido ou um valor positivo se houve um erro no processo (*e. g.*, se não houver espaço suficiente na memória para alocar a matriz). Se o especificador STAT= não estiver presente e ocorrer um erro no processo, o programa é abortado.

A palavra-chave ERRMSG também se refere ao status da alocação/dealocação, sendo que o campo <err-msg> contém uma variável de caractere escalar padrão. Se ERRMSG= estiver presente e um erro ocorrer na execução do comando, o sistema gera uma mensagem de erro que é gravada em <err-msg>. Essa mensagem pode ser utilizada para controle de erros na execução do programa.

Um exemplo curto, mas incluindo todas as opções seria:

```
CHARACTER(LEN= 200) :: ERR_MESS ! Ou outro comprimento apropriado
INTEGER :: N, ERR_STAT
REAL, DIMENSION(:), ALLOCATABLE :: X
ALLOCATE(X(N), STAT= ERR_STAT, ERRMSG= ERR_MESS)
IF(ERR_STAT > 0)THEN
   PRINT*, 'Alocação de X falhou:', TRIM(ERR_MESS) ! Função TRIM(): seção 8.7.2
END IF
```

É possível alocar ou dealocar mais de uma matriz simultaneamente na <lista-alloc>. Um breve exemplo do uso destes comandos seria:

```
REAL, DIMENSION(:), ALLOCATABLE :: A, B
REAL, DIMENSION(:,:), ALLOCATABLE :: C
INTEGER :: N
...
ALLOCATE (A(N), B(2*N), C(N,2*N))
B(:N)= A
B(N+1:)= SQRT(A)
DO I= 1,N
   C(I,:)= B
END DO
DEALLOCATE (A, B, C)
```

Matrizes alocáveis satisfazem a necessidade frequente de declarar uma matriz tendo um número variável de elementos. Por exemplo, pode ser necessário ler variáveis, digamos tam1 e tam2, e então declarar uma matriz com tam1 × tam2 elementos:

```
INTEGER :: TAM1, TAM2
REAL, DIMENSION (:,:), ALLOCATABLE :: A
INTEGER :: STATUS
...
READ*, TAM1, TAM2
ALLOCATE (A(TAM1,TAM2), STAT= STATUS)
IF (STATUS > 0)THEN
... ! Comandos de processamento de erro.
END IF
... ! Uso da matriz A.
DEALLOCATE (A)
...
```

No exemplo acima, o uso do especificador STAT= permite que se tome providências caso não seja possível alocar a matriz, por alguma razão.

Como foi mencionado anteriormente, uma matriz alocável possui um *status de alocação* (*allocation status*). Quando a matriz é declarada, mas ainda não alocada, o seu status é *unallocated* ou *not currently allocated*. Quando o comando ALLOCATE e bem sucedido, o status da matriz passa a *alocado* (*allocated)*; uma vez que ela é dealocada, o seu status retorna a *not currently allocated*. Assim, o comando ALLOCATE somente pode ser usado em matrizes não correntemente alocadas, ao passo que o comando DEALLOCATE somente pode ser usado em matrizes alocadas; caso contrário, ocorre um erro.

É possível verificar se uma matriz está ou não correntemente alocada usando-se a função intrínseca ALLOCATED.[2] Esta é uma função lógica com um argumento, o qual deve ser o nome de uma matriz alocável. Se a matriz está alocada, a função retorna .TRUE., caso contrário, retorna .FALSE.. Usando-se esta função, comandos como os seguintes são possíveis:

```
IF (ALLOCATED(A)) DEALLOCATE (A)
```

ou

```
IF (.NOT. ALLOCATED(A)) ALLOCATE (A(5,20))
```

[2] Ver seção 8.2.

Finalmente, há duas restrições no uso de matrizes alocáveis:

1. Matrizes alocáveis não podem ser argumentos mudos de uma rotina e devem, portanto, ser alocadas e dealocadas dentro da mesma unidade de programa (ver capítulo 9).
2. O resultado de uma função não pode ser uma matriz alocável (embora possa ser uma matriz).

O programa-exemplo a seguir ilustra o uso de matrizes alocáveis.

```
program testa_aloc
implicit none
integer, dimension(:,:), allocatable :: b
integer :: i,j,n= 2

print*, "Valor inicial de n:",n
allocate (b(n,n))
b= n
print*, "Valor inicial de B:"
print"(2(i2))", ((b(i,j), j= 1,n), i= 1,n)
deallocate (b)
n= n + 1
print*, "Segundo valor de n:",n
allocate (b(n,n))
b= n
print*, "Segundo valor de B:"
print"(3(i2))", ((b(i,j), j= 1,n), i= 1,n)
deallocate (b)
n= n + 1
print*, "Terceiro valor de n:",n
allocate (b(n+1,n+1))
b= n + 1
print*, "Terceiro valor de B:"
print"(5(i2))", ((b(i,j), j= 1,n+1), i= 1,n+1)
end program testa_aloc
```

Sugestões de uso & estilo para programação

Uma prática útil para evitar erros em códigos extensos: sempre inclua a cláusula `STAT=` nas alocações e sempre verifique o status. Desta maneira, é possível tomar providências caso ocorra algum erro de alocação.

7.1.2 Recursos avançados de alocação/realocação de matrizes

Eventualmente torna-se necessário alterar a forma de uma matriz previamente alocada. O Fortran oferece mais de uma alternativa para realizar essa tarefa.

7.1.2.1 A rotina intrínseca MOVE_ALLOC

A sub-rotina intrínseca `MOVE_ALLOC` realiza a transferência da alocação de uma matriz para outra, ambas previamente alocadas e com os mesmos tipos, espécies e postos. Se não for empregada em coarrays, a rotina é *pura*.[3]

A chamada desta subrotina é realizada pela instrução

```
CALL MOVE_ALLOC(FROM, TO)
```

onde:

[3] A respeito de rotinas puras, ver seção 9.10.

FROM é uma matriz alocável de qualquer tipo e espécie. No âmbito da subrotina, FROM possui a *intenção* (*intent*) INOUT.[4]

TO é uma matriz alocável com os mesmos tipo, espécie e posto de FROM. No âmbito da subrotina, TO possui a intenção OUT.

Após a chamada e execução da subrotina, o status de alocação, a geometria e os elementos de TO são aqueles que FROM possuia anteriormente e FROM se torna dealocado. Se TO tem o atributo TARGET, qualquer ponteiro que estava anteriormente associado com FROM passa a se associar com TO; caso contrário, esses ponteiros se tornam indefinidos.

A subrotina MOVE_ALLOC pode ser usada para minimizar o número de operações de cópia requeridas quando se torna necessário alterar o tamanho de uma matriz, uma vez que somente uma operação de cópia da matriz original é necessária, ao invés de duas como aconteceria com o emprego de laços.

O programa abaixo ilustra o uso desta subrotina.

```
! Este programa usa MOVE_ALLOC para aumentar o tamanho
!  do vetor X, mantendo os seus elementos iniciais.
program tes_Move_Alloc
implicit none
integer :: i, n= 5
real, dimension(:), allocatable :: x, y

allocate (x(n), y(2*n))          ! Y é maior que X
! Testa os status de alocação
print'(2(a,g0))', 'X está alocado? ',allocated (X),'  Tamanho de X: ',size(x)
print'(2(a,g0))', 'Y está alocado? ',allocated (Y),'  Tamanho de Y: ',size(y)

call random_number(x)   ! Atribui valores a X
print '(/,a)', 'Elementos originais de X:'
print'(5(g0,x))', x

do i= 1, 5         ! Copia todo X nos primeiros elementos de Y
   y(i) = x(i)     ! Esta é a única cópia de dados necessária
end do
print '(/,a)', 'Elementos de Y:'
print'(10(g0,x))', y

call move_alloc (y, x)   ! X agora tem o dobro do tamanho original, Y foi
                         ! dealocado e os elementos de X foram preservados.

print'(/,a)', 'Após MOVE_ALLOC:'
print'(2(a,g0))', 'X está alocado? ',allocated (X),'  Tamanho de X: ',size(x)
print'(a,g0)', 'Y está alocado? ', allocated (Y)
print'(/,a)', 'Novo vetor X é:'
print'(10(g0,x))', x
end program tes_Move_Alloc
```

7.1.2.2 ALOCAÇÃO/REALOCAÇÃO AUTOMÁTICA

O recurso de *alocação/realocação automática* (ou *alocação na atribuição*) foi introduzido mais recentemente e torna a manipulação de matrizes alocáveis ainda mais conciso e automatizado. Com este recurso, praticamente não há mais a necessidade do comando ALLOCATE ou da subrotina MOVE_ALLOC, exceto em casos muito particulares.

O exemplo a seguir tipifica usos deste recurso:

```
INTEGER, DIMENSION(5) :: A= [ 1, 2, 3, 4, 5]
INTEGER, DIMENSION(:), ALLOCATABLE :: B
```

[4] Acerca da qualificação INTENT, ver seção 9.3.2.

```
REAL, DIMENSION(10) :: RA
REAL, DIMENSION(:), ALLOCATABLE :: RB
B= [ -2, -1, 0 ]  ! B é automaticamente alocado e definido
B= [ B, A ]       ! Agora, B= [-2, -1, 0, 1, 2, 3, 4, 5]
B= [ B, 6, 7, 8 ] ! Agora, B= [-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
CALL RANDOM_NUMBER(RA) ; RA= RA - 0.5
RB= PACK(RA, RA > 0.0) ! RB é alocado com os valores positivos de RA
```

Na última linha acima, foi empregada a função `PACK` (seção 8.14.2), a qual irá selecionar somente os elementos positivos de `RA`, automaticamente alocar `RB` com o número de elementos correto e depositar no mesmo esses valores positivos.

O programa 7.1 mostra usos mais sofisticados deste recurso. O recurso da alocação/realocação automáticas fornece uma solução possível ao problema colocado no início desta seção: o armazenamento de um número previamente indeterminado de dados. Esse recurso se torna ainda mais poderoso quando aplicado em matrizes alocáveis de tipos derivados, com componentes também alocáveis, o que é o assunto discutido na seção 7.1.3.

Listagem 7.1: Exemplos de uso de alocação/realocação automáticas

```
program tes_reallocation_on_assignment
use iso_fortran_env, only: dp => real64
implicit none
logical, dimension(:), allocatable :: tes
real(kind= dp), dimension(:), allocatable :: xd, yd, hd
complex(kind= dp), dimension(:), allocatable :: zd
integer :: n, i

write(*, fmt= '(a)', advance= 'no')'Enter n= ' ; read(*,*)n
allocate(yd(n), hd(n))
call random_number(yd)
xd= yd
tes= xd > 0.5_dp
print'(/,a)', 'Array xd:'
write(*, fmt= '(g0, x)') (xd(i), i= 1, n)
print'(/,2(a,g0))', 'size of tes: ', size(tes), '  # of x > 1/2: ', count(tes)
print'(a)', 'Array tes:'
print*, tes

call random_number(yd) ; call random_number(hd)
zd= cmplx(yd, hd, kind= dp)
print'(/,a)', 'Array zd:'
write(*, fmt= '(a,g0,a,g0,a,x)') ('(', zd(i)%re, ',', zd(i)%im, ')', i= 1, n)

xd= pack(xd, tes)
print'(2/,a,g0)', 'Array xd packed: size= ', size(xd)
write(*, fmt= '(g0, x)') (xd(i), i= 1, size(xd))
zd= pack(zd, tes)
print'(/,a,g0)', 'Array zd packed: size= ', size(zd)
write(*, fmt='(a,g0,a,g0,a,x)') ('(',zd(i)%re,',', zd(i)%im,')',i= 1,size(zd))
end program tes_reallocation_on_assignment
```

O uso do recurso da alocação/realocação é padrão na linguagem. Para evitar que o código executável rode com este recurso, é necessário utilizar opções de compilação:

GFortran: `-fno-realloc-lhs`[5]

Intel® fortran: `-assume norealloc_lhs`[6]

Por exemplo, compilando o programa 7.1 com a linha

[5] Ref: https://gcc.gnu.org/onlinedocs/gfortran/Code-Gen-Options.html. Ver também opção `-Wrealloc-lhs`.

[6] Ref: https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-assume.

```
user@machine|dir>gfortran -fno-realloc-lhs tes_auto_reallocation.f90
```

irá gerar um executável, mas a execução do mesmo será abortada, com uma mensagem de erro na primeira tentativa de realizar a alocação automática:

```
user@machine|dir>./a.out
Enter n= 10
Program received signal SIGSEGV: Segmentation fault - invalid memory
    reference.
Backtrace for this error:
#0  0x7f090babbcea
#1  0x7f090babae75
#2  0x7f090b75feff
#3  0x401646
#4  0x40222f
#5  0x7f090b74bf32
#6  0x40117d
#7  0xffffffffffffffff
Segmentation fault (core dumped)
```

7.1.3 ALOCAÇÃO DINÂMICA DE COMPONENTES DE TIPOS DERIVADOS

Obviamente, também é possível declarar matrizes alocáveis de tipos derivados. Adicionalmente, os componentes do tipo derivado também podem ser alocáveis, com o benefício adicional do recurso de alocação na atribuição ou realocação. Isto permite criar objetos dinâmicos com complexidade grande o suficiente para suprir a maior parte dos casos que requerem tais objetos. Um cuidado que precisa ser tomado se refere a inicializações-padrão de componentes do tipo derivado. Uma tentativa de inicialização de um componente alocável não faz sentido e não é permitida pelo padrão.

O programa 7.2 ilustra o uso de matrizes alocáveis de tipos derivados com componentes alocáveis.

Listagem 7.2: Alocação dinâmica de tipos derivados e seus componentes.

```
program tes_derived_type_realloc
implicit none
integer, parameter :: nloop= 5
integer :: i
real, dimension(nloop) :: vr2
type :: t1
   real, dimension(:), allocatable :: vr1
end type t1
type(t1) :: st1                          ! Escalar estático do tipo t1
type(t1), dimension(nloop) :: vt2        ! Vetor estático do tipo t1
type(t1), dimension(:), allocatable :: vt1 ! Vetor dinâmico do tipo t1

print'(a)', 'Alocação automática de componente de um tipo &
           &derivado escalar estático:'
call random_number(vr2)
print'(a,*(g0,x))', 'vr2: ', vr2
st1= t1(vr1= vr2)
print'(a,*(g0,x))', 'st1: ', st1%vr1

print'(/,a)', 'Primeiro laço: alocação automática de componente &
             &de vetor estático de tipo derivado:'
do i= 1, nloop
   call random_number(vr2)
   print'(a,g0,a,*(g0,x))', 'i= ', i, '  vr2: ', vr2
```

```
   vt2(i)= t1(vr1= vr2)
end do
do i= 1, nloop
   print'(a,g0,a,*(g0,x))', 'i= ', i, '  vt2: ', vt2(i)%vr1
end do

print'(/,a)', 'Segundo laço: alocação automática de componente &
             &de vetor dinâmico de tipo derivado:'
allocate(vt1(0)) ! Primeiro, aloque vt1 como uma matriz de tamanho zero
do i= 1, nloop
   call random_number(vr2)
   print'(a,g0,a,*(g0,x))', 'i= ', i, '  vr2: ', vr2
   vt1= [vt1, t1(vr1= vr2)] ! Executa realocação automática
end do

do i= 1, nloop
   print'(a,g0,a,*(g0,x))', 'i= ', i, '  vt1: ', vt1(i)%vr1
end do
end program tes_derived_type_realloc
```

Sugestões de uso & estilo para programação

Quando matrizes alocáveis são empregadas, é conveniente sempre dealocar as mesmas assim que não mais são necessárias. Isto porque matrizes alocáveis, como objetos dinâmicos, são alocadas no *heap*, o qual não é liberado automaticamente com uma saída de rotina, por exemplo. Isto cria os chamados *vazamentos de memória* (*memory leaks*), que podem facilmente esgotar toda a memória disponível.

7.2 Ponteiros (*pointers*)

Um ponteiro (*pointer*) é um outro tipo dinâmico de dado. Até agora, todos os objetos de dados considerados, mesmo as matrizes alocáveis, armazenam valores de algum dos tipos intrínsecos ou de tipos derivados. Um ponteiro, por outro lado, armazena um *endereço de memória*, o qual corresponde à localização da região da memória do computador onde o dado está contido. Como subprogramas também ocupam espaços na memória do computador, um ponteiro também pode ser empregado para armazenar o endereço de um subprograma. Ponteiros também podem ser empregados em conjunto com tipos derivados para estabelecer estruturas dinâmicas de dados mais gerais do que matrizes alocáveis. Exemplos dessas estruturas dinâmicas são listas encadeadas e árvores, entre outras. Ponteiros também oferecem em certos casos métodos para realizar manipulações de um número grande de dados de forma eficiente, pois permitem reduzir o número de operações de leitura/escrita desses dados na memória, diminuindo assim o tempo de processamento.

A figura 7.1 é uma ilustração típica da diferença entre um objeto de dados usual e um ponteiro. Sejam `var = 100` uma variável inteira padrão e o seu valor. Supondo que um inteiro padrão ocupe 4 bytes = 32 bits de memória, o endereço de memória `0x123` que aparece acima da variável corresponde à localização do primeiro destes 32 bits. O ponteiro, `ptr` então "aponta" para `var`, a qual se tornou o "alvo" do ponteiro; mais especificamente, `ptr` passou a armazenar o endereço do primeiro bit ocupado por `var`, como está ilustrado na figura. Como `var` ocupa ao todo 32 bits de memória, é necessário que o programa também tenha essa informação. Por isso, em diversas linguagens tais como Fortran, C ou C++, os ponteiros também têm diferentes tipos, os quais podem ser os tipos intrínsecos ou outros tipos derivados. Portanto `ptr` é um ponteiro do tipo inteiro que armazena o endereço inicial de `var`. Como o ponteiro possui um valor (um endereço), ele também precisa ocupar uma região da memória (o endereço de `ptr` é `0x155`).

A descrição apresentada acima sobre a diferença entre variáveis e ponteiros é aplicável para qualquer linguagem que trabalhe com ponteiros, em particular para C/C++. Contudo, os ponteiros em Fortran podem armazenar também outras informações a respeito de seus alvos, além

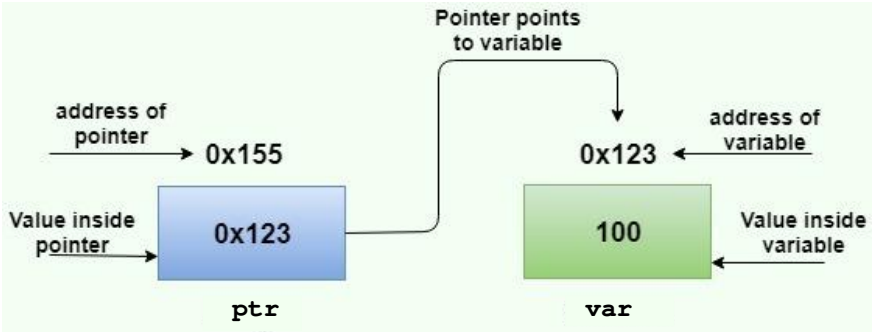


Figura 7.1: O ponteiro `ptr` armazena o endereço de memória da variável inteira `var`.

de seus endereços. Como será visto posteriormente, ponteiros para matrizes também armazenam informações a respeito da geometria das mesmas. Além disso, a maneira como o Fortran manipula ponteiros e os emprega em expressões ou atribuições é também ligeiramente diferente da estratégia adotada no C/C++. Um ponteiro somente pode "apontar" para um objeto estático de dados se este último for explicitamente declarado como um **alvo** (*target*).

Pode-se perguntar então: "por que objetos estáticos de dados devem ser declarados como alvos em Fortran, uma vez que outras linguagens, como C/C++, não precisam disso?" A razão está nos métodos usuais empregados por compiladores para a otimização do código. Quando se solicita ao compilador que este gere o código executável mais eficiente possível, o compilador pode realizar modificações próprias no código-fonte criado pelo programador. Essas otimizações envolvem diversas estratégias como o desenrolamento de laços (*loop unwrapping*), substituição de partes do código por algoritmos mais eficientes, inclusão explícita de subprogramas (*inlining*), entre outros. Nesses processos, eventualmente variáveis que foram declaradas e usadas pelo programador são eliminadas ou movidas para outras regiões da memória. Se um ponteiro for previamente associado a uma dessas variáveis modificadas pelo processo de otimização, a referência armazenada no mesmo se torna inválida e o seu uso em expressões irá provavelmente gerar resultados incorretos. Este é um exemplo de criação de um *ponteiro pendente* (*dangling pointer*). Por esta razão, se o objeto estático for declarado como um possível alvo de ponteiros, o compilador será informado que o mesmo não pode ser eliminado ou ter seu endereçamento alterado.

7.2.1 DECLARAÇÕES BÁSICAS DE PONTEIROS E ALVOS

Ponteiros e alvos de ponteiros podem ser respectivamente declarados empregando as palavras-chave `POINTER` e `TARGET` como atributos ou em declarações próprias. Por exemplo, o ponteiro `ptr` da figura 7.1 pode ser declarado um ponteiro inteiro usando o atributo:

```
INTEGER, POINTER :: PTR
```

ou via as declarações:

```
INTEGER :: PTR
POINTER :: PTR
```

Já a variável `var` da figura 7.1 deve ter sido declarada

```
INTEGER, TARGET :: VAR= 100 ! A atribuição do valor pode ocorrer posteriormente
```

ou

```
INTEGER :: VAR
TARGET :: VAR
```

Ponteiros somente podem ser *associados* a alvos que tenham o *atributo de alvo*, adquirido de algumas das duas formas anteriores. Um mesmo objeto não pode ter ambos os atributos de ponteiro e alvo.

Ponteiros também podem ser criados com objetos compostos e/ou matriciais. Por exemplo,

```
REAL, POINTER, DIMENSION(:) :: X, Y
COMPLEX, POINTER, DIMENSION(:,:) :: Z
```

declara dois ponteiros para vetores do tipo real (`X` e `Y`) e um ponteiro para uma matriz complexa de posto 2 (`Z`). Da mesma forma como acontece durante a declaração de matrizes alocáveis, ponteiros para matrizes devem ser declarados como *matrizes de forma deferida*, isto é, o posto da matriz é determinado, mas não sua forma, o que fica evidenciado pela presença dos caracteres ":" no atributo/declaração `DIMENSION`. Os exemplos abaixo ilustram alvos adequados para os ponteiros definidos acima:

```
REAL, TARGET :: A(100), B(200)         ! Para X e Y
COMPLEX, TARGET, DIMENSION(10,20) :: C ! Para Z
```

Também é possível definir ponteiros para e alvos de tipos derivados. Por exemplo, dado o tipo `aluno_t` definido no programa 3.6, pode-se definir:

```
TYPE(ALUNO_T), POINTER, DIMENSION(:) :: ALUNOS_P
TYPE(ALUNO_T), TARGET, ALLOCATABLE, DIMENSION(:) :: ALUNOS_V
```

Observa-se que `ALUNOS_V` possui tanto o atributo `TARGET` quanto `ALLOCATABLE`, o que permite que o tamanho do vetor seja determinado em tempo de execução. Uma vez que `ALUNOS_V` foi declarado com o atributo `TARGET`, todos os seus componentes automaticamente possuem o mesmo atributo e podem ser individualmente associados a outros ponteiros. Além disso, componentes individuais de tipos derivados podem ter atributos de ponteiro ou alvo, mesmo que a estrutura completa não tenha esse atributo.

A forma geral das declarações de ponteiros e alvos será apresentada posteriormente.

7.2.2 Atribuições de ponteiros e seus usos em expressões

Esta seção explora como ponteiros podem ser associados a alvos e como podem ser subsequentemente empregados em expressões e atribuições.

7.2.2.1 Status de associação

O **status de associação de um ponteiro** indica se o mesmo está ou não apontando para um alvo. Este status pode ser:

Indefinido (*undefined*). Status assumido pelo ponteiro no momento da declaração, como nos exemplos acima.

Definido (*defined*). Status adquirido quando o ponteiro foi associado com algum objeto com atributo de alvo.

Desassociado (*disassociated*). Quando um ponteiro previamente associado perde este status por algum dos mecanismos apresentados abaixo e não é associado com nenhum outro alvo.

Mesmo que um ponteiro seja associado a um alvo (tornando-se assim definido), o alvo *per se* pode ter o status de definido ou indefinido.

Um ponteiro somente pode ser empregado se ele estiver associado a um alvo. Uma ocorrência de uso de um ponteiro indefinido nem sempre pode ser detectada pelo compilador e isto poderá acarretar em um erro durante a execução do programa. Existem circunstâncias nas quais um ponteiro previamente associado pode ser tornar indefinido. Um exemplo disso ocorre com o uso da rotina `MOVE_ALLOC` (seção 7.1.2.1); como foi mencionado, se o objeto identificado pelo qualificador `TO=` tiver o atributo de alvo, então qualquer ponteiro que esteja previamente associado com o objeto em `FROM=` terá sua associação transferida junto com o objeto de dados. Em caso contrário, o ponteiro se torna indefinido.

Para minimizar a possibilidade de uso de um ponteiro indefinido, pode-se empregar a função intrínseca `ASSOCIATED`,[7] a qual retorna um valor lógico indicando o status de associação do ponteiro. Existem duas maneiras de empregar a função `ASSOCIATED`: para testar a associação do ponteiro a *qualquer* alvo ou a um alvo em particular. Por exemplo, para testar se o ponteiro `ptr` está associado a algum alvo, emprega-se

[7] Ver seção 8.2.

```
STATUS= ASSOCIATED(PTR)
```

sendo STATUS uma variável lógica. Se ptr já estiver associado a var, o resultado será STATUS= .TRUE., caso contrário será .FALSE.. Por outro lado, para testar se ptr está associado ao inteiro var, usa-se

```
STATUS= ASSOCIATED(PTR, VAR)
```

se ptr estiver associado a algum outro alvo, o resultado será STATUS= .FALSE..

Contudo, a função ASSOCIATED somente pode testar dois status: definido ou desassociado. Se o ponteiro estiver no estado indefinido, o resultado será ambíguo (pois o padrão não determina qual deve ser o resultado neste caso), o que poderá levar a erros de processamento. Para evitar essa possibilidade, existem dois recursos:

1. Uso do função intrínseca NULL().[8] Esta função confere o status de desassociado a um ponteiro, independente de seu status inicial. O uso deste recurso nos exemplos acima é:

   ```
   INTEGER, POINTER :: PTR => NULL()
   REAL, POINTER, DIMENSION(:) :: X => NULL(), Y => NULL()
   COMPLEX, POINTER, DIMENSION(:,:) :: Z => NULL()
   TYPE(ALUNO_T), POINTER, DIMENSION(:) :: ALUNOS_P => NULL()
   ```

2. Uso do comando NULLIFY. Este comando possui a sintaxe

   ```
   NULLIFY(<lista-obj-pointer>)
   ```

   A execução deste comando confere a todos os objetos na <lista-obj-pointer> o status de desassociado. Esses objetos devem ter o atributo de ponteiro e não podem ser interdependentes, para que o processador possa possa realizar as desassociações em qualquer ordem. Como se trata de um comando, a instrução NULLIFY deve vir após todas as declarações. Para os exemplos acima, o uso pode ser:

   ```
   INTEGER, POINTER :: PTR
   REAL, POINTER, DIMENSION(:) :: X, Y
   COMPLEX, POINTER, DIMENSION(:,:) :: Z
   TYPE(ALUNO_T), POINTER, DIMENSION(:) :: ALUNOS_P
   ⋮ ! Outras declarações
   NULLIFY(PTR, X, Y, Z, ALUNOS_P)
   ```

Sugestões de uso & estilo para programação

Recomenda-se sempre realizar a desassociação de todos os ponteiros, via NULL() ou NULLIFY(...), antes de qualquer uso dos mesmos. Esta prática possibilita a realização de testes de status e evita a ocorrência de erros devidos ao uso de ponteiros indefinidos.

7.2.2.2 ASSOCIAÇÃO DE PONTEIROS COM OBJETOS DE DADOS

Para que um ponteiro possa ser empregado, ele primeiro precisa primeiro estar associado a um objeto de dados, ou seja, deve conter, no mínimo, o endereço de memória do objeto. Isto pode ocorrer de duas maneiras: ou pela *atribuição de ponteiro* (*pointer assignment*) a um objeto que tenha atributos de alvo ou ponteiro ou pela alocação dinâmica de um novo espaço na memória.

ATRIBUIÇÃO DE PONTEIRO. O primeiro processo (*atribuição de ponteiro*) é realizada com o operador "=>" pelas instruções

```
<ponteiro> => <alvo> ou <ponteiro 1> => <ponteiro 2>
```

A atribuição pode ocorrer na inicialização do ponteiro:

[8] Ver seção 8.16.

```
REAL, TARGET :: X        ! Pode ou não ser inicializado.
REAL, POINTER :: PX => X ! Ponteiro PX inicializado com o endereço de X
```

O ponteiro PX automaticamente adquire o status de *definido* e é inicializado com o endereço de X.

Ou a atribuição pode ocorrer ao longo da seção de instruções do código, onde diversas possibilidades podem ocorrer. O programa 7.3 mostra diversos processos de atribuições entre ponteiros.

Listagem 7.3: Ilustra diversos casos de associação de ponteiros para objetos escalares.

```
1  program tes_pointer
2  implicit none
3  real, target :: x1= 10.0, x2= -17.0
4  real, pointer :: px1 => null(), px2 => null()

6  print'(2(a,g0))', 'px1 associado? ', associated(px1), &
7                   '  px2 associado? ', associated(px2)
8  px1 => x1 ! px1 está associado a x1.
9  print'(3(a,g0))', 'px1 associado? ', associated(px1), &
10                  '  px1= ', px1, '  x1= ', x1
11 px2 => x2 ! px2 está associado a x2.
12 print'(3(a,g0))', 'px2 associado? ', associated(px2), &
13                  '  px2= ', px2, '  x2= ', x2

15 ! Alterações nas associações de ponteiros.
16 px2 => px1
17 print'(/,2(a,g0))', 'px1= ', px1, '  px2= ', px2
18 px1 => x2
19 print'(2(a,g0))', 'px1= ', px1, '  px2= ', px2
20 px2 => px1
21 px1 => null() ! Ou: nullify(px1)
22 print'(2(a,g0))', 'px1 associado? ', associated(px1), '  px2= ', px2
23 end program tes_pointer
```

Neste programa, os ponteiros px1 e px2 são inicializados com o status de desassociados. Em seguida, as instruções px1 => x1 e px2 => x2 realiza as atribuições indicadas nas linhas 8 e 11. Estas atribuições estão representadas na figura 7.2. Isto significa que px1 adquiriu o endereço de x1, enquanto que px2 adquiriu o endereço de x2.

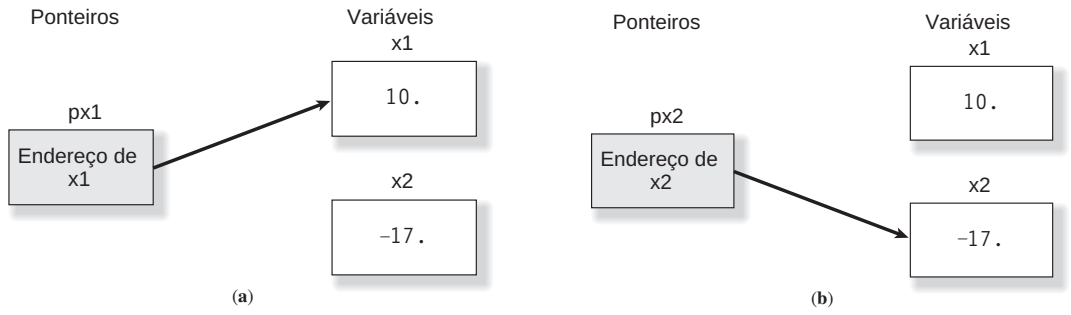


Figura 7.2: atribuições iniciais entre as variáveis *x1* e *x2* com os ponteiros *px1* e *px2* no programa 7.3. No painel (a), *px1* adquire o endereço de *x1* (linha 8), enquanto que no painel (b), *px2* adquire o endereço de *x2* (linha 11).

Posteriormente, na linha 16, ocorre uma atribuição do tipo <ponteiro 1> => <ponteiro 2>. Neste tipo de atribuição, o que ocorre é que o <ponteiro 1> adquire o valor do <ponteiro 2>, ou seja, o endereço do alvo do <ponteiro 2> se este estiver definido ou o seu status em caso contrário. Após a execução desta instrução, *ambos os ponteiros apontam diretamente e de forma independente ao mesmo alvo.* Na linha 16, portanto, o que ocorre é que px2 copia o endereço armazenado em px1, o qual é o endereço de x1; ambos os ponteiros apontam agora para o

mesmo alvo, como está ilustrado no painel (a) da figura 7.3. Em seguida, na linha 18, `px1` passa a apontar para `x2`, mas a associação de `px2` não foi alterada (continua associado a `x1`). Esta situação está ilustrada no painel (b) da figura 7.3.

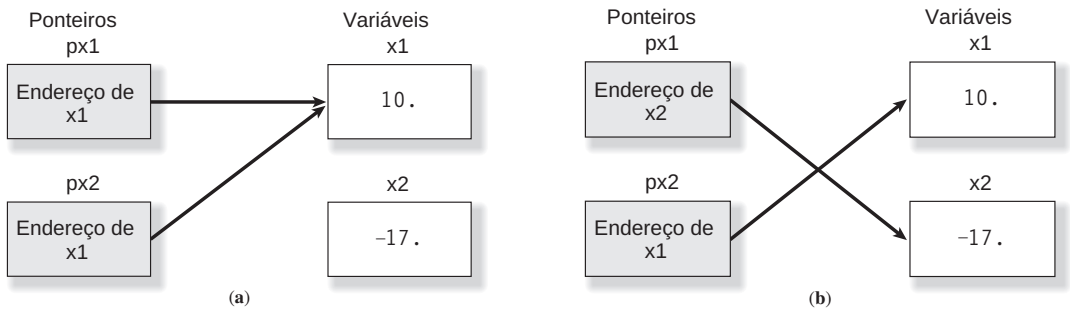


Figura 7.3: atribuições de ponteiros no programa 7.3. Painel (a): *px2* passa a apontar para *x1* (linha 16). Painel (b): *px1* passa a apontar para *x2* (linha 18).

Finalmente, nas linhas 20 e 21, primeiro `px2` adquire o endereço em `px1` (o endereço de `x2`) e logo em seguida `px1` é desassociado. Como os ponteiros retêm os endereços dos alvos de forma independente, a desassociação de `px1` não altera o estado e a associação de `px2`. No programa observa-se também que os valores de `x1` e `x2` podem ser acessados tanto de forma direta quanto de forma indireta, através dos ponteiros associados às variáveis. Expressões e atribuições envolvendo ponteiros serão discutidas na seção 7.2.2.3.

Quando o ponteiro é de um tipo composto, como matrizes ou tipos derivados, além do endereço do primeiro objeto contido no alvo, um ponteiro em Fortran também armazena informações a respeito da estrutura do objeto composto. Estas informações a respeito do alvo que ficam armazenadas no ponteiro são denominadas em alguns textos como *descritores de ponteiros.*

Quando o alvo é uma matriz, um ponteiro pode adquirir o descritor correspondente à matriz completa ou a uma seção da mesma. A flexibilidade do mecanismo de atribuição de ponteiros para matrizes no Fortran é exemplificada no programa 7.4, no qual estão definidos uma matriz real de posto 2 `table` e um vetor inteiro `sunspot_number`, o qual contém o número de manchas solares observadas entre os anos de 1700 e 2018. O programa define também um ponteiro para matriz real de posto 2 `window` e dois ponteiros para vetores inteiros `ps1` e `psn2`.

Listagem 7.4: Exemplos de associação de ponteiros a matrizes.

```
1  program tes_array_pointer_assignment
2  implicit none
3  integer :: m= 50, n= 130, p= 30, q= 85
4  integer, dimension(1700:2018), target :: sunspot_number
5  real, dimension(200,100), target :: table
6  integer, dimension(:), pointer :: psn1, psn2
7  real, dimension(:,:), pointer :: window

9  ! Associações entre table e window.
10 print'(2(a,*(g0,x)),a)', 'Limites de table:  Inferiores: [ ', &
11                         lbound(table), ']  Superiores: [', ubound(table), ']'
12 window => table
13 print'(2(a,*(g0,x)),a)', 'Limites de window (1): Inferiores: [ ', &
14                         lbound(window), ']  Superiores: [', ubound(window), ']'
15 window => table(m:n, p:q)
16 print'(2(a,*(g0,x)),a)', 'Limites de window (2): Inferiores: [ ', &
17                         lbound(window), ']  Superiores: [', ubound(window), ']'

19 ! Associações entre sunspot_number e psn1 e psn2.
20 print'(/,2(a,*(g0,x)))', 'Limites de sunspot_number: Inferior: ', &
21                           lbound(sunspot_number), '  Superior: ',  &
22                           ubound(sunspot_number)
23 psn1 => sunspot_number
24 print'(2(a,*(g0,x)))', 'Limites de psn1: Inferior: ', lbound(psn1), &
```

```
25                                          '  Superior: ', ubound(psn1)
26 psn2 => sunspot_number(1901:2000)
27 print'(2(a,*(g0,x)))', 'Limites de psn2 (1): Inferior: ', lbound(psn2), &
28                                             '  Superior: ', ubound(psn2)
29 psn2(1901:) => sunspot_number(1901:2000)
30 print'(2(a,*(g0,x)))', 'Limites de psn2 (2): Inferior: ', lbound(psn2), &
31                                             '  Superior: ', ubound(psn2)
32
33 psn2 => sunspot_number(::50)
34 print'(2(a,*(g0,x)))', 'Limites de psn2 (2): Inferior: ', lbound(psn2), &
35                                             '  Superior: ', ubound(psn2)
36 end program tes_array_pointer_assignment
```

O primeiro conjunto de instruções no programa 7.4 (entre as linhas 10 e 17) ilustra diversas atribuições entre as matrizes `table` e `window`. Primeiramente, as funções intrínsecas `LBOUND` e `UBOUND` (seção 8.13) reafirmam os limites das dimensões da matriz `table`. Em seguida, na linha 12 o ponteiro `window` é associado à matriz `table` completa, obtendo assim todos os descritores desta última (endereço inicial e informações da geometria), o que fica demonstrado na impressão dos limites de `window`, realizada na linha 13. Porém, na linha 15 o ponteiro `window` se torna associado a uma seção da matriz `table`, com uma geometria determinada pelas variáveis inteiras `m`, `n`, `p` e `q`. Neste caso, a sintaxe da atribuição segue as mesmas regras para seções de matrizes discutidas na seção 6.3. Na linha 16 a geometria resultante de `window` é impressa na saída padrão, onde se observa que o ponteiro armazenou os descritores para os elementos em 81 linhas (entre as linhas 50 e 130) e 56 colunas (entre colunas 30 e 85) de `table`, exatamente como é determinado pela álgebra de seções de matrizes. Nota-se contudo que os limites de `window` partem de 1 em todas as dimensões, o que é o comportamento esperado na definição das funções `LBOUND` e `UBOUND`. Porém, caso seja necessário ou adequado, é possível alterar também os limites registrados no ponteiro. Este mecanismo será ilustrado a seguir, com os ponteiros `psn1` e `psn2`.

Entre as linhas 20 e 35 ocorrem diversas atribuições entre o vetor `sunspot_number` e os ponteiros para vetores. Na linha 23, `psn1` adquire os descritores do vetor completo. Nota-se que os limites de `sunspot_number` são automaticamente transferidos para `psn1`. Já na linha 26, `psn2` passa a ser um atalho para o número de manchas solares observadas somente durante o século XX. Nota-se que o limite inferior de `psn2` é 1 e o superior é 100, de maneira análoga ao que ocorreu com `window` na linha 15. Porém, na linha 29 a mesma atribuição é realizada, mas agora a sintaxe `psn2(1901:)` determina que a atribuição dos limites de `psn2` ocorra com o limite interior igual a 1901. Ou seja, as regras de seções de matrizes determinadas por tripletos de subscritos podem ser empregadas também em ambos os lados de uma atribuição de ponteiros para matrizes, desde que sejam realizadas de uma forma consistente. Finalmente, na linha 33, `psn2` passa a apontar a uma seção de `sunspot_number` que varre os índices do limite inferior ao superior, porém com um passo igual a 50. Neste caso, `psn2` passou a ser um atalho para o número de manchas solares observadas de 1700 a 2018 a cada 50 anos, totalizando 7 registros.

Os mecanismos de atribuições de ponteiros para matrizes exemplificados acima também podem ser empregados entre objetos de diferentes postos. Neste conjunto de instruções:

```
INTEGER :: N
REAL, DIMENSION(:), ALLOCATABLE, TARGET :: VETOR_BASE
REAL, POINTER :: MATRIZ(:,:), DIAGONAL(:)
ALLOCATE(VETOR_BASE(N*N))
MATRIZ(1:N, 1:N) => VETOR_BASE
DIAGONAL => VETOR_BASE(::N+1)
```

o `VETOR_BASE`, após a alocação de memória, passará a ocupar um intervalo contíguo de espaços de memória contendo N^2 objetos reais. Após a sua atribuição, o ponteiro `MATRIZ` conterá os descritores dos elementos de `VETOR_BASE` na forma de uma matriz $N \times N$, sendo que os descritores serão armazenados em `MATRIZ` na ordem dos elementos de matriz (seção 6.6.2), de forma que esses elementos podem ser referenciados via `MATRIZ` de uma maneira contígua. Finalmente, o vetor `DIAGONAL` é um atalho para os elementos da diagonal de `MATRIZ`.

Quando os objetos envolvidos são tipos derivados, a variadade de diferentes situações é ainda maior. Um objeto de um tipo derivado pode ser declarado com o atributo `TARGET`, em cuja situação todos os seus componentes automaticamente têm o mesmo atributo. Ponteiros desse tipo

derivado podem se associar a uma estrutura-alvo por completo ou ponteiros dos tipos dos componentes da estrutura podem se associar com estes últimos de forma individual. O exemplo abaixo ilustra essa situação:

```
INTEGER :: M, N, P
TYPE :: F3V_T
   REAL :: U, DU(3), D2U(3,3)
END TYPE F3V_T
TYPE(F3V_T), DIMENSION(:,:,:), ALLOCATABLE, TARGET :: MU
REAL, POINTER, DIMENSION(:,:,:) :: PMU
ALLOCATE(MU(M, N, P))
PMU => MU%U
```

Neste exemplo, `F3V_T` é um tipo derivado que armazena o valor de uma função de três variáveis, suas três derivadas parciais de primeira ordem e suas derivadas segundas. A matriz `MU` armazena esses valores em uma grade com `M` $\times$ `N` $\times$ `P` pontos, ao passo que o ponteiro `PMU` possibilita um acesso rápido somente aos valores da função nos pontos de grade. A geometria de `PMU` é igual à geometria de `MU`.

Um componente de um tipo derivado também pode ser declarado de uma forma recursiva como um ponteiro para o mesmo tipo. Toda essa flexibilização permite a definição de novas estruturas complexas de dados, algumas das quais serão discutidas na seção 7.4. Em particular, nessa seção será discutida a construção de uma *lista encadeada*, formada a partir da definição de um tipo derivado semelhante a:

```
TYPE :: ENTRY
   INTEGER :: INDEX
   REAL :: VALOR= 2.0
   REAL, DIMENSION(:), POINTER :: VETOR
   TYPE(ENTRY), POINTER :: NEXT => NULL()
END TYPE ENTRY
```

Antes de discutir o segundo método de associação de ponteiros, é importante mais uma vez frisar que um ponteiro definido adquire os descritores do objeto de dados no momento da associação. Se o ponteiro estiver associado a um objeto alocável, pode ocorrer que posteriormente o estado de alocação do objeto seja alterado via o mecanismo de realocação automática (seção 7.1.2.2). Neste caso, a associação do ponteiro deve ser renovada, para que os novos descritores sejam registrados. No exemplo abaixo, se `V` é um vetor alocável e `PV` um ponteiro a um vetor, o último passo na sequência de instruções

```
PV => V     ! PV adquire os descritores de V
V= [ V, 1 ] ! A extensão de V foi alterada
PV => V     ! PV adquire os novos descritores de V
```

se faz necessário porque a forma de `V` foi alterada via alocação automática. Se houver outros ponteiros associados com `V` ou com uma seção de `V`, estes também deverão ter suas associações renovadas.

Sugestões de uso & estilo para programação

Se ponteiros estão associados a objetos alocáveis, é importante que alterações nos estados de alocação desses objetos sejam realizadas com a rotina `MOVE_ALLOC`, pois assim todos os ponteiros associados a esses objetos terão os descritores automaticamente atualizados.

ALOCAÇÃO DE MEMÓRIA PARA PONTEIRO. O segundo tipo de processo de atribuição de ponteiro ocorre pela alocação dinâmica de um novo espaço de memória, que não estava previamente ocupado por algum objeto de dados. Este procedimento ocorre por meio da instrução `ALLOCATE`, como é exemplificado nas instruções

```
INTEGER :: M, N
REAL, POINTER :: PX1, PV(:), PM(:,:)
ALLOCATE(PX1, PV(10), PM(M, N))
```

A execução da instrução ALLOCATE acima reserva automaticamente espaços de memória e cria objetos de dados sem nomes (*i. e.*, anônimos), com os tipos, espécies e geometrias adequados. Os ponteiros passam a ter status de definidos e os descritores dos correspondentes espaços de memória são atribuídos aos ponteiros, sem haver a necessidade de existência prévia de objetos com atributos de alvo. Uma vez que esses espaços de memória são anônimos, eles somente podem ser acessados pelos ponteiros. Estes espaços de memória podem ser posteriormente liberados por meio da instrução DEALLOCATE.

A forma geral dos comandos ALLOCATE e DEALLOCATE para ponteiros é igual às suas formas para matrizes alocáveis, apresentadas na seção 7.1.1. No caso particular de ponteiros, se o procedimento de alocação de memória falhar, o ponteiro retém o seu status de associação. As ações executadas quando as palavras-chave STAT= e ERRMSG= estão presentes são as mesmas que ocorrem no caso de matrizes alocáveis.

Um ponteiro pode se associar a um novo alvo com qualquer um dos dois métodos discutidos, mesmo se este já estava previamente associado com um alvo. Neste caso a associação prévia é removida. Contudo, se o alvo prévio consistir em uma área de memória reservada para o ponteiro por um comando ALLOCATE, a mesma se tornará inacessível, exceto se um outro ponteiro foi também associado a ela.

Quando o espaço de memória associado a um ponteiro via o comando ALLOCATE não for mais necessário, este espaço pode ser liberado com a instrução DEALLOCATE. Se a operação for bem sucedida, o status do ponteiro muda automaticamente para desassociado. Se a operação falhar, o ponteiro mantém o seu status de associação; contudo, se a cláusula STAT= não for empregada neste caso, a execução do programa será interrompida. É importante ressaltar que o comando DEALLOCATE não deve ser empregado se o ponteiro tiver sido associado a um objeto de dados via atribuição (com o operador =>), como pode ocorrer com objetos estáticos ou matrizes alocáveis.

Exemplos de uso dos comandos ALLOCATE e DEALLOCATE com ponteiros para a construção de estruturas de dados mais complexas serão apresentados na seção 7.4.

7.2.2.3 Uso de ponteiros em expressões ou atribuições

Um ponteiro associado com um objeto de dados contém o descritor desse objeto, isto é, o endereço do objeto na memória e informações sobre a geometria e conteúdo do mesmo, caso este seja um objeto composto. Contudo, um dos principais objetivos de um ponteiro consiste em usar essa informação como um atalho para o *valor* do objeto de dados. Para tanto, deve existir um mecanismo pelo qual esse valor se torne acessível a partir do ponteiro. O processo de acesso ao valor do objeto é denominado a **dereferenciação** (*dereferencing*) de um ponteiro e diferentes linguagens implementam esse mecanismo de diferentes maneiras. No Fortran, a dereferenciação é automática; isto é, o uso do nome do ponteiro em expressões ou atribuições intrínsecas (não em atribuições de ponteiro) automaticamente torna acessível o valor do objeto com o qual o ponteiro está associado. A dereferenciação automática já foi empregada nas instruções PRINT dos programas 7.3 e 7.4 quando os nomes dos ponteiros foram empregados para imprimir os valores das variáveis na tela.

Existem, portanto, duas maneiras distintas de transferir informações entre ponteiros via atribuições. Sendo p1 e p2 ponteiros, a *atribuição de ponteiro* p1 => p2 transfere para p1 o status de associação de p2 e o descritor de um eventual objeto de dados associado a p2; assim, ambos p1 e p2 servem de atalho ao mesmo objeto de dados, cujo valor não foi alterado. Contudo a *atribuição intrínseca* p1 = p2 não transfere o descritor de p2 para p1; o que ocorre é a cópia do valor do objeto associado a p2 para o objeto associado a p1, alterando assim o valor deste último objeto, enquanto que as associações dos ponteiros permanecem inalteradas. As regras usuais de expressões e atribuições discutidas no capítulo 4 continuam válidas neste caso.

O programa 7.5 exemplifica diversas instâncias de emprego de ponteiros em expressões e atribuições envolvendo tanto objetos escalares.

Listagem 7.5: Uso de ponteiros com objetos escalares.

```
1 program tes_pointer_att_scalar
2 implicit none
3 real, target :: x1= 11.0, x2= -25.5, x3
4 real, pointer :: p1 => null(), p2 => null(), p3 => null()
```

```
6  !Realiza associações iniciais: p1 aponta para x1, etc.
7  p1 => x1 ; p2 => x2 ; p3 => x3
8  p3= p1 + p2 ! O mesmo que x3 = x1 + x2
9  print'(a,g0)', 'Valor de x3:           ', x3
10 print'(a,g0)', 'Valor de x3 (via p3): ', p3

12 p2 => p1     ! Agora p2 aponta para x1
13 p3= p1 + p2 ! O mesmo que x3 = x1 + x1
14 print'(/,a,g0)', 'Novo valor de x3: ', x3

16 p3 = p1 ! Atribuição intrínseca: o mesmo que x3 = x1
17 print'(/,2(a,g0))', 'Valores de: x3= ', x3, '  p3= ', p3

19 !Realiza novas associações
20 x3 = 2*x1 + x2 ! p3 ainda está associado a x3
21 p2 => x2 ! Retorna à associação inicial
22 p3 => p2 ! p3 aponta agora para x2.  Ocorreu cópia de descritores, não de dados
23 print'(/,a)', '---> Cópia de descritores: p3 => p2, x3 /= x2:'
24 print'(3(a,g0))', 'Valores nas variáveis:   x1= ', x1, '  x2= ', x2, '  x3= ', x3
25 print'(3(a,g0))', 'Valores pelos ponteiros: p1= ', p1, '  p2= ', p2, '  p3= ', p3

27 !Realiza agora atribuição intrínseca
28 p3 => x3 ! Retorna à associação inicial
29 p3 = p2  ! Houve cópia de dados, não de descritores
30 print'(/,a)', '---> Cópia de dados: p3 = p2 -> x3 = x2:'
31 print'(3(a,g0))', 'Valores nas variáveis:   x1= ', x1, '  x2= ', x2, '  x3= ', x3
32 print'(3(a,g0))', 'Valores pelos ponteiros: p1= ', p1, '  p2= ', p2, '  p3= ', p3

34 end program tes_pointer_att_scalar
```

A execução deste programa gerou os seguintes resultados:

```
user@machine|dir>./a.out
Valor de x3:           -14.5000000
Valor de x3 (via p3): -14.5000000

Novo valor de x3: 22.0000000

Valores de: x3= 11.0000000  p3= 11.0000000

---> Cópia de descritores: p3 => p2, x3 /= x2:
Valores nas variáveis:   x1= 11.0000000  x2= -25.5000000  x3= -3.50000000
Valores pelos ponteiros: p1= 11.0000000  p2= -25.5000000  p3= -25.5000000

---> Cópia de dados: p3 = p2 -> x3 = x2:
Valores nas variáveis:   x1= 11.0000000  x2= -25.5000000  x3= -25.5000000
Valores pelos ponteiros: p1= 11.0000000  p2= -25.5000000  p3= -25.5000000
```

Os resultados que mostram somente cópias de descritores, sem alterações nos dados das variáveis foram gerados pelas instruções entre as linhas 20 e 25 do programa, ao passo que os resultados mostrando cópia de dados mas não de descritores foram gerados nas linhas 28 – 32.

Já o programa 7.6 exemplifica o emprego de ponteiros em expressões e atribuições envolvendo vetores:

Listagem 7.6: Uso de ponteiro com matrizes

```
1  program tes_pointer_att_vector
2  implicit none
3  integer :: i, n
4  real, target,  dimension(:), allocatable :: v1, v2
5  real, pointer, dimension(:) :: pv1 => null(), pv2 => null()

7  write(*, '(a)', advance= 'no') 'Número de elementos no vetor v1: ' ; read(*,*)n
8  allocate(v1(n))
```

```
9  call random_number(v1)
10 print'(a)', '---> Vetor v1 original:'
11 print'(5(g0,x))', v1
12 v2= v1 ! v2 é uma cópia de segurança de v1

14 !Primeiras associações:
15 pv1 => v1        ! pv1 aponta para v1 completo
16 pv2 => v1(2::3) ! pv2 aponta para uma seção de v1
17 print'(/,a)', '---> Dereferências do ponteiro pv1 (todo o vetor v1):'
18 print'(5(g0,x))', pv1
19 print'(/,a)', '---> Dereferências do ponteiro pv2 (seção de v1: v1(2), v1(5), etc):'
20 print'(5(g0,x))', pv2

22 !Cópia de descritores:
23 pv1 => pv2 ! pv1 adquire os descritores de pv2
24 print'(/,a)', '---> Novas dereferências do ponteiro pv1 (1):'
25 print'(5(g0,x))', pv1
26 print'(a)', 'Vetor v1 não é alterado, pois houve cópias de descritores: pv1 => pv2'

28 !Cópia de dados:
29 pv1 => v1 ! Retorna à associação inicial
30 pv1(:size(pv2)) = pv2 ! Agora há cópia de dados
31 print'(/,a)', '---> Novas dereferências do ponteiro pv1 (2) (cópias de dados):'
32 print'(5(g0,x))', pv1
33 print'(/,a)', '---> Novos valores do vetor v1 (1) (houve cópia de dados):'
34 print'(5(g0,x))', v1

36 v1= v2 !Usa v2 para retornar ao vetor v1 original
37 print'(/,a)', '---> Vetor v1 original:'
38 print'(5(g0,x))', v1

40 pv1(3:2 + size(pv2)) = pv2 ! Nova cópia de dados
41 print'(/,a)', '---> Novos valores do vetor v1 (2) (houve cópia de dados):'
42 print'(5(g0,x))', v1
43 end program tes_pointer_att_vector
```

Uma determinada execução deste programa gerou os resultados:

```
user@machine|dir>./a.out
Número de elementos no vetor v1: 10
---> Vetor v1 original:
0.649878383E-01 0.865801036 0.596078634 0.159959137 0.267309129
0.200994670 0.360674262E-01 0.353242338 0.916801214 0.403821468

---> Dereferências do ponteiro pv1 (todo o vetor v1):
0.649878383E-01 0.865801036 0.596078634 0.159959137 0.267309129
0.200994670 0.360674262E-01 0.353242338 0.916801214 0.403821468

---> Dereferências do ponteiro pv2 (seção de v1: v1(2), v1(5), etc):
0.865801036 0.267309129 0.353242338

---> Novas dereferências do ponteiro pv1 (1):
0.865801036 0.267309129 0.353242338
Vetor v1 não é alterado, pois houve cópias de descritores: pv1 => pv2

---> Novas dereferências do ponteiro pv1 (2) (cópias de dados):
0.865801036 0.267309129 0.353242338 0.159959137 0.267309129
0.200994670 0.360674262E-01 0.353242338 0.916801214 0.403821468

---> Novos valores do vetor v1 (1) (houve cópia de dados):
0.865801036 0.267309129 0.353242338 0.159959137 0.267309129
0.200994670 0.360674262E-01 0.353242338 0.916801214 0.403821468

---> Vetor v1 original:
0.649878383E-01 0.865801036 0.596078634 0.159959137 0.267309129
```

```
0.200994670  0.360674262E-01  0.353242338  0.916801214  0.403821468

---> Novos valores do vetor v1 (2) (houve cópia de dados):
0.649878383E-01  0.865801036  0.865801036  0.267309129  0.353242338
0.200994670  0.360674262E-01  0.353242338  0.916801214  0.403821468
```

O vetor `v1` foi alocado com 10 elementos escolhidos aleatoriamente. No intervalo de linhas 15 – 20 do programa, o ponteiro `pv1` é associado com o vetor inteiro, enquanto que o ponteiro `pv2` se torna associado à seção `[ v(2), v(5), v(8) ]`. Em seguida, nas linhas 23 – 26 o ponteiro `pv1` adquire os descritores de `pv2`, sem que ocorra alteração nos valores do vetor.

As expressões seguintes ilustram cópias de dados empregando ponteiros. Nas linhas 29 – 34 os três primeiros elementos de `v1` são alterados pela atribuição intrínseca `pv1(1:3) = pv2` ocorrida na linha 30. Finalmente, nas linhas 36 – 42 o vetor `v1` original é novamente alterado pela atribuição `pv1(3:5) = pv2`, a qual alterou os valores de `v1(3)` – `v1(5)`.

Para certas aplicações, o uso de ponteiros pode ter um impacto significativo na eficiência de um programa. Uma situação típica ocorre quando há necessidade de manipulações de matrizes com um número grande de elementos. Suponha que seja necessário trocar os nomes de duas matrizes `m1` e `m2`, ambas com dimensões $10^4 \times 10^4$. O código para realizar essa manipulação simples é:

```
1  program tes_swap_mat1
2  implicit none
3  integer, parameter :: nlc= 10000
4  real :: start_time, end_time
5  real, dimension(nlc, nlc) :: m1, m2, te

7  call random_number(m1)
8  call random_number(m2)

10 call cpu_time(start_time)
11 te= m1
12 m1= m2
13 m2= te
14 call cpu_time(end_time)
15 print'(a, g0)', 'Tempo de CPU (s): ', end_time - start_time

17 end program tes_swap_mat1
```

A troca propriamente dita ocorre nas linhas 11 – 13 e envolve o uso da matriz temporária `te`. O tempo de CPU necessário para realizar esta manipulação foi medido com a rotina `CPU_TIME` (seção 8.17.2), a qual adquire o valor do tempo (em segundos) do relógio do processador. O tempo total de processamento da troca das matrizes é então dado pela diferença `end_time - start_time`. Compilando este programa sem otimização, o tempo de processamento foi de aproximadamente 0,3 segundos, uma vez que ocorrem três processos de cópias de matrizes com dimensões $10^4 \times 10^4$.

O mesmo tipo de procedimento pode ser realizado de maneira muito mais rápida se forem realizadas cópias dos descritores das matrizes, ao invés dos dados. O programa abaixo implementa o mesmo procedimento com ponteiros:

```
1  program tes_swap_mat2
2  implicit none
3  integer, parameter :: nlc= 10000
4  real :: start_time, end_time
5  real, target,  dimension(nlc, nlc) :: m1, m2
6  real, pointer, dimension(:,:)      :: p1, p2, te

8  call random_number(m1)
9  call random_number(m2)

11 call cpu_time(start_time)
12 p1 => m1 ! p1 associado a m1
```

```
13 p2 => m2 ! p2 associado a m2
14 te => p1 ! te associado a m1
15 p1 => p2 ! p1 agora associado a m2
16 p2 => te ! p2 agora associado a m1
17 call cpu_time(end_time)
18 print'(a, g0)', 'Tempo de CPU (s): ', end_time - start_time

20 end program tes_swap_mat2
```

Agora, o tempo de processamento foi reduzido para cerca de $10^{-6} = 1,0\,\mu s$ apenas. Na verdade, a maior parte do tempo de execução do programa ocorre nas atribuições de valores às matrizes pela rotina `RANDOM_NUMBER`.

7.3 Objetos alocáveis

O conceito de um *objeto alocável* é uma extensão do conceito de uma matriz alocável (seção 7.1) para qualquer tipo de objeto, escalar ou composto. Observa-se que um objeto alocável, da mesma forma como ocorre com as matrizes alocáveis ou com ponteiros, são objetos que são declarados, mas para os quais não são reservados espaços de memória no momento em que o código é executado da primeira vez. Ao invés disso, o espaço de memória é criado dinâmicamente no momento em que se faz necessário e depois, quando não for mais necessário, esse espaço de memória pode ser disponibilizado de volta ao sistema operacional. Contudo, de forma distinta ao que ocorre com ponteiros, objetos alocáveis armazenam dados, ao invés de descritores.

As matrizes alocáveis, discutidas na seção 7.1, devem ser consideradas como casos particulares de objetos alocáveis em geral. Quando o uso de um objeto alocável se faz necessário, o espaço de memória necessário para armazenar o valor do dado pode ser criado pela instrução `ALLOCATE` e posteriormente liberado pela instrução `DEALLOCATE`. Se o espaço em memória a ser alocado se refere a um objeto escalar, a instrução `ALLOCATE` pode tomar a forma já discutida na seção 7.1.1. Contudo, para certos objetos compostos é necessário empregar a forma mais geral da instrução `ALLOCATE` que será apresentada mais adiante. Abaixo serão discutidos os diversos objetos alocáveis suportados pelo padrão atual da linguagem.

7.3.1 Objetos com parâmetros de tipo deferido

No padrão do Fortran, uma quantidade ou qualidade é *deferida* quando o seu valor não é conhecido no momento da compilação do código, mas sim durante a execução do programa por meio de um comando `ALLOCATE`, por uma atribuição intrínseca, por uma atribuição de ponteiro ou por *associação de argumento* (seção 9.2). No atual padrão, um *parâmetro de tipo deferido* (*deferred type parameter*) se refere ao valor do parâmetro de comprimento de uma variável de caractere ou *string*.[9]

Uma string de comprimento deferido é declarada através de uma das linhas de declaração abaixo

```
CHARACTER(LEN= :), POINTER[, <outros-atr>] :: <lista-nomes>
CHARACTER(LEN= :), ALLOCATABLE[, <outros-atr>] :: <lista-nomes>
```

onde se observa o caractere "`:`" no lugar de uma constante inteira, indicando que o comprimento dos objetos declarados na `<lista-nomes>` deve ser definido posteriormente, durante a execução do programa.

Este recurso possibilita a manipulação dinâmica de strings de comprimentos variáveis em uma miríade de situações distintas. Alguns exemplos são:

```
CHARACTER(LEN= :), POINTER :: VARSTR
CHARACTER(LEN= 100), TARGET :: NOME
CHARACTER(LEN= 200), TARGET :: ENDERECO
VARSTR => NOME
VARSTR => ENDERECO
```

[9] Recursos genéricos para a mudança do tipo um de objeto de dados são usualmente fornecidos por técnicas de programação orientada a objeto. O Fortran possui essa capacidade, mas isso não será abordado nesta Apostila.

Neste exemplo, o comprimento do ponteiro VARSTR é definido por atribuição de ponteiro: primeiro o seu comprimento é 100 (comprimento de NOME) e depois seu comprimento é 200 (ENDERECO).

O recurso da alocação/realocação automática, introduzido na seção 7.1.2.2 para matrizes, também se aplica neste caso. O programa abaixo apresenta alguns casos simples de uso deste recurso. Nota-se que foi empregada a constante iostat_eor, a qual é parte do módulo intrínseco ISO_FORTRAN_ENV (seção 9.13.6).

Listagem 7.7: Alguns exemplos de uso de strings com comprimento deferido para objetos escalares.

```
program tes_deferred_type_scalar
use iso_fortran_env, only: iostat_eor
implicit none
character(len= 1) :: buffer
character(len= :), allocatable :: str1, str2, input
integer :: ion

str1= 'Bom dia!'  ! Alocação automática
print'(3a,g0)', 'String (1): ->|', str1, '|<- Comp. string: ', len(str1)
str2= 'Tudo bem?' ! Alocação automática
print'(3a,g0)', 'String (2): ->|', str2, '|<- Comp. string: ', len(str2)
str1= str1//'  '//str2 ! Realocação automática
print'(3a,g0)', 'String (3): ->|', str1, '|<- Comp. string: ', len(str1)

!Entre com uma frase qualquer no teclado.
input= ''
print'(/,a)', 'Escreva uma frase abaixo:'
do
   read(unit= *, fmt= '(a)', advance= 'no', iostat= ion)buffer
   select case(ion)
   case(0)
      input= input//buffer
   case(iostat_eor)
      exit
   end select
end do
print'(3a,g0)', 'String lida: ->|', input, '|<- Comp. string: ', len(input)
end program tes_deferred_type_scalar
```

Os recursos apresentados no programa 7.7 são suficientes para manipular strings variáveis escalares. Para se trabalhar com *matrizes alocáveis de strings alocáveis* torna-se necessário o uso da forma mais geral do comando ALLOCATE, o que será discutido na próxima seção.

7.3.2 A FORMA GERAL DA INSTRUÇÃO ALLOCATE

A forma mais geral da instrução ALLOCATE é:

```
ALLOCATE([<type-spec> ::] <lista-alloc> [, <spec-alloc>] ...)
```

Nesta instrução, <lista-alloc> é uma lista de alocações, sendo que cada elemento da lista tem a forma

```
<obj-allocate>[(<lista-ext-array>)]
```

e cada <ext-array> na <lista-ext-array> é uma declaração de extensões na forma

```
[<lim-inferior>:]<lim-superior>
```

Quando presente, <spec-alloc> é pelo menos uma das especificações de alocação seguintes:

```
ERRMSG= <err-mess>
STAT= <status>
SOURCE= <type-expr>
MOLD= <type-expr>
```

Os campos ERRMSG= e STAT= são os mesmos já discutidos na forma mais simples do comando, apresentada na seção 7.1.1.

Os especificadores <type-spec>, SOURCE= e MOLD= contêm informações a respeito dos tipos dos objetos de dados sendo alocados. Quando empregados, o uso destes especificadores se dá de forma excludente: ou é empregado somente o especificador <type-spec>, quando então ocorre uma *alocação de tipo* (*typed allocation*), ou somente é empregado um dos especificadores SOURCE= ou MOLD=, quando então ocorre a *alocação de fonte* (*sourced allocation*). Estes distintos tipos de alocação são discutidos a seguir, no contexto dos objetos alocáveis em estudo nesta seção. Contudo, este recurso também é empregado em técnicas de programação orientada a objeto.

7.3.2.1 Alocação de tipo

Na alocação de tipo, as informações sobre o tipo e parâmetros de espécie dos objetos a ser alocados são mencionados explicitamente no campo <type-spec> ::. Este recurso pode ser empregado para alocar espaço de memória para strings escalares de comprimento deferido, como em

```
INTEGER :: COMP
CHARACTER(LEN= :), ALLOCATABLE :: STRV
READ*, COMP
ALLOCATE(CHARACTER(LEN= COMP) :: STRV)
READ'(A)', STRV
```

Neste exemplo, a string STRV é declarada ser de comprimento deferido. Então, a variável inteira COMP é lida, a qual determinará o comprimento de STRV, determinado via alocação de tipo pelo comando ALLOCATE.

A alocação de tipo também pode ser empregada para declarar a forma de uma *matriz alocável* composta por *strings de comprimento deferido*, como no exemplo:

```
INTEGER :: M, N
CHARACTER(LEN= :), DIMENSION(:), ALLOCATABLE :: VSTRV
⋮ ! M e N são determinados
ALLOCATE(CHARACTER(LEN= M) :: VSTRV(N))
```

Neste exemplo, o programa executável irá alocar espaço em memória suficiente para armazenar um vetor VSTRV composto por N elementos, sendo que cada elemento é uma string de comprimento M. Observa-se que neste caso, após a alocação de memória, VSTRV passa a ser um vetor de objetos homogêneos; ou seja, todos os seus elementos têm o mesmo comprimento. Matrizes compostas por strings heterogêneas podem ser implementadas usando tipos derivados com componentes de tipo deferido. Isto será discutido na seção 7.3.3

7.3.2.2 Alocação de fonte

De forma alternativa à alocação de tipo, as informações sobre o tipo e parâmetros de espécie dos objetos a ser alocados podem ser determinadas a partir de objetos já existentes ou de expressões que determinam o tipo e espécie. No primeiro caso, com o qualificador SOURCE=, como em

```
ALLOCATE(OBJ1, SOURCE= OBJ2)
```

o tipo e espécie do objeto alocável OBJ1 (e, portanto, o tamanho de memória requerido) são tomados do objeto OBJ2, o qual por hipótese já existe. Além disso, o valor de OBJ2 é copiado em OBJ1; em essência, OBJ1 é um clone de OBJ2.

Já com o outro qualificador (MOLD=), como em

```
ALLOCATE(OBJ3, MOLD= OBJ4)
```

o tipo e espécie do objeto OBJ3 são determinados a partir de OBJ4, porém não ocorre cópia do valor de OBJ4.

O exemplo a seguir mostra alguns usos simples destes recursos.

```
program tes_typed_sourced_allocation
implicit none
integer :: m
character(len= :), allocatable :: str1               ! Escalar alocável
character(len= :), dimension(:), allocatable :: str2 ! Vetor alocável
character(len= *), parameter :: cst1= 'String 1', &
                                cst2= 'Constante de string 2 (mais longa)'

print'(a)', '===========> Alocações de escalar <==========='
allocate(character(len= 20) :: str1) ! Alocação de tipo: escalar
print*, 'Digite uma frase:'
read'(a)', str1
print'(3a,g0)', 'str1 (1): ->|', str1, '|<- Comp. string: ', len(str1)

! Realocação automática continua sendo possível
str1= 'Esta é uma frase que tem mais de 20 caracteres de comprimento.'
print'(/,3a,g0)', 'str1 (2): ->|', str1, '|<- Comp. string: ', len(str1)

! Alocação de fonte com source=
deallocate(str1) ! Primeiro libere o espaço anterior
allocate(str1, source= cst1)
print'(/,3a,g0)', 'str1 (3): ->|', str1, '|<- Comp. string: ', len(str1)

! Alocação de fonte com mold=
deallocate(str1) ! Primeiro libere o espaço anterior
allocate(str1, mold= cst2)
print'(/,3a,g0)', 'str1 (4): ->|', str1, '|<- Comp. string: ', len(str1)
print*, 'Digite uma frase:'
read'(a)', str1
print'(3a,g0)', 'str1 (1): ->|', str1, '|<- Comp. string: ', len(str1)

print'(2/,a)', '===========> Alocações de vetor <==========='
write(*,'(a)',advance='no')'Nº elementos do vetor (>= 2): ' ; read*, m
allocate(character(len= len(cst2)) :: str2(m)) ! Alocação de tipo
print'(a)', 'Alocação de tipo:'
str2(1)= cst1 ; print'(3a)', 'str2(1)= ->|', str2(1), '|<-'
str2(2)= cst2 ; print'(3a)', 'str2(2)= ->|', str2(2), '|<-'
! Entre com outros elementos...

! Alocação de fonte com mold=
deallocate(str2) ! Primeiro libere o espaço anterior
allocate(str2(m), mold= cst1)
print'(/,a)', 'Alocação de fonte:'
str2(1)= cst1 ; print'(3a)', 'str2(1)= ->|', str2(1), '|<-'
str2(2)= cst2 ; print'(3a)', 'str2(2)= ->|', str2(2), '|<-'
end program tes_typed_sourced_allocation
```

7.3.3 TIPOS DERIVADOS COM COMPONENTES ALOCÁVEIS

Já foi mencionado na seção 7.1.3, no contexto de matrizes alocáveis, que componentes de tipos derivados também podem ser alocáveis. No presente contexto este conceito se estende para qualquer tipo de componente que, individualmente, é um objeto de dados alocável. Dessa forma é possível implementar matrizes cujos elementos são strings heterogêneas, isto é, strings que possuem comprimentos distintos. Uma implementação deste recurso é mostrada no programa abaixo.

```
program tes_derived_type_string
use iso_fortran_env, only: iostat_eor
```

```
implicit none
integer :: n, ion
character(len= 1) :: buffer
character(len= :), allocatable :: str
type :: td
   character(len= :), allocatable :: cs
end type td
type(td), dimension(:), allocatable :: vtd

allocate(vtd(0)) ! Cria vetor tamanho zero.
print'(a)', 'Entre com diversas frases.  Escreva "Fim" para encerrar.'
loop1: do
   write(*, '(a)', advance= 'no')'Frase: '
   str= ''
   loop2: do
      read(*, '(a)', advance= 'no', iostat= ion)buffer
      select case(ion)
      case(0)
         str= str//buffer
      case(iostat_eor)
         exit loop2
      end select
   end do loop2
   if(str == "Fim")exit loop1
   vtd= [ vtd, td(str) ]
end do loop1

print'(2/,a)', 'Frases lidas:'
print'(3a)', ('->|', vtd(n)%cs, '|<-', n= 1, size(vtd))
end program tes_derived_type_string
```

Um componente alocável pode ser de qualquer tipo intrínseco ou derivado. É permitido também que um componente alocável seja do mesmo tipo derivado sendo definido ou de um tipo definido posteriormente na mesma unidade de programa. Este recurso é exemplificado no tipo `MY_REAL_LIST` abaixo:

```
TYPE :: MY_REAL_LIST
   REAL :: VALOR
   TYPE(MY_REAL_LIST), ALLOCATABLE :: NEXT
END TYPE MY_REAL_LIST
```

Uma definição como esta serve para contruir *listas alocáveis*, as quais são discutidas na seção 7.4.1.

7.4 Estruturas dinâmicas de dados

Com os recursos disponíveis para a criação e manipulação de objetos dinâmicos de dados, estruturas extremamente complexas podem ser implementadas, as quais atendem as mais diversas exigências. Nesta seção algumas das estruturas dinâmicas de dados mais comuns serão abordadas, começando por *listas encadeadas.*

7.4.1 Listas encadeadas ou alocáveis

Nesta seção serão discutidas duas implementações de um dos tipos mais simples de estrutura dinâmica de dados: uma estrutura linear, na qual é estabelecida uma coleção linear de uma quantidade indeterminada de objetos de dados. As implementações apresentadas são *listas encadeadas* e *listas alocáveis.*

7.4.1.1 LISTAS ENCADEADAS

Uma *lista encadeada* (*linked list*) é uma coleção linear de objetos de dados de tamanho indeterminado. Cada elemento da lista possui componentes que armazenam valores de dados e pelo menos uma *referência* ou *conexão* (*link*) ao próximo elemento da lista. Em linguagens modernas de programação, listas encadeadas são implementadas por meio de estruturas que contêm componentes de dados e ponteiros que estabelecem a referência ao próximo elemento.

A figura 7.4 ilustra um exemplo básico de lista encadeada. Cada elemento da lista (um *nodo* da lista) é composto por um dado (`valor`) e por um ponteiro (`next`) que pode aponta para o próximo elemento da lista, exceto pelo último, o qual aponta para `NULL()`. Esta ilustração faz lembrar o elos de uma corrente; por esta razão esta estrutura é denominada uma lista encadeada. O início da lista é identificado pelo ponteiro `head`, enquanto que o último elo na lista é localizado por `tail`.

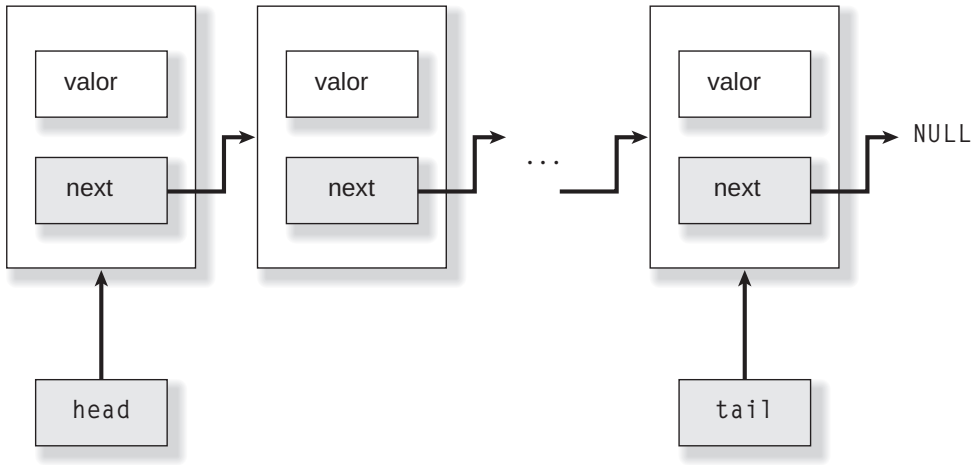


Figura 7.4: Uma lista encadeada típica. Cada ponteiro (*`next`*) aponta para o próximo elemento da lista.

Listas encadeadas oferecem uma outra solução ao problema apresentado no início deste capítulo: como armazenar um conjunto previamente indeterminado de objetos de dados. A solução apresentada por uma lista encadeada pode ser equivalente a uma matriz alocável em qualquer circunstância, mas listas encadeadas oferecem mais opções de armazenamento do que matrizes.

A criação de uma lista encadeada simples será exemplificada agora. Primeiro, define-se um tipo derivado que contém um valor real e um componente de ponteiro do mesmo tipo, o qual irá estabelecer a conexão com o nodo seguinte:

```
TYPE :: NODO
   REAL :: VALOR
   TYPE(NODO), POINTER :: NEXT => NULL()
END TYPE NODO
```

Este tipo pode, obviamente, conter outros componentes; por exemplo, pode-se declarar um componente inteiro para servir de contador (ou índice) do número de nodos presentes na lista.

Uma vez definido o tipo, declaram-se ponteiros para compor os nodos da lista. São necessários pelo menos dois ponteiros, denominados `HEAD` e `CURRENT`, os irão armazenar os descritores do início da lista e do nodo corrente, respectivamente. Se for conveniente, pode-se declarar um outro ponteiro, `TAIL`, o qual irá armazenar o endereço do último nodo. Os ponteiros `HEAD` e `TAIL` fornecem acesso imediato aos pontos extremos da lista. Os ponteiros são declarados então:

```
TYPE(NODO), POINTER :: HEAD => NULL(), CURRENT => NULL(), TAIL => NULL()
```

A figura 7.5 ilustra os passos na construção da lista. No painel (a) mostra-se o status dos ponteiros no momento das suas declarações.

Até o momento, todos os ponteiros estão desassociados e nenhum espaço em memória foi alocado. Assim, se não houver valores para serem lidos, basta seguir adiante com o programa

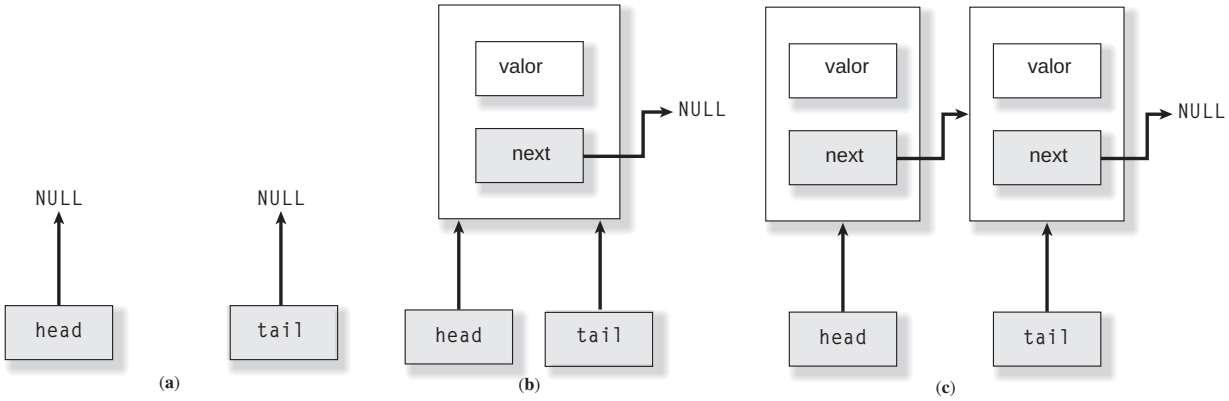


Figura 7.5: Construindo uma lista encadeada: (a) Ponteiros inicialmente desassociados. (b) Situação após a inclusão do primeiro nodo na lista. (c) Situação após a inclusão do segundo nodo na lista.

sem que haja disperdício de recursos. Supondo que exista pelo menos um valor real lido, aloca-se então um espaço anônimo na memória para o componente real, criando também um descritor para o componente ponteiro. Os ponteiros `HEAD` e `TAIL` adquirem então o descritor desse espaço de memória:

```
REAL :: VAR ! Armazena os valores lidos.
⋮ ! O valor real é lido
ALLOCATE(HEAD)
HEAD%VALOR= VAR
TAIL => HEAD
```

Desta maneira, o primeiro nodo da lista encadeada é criado. O painel (b) da figura 7.5 ilustra o status dos ponteiros. Nota-se que os ponteiros `HEAD` e `TAIL` ambos contém o endereço do primeiro bit do espaço de memória alocado. Observa-se também que o seus componentes `NEXT` continuam desassociados, mas os ambos os ponteiros irão conter o descritor do mesmo se `NEXT` for alocado.

Suponha agora que exista pelo menos mais um valor real a ser armazenado. Então, após o mesmo ser lido, um novo nodo da lista deve ser criado. Isto é realizado através dos seguintes passos: *(i)* aloca-se memória para o componente `TAIL%NEXT`. *(ii)* Realiza-se a atribuição `TAIL => TAIL%NEXT`; isto significa que o ponteiro `TAIL` agora contém o descritor (o endereço) do novo espaço de memória, mas como o componente `TAIL%NEXT` era compartilhado com o ponteiro `HEAD`, no momento da alocação o seu componente `HEAD%TAIL` continuou apontando para o início do novo espaço de memória. *(iii)* Finalmente, atribui-se a `TAIL%VALOR` o valor lido. O painel (c) da figura 7.5 ilustra a situação após a inclusão do segundo nodo na lista. A sequencia de comandos é a seguinte:

```
⋮ ! Novo valor de VAR lido
ALLOCATE(TAIL%NEXT)
TAIL => TAIL%NEXT
TAIL%VALOR= VAR
```

Caso existam novos valores a serem incluídos, os passos *(i)* – *(iii)* acima são repetidos até que as entradas se encerrem. A cada novo nodo incluído, o ponteiro `TAIL` estará sempre apontando para o último nodo, enquanto que o ponteiro `HEAD` permanece estoicamente apontando para o primeiro nodo.

Após a lista toda ser lida, pode-se voltar para o início da mesma para armazenar o seus valores em um vetor ou para outros fins quaisquer. Isto é realizado com a atribuição inicial `CURRENT => HEAD`, a qual irá resultar com `CURRENT` contendo o endereço do início da lista. Para acessar os outros nodos, deve-se lembrar que `HEAD%NEXT` contém o endereço do segundo nodo e, portanto, `CURRENT%NEXT` também. Assim, basta realizar a atribuição `CURRENT => CURRENT%NEXT` que o segundo nodo estará acessível. Repetindo esta última atribuição, percorre-se a lista no sentido `HEAD` $\longrightarrow$ `TAIL`. Para verificar se o final da lista foi atingido, basta lembrar que o componente `TAIL%NEXT` está desassociado. Portanto, quando `CURRENT` estiver com os descritores de `TAIL`, a

atribuição CURRENT => CURRENT%NEXT irá levar CURRENT ao status de desassociado. Assim, basta testar o seu status para determinar quando o final da lista foi atingido. A sequencia de comandos fica assim:

```
CURRENT => HEAD
DO
   IF(.NOT. ASSOCIATED(CURRENT))EXIT
   ⋮ ! Faça algo com CURRENT%VALOR
   CURRENT => CURRENT%NEXT
END DO
```

O programa abaixo é uma implementação deste algoritmo. Deseja-se ler da entrada padrão uma quantidade indeterminada de valores reais não nulos. Assim, para verificar o final das entradas, o programa simplesmente testa se o valor é zero. Uma vez que todos os números não nulos foram lidos, o programa imprime os valores acumulados na lista na saída padrão. É importante ressaltar que o ponteiro TAIL pode ser substituído por CURRENT, uma vez que sua presença neste programa não é estritamente necessária.

Listagem 7.8: Exemplo de implementação de uma lista singularmente encadeada.

```
1  program linked_list_01
2  implicit none
3  real :: var
4  type :: nodo
5     real :: valor
6     type(nodo), pointer :: next => null()
7  end type nodo
8  type(nodo), pointer :: head => null(), tail => null(), current => null()
9  ! Procede a atribuição de valores na lista
10 do
11    write(*,'(a)',advance='no')'valor (0.0: saída)= ' ; read*, var
12    if(var == 0.0)exit
13    if(.not. associated(head))then ! Inicia a lista
14       allocate(head)              ! Aloca espaço na memória para head
15       head%valor= var             ! Acumula valor lido em head
16       tail => head                ! Tail aponta para head
17    else                     ! Inicia um novo nodo
18       allocate(tail%next) ! Aloca espaço para o próximo nodo
19       tail => tail%next   ! Tail aponta para o novo nodo
20       tail%valor= var     ! Acumula valor no nodo
21    end if
22 end do
23 !Lista lida.  Agora retorna ao início e imprime valores na tela
24 print'(/,a)', 'Valores lidos:'
25 if(associated(head))then ! Primeiro confirma que lista existe
26    current => head ! Retorna ao início
27    do
28       if(.not. associated(current))then
29          tail => null()
30          exit ! Final da lista
31       end if
32       print'(a,g0)', 'Valor= ', current%valor
33       head => current         ! Move início para o nodo corrente
34       current => current%next ! Aponta para o próximo nodo
35       deallocate(head)        ! Libera espaço do nodo anterior
36    end do
37 end if
38 ! Tudo pronto.  Realize a limpeza final e verifique
39 print'(/,a,g0)', 'Ponteiro head está associado?    ', associated(head)
40 print'(a,g0)', 'Ponteiro tail está associado?    ', associated(tail)
41 print'(a,g0)', 'Ponteiro current está associado? ', associated(current)
```

```
42 end program linked_list_01
```

Uma leitura atenta do programa mostra que há instruções adicionais às que constam no algoritmo, especificamente nas linhas 13, 25, 29, 33 e 35. Se o objetivo do programa for simplesmente ler a lista uma única vez e imprimir os valores na saída padrão, então essas instruções não são necessárias. Contudo, em um programa mais longo, no qual a lista pode ser reconstruída diversas vezes, tanto no programa principal quanto em outras unidades de programa, as instruções citadas são importantes, principalmente nas linhas 29, 33 e 35. Na linha 33, o ponteiro `HEAD` adquire os descritores de `CURRENT` antes que este último aponte para o próximo nodo. Em seguida, `CURRENT` se move para o nodo seguinte e, na linha 35, o comando `DEALLOCATE(HEAD)` libera o espaço em memória ocupado pelo nodo anterior e ao mesmo tempo desassocia `HEAD`. Estas instruções são importantes porque se posteriormente fosse necessário iniciar uma nova lista encadeada, uma nova execução do `ALLOCATE(HEAD)` (linha 14) iria inicar a nova lista em um outro espaço de memória e todos os novos nodos também ocupariam outros espaços. Contudo, os espaços de memória reservados para a lista anterior permaneceriam alocados caso não fossem executadas as instruções nas linhas 33 e 35, e esse espaço permaneceria inacessível e ocioso durante todo o restante da execução do programa. Em uma situação onde a lista é recriada diversas vezes, a existência desses espaços ociosos e reservados de memória pode resultar em uma condição de esgotamento da memória disponível no computador. Este acontecimento é denominado *vazamento ou esgotamento de memória* (*memory leak*). As instruções nas linhas 33 e 35 evitam a ocorrência do vazamento de memória.

Por outro lado, caso a instrução na linha 29 não fosse aplicada, o ponteiro `TAIL` permaneceria com o status de definido e com os descritores de um espaço de memória que foi dealocado, isto é, que foi disponibilizado de volta ao sistema operacional. Nesta situação, o descritores em `TAIL` perdem o sentido de ser, uma vez que o programa pode usar esse espaço de memória para armazenar outros tipos de dados. Um uso posterior de `TAIL` poderia primeiro constatar que o mesmo se encontra associado e assim supor que os descritores contidos no mesmo permanecem válidos. Contudo, se entrementes o programa gravou outro dado nesse espaço, ou em parte desse espaço, os valores acessíveis por `TAIL` serão completamente errôneos. Quando um ponteiro retém o endereço de um espaço de memória que foi dealocado em um outro ponto do programa, este se torna um *ponteiro pendente* (*dangling pointer*). A instrução na linha 29 evita essa ocorrência. As instruções nas linhas 37 – 39 confirmam que todos os ponteiros estão desassociados no final do programa.

Sugestões de uso & estilo para programação

Quando ponteiros são empregados em um programa, é importante sempre tomar providências que evitem a ocorrência de vazamentos de memória (*memory leaks*) ou de ponteiros pendentes (*dangling pointers*).

7.4.1.2 Listas alocáveis

No Fortran existe uma implementação alternativa às listas encadeadas para a formação de uma estrutura linear de dados. Tratam-se das *listas alocáveis* (*allocatable lists*). Na seção 7.3.3 observou-se que componentes alocáveis de tipos derivados podem ser de qualquer tipo, inclusive do próprio tipo sendo definido ou de um tipo que ainda não foi definido.

Com este recurso, pode-se criar uma lista alocável alternativa à lista encadeada mostrada no programa 7.8. O programa abaixo exemplifica o uso deste recurso.

Listagem 7.9: Exemplo de implementação de uma lista alocável.

```
1 program allocatable_list_02
2 implicit none
3 real :: var
4 type :: alloc_list
5    real :: valor
6    type(alloc_list), allocatable :: next
7 end type alloc_list
8 type(alloc_list), allocatable, target :: lista
9 type(alloc_list), pointer :: nodo => null()
```

```
10 ! Procede a atribuição de valores na lista
11 do
12    write(*,'(a)',advance='no')'valor (0.0: saída)= ' ; read*, var
13    if(var == 0.0)exit
14    if(.not. associated(nodo))then ! Inicia a lista
15       allocate(lista, source= alloc_list(valor= var)) ! Inicia lista
16       nodo => lista                ! Nodo aponta para início da lista
17    else                    ! Inicia um novo nodo
18       allocate(nodo%next, source= alloc_list(valor= var)) ! Cria novo nodo
19       nodo => nodo%next    ! Aponta para o novo nodo
20    end if
21 end do ! Nodo continua associado ao final da lista.
22 !Lista lida.  Agora retorna ao início e imprime valores na tela
23 print'(/,a)', 'Valores lidos:'
24 if(allocated(lista))then ! Primeiro confirma que lista existe
25    nodo => lista ! Retorna ao início da lista
26    do
27       if(.not. associated(nodo))exit ! Final da lista
28       print'(a,g0)', 'Valor= ', nodo%valor
29       nodo => nodo%next ! Aponta para o próximo nodo
30    end do
31 end if ! Nodo resulta desassociado ao final da lista
32 ! Tudo pronto.  Realize a limpeza final e verifique
33 deallocate(lista) ! Dealoca todos os nodos da lista
34 print'(/,a,g0)', 'A lista está alocada? ', allocated(lista)
35 print'(a,g0)', 'O ponteiro nodo está associado? ', associated(nodo)
36 end program allocatable_list_02
```

A implementação realizada no programa 7.9 possui algumas diferenças importantes em relação ao programa 7.8. Como os nodos são agora objetos alocáveis ao invés de objetos anônimos, pode-se empregar alocação de fonte (seção 7.3.2.2) juntamente com um construtor de estrutura. Isto é realizado nas linhas 15 e 18, onde na mesma instrução aloca-se o espaço e realiza-se a atribuição de valor do nodo.

Mas a diferença mais importante se revela na parte final do programa. Por ser um objeto alocável, a instrução `deallocate(lista)` na linha 33 irá automaticamente liberar o espaço de memória reservado para *toda a lista*. Neste caso não é necessário dealocar nodo por nodo como foi preciso com o uso de ponteiros. Uma outra vantagem tem impacto na optimização do código. Por exigência do padrão, objetos alocáveis são alocados de forma *contígua*. Por outro lado, tal exigência não é imposta a listas encadeadas formadas por ponteiros; o sistema operacional pode escolher onde reservar espaço para um determinado nodo, sem que este seja necessariamente contíguo em relação aos nodos vizinhos.

7.4.1.3 LISTAS ENCADEADAS CIRCULARES

Uma das limitações de listas encadeadas lineares como as criadas nos programas 7.9 e 7.8 está no fato de que em cada vez que for necessário procurar um determinado valor armazenado na mesma, deve-se iniciar a busca sempre a partir do início da lista, o que terá um impacto no tempo de processamento da busca. Existem diversas estratégias que reduzem o tempo necessário para a realização da busca no interior de uma lista. Uma dessas estratégias consiste no estabelecimento de uma *lista encadeada circular*.

Listas lineares, como aquelas descritas nas seções 7.4.1.1 e 7.4.1.2, são listas *abertas*, isto é, no final da lista a referência ao próximo modo é sempre indefinida (um ponteiro para `NULL` ou objeto não alocado). Por outro lado, uma lista circular é estabelecida quando, no término da incorporação de dados à lista, executa-se a atribuição de ponteiro

```
TAIL%NEXT => HEAD
```

Assim, durante a varredura da lista, ao se chegar ao seu final (`TAIL`) o próximo nodo será imediatamente o início (`HEAD`), conforme está representado na figura 7.6. Uma outra estratégia consiste no estabelecimento de uma *lista duplamente encadeada*, discutida na seção 7.4.2.

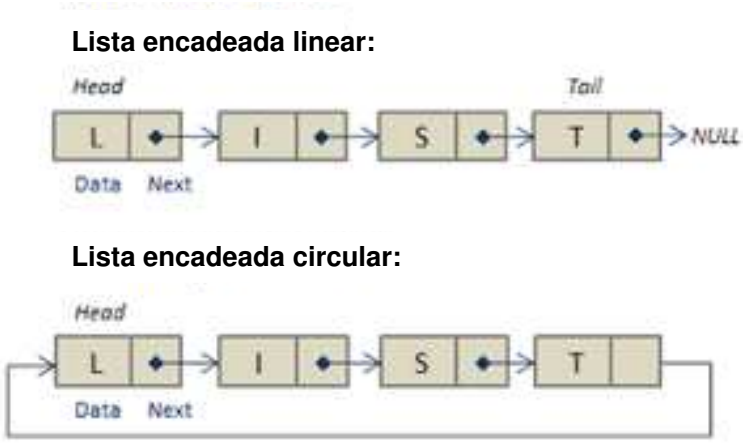


Figura 7.6: Diferença entre uma lista linear e uma lista circular.

7.4.2 Listas duplamente encadeadas e árvores binárias

Serão rapidamente discutidos agora dois tipos comuns de estruturas dinâmicas de dados contendo dois ponteiros em cada nodo: *listas duplamente encadeadas* e *árvores binárias*.

Listas duplamente encadeadas

As listas encadeadas apresentadas na seção 7.4.1 são exemplos de listas *simplesmente* encadeadas. Uma vez que essas listas possuem somente um ponteiro em cada nodo, elas somente podem ser percorridas no sentido `HEAD` $\longrightarrow$ `TAIL`. Um processo de busca no interior da lista poderia ser executado mais rapidamente se a mesma pudesse ser percorrida em ambos os sentidos. Uma *lista duplamente encadeada* permite justamente isso. A definição básica de um tipo derivado que contém um nodo de uma lista duplamente encadeada é :

```
TYPE :: DLNODO
   REAL :: VALOR
   TYPE(DLNODO), POINTER :: PREVIOUS => NULL()
   TYPE(DLNODO), POINTER :: NEXT => NULL()
END TYPE DLNODO
TYPE(DLNODO), POINTER :: HEAD => NULL(), CURRENT => NULL(), TAIL => NULL()
```

O início da lista será novamente determinado por

```
REAL :: VAR ! Armazena os valores lidos.
⋮ ! O valor real é lido
ALLOCATE(HEAD)
HEAD%VALOR= VAR
TAIL => HEAD
```

mas os próximos nodos serão estabelecidos por:

```
⋮ ! Novo valor de VAR lido
CURRENT => TAIL
ALLOCATE(TAIL%NEXT)
TAIL => TAIL%NEXT
TAIL%VALOR= VAR
TAIL%PREVIOUS => CURRENT
```

Desta forma, a cada novo nodo incluído na lista, o componente `PREVIOUS` irá sempre apontar para o nodo anterior, enquanto que o componente `NEXT` permanecerá desassociado. A estrutura final da lista é representada na figura 7.7.

Esta lista se torna uma *lista duplamente encadeada circular* se o componente `HEAD%PREVIOUS` apontar para `TAIL`, ao mesmo tempo em que o componente `TAIL%PREVIOUS` apontar para `HEAD`.

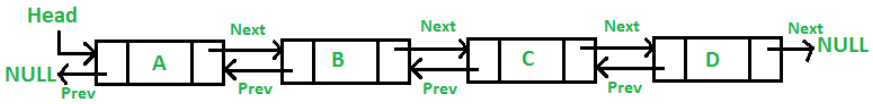


Figura 7.7: Uma lista duplamente encadeada linear.

ÁRVORES BINÁRIAS

Uma *árvore binária* é uma outra estrutura dinâmica cujos nodos possuem dois ponteiros. Contudo, ao invés de estabelecer uma estrutura linear com os mesmos, pode-se estabelecer uma estrutrura que se assemelha às raízes de uma árvore partindo do tronco ou aos galhos de uma árvore invertida. A figura 7.8 ilustra uma árvore binária com diversos nodos estabelecidos.

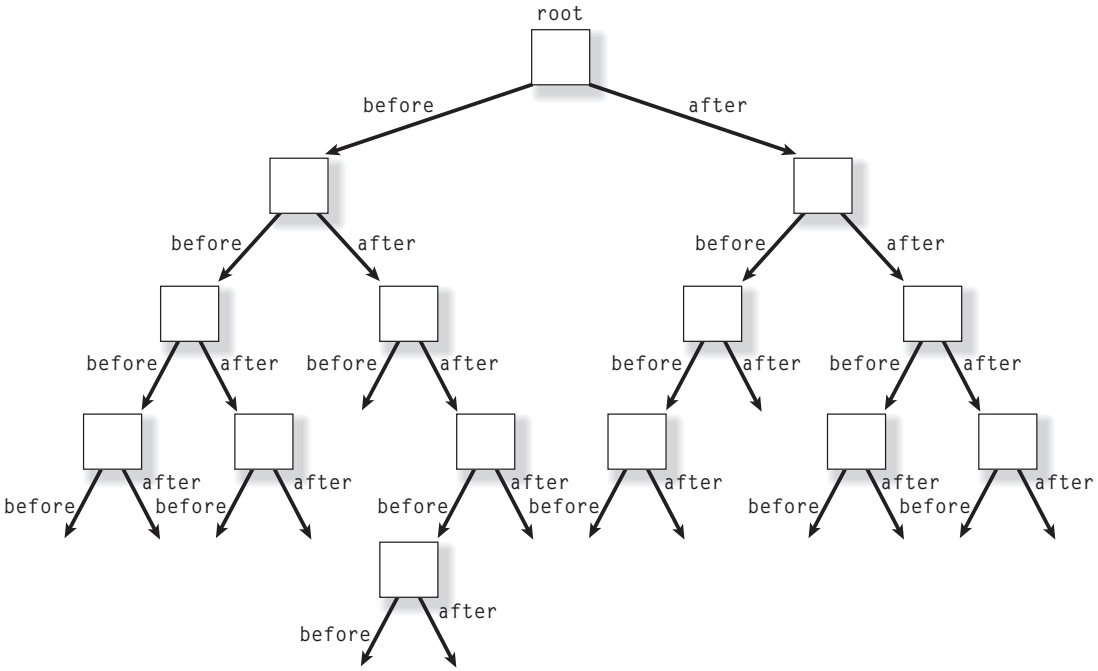


Figura 7.8: Uma árvore binária cujos nodos finais não estão associados.

7.4.3 MATRIZES DE PONTEIROS E LISTAS MULTIPLAMENTE CONECTADAS

Ao longo de toda a seção 7.2, sempre que uma referência foi feita envolvendo ponteiros e matrizes, a menção era referente a *ponteiros para matrizes* (*pointers to arrays* ou *array pointers*). Ou seja, o ponteiro sempre apontava para um alvo que no final das contas era um objeto de dados matricial.

Um tipo de estrutura distinto é uma *matriz de ponteiros* (*array of pointer* ou *pointer array*), isto é, matrizes cujos elementos são ponteiros. No Fortran, uma matriz de ponteiros pode ser criada a partir de um tipo derivado que contém componentes que são ponteiros para matrizes. Por exemplo,

```
TYPE :: ARR_PTR
   REAL, DIMENSION(:), POINTER :: P
END TYPE ARR_PTR
```

estabelece um tipo cujo componente é um ponteiro a um vetor. Declarando matrizes do tipo `ARR_PTR`, cada elemento dessas matrizes são ponteiros a vetores de tamanhos arbitrários. Esta é uma maneira para se criar *matrizes irregulares* (*jagged arrays*).

O programa abaixo implementa uma matriz irregular VPTR na qual o número de linhas, o número de colunas em cada linha e os valores dos elementos são todos determinados aleatoriamente.

```
program jagged_array_01
implicit none
integer, parameter :: mnelem= 2 ! Menor nº de elementos no vetor
integer :: nelem, i, j
real :: temp
integer, dimension(:), allocatable :: tams ! Contém os tamanhos dos vetores
real, dimension(:), pointer :: vtemp
type :: vec_ptr
   real, dimension(:), pointer :: p
end type vec_ptr
type(vec_ptr), dimension(:), allocatable :: vptr

do
   call random_number(temp)
   nelem= int(20.*temp)
   if(nelem >= mnelem)exit
end do
print'(a,g0)', 'Nº elementos do vetor de ponteiros: ', nelem
allocate(tams(nelem), vtemp(nelem))
call random_number(vtemp)
tams= int(20.*vtemp)
print'(/,a,*(g0,x))', 'Tamanhos dos elementos: ', tams

allocate(vptr(nelem))
do i= 1, nelem
   allocate(vtemp(tams(i)))
   call random_number(vtemp)
   vptr(i)%p => vtemp
end do

print'(2/,a)', 'Valores nos elementos de vptr:'
do i= 1, nelem
   print'(/,2(a,g0))', 'Elemento ', i, ':  Nº de elementos: ', size(vptr(i)%p)
   print'(5(g0,x))', (vptr(i)%p(j), j= 1, size(vptr(i)%p))
end do
end program jagged_array_01
```

Estruturas dinâmicas de extrema complexidade podem ser construídas desta maneira. Por exemplo, o par de tipos derivados definidos abaixo:

```
TYPE :: PTR
   TYPE(ENTRY), POINTER :: POINT
END TYPE PTR
TYPE :: ENTRY
   INTEGER :: INDEX
   REAL :: VALOR
   TYPE(PTR), DIMENSION(:), POINTER :: CHILDREN
END TYPE ENTRY
```

estabelece que cada elemento do vetor CHILDREN é um ponto (POINT) a partir do qual um novo vetor CHILDREN de tamanho arbitrário pode ser definido. Isto permite a criação de *listas multiplamente conectadas* ou *encadeadas* (*multiply linked lists*), nas quais cada nodo está conectado com um número arbitrário de outros nodos.

7.5 RECURSOS ORIENTADOS À COMPUTAÇÃO DE ALTA PERFORMANCE

[EM CONSTRUÇÃO]

8 Rotinas Intrínsecas

Em uma linguagem de programação destinada a aplicações científicas há uma exigência óbvia para que as funções matemáticas mais requisitadas sejam oferecidas como parte da própria linguagem, ao invés de terem de ser todas implementadas pelo programador. Além disso, ao serem fornecidas como parte do compilador, espera-se que estas funções tenham sido altamente otimizadas e testadas.

A eficiência das rotinas intrínsecas quando estas manipulam matrizes em computadores vetoriais ou paralelos deve ser particularmente marcante, porque uma única chamada à rotina deve causar um número grande de operações individuais sendo feitas simultaneamente, sendo o código que as implementa otimizado para levar em conta todos os recursos de hardware disponíveis.

No padrão do Fortran há mais de 170 rotinas intrínsecas ao todo. Elas se dividem em grupos distintos, os quais serão descritos em certo detalhe. No padrão do Fortran, uma *rotina* (*procedure*) ou *subprograma* (*subprogram*) é uma designação genérica para dois tipos distintos de unidades de programas: *funções* (*functions*) ou *sub-rotinas* (*subroutines*). Uma discussão detalhada sobre unidades de programa no Fortran é realizada no capítulo 9.

Certos compiladores em particular poderão oferecer rotinas adicionais, além das oferecidas pelo padrão da linguagem; contudo, o preço a pagar é a falta de portabilidade de programas que usam rotinas fora do grupo padrão. Além disso, rotinas extras somente poderão ser oferecidas ao usuário através de *módulos* (capítulo 9).

Ao longo deste capítulo, as rotinas intrínsecas serão naturalmente divididas em funções e sub-rotinas. A distinção entre estas duas classes de rotinas será discutida com mais detalhes no capítulo 9. Todas as rotinas intrínsecas são *genéricas* (seção 9.13.4).

8.1 Conceitos genéricos

As seguintes observações são genéricas para as rotinas apresentadas nesta seção.

8.1.1 Uso de palavras-chave

As rotinas apresentadas neste capítulo podem ter seus argumentos passados com o uso de *palavras-chave*.[1] Por exemplo,

```
CALL DATE_AND_TIME(date= d)
```

irá retornar a data na string escalar `d`.

Argumentos mudos de rotinas que são *opcionais* (seção 9.3.4) são indicados entre colchetes ("`]`" e "`[`"), em comformidade com a notação empregada nesta Apostila.

8.1.2 Categorias de rotinas intrínsecas

Há quatro categorias de rotinas intrínsecas:

Rotinas elementais (*elemental procedures*). São especificadas para argumentos escalares, mas podem também ser aplicadas a matrizes conformáveis. No caso de uma função elemental,

[1] Uma discussão detalhada sobre palavras-chave é realizada na seção 9.3.3.

cada elemento da matriz resultante, caso exista, será obtido como se a função tivesse sido aplicada a cada elemento individual da matriz que foi usada como argumento desta função. No caso de uma sub-rotina elemental com um argumento matricial, cada argumento com intento `IN` ou `INOUT` (seção 9.3.2) deve ser uma matriz e cada elemento resultante será obtido como se a sub-rotina tivesse sido aplicada a cada elemento da matriz que é passada como argumento à sub-rotina.

Funções inquiridoras (ou inquisidoras) (*inquiry functions*). Retornam propriedades dos seus argumentos principais que não dependem dos seus valores, somente do seu tipo e/ou espécie.

Funções transformacionais (*transformational functions*). São funções que não são nem elementais nem inquiridoras. Elas usualmente têm argumentos matriciais e o resultado também é uma matriz cujos elementos dependem dos elementos dos argumentos.

Sub-rotinas não elementais (*non-elemental subroutines*). Rotinas que não se enquadram em nenhum dos tipos acima.

Todas as funções são *puras* (seção 9.10). A sub-rotina `MVBITS` é pura. A sub-rotina `MOVE_ALLOC` é pura, exceto quando aplicada a *coarrays*.

Como todas as funções são puras, seus argumentos todos têm *intent* `IN`. Já quanto as sub-rotinas, o *intent* depende das suas respectivas definições.

8.1.3 Declaração e atributo INTRINSIC

Um nome pode ser especificado como o nome de uma rotina intrínseca com a declaração `INTRINSIC`, o qual possui a forma geral:

```
INTRINSIC [::] <lista-nomes-intrínsecos>
```

onde `<lista-nomes-intrínsecos>` é uma lista de nomes de rotinas intrínsecas. O uso desta declaração é recomendado quando se quer enfatizar ao compilador que certos nomes são destinados a rotinas intrínsecas. Isto pode ser útil quando o programador está estendendo a definição de uma rotina intrínseca, ao fazer uso de *interfaces genéricas* (seção 9.13.4). Alternativamente, uma ou mais funções intrínsecas podem ser declaradas com o atributo `INTRINSIC`, no lugar da declaração.

O atributo ou declaração `INTRINSIC` são excludentes em relação ao atributo ou declaração `EXTERNAL` (seção 9.3.7).

8.2 Funções inquiridoras de qualquer tipo

As seguintes são funções inquiridoras, com argumentos que podem ser de qualquer tipo:

`ALLOCATED(<matriz>)` ou `ALLOCATED(<escalar>)`. Retorna o valor `.TRUE.` quando a matriz alocável `<matriz>` ou o escalar `<escalar>` estão correntemente alocados; na outra situação, o valor `.FALSE.` é retornado.

`ASSOCIATED(<pointer>[, <target>])`. Quanto `<target>` está ausente, a função retorna o valor `.TRUE.` se o ponteiro (*pointer*) está associado com um alvo (*target*) ou retorna o valor `.FALSE.` em caso contrário. O status de associação de ponteiro de `<pointer>` não deve ser indefinido. Se `<target>` estiver presente, este deve ser uma alvo válido para `<pointer>`. O valor da função é `.TRUE.` se `<pointer>` estiver associado a `<target>` ou `.FALSE.` em caso contrário.

No caso de matrizes, `.TRUE.` é obtido somente se as formas são idênticas e elementos de matriz correspondentes, na ordem de elementos de matriz, estão associadas umas com as outras. Se o comprimento do string ou tamanho da matriz são iguais a zero, `.FALSO.` é obtido. Um limite diferente, como no caso de `ASSOCIATED(P,A)` seguindo a atribuição de ponteiro `P => A(:)` quando `LBOUND(A)= 0`, é insuficiente para causar um resultado `.FALSE.`.

O argumento `<target>` pode ser também um ponteiro, em cujo caso o seu alvo é comparado com o alvo de `<pointer>`; o status de associação de ponteiro de `<target>` não deve ser indefinido e se ou `<pointer>` ou `<target>` estão dissociados, o resultado é `.FALSE.`

Se <target> é uma rotina interna (seção 9.2.2) ou um ponteiro associado com uma rotina interna, o resultado é .TRUE. somente se <pointer> e <target> possuírem a mesma instância de hospedeiro.

PRESENT(<arg>). Pode ser chamada em um subprograma que possui um argumento mudo opcional <arg> (ver seção 9.3.4) ou acesse este argumento mudo a partir de seu hospedeiro. A função retorna o valor .TRUE. se o argumento está presente na chamada do subprograma ou .FALSE. em caso contrário. Se um argumento mudo opcional é usado como um argumento real na chamada de outra rotina, ele é considerado também ausente pela rotina chamada.

KIND(X). Aceita o argumento X de qualquer tipo e espécie e o seu resultado tem o tipo inteiro padrão e de valor igual ao valor do parâmetro de espécie de X.

RANK(A). Retorna o inteiro escalar padrão cujo valor é o posto de A, o qual pode ser um escalar ou matriz de qualquer tipo.

8.3 FUNÇÕES ELEMENTAIS NUMÉRICAS

Há 17 funções elementais destinadas à realização de operações numéricas simples, muitas das quais realizam conversão de tipo de variáveis para alguns ou todos os argumentos.

8.3.1 FUNÇÕES ELEMENTAIS QUE PODEM CONVERTER

Se o especificador KIND estiver presente nas funções elementais seguintes, este deve ser uma expressão inteira e fornecer um parâmetro de espécie de tipo que seja suportado pelo processador.

ABS(A). Retorna o valor absoluto de um argumento dos tipos inteiro, real ou complexo. O resultado é do tipo inteiro se A for inteiro e é real nos demais casos. O resultado é da mesma espécie que A.

AIMAG(Z). Retorna a parte imaginária do valor complexo Z. O tipo do resultado é real e a espécie é a mesma que Z.

AINT(A[, KIND]). Trunca um valor real A em sentido ao zero para produzir um valor real cujos valores decimais são iguais a zero. Se o argumento KIND estiver presente, o resultado é da espécie dada por este; caso contrário, retorna o resultado na mesma espécie de A.

ANINT(A[, KIND]). Retorna um real cujo valor é o número completo (sem parte fracionária) mais próximo de A. Se KIND estiver presente, o resultado é do tipo real da espécie definida; caso contrário, o resultado será da mesma espécie de A.

CEILING(A[, KIND]). Retorna o menor inteiro maior ou igual ao seu argumento real. Se KIND estiver presente, o resultado será do tipo inteiro com a espécie definida pelo parâmetro; caso contrário, o resultado será inteiro da espécie padrão.

CMPLX(X[, Y][, KIND]). Converte X ou (X,Y) ao tipo complexo com a espécie definida pelo argumento KIND, caso presente, ou na espécie padrão do tipo complexo em caso contrário. O argumento X pode ser dos tipos inteiro, real ou complexo, ou pode ser uma constante boz. Se X for complexo, Y deve estar ausente. Se Y for ausente e X não for complexo, o resultado será igual àquele obtido se Y = 0.0. Se X é complexo, o ser valor é assumido como sendo dado por REAL(X,KIND) e AIMAG(X,KIND). O valor de CMPLX(X,Y,KIND) tem parte real REAL(X,KIND) e parte imaginária REAL(Y,KIND).

FLOOR(A[, KIND]). Retorna o maior inteiro menor ou igual a seu argumento real A. Se KIND estiver presente, o resultado é do tipo inteiro da espécie definida; caso contrário, o resultado é inteiro da espécie padrão.

INT(A[, KIND]). Converte ao tipo inteiro da espécie padrão ou dada pelo argumento KIND. O argumento A pode ser:

- inteiro, em cujo caso `INT(A)= A`;
- real, em cujo caso o valor é truncado em sentido ao zero, ou
- complexo, em cujo caso somente a parte real é considerada e esta é truncada em sentido ao zero, ou
- uma constante `boz`, quando então o resultado tem a sequência de bits do inteiro especificado após o truncamento à esquerda ou complementação com zeros à esquerda ao comprimento em bits do resultado; se o bit principal for 1, o valor depende do processador.

`NINT(A[, KIND]).` Retorna o valor inteiro mais próximo do real `A`. Se `KIND` estiver presente, o resultado é do tipo inteiro da espécie definida; caso contrário, o resultado será inteiro da espécie padrão.

`REAL(A[, KIND]).` Converte ao tipo real da espécie dada pelo argumento `KIND`, se estiver presente. Se `KIND` estiver ausente, a espécie será a de `A`, caso este seja complexo ou real padrão para outros tipos. O argumento `A` pode ser:

- inteiro ou real, quando então o resultado será uma aproximação de `A` dependente do processador;
- complexo, quando então o resultado será uma aproximação da parte real de `A`, dependente do processador; ou
- uma constante `boz`, quando então o resultado tem a sequência de bits do inteiro especificado após o truncamento à esquerda ou complementação com zeros à esquerda ao comprimento em bits do resultado.

8.3.2 Funções elementais que não convertem

As seguintes são funções elementais cujos resultados são do tipo e da espécie iguais aos do primeiro ou único argumento. Para aquelas que possuem mais de um argumento, todos devem ser do mesmo tipo e espécie.

`CONJG(Z).` Retorna o conjugado do valor complexo `Z`.

`DIM(X, Y).` Retorna `MAX(X-Y,0.)` para argumentos que são ambos inteiros ou ambos reais.

`MAX(A1, A2[, A3, ...]).` Retorna o máximo (maior valor) entre dois ou mais valores, os quais devem ser todos inteiros, reais ou de caracteres.

`MIN(A1, A2[, A3, ...]).` Retorna o mínimo (menor valor) entre dois ou mais valores, os quais devem ser todos inteiros, reais ou de caracteres.

`MOD(A, P).` Retorna o restante de `A` módulo `P`, ou seja, `A-INT(A/P)*P`. O valor de `P` não pode ser nulo; `A` e `P` devem ambos ser inteiros ou ambos reais.

`MODULO(A, P).` Retorna `A` módulo `P` quando `A` e `P` são ambos inteiros ou ambos reais; isto é, `A-FLOOR(A/P)*P` no caso real ou `A-FLOOR(A÷P)*P` no caso inteiro, onde ÷ representa divisão matemática ordinária. O valor de `P` não pode ser nulo.

`SIGN(A, B).` Retorna o valor absoluto de `A` vezes o sinal de `B`. Os argumentos `A` e `B` podem ser de diferentes tipos e/ou espécies, sendo que o tipo e espécie do resultado são os mesmos de `A`. Se `B= 0` e é inteiro, seu sinal é assumido positivo. Se o processador tem a capacidade de distinguir entre zeros reais positivos ou negativos então, para `B= 0.0`, o resultado tem o sinal de `B` (seção 8.8.1); em caso contrário, o sinal é assumido positivo.

8.4 Funções elementais matemáticas

As seguintes são funções elementais que calculam o conjunto imagem de funções matemáticas elementares. O tipo e espécie do resultado são iguais aos do primeiro argumento, o qual, usualmente, é o único argumento.

ACOS(X). Retorna a função arco cosseno (ou $\cos^{-1}$) para valores reais ou complexos do argumento `X` ($|\texttt{X}| \leqslant 1$). O resultado é obtido em radianos no intervalo $0 \leqslant \texttt{REAL(ACOS(X))} \leqslant \pi$.

ACOSH(X). Retorna o arco cosseno hiperbólico (ou $\cosh^{-1}$)para valores reais ou complexos do argumento `X`.

ASIN(X). Retorna a função arco seno (ou $\sin^{-1}$) para valores reais ou complexos do argumento `X` ($|\texttt{X}| \leq 1$). O resultado é obtido em radianos no intervalo $-\pi/2 \leqslant \texttt{REAL(ASIN(X))} \leqslant \pi/2$.

ASINH(X). Retorna o arco seno hiperbólico (ou $\sinh^{-1}$) para valores reais ou complexos do argumento `X`.

ATAN(X). Retorna a função arco tangente (ou $\tan^{-1}$) para valores reais ou complexos do argumento `X`. O resultado é obtido em radianos no intervalo $-\pi/2 \leqslant \texttt{REAL(ATAN(X))} \leqslant \pi/2$.

ATAN(Y,X). O mesmo que `ATAN2(Y,X)`; veja abaixo.

ATAN2(Y,X). Retorna a função arco tangente (ou $\tan^{-1}$) para pares de argumentos reais, `X` e `Y`, ambos do mesmo tipo e espécie. O resultado é o valor principal do argumento do número complexo `(X,Y)`, expresso em radianos no intervalo $-\pi \leqslant \texttt{ATAN2(Y,X)} \leqslant \pi$. Os valores de `X` e `Y` não devem ser simultaneamente nulos. Se o processador consegue distinguir entre zero real positivo ou negativo, uma aproximação de $-\pi$ é retornada se $\texttt{X} < 0.0$ e `Y` é um zero negativo. Se `X` é zero, o valor absoluto do resultado é aproximadamente $\pi/2$.

ATANH(X). Retorna o arco tangente hiperbólico para valores reais ou complexos do argumento `X`. Se o resultado é complexo, a parte imaginária é expressa em radianos e satisfaz a inegualdade $-\pi/2 \leqslant \texttt{AIMAG(ATANH(X))} \leqslant \pi/2$.

BESSEL_J0(X). Retorna a função de Bessel do primeiro tipo e ordem zero. O argumento `X` deve ser real.

BESSEL_J1(X). Retorna a função de Bessel do primeiro tipo e ordem um. O argumento `X` deve ser real.

BESSEL_JN(N, X). Retorna a função de Bessel do primeiro tipo e ordem `N`. O argumento `X` deve ser real e `N` deve ser um inteiro não negativo. Veja a seção 8.5 para `BESSEL_JN(N1, N2, X)`.

BESSEL_Y0(X). Retorna a função de Bessel do segundo tipo e ordem zero. O argumento `X` deve ser real.

BESSEL_Y1(X). Retorna a função de Bessel do segundo tipo e ordem um. O argumento `X` deve ser real.

BESSEL_YN(N, X). Retorna a função de Bessel do segundo tipo e ordem `N`. O argumento `X` deve ser real e `N` deve ser um inteiro não negativo. Veja a seção 8.5 para `BESSEL_YN(N1, N2, X)`.

COS(X). Retorna o valor da função cosseno para um argumento dos tipos real ou complexo. A unidade do argumento `X` é suposta ser radianos.

COSH(X). Retorna o valor da função cosseno hiperbólico para um argumento `X` real ou complexo. Se `X` for complexo, a parte imaginária é tratada como um valor em radianos.

ERF(X). Retorna o valor da função erro de `X`, $\frac{2}{\sqrt{\pi}} \int_0^{\texttt{X}} e^{-t^2}\,dt$. O argumento `X` deve ser real.

ERFC(X). Retorna o valor da função erro complementar $1 - \texttt{ERF(X)} = \frac{2}{\sqrt{\pi}} \int_{\texttt{X}}^{\infty} e^{-t^2}\,dt$. O argumento `X` deve ser real.

ERFC_SCALED(X). Retorna a função erro complementar exponencialmente escalonada, $e^{\texttt{X}^2}\texttt{ERFC(X)}$. O argumento `X` deve ser real. Se a espécie de `X` corresponder à precisão dupla do padrão IEEE, a função `ERF(X)` é igual a um para $\texttt{X} > 6$ e `ERFC(X)` gera *underflow* para $\texttt{X} > 26.7$. Portanto, `ERFC_SCALED(X)` é mais útil quando `X` é grande.

EXP(X). Retorna o valor da função exponencial para um argumento `X` real ou complexo. Se `X` é complexo, a parte imaginária do resultado é expressa em radianos.

GAMMA(X). Retorna a função gama de `X`. O argumento `X` deve ser real e distinto de inteiro não positivo.

HYPOT(X, Y). Retorna a distância Euclideana, isto é, $\sqrt{\mathtt{X}^2 + \mathtt{Y}^2}$, calculada sem ocorrência de *overflow* ou *underflow* quando o resultado está dentro da região representável. Os argumentos `X` e `Y` devem ser reais da mesma espécie e o resultado também é um real da mesma espécie.

LOG(X). Retorna o valor da função logaritmo natural para um argumento `X` real ou complexo. No caso real, `X` deve ser positivo. No caso complexo, `X` não pode ser nulo e a parte imaginária do resultado $(\mathtt{Z}_i)$ está no intervalo $-\pi \leqslant \mathtt{Z}_i \leqslant \pi$. Se o processador consegue distinguir entre zero real positivo ou negativo, uma aproximação para $-\pi$ é retornada se a parte real de `X` é menor que zero e a parte imaginária é um zero negativo.

LOG_GAMMA(X). Retorna o logaritmo natural do valor absoluto da função gama, `LOG(ABS(GAMMA(X)))`. O argumento `X` deve ser real e não pode ser um inteiro não positivo.

LOG10(X). Retorna o valor da função logaritmo de base 10 para um argumento `X` real e positivo.

SIN(X). Retorna o valor da função seno para um argumento dos tipos real ou complexo. A unidade do argumento `X` é suposta ser radianos.

SINH(X). Retorna o valor da função seno hiperbólico para um argumento `X` real ou complexo. Se `X` é complexo, a parte imaginária do resultado é tratada como um valor em radianos.

SQRT(X). Retorna o valor da função raiz quadrada para um argumento `X` real ou complexo. No caso real, `X` não pode ser negativo. No caso complexo, o resultado consiste na raiz principal, ou seja, a parte real do resultado é positiva ou nula. Quando a parte real do resultado for nula, a parte imaginária é positiva ou nula. Se a artimética é IEEE, um resultado puramente imaginário negativo é retornado se a parte real do resultado é zero e a parte imaginária de `X` é menor que zero.

TAN(X). Retorna o valor da função tangente para um argumento `X` real ou complexo. Se `X` é real, o resultado é tratado como um valor em radianos. Se `X` é complexo, a parte real do resultado é tratado como um valor em radianos.

TANH(X). Retorna o valor da função tangente hiperbólica para um argumento `X` real ou complexo. Se `X` é complexo, a parte imaginária do resultado é tratada como um valor em radianos.

8.5 Funções transformacionais para as funções de Bessel

Há duas funções transformacionais que retornam matrizes de posto um (vetores) contendo valores das funções de Bessel:

BESSEL_JN(N1, N2, X). Retorna a sequência de funções de Bessel do primeiro tipo do argumento real `X`, com ordens no intervalo `N1` a `N2`.

BESSEL_YN(N1, N2, X). Retorna a sequência de funções de Bessel do segundo tipo do argumento real `X`, com ordens no intervalo `N1` a `N2`.

Os valores de `N1` e `N2` devem ser do tipo inteiro e devem ser não negativos. Todos os argumentos devem ser escalares. Se $\mathtt{N2} < \mathtt{N1}$, o vetor resultante tem tamanho nulo.

Quando vários valores das funções de Bessel são necessários, é mais eficiente computá-las com as funções desta seção, ao invés de computá-las individualmente.

8.6 Funções elementais lógicas e de caracteres

8.6.1 Conversões caractere-inteiro

As seguintes são funções elementais para converter um único caractere em um inteiro e vice-versa.

ACHAR(I[, KIND]). O resultado é do tipo de caractere padrão de comprimento um e retorna o caractere que está na posição especificada pelo valor inteiro I na tabela ASCII de caracteres (tabela 3.3). I deve estar no intervalo $0 \leqslant \texttt{I} \leqslant 127$; caso contrário, o resultado depende do processador. O argumento opcional KIND especifica a espécie do resultado, sendo da espécie padrão caso esteja ausente.

CHAR(I[, KIND]). O resultado é do tipo caractere de comprimento um. A espécie é dada pelo argumento opcional KIND, se presente ou, em caso contrário, será a espécie padrão. A função retorna o caractere na posição I (tipo inteiro) da sequência de intercalação (*collating*) de caracteres interna do processador associada com o parâmetro de espécie relevante. I deve estar no intervalo $0 \leqslant \texttt{I} \leqslant n-1$, onde n é o número de caracteres na sequência de intercalação interna do processador.

IACHAR(C[, KIND]). O resultado é do tipo inteiro padrão e retorna a posição do caractere C na tabela ASCII. Se C não está na tabela, o resultado depende do processador. Se o argumento opcional KIND estiver ausente, o resultado é um inteiro da espécie padrão; caso esteja presente, o argumento determina a espécie do resultado.

ICHAR(C[, KIND]). O resultado é do tipo inteiro padrão e retorna a posição do caractere C na sequência de intercalação interna do processador associada com a espécie de C. Se o argumento opcional KIND estiver ausente, o resultado é um inteiro da espécie padrão; caso esteja presente, o argumento determina a espécie do resultado.

8.6.2 FUNÇÕES DE COMPARAÇÃO LÉXICA

As seguintes funções elementais aceitam strings de caracteres da espécie padrão, realizam uma comparação léxica baseada na sequência de intercalação da tabela ASCII e retornam um resultado lógico da espécie padrão. Se as strings tiverem comprimentos distintos, a mais curta é complementada com espaços em branco à direita.

LGE(STRING_A, STRING_B). Retorna o valor .TRUE. se STRING_A segue STRING_B na sequência estabelecida pela tabela ASCII, ou a iguala. O resultado é .FALSE. em caso contrário.

LGT(STRING_A, STRING_B). Retorna o valor .TRUE. se STRING_A segue STRING_B na sequência estabelecida pela tabela ASCII. O resultado é .FALSE. em caso contrário.

LLE(STRING_A, STRING_B). Retorna o valor .TRUE. se STRING_A precede STRING_B na sequência estabelecida pela tabela ASCII, ou a iguala. O resultado é .FALSE. em caso contrário.

LLT(STRING_A, STRING_B). Retorna o valor .TRUE. se STRING_A precede STRING_B na sequência estabelecida pela tabela ASCII. O resultado é .FALSE. em caso contrário.

8.6.3 FUNÇÕES ELEMENTAIS PARA MANIPULAÇÕES DE STRINGS

As seguintes são funções elementais que manipulam strings. Os argumentos STRING, SUBSTRING e SET são sempre do tipo de caractere e, quando dois estão presentes, ambos devem ser da mesma espécie. O resultado é da mesma espécie de STRING.

ADJUSTL(STRING). Ajusta uma string à esquerda. Ou seja, remove os espaços em branco no início da string e coloca o mesmo número de brancos no final da string.

ADJUSTR(STRING). Ajusta uma string à direita. Ou seja, remove os espaços em branco no final da string e coloca o mesmo número de brancos no início da string.

INDEX(STRING, SUBSTRING[, BACK][, KIND]). Retorna um valor do tipo inteiro padrão, o qual consiste na posição inicial de SUBSTRING como uma substring de STRING, ou zero se tal substring não existe. Se BACK (variável lógica) está ausente, ou se está presente com o valor .FALSE., a posição inicial da primeira das substrings é retornada; o valor 1 é retornado se SUBSTRING tem comprimento zero. Se BACK está presente com valor .TRUE., a posição inicial da última das substrings é retornada; o valor LEN(STRING)+1 é retornado se SUBSTRING tem comprimento zero. Se KIND está presente, esta especifica a espécie do resultado; senão este é da espécie padrão.

LEN_TRIM(STRING[, KIND]). Retorna um valor inteiro padrão correspondente ao comprimento de STRING, ignorando espaços em branco no final do argumento. Se KIND está presente, esta especifica a espécie do resultado; senão este é da espécie padrão.

SCAN(STRING, SET[, BACK][, KIND]). Retorna um valor inteiro padrão correspondente à posição de um caractere de STRING que esteja em SET, ou zero se não houver tal caractere. Se a variável lógica BACK está ausente, ou presente com valor .FALSE., a posição mais à esquerda deste caractere é retornada. Se BACK está presente com valor .TRUE., a posição mais à direita deste caractere é retornada. Se KIND está presente, esta especifica a espécie do resultado; senão este é da espécie padrão.

VERIFY(STRING, SET[, BACK][, KIND]). Retorna o valor inteiro padrão 0 se cada caractere em STRING aparece em SET, ou a posição de um caractere de STRING que não esteja em SET. Se a variável lógica BACK está ausente, ou presente com valor .FALSE., a posição mais à esquerda de tal caractere é retornada. Se BACK está presente com valor .TRUE., a posição mais à direita de tal caractere é retornada. Se KIND está presente, esta especifica a espécie do resultado; senão este é da espécie padrão.

8.6.4 Conversão lógica

A função elemental a seguir converte de uma espécie do tipo lógico em outra.

LOGICAL(L[, KIND]). Retorna o valor lógico idêntico ao valor de L. A espécie do resultado é definida pelo argumento opcional KIND, caso esteja presente, ou é da espécie padrão em caso contrário.

8.7 Funções não elementais para manipulação de strings

8.7.1 Função inquiridora para manipulação de strings

LEN(STRING[, KIND]). Trata-se de uma função inquiridora que retorna um valor inteiro padrão escalar que corresponde ao número de caracteres em STRING se esta é escalar ou de um elemento de STRING se a mesma é uma matriz. O valor de STRING não precisa estar definido. Se KIND está presente, esta especifica a espécie do resultado; senão este é da espécie padrão.

8.7.2 Funções transformacionais para manipulação de strings

Existem duas funções que não são elementais porque o comprimento do resultado depende do valor de um argumento.

REPEAT(STRING, NCOPIES). Forma a string que consiste na concatenação de NCOPIES cópias de STRING, onde NCOPIES é do tipo inteiro e seu valor não deve ser negativo. Ambos argumentos devem ser escalares.

TRIM(STRING). Retorna STRING com todos os espaços em branco no final da variável removidos. STRING deve ser escalar.

8.7.3 Função inquiridora de caractere

A função intrínseca inquiridora NEW_LINE(A) retorna o caractere que pode ser empregado para gerar o término de gravação (o equivalente ao caractere "\n" da linguagem C).

NEW_LINE(A). Retorna o caractere de *nova linha* (*newline*) empregado para saída formatada em *stream*. O argumento A deve ser do tipo de caractere. O resultado é do tipo caractere com a mesma espécie de A. Se o processador não suportar o caractere de nova linha, um branco é retornado.

8.8 FUNÇÕES INQUIRIDORAS E DE MANIPULAÇÕES NUMÉRICAS

8.8.1 MODELOS PARA DADOS INTEIROS E REAIS

As funções inquiridoras e de manipulações numéricas são definidas em termos de modelos de representação de números inteiro e reais para cada espécie suportada pelo processador.

Para cada espécie do tipo inteiro, o modelo gera um conjunto $\{i\}$ de números inteiros dados por:

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}, \tag{8.1}$$

onde $s = \pm 1$, q é um inteiro positivo, r é um inteiro maior que um (usualmente 2) e cada valor de w_k é um inteiro no intervalo $0 \leqslant w_k < r$.

Para cada espécie do tipo real, o modelo gera o conjunto $\{x\} \in \mathbb{R}$ dado por:

$$x = 0$$

e

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, \tag{8.2}$$

onde $s = \pm 1$, p e b são inteiros maiores que um, e é um inteiro no intervalo $e_{\min} \leqslant e \leqslant e_{\max}$ e cada f_k é um inteiro no intervalo $0 \leqslant f_k < b$, exceto f_1, que é não nulo.

Os valores de todos os parâmetros nestes modelos são escolhidos para o processador de tal forma que o modelo melhor se ajuste ao hardware, desde que todos os números sejam representáveis.

8.8.2 FUNÇÕES NUMÉRICAS INQUIRIDORAS

Há nove funções inquiridoras que retornam valores para os modelos de dados associados com seus argumentos. Cada função tem um único argumento que pode ser escalar ou matricial e todas retornam um valor escalar. O valor *per se* do argumento não precisa ser definido, somente o seu tipo e espécie.

DIGITS(X). Para `X` real ou inteiro, retorna o inteiro padrão cujo valor é o número de dígitos significantes no modelo que inclui `X`; isto é, retorna p para `X` real ou q para `X` inteiro.

EPSILON(X). Para `X` real, retorna um resultado real com o mesmo tipo de `X` e que é quase indistinguível do valor `1.0` no modelo que inclui `X`. Ou seja, a função calcula b^{1-p}.

HUGE(X). Para `X` real ou inteiro, retorna o maior valor representável no modelo que inclui `X`. O resultado possui o mesmo tipo e espécie de `X`. O valor é

$$\left(1 - b^{-p}\right) b^{e_{\max}}$$

para `X` real ou

$$r^q - 1$$

para `X` inteiro.

MAXEXPONENT(X). Para `X` real, retorna o inteiro padrão $e_{\max}$, ou seja, o expoente máximo no modelo que inclui `X`.

MINEXPONENT(X). Para `X` real, retorna o inteiro padrão $e_{\min}$, ou seja, o expoente mínimo no modelo que inclui `X`.

PRECISION(X). Para `X` real ou complexo, retorna um inteiro padrão que contém a precisão decimal equivalente no modelo que representa números reais da mesma espécie de `X`. O valor da função é $\texttt{INT}((p-1)) * \texttt{LOG10}(b)) + k$, onde k é 1 se b é uma potência inteira de 10 ou 0 em outro caso.

RADIX(X). Para X real ou inteiro, retorna o inteiro padrão que é a base no modelo que inclui X. Isto é, retorna b para X real ou r para X inteiro.

RANGE(X). Para X inteiro, real ou complexo, retorna o inteiro padrão que contém o intervalo de expoentes decimais nos modelos representando números reais ou inteiro da mesma espécie de X. O valor da função é INT(LOG10($huge$)) para X inteiro e

$$\texttt{INT}(\texttt{MIN}(\texttt{LOG10}(huge), -\texttt{LOG10}(tiny)))$$

para X real, onde $huge$ e $tiny$ são, respectivamente, o maior e o menor números positivos nos modelos.

TINY(X). Para X real, retorna o menor número positivo,

$$b^{e_{\min}-1}$$

no modelo que inclui X. O resultado é do mesmo tipo e espécie de X.

8.8.3 Função elemental para tipos numéricos

A função intrínseca elemental OUT_OF_RANGE() testa se um valor real ou inteiro pode ser convertido de maneira segura a um outro real ou inteiro de qualquer espécie.

OUT_OF_RANGE(X, MOLD[, ROUND]). Retorna um valor lógico escalar padrão.

X é do tipo inteiro ou real.

MOLD é um escalar do tipo inteiro ou real.

ROUND é um escalar lógico e somente pode estar presente se X for do tipo real e MOLD do tipo inteiro.

O valor retornado será .TRUE. se e somente se:

- O valor de X for um infinito ou NaN (padrão IEEE) e MOLD é do tipo inteiro ou do tipo real e de uma espécie que não suporta esse valor.
- MOLD é do tipo inteiro, ROUND está ausente ou presente com valor .FALSE. e o inteiro com maior magnitude que se encontra entre zero e X (inclusive) não for representável por objetos com o tipo e espécie de MOLD.
- MOLD é do tipo inteiro, ROUND está presente com o valor .TRUE. e o inteiro mais próximo de X, ou o inteiro de maior magnitude (se houver dois inteiros igualmente próximos a X) não for representável por objetos com o tipo e espécie de MOLD.
- MOLD é do tipo real e o resultado do arredondamento do valor de X ao modelo estendido para a espécie de MOLD possuir magnitude maior que o maior número finito com o mesmo sinal de X que é representável por objetos do tipo e espécie de MOLD.

8.8.4 Funções elementais que manipulam quantidades reais

Há sete funções elementais cujo primeiro ou único argumento é do tipo real e que retornam valores relacionados aos componentes dos modelos de valores associados ao valor do argumento. Para as funções EXPONENT, FRACTION e SET_EXPONENT, se o valor do argumento X está além do intervalo de números representáveis pelo model, o valor da parte exponencial (e na fórmula 8.2) é determinado como se o modelo não tivesse limites na parte exponencial.

EXPONENT(X). Retorna o inteiro padrão cujo valor é a parte de expoente e de X quando representado como um número de modelo. Se X=0, o resultado é nulo.

FRACTION(X). Retorna uma quantidade real da mesma espécie que X e cujo valor é a parte fracionária de X quando representado como um número de modelo; isto é, a função retorna $\texttt{X}b^{-e}$.

NEAREST(X, S). Retorna uma quantidade real da mesma espécie que X e cujo valor é o número distinto mais próximo de X no sentido dado pelo sinal da quantidade real S. O valor de S não pode ser nulo.

RRSPACING(X). Retorna uma quantidade real da mesma espécie que X cujo valor é a recíproca do espaçamento relativo dos números de modelo próximos a X; isto é, a função retorna $|\mathtt{X}b^{-e}|\,b^{p}$.

SCALE(X, I). Retorna uma quantidade real da mesma espécie que X cujo valor é $\mathtt{X}b^{\mathtt{I}}$, onde b é a base do modelo para X e I é do tipo inteiro.

SET_EXPONENT(X, I). Retorna uma quantidade real da mesma espécie que X cuja parte fracionária é a parte fracionária da representação de X e cuja parte exponencial é I; isto é, a função retorna $\mathtt{X}b^{1-e}$.

SPACING(X). Retorna uma quantidade real da mesma espécie que X cujo valor é o espaçamento absoluto do modelo de números próximos a X. O resultado é b^{e-p} se X é não nulo e este resultado está dentro do intervalo de números representáveis. Caso contrário, a função retorna TINY(X).

8.8.5 FUNÇÕES TRANSFORMACIONAIS PARA VALORES DE ESPÉCIE

Há três funções que retornam o menor valor do parâmetro de espécie que irá satisfazer um dado requerimento de um literal de caractere ou numérico. As funções têm argumentos e resultados escalares; porém são classificadas como transformacionais. Estas funções já foram discutidas na seção 3.3.4.1.

SELECTED_CHAR_KIND(NOME). Retorna o inteiro escalar padrão que é o valor do parâmetro da espécie para o conjunto de caracteres cujo nome é fornecido pela variável de caracteres NOME, ou -1 se o conjunto de caracteres não é suportado ou não é reconhecido. Em particular, os seguintes valores para NOME vêm com o padrão:

default, que irá sempre resultar na espécie do conjunto de caracteres padrão, ou o único suportado. Neste caso, o resultado da função é igual a KIND('a').

ascii, que irá resultar na espécie do conjunto de caracteres ASCII, de acordo com a tabela 3.3.

iso_10646, que irá resultar na espécie do conjunto de caracteres definido na norma internacional ISO/IEC 10646 UCS-4.[2]

O processador ou o compilador podem suportar outras espécies, mas o padrão exige que somente 'default' seja suportado. O nome do conjunto de caracteres não é dependente de caso; assim, 'DEFAULT' ou 'ASCII' também são válidos. Espaços em branco no final do NOME são ignorados.

SELECTED_INT_KIND(R). Retorna o inteiro escalar padrão que é o valor do parâmetro da espécie para um dado do tipo inteiro capaz de representar todos os valor inteiros n no intervalo $-10^{\mathtt{R}} < n < 10^{\mathtt{R}}$, onde R é um inteiro escalar. Se mais de uma espécie for disponível, a espécie com menor intervalo exponencial é escolhida. Se nenhuma espécie é disponível, o resultado é -1.

SELECTED_REAL_KIND([P][, R][, RADIX). Retorna o inteiro escalar padrão que é o valor do parâmetro da espécie para um dado do tipo real com precisão decimal (conforme retornada pela função PRECISION) no mínimo igual a P, intervalo de expoente decimal (conforme retornado pela função RANGE) no mínimo igual a R e radix (conforme retornado pela função RADIX) igual a RADIX. Se mais de uma espécie for disponível, a espécie com a menor precisão decimal é escolhida. Se RADIX está ausente, não há exigência sobre o radix selecionado. Tanto P quanto R são inteiros escalares; pelo menos um destes deve estar presente. Se não houver espécie disponível, o resultado é:

-1: se a precisão requerida não for disponível;

[2] https://pt.wikipedia.org/wiki/ISO/IEC_10646.

-2: se um intervalo de expoente grande o suficiente não for disponível;

-3: se ambos não forem disponíveis.

-4: se o radix está disponível com a precisão ou o intervalo exigidos, mas não para ambos.

-5: se o radix não está disponível.

8.9 Rotinas de manipulação de bits

Há diversas rotinas intrínsercas para manipular bits contidos em quantidades inteiras. Estas rotinas são elementais, quando apropriado. As rotinas são baseadas em um modelo no qual um valor inteiro é representado por s bits w_k, com $k = 0, 1, \ldots, s-1$, em uma sequência da direita para a esquerda, baseada no valor não-negativo

$$\sum_{k=0}^{s-1} w_k \times 2^k.$$

Este modelo é válido somente no contexto destas rotinas intrínsecas e é idêntico ao modelo de números inteiros (8.1) quando $r = 2$ e $w_{s-1} = 0$; mas quando $r \neq 2$ ou $w_{s-1} = 1$ os modelos diferem, e o valor expresso como um inteiro pode variar conforme o processador.

8.9.1 Função inquiridora

BIT_SIZE(I). Retorna o número de bits no modelo para bits dentro de um inteiro da mesma espécie que `I`. O resultado é um inteiro escalar da mesma espécie que `I`.

8.9.2 Funções elementais básicas

BTEST(I, POS). Retorna o valor lógico da espécie padrão `.TRUE.` se o bit `POS` do inteiro `I` tem valor 1 e retorna `.FALSE.` em outra situação. `POS` deve ser um inteiro com valor no intervalo $0 \leqslant$ `POS` $<$ `BIT_SIZE(I)`.

IAND(I, J). Retorna o lógico `AND` de todos os bits em `I` e bits correspondentes em `J`, de acordo com a tabela-verdade

`I`	1	1	0	0
`J`	1	0	1	0
`IAND(I, J)`	1	0	0	0

`I` e `J` devem ser do mesmo tipo; o resultado também será do mesmo tipo.

IBCLR(I, POS). Retorna um inteiro do mesmo tipo que `I` e valor igual ao de `I` exceto que o bit `POS` é levado a 0. `POS` deve ser um inteiro com valor no intervalo $0 \leqslant$ `POS` $<$ `BIT_SIZE(I)`.

IBITS(I, POS, LEN). Retorna um inteiro do mesmo tipo que `I` e valor igual aos `LEN` bits de `I` iniciando no bit `POS` ajustado à direita e com todos os outros bits iguais a 0. `POS` e `LEN` devem ser inteiros positivos tais que `POS` $+$ `LEN` $\leqslant$ `BIT_SIZE(I)`.

IBSET(I, POS). Retorna um inteiro do mesmo tipo que `I` e valor igual ao de `I` exceto que o bit `POS` é levado a 1. `POS` deve ser um inteiro com valor no intervalo $0 \leqslant$ `POS` $<$ `BIT_SIZE(I)`.

IEOR(I, J). Retorna o `OU` lógico exclusivo de todos os bits de `I` e correspondentes bits em `J`, de acordo com a tabela-verdade

`I`	1	1	0	0
`J`	1	0	1	0
`IEOR(I, J)`	0	1	1	0

`I` e `J` devem ser do mesmo tipo; o resultado também será do mesmo tipo.

IOR(I,J). Retorna o `OU` lógico inclusivo de todos os bits de `I` e correspondentes bits em `J`, de acordo com a tabela-verdade

`I`	1	1	0	0
`J`	1	0	1	0
`IOR(I, J)`	1	1	1	0

`I` e `J` devem ser do mesmo tipo; o resultado também será do mesmo tipo.

NOT(I). Retorna o complemento lógico de todos os bits em I, de acordo com a tabela verdade

I	1	0
NOT(I)	0	1

8.9.3 OPERAÇÕES DE DESLOCAMENTO (*shift operations*)

Há sete funções elementais para deslocamento de bits (*bit shifting*): cinco são descritas aqui e duas funções descritas na seção 8.9.6.

ISHFT(I, SHIFT). Retorna um inteiro do mesmo tipo que I e valor igual ao de I exceto que os bits são deslocados SHIFT posições para a esquerda (-ISHIFT desloca para a direita se SHIFT for positivo). Zeros são inseridos a partir da extremidade oposta. SHIFT deve ser um inteiro com valor que satisfaz a inegualdade $|\text{SHIFT}| \leqslant \text{BIT_SIZE}(\text{I})$.

ISHFTC(I, SHIFT[, SIZE]). Retorna um inteiro do mesmo tipo que I e valor igual ao de I exceto que os SIZE bits mais à direita (ou todos os bits se SIZE for ausente) são deslocados de forma circular SHIFT posições para a esquerda (-ISHIFT desloca para a direita se SHIFT for positivo). Zeros são inseridos a partir da extremidade oposta. SHIFT deve ser um inteiro com valor que não excede o valor de SIZE ou BIT_SIZE(I), se SIZE estiver ausente.

SHIFTA(I, SHIFT). Retorna os bits de I deslocados para a direita por SHIFT bits, mas ao invés de introduzir bits zero à esquerda, o bit mais à esquerda é reproduzido. O argumento SHIFT deve ser um inteiro com valor satisfazendo $0 \leqslant \text{SHIFT} \leqslant \text{BIT_SIZE(I)}$.

SHIFTL(I, SHIFT). Retorna os bits de I deslocados para a esquerda, de forma equivalente a ISHFT(I, SHIFT). O argumento SHIFT deve ser um inteiro cujo valor satisfaz $0 \leqslant \text{SHIFT} \leqslant \text{BIT_SIZE(I)}$.

SHIFTR(I, SHIFT). Retorna os bits de I deslocados para a direita, de forma equivalente a ISHFT(I, -SHIFT). O argumento SHIFT deve ser um inteiro cujo valor satisfaz $0 \leqslant \text{SHIFT} \leqslant \text{BIT_SIZE(I)}$.

8.9.4 SUBROTINA ELEMENTAL

MVBITS(FROM, FROMPOS, LEN, TO, TOPOS). Copia a sequência de bits em FROM que inicia na posição FROMPOS e tem comprimento LEN para TO, iniciando na posição TOPOS. Os outros bits de TO permanecem inalterados. FROM, FROMPOS, LEN e TOPOS são todos inteiros com intent IN e devem ter valores que satisfazem as inegualdades: $\text{LEN} \geqslant 0$, $\text{FROMPOS+LEN} \leqslant \text{BIT_SIZE(FROM)}$, $\text{FROMPOS} \geqslant 0$, $\text{TOPOS} \geqslant 0$ e $\text{TOPOS+LEN} \leqslant \text{BIT_SIZE(TO)}$. TO é um inteiro com intent INOUT; ele deve ser da mesma espécie que FROM. A mesma variável pode ser especificada para FROM e TO.

8.9.5 COMPARAÇÃO BIT A BIT (*bitwise unsigned comparison*)

Quatro funções elementais executam comparações bit a bit (*bitwise*), retornando um resultado lógico padrão. Comparações bit a bit tratam valores inteiros como inteiros **sem sinal** (*unsigned*); isto é, o bit mais significante não é tratado como um bit de sinal mas como tendo o valor de 2^{b-1}, sendo b é número de bits no inteiro.

BGE(I, J). Retorna o valor .TRUE. se I é bit a bit maior ou igual a J; retorna .FALSE. em caso contrário.

BGT(I, J). Retorna o valor .TRUE. se I é bit a bit maior que J; retorna .FALSE. em caso contrário.

BLE(I, J). Retorna o valor .TRUE. se I é bit a bit menor ou igual a J; retorna .FALSE. em caso contrário.

BLT(I, J). Retorna o valor .TRUE. se I é bit a bit menor que J; retorna .FALSE. em caso contrário.

Os argumentos I e J devem ambos ser ou do tipo inteiro ou constantes boz; se forem inteiros, não precisam ter a mesma espécie.

8.9.6 Deslocamento de dupla largura (*double-width shifting*)

Duas funções elementais fornecem deslocamento de dupla largura. Estas funções concatenam `I` e `J` e deslocam o valor combinado para a esquerda ou direita por `SHIFT`; o resultado é a metade mais significante para um deslocamento à esquerda ou o menos significante para uma deslocamento à direita.

`DSHIFTL(I, J, SHIFT).` Retorna a metade mais significante de um deslocamento para a esquerda de dupla largura.

`DSHIFTR(I, J, SHIFT).` Retorna a metade menos significante de um deslocamento para a direita de dupla largura.

Um dos argumentos `I` ou `J` deve ser inteiro. O outro ou é inteiro da mesma espécie ou é uma constante literal `boz`, a qual será convertida para inteiro. O resultado é inteiro da mesma espécie. O argumento `SHIFT` deve ser inteiro de qualquer espécie.

8.9.7 Reduções bit a bit (*bitwise reductions*)

Três funções transformacionais executam reduções bit a bit para reduzir uma matriz inteira por um posto ou a um escalar. Estas são as funções `IALL`, `IANY` e `IPARITY`, discutidas na seção 8.12.1.

8.9.8 Contagem de bits

As funções abaixo executam contagem de bits em um inteiro.

`LEADZ(I).` Retorna o número de bits zero mais significantes em `I`.

`POPCNT(I).` Retorna o número de bit distintos de zero em `I`.

`POPPAR(I).` Retorna o valor `1` se `POPCNT(I)` é ímpar ou `0` em caso contrário.

`TRAILZ(I).` Retorna o número de bits zero menos significantes em `I`.

O argumento `I` é um inteiro de qualquer espécie e o resultado é um inteiro padrão.

8.9.9 Produzindo máscaras de bits (*bitmasks*)

Funções elementais que produzem máscaras de bits:

`MASKL(I[, KIND]).` Retorna um inteiro com os `I` bits mais à esquerda dados e o resto zero.

`MASKR(I[, KIND]).` Retorna um inteiro com os `I` bits mais à direita dados e o resto zero.

O resultado é inteiro com a espécie dada por `KIND` (ou inteiro padrão). O argumento `I` deve ser inteiro de qualquer espécie e com valor no intervalo $0 \leqslant$ `I` $\leqslant b$, onde b é o tamanho em bits do resultado. Por exemplo, se `INTEGER(INT8)` denota inteiros de 8 bits, `MASKL(3,INT8)` retorna `INT(b'11100000',INT8)` e `MASKR(3,INT8)` retorna `7_INT8`.

8.9.10 Fusão de bits

Uma função elemental funde bits oriundos de inteiros distintos:

`MERGE_BITS(I, J, MASK).` Retorna os bits de `I` e `J` fundidos sob o controle de `MASK`. Os argumentos `I`, `J` e `MASK` devem ser inteiros da mesma espécie ou constantes `boz`. Ou `I` ou `J` deve ser inteiro e a constante `boz` é convertida a inteiro. O resultado é do tipo inteiro com a mesma espécie.

8.10 FUNÇÃO DE TRANSFERÊNCIA

A função de transferência permite que dados de um tipo sejam transferidos a outro tipo sem que a representação física dos mesmos seja alterada. Esta função pode ser útil, por exemplo, ao se escrever um sistema de armazenamento e recuperação de dados; o sistema pode ser escrito para uma dado tipo, por exemplo, inteiro, e os outros tipos são obtidos através de transferências de e para este tipo.

TRANSFER(SOURCE, MOLD[, SIZE]). Retorna um valor do tipo e espécie de MOLD. Quando SIZE estiver ausente, o resultado é escalar se MOLD for escalar e é de posto 1 e tamanho suficiente para armazenar toda a SOURCE se MOLD for uma matriz. Quando SIZE estiver presente, o resultado é de posto 1 e tamanho SIZE. Se a representação física do resultado é tão ou mais longa que o tamanho de SOURCE, o resultado contém SOURCE como sua parte inicial e o resto é indefinido; em outros casos, o resultado é a parte inicial de SOURCE. Como o posto do resultado depende se SIZE é ou não especificado, o argumento em si não pode ser um argumento opcional.

8.11 FUNÇÕES DE MULTIPLICAÇÃO VETORIAL OU MATRICIAL

Há duas funções transformacionais que realizam multiplicações vetorial ou matricial. Elas possuem dois argumentos que são ambos de um tipo numérico (inteiro, real ou complexo) ou ambas do tipo lógico. O resultado é do mesmo tipo e espécie como seria resultante das operações de multiplicação ou .AND. entre dois escalares. As funções SUM e ANY, usadas nas definições abaixo, são definidas na seção 8.12.1 abaixo.

DOT_PRODUCT(VETOR_A, VETOR_B). Requer dois argumentos, ambos de posto 1 e do mesmo tamanho. Se VETOR_A é dos tipos inteiro ou real, a função retorna SUM(VETOR_A*VETOR_B); se VETOR_A é do tipo complexo, a função retorna SUM(CONJG(VETOR_A)*VETOR_B); e se VETOR_A é do tipo lógico, a função retorna ANY(VETOR_A .AND. VETOR_B).

MATMUL(MATRIZ_A, MATRIZ_B). Realiza a multiplicação escalar. Para argumentos numéricos, três casos são possíveis:

1. MATRIZ_A tem forma [n,m] e MATRIZ_B tem forma [m,k]. O resultado tem forma [n,k] e o elemento (i, j) tem o valor dado por SUM(MATRIZ_A(I,:)*MATRIZ_B(:,J)).
2. MATRIZ_A tem forma [m] e MATRIZ_B tem forma [m,k]. O resultado tem forma [k] e o elemento j tem o valor dado por SUM(MATRI_A*MATRIZ_B(:,J)).
3. MATRIZ_A tem forma [n,m] e MATRIZ_B tem forma [m]. O resultado tem forma [n] e o elemento i tem o valor dado por SUM(MATRIZ_A(I,:)*MATRIZ_B).

Para argumentos lógicos, as formas são as mesmas que para argumentos numéricos e os valores são determinados pela substituição da função SUM e do operador "*" pela função ANY e pelo operador .AND..

8.12 FUNÇÕES TRANSFORMACIONAIS QUE REDUZEM MATRIZES

Há doze funções transformacionais que executam operações em matrizes, tais como somar seus elementos.

8.12.1 CASO DE ARGUMENTO ÚNICO

Nas suas formas mais simples, estas funções têm um único argumento matricial e o resultado é um valor escalar. Todas, exceto COUNT tem o resultado do mesmo tipo e espécie que o argumento.

ALL(MASK). Retorna o valor .TRUE. se todos os elementos da matriz lógica MASK forem verdadeiros ou se MASK for nula; caso contrário, retorna o valor .FALSE.

ANY(MASK). Retorna o valor .TRUE. se algum dos elementos da matriz lógica MASK for verdadeiro e o valor .FALSE. se todos os elementos forem falsos ou se a matriz for nula.

COUNT(MASK[, DIM][, KIND]). Retorna o valor inteiro padrão igual ao número de elementos da matriz MASK que têm o valor .TRUE.. O argumento opcional DIM é descrito abaixo. O argumento opcional KIND especifica a espécie do resultado, se presente.

IALL(ARRAY). Retorna um valor inteiro no qual cada bit é 1 se todos os bits correspondentes dos elementos da matriz inteira ARRAY forem 1, ou 0 em caso contrário.

IANY(ARRAY). Retorna um valor inteiro no qual cada bit é 1 se qualqer um dos bits correspondentes dos elementos da matriz inteira ARRAY forem 1, ou 0 em caso contrário.

IPARITY(ARRAY). Retorna um valor inteiro no qual cada bit é 1 se o número dos bits correspondentes dos elementos da matriz inteira ARRAY forem 1, ou 0 em caso contrário.

MAXVAL(ARRAY). Retorna o valor máximo dentre os elementos da matriz ARRAY, a qual pode ser inteira ou real. Se ARRAY for nula, a função retorna o valor negativo de maior magnitude suportado pelo processador.

MINVAL(ARRAY). Retorna o valor mínimo dentre os elementos da matriz ARRAY, a qual pode ser inteira ou real. Se ARRAY for nula, a função retorna o valor positivo de maior magnitude suportado pelo processador.

NORM2(X). Retorna a norma L_2 de uma matriz real X, isto é, a raiz quadrada da soma dos quadrados de seus elementos. A função retorna 0 se X tem tamanho zero.

PARITY(MASK). Retorna o valor .TRUE. se um número ímpar de elementos da matriz lógica MASK forem .TRUE., e .FALSE. em caso contrário.

PRODUCT(ARRAY). Retorna o produto de todos os elementos de uma matriz inteira, real ou complexa. A função retorna o valor 1 se ARRAY for nula.

SUM(ARRAY). Retorna a soma de todos os elementos de uma matriz inteira, real ou complexa. A função retorna o valor 0 se ARRAY for nula.

8.12.2 Argumento opcional DIM

Todas as funções acima exceto COUNT têm um segundo argumento DIM, o qual é um inteiro escalar. A função COUNT tem um segundo argumento opcional DIM. Se este argumento está presente, a operação é aplicada a todas as seções de posto 1 que varrem através da dimensão DIM para produzir uma matriz de posto reduzido por um e extensões iguais às extensões nas outras dimensões. Por exemplo, se A é uma matriz real de forma [4,5,6], SUM(A,DIM=2) é uma matriz real de forma [4,6] e o seu elemento (i, j) tem o valor dado por SUM(A(I,:,J)).

Uma vez que o posto do resultado de COUNT depende se DIM é especificado ou não (exceto se o posto original é 1), o argumento real da função não pode ser um argumento mudo opcional (seção 9.3.4), um ponteiro desassociado ou um objeto alocável não alocado.

8.12.3 Argumento opcional MASK

As funções IALL, IANY, IPARITY, MAXVAL, MINVAL, PRODUCT e SUM têm um terceiro argumento opcional: uma matriz lógica MASK. Se esta matriz estiver presente, ela deve ter a mesma forma que o primeiro argumento e a operação é aplicada aos elementos do primeiro argumento correspondentes aos elementos verdadeiros de MASK. Por exemplo, SUM(A, MASK= A > 0) soma todos os elementos positivos da matriz A. O argumento MASK afeta somente o valor da função e não afeta o desenvolvimento de argumentos que são expressões matriciais. O argumento MASK pode ser o segundo argumento posicional quando DIM estiver ausente.

8.12.4 REDUÇÃO GENERALIZADA DE MATRIZ

A função intrínseca REDUCE() realiza a redução de uma matriz de uma maneira determinada pelo programador.

REDUCE(ARRAY, OPERATION[, MASK, IDENTITY, ORDERED]) ou

REDUCE(ARRAY, OPERATION, DIM[, MASK, IDENTITY, ORDERED])

ARRAY é uma matriz de qualquer tipo.

OPERATION é uma função pura que fornece a operação binária para a redução da matriz. É recomendado que a operação seja matematicamente associativa. A função deve ter dois argumentos escalares e um resultado escalar, todos não polimórficos[3] e com o mesmo tipo e parâmetros de tipo de ARRAY. Os argumentos não devem possuir os atributos ALLOCATABLE, OPTIONAL ou POINTER. Se um dos argumentos possuir algum dos atributos ASYNCHRONOUS, TARGET ou VOLATILE, o outro argumento deve possuir o(s) mesmo(s) atributo(s).

DIM é um escalar inteiro com valor $1 \leqslant \texttt{DIM} \leqslant n$, onde n é o posto de ARRAY.

MASK é uma matriz lógica conformável com ARRAY.

IDENTITY é um escalar com os mesmo tipo e parâmetros de tipo de ARRAY.

ORDERED é um escalar lógico.

A primeira forma da função retorna um escalar com o mesmo tipo e parâmetros de tipo de ARRAY. O valor do resultado é obtido ao se tomar a sequência de elementos de ARRAY na ordem dos elementos de matriz e repetidamente combinar valores adjacentes da sequência pelas regras fornecidas por OPERATION até restar um único valor final. Se ORDERED está presente com o valor .TRUE., o primeiro valor da sequência é repetidamente combinado com o próximo; em caso contrário, quaisquer valores adjacentes podem ser combinados. Se MASK está presente, a sequência inicial consiste somente daqueles elementos de ARRAY para os quais MASK= .TRUE.. Se a sequência inicial resulta vazia, o resultado é IDENTITY, caso esteja presente; em caso contrário, um processo de encerramento por erro é iniciado.

Na segunda forma, se ARRAY tem posto um, o resultado é o mesmo da primeira forma. Caso contrário, o resultado é uma matriz de posto uma vez menor que ARRAY, e com a forma de ARRAY com a dimensão DIM removida, isto é,

```
[ SIZE(ARRAY,1)...SIZE(ARRAY,DIM-1),SIZE(ARRAY,DIM+1)...SIZE(ARRAY,n) ]
```

Cada elemento $(i_1, \ldots, i_{\texttt{DIM}-1}, i_{\texttt{DIM}+1}, \ldots, i_n)$ do resultado tem o valor obtido pela aplicação de REDUCE à seção $\texttt{ARRAY}(i_1, \ldots, i_{\texttt{DIM}-1}, :, i_{\texttt{DIM}+1}, \ldots, i_n)$.

8.13 FUNÇÕES INQUIRIDORAS DE MATRIZES

Há cinco funções que inquirem sobre a contiguidade e os limites, forma e tamanho de uma matriz de qualquer tipo. Uma vez que o resultado depende somente nas propriedades da matriz, o valor desta não necessita ser definido.

8.13.1 CONTIGUIDADE

Testa a contiguidade dos elementos de uma matriz.

IS_CONTIGUOUS(ARRAY). Sendo ARRAY uma matriz de qualquer tipo, retorna um escalar lógico padrão com o valor .TRUE. se ARRAY é contíguo, ou .FALSE. em caso contrário. Se ARRAY for um ponteiro, deve estar associado a um alvo.

[3] Caso seja empregada programação orientada a objetos.

8.13.2 LIMITES, FORMA E TAMANHO

As funções a seguir inquirem sobre as propriedades de uma matriz. No caso de uma matriz alocável, esta deve estar alocada. No caso de um ponteiro, ele deve estar associado a um alvo. Uma seção de matriz ou uma expressão matricial é assumida ter limite inferior 1 e limite superior igual às extensões. Se uma dimensão tem tamanho zero, o limite inferior é tomado 1 enquanto que o limite superior é tomado 0.

LBOUND(ARRAY[, DIM]). Quando DIM é ausente, retorna uma matriz inteira padrão de posto um, a qual contém os limites inferiores. Quando DIM é presente, este deve ser um inteiro escalar e o resultado é um escalar inteiro padrão com o valor do limite inferior na dimensão DIM. Como o posto do resultado depende da presença de DIM, o argumento da função não pode ser, por sua vez, um argumento mudo opcional, um ponteiro desassociado ou um objeto alocável não alocado.

SHAPE(SOURCE). Retorna um vetor inteiro padrão contendo a forma da matriz ou escalar SOURCE. No caso de um escalar, o resultado tem tamanho zero.

SIZE(ARRAY[, DIM]). Retorna um escalar inteiro padrão igual ao tamanho da matriz ARRAY ou a extensão ao longo da dimensão DIM, caso este argumento esteja presente.

UBOUND(ARRAY[, DIM]). Similar a LBOUND exceto que retorna limites superiores.

Estas funções têm um argumento opcional KIND. Este argumento especifica a espécie dos inteiros nos resultados das funções.

8.14 FUNÇÕES DE CONSTRUÇÃO E MANIPULAÇÃO DE MATRIZES

Há oito funções que constroem ou manipulam matrizes de qualquer tipo.

8.14.1 FUNÇÃO ELEMENTAL MERGE

MERGE(TSOURCE, FSOURCE, MASK). Trata-se de uma função elemental. TSOURCE pode ter qualquer tipo e FSOURCE deve ser do mesmo tipo e espécie. MASK deve ser do tipo lógico. O resultado é TSOURCE se MASK é verdadeiro ou FSOURCE no contrário.

8.14.2 AGRUPANDO E DESAGRUPANDO MATRIZES

A função transformacional PACK agrupa dentro de um vetor aqueles elemento de uma matriz que são selecionados por uma matriz lógica conforme e a função transformacional UNPACK executa a operação reversa. Os elementos são tomados na ordem dos elementos das matrizes.

PACK(ARRAY, MASK[, VECTOR]). Quando VECTOR é ausente, retorna um vetor contendo os elementos de ARRAY correspondentes aos valores verdadeiros de MASK na ordem dos elementos das matrizes. MASK pode ser um escalar de valor .TRUE.; em cujo caso, todos os elementos são selecionados. Se VECTOR é presente, este deve ser um vetor do mesmo tipo e espécie de ARRAY e tamanho no mínimo igual ao número t de elementos selecionados. O resultado, neste caso, tem tamanho igual ao tamanho n do VECTOR; se $t < n$, elementos i do resultado para $i > t$ são os elementos correspondentes de VECTOR.

UNPACK(VECTOR, MASK, FIELD). Retorna uma matriz do tipo e espécie de VECTOR e da forma de MASK. MASK deve ser uma matriz lógica e VECTOR deve ser um vetor de tamanho no mínimo igual ao número de elementos verdadeiros de MASK. FIELD deve ser do mesmo tipo e espécie de VECTOR e deve ou ser escalar ou ter a mesma forma que MASK. O elemento do resultado correspondendo ao i-ésimo elemento verdadeiro de MASK, na ordem dos elementos da matriz, é o i-ésimo elemento de VECTOR; todos os outros são iguais aos correspondentes elementos de FIELD se este for uma matriz ou igual a FIELD se este for um escalar.

8.14.3 ALTERANDO A FORMA DE UMA MATRIZ

A função transformacional RESHAPE permite que a forma de uma matriz seja alterada, com a possível permutação dos índices.

RESHAPE(SOURCE, SHAPE[, PAD][, ORDER]). Retorna uma matriz com forma dada pelo vetor inteiro SHAPE e tipo e espécie iguais aos da matriz SOURCE. O tamanho de SHAPE deve ser constante e seus elementos não podem ser negativos. Se PAD está presente, esta deve ser uma matriz do mesmo tipo e espécie de SOURCE. Se PAD estiver ausente ou for uma matriz de tamanho zero, o tamanho do resultado não deve exceder o tamanho de SOURCE. Se ORDER estiver ausente, os elementos da matriz resultante, arranjados na ordem de elementos de matrizes (seção 6.6.2), são os elementos de SOURCE, seguidos por cópias de PAD, também na ordem de elementos de matrizes. Se ORDER estiver presente, este deve ser um vetor inteiro com um valor que é uma permutação de (1, 2, ..., *n*); os elementos R($s_1, \ldots, s_n$) do resultado, tomados na ordem dos índices para a matriz tendo elementos R($s_{\text{ordem}(1)}, \ldots, s_{\text{ordem}(n)}$), são aqueles de SOURCE na ordem de elementos de matriz seguidos por cópias de PAD, também na ordem de elementos de matriz. Por exemplo, se ORDER tem o valor (/3,1,2/), os elementos R(1,1,1), R(1,1,2), ..., R(1,1,*k*), R(2,1,1), R(2,1,2), ... correspondem aos elementos de SOURCE e PAD na ordem de elementos de matriz.

8.14.4 FUNÇÃO TRANSFORMACIONAL PARA DUPLICAÇÃO

SPREAD(SOURCE, DIM, NCOPIES). Retorna uma matriz do tipo e espécie de SOURCE e de posto acrescido por um. SOURCE pode ser escalar ou matriz. DIM e NCOPIES são inteiros escalares. O resultado contém MAX(NCOPIES,0) cópias de SOURCE e o elemento $(r_1, \ldots, r_{n+1})$ do resultado é SOURCE$(s_1, \ldots, s_n)$, onde $(s_1, \ldots, s_n)$ é $(r_1, \ldots, r_{n+1})$ com subscrito DIM omitido (ou a própria SOURCE se esta for um escalar).

8.14.5 FUNÇÕES DE DESLOCAMENTO MATRICIAL

CSHIFT(ARRAY, SHIFT[, DIM]). Retorna uma matriz do mesmo tipo, espécie e forma de ARRAY. DIM é um escalar inteiro. Se DIM for omitido, o seu valor será assumido igual a 1. SHIFT é do tipo inteiro e deve ser um escalar se ARRAY tiver posto um. Se SHIFT é escalar, o resultado é obtido deslocando-se toda seção vetorial que se estende ao longo da dimensão DIM de forma circular SHIFT vezes. O sentido do deslocamento depende do sinal de SHIFT e pode ser determinado a partir do caso com SHIFT= 1 e ARRAY de posto um e tamanho *m*, quando o elemento *i* do resultado é ARRAY(*i* + 1), *i*= 1, 2, ..., *m*-1 e o elemento *m* é ARRAY(1). Se SHIFT for uma matriz, esta deve ter a forma de ARRAY com a dimensão DIM omitida; desta forma, SHIFT oferece um valor distinto para cada deslocamento.

EOSHIFT(ARRAY, SHIFT[, BOUNDARY][, DIM]). É idêntica a CSHIFT, exceto que um deslocamento final é executado e valores de borda são inseridos nas lacunas assim criadas. BOUNDARY pode ser omitida quando ARRAY tiver tipo intrínsecos, em cujo caso o valor zero é inserido para os casos inteiro, real e complexo; .FALSE. no caso lógico e brancos no caso de caracteres. Se BOUNDARY estiver presente, deve ser do mesmo tipo e espécie de ARRAY; ele pode ser um escalar e prover todos os valores necessários ou pode ser uma matriz cuja forma é a de ARRAY com dimensão DIM omitida e prover um valor separado para cada deslocamento.

8.14.6 TRANSPOSTA DE UMA MATRIZ

A função TRANSPOSE executa a transposição matricial para qualquer matriz de posto dois.

TRANSPOSE(MATRIX). Retorna uma matriz do mesmo tipo e espécie da matriz de posto dois MATRIX. Uma matriz de forma [m,n] passa a ter a forma [n,m]. Se MATRIX tem limites inferiores $lb1$ e $lb2$ (resultantes de LBOUND(MATRIX,1) e LBOUND(MATRIX,2)), o elemento (i, j) do resultado é MATRIX$(j + lb1 - 1, i + lb2 - 1)$.

8.15 FUNÇÕES TRANSFORMACIONAIS PARA LOCALIZAÇÃO GEOMÉTRICA

Há duas funções transformacionais que localizam as posições dos valores máximos e mínimos de uma matriz inteira, real ou de caracteres. Cada uma possui um argumento opcional KIND que determina a espécie do resultado, se presente.

Elas também possuem o argumento opcional BACK (lógico escalar) que indica se a primeira ou a última ocorrência é desejada. Por exemplo, MAXLOC([1,4,4,1]) retorna 2, enquanto que MAXLOC([1,4,4,1], BACK= .TRUE.) retorna 3.

MAXLOC(ARRAY[, MASK][, KIND][, BACK]). Retorna um vetor inteiro padrão de tamanho igual ao posto de ARRAY. Seu valor é a sequência de subscritos de um elemento de valor máximo (dentre aqueles correspondentes aos elementos de valor .TRUE. da matriz lógica MASK, caso esta exista), como se todos os limites inferiores de ARRAY fossem iguais a 1. Se houver mais de um elemento com o mesmo valor máximo, o primeiro na ordem de elementos de matriz é assumido. Se não há elementos, o resultado tem todos os elementos zero.

MAXLOC(ARRAY, DIM[, MASK][, KIND][, BACK]). Retorna um vetor inteiro padrão de forma igual à forma de ARRAY com a dimensão DIM omitida, sendo DIM um inteiro escalar de valor no intervalo $1 \leqslant$ DIM $\leqslant$ RANK(ARRAY). O valor de cada elemento do resultado é a posição do primeiro elemento de valor máximo na correspondente seção vetorial de ARRAY variando ao longo da dimensão DIM, dentre aqueles correspondentes aos elementos de valor .TRUE. da matriz lógica MASK, caso esta exista. O compilador pode distinguir sem auxílio caso o segundo argumento seja DIM ou MASK pelo fato de que o tipo de variável é distinto. Se não há elementos, o resultado tem todos os elementos zero.

MINLOC(ARRAY[, MASK][, KIND][, BACK]). Idêntica a MAXLOC(ARRAY [, MASK]), exceto que agora é obtida a posição do elemento de menor valor.

MINLOC(ARRAY, DIM[, MASK][, KIND][, BACK]). Idêntica a MAXLOC(ARRAY, DIM [, MASK]), exceto que agora são obtidas as posições dos elementos de valor mínimo.

Para uma matriz de qualquer tipo intrínseco, a função transformacional FINDLOC retorna a posição de um elemento com o valor especificado, ou zero se tal elemento não existe. Suas formas são as seguintes:

FINDLOC(ARRAY, VALOR[, MASK][, KIND][, BACK]). Procura em toda a matriz ARRAY, possivelmente sob a orientação de MASK, pelo VALOR, e retorna o vetor de subscritos contendo a posição deste elemento. Os argumentos de ARRAY devem ser do tipo intrínseco e VALOR deve ser um escalar de tipo e espécie compatíveis. Caso presente, MASK deve ser do tipo lógico e conformável com ARRAY, KIND deve ser uma expressão constante inteira escalar e BACK deve ser um escalar lógico. Se BACK está presente e é .TRUE., a função tenta localizar a última ocorrência de VALOR em ARRAY; em caso contrário, tenta encontrar a primeira ocorrência de VALOR.

FINDLOC(ARRAY, VALOR, DIM[, MASK][, KIND][, BACK]). Reduz a dimensão DIM de ARRAY, com o resultado sendo a posição em cada vetor ao longo da dimensão DIM onde o elemento é encontrado. O argumento DIM deve ser um inteiro escalar; os outros argumentos deve ser como acima.

Por exemplo, FINDLOC([(I, I= 10, 1000, 10)], 470) tem como resultado 47.

Qualquer tipo intrínseco pode ser empregado; para o tipo lógico, a operação .EQV. é empregada para comparações.

8.16 FUNÇÃO TRANSFORMACIONAL PARA DESASSOCIAÇÃO OU DEALOCAÇÃO

A função NULL está disponível para fornecer status de desassociado a ponteiros ou não alocado a um objeto alocável.

NULL([MOLDE]). Retorna um ponteiro desassociado. O argumento MOLDE é um ponteiro de qualquer tipo e que pode ter qualquer status, inclusive de indefinido. O tipo, espécie e posto do resultado são aqueles de MOLDE se ele está presente; em caso contrário, são aqueles do objeto com o qual o ponteiro está associado. Em um argumento real associado com um argumento mudo de uma variável de caractere de comprimento assumido, MOLDE deve estar presente.

8.17 SUBROTINAS INTRÍNSECAS NÃO-ELEMENTAIS

Há diversas sub-rotinas intrínsecas não elementais, as quais foram escolhidas ser sub-rotinas no lugar de funções devido à necessidade destas retornarem diversas informações através de seus argumentos.

8.17.1 RELÓGIO DE TEMPO REAL

Há duas sub-rotinas que retornam informação acerca do relógio de tempo real; a primeiro é baseada no padrão ISO 8601 (Representação de datas e tempos).[4] Assume-se que exista um relógio básico no sistema que é incrementado por um para cada contagem de tempo até que um valor máximo COUNT_MAX é alcançado e daí, na próxima contagem de tempo, este é zerado. Valores padronizados são retornados em sistemas sem relógio. Todos os argumentos têm intenção OUT (seção 9.3.2). Adicionalmente, há uma sub-rotina que acessa o relógio interno do processador.

Acesso a qualquer argumento opcional pode ser realizado através do uso das palavras-chave (seções 9.3.3 e 9.3.4).

CALL DATE_AND_TIME([DATE][, TIME][, ZONE][, VALUES]). Retorna os seguintes valores (com valores-padrão nulos ou -HUGE(0), conforme apropriado, caso não haja relógio):

DATE é uma variável escalar de caractere da espécie padrão que contém a data na forma *ccyymmdd*, correspondendo ao século, ano, mês e dia, respectivamente.

TIME é uma variável escalar de caractere padrão que contém o tempo na forma *hhmmss.sss*, correspondendo a horas, minutos, segundos e milisegundos, respectivamente.

ZONE é uma variável escalar de caractere padrão que é fixada como a diferença entre o tempo local e o Tempo Coordenado Universal (UTC, de *Coordinated Universal Time*, também conhecido como Tempo Médio de Greenwich, ou *Greenwich Mean Time*) na forma *Shhmm*, correspondendo ao sinal, horas e minutos, respectivamente. Por exemplo, a hora local de Brasília corresponde a UTC= -0300.

VALUES é um vetor inteiro que contém a seguinte sequência de valores: o ano, o mês do ano, o dia do mês, a diferença temporal em minutos com relação ao UTC, a hora do dia, os minutos da hora, os segundos do minuto e os milisegundos do segundo. Os elementos de VALUES podem ser de qualquer espécie determinada por $R \geqslant 4$, sendo R o argumento de SELECTED_INT_KIND(R).

CALL SYSTEM_CLOCK([COUNT][, COUNT_RATE][, COUNT_MAX]). Retorna o seguinte:

COUNT é uma variável escalar inteira que contém o valor, dependente do processador,[5] que correspondem ao valor corrente do relógio do processador, ou -HUGE(0) caso não haja relógio. Na primeira chamada da sub-rotina, o processador pode fixar um valor inicial igual a zero.

COUNT_RATE é uma variável escalar inteira ou real[6] que contém o número de contagens por segundo do relógio, ou zero caso não haja relógio.

COUNT_MAX é uma variável escalar inteira que contém o valor máximo que COUNT pode assumir, ou zero caso não haja relógio.

[4] Ver, por exemplo, https://en.wikipedia.org/wiki/ISO_8601.
[5] Pode ser de qualquer espécie, pois o valor do relógio do sistema pode ser muito grande para um inteiro padrão.
[6] Para o caso em que o relógio do sistema não se altera por um número inteiro de vezes por segundo.

8.17.2 Tempo da CPU

No Fortran, há uma sub-rotina intrínseca não elemental que retorna o tempo do processador.

CALL CPU_TIME(TIME). Retorna o seguinte:

TIME é uma variável real escalar padrão à qual é atribuída uma aproximação (dependente de processador) do tempo do processador em segundos, ou um valor negativo (dependente de processador) caso não haja relógio.

8.17.3 Números aleatórios

Uma sequência de números pseudo-aleatórios é gerada a partir de uma semente que é fornecida como um vetor de números inteiros. A sub-rotina RANDOM_NUMBER retorna os números pseudo-aleatórios e a sub-rotina RANDOM_SEED permite que uma inquirição seja feita a respeito do tamanho ou valor do vetor-semente e, então, redefinir o mesmo. As sub-rotinas fornecem uma interface a uma sequência que depende do processador.

CALL RANDOM_NUMBER(COLHE) Retorna um número pseudo-aleatório a partir da distribuição uniforme de números no intervalo $0 \leqslant x < 1$ ou uma matriz destes números. COLHE tem intenção OUT, pode ser um escalar ou matriz e deve ser do tipo real.

CALL RANDOM_SEED([SIZE][, PUT][, GET]). Tem os argumentos abaixo. Não mais de um argumento pode ser especificado; caso nenhum argumento seja fornecido, a semente é fixada a um valor dependente do processador.

SIZE tem intenção OUT e é um inteiro escalar padrão que o processador fixa com o tamanho n do vetor-semente.

PUT tem intenção IN e é um vetor inteiro padrão de tamanho n que é usado pelo processador para fixar uma nova semente. Um processador pode fixar o mesmo valor de semente para mais de um valor de PUT.

GET tem intenção OUT e é um vetor inteiro padrão de tamanho n que o processador fixa como o valor atual da semente.

Alguns problemas foram observados com o uso da rotina RANDOM_NUMBER:

- Alguns processadores sempre inicializam a semente pseudo-aleatório da mesma maneira, resultando que a mesma sequência de números pseudo-aleatórios é gerada cada vez que a rotina é invocada durante a mesma execução do programa. Outros processadores inicializam a semente de forma aleatória.
- Empregando coarrays, cada imagem pode ter a sua própria semente ou podem empregar uma semente comum.

Estas diferenças são às vezes observadas mesmo em diferentes compiladores gerando executáveis para o mesmo processador. A rotina RANDOM_SEED pode ser empregada para forçar que uma nova semente seja empregada a cada vez que RANDOM_NUMBER é invocado.

A rotina RANDOM_INIT permite inicialização da semente de maneira mais controlada entre as diferentes imagens do código.

CALL RANDOM_INIT(REPEATABLE, IMAGE_DISTINCT). Controla a inicialização do gerador de números pseudo-aleatórios.

REPEATABLE é um escalar lógico com intenção IN. Se o valor é .TRUE., a semente é levada a um valor dependente do processador que é o mesmo cada vez que RANDOM_INIT é invocada dentro de um dado ambiente de execução a partir de uma imagem com o mesmo índice de imagem no time inicial.[7] Se o seu valor é .FALSE., a semente é levada a um valor dependente de processador em cada invocação.

IMAGE_DISTINCT é um escalar lógico com intenção IN. Se o valor é .TRUE., a semente é levada a um valor dependente de processador que é distinto do valor que seria assumido por uma chamada de RANDOM_INIT em outra imagem com os mesmos valores no argumento. Se o valor é .FALSE., o valor assumido pela semente não depende de qual image invoca RANDOM_INIT.

[7] *Times* (*teams*) é um recurso relacionado ao uso de coarrays, que não são abordados nesta Apostila.

8.17.4 EXECUTANDO OUTROS PROGRAMAS

Durante a execução de um código gerado com Fortran, é possível invocar outros programas residentes no sistema empregando a sub-rotina EXECUTE_COMMAND_LINE. Como o nome sugere, a rotina transfere uma linha de comando para o sistema que o processador irá interpretar de uma maneira dependente do sistema.

CALL EXECUTE_COMMAND_LINE(COMMAND[, WAIT][, EXITSTAT][, CMDSTAT][, CMDMSG]). Gera uma linha de comando para ser executada pelo sistema.

COMMAND é uma string escalar da espécie padrão com intenção IN contendo a linha de comando a ser submetida ao sistema operacional.

WAIT é um escalar lógico com intenção IN que indica: se WAIT= .FALSE., o comando deve ser executado de forma asíncrona; se WAIT= .TRUE. (valor padrão), o código deve esperar o término da execução do comando antes de prosseguir com o processamento.

EXITSTAT é uma variável inteira com intenção INOUT que, exceto se WAIT= .FALSE., terá o seu valor determinado pelo *status de término de processo* gerado após a execução da linha de comando. O valor do status depende do sistema. EXITSTAT pode ser de qualquer espécie determinada por $R \geqslant 9$, sendo R o argumento de SELECTED_INT_KIND(R).

CMDSTAT é uma variável inteira com intenção OUT que assume valor zero se o comando EXECUTE_COMMAND_LINE é executado sem erro, -1 se o processador não suporta a linha de comando, -2 se WAIT= .TRUE. foi especificado mas o processador não suporta execução de comandos de forma asíncrona ou um valor positivo se algum outro erro ocorrer. CMDSTAT pode ser de qualquer espécie determinada por $R \geqslant 4$, sendo R o argumento de SELECTED_INT_KIND(R).

CMDMSG é uma string escalar padrão com intenção INOUT a qual recebe uma descrição do erro ocorrido, caso CMDSTAT receba um valor positivo.

Caso algum erro ocorra e CMDSTAT não esteja presente, o código tem sua execução interrompida.

Por exemplo, a instrução

```
CALL EXECUTE_COMMAND_LINE('ls -l')
```

deve gerar, na saída padrão, a listagem de arquivos existentes no diretório de trabalho, em estações operando com sistemas Unix/Linux. Caso o hardware esteja executando outro sistema operacional, este comando deve gerar uma mensagem de erro.

8.18 ACESSO AO AMBIENTE COMPUTACIONAL

Existem rotinas intrínsecas que acessam informações a respeito do ambiente computacional ou alteram o seu status. No padrão atual, as rotinas retornam os valores das variáveis de ambiente do sistema e informações acerca da linha de execução do código.

8.18.1 VARIÁVEIS DE AMBIENTE

Quando o sistema operacional tem o conceito de *variáveis de ambiente* (*environment variables*), a sub-rotina abaixo acessa os seus valores atuais:

CALL GET_ENVIRONMENT_VARIABLE(NAME[, VALUE][, LENGTH][, STATUS][, TRIM_NAME][, ERRMSG]).

Retorna o valor de uma variável de ambiente.

NAME é uma string padrão com intenção IN que contém o nome da variável de ambiente cujo valor é requerido. Espaços em branco ao final da string não são significativos, exceto se TRIM_NAME está presente e tem valor .FALSE.. A sintaxe da variável de ambiente dependende do sistema operacional.

VALUE é uma string padrão com intenção OUT que recebe o valor da variável de ambiente. A string pode resultar com o valor truncado (se o seu comprimento for insuficiente) ou ter espaços em branco ao final (caso o comprimento seja maior que o necessário). Caso a variável não exista, ou a mesma não possua valor ou o sistema não suporta variáveis de ambiente, o seu valor somente possuirá brancos.

LENGTH é um inteiro escalar com intenção OUT. Se a variável de ambiente existe e possui valor, LENGTH receberá o seu comprimento; em outra circunstância, LENGTH terá valor nulo. LENGTH pode ser de qualquer espécie determinada por $R \geqslant 4$, sendo R o argumento de SELECTED_INT_KIND(R).

STATUS é um inteiro escalar com intenção OUT. O seu valor será: 1 se a variável de ambiente não existe, 2 se o sistema não suporta variáveis de ambiente, um valor maior que 2 se algum erro ocorrer, -1 se o argumento VALUE está presente mas o seu comprimento é insuficiente e zero nenhuma condição de erro ou aviso ocorrer. STATUS pode ser de qualquer espécie determinada por $R \geqslant 4$, sendo R o argumento de SELECTED_INT_KIND(R).

TRIM_NAME é uma variável lógica escalar com inteção IN. Se TRIM_NAME= .FALSE., espaços em branco em NAME serão considerados significantes se o sistema permitir que variáveis de ambiente contenham os mesmos.

ERRMSG é uma string escalar padrão com intenção INOUT que recebe uma mensagem de erro caso o mesmo ocorra. Caso STATUS= -1 ocorra, a string não recebe valor.

8.18.2 Informações acerca da invocação do programa

Existem três rotinas complementares (uma função e duas sub-rotinas) que acessam informações acerca do comando que disparou a execução do código.

As duas rotinas abaixo são típicas de sistemas Unix/Linux:

COMMAND_ARGUMENT_COUNT(). Retorna um inteiro escalar padrão igual ao número de argumentos do comando. Se o resultado é zero, ou não existem argumentos ou o sistema não suporta este recurso. Se o nome do comando está disponível como um argumento, o mesmo não é incluído no resultado.

CALL GET_COMMAND_ARGUMENT(NUMBER[, VALUE][, LENGTH][, STATUS][, ERRMSG]). Retorna diversas informações acerca da linha de comando.

NUMBER é um inteiro escalar com intenção IN indicando o número do argumento a ser retornado. Se o nome do comando está disponível como um argumento, este será obtido por NUMBER= 0.

VALUE é uma string escalar padrão com intenção OUT que recebe o valor do argumento indicado por NUMBER. O valor de VALUE pode ser truncado (se o seu comprimento for insuficiente) ou preenchido à direita por brancos (caso o argumento seja curto).

LENGTH é um inteiro escalar com inteção OUT que recebe o comprimento do argumento indicado. LENGTH pode ser de qualquer espécie determinada por $R \geqslant 4$, sendo R o argumento de SELECTED_INT_KIND(R).

STATUS é um inteiro escalar com intenção OUT. O seu valor será: positivo se o argumento não pode ser recuperado, -1 para indicar que o comprimento de VALUE foi insuficiente ou zero em outras circunstâncias. STATUS pode ser de qualquer espécie determinada por $R \geqslant 4$, sendo R o argumento de SELECTED_INT_KIND(R).

ERRMSG é uma string escalar padrão com intenção INOUT que recebe uma mensagem de erro caso o mesmo ocorra. Caso STATUS= -1 ocorra, a string não recebe valor.

A próxima sub-rotina é mais adequada para sistemas operacionais diferentes do Unix/Linux:

CALL GET_COMMAND([COMMAND][, LENGTH][, STATUS][, ERRMSG]). Retorna informações sobre a linha de comando.

COMMAND é uma string escalar padrão com intenção OUT que recebe o valor da linha de comando. O valor de COMMAND pode ser truncado (se o seu comprimento for insuficiente) ou preenchido à direita por brancos (para linhas curtas).

LENGTH é um inteiro escalar com intenção OUT que recebe o comprimento da linha de comando ou zero caso o comprimento não possa ser determinado. LENGTH pode ser de qualquer espécie determinada por R $\geqslant 4$, sendo R o argumento de SELECTED_INT_KIND(R).

STATUS é um inteiro escalar com intenção OUT. O seu valor será: positivo se a linha de comando não pode ser recuperada, -1 caso COMMAND esteja presente, mas o seu o comprimento é insuficiente para armazenar toda a linha ou zero em outras circunstâncias. STATUS pode ser de qualquer espécie determinada por R $\geqslant 4$, sendo R o argumento de SELECTED_INT_KIND(R).

ERRMSG é uma string escalar padrão com intenção INOUT que recebe uma mensagem de erro caso o mesmo ocorra. Caso STATUS= -1 ocorra, a string não recebe valor.

8.19 FUNÇÕES ELEMENTAIS PARA TESTE DE STATUS DE ENTRADA/SAÍDA

Há duas funções elementais intrínsecas que testa o status de Entrada/Saída retornado pelo especificador IOSTAT= (seções 10.7 e 10.8). Ambas aceitam um argumento inteiro e retornam um resultado lógico.

IS_IOSTAT_END(I). Retorna o valor .TRUE. se I tem um status E/S que correponda a uma condição de final de arquivo e .FALSE. em caso contrário.

IS_IOSTAT_EOR(I). Retorna o valor .TRUE. se I tem um status E/S que correponda a uma condição de final de registro e .FALSE. em caso contrário.

8.20 ESPAÇO DE MEMÓRIA OCUPADO POR UM OBJETO

STORAGE_SIZE(A[, KIND]). Retorna o tamanho, em bits, que seria ocupado na memória por um elemento de matriz com o tipo dinâmico e parâmetros de tipo do argumento A. O tipo do resultado é inteiro, da espécie indicada por KIND, caso esteja presente; senão será da espécie padrão.

O argumento A pode ser de qualquer posto ou tipo (inclusive um escalar). É permitido que seja um ponteiro indefinido, exceto se for polimórfico.[8] É permitido também que seja um ponteiro desassociado ou um alocável não alocado, exceto se possuir um parâmetro de tipo deferido ou se for ilimitadamente polimórfico.

Observa-se que o valor retornado por STORAGE_SIZE() pode variar dependendo das seguintes circunstâncias:

- Dependendo das opções de compilação que determinam a otimização do código, o tamanho ocupado por um objeto de um determinado tipo e espécie pode variar caso o mesmo seja declarado como uma variável escalar, como elemento de matriz ou como componente de um tipo derivado.
- Se o argumento A for um tipo derivado com componentes alocáveis ou com componentes cujos comprimentos dependem dos valores de parâmetros de comprimento, o compilador pode determinar o armazenamento desses componentes em regiões separadas dos demais componentes. O uso da função STORAGE_SIZE() não é recomendado nesses casos, uma vez que o valor retornado é ambíguo.

8.21 ROTINAS ADICIONAIS

As rotinas abaixo são oferecidas pelo padrão, mas não são descritas neste capítulo por diversas razões.

MOVE_ALLOC. Esta sub-rotina é descrita na seção 7.1.2.1.

[8] Quando são empregados recursos de programação orientada a objetos.

As rotinas a seguir se referem a recursos avançados que não são abordados nesta Apostila:[9]

- Programação orientada a objetos: EXTENDS_TYPE_OF e SAME_TYPE_AS.
- Coarrays:

  IMAGE_INDEX, LCOBOUND, UCOBOUND, NUM_IMAGES e THIS_IMAGE, ATOMIC_DEFINE, ATOMIC_REF, GET_TEAM, TEAM_NUMBER, IMAGE_INDEX, NUM_IMAGES, THIS_IMAGE, COSHAPE, FAILED_IMAGES, STOPPED_IMAGES, IMAGE_STATUS, CO_BROADCAST, CO_MAX, CO_MIN, CO_REDUCE, ATOMIC_ADD, ATOMIC_AND, ATOMIC_OR, ATOMIC_XOR, ATOMIC_FETCH_ADD, ATOMIC_FETCH_AND, ATOMIC_FETCH_OR, ATOMIC_FETCH_XOR, ATOMIC_CAS e EVENT_QUERY.

- Controle de exceções de ponto flutuante:

  IEEE_SUPPORT_FLAG, IEEE_SUPPORT_HALTING, IEEE_GET_FLAG, IEEE_GET_HALTING_MODE, IEEE_SET_FLAG, IEEE_SET_HALTING_MODE, IEEE_GET_STATUS, IEEE_SET_STATUS, IEEE_SUPPORT_DATATYPE, IEEE_SUPPORT_DENORMAL, IEEE_SUPPORT_DIVIDE, IEEE_SUPPORT_INF, IEEE_SUPPORT_IO, IEEE_SUPPORT_NAN, IEEE_SUPPORT_ROUDING, IEEE_SUPPORT_SQRT, IEEE_SUPPORT_STANDARD, IEEE_SUPPORT_UNDERFLOW_CONTROL, IEEE_CLASS, IEEE_COPY_SIGN, IEEE_IS_FINITE, IEEE_IS_NAN, IEEE_IS_NEGATIVE, IEEE_IS_NORMAL, IEEE_LOGB, IEEE_NEXT_AFTER, IEEE_REM, IEEE_RINT, IEEE_SCALB, IEEE_UNORDERED, IEEE_VALUE, IEEE_GET_ROUNDING_MODE, IEEE_GET_UNDERFLOW_MODE, IEEE_SET_ROUNDING_MODE, IEEE_SET_UNDERFLOW_MODE e IEEE_SELECTED_REAL_KIND.

- Interoperabilidade com a linguagem C:[10]

  C_SIZEOF, C_LOC, C_FUNLOC, C_ASSOCIATED, C_F_POINTER, C_F_PROCPOINTER, C_ASSOCIATED, CFI_address, CFI_allocate, CFI_deallocate, CFI_establish, CFI_is_contiguous, CFI_section, CFI_select_part e CFI_setpointer.

[9] Ver também seção 1.9.
[10] As funções escritas em letras minúsculas são funções do C.

9 Unidades de Programa do Fortran

Como foi visto nos capítulos anteriores, é possível escrever um programa completo em Fortran em um único arquivo, ou como uma unidade simples. Contudo, se o código é demasiado complexo, pode ser necessário que um determinado conjunto de instruções seja realizado repetidas vezes, em pontos distintos do programa. Um exemplo desta situação seria a definição de uma função *extrínseca*, ou seja, não contida no conjunto de funções intrínsecas discutidas no capítulo 8. Neste caso, é melhor quebrar o programa em unidades distintas.

Cada uma das *unidades de programa* corresponde a um conjunto completo e consistente de tarefas que podem ser, idealmente, escritas, compiladas e testadas individualmente, sendo posteriormente incluídas no programa principal para gerar um arquivo executável. Em Fortran há dois tipos de estruturas que se encaixam nesta categoria: *sub-rotinas* e *funções* (externas ou extrínsecas).

No padrão do Fortran existem, ao todo, três unidades distintas de programa:

- Programa principal.
- Rotinas externas.
- Módulos.

Cada uma destas unidades de programas serão discutidas nas seções seguintes.

Neste capítulo (e nos demais), os termos **rotina** *(procedure)* ou **subprograma** (*subprogram*) serão usados alternadamente para indicar de forma genérica tanto sub-rotinas quanto funções. Outros textos podem vir a tratar estes dois termos como representando entidades ligeiramente distintas.

9.1 Unidades de programa

Em todos os padrões do Fortran, um código executável é criado a partir de um e somente um *programa principal*, o qual pode invocar *rotinas externas* e pode usar também *módulos.* A única unidade de programa que deve necessariamente existir sempre é o programa principal.

O programa principal ou uma rotina externa podem também invocar *rotinas internas*, as quais têm estrutura semelhante às rotinas externas, porém não podem ser testadas isoladamente. Um módulo, por sua vez, também pode conter rotinas, as quais são neste caso denominadas *rotinas de módulo.* Estas, por sua vez, também podem conter outras rotinas internas. Um diagrama ilustrativo das três unidades de programas existentes no Fortran, juntamente com suas possíveis rotinas internas ou de módulo, pode ser visto na figura 9.1.

Uma descrição mais completa das unidades de programas é realizada nas próximas seções.

9.1.1 Programa principal

Todo código executável deve ser composto a partir de um, e somente um, programa principal. Opcionalmente, este pode invocar subprogramas. Um programa principal possui a seguinte estrutura:

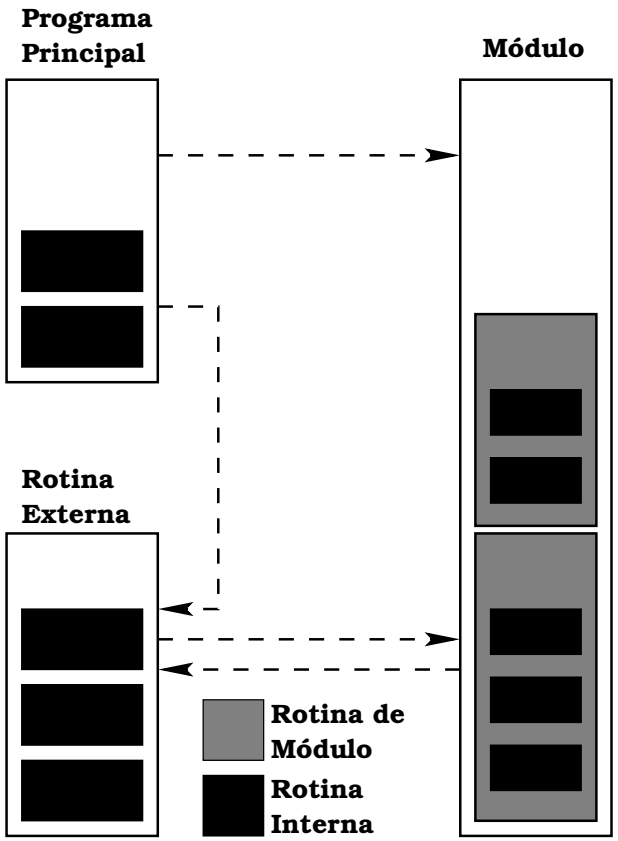


Figura 9.1: As três unidades de programa, as quais podem possuir também rotinas internas. As setas indicam as possíveis referências entre as distintas unidades.

```
[PROGRAM <nome do programa>
   [<declarações de variáveis>]
   [<comandos executáveis>]
[CONTAINS
   <subprogramas internos>]
END [PROGRAM [<nome do programa>]]
```

A declaração `PROGRAM` é opcional, porém o seu uso é recomendado. O único campo não opcional na estrutura de um programa, na definição do padrão da linguagem, é a instrução `END`, a qual possui dois propósitos: indicar ao compilador que o programa chegou ao fim e, quando ocorre a execução do código, provoca a parada do mesmo.

Nos capítulos anteriores, já foram dados vários exemplos da estrutura do programa principal e, portanto, maiores detalhes não são necessários aqui.

A palavra-chave `CONTAINS` indica a presença de um ou mais subprogramas internos. Estes serão descritos em mais detalhes na seção 9.2.2. Se a execução do último comando anterior a `CONTAINS` não provoca um desvio de percurso na execução do programa, o controle passa por sobre os subprogramas internos ao comando `END` e o programa é encerrado.

Sugestões de uso & estilo para programação

A estrutura recomendada para o programa principal é:

```
PROGRAM <nome do programa>
   IMPLICIT NONE
   [<declarações de variáveis>]
   [<comandos executáveis>]
[CONTAINS
   <subprogramas internos>]
END PROGRAM <nome do programa>
```

OS COMANDOS STOP E ERROR STOP

Outras formas de interromper a execução do programa são fornecidas pelos comandos `STOP` e `ERROR STOP`, os quais podem ser rotulados, podem estar contido no interior de um bloco de um construto de controle de fluxo e podem aparecer em qualquer ponto no programa principal ou em um subprograma.

Ambos podem servir como instruções de encerramento do código devido a ocorrência de erros durante a execução do programa, mas o comando `STOP` também é empregado por alguns programadores logo antes da declaração `END PROGRAM` para determinar o encerramento normal do programa. Por outro lado, o comando `ERROR STOP` foi deliberadamente criado para determinar uma *terminação por erro*.

Caso estes comandos estejam distribuídos pelas diferentes unidades que compõe um programa completo, estes podem ser distinguidos por um *código de parada* (*stop code*), o qual pode transferir informações ao sistema operacional acerca do problema que levou à execução do comando.

As sintaxes dos comandos são as seguintes:

```
STOP [<stop-code>][, QUIET= <log-expr>]
ERROR STOP [<stop-code>][, QUIET= <log-expr>]
```

onde `<stop-code>` é qualquer literal ou expressão escalar dos padrões dos tipos inteiro ou de caractere e `<log-expr>` é uma expressão escalar do tipo lógico. Se o especificador `QUIET=` está presente e seu valor é `.TRUE.`, nenhum código de parada é gerado. Alguns exemplos de uso dos comandos são:

```
STOP 'Dados incompletos.  Programa interrompido.'
STOP 12345
STOP MENSAGEM_ERRO, QUIET= SEM_MENSAGENS
```

9.1.2 ROTINAS EXTERNAS

Rotinas externas são chamadas de dentro de um programa principal ou de outra unidade, usualmente com o intuito de executar uma tarefa bem definida dentro dos objetivos cumpridos pelo programa completo. Rotinas externas são escritas em arquivos distintos, separados do arquivo do programa principal, ou no mesmo arquivo, mas são sempre consideradas unidades distintas do programa principal.

Durante o processo de criação do programa executável, estas rotinas podem ser compiladas conjuntamente com o programa principal ou em separado. Nesta última situação, os programas-objeto criados durante compilações prévias das rotinas externas são linkados junto com o programa-objeto do programa principal. Embora seja possível colocar-se mais de uma rotina externa no mesmo arquivo, esta prática não é recomendada.

A estrutura genérica de uma rotina externa será apresentada na seção 9.2.1. Uma rotina externa pode conter uma ou mais rotinas internas, descritas na seção 9.2.2.

9.1.3 MÓDULOS

O terceiro tipo de unidade de programa é o módulo. Trata-se de uma maneira de se agrupar dados globais, tipos derivados e suas operações associadas, blocos de interface, grupos `NAMELIST` (seção 10.2) e rotinas internas. Tudo associado com uma determinada tarefa pode ser coletado dentro de um módulo e acessado quando necessário. Aquelas partes que são restritas ao funcionamento interno da tarefa e não são do interesse direto do programador podem ser feitas "invisíveis" a este, o que permite que o funcionamento interno do módulo possa ser alterado sem a necessidade de alterar o programa que o usa, prevenindo-se assim alteração acidental dos dados.

Um módulo tem a seguinte estrutura:

```
MODULE <nome módulo>
   <declarações de variáveis>
[CONTAINS
   <rotinas de módulo>
END [MODULE [<nome módulo>]]
```

onde <nome módulo> é qualquer nome válido no Fortran. Assim como para END PROGRAM, é recomendado o uso da forma completa do comando END.

Um módulo pode conter <rotinas de módulo>, as quais são subprogramas, como rotinas externas. Para tanto, o código-fonte destas rotinas deve ser precedido dentro do módulo pela palavra-chave CONTAINS. A estrutura de uma rotina de módulo segue a forma genérica que será fornecida na seção 9.2.1. Uma rotina de módulo pode conter uma ou mais rotinas internas, abordadas na seção 9.2.2. Uma rotina de módulo tem acesso automático às demais entidades definidas no módulo, incluindo a habilidade de acessar outros subprogramas definidos no mesmo módulo.

Uma discussão mais extensa acerca dos módulos em Fortran será feita na seção 9.13.

9.2 SUBPROGRAMAS

Subprogramas podem ser *sub-rotinas* ou *funções*. Como já foi mencionado, tarefas que podem ser delimitadas por um número finito de operações e que devem ser realizadas repetidas vezes durante a execução de um programa devem ser preferencialmente escritas como subprogramas.

Uma função retorna um único valor, o qual pode ser um escalar ou uma matriz, e esta usualmente não altera os valores de seus argumentos.[1] Neste sentido, uma função em Fortran age como uma função em análise matemática. Já uma sub-rotina pode executar uma tarefa mais complexa e retornar diversos valores através de seus argumentos, os quais podem ser modificados ao longo da computação da sub-rotina.

9.2.1 FUNÇÕES E SUB-ROTINAS EXTERNAS OU DE MÓDULO

Exceto pela declaração inicial, os subprogramas apresentam uma forma semelhante a de um programa principal. Um subprograma pode ser uma sub-rotina:

```
[<prefixo>] SUBROUTINE <nome subrotina> [(<lista argumentos mudos>)]
   [<declarações de variáveis>]
   [<comandos executáveis>]
[CONTAINS
   <subprogramas internos>]
END [SUBROUTINE [<nome subrotina>]]
```

ou pode ser uma função:

```
[<prefixo>] [<tipo>] FUNCTION <nome função> &
                    (<lista argumentos mudos>)[RESULT (<nome resultado>)]
   [<declarações de variáveis>]
   [<comandos executáveis>]
[CONTAINS
   <subprogramas internos>]
END [FUNCTION [<nome função>]]
```

onde <nome subrotina> ou <nome função> são quaisquer nomes válidos em Fortran. O campo <prefixo> pode conter uma ou mais das palavras-chave

```
ELEMENTAL   NON_RECURSIVE
IMPURE      PURE
MODULE      RECURSIVE
```

Para uma função, o prefixo também pode conter o tipo e espécie do resultado.

Os diversos constituintes das formas gerais apresentadas acima serão discutidos em detalhes nas seções seguintes. Por enquanto, serão apresentados alguns dos atributos usualmente empregados no campo de <declarações de variáveis> de um subprograma. Além dos atributos já vistos, relacionados com as declarações de tipo e espécie de variáveis, dos atributos DIMENSION e ALLOCATABLE, os seguintes atributos podem ser usados:

[1] Situações onde este requisito pode ser relaxado e como estabelecer salvaguardas são discutidas na seção 9.10.

```
INTENT([IN][OUT][INOUT])  EXTERNAL   TARGET
OPTIONAL                  INTRINSIC  VALUE
SAVE                      POINTER    VOLATILE
```

Todos estes atributos, exceto `VALUE`, também podem ser empregados como declarações.

A palavra-chave `CONTAINS` cumpre exatamente o mesmo papel cumprido dentro do programa principal. O efeito do comando `END` em um subprograma consiste em retornar o controle à unidade que o chamou, ao invés de interromper a execução do programa. Aqui também recomenda-se o uso da forma completa do comando para deixar claro ao compilador e ao programador qual parte do programa está sendo terminada.

9.2.1.1 INVOCAÇÕES DE FUNÇÕES

Uma função é *invocada* ou *chamada* de forma semelhante como se usa uma função em análise matemática. Por exemplo, seja a função `BESSEL_JN(n,x)` (seção 8.4), a qual calcula $J_n(x)$, o valor da função de Bessel do primeiro tipo de ordem n no ponto x. Esta função pode ser chamada para atribuir seu valor a uma variável escalar ou a um elemento de matriz:

```
y= BESSEL_JN(n,x)
```

sendo que o tipo e espécie de `y` devem, em princípio, concordar com o tipo e espécie do resultado de `BESSEL_JN(n,x)`. Contudo, caso isto não ocorra, valem as regras de conversão definidas no capítulo 4. Uma função pode também fazer o papel de um operando em uma expressão:

```
y= BESSEL_JN(n,x) + 2*BESSEL_JN(n,x**2)
```

ou ainda servir de argumento para uma outra rotina.

9.2.1.2 INVOCAÇÕES DE SUB-ROTINAS

Por sua vez, na chamada ou invocação de uma sub-rotina, devido ao fato de esta retornar, em geral, mais de um valor em cada chamada, esta não pode ser operada como uma função em análise matemática mas deve, outrossim, ser *chamada* através da instrução `CALL`. Supondo que exista a sub-rotina `BASCARA`, a qual calcula as raízes de um polinômio de segundo grau, `x1` e `x2`, estas serão obtidas através da chamada:

```
CALL BASCARA(A0,A1,A2,X1,X2)
```

onde os argumentos da sub-rotina serão discutidos na seção 9.3 e posteriores.

Por exemplo, é possível usar-se uma função como argumento da sub-rotina:

```
CALL BASCARA(A0,A1,BESSEL_JN(n,x),x1,x2)
```

9.2.2 ROTINAS INTERNAS

Já foi observado nas seções anteriores que rotinas internas podem ser definidas dentro de quaisquer unidades de programa. Uma rotina interna possui a seguinte estrutura:

```
[<prefixo>] SUBROUTINE <nome subrotina> [(<lista argumentos mudos>)]
   [<declarações de variáveis>]
   [<comandos executáveis>]
END [SUBROUTINE [<nome subrotina>]]
```

ou:

```
[<prefixo>] [<tipo>] FUNCTION <nome função> &
                    (<lista argumentos mudos>)[RESULT (<nome resultado>)]
   [<declarações de variáveis>]
   [<comandos executáveis>]
END [FUNCTION [<nome função>]]
```

ou seja, exatamente a mesma estrutura de rotinas externas ou de módulo, abordadas na seção 9.2.1, exceto que uma rotina interna não pode conter outras rotinas internas.

Uma rotina interna automaticamente tem acesso a todas as entidades do hospedeiro, tais como variáveis, e possuem também a habilidade de chamar as outras rotinas deste. Contudo, uma determinada rotina interna não possui acesso às entidades de outra rotina interna. Rotinas internas devem ser precedidas pela palavra-chave `CONTAINS` dentro do hospedeiro.

Um exemplo simples do uso de uma função interna é apresentado a seguir. Note que a variável a é declarada no programa principal, tornando-se assim uma variável declarada também dentro do âmbito da função `calc_a_raiz`:

Listagem 9.1: Programa com função interna. Os valores dos argumentos da função nas linhas 10 e 12 são transferidos para a *variável muda* y na linha 14.

```
1  program rot_int
2  implicit none
3  real :: x,a

5  print*, "Entre com o valor de x:"
6  read*, x
7  print*, "Entre com o valor de a:"
8  read*, a
9  print*, "O resultado de a + sqrt(x) é:"
10 print*, calc_a_raiz(x)
11 print*, "O resultado de a + sqrt(1+ x**2) é:"
12 print*, calc_a_raiz(1.0 + x**2)
13 CONTAINS
14    function calc_a_raiz(y)
15    real             :: calc_a_raiz
16    real, intent(in) :: y
17    calc_a_raiz= a + sqrt(y)
18    return
19    end function calc_a_raiz
20 end program rot_int
```

No restante deste capítulo, serão descritas várias propriedades que se aplicam a subprogramas internos, externos e a rotinas de módulos.

9.3 Argumentos de subprogramas

Argumentos de subprogramas fornecem um mecanismo para que duas unidades de programa compartilhem os mesmos dados. Estes consistem em uma lista de variáveis escalares ou matriciais, constantes, expressões escalares ou matriciais e até mesmo os nomes de outras rotinas. Os argumentos de um subprograma são denominados **argumentos mudos** *(dummy arguments)*. Estes devem coincidir com os argumentos transferidos a partir da unidade que chama a rotina, denominados **argumentos reais**, em tipo, espécie e forma. Contudo, os nomes não necessitam ser os mesmos, como se pode ver no programa-exemplo 9.1. O exemplo mostra como é possível realizar duas chamadas da função `calc_a_raiz` transferindo valores distintos no seu argumento, inclusive empregando expressões:

```
...
read*, a,x
print*, calc_a_raiz(x)
print*, calc_a_raiz(a + x**2)
...
```

Sempre que possível, é recomendado que o nome do argumento transferido ao subprograma seja igual ao nome da variável muda.

O ponto importante é que subprogramas podem ser escritos de forma independente uns dos outros e das outras unidades. Podem ser também ser compilados separadamente, de modo a

gerar os seus respectivos programas-objeto (seção 2.1). Desta forma, bibliotecas de subprogramas podem ser construídas, compostas pelos seus programas-objeto, possibilitando que estes sejam invocados por diferentes programas (ver seção 9.4). Programas principais e outras unidades podem acessar os objetos armazenados nessas bibliotecas durante o processo de linkagem, gerando assim o código executável.

No momento em que uma determinada unidade de programa invoca um subprograma, ocorre a associação entre os argumentos reais e os argumentos mudos do subprograma, termo a termo, em um processo denominado **associação de argumentos** (*argument association*). Existe um amplo grau de liberdade na associação dos argumentos e, por consequência, o padrão fornece diversos mecanismos que verificam, no momento da linkagem, que as associações irão ocorrer da maneira correta. Estes recursos são discutidos abaixo.

O programa-exemplo `bascara1` (programa 9.2), implementa, de forma ingênua, o cálculo das raízes de um polinômio de segundo grau através da Fórmula de Báscara usando uma sub-rotina interna. Exemplos semelhantes a este serão usados para ilustrar o uso de rotinas em outras unidades de programa.

9.3.1 INSTRUÇÃO RETURN

No padrão da linguagem, um subprograma pode ser encerrado pelo comando `END`, sendo o controle do fluxo retornado à unidade que chamou este subprograma. Contudo, o meio recomendado de se retornar controle a partir de um subprograma consiste na utilização da instrução `RETURN`, a qual pode surgir em mais de um ponto da rotina, ao contrário do `END`.

No programa 9.2, a sub-rotina `bascara` é chamada na linha 11 do programa principal. Após a associação de argumentos `a2 = a`, `a1 = b`, `a0 = c`, `controle = raizes_reais`, `x1 = r1` e `x2 = r2`, o processamento prossegue a partir da linha 30 da sub-rotina. Na mesma linha, a função `testa_disc` é chamada para testar o discriminante e determinar o valor (lógico) de `raizes_reais`. Na linha 31, se `raizes_reais = .false.`, a instrução `return` é executada e o processamento retorna à linha 11, com as atribuições dos valores aos argumentos reais; no caso, `controle = .false.` (as raízes são complexas). Se `raizes_reais = .true.`, significa que as raízes são reais e o processamento continua até a linha 35, na próxima instrução `return`. Então, o processamento retorna à linha 11 com os valores atribuídos aos argumentos reais.

Observa-se também que na linha 30, quando a função `testa_disc` é invocada, ocorre a associação `c2 = a2`, `c1 = a1` e `c0 = a0`, sendo que o processamento passa a ocorrer a partir da linha 42 até a instrução `return` na linha 47, quando então o processamento retorna à linha 30 para a atribuição do valor a `raizes_reais`.

Como no caso dos comandos `STOP` e `END`, o `RETURN` pode ser rotulado, pode fazer parte de um construto como o `IF` e é um comando executável. Um `RETURN` não pode aparecer entre as instruções de um programa principal.

9.3.2 CONTROLE DE ACESSO AOS ARGUMENTOS MUDOS

O padrão oferece diferentes tipos de acessos possíveis aos argumentos mudos. Esse acesso é determinado pelos atributos/declarações `INTENT(IN)`, `INTENT(OUT)`, `INTENT(INOUT)` ou pelo atributo `VALUE`.

No programa `bascara1` (programa 9.2), as variáveis mudas `a0`, `a1` e `a2` foram utilizadas para transferir à sub-rotina informações acerca dos coeficientes do polinômio de 2º grau. Por isto, estes nomes foram declarados ter a *intenção* `INTENT(IN)`. Por outro lado, a sub-rotina devolveu ao programa os valores das raízes reais (caso existam) através das variáveis mudas `r1` e `r2`, as quais foram declaradas ter a *intenção* `INTENT(OUT)`. Uma terceira possibilidade seria a existência de uma variável cujo valor é inicialmente transferido *para dentro* da sub-rotina, modificado por esta e, então, transferido *para fora* da sub-rotina. Neste caso, esta variável deveria ser declarada com a *intenção* `INTENT(INOUT)`.

Se uma variável muda é especificada com atributo `INTENT(IN)`, o seu valor (ou qualquer parte do valor) não pode ser alterado pela rotina, seja através de atribuição de valor a partir de uma expressão, ou através de sua transferência para uma outra rotina que, por sua vez, alteraria este valor. Este tipo de intenção é o recomendado para funções, pois proíbe a alteração dos valores de seus argumentos.

Listagem 9.2: Exemplo de emprego de rotinas internas.

```
1  program bascara1
2  implicit none
3  logical :: controle
4  real :: a, b, c
5  real :: x1, x2 ! Raízes reais.

7  do
8     print*, "Entre com os valores dos coeficientes (a,b,c),"
9     print*, "onde a*x**2 + b*x + c."
10    read*, a,b,c
11    call bascara(a,b,c,controle,x1,x2)
12    if(controle)then
13       print*, "As raízes (reais) são:"
14       print*, "x1=", x1
15       print*, "x2=", x2
16       exit
17    else
18       print*, "As raízes são complexas.  Tente novamente."
19    end if
20 end do
21 CONTAINS
22    subroutine bascara(a2,a1,a0,raizes_reais,r1,r2)
23 ! Variáveis mudas:
24    real, intent(in)     :: a0, a1, a2
25    logical, intent(out) :: raizes_reais
26    real, intent(out)    :: r1, r2
27 ! Variáveis locais:
28    real :: disc

30    raizes_reais= testa_disc(a2,a1,a0)
31    if(.not. raizes_reais)return
32    disc= a1*a1 - 4*a0*a2
33    r1= 0.5*(-a1 - sqrt(disc))/a2
34    r2= 0.5*(-a1 + sqrt(disc))/a2
35    return
36    end subroutine bascara

38    function testa_disc(c2,c1,c0)
39    logical :: testa_disc
40 ! Variáveis mudas:
41    real, intent(in) :: c0, c1, c2
42    if(c1*c1 - 4*c0*c2 >= 0.0)then
43       testa_disc= .true.
44    else
45       testa_disc= .false.
46    end if
47    return
48    end function testa_disc
49 end program bascara1
```

Se a variável é especificada com `INTENT(OUT)`, o seu valor é indefinido na chamada da rotina, sendo este então definido durante a execução da mesma e finalmente transferido à unidade que chamou a rotina. Por esta razão, uma variável muda declarada com este atributo deve necessariamente ter o seu valor atribuído no subprograma. O argumento mudo não pode ser declarado também com o atributo `VALUE`.

Se a variável é especificada com `INTENT(INOUT)`, esta tem o seu valor inicial transferido na chamada da rotina, o valor pode ser alterado e, então, o novo valor (ou o original) será devolvido à unidade que chamou a rotina. O argumento mudo não pode ser declarado também com o atributo `VALUE`.

Como alternativa ao uso do atributo `INTENT(INOUT)`, pode-se declarar a intenção de nomes de argumentos mudos de rotinas, que não sejam subprogramas mudos, através da declaração `INTENT(INOUT)`, a qual tem a forma geral:

```
INTENT(<inout>) [::] <lista nomes argumentos mudos>
```

Por exemplo,

```
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
...
```

Uma quarta forma de controle de acesso a argumentos mudos é fornecida pelo atributo `VALUE`. Quando um argumento mudo é declarado com atributo `VALUE`, no momento da chamada do subprograma é realizada uma cópia local do valor do argumento real associado. A variável muda pode então ser empregada tanto em expressões quanto em atribuições, mas o possível novo valor da variável muda não será transferido de volta à variável real quando o processamento retornar à unidade invocadora.

O argumento mudo com atributo `VALUE` pode ser uma matriz, mas não pode ser ponteiro ou objeto alocável (seções 7.2 e 7.3), ser declarado também com `INTENT(OUT)` ou `INTENT(INOUT)`, ser o nome de uma rotina muda ou ter o atributo `VOLATILE`.

O tipo de acesso determinado pelo atributo `VALUE` é o tipo de acesso padrão para argumentos mudos de funções nas linguagens C e C++. Este mecanismo foi adicionado ao padrão do Fortran para a implementação da interoperabilidade com a linguagem C.

A função `nth_word_position` (programa 9.3) possui dois argumentos mudos: uma `string` de comprimento assumido (seção 3.1.4) com `INTENT(IN)` e o inteiro `n`, o qual é passado por valor. A função supõe que `string` é composta por diversas palavras separadas por espaço(s) em branco e então localiza a posição inicial da `n`-ésima palavra em `string`. O valor que `n` possui na unidade invocadora será copiado para o valor local assumido pela variável muda `n` na área de stack da memória reservada à função. Durante a execução do código da função, o valor de `n` é alterado, mas o último valor atribuído a `n` não é copiado de volta à rotina invocadora quando da execução da instrução `RETURN`.

Se um argumento mudo não possui controle explícito de acesso (via `INTENT` ou `VALUE`), o argumento real da unidade invocadora ainda pode ser uma variável nomeada ou uma expressão. Contudo, o argumento real deve ser uma variável nomeada se o argumento mudo é redefinido (*i.e.*, tem valores atribuídos). Por outro lado, uma variável real nomeada, digamos `X`, é transformada em uma expressão se ela é envolvida por parênteses, isto é, `(X)` na invocação da rotina. Neste caso, o valor de `X` é passado como o resultado de uma expressão e o valor da correspondente variável muda não pode ser alterado.

Sugestões de uso & estilo para programação

Recomenda-se que o controle de acesso aos argumentos seja sempre explícito, com o uso de `INTENT([IN][OU][INOUT])` ou `VALUE`. Isto facilita na documentação da rotina e desta maneira o compilador pode gerar um código adequado para os argumentos, visando a otimização do programa executável.

Listagem 9.3: Encontra a posição inicial da n-ésima palavra em uma string

```
function nth_word_position(string, n) result(pos)
integer :: pos
character(len=*), intent(in) :: string
integer, value               :: n ! Argumento passado por valor
logical                      :: in_word
in_word = .false.
do pos = 1, len(string)
   if (string(pos:pos) == ' ')then
      in_word= .false.
   else if (.not. in_word) then
      in_word= .true.     ! No primeiro caractere de uma palavra
      n= n - 1
      if (n == 0) return ! n-ésima palavra encontrada, retorna posição
   end if
end do
pos= 0                    ! Não há n palavras, retorna zero
return
end function nth_word_position
```

9.3.3 Argumentos com palavras-chave

No Fortran, argumentos com *palavras-chave (keyword arguments)* podem ser utilizados na chamada de rotinas. Quando uma rotina tem diversos argumentos, palavras-chave são um excelente recurso para evitar confusão entre os mesmos. A grande vantagem no uso de palavras-chave está em que não é necessário lembrar a ordem dos argumentos. Contudo, é necessário conhecer os nomes dos argumentos mudos definidos no campo de declarações da rotina.

Por exemplo, dada a seguinte função `AREA`:

```
FUNCTION AREA(INICIO, FINAL, TOL)
REAL             :: AREA
REAL, INTENT(IN) :: INICIO, FINAL, TOL
...
END FUNCTION AREA
```

esta pode ser chamada por quaisquer uma das quatro seguintes invocações:

```
A= AREA(0.0, 100.0, 1.0E-5)
B= AREA(INICIO= 0.0, TOL= 1.0E-5, FINAL= 100.0)
C= AREA(0.0, TOL= 1.0E-5, FINAL= 100.0)
VALOR= 100.0
ERRO= 1.0E-5
D= AREA(0.0, TOL= ERRO, FINAL= VALOR)
```

onde `A`, `B`, `C` e `D` são variáveis declaradas previamente como reais.

Todos os argumentos antes da primeira palavra-chave devem estar ordenados na ordem definida pela função. Contudo, se uma palavra-chave é utilizada, esta não necessita estar na ordem da definição. Além disso, o valor atribuído ao argumento mudo pode ser uma constante, uma variável ou uma expressão, desde que a interface[2] seja obedecida.

Depois que uma palavra-chave é utilizada pela primeira vez, todos os argumentos restantes também devem ser transferidos através de palavras-chave. Consequentemente, a seguinte chamada não é válida:

```
C= AREA(0.0, TOL= 1.0E-5, 100.0) ! Não é válido.
```

O exemplo acima não é válido porque tentou-se transferir o valor `100.0` sem haver a informação da palavra-chave.

[2] Interfaces de rotinas são discutidas na seção 9.5.

9.3.4 ARGUMENTOS OPCIONAIS

Em algumas situações, nem todos os argumentos mudos de um subprograma necessitam ser empregados durante uma chamada do mesmo. Um argumento que não necessita ser transferido em todas as chamadas possíveis de um subprograma é denominado *opcional*. Estes argumentos opcionais podem ser declarados ou pelo atributo `OPTIONAL` na declaração do tipo de variáveis ou pela declaração `OPTIONAL`.

Mantendo o exemplo da função `AREA` acima, a seguinte definição pode ser feita com o atributo:

```
FUNCTION AREA(INICIO, FINAL, TOL)
REAL                          :: AREA
REAL, INTENT(IN), OPTIONAL :: INICIO, FINAL, TOL
...
END FUNCTION AREA
```

ou, de forma equivalente, empregando a declaração:

```
FUNCTION AREA(INICIO, FINAL, TOL)
REAL                          :: AREA
REAL, INTENT(IN) :: INICIO, FINAL, TOL
OPTIONAL :: INICIO, FINAL, TOL
...
END FUNCTION AREA
```

Agora, a função tem as seguintes chamadas válidas, entre outras:

```
A= AREA(0.0, 100.0, 1.0E-2)
B= AREA(INICIO= 0.0, FINAL= 100.0, TOL= 1.0E-2)
C= AREA(0.0)
D= AREA(0.0, TOL= 1.0E-2)
```

onde se fez uso tanto da associação posicional para os valores dos argumentos, quanto da associação via o uso de palavras-chave.

Um argumento obrigatório (que não é declarado opcional) **deve** aparecer exatamente uma vez na lista de argumentos da chamada de uma rotina, ou na ordem posicional ou na lista de palavras-chave. Já um argumento opcional **pode** aparecer, no máximo uma vez, ou na lista posicional de argumentos ou na lista de palavras-chave.

Da mesma forma como na seção 9.3.3, a lista de palavras-chave pode aparecer em qualquer ordem; porém, depois que o primeiro argumento é transferido via uma palavra-chave, todos os restantes, obrigatórios ou opcionais, devem ser transferidos da mesma forma.

A rotina necessita de algum mecanismo para detectar se um argumento opcional foi transferido na sua chamada, para que esta possa tomar a medida adequada no caso de presença ou de ausência. Este mecanismo é fornecido pela função intrínseca `PRESENT` (seção 8.2), a qual retorna um valor lógico. Por exemplo, na função `AREA` acima, pode ser necessário verificar se a variável `TOL` foi transferida na chamada da função, para definir a tolerância do cálculo ou usar um valor padrão caso ela não tenha sido transferida:

```
FUNCTION AREA(INICIO, FINAL, TOL)
...
REAL :: TTOL
...
IF(PRESENT(TOL))THEN
   TTOL= TOL
ELSE
   TTOL= 1.0E-3
END IF
...
```

A variável `TTOL` é utilizada aqui porque ela pode ser redefinida, ao passo que a variável `TOL` não, uma vez que ela foi declarada com `INTENT(IN)`.

Uma pequena complicação pode surgir quando um argumento mudo opcional é empregado na rotina como um argumento real na invocação de outro subprograma. Por exemplo, suponha que no campo das instruções executáveis da função `AREA` acima ocorre a chamada

```
...
CALL INTEG(FX, INICIO, FINAL)
...
```

sendo `FX` o nome da rotina que implementa a função cuja área entre `INICIO` e `FINAL` é desejada.[3] Se a unidade que invoca `AREA` não fornecer o(s) valor(es) de `INICIO` e/ou `FINAL` como argumento(s) real(is), então na chamada da sub-rotina `INTEG` em `AREA` este(s) argumento(s) será(ão) considerado(s) ausente(s) também. Portanto, na definição de `INTEG` os mesmos argumentos também devem ser declarados como opcionais.

9.3.5 Tipos derivados como argumentos de rotinas

Argumentos de rotinas podem ser do tipo derivado se este está definido em somente uma única unidade de programa. Isto pode ser obtido de duas maneiras:

1. A rotina é interna à unidade de programa na qual o tipo derivado é definido.
2. O tipo derivado é definido em um módulo o qual é acessado pela rotina.

9.3.6 Matrizes como argumentos de rotinas

Um outro recurso importante em Fortran é a capacidade de usar matrizes como argumentos mudos de subprogramas. Nesta seção, será feito uso abundante dos parâmetros geométricos de matrizes, definidos na seção 6.1. Se um argumento mudo de uma rotina é uma matriz, então o argumento real pode ser:

- o nome da matriz (sem subscritos);
- uma seção de matriz (seção 6.3).

A primeira forma transfere a matriz inteira, enquanto que a segunda forma transfere somente uma seção da matriz.

Na definição de um argumento mudo matricial, os recursos suportados pelo padrão são descritos a seguir.

9.3.6.1 Matrizes de forma explícita ou ajustáveis

A maneira mais simples de declarar um argumento mudo matricial em um subprograma é como uma *matriz de forma explicita* (*explicit-shape array*). A matriz é declarada com posto e forma explícitos empregando literais inteiros ou constantes inteiras nomeadas. Por exemplo,

```
SUBROUTINE EXPLICIT1(A, B, C)
REAL, INTENT(IN),    DIMENSION(400,500) :: A
REAL, INTENT(OUT),   DIMENSION(500) :: B
REAL, INTENT(INOUT), DIMENSION(400) :: C
...
END SUBROUTINE EXPLICIT1
```

Neste exemplo, as matrizes `A`, `B` e `C` são matrizes de forma explícita declaradas como argumentos mudos da sub-rotina `EXPLICIT1`.

A vantagem deste tipo de declaração está no fato de que o compilador já sabe, no momento da compilação, o espaço de memória total que será ocupado por estas matrizes e assim pode sempre requisitar endereços de memória contíguos para armazenar os elementos das mesmas. Desta maneira o código será gerado da forma mais otimizada possível. Os endereços de memória ocupados por estas matrizes no stack de memória da rotina serão contíguos mesmo se os argumentos reais na unidade invocadora forem seções de matrizes (não necessariamente contíguas, portanto).

A evidente desvantagem deste recurso está em ser inflexível, porque não permite o uso simples de matrizes de qualquer outro tamanho. Se um argumento real for uma matriz de tamanho

[3] Como nomes de rotinas podem ser empregados como argumentos de subprogramas será discutido na seção 9.3.7.

maior que o tamanho reservado no respectivo argumento mudo, então somente uma parte do argumento real será passado para a rotina. A situação pode ser ainda pior porque a seção restante do argumento real pode ser automaticamente atribuído de forma errônea para outros argumentos mudos da rotina.

As evidentes desvantagens do uso de matrizes de forma explícita são contornadas com a declaração de argumentos mudos na forma de *matrizes ajustáveis* (*adjustable arrays*). Nesta forma, a lista de argumentos da rotina deve conter parâmetros inteiros em número suficiente para suprir todas as extensões das matrizes mudas. Um exemplo é apresentado abaixo:

```
SUBROUTINE ADJUSTABLE1(N, Y, Z)
INTEGER, INTENT(IN) :: N
REAL, INTENT(IN),  DIMENSION(N) :: Y
REAL, INTENT(OUT), DIMENSION(2*N) :: Z
...
END SUBROUTINE ADJUSTABLE1
```

Neste exemplo, o argumento mudo `N` fornece informações a respeito das extensões das matrizes ajustáveis `Y` e `Z`. O valor de `N` e, portanto, as extensões das matrizes, somente serão conhecidos durante a execução do programa. Observa-se que os argumentos reais na invocadora ainda podem ser seções de matrizes, desde que o valor do parâmetro `N` possa ser definido sem ambiguidades.

A sub-rotina do exemplo acima pode ser invocada pela linha

```
CALL ADJUSTABLE(100, A, B)
```

sendo `A` e `B` matrizes de forma e extensões compatíveis com a definição da sub-rotina.

O recurso das matrizes ajustáveis pode ser estendido de forma trivial para cobrir o caso de matrizes multidimensionais, com limites inferior e superior distintos. Por exemplo,

```
SUBROUTINE MULTI(MAP, K1, L1, K2, L2, TRACO)
INTEGER, INTENT(IN) :: K1, L1, K2, L2
REAL :: TRACO(L1-K1+1)
REAL :: MAP(K1:L1, K2:L2)
...
END SUBROUTINE MULTI
```

Como o exemplo mostra, os limites das dimensões das matrizes mudas pode ser expressões inteiras envolvendo não somente constantes mas também variáveis inteiras transferidas à rotina na lista de argumentos.

O mecanismo de matrizes ajustáveis pode ser usado para matrizes de qualquer tipo. Uma matriz ajustável também pode ser transferida a uma outra rotina como um argumento real com, se necessário, os limites das dimensões sendo passados de forma concomitante.

9.3.6.2 MATRIZES DE FORMA ASSUMIDA

No Fortran, como já foi visto na seção 7.1, a alocação do tamanho das matrizes pode ser realizada de maneira inteiramente dinâmica, sendo o espaço na memória da CPU alocado em tempo real de processamento. Para fazer uso deste recurso, existem dois mecanismos para a definição de matrizes mudas variáveis como argumentos de subprogramas: as *matrizes de forma assumida* e os *objetos automáticos*, os quais não são argumentos da rotina, mas tem suas extensões definidas pelas matrizes mudas desta.

Novamente como no caso de matrizes ajustáveis, os argumentos reais e mudos devem concordar em tipo e espécie. Contudo, agora uma matriz muda pode assumir a forma da matriz real usada no argumento da rotina. Tal matriz muda é denominada *matriz de forma assumida*. Quando a forma é declarada com a especificação `DIMENSION`, cada dimensão tem a sintaxe:

```
[<lim inferior>]:
```

isto é, pode-se definir o limite inferior da dimensão onde `<lim inferior>`, no caso mais geral, é uma expressão inteira que pode depender dos outros argumentos da rotina ou de variáveis globais definidas em módulos. Se `<lim inferior>` é omitido, o valor padrão é 1. Deve-se notar que é a forma da matriz que é transferida, não os seus limites. Por exemplo:

```
PROGRAM TESTA_MATRIZ
IMPLICIT NONE
REAL, DIMENSION(0:10, 0:20) :: A
...
CALL SUB_MATRIZ(A)
...
CONTAINS
   SUBROUTINE SUB_MATRIZ(DA)
   REAL, DIMENSION(:,:), INTENT(INOUT) :: DA
   ...
   END SUBROUTINE SUB_MATRIZ
END PROGRAM TESTA_MATRIZ
```

A sub-rotina `SUB_MATRIZ` declara a matriz de forma assumida `DA`. A correspondência entre os elementos da matriz real `A` e a matriz muda `DA` é: `A(0,0)`↔`DA(1,1)` ... `A(I,J)`↔`DA(I+1,J+1)` ... `A(10,20)`↔`DA(11,21)`. Para que houvesse a mesma correspondência entre os limites de `A` e `DA`, seria necessário declarar esta última como:

```
REAL, DIMENSION(0:,0:), INTENT(INOUT) :: DA
```

Neste caso, a correspondência seria a desejada: `A(0,0)`↔`DA(0,0)` ... `A(I,J)`↔`DA(I,J)` ... `A(10,20)`↔`DA(10,20)`.

Para se definir matrizes de forma assumida, é necessário que a interface seja explícita na unidade que chama a rotina. A matriz real pode ter sido declarada, na unidade que chama a rotina ou em outra unidade anterior, como uma matriz alocável. Como exemplo, considere a seguinte sub-rotina externa, seguida do programa que a chama, programa 9.4. Blocos de interface são discutidos em detalhes na seção 9.5.

```
subroutine sub(ra, rb, rc, max_a)
implicit none
real, dimension(:,:), intent(in)    :: ra, rb
real, dimension(:,:), intent(out)   :: rc
real, intent(out)                   :: max_a

max_a= maxval(ra)
rc= 0.5*rb
return
end subroutine sub
```

Listagem 9.4: *Exemplifica o uso de matrizes de forma assumida. Usa um bloco de interface para invocar a sub-rotina externa sub.*

```
program prog_sub
implicit none
real, dimension(0:9,10) :: a
real, dimension(5,5)    :: c
real     :: valor_max
integer  :: i
INTERFACE
   subroutine sub(ra, rb, rc, max_a)
   real, dimension(:,:), intent(in)    :: ra, rb
   real, dimension(:,:), intent(out)   :: rc
   real, intent(out)                   :: max_a
   end subroutine sub
END INTERFACE

a= reshape(source= [(cos(real(i)), i= 1,100)], shape= [10,10])
call sub(a, a(0:4,:5), c, valor_max)
```

```
print*, "A matriz a:"
do i= 0, 9
   print*, a(i,:)
end do
print*, ""
print*, "O maior valor em a:", valor_max
print*, ""
print*, "A matriz c:"
do i= 1, 5
   print*, c(i,:)
end do
end program prog_sub
```

9.3.6.3 OBJETOS AUTOMÁTICOS

Uma rotina com argumentos mudos que são matrizes cujas formas variam de uma chamada a outra pode também precisar de matrizes locais (não mudas) cujos tamanhos variem ao sabor dos tamanhos das matrizes mudas. Um exemplo simples é a matriz TEMP na sub-rotina abaixo, destinada a trocar os elementos entre duas matrizes:

```
SUBROUTINE TROCA(A,B)
IMPLICIT NONE
REAL, DIMENSION(:), INTENT(INOUT) :: A, B
REAL, DIMENSION(SIZE(A)) :: TEMP   !A função SIZE fornece o tamanho de TEMP.
TEMP= A
A= B
B= TEMP
RETURN
END SUBROUTINE TROCA
```

Matrizes como TEMP, cujas extensões variam da maneira apresentada, são chamadas *matrizes automáticas* (*automatic arrays*) e são exemplos de *objetos automáticos de dados* (*automatic data object*). Estes são objetos de dados cujas declarações dependem do valor de expressões não constantes e que não são argumentos mudos de rotinas.

Um outro tipo de objeto automático de dados, relacionado com variáveis de caracteres, surge através do comprimento variável de uma string muda, como no exemplo:

```
SUBROUTINE EXEMPLO(PALAVRA1)
IMPLICIT NONE
CHARACTER(LEN= *), INTENT(IN) :: PALAVRA1 ! String muda
CHARACTER(LEN= LEN(PALAVRA1)) :: PALAVRA2 ! String local automática
...
```

onde foi feito uso da função intrínseca LEN (seção 8.7.1) para determinar o comprimento da string PALAVRA2 igual ao comprimento da string muda PALAVRA1.

Neste exemplo, diz-se que PALAVRA1, a qual é argumento mudo da sub-rotina EXEMPLO, possui *comprimento de caractere assumido* (*assumed character length*), porque o seu comprimento será determinado em tempo de execução do programa como sendo igual ao comprimento da string passada pela unidade invocadora. Já PALAVRA2, cujo comprimento será automaticamente igual ao comprimento de PALAVRA1, é um outro exemplo de um objeto automático de dados.

Se este objeto consiste em uma função de caractere de tamanho variável, a interface deve ser explícita, como no programa-exemplo loren abaixo.

A definição dos limites das dimensões de um objeto automático pode ser realizada também por meio de argumentos mudos ou de variáveis definidas por *associação por uso* ou por *associação ao hospedeiro*. *Associação por uso* ocorre quando variáveis globais públicas declaradas no corpo de um módulo são disponibilizadas à rotina através da instrução USE; enquanto que *associação ao hospedeiro* ocorre quando variáveis declaradas em uma unidade de programa são disponibilizadas às suas rotinas internas. Uma discussão mais detalhada destas formas de associação é realizada na seção 9.14.2.

```
program loren
implicit none
character(len= *), parameter :: a= "Só um pequeno exemplo."
print*, dobro(a)
CONTAINS
   function dobro(a)
   character(len= *), intent(in) :: a
   character(len= 2*len(a)+2)       :: dobro
   dobro= a // "  " // a
   return
   end function dobro
end program loren
```

Matrizes automáticas são automaticamente criadas (alocadas) na entrada da rotina e automaticamente dealocadas na saída. Assim, o tamanho das matrizes automáticas pode variar em diferentes chamadas da rotina. Note que não há mecanismo para verificar a disponibilidade de memória para a criação de matrizes automáticas. Caso não haja memória suficiente, o programa é interrompido. Além disso, uma matriz automática não pode aparecer em uma declaração `SAVE` (seção 9.9) ou `NAMELIST` (seção 10.2), ou possuir o atributo `SAVE` (seção 9.9) na sua declaração. Além disso, a matriz não pode ser inicializada na sua declaração.

O seguinte programa-exemplo usa matrizes alocáveis, de forma assumida e automáticas:

```
subroutine sub_mat(a,res)
implicit none
real, dimension(:,:), intent(in) :: a          !Matriz de forma assumida
real, intent(out) :: res
real, dimension(size(a,1),size(a,2)) :: temp !Matriz automática
temp= sin(a)
res= minval(a+temp)
return
end subroutine sub_mat
```

```
program matriz_aut
implicit none
real, dimension(:,:), allocatable :: a
real    :: res
integer :: n, m, i
INTERFACE
   subroutine sub_mat(a,res)
   real, dimension(:,:), intent(in) :: a
   real, intent(out) :: res
   real, dimension(size(a,1),size(a,2)) :: temp
   end subroutine sub_mat
END INTERFACE
print*, "Entre com dimensões da matriz:"
read*, n,m
allocate(a(n,m))
a= reshape(source= [(tan(real(i)), i= 1,n*m)], shape= [n,m])
print*, "Matriz a:"
do i= 1, n
   print*, a(i,:)
end do
call sub_mat(a,res)
print*, ""
print*, "O menor valor de a + sin(a) é:",res
end program matriz_aut
```

9.3.7 SUBPROGRAMAS COMO ARGUMENTOS DE ROTINAS

Até este ponto, os argumentos reais passados a uma rotina foram supostos ser variáveis ou expressões. Contudo, uma outra possibilidade é um ou mais argumentos sendo nomes de outras rotinas que serão invocadas. Um exemplo de situação onde é necessário mencionar o nome de uma rotina como argumento de outra é quando a segunda executa a integração numérica da primeira. Desta forma, é possível escrever-se um integrador numérico genérico, que integra qualquer função que satisfaça a interface imposta a ela.

Um outro exemplo de uso do nome de uma rotina como argumento ocorre abaixo. A função `MINIMO` calcula o mínimo (menor valor) da função `FUNC` em um intervalo:

```
FUNCTION MINIMO(A, B, FUNC)
! Calcula o mínimo de FUNC no intervalo [A,B].
REAL :: MINIMO
REAL, INTENT(IN) :: A, B
INTERFACE
   FUNCTION FUNC(X)
   REAL :: FUNC
   REAL, INTENT(IN) :: X
   END FUNCTION FUNC
END INTERFACE
REAL :: F,X
...
F= FUNC(X) !Invocação da função.
...
END FUNCTION MINIMO
```

Note o uso de um bloco de interface (seção 9.5) para indicar ao compilador que o nome `FUNC` corresponde a uma função definida pelo usuário.

Uma alternativa ao uso de um bloco de interface consiste em empregar o atributo ou declaração `EXTERNAL`, como em

```
FUNCTION MINIMO(A, B, FUNC)
...
REAL, EXTERNAL :: FUNC
...
END FUNCTION MININO
```

para o primeiro caso ou

```
FUNCTION MINIMO(A, B, FUNC)
...
REAL :: FUNC
EXTERNAL :: FUNC
...
END FUNCTION MININO
```

para o segundo. Contudo, como será mencionado na seção 9.5, o simples uso da declaração ou atributo `EXTERNAL` não possibilita o controle nos argumentos desta função, ao contrário do que ocorre quando a interface é explícita.

Um exemplo de programa que chama a função `MINIMO` seria o seguinte:

```
PROGRAM MIN
IMPLICIT NONE
REAL :: MENOR
INTERFACE
   FUNCTION FUN(X)
   REAL :: FUN
   REAL, INTENT(IN) :: X
   END FUNCTION FUN
END INTERFACE
```

```
...
MENOR= MINIMO(1.0, 2.0, FUN)
...
END PROGRAM MIN
```

O uso de um bloco de interface não é necessário se a interface de FUN for explícita, como no caso de uma rotina de módulo. Rotinas de módulo serão discutidas em maiores detalhes na seção 9.13.2.

O programa-exemplo tes_plota abaixo chama a sub-rotina plota, a qual gera uma matriz de pontos a partir de uma função fornecida pelo programador. Desta forma, essa matriz de pontos pode ser gravada em um arquivo externo e ser então empregada para gerar o gráfico da função. Como exemplo, o programa principal passa à plota o nome da função Meu_F. As interfaces de plota e Meu_F são escritas explicitamente em blocos no programa principal e na sub-rotina porque ambas foram supostas ser externas. Contudo, elas poderiam ser internas a tes_plota ou contidas em módulos usados pelo programa e por plota, em cujas situações os blocos de interface são desnecessários, conforme será discutido na seção 9.5.

```
! Usa subrotina plota e chama a função externa Meu_F.
program tes_plota
implicit none
integer :: pts, j
real    :: yi, yf
real, dimension(:,:), allocatable :: xy
INTERFACE
   subroutine plota(f, xi, xf, npt, plot)
   integer, intent(in) :: npt
   real, intent(in)    :: xi, xf
   real, dimension(2,npt), intent(out) :: plot
   INTERFACE
      function f(x)
      real :: f
      real, intent(in) :: x
      end function f
   END INTERFACE
   end subroutine plota
!***
   function Meu_F(y)
   real :: Meu_F
   real, intent(in) :: y
   end function Meu_F
END INTERFACE

write(*, '(a)', advance= 'no') "Entre com o número de pontos: " ; read*, pts
print*, "Entre com os limites inferior e superior:"
read*, yi, yf
allocate(xy(2,pts))

call plota(Meu_F, yi, yf, pts, xy) ! Gera a matriz de pontos de Meu_F

do j= 1, pts
   print*, xy(:,j)
end do
end program tes_plota
```

```
! Gera uma matriz de pontos de uma função qualquer destinada à plotagem
!  do gráfico desta função.
! Parâmetros:
!  f: Função externa a ser plotada.
!  xi: Ponto inicial                    (entrada)
!  xf: Ponto final                      (entrada)
```

```
!  npt: Número de pontos a ser gerados (entrada)
!  plot: matriz de forma [ npt, 2 ] contendo as abcissas e ordenadas dos
!        pontos                              (saída)
subroutine plota(f, xi, xf, npt, plot)
implicit none
integer, intent(in) :: npt
real, intent(in)    :: xi, xf
real, dimension(2,npt), intent(out) :: plot
INTERFACE
   function f(x)
   real :: f
   real, intent(in) :: x
   end function f
END INTERFACE
! Variáveis locais:
integer :: i
real    :: passo, x
passo= (xf - xi)/real(npt - 1)
x= xi
do i= 1, npt
   plot(:,i)= [ x, f(x) ]
   x= x + passo
end do
return
end subroutine plota
```

```
! Função sen(exp(x)).
function Meu_F(x)
real :: Meu_F
real, intent(in) :: x
Meu_F= sin(exp(x))
return
end function Meu_F
```

9.3.8 PONTEIROS COMO ARGUMENTOS DE ROTINAS

Um argumento mudo de um sobprograma pode ser um ponteiro (seção 7.2), isto é, pode ser declarado com o atributo ou declaração `POINTER`. Neste caso, o argumento real na invocadora também deve possuir atributo de ponteiro. Quando o subprograma é invocado, o posto do argumento real deve coincidir com o posto do argumento mudo e o seu status de associação (seção 7.2.2.1) é automaticamente transferido ao argumento mudo.

Quando o processamento é retornado à invocadora, o status de associação do argumento real é normalmente tomado do status final do argumento mudo, mas pode se tornar indefinido se o argumento mudo estava associado na rotina com um alvo que se torna indefinido quando a instrução `RETURN` é executada. Um exemplo de situação onde isso ocorre é se o alvo for um objeto de dados local que não possui o atributo `SAVE`, conforme será discutido na seção 9.9.

Durante o processo de linkagem, o compilador saberá quando o argumento mudo é um ponteiro se a rotina for interna ou de módulo. Por outro lado, se a rotina for externa, o compilador irá assumir por padrão que aquele argumento real está sendo passado a um argumento mudo que não é um ponteiro. Para corrigir esta situação, é necessário informar o compilador que o argumento é, na realidade, um ponteiro, por meio de um bloco de interface.

A situação descrita acima pode ocorrer porque um ponteiro que corresponde a um argumento real pode ser passado para uma rotina cujo argumento mudo correspondente não é ponteiro. Por exemplo:

```
REAL, POINTER :: A(:,:)
...
ALLOCATE (A(80,80))
```

```
CALL FIND(A)
...
   SUBROUTINE FIND(C)
   REAL :: C(:,:) ! Matriz de forma assumida
```

Um dos principais usos de ponteiros como argumentos mudos corresponde ao processo inverso à passagem de matrizes de forma assumida para a rotina. Se a forma da matriz não era conhecida previamente pela unidade invocadora, mas é, outrossim, definida pela rotina, então declarar essa matriz como um ponteiro permite executar essa tarefa.

Declarar argumentos mudos como ponteiros aumenta consideravelmente a flexibilidade do processo de passagem de argumentos entre a unidade invocadora e o subprograma. Contudo, cuidado adicional é recomendado se o subprograma fizer parte de um conjunto grande de unidades que estão envolvidas na criação do código executável. Facilmente ocorre situações onde ponteiros são alocados em um subprograma, usados por outros, e finalmente dealocados ou anulados em outras rotinas. Devido a tal complexidade, facilmente pode acontecer de uma unidade tentar acessar um ponteiro que foi previamente desassociado, ou de tentar alocar novo espaço de memória para ponteiros que já estão alocados.

Sugestões de uso & estilo para programação

Em projetos de grande complexidade envolvendo ponteiros como argumentos mudos, sempre verifique os status de associação e alocação dos mesmos.

A sub-rotina `get_diagonal` abaixo mostra um exemplo onde o uso de ponteiros mudos é necessário. A sub-rotina deve receber um ponteiro para uma matriz quadrada `ptr_a` já definida e retornar um ponteiro para o vetor `ptr_b`, que irá corresponder aos elementos da diagonal da matriz quadrada. Antes de apontar para a diagonal da matriz, a sub-rotina realiza as seguintes verificações: *(i)* o ponteiro `ptr_a` deve estar associado na entrada; *(ii)* o ponteiro `ptr_b` não deve estar associado na entrada; *(iii)* a matriz em `ptr_a` deve ser quadrada e *(iv)* a alocação de espaço para `ptr_b` foi bem sucedida. O programa `test_diagonal` a seguir invoca `get_diagonal` como uma sub-rotina externa. O programa realiza diversos testes abrangendo todas as possíveis causas de erro no uso da sub-rotina.

```
! SUB-ROTINA get_diagonal:
!   Extrai os elementos da diagonal de uma matriz quadrada.
!   Inclui controles de erros
! Argumentos:
!   ptr_a: ponteiro para uma matriz quadrada          (Entrada)
!   ptr_b: ponteiro para o vetor contendo a diagonal  (Saída)
!   erro: controle de erros                           (Saída)
!      Valores de erro: 0 -- Sem erros
!                       1 -- ptr_a não estava associada na entrada
!                       2 -- ptr_b já estava associado na entrada
!                       3 -- matriz em ptr_a não é quadrada
!                       4 -- não foi possível alocar espaço para ptr_b
subroutine get_diagonal(ptr_a, ptr_b, erro)
implicit none
real, dimension(:,:), pointer :: ptr_a ! Ponteiro para matriz quadrada
real, dimension(:),   pointer :: ptr_b ! Ponteiro para vetor com diagonal
integer, intent(out) :: erro           ! Valor do erro
! Variáveis locais
integer :: i, istat                  ! Status de alocação
integer, dimension(2) :: linf, lsup  ! Limites inferior e superior em ptr_a
integer, dimension(2) :: exten       ! Extensões da matriz em ptr_a

! Verifica condições de erro
erro1: if(.not. associated(ptr_a))then
   erro= 1
else erro1
   erro2: if (associated(ptr_b)) then
      erro= 2
```

```
   else erro2
      linf= lbound(ptr_a) ; lsup= ubound(ptr_a)
      exten= lsup - linf + 1
      erro3: if(exten(1) /= exten(2))then
         erro= 3
      else erro3
         allocate(ptr_b(exten(1)), stat= istat)
         erro4: if(istat /= 0)then
            erro= 4
         else erro4 ! Sem erros.  Prossiga
            erro= 0
            do i= 1, exten(1)
               ptr_b(i)= ptr_a(linf(1)+i-1, linf(2)+i-1)
            end do
         end if erro4
      end if erro3
   end if erro2
end if erro1
return
end subroutine get_diagonal
```

```
program test_diagonal
implicit none
integer :: i, j, k
real, dimension(:,:), pointer :: pa
real, dimension(:),   pointer :: pb
integer :: erro
INTERFACE ! Interface de get_diagonal
   subroutine get_diagonal(ptr_a, ptr_b, erro)
   real, dimension(:,:), pointer :: ptr_a
   real, dimension(:),   pointer :: ptr_b
   integer, intent(out) :: erro
   end subroutine get_diagonal
END INTERFACE

! Primeiro teste: chama get_diagonal sem definir pa
call get_diagonal(pa, pb, erro)
print'(a,g0)', 'Teste 1: Ponteiro pa não alocado.   erro= ', erro

! Segundo teste: aloca ambos os ponteiros e chama get_diagonal
allocate(pa(10,10), pb(10))
call get_diagonal(pa, pb, erro)
print'(a,g0)', 'Teste 2: Ponteiro pb alocado.       erro= ', erro
deallocate(pa, pb)

! Terceiro teste: aloca pa não quadrada
allocate(pa(-5:5, 10))
call get_diagonal(pa, pb, erro)
print'(a,g0)', 'Teste 3: Matriz em pa não quadrada. erro= ', erro
deallocate(pa)

! Teste bem sucedido
allocate(pa(-4:4, 0:8))
call random_number(pa)
print'(/,a)', 'Matriz em pa:'
do i= -4, 4
   print'(11(g0, x))', pa(i,:)
end do
call get_diagonal(pa, pb, erro)
```

```
print'(/,a,g0)', 'Teste 4: Teste bem sucedido.      erro= ', erro
print'(a)', 'Diagonal:'
print'(9(g0, x))', pb
end program test_diagonal
```

USO DO ATRIBUTO INTENT COM PONTEIROS

Um argumento mudo com atributo de ponteiro também pode ser declarado com o atributo `INTENT`. Contudo, neste caso é importante enfatizar que o `INTENT` se refere ao *valor do ponteiro*, isto é, ao seu *descritor*, e não ao(s) valor(es) do seu alvo.

Assim, se o ponteiro mudo for declarado com:

`INTENT(OUT)`: o ponteiro terá status de associação indefinido na entrada do subprograma.

`INTENT(IN)`: o ponteiro não pode ser desassociado ou associado a um outro alvo durante a execução da rotina.

`INTENT(INOUT)`: o argumento real deve ser uma variável de ponteiro. Não pode ser uma referência a uma função com valor de ponteiro (seção 9.7).

Embora o descritor de um ponteiro mudo declarado com `INTENT(IN)` não possa ser alterado na rotina, se o mesmo estiver associado a um alvo, o valor do seu alvo pode sim ser alterado. Por exemplo:

```
SUBROUTINE PONT_FIXO(P)
REAL, POINTER, INTENT(IN) :: P(:)
IF(ASSOCIATED(P))P = 0.0
RETURN
END SUBROUTINE PONT_FIXO
```

9.3.9 OBJETOS ALOCÁVEIS COMO ARGUMENTOS DE ROTINAS

Uma alternativa ao uso de ponteiros como argumentos mudos (seção 9.3.8) que pode oferecer a mesma flexibilidade em muitas situações consiste no emprego de objetos alocáveis (seção 7.3) como argumentos mudos. Neste caso, o argumento real correspondente deve ser alocável dos mesmos posto, tipo e espécie do argumento mudo e a interface deve ser explícita.

O argumento mudo recebe o status de alocação do argumento real na entrada (na forma de descritores) e o argumento real recebe o status de alocação do argumento mudo na saída da rotina. Em ambos os casos o status pode ser dealocado mas, se alocado, os limites são também passados.

Um argumento mudo alocável pode ser declarado também com `INTENT`, mas, diferente do que ocorre com ponteiros mudos, o `INTENT` se aplica tanto ao status de alocação (o descritor) quanto ao valor do objeto de dados. Se `INTENT(IN)`, o objeto não pode ser alocado ou dealocado e o valor do objeto não pode ser alterado durante a execução da rotina. Se `INTENT(OUT)`, e o objeto está alocado na entrada a rotina, este se torna automaticamente dealocado.

O programa `allocatable_argument` abaixo recebe um vetor alocável da sub-rotina interna `svec`. A sub-rotina tem o vetor alocável `vec` como argumento mudo. A extensão e os valores do vetor são determinados durante a execução da sub-rotina. Na saída, o vetor alocável `mvec` no programa principal automaticamente recebe tanto o status de alocação de `vec` quanto os valores dos dados.

```
program allocatable_argument
implicit none
integer :: i
real, dimension(:), allocatable :: mvec
call svec(mvec)
print'(a, *(g0, x))', 'main: mvec: ', (mvec(i), i= 1, size(mvec))
CONTAINS
   subroutine svec(vec)
   real, dimension(:), allocatable, intent(out) :: vec
```

```
    integer :: num
    write(*, '(a)', advance= 'no') 'svec: num= ' ; read(*,*)num
    allocate(vec(num))
    call random_number(vec)
    print '(a, *(g0, x))', 'svec: vec:  ', (vec(i), i= 1, size(vec))
    return
    end subroutine svec
end program allocatable_argument
```

9.4 ROTINAS EXTERNAS E BIBLIOTECAS

Em muitas situações, uma determinada rotina é escrita com o intuito de ser utilizada em diversas aplicações distintas. Um exemplo seria uma rotina genérica que calcula a integral de uma função qualquer entre dois pontos reais. Nesta situação, é mais interessante manter a rotina como uma entidade distinta do programa principal, para que esta possa ser utilizada por outras unidades de programas.

Por isto, uma ou mais rotinas podem ser escritas em um arquivo em separado, constituindo uma unidade de programa própria, uma rotina externa, já mencionada na seção 9.1.2. Esta unidade pode então ser compilada em separado, quando será então gerado o programa-objeto correspondente a esta parte do código. Este programa-objeto é então acessado pelo programa linkador para gerar o programa executável, ligando a(s) rotina(s) contida(s) nesta unidade com as outras unidades que fazem referência à(s) mesma(s). O procedimento para realizar esta linkagem é realizado, usualmente, de duas maneiras distintas:

1. O(s) programa-objeto(s) é(são) linkado(s) diretamente com as demais unidades de programa. Neste caso, o compilador e o linkador identificam a existência do programa-objeto pela sua extensão, fornecida na linha de comando que demanda a criação do código executável. A extensão de um programa-objeto é usualmente `*.o` em sistemas operacionais Linux/Unix ou `*.obj` em sistemas DOS/Windows.

2. O programa-objeto recém criado é inicialmente armazenado em uma biblioteca de programas-objeto criados anteriormente. Então, no momento de se gerar o código executável a partir de um programa principal, o linkador é instruído com o nome e o endereço da biblioteca que contém o(s) programa-objeto(s) da(s) rotina(s) chamadas pelo programa principal e/ou por outras unidades de programas. O linkador procura nesta biblioteca pelos nomes das rotinas invocadas e incorpora o código destas (e somente destas) ao programa executável.

Com base nesta estratégia, o programa `bascara1` (programa 9.2) pode ser agora desmembrado em três arquivos distintos, dois contendo a sub-rotina `bascara` e a função `testa_disc` e um contendo o programa principal `bascara2`, todos apresentados na página 166. Note o uso do atributo `EXTERNAL` para indicar que o nome `TESTA_DISC` consiste em uma função externa do tipo lógico. Neste caso, o compilador não pode verificar a concordância entre os argumentos reais e os mudos para a função.

9.5 INTERFACES IMPLÍCITAS E EXPLÍCITAS

No exemplo apresentado na página 166, o nome `testa_disc` é identificado como pertencente a uma função lógica pelo atributo `EXTERNAL`:

```
...
LOGICAL, EXTERNAL :: TESTA_DISC
...
```

Caso esta declaração não fosse realizada, o compilador iria gerar uma mensagem de erro indicando que este nome não foi declarado.

Uma outra maneira de declarar um nome como pertencente a uma função externa é usando a declaração `EXTERNAL`, em vez do atributo. Desta maneira, o mesmo resultado seria obtido através das seguintes linhas:

```
! Chamada pela subrotina bascara para testar se as raízes são reais.
function testa_disc(c2,c1,c0)
implicit none
logical :: testa_disc
real, intent(in) :: c0, c1, c2
if(c1*c1 - 4*c0*c2 >= 0.0)then
   testa_disc= .true.
else
   testa_disc= .false.
end if
return
end function testa_disc
```

```
! Calcula as raízes reais, caso existam, de um polinônio de grau 2.
! Usa função lógica externa testa_disc.
subroutine bascara(a2,a1,a0,raizes_reais,r1,r2)
implicit none
real, intent(in)     :: a0, a1, a2
logical, intent(out) :: raizes_reais
real, intent(out)    :: r1, r2
! Entidades locais:
logical, external :: testa_disc ! Função externa
real :: disc

raizes_reais= testa_disc(a2,a1,a0)
if(.not. raizes_reais)return
disc= a1*a1 - 4*a0*a2
r1= 0.5*(-a1 - sqrt(disc))/a2
r2= 0.5*(-a1 + sqrt(disc))/a2
return
end subroutine bascara
```

```
!Calcula as raízes reais de um polinômio de grau 2.
program bascara2
implicit none
logical :: controle
real :: a, b, c
real :: x1, x2 ! Raízes reais.

do
   print*, "Entre com os valores dos coeficientes (a,b,c),"
   print*, "onde a*x**2 + b*x + c."
   read*, a,b,c
   call bascara(a,b,c,controle,x1,x2)
   if(controle)then
      print*, "As raízes (reais)  são:"
      print*, "x1=", x1
      print*, "x2=", x2
      exit
   else
      print*, "As raízes são complexas.  Tente novamente."
   end if
end do
end program bascara2
```

```
...
LOGICAL :: TESTA_DISC
EXTERNAL :: TESTA_DISC
...
```

Em ambas as opções anteriores, somente é fornecido ao compilador o nome da função. Nenhum controle existe sobre os argumentos de `testa_disc`.

Note também que, como os subprogramas deste exemplo são externos, o programa chama a sub-rotina `bascara` sem ter conhecimento prévio da natureza dos seus argumentos (tal como o seu número, tipo e espécie). Por isto, seria possível o programa `bascara2` chamar a sub-rotina `bascara` tentando passar variáveis complexas, por exemplo, quando os argumentos mudos da sub-rotina esperam variáveis reais. Sem a existência de um mecanismo específico de verificação dos argumentos reais com os argumentos mudos, os programas-objeto podem ser linkados e o executável gerado, o que provavelmente levará a erros durante o processamento do código.

Para gerar chamadas a um subprograma de forma correta, o compilador necessita conhecer a **interface** do mesmo, isto é, se o subprograma é uma função ou subrotina e os nomes e o número de argumentos mudos e a natureza dos mesmos, tal como o tipo e espécie dos argumentos.

No caso de rotinas intrínsecas, subprogramas internos e rotinas de módulo (seção 9.13.2), esta informação é sempre conhecida pelo compilador; por isso, diz-se que as interfaces são **explícitas**.

Contudo, quando o compilador chama uma rotina externa, esta informação não é conhecida de antemão e daí a interface é dita **implícita**. Um **bloco de interface** (*interface block*), então, fornece a informação necessária ao compilador. A forma geral de um bloco de interface é:

```
INTERFACE
  <corpo interface>
END INTERFACE
```

Nesta forma, este bloco de interface não pode ser nomeado. Assim, a interface deixa de ser implícita e se torna explícita.

O `<corpo interface>` consiste das declarações `FUNCTION` ou `SUBROUTINE`, declarações dos tipos e espécies dos argumentos dos subprogramas e do comando `END FUNCTION` ou `END SUBROUTINE`. Em outras palavras, consiste, normalmente, em uma cópia exata do subprograma sem seus comandos executáveis, objetos locais ou rotinas internas, cujas interfaces sempre são explícitas. Por exemplo,

```
INTERFACE
   FUNCTION FUNC(X)
   REAL             :: FUNC
   REAL, INTENT(IN) :: X
   END FUNCTION FUNC
END INTERFACE
```

Neste exemplo, o bloco fornece ao compilador a interface da função `FUNC`, a qual é uma função de valor real com um argumento mudo também real.

Existe uma certa flexibilidade na definição de um bloco de interfaces. Os nomes usados no bloco para os argumentos da rotina (mas não os seus tipos e espécies) podem ser distintos dos nomes usados na definição da rotina ou usados na chamada da mesma. Outras especificações podem ser incluídas, como por exemplo para uma variável local à rotina. Contudo, rotinas internas e comandos `DATA` ou `FORMAT` não podem ser incluídos.

Um bloco de interfaces pode conter as interfaces de mais de um subprograma externo e este bloco pode ser colocado na unidade de programa que chama as rotinas externas ou através do uso de módulos, como será discutido na seção 9.13. Em qualquer unidade de programa, o bloco de interfaces é sempre colocado após as declarações de variáveis mudas ou de variáveis locais.

Em certas situações, é necessário fornecer a uma rotina externa o nome de outra rotina externa como um de seus argumentos. Isto pode ser realizado de duas maneiras:

1. Usando o atributo ou declaração `EXTERNAL`, como foi realizado na sub-rotina `bascara`, na página 166.

2. Com o uso de um bloco próprio de interface. Neste caso, a interface indicará ao compilador que um determinado nome consiste, na verdade, no nome de um outro subprograma.

As situações onde isto pode ocorrer são discutidas na seção 9.3.7

Quando a unidade invocadora emprega palavras-chave (seção 9.3.3) na chamada de um subprograma ou a rotina declara argumentos opcionais (seção 9.3.4), a interface do subprograma deve ser explícita, pois, em caso contrário, o compilador não será capaz de realizar as associações apropriadas entre os argumentos reais na invocadora e os argumentos mudos na definição da rotina. No caso de rotinas internas ou rotinas de módulo, a interface já é explícita. No caso de rotinas externas, faz-se necessário o uso de um bloco de interfaces.

Por exemplo, se a função AREA, empregada como exemplo na seção 9.3.4, for definida com argumentos opcionais, e o seu código fonte for criado como uma rotina externa, um bloco de interface deve ser fornecido em algum momento e tornado acessível para qualquer unidade invocadora. Neste caso, o bloco de interface pode ser escrito:

```
INTERFACE
   FUNCTION AREA(INICIO, FINAL, TOL)
   REAL                        :: AREA
   REAL, INTENT(IN), OPTIONAL :: INICIO, FINAL, TOL
   END FUNCTION AREA
END INTERFACE
```

Como exemplo do uso de interfaces explícitas será apresentado o programa bascara3, o qual chama novamente a função testa_disc (listada na página 166) e uma nova versão para a subrotina bascara. As unidades de programa estão listadas abaixo.

```
! Calcula as raízes reais, caso existam, de um polinônio de grau 2.
! Usa a função externa testa_disc.
subroutine bascara(a2,a1,a0,raizes_reais,r1,r2)
implicit none
real, intent(in)     :: a0, a1, a2
logical, intent(out) :: raizes_reais
real, intent(out)    :: r1, r2
! Entidades locais:
real :: disc
INTERFACE
   function testa_disc(c2,c1,c0)
   logical :: testa_disc
   real, intent(in) :: c0, c1, c2
   end function testa_disc
END INTERFACE

raizes_reais= testa_disc(a2,a1,a0)
if(.not. raizes_reais)return
disc= a1*a1 - 4*a0*a2
r1= 0.5*(-a1 - sqrt(disc))/a2
r2= 0.5*(-a1 + sqrt(disc))/a2
return
end subroutine bascara
```

```
!Calcula as raízes reais de um polinômio de grau 2.
program bascara3
implicit none
logical :: controle
real :: a, b, c
real :: x1, x2 ! Raízes reais.
INTERFACE
   subroutine bascara(a2,a1,a0,raizes_reais,r1,r2)
   real, intent(in)     :: a0, a1, a2
   logical, intent(out) :: raizes_reais
   real, intent(out)    :: r1, r2
   end subroutine bascara
END INTERFACE
```

```
do
   print*, "Entre com os valores dos coeficientes (a,b,c),"
   print*, "onde a*x**2 + b*x + c."
   read*, a,b,c
   call bascara(a,b,c,controle,x1,x2)
   if(controle)then
      print*, "As raízes (reais)  são:"
      print*, "x1=",x1
      print*, "x2=",x2
      exit
   else
      print*, "As raízes são complexas.  Tente novamente."
   end if
end do
end program bascara3
```

9.5.1 A INSTRUÇÃO IMPORT

Como foi visto nesta seção, um bloco de interface pode ser empregado em qualquer unidade de programa. Contudo, o bloco *per se* não tem acesso às entidades declaradas ou acessíveis nessa unidade por associação ao hospedeiro (seção 9.14.2). Isto significa que se no bloco for empregada alguma constante nomeada, parâmetro de espécie de tipo, tipo derivado ou rotina, que estejam acessíveis à unidade hospedeira, o bloco em princípio não terá acesso a essas entidades. A instrução IMPORT resolve esta situação ao estabelecer um mecanismo de associação ao hospedeiro dentro de um bloco de interfaces.

A instrução IMPORT possui a seguinte sintaxe, quando a mesma se encontra em um bloco de interfaces:

```
IMPORT [[::] <lista-nomes-import>]
```

onde cada nome na <lista-nomes-import> deve pertencer a uma entidade que está acessível na unidade que contém o bloco. Se a entidade importada foi definida na própria unidade, então esta deve ter sido declarada antes do bloco de interface.

Um IMPORT sem a <lista-nomes-import> irá importar todas as entidades acessíveis à unidade hospedeira do bloco e que não têm os mesmos nomes de entidades declaradas no bloco.

O seguinte exemplo mostra como a instrução torna acessíveis ao bloco de interfaces constantes e tipos derivados.

```
PROGRAM PROG
INTEGER, PARAMETER :: WP= KIND(0.0D0)
TYPE :: T
   ...
END TYPE T
CONTAINS
   SUBROUTINE APPLY(FUN, ...)
   INTERFACE
      IMPORT :: WP, T
      FUNCTION FUN(F)
      TYPE(T) :: FUN
      REAL(WP), INTENT(IN) :: F
      ...
      END FUNCTION FUN
   END INTERFACE
   ...
   END SUBROUTINE APPLY
END PROGRAM PROG
```

9.6 Funções de valor matricial e funções alocáveis

Funções, além de fornecer os resultados usuais na forma de valores dos tipos intrínsecos inteiro, real, complexo, lógico e de caractere, podem também fornecer como resultado valores de tipo derivado, matrizes e ponteiros. No caso de funções de valor matricial, o tamanho do resultado pode ser determinado de forma semelhante à maneira como matrizes automáticas são declaradas.

Considere o seguinte exemplo de uma função de valor matricial:

```
PROGRAM TES_VAL_MAT
IMPLICIT NONE
INTEGER, PARAMETER :: M= 6
INTEGER, DIMENSION(M,M) :: IM1, IM2
...
IM2= FUN_VAL_MAT(IM1,1) !Chama função matricial.
...
CONTAINS
   FUNCTION FUN_VAL_MAT(IMA, SCAL)
   INTEGER, DIMENSION(:,:), INTENT(IN) :: IMA
   INTEGER, INTENT(IN) :: SCAL
   INTEGER, DIMENSION(SIZE(IMA,1),SIZE(IMA,2)) :: FUN_VAL_MAT
   FUN_VAL_MAT= IMA*SCAL
   RETURN
   END FUNCTION FUN_VAL_MAT
END PROGRAM TES_VAL_MAT
```

Neste exemplo, o limites das dimensões de `FUN_VAL_MAT` são herdadas do argumento verdadeiro transferido à função e usadas para determinar o tamanho da matriz resultante.

Para que o compilador conheça a forma da matriz resultante, a interface de uma função de valor matricial deve ser explícita em todas as unidades onde esta função é invocada.

Uma função também pode ter seu resultado declarado com o atributo `ALLOCATABLE`. Este recurso é útil quando o tamanho do resultado somente se torna conhecido durante a execução da função. O status de alocação do resultado da função é não alocado no momento da entrada da mesma. Este status pode ser alocado e dealocado diversas vezes durante a execução da rotina, mas o resultado deve estar alocado e ter valores definidos no momento da saída da função.

Novamente, a interface da função deve ser sempre explícita. O resultado da função se torna automaticamente dealocado após a execução da(s) instrução(ões) onde a referência à função é realizada, mesmo que tenho o atributo `TARGET`.

A função de resultado alocável `compact`, definida abaixo, é um exemplo de aplicação deste recurso. Esta função pode ser empregada para eliminar valores duplicados em um vetor. O programa `remove_dups` chama `compact` como uma função interna.

```
program remove_dups
implicit none
integer :: num
integer, dimension(:), allocatable :: x, y
real, dimension(:), allocatable :: temp
write(*, '(a)', advance= 'no')'Num. de elementos iniciais: ' ; read*, num
allocate(x(num), temp(num))
call random_number(temp)
x= int(10*temp)
deallocate(temp)
print'(/,a)', 'Valores no vetor x:'
print'(*(g0,x))', x
! O vetor y conterá o vetor x compactado
y= compact(x)
print'(/,a)', 'Valores no vetor y:'
print'(*(g0,x))', y
```

```
CONTAINS
   function compact(x)
   integer, dimension(:), allocatable :: compact
   integer, dimension(:), intent(in)  :: x
   integer :: i
   allocate(compact(1))
   compact= x(1)
   do i= 2, size(x)
      if(any(compact == x(i)))cycle
      compact= [ compact, x(i) ]
   end do
   return
   end function compact
end program remove_dups
```

9.7 FUNÇÕES COM VALOR DE PONTEIRO

Uma função também pode retornar um ponteiro. O resultado da função é inicialmente indefinido e, durante a execução da mesma, o ponteiro deve se tornar associado a um alvo ou definido como desassociado.

Uma função com valor de ponteiro será usualmente empregada do lado direito de uma atribuição de ponteiro ou como um componente ponteiro de um construtor de estrutura (seção 4.8). A referência à função de ponteiro também pode ocorrer em uma expressão ou do lado direito de uma atribuição ordinária, em cuja situação o resultado deve se tornar associado com um alvo que está definido e o valor deste alvo deve ser usado.

O programa `tes_pointer_function` abaixo emprega a função (interna) de valor de ponteiro `cada_quinto` para retornar o primeiro e cada quinto elementos de um vetor que é passado como argumento da função. O programa emprega a função na atribuição de ponteiro

```
p2 => cada_quinto(p1)
```

A função é novamente empregada no lugar de uma expressão ordinária, no comando

```
print'(*(g0, x))', cada_quinto(p1)
```

```
program tes_pointer_function
implicit none
integer :: num
real, dimension(:), pointer :: p1, p2
write(*, '(a)', advance= 'no')'Num. de elementos: ' ; read*, num
allocate(p1(num))
call random_number(p1)
print'(a)', 'Vetor original:'
print'(*(g0, x))', p1
p2 => cada_quinto(p1)
print'(/,a)', 'Vetor com cada quinto elemento:'
print'(*(g0, x))', p2
print'(/,a)', 'Vetor com cada quinto elemento (novamente):'
print'(*(g0, x))', cada_quinto(p1)
CONTAINS
! Função CADA_QUINTO
! Retorna um ponteiro para cada quinto elemento de um vetor
   function cada_quinto(ptr)
   real, dimension(:), pointer :: cada_quinto
   real, dimension(:), pointer, intent(in) :: ptr
   integer :: low, high
   low= lbound(ptr,1) ; high= ubound(ptr,1)
   cada_quinto => ptr(low:high:5)
```

```
    return
    end function cada_quinto
end program tes_pointer_function
```

9.8 Recursividade e rotinas recursivas

Recursividade ocorre quando uma rotina invoca a si mesma, seja de forma direta ou indireta. Por exemplo, sejam as rotinas `A` e `B`. Recursividade pode ocorrer de duas formas:

Recursividade direta: `A` invoca `A` diretamente.

Recursividade indireta: `A` invoca `B`, a qual invoca `A`.

Qualquer cadeia de invocações de rotinas com um componente circular (isto é, invocação direta ou indireta) exibe recursividade. Embora recursividade seja uma técnica bonita e sucinta para implementar uma grande variedade de problemas, o uso incorreto da mesma pode provocar uma perda na eficiência da computação do código.

9.8.1 Recursividade direta

Para fins de eficiência na execução do código, rotinas recursivas devem ser explicitamente declaradas usando-se como prefixo a palavra-chave `RECURSIVE`:

```
RECURSIVE SUBROUTINE SUB(...)
```

ou

```
RECURSIVE FUNCTION FF(...) RESULT (VAR_RESULT)
```

A declaração de uma função recursiva é realizada com uma sintaxe um pouco distinta da até então abordada: uma função explicitamente declarada recursiva deve conter também a palavra-chave `RESULT`, a qual especifica o nome de uma variável à qual o valor do resultado do desenvolvimento da função deve ser atribuído, em lugar do nome da função propriamente dito. No exemplo acima, o nome da função recursiva é `FF`, mas a variável que terá o valor do resultado atribuído é `VAR_RESULT`. Na definição da função `FF`, portanto, será `VAR_RESULT` a variável que deve ser declarada com o tipo e espécie do resultado, ao invés de `FF`.

A palavra-chave `RESULT` é necessária, uma vez que não é possível usar-se o nome da função para retornar o resultado, pois isso implicaria em perda de eficiência no código. De fato, o nome da variável determinado pela palavra-chave deve ser usado tanto na declaração do tipo e espécie do resultado da função quanto para atribuição de resultados e em expressões escalares ou matriciais. A palavra-chave `RESULT` pode também ser usada para funções não recursivas.

Cada vez que uma rotina recursiva é invocada, um conjunto novo de objetos de dados locais é criado, o qual é eliminado na saída da rotina. Este conjunto consiste de todos os objetos definidos no campo de declarações da rotina ou declarados implicitamente, exceto aqueles com atributos `DATA` ou `SAVE` (ver seção 9.9). Funções de valor matricial recursivas são permitidas e, em algumas situações, a chamada de uma função recursiva deste tipo é indistinguível de uma referência a matrizes.

O exemplo tradicional do uso de rotinas recursivas consiste na implementação do cálculo do fatorial de um inteiro positivo. A implementação é realizada a partir das seguintes propriedades do fatorial:

$$0! = 1; \quad N! = N(N-1)!$$

A seguir, o cálculo do fatorial é implementado tanto na forma de uma função quanto na forma de uma sub-rotina recursivas:

```
RECURSIVE FUNCTION FAT(N) RESULT (N_FAT)
IMPLICIT NONE
INTEGER :: N_FAT !Define também o tipo de FAT.
```

```
INTEGER, INTENT(IN) :: N
IF(N == 0)THEN
   N_FAT= 1
ELSE
   N_FAT= N*FAT(N-1)
END IF
RETURN
END FUNCTION FAT
-------------------------------------------
RECURSIVE SUBROUTINE FAT(N, N_FAT)
IMPLICIT NONE
INTEGER, INTENT(IN)  :: N
INTEGER, INTENT(INOUT) :: N_FAT
IF(N == 0)THEN
   N_FAT= 1
ELSE
   CALL FAT(N - 1, N_FAT)
   N_FAT= N*N_FAT
END IF
RETURN
END SUBROUTINE FAT
```

9.8.2 RECURSIVIDADE INDIRETA

Uma rotina também pode ser invocada por recursividade indireta, isto é, uma rotina chama outra a qual, por sua vez, invoca a primeira. Para ilustrar a utilidade deste recurso, supõe-se que se queira realizar uma integração numérica bi-dimensional quando se dispõe somente de um código que executa integração unidimensional. Um exemplo de função que implementaria integração unidimensional é dado a seguir. O exemplo usa um módulo, o qual será discutido em maiores detalhes na seção 9.13.

```
! Função INTEGRA: Integra F(x) de LIMITES(1) a LIMITES(2).
FUNCTION INTEGRA(F, LIMITES)
IMPLICIT NONE
REAL, DIMENSION(2), INTENT(IN) :: LIMITES
INTERFACE
   FUNCTION F(X)
   REAL :: F
   REAL, INTENT(IN) :: X
   END FUNCTION F
END INTERFACE
...
INTEGRA= <implementação integração numérica>
RETURN
END FUNCTION INTEGRA
-------------------------------------------
MODULE FUNC
IMPLICIT NONE
REAL :: YVAL
REAL, DIMENSION(2) :: LIMX, LIMY
CONTAINS
   FUNCTION F(XVAL)
   REAL :: F
   REAL, INTENT(IN) :: XVAL
   F= <expressão envolvendo XVAL e YVAL>
   RETURN
   END FUNCTION F
END MODULE FUNC
-------------------------------------------
```

```
FUNCTION FY(Y)
! Integra em X, para um valor fixo de Y.
USE FUNC
REAL :: FY
REAL, INTENT(IN) :: Y
YVAL= Y
FY= INTEGRA(F, LIMX)
RETURN
END FUNCTION FY
```

Com base nestas três unidades de programa, um programa pode calcular a integral sobre o retângulo no plano (x, y) através da chamada:

```
AREA= INTEGRA(FY, LIMY)
```

9.9 Atributo e declaração SAVE

A declaração ou o atributo SAVE são utilizados quando se deseja manter o valor de uma variável local em um subprograma após a saída. Desta forma, o valor anterior desta variável está acessível quando o subprograma é invocado novamente.

Como exemplo, a sub-rotina abaixo contém as variáveis locais CONTA, a qual conta o número de chamadas da sub-rotina e é inicializada a zero e a variável A, que é somada ao valor do argumento.

```
SUBROUTINE CONTA_SOMA(X)
IMPICIT NONE
REAL, INTENT(IN) :: X
REAL, SAVE :: A
INTEGER :: CONTA= 0 !Inicializa o contador.  Mantém o valor da variável.
...
CONTA= CONTA + 1
IF(CONTA == 1)THEN
   A= 0.0
ELSE
   A= A + X
END IF
...
RETURN
END SUBROUTINE CONTA_SOMA
```

Neste exemplo, tanto a variável A quanto CONTA têm o seu valor mantido quando a sub-rotina é abandonada. Isto é garantido porque a variável A é declarada com o atributo SAVE. Já a variável CONTA não precisa ser declarada com o mesmo atributo porque ela tem o seu valor inicializado na declaração. De acordo com o padrão da linguagem, as variáveis que têm o seu valor inicializado no momento da declaração automaticamente adquirem o atributo SAVE.

Alternativamente ao uso do atributo, pode-se declarar uma lista de nomes de variáveis com a mesma propriedade através da declaração SAVE. Desta forma, a variável A, no exemplo acima, podia ser declarada através das linhas:

```
REAL :: A
SAVE :: A
```

A forma geral desta declaração é:

```
SAVE [[::] <lista nomes variáveis>]
```

onde uma declaração SAVE sem a subsequente <lista nomes variáveis> equivale à mesma declaração aplicada a todos os nomes da rotina, em cujo caso nenhuma outra variável pode ser declarada com o atributo SAVE.

O atributo SAVE não pode ser especificado para um argumento mudo, um resultado de função ou um objeto automático (seção 9.3.6.3). O atributo pode ser especificado para um ponteiro, em cuja situação seu status de associação é preservado. Pode ser aplicado também a uma matriz alocável, em cuja situação o status de alocação e valores são preservados. Uma variável preservada em um subprograma recursivo é compartilhada por todas as instâncias da rotina.

O comando ou atributo SAVE podem aparecer no campo de declarações de um programa principal, mas neste caso o seu efeito é nulo. O uso prático deste recurso está restrito às outras unidades de programa.

Sugestões de uso & estilo para programação

O uso do atributo SAVE pode acarretar em uma perda de eficiência em certo processadores. Por esta razão, recomenda-se que o seu uso seja restrito somente àqueles objetos que o necessitam.

9.10 FUNÇÕES DE EFEITO LATERAL E ROTINAS PURAS

Na seção 9.2, foi mencionado que funções normalmente não alteram o valor de seus argumentos. Entretanto, esta e certas outras ações são na realidade permitidas. Funções que alteram os valores de seus argumentos, modificam os valores de dados globais definidos em módulos (seção 9.13.1), declaram variáveis locais com atributo SAVE ou executam operações de entrada/saída (capítulo 10) são denominadas *funções de efeito lateral (functions with side-effects)*. Diversas dessas operações, quando realizadas por sub-rotinas, também as classificam como efeitos laterais.

Para auxiliar na otimização do código, o padrão da linguagem estabelece uma proibição no uso de funções de efeito lateral quando o uso de tal função em uma expressão altera o valor de um outro operando na mesma expressão, seja este operando uma variável ou um argumento de uma outra função. Caso este tipo de função fosse possível neste caso, a atribuição

```
RES= FUN1(A,B,C) + FUN2(A,B,C)
```

onde ou FUN1 ou FUN2, ou ambas, alterassem o valor de um ou mais argumentos, a ordem de execução desta expressão seria importante; o valor de RES seria diferente caso FUN1 fosse calculada antes de FUN2 do que seria caso a ordem de cálculo fosse invertida. Exemplos de funções de efeito lateral são dados a seguir:

```
FUNCTION FUN1(A,B,C)
INTEGER :: FUN1
REAL, INTENT(INOUT) :: A
REAL, INTENT(IN)    :: B,C
A= A*A
FUN1= A/B
RETURN
END FUNCTION FUN1
-----------------------------------
FUNCTION FUN2(A,B,C)
INTEGER :: FUN2
REAL, INTENT(INOUT) :: A
REAL, INTENT(IN)    :: B,C
A= 2*A
FUN2= A/C
RETURN
END FUNCTION FUN2
```

Nota-se que ambas as funções alteram o valor de A; portanto, o valor de RES é totalmente dependente na ordem de execução da expressão.

Como o padrão da linguagem não estabelece uma ordem para a execução das operações e desenvolvimentos em uma expressão, este efeito lateral é proibido neste caso. Isto possibilita que os compiladores executem as operações na ordem que otimiza a execução do código.

O uso de funções de efeito lateral em comandos ou construtos acarretaria em um impedimento severo na otimização da execução do comando em um processador paralelo, efetivamente anulando o efeito desejado pela definição deste recurso.

Para controlar esta situação, o programador pode assegurar ao compilador que uma determinada rotina (não somente funções) não possui efeitos laterais ao incluir a palavra-chave `PURE` ao prefixo de `SUBROUTINE` ou `FUNCTION`:

```
PURE SUBROUTINE ...
---------------------------
PURE FUNCTION ...
```

Em termos práticos, esta palavra-chave assegura ao compilador que:

- se a rotina for uma função, esta não altera os seus argumentos mudos. Contudo, é permitido declarar argumentos mudos com o atributo `VALUE`, pois estes são efetivamente variáveis locais;
- a rotina não altera nenhuma parte de uma variável acessada por associação ao hospedeiro (rotina interna) ou associação por uso (módulos);
- a rotina não possui nenhuma variável local com o atributo `SAVE`;
- a rotina não executa operações em um arquivo externo;
- a rotina não contém o comando `STOP`;
- a rotina não contém instruções de controle de imagens.[4]

Para assegurar que estes requerimentos sejam cumpridos e que o compilador possa facilmente verificar o seu cumprimento, as seguintes regras adicionais são impostas:

- qualquer argumento mudo que seja o nome de uma rotina e qualquer rotina invocada devem também ser puras e ter a interface explícita;
- a intenção de um argumento mudo qualquer deve ser declarada, exceto caso este seja uma rotina ou um ponteiro, e a intenção deve sempre ser `IN` no caso de uma função;
- qualquer rotina interna de uma rotina pura também deve ser pura;
- uma variável, ou qualquer componente da mesma, que é acessada por associação ao hospedeiro ou por uso, ou é um argumento mudo de intenção `IN` ou é um objeto coindexado (corrays) não deve ser usado de forma a alterar seu valor ou status de associação de ponteiro, ou se tornar o alvo de um ponteiro.

Esta última regra assegura que um ponteiro local não pode causar um efeito lateral.

Uma rotina externa ou muda que seja usada como rotina pura deve possuir uma interface explícita que a caracterize inequivocamente como tal. Contudo, a rotina pode ser usada em outros contextos, quer sejam com o uso de um bloco de interface ou com uma interface que não a caracterize como pura. Isto permite que rotinas em bibliotecas sejam escritas como puras sem que elas sejam obrigatoriamente usadas como tal.

Ao contrário de funções puras, uma sub-rotina pura pode ter argumentos mudos com intenções `OUT` ou `INOUT` ou atributo `POINTER`. A principal razão para a existência de sub-rotinas puras está na possibilidade de se realizar atribuições definidas em um bloco `DO CONCURRENT` (seção 6.9). A sua existência também oferece a possibilidade de se fazer chamadas a sub-rotinas de dentro de funções puras.

Todas as funções intrínsecas (capítulo 8) são puras e, portanto, podem ser chamadas livremente de dentro de qualquer rotina pura. Adicionalmente, a sub-rotina intrínseca elemental `MVBITS` (seção 8.9.4) também é pura.

O atributo `PURE` é dado automaticamente a qualquer rotina que seja definida com o atributo `ELEMENTAL` (seção 9.11).

[4] Relacionado aos coarrays, que não são discutidos aqui.

9.11 ROTINAS ELEMENTAIS

Na seção 6.7.1 já foi introduzida a noção de rotinas intrínsecas elementais, as quais são rotinas com argumentos mudos escalares que podem ser invocadas com argumentos reais matriciais, desde que os argumentos matriciais tenham todos a mesma forma (isto é, que sejam conformáveis). Para uma função, a forma do resultado é a forma dos argumentos matriciais.

O Fortran estende este conceito a rotinas não intrínsecas. Uma rotina elemental criada pelo programador deve ter um dos seguintes cabeçalhos:

```
ELEMENTAL SUBROUTINE ...
-------------------------
ELEMENTAL FUNCTION ...
```

Um exemplo é fornecido abaixo. Dado o tipo derivado `INTERVALO`:

```
TYPE :: INTERVALO
   REAL :: INF, SUP
END TYPE INTERVALO
```

pode-se definir a seguinte função elemental:

```
ELEMENTAL FUNCTION SOMA_INTERVALOS(A,B)
INTRINSIC NONE
TYPE(INTERVALO)              :: SOMA_INTERVALOS
TYPE(INTERVALO), INTENT(IN) :: A, B
SOMA_INTERVALOS%INF= A%INF + B%INF
SOMA_INTERVALOS%SUP= A%SUP + B%SUP
RETURN
END FUNCTION SOMA_INTERVALOS
```

a qual soma dois intervalos de valores, entre um limite inferior e um limite superior. Nota-se que os argumentos mudos são escritos como escalares, mas a especificação `ELEMENTAL` possibilita o uso de matrizes conformáveis como argumentos reais. Neste caso, as operações de soma dos intervalos são realizadas elemento a elemento das matrizes `A` e `B` da maneira mais eficiente possível.

Uma rotina elemental deve satisfazer todos os requisitos de uma rotina pura; de fato, ela já possui automaticamente o atributo `PURE`. Qualquer um dos seus argumentos mudos deve ser uma variável escalar e não pode ser um coarray, um objeto alocável ou ponteiro. O resultado de uma função elemental deve ser também uma variável escalar que não é alocável, não é ponteiro e não pode possuir um parâmetro de tipo definido por uma expressão que não seja uma expressão constante.

Se o valor de um argumento mudo é usado em uma declaração, especificando o tamanho ou outra propriedade de alguma variável, como no exemplo

```
ELEMENTAL FUNCTION BRANCO(N)
IMPLICIT NONE
INTEGER, INTENT(IN) :: N
CHARACTER(LEN= N)   :: BRANCO
BRANCO= ' '
RETURN
END FUNCTION BRANCO
```

esta função *deve* ser chamada com argumento real *escalar*, uma vez que um argumento real matricial resultaria em uma matriz cujos elementos seriam caracteres de comprimentos variáveis.

Uma rotina externa elemental deve possuir uma interface explícita que sempre a caracterize de forma inequívoca como elemental. Isto é exigido para que o compilador possa determinar o mecanismo de chamada que acomode os elementos de matriz de forma mais eficiente. Isto contrasta com o caso de uma rotina pura, onde a interface nem sempre necessita caracterizar a rotina como pura.

No caso de uma sub-rotina elemental, se algum argumento real é matriz, todos os argumentos com intenções `OUT` ou `INOUT` devem ser matrizes. No exemplo abaixo,

```
ELEMENTAL SUBROUTINE TROCA(A,B)
IMPLICIT NONE
REAL, INTENT(INOUT) :: A, B
REAL :: TEMP
TEMP= A
A= B
B= TEMP
RETURN
END SUBROUTINE TROCA
```

Chamar esta sub-rotina com um argumento escalar e um matricial está obviamente errado, uma vez que o mecanismo para distribuir o valor da variável escalar por uma matriz conformável com o outro argumento não pode ser aplicado aqui.

Se a referência a uma rotina genérica (seção 9.13.4) é consistente tanto com uma rotina elemental quanto com uma não-elemental, a segunda versão é invocada, pois espera-se que a versão elemental de uma rotina rode, em geral, mais lentamente.

Finalmente, uma rotina elemental não pode ser usada como um argumento real de outra rotina.

Rotinas elementais impuras

Quando uma rotina não cumpre os requisitos de rotina pura, mas é desejado empregá-la de forma elemental, deve-se incluir o prefixo `IMPURE` no cabeçalho da rotina:

```
IMPURE ELEMENTAL SUBROUTINE ...
----------------------
IMPURE ELEMENTAL FUNCTION ...
```

Deve ser ressaltado que somente os requisitos para uma rotina pura são dispensáveis nesta situação (*i.e.,* a rotina pode exibir efeitos laterais). Os requisitos adicionais impostos a uma rotina elemental continuam sendo válidos.

9.12 Interfaces abstratas e ponteiros de rotinas

Nas seções anteriores, particularmente na seção 9.5, enfatizou-se repetidamente as vantagens de se fornecer interfaces explícitas para todos os subprogramas envolvidos na criação de um programa executável. Se as rotinas são internas ou são rotinas de módulo, as suas interfaces sempre serão explícitas; contudo, para rotinas externas, somente foi oferecido até o momento o recurso de blocos de interfaces os quais, embora tornem as interfaces explícitas, podem ser questionáveis do ponto de vista estético, pois aumentam a verbosidade na escrita do código por demandarem diversas linhas de declarações que, efetivamente, estarão repetindo as mesmas declarações incluídas nas definições das rotinas. Esta desvantagem estética se torna evidente, por exemplo, no programa `tes_plota`, listado na página 160.

Nesta seção serão introduzidos dois recursos que não somente reduzem a verbosidade envolvida na escrita do código-fonte, como são também recursos empregados nas técnicas de programação orientada a objetos, assunto que não será abordado nesta Apostila. Tratam-se das *interfaces abstratas* (*abstract interfaces*) e dos *ponteiros de rotinas* (*procedure pointers*).

9.12.1 Interfaces abstratas e declaração PROCEDURE

Frequentemente ocorre que exatamente a mesma interface é compartilhada por diversas rotinas distintas. Por isso, ao invés de se repetir diversas vezes o mesmo bloco de interfaces em diversas unidades distintas, o que contribui para o aumento na verbosidade do código, a informação contida em todas essas interfaces pode ser repetida uma única vez pela construção de um único bloco interfaces abstratas em uma unidade em particular.

Uma interface abstrata provê nomes genéricos para rotinas e seus argumentos mudos sem que esses nomes sejam realmente adotados por rotinas específicas. Essas interfaces abstratas podem ser declaradas em programas principais ou em módulos, de tal forma que as demais unidades do programa tenham acesso às informações fornecidas pelas mesmas. O recurso dos ponteiros de rotinas, discutidos na próxima seção, pode ser então empregado para vincular uma interface abstrata à interface de uma rotina que de fato existe. Uma interface genérica também pode ser empregada em uma declaração PROCEDURE, que será descrita a seguir.

Um bloco de interfaces abstratas contém a palavra-chave ABSTRACT, e cada campo de declarações de rotinas contido neste bloco define uma nova interface abstrata. Por exemplo, o bloco:

```
ABSTRACT INTERFACE
   SUBROUTINE SUB_SIMPLES_SEM_ARGS
   END SUBROUTINE SUB_SIMPLES_SEM_ARGS
!***
   REAL FUNCTION R2_TO_R(A, B)
   REAL, INTENT(IN) :: A, B
   END FUNCTION R2_TO_R
END INTERFACE
```

declara interfaces para a sub-rotina SUB_SIMPLES_SEM_ARGS (auto-explanatória) e para a função real R2_TO_R, contendo dois argumentos reais. Então, as declarações

```
PROCEDURE(SUB_SIMPLES_SEM_ARGS) :: SUB1, SUB2
PROCEDURE(R2_TO_R) :: MODULO, XYZ
```

declaram que SUB1 e SUB2 são sub-rotinas com interfaces idênticas a SUB_SIMPLES_SEM_ARGS e que MODULO e XYZ são funções com interfaces idênticas a R2_TO_R. Este tipo de declaração pode ser empregado, por exemplo em uma interface genérica, discutida na seção 9.13.4. É necessário ressaltar que as interfaces abstratas devem ser acessíveis à unidade onde as declarações PROCEDURE são realizadas.

Uma declaração PROCEDURE também pode ser empregada com qualquer rotina específica que contenha uma interface explícita. Por exemplo, se a função FUN existe e possui uma interface explícita, então

```
PROCEDURE(FUN) :: FUN2
```

declara que a função FUN2 (a qual também existe) possui interface idêntica à FUN.

A declaração também pode ser usada para declarar rotinas com interfaces implícitas ou com somente a especificação do tipo de uma função. Por exemplo:

```
PROCEDURE() :: X
PROCEDURE(REAL) :: Y
PROCEDURE(COMPLEX(KIND(0.0D0))) :: Z
```

declaram que X é uma rotina com interface implícita, que Y é uma função do tipo real e que Z é uma função complexa de dupla precisão. Estas declarações são equivalentes a

```
EXTERNAL :: X
REAL, EXTERNAL :: Y
COMPLEX(KIND(0.0D0)), EXTERNAL :: Z
```

A sintaxe completa da declaração PROCEDURE é:

```
PROCEDURE([<proc-interface>]) [[, <proc-atrib-spec>] ::] <proc-decl-list>
```

onde <proc-interface> indica uma interface explícita, abstrata ou não, <proc-atrib-spec> é um dos seguintes atributos:

```
BIND(C[, NAME= <string>])   POINTER     PUBLIC
INTENT([IN][OUT][INOUT])    PRIVATE     SAVE
OPTIONAL                    PROTECTED
```

e <proc-decl-list> é uma lista de declarações de rotinas, sendo que cada declaração é

```
<nome-rotina> [=> <ponteiro-rotina-init>]
```

sendo <nome-rotina> um nome válido em Fortran e <ponteiro-rotina-init> é uma referência à função intrínseca NULL() (seção 8.16) ou ao nome de uma rotina externa não elemental ou a uma rotina de módulo.

Cada atributo em <proc-atrib-spec> é válido para todas as rotinas na <proc-decl-list>. Os atributos INTENT, PROTECTED e SAVE e a inicialização (realizada via =>) somente podem ser empregados se as rotinas são ponteiros. O atributo BIND é específico para interoperacionalidade com a linguagem C, o que não será discutido nesta Apostila.

9.12.2 Ponteiros de rotinas

Ponteiros, conforme foram discutidos nas seções e capítulos anteriores, particularmente na seção 7.2, foram sempre associados a objetos de dados. Um *ponteiro de rotina* (*procedure pointer*) é um ponteiro que pode ser associado com uma rotina, em vez de um objeto de dados. Sua interface pode ser explícita ou implícita e sua associação com um alvo ocorre como com uma rotina muda (seção 9.3.7); assim, sua interface não pode ser genérica ou elemental.

9.12.2.1 Ponteiros de rotinas nomeados

Para se declarar um ponteiro de rotina, é necessário especificar que essa entidade é simultaneamente uma rotina e um ponteiro. Isto pode ser realizado de uma maneira explícita, como em:

```
POINTER :: SP
INTERFACE
   SUBROUTINE SP(A, B)
   REAL, INTENT(INOUT) :: A
   REAL, INTENT(IN) :: B
   END SUBROUTINE SP
END INTERFACE
REAL, EXTERNAL, POINTER :: FP
```

os quais declaram que SP é um ponteiro para uma sub-rotina com a interface acima e que FP é um ponteiro a uma função real com uma interface implícita.

As mesmas declarações podem ser realizadas com a declaração PROCEDURE, incluindo o atributo POINTER, como em

```
PROCEDURE(R2_TO_R), POINTER :: PR2 => NULL()
PROCEDURE(SP), POINTER :: PSP => NULL()
```

nas quais PR2 é declarado ser um ponteiro para uma função com interface idêntica à interface abstrata R2_TO_R, ao passo que PSP é um ponteiro para uma sub-rotina com interface idêntica a SP. Ambos os ponteiros são inicializados com a função NULL(), o que imediatamente confere o status de desassociados aos ponteiros.

Caso a função FF1 e as subrotinas SUB1 e SUB2 existam e tenham interfaces compatíveis, as atribuições de ponteiros

```
PR2 => FF1 ; PSP => SUB1
```

são válidas e as seguintes linhas de comando são equivalentes:

```
Y= FF1(A, B)
Y= PR2(A, B)   ! Equivalente à linha acima
CALL SUB1(A, B)
CALL PSP(A, B) ! Equivalente à linha acima
PSP => SUB2    ! PSP aponta agora para SUB2
CALL SUB2(A, B)
CALL PSP(A, B) ! Equivalente à linha acima
```

Uma rotina interna pode ser o alvo de um ponteiro de rotina. Quando a rotina interna é invocada via o nome do ponteiro, este tem acesso às variáveis da unidade hospedeira. Quando a rotina hospedeira retorna, o ponteiro se torna indefinido

9.12.2.2 PONTEIROS DE ROTINAS COMO COMPONENTES DE TIPOS DERIVADOS

Um componente de um tipo derivado pode ser um ponteiro de rotina. A declaração de componente deve ser realizada com a declaração PROCEDURE.

O tipo derivado definido abaixo possibilita a criação de uma lista encadeada (seção 7.4.1) contendo um número arbitrário de rotinas (todas com a mesma interface). As distintas rotinas específicas serão acessadas como componentes do tipo derivado:

```
TYPE :: PROCESS_LIST
   PROCEDURE(PROC_INTERFACE), POINTER :: PROC
   TYPE(PROCESS_LIST), POINTER        :: NEXT => NULL()
END TYPE PROCESS_LIST
!***
ABSTRACT INTERFACE
   SUBROUTINE PROC_INTERFACE(...)
   END SUBROUTINE PROC_INTERFACE
END INTERFACE
```

Componentes deste tipo que são ponteiros de rotinas podem ser associadas com outros ponteiros, passadas como argumentos reais de rotinas ou invocadas diretamente, como em:

```
TYPE(PROCESS_LIST) :: X, Y(10)
PROCEDURE(PROC_INTERFACE), POINTER :: P => NULL()
...
P => X%PROC
CALL OUTRA_SUBROTINA(X%PROC)
CALL Y(I)%PROC(...)
```

9.13 MÓDULOS

Módulos são o terceiro tipo de unidade de programa a ser discutido. Um módulo é um recurso muito poderoso para transferir dados entre subprogramas e para organizar a arquitetura global de um programa grande e complexo.

A funcionalidade de um módulo pode ser explorada por qualquer unidade de programa que deseja fazer uso (através de uma instrução USE) dos recursos disponibilizados por este. Os recursos que podem ser incluídos em um módulo são os seguintes:

Declarações de objetos globais. Se dados globais são necessários, por exemplo para transferir valores entre distintas unidades de programa sem que estes estejam em listas de argumentos de rotinas, então estes são são tornados visíveis sempre que o módulo é usado. Objetos em um módulo podem ser inicializados com valores estáticos, de tal forma que estes mantêm seu valor em diferentes unidades que usam o mesmo módulo.

Blocos de interfaces. Na maior parte das situações é vantajoso congregar todos os blocos de interfaces em um ou poucos módulos e então usar o módulo sempre que uma interface explícita for necessária. Isto pode realizado em conjunto com a cláusula ONLY, discutida abaixo.

Rotinas de módulos. Rotinas podem ser definidas internamente em um módulo, sendo estas tornadas acessíveis a qualquer unidade de programa que USE o módulo. Esta estratégia é mais vantajosa que usar uma rotina externa porque em um módulo a interface das rotinas contidas no mesmo é sempre explícita.

Acesso controlado a objetos. Variáveis, rotinas e declarações de operadores podem ter a sua visibilidade controlada por declarações ou atributos de acesso dentro de um módulo. Desta forma, é possível especificar que um determinado objeto seja visível somente no interior do módulo. Este recurso é frequentemente utilizado para impedir que um usuário possa alterar o valor de um objeto de âmbito puramente interno ao módulo.

Interfaces genéricas. Um módulo pode ser usado para definir um nome genérico para um conjunto de rotinas, com interfaces distintas, mas que executem todas a mesma função. Interfaces genéricas podem ser usadas também para estender a funcionalidade de rotinas intrínsecas.

Sobrecarga de operadores. Um módulo pode ser usado também para redefinir a operação executada pelos operadores intrínsecos da linguagem (= + - * / **) ou para definir novos tipos de operadores.

Extensão semântica. Um módulo de extensão semântica é uma coleção de definições de tipos derivados, rotinas e operadores sobrecarregados tais que, quando agregados a uma unidade de programa, permitem que o usuário use esta funcionalidade estendida como se fizesse parte da linguagem padrão.

A forma geral de um módulo é:

```
MODULE <nome-módulo>
<listas-especificações>
[CONTAINS
   <rotinas-de-módulo>]
END [MODULE [<nome-módulo>]]
```

Os diferentes usos de um módulo serão agora abordados.

9.13.1 Dados globais

Variáveis declaradas no âmbito de um certa unidade de programa são usualmente entidades locais, porque estas normalmente são necessárias somente dentro deste escopo. Contudo, em muitas situações é necessário que alguns objetos tenham o seu valor compartilhado entre diferentes unidades de programa. A maneira mais usual de se realizar este compartilhamento é através da lista de argumentos de uma rotina (seção 9.3). Entretanto, facilmente surge uma situação onde uma variável não pode ser compartilhada como argumento de um subprograma. Neste momento, é necessária uma outra estratégia para se definir dados globais.

O uso de um módulo fornece um mecanismo centralizado de definição de objetos globais. Por exemplo, suponha que se queira ter acesso às variáveis inteiras `I`, `J` e `K` e às variáveis reais `A`, `B` e `C` em diferentes unidades de programa. Um módulo que permitirá o compartilhamento destas variáveis é o seguinte:

```
MODULE GLOBAIS
IMPLICIT NONE
INTEGER :: I, J, K
REAL    :: A, B, C
END MODULE GLOBAIS
```

Estas variáveis se tornam acessíveis a outras unidades de programa (inclusive outros módulos) através da instrução `USE`, isto é:

```
USE GLOBAIS
```

A instrução `USE` é não executável e deve ser inserida logo após o cabeçalho da unidade de programa (`PROGRAM`, `FUNCTION`, `SUBROUTINE` ou `MODULE`) e antes de qualquer outra instrução não executável, tal como uma declaração de variáveis. Uma unidade de programa pode invocar um número arbitrário de módulos usando uma série de instruções `USE`. Um módulo pode usar outros módulos, porém um módulo não pode usar a si próprio, seja de forma direta ou indireta. Assim, o módulo `GLOBAIS` pode ser usado da seguinte forma:

```
FUNCTION USA_MOD(X)
USE GLOBAIS
IMPLICIT NONE
REAL              :: USA_MOD
REAL, INTENT(IN) :: X
USA_MOD= I*A - J*B + K*C - X
RETURN
END FUNCTION USA_MOD
```

Um módulo é um recurso eficiente para o compartilhamento de dados globais. A definição dos nomes, tipos e espécies das variáveis é realizada em uma única unidade de programa, a qual é simplesmente usada por outras, estabelecendo assim um controle centralizado sobre os objetos.

O uso de variáveis de um módulo pode causar problemas se o mesmo nome for usado para diferentes variáveis em partes diferentes de um programa. A declaração `USE` pode evitar este problema ao permitir a especificação de um nome local distinto daquele definido no módulo, porém que ainda permita o acesso aos dados globais. Por exemplo,

```
...
USE GLOBAIS, R => A, S => B
...
USA_MOD= I*R - J*S + K*C - X
...
```

Aqui, os nomes `R` e `S` são usado para acessar as variáveis globais `A` e `B`, definidas no módulo, caso estes últimos nomes sejam usados para diferentes variáveis na mesma unidade de programa. O vocábulo léxico "=>" promove a ligação do nome local com o nome no módulo. Tudo se passa como se `R` fosse o *apelido* (*alias*) de `A` e `S` o apelido de B nesta unidade de programa. Os ponteiros em Fortran também atuam como apelidos, porém, neste caso, pode-se usar tanto o nome verdadeiro de uma variável quanto o nome de um ponteiro que aponta a ela.

Existe também uma forma da declaração `USE` que limita o acesso somente a certos objetos dentro de um módulo. Este recurso não é exatamente igual ao uso dos atributos `PUBLIC` ou `PRIVATE` (seção 9.13.3), uma vez que ele atua somente na unidade que acessa o módulo, e não sobre o módulo em geral. O recurso requer o uso do qualificador `ONLY`, seguido de dois pontos ":" e uma `<lista-only>`:

```
USE <nome-módulo>, ONLY: <lista-only>
```

Por exemplo para tornar somente as variáveis `A` e `C` do módulo `GLOBAIS` acessíveis em uma dada unidade de programa, a declaração fica:

```
...
USE GLOBAIS, ONLY: A, C
...
```

Os dois últimos recursos podem ser também combinados:

```
...
USE GLOBAIS, ONLY: R => A
...
```

para tornar somente a variável `A` acessível, porém com o nome `R`. Uma unidade de programa pode ter mais de uma declaração `USE` referindo-se ao mesmo módulo. Por conseguinte, deve-se notar que que um `USE` com o qualificador `ONLY` não cancela uma declaração `USE` menos restritiva.

Um uso frequente de módulos para armazenar dados globais consiste em definições de parâmetros de espécie de tipo, constantes universais matemáticas e/ou físicas e outros objetos estáticos, como está apresentado no programa-exemplo `mod1` (programa 9.5), o qual ilustra o uso de um módulo para armazenar dados globais.

Torna-se possível, portanto, a criação de módulos destinados a armazenar valores de constantes fundamentais, tanto matemáticas quanto físicas. O módulo `Const_Fund` (programa 9.6) consiste em um exemplo que pode ser empregado em diversas aplicações distintas. O módulo usa o módulo intrínseco `iso_fortran_env`, o qual será abordado na seção 9.13.6.

9.13.2 ROTINAS DE MÓDULOS

Rotinas podem ser definidas em módulos e estas são denominadas *rotinas de módulos* (*module procedures*). Estas podem ser tanto sub-rotinas quanto funções e ter a mesma forma de

Listagem 9.5: Ilustra uso de modulo para armazenar dados globais.

```
MODULE modul
implicit none
real, parameter :: pi= 3.1415926536
real, parameter :: euler_e= 2.718281828
END MODULE modul
!****************************************
program mod1
use modul
implicit none
real :: x
do
   print*, "Entre com o valor de x:"
   read*, x
   print*, "sen(pi*x)=",sen()
   print*, "ln(e*x)=",ln()
end do
CONTAINS
   function sen()
   real :: sen
   sen= sin(pi*x)
   return
   end function sen
!
   function ln()
   real :: ln
   ln= log(euler_e*x)
   return
   end function ln
end program mod1
```

rotinas internas definidas dentro de outras unidades de programa. A forma genérica das rotinas de módulo foi apresentada na seção 9.2.1. O número de rotinas de módulo é arbitrário.

Rotinas de módulo podem ser chamadas usando a instrução `CALL` usual ou fazendo referência ao nome de uma função. Contudo, estas somente são acessíveis a unidades de programa que fazem uso do módulo através da instrução `USE`.

Uma rotina de módulo pode invocar outras rotinas de módulo contidas no mesmo módulo. As variáveis declaradas no módulo antes da palavra-chave `CONTAINS` são diretamente acessíveis a todas as rotinas deste, por associação ao hospedeiro. Contudo, variáveis declaradas localmente em um determinado subprograma de módulo são opacas aos outros subprogramas. Caso o módulo invoque outro módulo com uma instrução `USE` antes da palavra-chave `CONTAINS`, estes objetos também se tornam acessíveis a todas as rotinas de módulo. Por outro lado, uma determinada rotina de módulo pode usar localmente outro módulo (não o hospedeiro), em cuja situação os objetos somente são acessíveis localmente. Em suma, uma rotina de módulo possui todas as propriedades de rotinas internas em outras unidades de programa, exceto que uma rotina de módulo pode conter, por sua vez, rotinas internas a ela. Por exemplo, o seguinte módulo é válido:

```
MODULE INTEG
IMPLICIT NONE
REAL :: ALFA  ! Variável global.
CONTAINS
   FUNCTION FF(X)
   REAL :: FF
   REAL, INTENT(IN) :: X
   FF= EXP(-ALFA*X*X)
   FF= FF*X2(X)
```

Listagem 9.6: Modulo contendo constantes matemáticas e físicas universais.

```
!*** Module Const_Fund ***
!  Define espécies de tipo padrões e constantes universais.
!  Parâmetros de espécies: sp -> precisão simples
!                          dp -> precisão dupla
!                          qp -> precisão estendida ou quádrupla
MODULE Const_Fund
use, intrinsic :: iso_fortran_env, only: sp => real32, dp => real64, &
                                         qp => real128
implicit none
! Constantes matemáticas (precisão simples)
real(sp), parameter :: pi_s=    3.14159265358979323846264338327950288419717_sp
real(sp), parameter :: pid2_s=  1.57079632679489661923132169163975144209858_sp
real(sp), parameter :: twopi_s= 6.28318530717958647692528676655900576839434_sp
real(sp), parameter :: rtpi_s=  1.77245385090551602729816748334114518279755_sp
real(sp), parameter :: sqrt2_s= 1.41421356237309504880168872420969807856967_sp
! Constantes matemáticas (precisão dupla)
real(dp), parameter :: pi=      3.14159265358979323846264338327950288419717_dp
real(dp), parameter :: pid2=    1.57079632679489661923132169163975144209858_dp
real(dp), parameter :: twopi=   6.28318530717958647692528676655900576839434_dp
real(dp), parameter :: rtpi=    1.77245385090551602729816748334114518279755_dp
real(dp), parameter :: sqrt2=   1.41421356237309504880168872420969807856967_dp
real(dp), parameter :: euler_e= 2.71828182845904523536028747135266249775725_dp
!Constantes complexas (dupla precisão)
complex(kind= dp), parameter :: z1= (1.0_dp,0.0_dp), zi= (0.0_dp,1.0_dp)
!
!  Constantes físicas fundamentais (SI), medidas pelo NIST (valores de 2018):
!   http://physics.nist.gov/cuu/Constants/index.html
real(dp), parameter :: me= 9.1093837015e-31_dp  !Massa de repouso do elétron
real(dp), parameter :: mp= 1.67262192369e-27_dp !Massa de repouso do próton
real(dp), parameter ::  e= 1.602176634e-19_dp   !Carga elétrica elementar
END MODULE Const_Fund
```

```
      RETURN
      CONTAINS
         FUNCTION X2(Y)
         REAL :: X2
         REAL, INTENT(IN) :: Y
         X2= Y**2
         RETURN
         END FUNCTION X2
      END FUNCTION FF
   END MODULE INTEG
```

A outra grande diferença está no fato de que as interfaces das rotinas de módulo são explícitas no âmbito deste. Todavia, quando uma outra unidade de programa usa o módulo, todas os objetos públicos nele contidos se tornam acessíveis a esta, resultando que as interfaces das rotinas de módulo são automaticamente explícitas também para a unidade de programa que o invoca. Portanto, um módulo é considerado a unidade ideal para armazenar grupos de subprogramas criados para desempenhar uma determinada tarefa, juntamente com os objetos globais associados a estes subprogramas.

Ao se compilar um arquivo contendo um módulo, os compiladores automaticamente geram um novo tipo de arquivo, geralmente com a extensão `*.mod`, onde as informações contidas no módulo, tais como variáveis globais e interfaces, são armazenadas. Quando uma outra unidade de programa faz referência a este módulo, o compilador busca estas informações no arquivo `.mod` . Desta forma, cria-se um mecanismo para verificar se as variáveis e as interfaces estão sendo corretamente utilizadas pela unidade que chama o módulo. Quando o módulo não define

um rotina interna, não é necessário guardar o arquivo objeto (*.o ou *.obj) associado, bastando guardar o arquivo *.mod. Por esta razão, muitos compiladores oferecem uma chave extra de compilação através da qual nenhum arquivo objeto é criado, mas somente o arquivo de módulo.

Rotinas de módulo podem ser úteis por diversas razões. Por exemplo, um módulo que define a estrutura de um conjunto particular de dados pode também incluir rotinas especiais, necessárias para operar com estes dados; ou um módulo pode ser usado para conter uma biblioteca de rotinas relacionadas entre si.

Como exemplo, um módulo pode ser usado para executar a *adição* de variáveis de tipo derivado:

```
MODULE MOD_PONTO
IMPLICIT NONE
TYPE :: PONTO
   REAL :: X, Y
END TYPE PONTO
CONTAINS
   FUNCTION ADPONTOS(P,Q)
   TYPE(POINT) :: ADPONTOS
   TYPE(POINT), INTENT(IN) :: P, Q
   ADPONTOS%X= P%X + Q%X
   ADPONTOS%Y= P%Y + Q%Y
   RETURN
   END FUNCTION ADPONTOS
END MODULE MOD_PONTO
```

Neste caso, o programa principal usa este módulo:

```
PROGRAM P_PONTO
USE MOD_PONTO
IMPLICIT NONE
TYPE(POINT) :: PX, PY, PZ
...
PZ= ADPONTO(PX,PY)
...
END PROGRAM P_PONTO
```

O recurso avançado de *sobrecarga de operador (operator overloading)* permite redefinir o operador de adição (+) dentro do âmbito do módulo M_PONTO, de tal forma que o processo de adição de duas variáveis do tipo PONTO automaticamente iria chamar a função ADPONTO. Desta forma, ao invés do programa chamar esta função, bastaria realizar a operação:

```
PZ= PX + PY
```

Entretanto, este recurso não será discutido aqui.

O programa-exemplo mod2 (programa 9.7) é igual ao programa 9.5, porém usando rotinas de módulo.

9.13.3 Controle de acesso aos objetos em módulos

Exceto quando explicitamente determinado, todas as entidades em um módulo são acessíveis para qualquer unidade de programa que chame este módulo com a instrução USE. Contudo, este acesso pode ser controlado por meio dos atributos/declarações PUBLIC, PRIVATE e PROTECTED.

Em certas situações é recomendável proibir o uso de certas entidades (objetos globais e/ou rotinas) contidas no módulo para permitir ao usuário acesso somente às rotinas de módulo que realmente deveriam ser acessíveis a este, ou para permitir flexibilidade no aperfeiçoamento das entidades contidas no módulo sem haver a necessidade de informar o usuário a respeito destes aperfeiçoamentos.

Este controle é exercido através dos atributos ou declarações PUBLIC ou PRIVATE. Por exemplo, abaixo temos a declarações de variáveis com dois atributos distintos:

Listagem 9.7: Ilustra o uso de rotinas de modulo.

```
MODULE modu2
implicit none
real, parameter :: pi= 3.1415926536
real, parameter :: euler_e= 2.718281828
real :: x
CONTAINS
   function sen()
   real :: sen
   sen= sin(pi*x)
   return
   end function sen
!***
   function ln()
   real :: ln
   ln= log(euler_e*x)
   return
   end function ln
END MODULE modu2
!****************************************
program mod2
use modu2
implicit none
do
   print*, "Entre com o valor de x:"
   read*, x
   print*, "sen(pi*x)=",sen()
   print*, "ln(e*x)=",ln()
end do
end program mod2
```

```
REAL, PUBLIC :: X, Y, Z
INTEGER, PRIVATE :: U, V, W
```

Dentro deste conjunto de variáveis, somente `X`, `Y` e `Z` são acessíveis a uma unidade de programa que use este módulo.

Outra maneira de se estabelecer o controle de acesso é através de declarações, as quais listam os nomes dos objetos que são públicos ou privados:

```
PUBLIC :: X, Y, Z
PRIVATE :: U, V, W
```

A forma geral da declaração é:

```
PUBLIC [[::] <lista-acesso>]
PRIVATE [[::] <lista-acesso>]
```

Caso nenhum controle seja explicitamente estabelecido em um módulo, seja através de um atributo ou de uma declaração, todas as entidades têm o atributo `PUBLIC`. Se uma declaração `PUBLIC` ou `PRIVATE` não possui uma lista de entidades, esta confirma ou altera o acesso padrão. Assim, a declaração

```
PUBLIC
```

confirma o acesso padrão, ao passo que a declaração

```
PRIVATE
```

altera o acesso padrão. Desta forma, uma sequência de declarações como as abaixo:

```
...
PRIVATE
PUBLIC <lista-acesso>
...
```

confere aos nomes na <lista-acesso> o atributo PUBLIC enquanto que todas as entidades restantes no módulo são privadas, podendo ser acessadas somente dentro do âmbito do módulo.

Em outras situações, pode ser vantajoso permitir o acesso a certas entidades de um módulo de uma maneira protegida; isto é, a unidade que usa o módulo pode fazer referência a essas entidades, mas não pode alterar os seus valores. Este tipo de controle de acesso é realizado pelo atributo/declaração PROTECTED. Este atributo não afeta a visibilidade do objeto, o qual ainda precisa ser público para ser visível, mas confere a mesma proteção contra alterações do seu valor fornecida pelo atributo INTENT(IN) para argumentos mudos. Este atributo somente pode ser empregado em objetos declarados em módulos.

Exemplos são:

```
MODULE MOD
PUBLIC
REAL, PROTECTED :: V= 10.5
INTEGER :: I= 100
PROTECTED :: I
END MODULE MOD
```

Neste exemplo, V é uma variável real e I é uma variável inteira, ambas públicas e protegidas. Assim, uma unidade de programa que use o módulo MOD poderá fazer referência a V e I mas não poderá alterar os seus valores.

Variáveis com atributo PROTECTED podem ter seu valor alterado, mas somente por outras entidades contidas no mesmo módulo (rotinas de módulo, por exemplo). Neste sentido, a ação do atributo PROTECTED é distinto da ação do atributo PARAMETER. Em outras unidades, essas variáveis não podem aparecer em contextos que possam alterar os seus valores, como no lado esquerdo de atribuições.

9.13.4 Interfaces e rotinas genéricas

Um outro recurso poderoso no Fortran é a habilidade do programador definir suas próprias *rotinas genéricas* (*generic procedures*), de tal forma que um único nome é suficiente para invocar uma determinada rotina, enquanto que a ação que realmente é executada quando este nome é usado depende dos tipos e espécies de seus argumentos. Embora possam ser declaradas em quaisquer unidades de programa, rotinas genéricas são usualmente definidas em módulos.

Uma rotina genérica é definida usando-se um bloco de interfaces e um nome genérico é usado para todas as rotinas definidas dentro deste bloco de interfaces. Assim, a forma geral é:

```
INTERFACE <nome-genérico>
   <bloco-interface-rotina-específica-1>
   <bloco-interface-rotina-específica-2>
   ...
END INTERFACE [<nome-genérico>]
```

onde <bloco-interface-rotina-específica-1>, etc, são os blocos das interfaces das rotinas específicas, isto é, que se referem a um dado conjunto de tipos e espécies de variáveis, e que são acessadas através do <nome-genérico>. As rotinas *per se* podem se encontrar em outras unidades; por exemplo, elas podem ser rotinas externas.

Como um módulo é a unidade de programa ideal para armazenar todas estas rotinas específicas relacionadas entre si e como as interfaces das rotinas de módulo são sempre explícitas, estas rotinas genéricas são, em geral, definidas dentro de módulos. Neste caso, a declaração

```
MODULE PROCEDURE <lista-nomes-rotinas>
```

é incluída no bloco genérico de interfaces para nomear as rotinas de módulo que são referidas através do nome genérico. Assim, a declaração geral é:

```
INTERFACE <nome-genérico>
   [<blocos-interfaces>]
   [MODULE PROCEDURE <lista-nomes-rotinas>]
! Em Fortran blocos de interfaces e declarações MODULE PROCEDURE
!   podem aparecer em qualquer ordem.
END INTERFACE [<nome-genérico>]
```

Deve-se notar que todos os nomes na `<lista-nomes-rotinas>` devem ser de rotinas de módulo acessíveis; portanto, elas não necessariamente devem estar definidas no mesmo módulo onde a interface genérica é estabelecida, bastando que elas sejam acessíveis por associação de uso.

Para demonstrar o poderio de uma interface genérica, será utilizada novamente a sub-rotina `TROCA`, a qual foi definida primeiramente na página 157 e depois, na forma de uma sub-rotina elemental, na seção 9.11. O módulo `gentroca` abaixo define o nome genérico de uma série de sub-rotinas elementais `TROCA` associadas a variáveis dos tipos real, inteiro, lógico e do tipo derivado `ponto`, o qual é definido no mesmo módulo.

```
! Define nome genérico para troca de duas variáveis quaisquer.
MODULE gentroca
implicit none
type :: ponto
   real :: x,y
end type ponto

INTERFACE troca
   MODULE PROCEDURE troca_ponto, troca_real, troca_int, troca_log
END INTERFACE troca

CONTAINS
   elemental subroutine troca_ponto(a,b)
   type(ponto), intent(inout) :: a, b
   type(ponto) :: temp
   temp= a
   a= b
   b= temp
   end subroutine troca_ponto
!***
   elemental subroutine troca_real(a,b)
   real, intent(inout) :: a, b
   real :: temp
   temp= a
   a= b
   b= temp
   end subroutine troca_real
!***
   elemental subroutine troca_int(a,b)
   integer, intent(inout) :: a, b
   integer :: temp
   temp= a
   a= b
   b= temp
   end subroutine troca_int
!***
   elemental subroutine troca_log(a,b)
   logical, intent(inout) :: a, b
   logical :: temp
   temp= a
   a= b
   b= temp
   end subroutine troca_log
END MODULE gentroca
```

Este módulo é utilizado então em duas situações distintas. Na primeira vez, o módulo será utilizado para trocar o valor de duas variáveis escalares do tipo `ponto`, como no programa `usa_gentroca` abaixo.

```
program usa_gentroca
use gentroca
type(ponto) :: b= ponto(1.0,0.0), c= ponto(1.0,1.0)
print*, 'Valores originais:'
print*, b,c
call troca(b,c)
print*, 'Novos valores:'
print*, b,c
end program usa_gentroca
```

Posteriormente, o mesmo módulo será utilizado para trocar os elementos de duas matrizes inteiras, como no programa `usa_gt_mat` abaixo.

```
program usa_gt_mat
use gentroca
integer, dimension(2,2) :: b, c
b= reshape(source= [ ((i+j, i= 1,2), j= 1,2) ], shape= [2,2])
c= reshape(source= [ ((i+j, i= 3,4), j= 3,4) ], shape= [2,2])
print*, 'Valores originais:'
do i= 1, 2
   print*, b(i,:),"     ",c(i,:)
end do
call troca(b,c)
print*, 'Novos valores:'
do i= 1, 2
   print*, b(i,:),"     ",c(i,:)
end do
end program usa_gt_mat
```

Uma declaração `MODULE PROCEDURE` somente é permitida se um `<nome-genérico>` for fornecido. Porém, o nome genérico pode ser igual ao nome de uma das rotinas declaradas na `<lista-nomes-rotinas>`. Em conjunto com esta propriedade, o nome genérico definido em um módulo pode ser o mesmo de outro nome genérico acessível ao módulo, inclusive no caso onde o nome genérico corresponde ao de uma rotina intrínseca. Neste caso, o módulo estará estendendo o intervalo de aplicação de uma rotina intrínseca.

As rotinas às quais são dadas um certo nome genérico devem ser todas ou sub-rotinas ou funções, incluindo as intrínsecas quando uma rotina intrínseca é estendida. Quaisquer duas rotinas não intrínsecas associadas ao mesmo nome genérico devem ter argumentos que diferem de tal forma que qualquer invocação é feita de forma inequívoca. As regras são que:

1. uma delas tenha mais argumentos obrigatórios mudos de um tipo, espécie e posto particulares que a outra ou
2. que ao menos uma delas tenha um argumento mudo obrigatório tal que
   (a) corresponda por posição na lista de argumentos a um argumento mudo que não esteja presente na outra, ou esteja presente com um tipo e/ou espécie distinta ou com posto distinto, e
   (b) corresponda por nome a um argumento mudo que não esteja presente na outra, ou presente com tipo e/ou espécie diferente ou com posto diferente.

Para o caso (2), ambas as regras são necessárias para descartar a possibilidade de invocação ambígua por uso de palavras-chave. Como exemplo onde haverá ambiguidade, o exemplo abaixo:

```
!Exemplo de definição ambígua de nome genérico
INTERFACE F
   MODULE PROCEDURE FXI, FIX
END INTERFACE F
```

```
CONTAINS
   FUNCTION FXI(X,I)
   REAL :: FXI
   REAL, INTENT(IN) :: X
   INTEGER, INTENT(IN) :: I
   ...
   END FUNCTION FXI
!***
   FUNCTION FIX(I,X)
   REAL :: FIX
   REAL, INTEN(IN) :: X
   INTEGER, INTENT(IN) :: I
   ...
   END FUNCTION FIX
```

Neste caso, a chamada ao nome genérico `F` não será ambígua no caso de argumentos posicionais:

```
A= F(INT,VAR)
```

porém será ambígua para chamada usando palavras-chave:

```
A= F(I= INT, X= VAR)
```

Se uma invocação genérica é ambígua entre uma rotina intrínseca e uma não-intrínseca, esta última é sempre invocada.

9.13.5 ESTENDENDO ROTINAS INTRÍNSECAS *via* BLOCOS DE INTERFACE GENÉRICOS

Como já foi mencionado, uma rotina intrínseca pode ser estendida ou também redefinida. Uma rotina intrínseca *estendida* suplementa as rotinas intrínsecas específicas já existentes. Uma rotina intrínseca *redefinida* substitui uma rotina intrínseca específica existente. Desta forma é possível ampliar um determinado cálculo de uma função, por exemplo, para interfaces não previstas pelo padrão da linguagem (*extensão*) e/ou substituir o processo de cálculo do valor da função por um código distinto daquele implementado no compilador (*substituição*).

Quando um nome genérico é igual ao nome genérico de uma rotina intrínseca e o nome é declarado com o atributo ou declaração `INTRINSIC` (ou aparece em um contexto intrínseco), a interface genérica estende a rotina genérica intrínseca.

Quando um nome genérico é igual ao nome genérico de uma rotina intrínseca e este nome não possui o atributo `INTRINSIC` (nem possui este atributo pelo contexto), a rotina genérica redefine a rotina genérica intrínseca.

Como exemplo, o módulo `EXT_SINH` abaixo estende o cálculo da função seno hiperbólico (seção 8.4) para um argumento escalar complexo. A biblioteca padrão da linguagem suporta argumentos complexos para a função seno hiperbólico; assim, o exemplo abaixo é apenas um exercício de extensão sintática.

```
MODULE EXT_SINH
IMPLICIT NONE
INTRINSIC :: SINH
!
INTERFACE SINH
   MODULE PROCEDURE SINH_C
END INTERFACE SINH
!
CONTAINS
   FUNCTION SINH_C(Z)
   COMPLEX             :: SINH_C
   COMPLEX, INTENT(IN) :: Z
   REAL :: X,Y
   X= REAL(Z)
```

```
    Y= AIMAG(Z)
    SINH_C= CMPLX(SINH(X)*COS(Y),COSH(X)*SIN(Y))
    RETURN
    END FUNCTION SINH_C
END MODULE EXT_SINH
```

9.13.6 Módulos intrínsecos

Um módulo intrínseco é aquele fornecido pela biblioteca padrão da linguagem, ao invés de ter sido escrito por um usuário. Até o presente momento, o padrão fornece cinco módulos intrínsecos:

`ISO_FORTRAN_ENV` `IEEE_ARITHMETIC` `IEEE_EXCEPTIONS` `IEEE_FEATURES` `ISO_C_BINDING`

Destes, somente o primeiro será abordado nesta seção. Os três seguintes se referem aos recursos de controle de exceções de ponto flutuante e o último se refere à interoperacionalidade com a linguagem C. Ambas as áreas não são abordadas nesta Apostila.

É possível que um programa completo faça uso de um módulo intrínseco e de um módulo definido pelo usuário, ambos com o mesmo nome, desde que ambos não sejam usados na mesma unidade de programa. Para usar um módulo intrínseco em preferência a um definido por usuário com o mesmo nome, a palavra-chave `INTRINSIC` deve ser incluída na instrução `USE`, como em

```
USE, INTRINSIC :: IEEE_ARITHMETIC
```

Da mesma forma, para assegurar que um módulo de usuário seja usado em preferência a um módulo intrínseco, a palavra-chave `NON_INTRINSIC` deve ser usada, como em

```
USE, NON_INTRINSIC :: RANDOM_NUMBER
```

Se ambos os módulos (intrínseco e definido por usuário) estão disponíveis com o mesmo nome, uma instrução `USE` sem nenhuma das palavras-chave acima irá acessar o módulo definido pelo usuário. Contudo, se o compilador não encontrar o módulo do usuário, então ele automaticamente irá acessar o módulo intrínseco sem emitir avisos.

Sugestões de uso & estilo para programação

Por razões de segurança, recomenda-se que um módulo definido por usuário tenha sempre um nome distinto de um módulo intrínseco, ou que seja sempre referido com a palavra-chave `NON_INTRINSIC`.

Recomenda-se também que um módulo intrínseco seja sempre usado com a cláusula `ONLY` (seção 9.13.1), uma vez que padrões futuros podem vir a adicionar novos recursos ao módulo.

O ambiente do Fortran

O módulo intrínseco `iso_fortran_env` fornece informações a respeito do ambiente da linguagem Fortran. O módulo contém dados globais e rotinas de módulos, os quais serão descritos a seguir.

Constantes nomeadas

As constantes nomeadas abaixo, todas escalares do tipo inteiro padrão, fornecem informações úteis para transferência de dados e acesso a arquivos externos (capítulo 10) e processamento de coarrays.

`character_storage_size` Tamanho em bits de uma unidade de armazenamento de caracteres.

`error_unit` O número de unidade para a unidade de saída adequada para reportar erros.

`file_storage_size` O tamanho em bits de uma unidade de armazanamento de arquivo (a unidade de medida para o comprimento do registro de um arquivo externo, conforme usado na cláusula `recl=` de comandos `open` ou `inquire`).

input_unit O número de unidade para a unidade padrão de entrada (a mesma empregada pela instrução `read` sem número de unidade ou com o especificador *).

iostat_end O valor retornado por `iostat=` para indicar uma condição de final de arquivo (*end-of-file*).

iostat_eor O valor retornado por `iostat=` para indicar uma condição de final de registro (*end-of-record*).

iostat_inquire_internal_unit O valor retornado por `iostat=` em um comando `INQUIRE` para indicar que um número de unidade de arquivo identifica uma unidade interna.

numeric_storage_size O tamanho em bits de uma unidade de armazenamento numérico.

stat_locked O valor retornado por `stat=` em um comando `LOCK` para uma variável bloqueada por uma imagem em execução (coarrays).

stat_locked_other_image O valor retornado por `stat=` em um comando `LOCK` para uma variável bloqueada por uma outra imagem (coarrays).

stat_unlocked O valor retornado por `stat=` em um comando `UNLOCK` para uma variável não bloqueada (coarrays).

stat_stopped_image O valor retornado por `stat=` para indicar uma imagem parada (coarrays).

output_unit O número da unidade para a unidade padrão de saída (a mesma empregada pelas instruções `PRINT*` ou `WRITE` com o especificador *).

De forma distinta ao que ocorre com números de unidades de arquivo normais, os números de unidade acima podem ser negativos, exceto que nunca serão iguais a `-1`.

INFORMAÇÕES SOBRE A COMPILAÇÃO

Existem duas funções inquiridoras disponíveis no módulo `iso_fortran_env`, as quais retornam informações a respeito do compilador.

compiler_version() Retorna uma string descrevendo o nome e a versão do compilador empregado.

compiler_options() Retorna uma string descrevendo as opções usadas durante a compilação.

Para ambas, a string retornada é um escalar padrão do tipo caractere.

O programa-exemplo abaixo:

```
! Mostra informações a respeito do compilador empregado.
! Usa rotinas do módulo intrínsico ISO_FORTRAN_ENV.
program tes_iso_fortran_env_comp
use, intrinsic :: iso_fortran_env, only: compiler_version, compiler_options
implicit none
print'(/,a)', '=============> Informações sobre a compilação <============='
print'(2a)', 'compiler_version: ', compiler_version()
print'(2a)', 'compiler options: ', compiler_options()
end program tes_iso_fortran_env_comp
```

Fornece as seguintes informações:

```
user@machine|dir>./a.out

=============> Informações sobre a compilação <=============
compiler_version: GCC version 9.2.1 20190827 (Red Hat 9.2.1-1)
compiler options: -mtune=generic -march=x86-64
```

Nomes para as espécies de tipo mais comuns

Constantes nomeadas para alguns valores de espécies dos tipos inteiro e real estão disponíveis no módulo `iso_fortran_env`, conforme já havia sido mencionado na seção 3.3.4.2. Os nomes são:

```
 int8  inteiro de 8 bits      real32  real de 32 bits
int16  inteiro de 16 bits     real64  real de 64 bits
int32  inteiro de 32 bits    real128  real de 128 bits
int64  inteiro de 64 bits
```

O uso destas constantes nomeadas já foi exemplificado no programa 3.4.

Se o compilador suportar mais do que uma espécie com um tamanho em particular, o padrão não especifica qual delas deve ser escolhida para a constante. Se o compilador não suportar um dos tamanhos listados acima, a constante terá valor igual a `-2` se existir uma espécie de tamanho maior ou `-1` se não suportar nenhum tamanho maior. O compilador também pode suportar mais espécies além das listadas na tabela acima.

O módulo `iso_fortran_env` também fornece as constantes nomeadas `atomic_int_kind` e `atomic_logical_kind`, as quais são escalares inteiros padrões para as espécies dos tipos inteiro e lógico usados como variáveis nas rotinas intrínsecas `atomic_define` e `atomic_ref` (coarrays).[5]

Matrizes de espécies

Em adição às constantes nomeadas acima, o módulo intrínseco `iso_fortran_env` também fornece matrizes constantes nomeadas contendo todas as espécies suportadas pelo compilador para todos os tipos intrínsecos da linguagem. As matrizes `character_kinds`, `integer_kinds`, `logical_kinds` e `real_kinds` contêm os valores das espécies dos tipos mencionados. Estas matrizes são do tipo inteiro padrão e possuem todas limite inferior igual a `1`. A ordem dos valores em cada matriz pode variar com o compilador.

O programa 3.4 também mostra os valores contidos nestas matrizes.

Lock type

O módulo intrínseco `iso_fortran_env` também contém o tipo derivado `LOCK_TYPE`, o qual não será descrito aqui por fazer parte do recurso dos coarrays.

9.13.7 Submódulos

Módulos, como foram apresentados até este ponto, são conceitos úteis para encapsular entidades (dados globais, rotinas, blocos de interfaces, *etc*) que compõe um determinado projeto ou executam uma determinada tarefa, mas quando esse projeto possui um tamanho modesto em termos de número total de entidades contidas no mesmo, suas interconexões e suas visibilidades às unidades que usam esse módulo.

Se o projeto for continuamente sendo aumentado, com a adição e/ou modificação de entidades contidas no mesmo, pode vir a ser necessário realizar uma nova modularização do projeto, com diferentes tipos de encapsulamento. Essa modularização adicional é fornecida pelo recurso dos *submódulos.* Com os mesmos, um determinado módulo pode ser dividido fisicamente em diversos submódulos (em arquivos distintos), os quais têm acesso às entidades contidas no módulo via associação ao hospedeiro.

Submódulos podem conter o corpo de definições de rotinas de módulos, por exemplo, enquanto que o módulo contém as interfaces dessas rotinas. Dessa forma, uma alteração em um submódulo em particular pode ser realizada sem alterar as interfaces no módulo e sem haver a necessidade de recompilação das unidades que usam o módulo, uma vez que um submódulo não pode ser usado, somente o módulo-pai.

As seções a seguir descrevem os recursos tornados disponíveis com submódulos e a sintaxe dos mesmos.

[5] Ver também seção 8.21.

9.13.7.1 ROTINAS DE MÓDULO SEPARADAS

A essencia deste recurso está na separação da definição de uma rotina de módulo em duas partes: a sua interface, a qual permanece no módulo, e o seu corpo, o qual será definido no submódulo. Este tipo de subprograma é denominado uma *rotina de módulo separada* (*separate module procedure*).

Um exemplo de uma rotina de módulo separada é fornecido abaixo. No módulo `POINTS`, a palavra-chave `MODULE` no prefixo da função `POINT_DIST` indica que a sua interface se refere a uma rotina de módulo. Já no submódulo `POINTS_A` a mesma palavra-chave indica que será realizada ali a implementação da mesma função cuja interface está no módulo. A palavra-chave `SUBMODULE` especifica, entre parênteses, o nome do módulo-pai. Tanto a interface quanto o submódulo ganham acesso ao tipo derivado `POINT` por associação ao hospedeiro.

```
MODULE POINTS
TYPE :: POINT
   REAL :: X, Y
END TYPE POINT
INTERFACE
   REAL MODULE FUNCTION POINT_DIST(A, B)
   TYPE(POINT), INTENT(IN) :: A, B
   END FUNCTION POINT_DIST
END INTERFACE
END MODULE POINTS

SUBMODULE (POINTS) POINTS_A
CONTAINS
   REAL MODULE FUNCTION POINT_DIST(A, B)
   TYPE(POINT), INTENT(IN) :: A, B
   POINT_DIST= SQRT((A%X - B%X)**2 + (A%Y - B%Y)**2)
   RETURN
   END FUNCTION POINT_DIST
END SUBMODULE POINTS_A
```

A interface especificada no submódulo deve ser a mesma especificada no bloco de interface.

Uma unidade que necessite invocar a função `POINT_DIST` deverá usar o módulo `POINTS`. O submódulo não pode ser usado por nenhuma unidade de programa. Gravando-se o módulo e o submódulo em arquivos separados, o compilador irá gerar dois arquivos auxiliares, o arquivo `*.mod` associado ao módulo e o arquivo `*.smod` contendo as informações no submódulo. Desta forma, caso seja necessário no futuro realizar alterações no corpo da função, somente o arquivo do submódulo deverá ser alterado e recompilado. O programa executável será então recriado com o linkador, o qual irá usar a nova versão do submódulo.

Uma vez que o submódulo tem acesso a todas as entidades do módulo-pai, não é necessário repetir as interfaces dos subprogramas cujos corpos são definidos no submódulo, ao invés disso, basta empregar o cabeçalho e o rodapé:

```
MODULE PROCEDURE <nome-subprograma>
...
END PROCEDURE <nome-subprograma>
```

e proceder-se com a redação do corpo do subprograma. Dessa maneira, uma forma alternativa e equivalente para o submódulo `POINTS_A` é:

```
SUBMODULE (POINTS) POINTS_A
CONTAINS
   MODULE PROCEDURE POINT_DIST
   POINT_DIST= SQRT((A%X - B%X)**2 + (A%Y - B%Y)**2)
   RETURN
   END PROCEDURE POINT_DIST
END SUBMODULE POINTS_A
```

9.13.7.2 Submódulos de submódulos

Um submódulo pode possuir submódulos, o que pode ser útil para projetos de grande complexidade. O módulo ou o submódulo do qual um dado submódulo é um subsidiário direto é denominado o seu **pai**, e este submódulo é o **filho** deste pai. Não existe limite no número de **ancestrais** e **descendentes**. Cada módulo ou submódulo é a raiz de uma árvore cujos nodos são seus descendentes e os quais têm acesso às entidades do mesmo por associação ao hospedeiro.

Outros submódulos, pertencentes a um ramo distinto da árvore, não têm acesso às entidades do primeiro ramo. Desta maneira, um projeto complexo pode ser desenvolvido na forma de subprojetos independentes. Como nenhum submódulo pode ser usado por qualquer outra unidade de programa, não é necessário estabelecer controle de acesso às suas entidades (seção 9.13.3); estas são, efetivamente, privadas.

Se uma alteração é realizada em um submódulo, somente seus descendentes necessitarão de recompilação. Para indicar que um submódulo possui um outro submódulo como seu ancestral imediato, emprega-se a notação `<nome-módulo>:<nome-submódulo>` no cabeçalho do mesmo. Por exemplo,

```
SUBMODULE (POINTS:POINTS_A) POINTS_B
```

declara que `POINTS_B` é filho do submódulo `POINTS_A`, o qual por sua vez é filho do módulo `POINTS`.

Enfatiza-se que esta sintaxe permite que dois submódulos tenham o mesmo nome, desde que ambos sejam descendentes de módulos distintos.

9.13.7.3 Entidades de submódulos

Um submódulo pode conter entidades próprias, as quais são inacessíveis a quaisquer outras unidades de programa e a quaisquer outros submódulos, exceto para seus descendentes diretos.

Um submódulo também pode conter rotinas, denominadas então **rotinas de submódulo**. Uma rotina de submódulo somente é acessível no submódulo e a seus descendentes.

Da mesma forma que uma rotina de módulo, uma rotina de submódulo também pode ser separada; um submódulo contém sua interface e um descendente contém o corpo da rotina.

9.13.7.4 Submódulos e associação por uso

Um submódulo tem acesso às entidades de seus ancestrais por associação ao hospedeiro. Um submódulo pode acessar entidades em outros módulos via associação por uso (seção 9.14.2). Inclusive, é possível ocorrer associação por uso circular: um submódulo do módulo `UM` acessa o módulo `DOIS` e um submódulo de `DOIS` acessa o módulo `UM`.

9.14 Âmbito *(Scope)*

Já foi mencionado neste texto, em diversos lugares, o *âmbito* ou o *escopo* de um certo nome de entidade na linguagem. O âmbito de um objeto nomeado ou de um rótulo é o conjunto de *unidades de âmbito*, que não se sobrepõe, onde o nome deste objeto pode ser usado sem ambiguidade.

Uma unidade de âmbito é qualquer um dos seguintes:

- uma definição de tipo derivado;
- o corpo de um bloco de interfaces de rotinas, excluindo quaisquer definições de tipo derivado e blocos de interfaces contidos dentro deste, ou
- uma unidade de programa ou subprograma, excluindo definições de tipo derivado, blocos de interfaces e rotinas internas contidas dentro desta unidade.

9.14.1 Âmbito dos rótulos

Toda unidade de programa ou subprograma, interno ou externo, tem seu conjunto independente de rótulos. Portanto, o mesmo rótulo pode ser usado em um programa principal e em seus subprogramas internos sem ambiguidade.

Assim, o âmbito de um rótulo é um programa principal ou subprograma, excluindo quaisquer subprogramas internos que eles contenham. O rótulo pode ser usado sem ambiguidade em qualquer ponto entre os comandos executáveis de seu âmbito.

9.14.2 ÂMBITO DOS NOMES

O âmbito de um nome declarado em uma unidade de programa estende-se do cabeçalho da unidade de programa ao seu comando `END`. O âmbito de um nome declarado em um programa principal ou rotina externa estende-se a todos os subprogramas que eles contêm, exceto quando o nome é redeclarado no subprograma.

O âmbito de um nome declarado em uma rotina interna é somente a própria rotina, e não os outros subprogramas internos. O âmbito do nome de um subprograma interno e do número e tipos dos seus argumentos estende-se por toda a unidade de programa que o contém, inclusive por todos os outros subprogramas internos.

O âmbito de um nome declarado em um módulo estende-se a todas as unidades de programa que usam este módulo, exceto no caso em que a entidade em questão tenha o atributo `PRIVATE`, ou é renomeada na unidade de programa que usa o módulo ou quando a instrução `USE` apresenta o qualificador `ONLY` e a entidade em questão não esteja na `<lista-only>`. O âmbito de um nome declarado em um módulo estende-se a todos os subprogramas internos, excluindo aqueles onde o nome é redeclarado.

Considerando a definição de unidade de âmbito acima,

- Entidades declaradas em diferentes unidades de âmbito são sempre distintas, mesmo que elas tenham o mesmo nome e propriedades.
- Dentro de uma unidade de âmbito, cada entidade nomeada deve possuir um nome distinto, com a exceção de nomes genéricos de rotinas.
- Os nomes de unidades de programas são globais; assim, cada nome deve ser distinto dos outros e distinto de quaisquer entidades locais na unidade de programa.
- O âmbito do nome de uma rotina interna estende-se somente por toda a unidade de programa que a contém.
- O âmbito de um nome declarado em uma rotina interna é esta rotina interna.

Este conjunto de regras resume a definição de âmbito de um nome.

Nomes de entidades são acessíveis por *associação ao hospedeiro* ou *associação por uso* quando:

Associação ao hospedeiro. O âmbito de um nome declarado em uma unidade de programa estende-se do cabeçalho da unidade de programa ao comando `END`.

Associação por uso. O âmbito de um nome declarado em um módulo, o qual não possui o atributo `PRIVATE`, estende-se a qualquer unidade de programa que usa o módulo.

Nota-se que ambos os tipos de associação não se estende a quaisquer rotinas externas que possam ser invocadas e não incluem quaisquer rotinas internas onde o nome é redeclarado.

Um exemplo contendo 5 unidades de âmbito é ilustrado a seguir:

```
MODULE AMBITO1               ! Ambito 1
...                          ! Ambito 1
CONTAINS                     ! Ambito 1
   SUBROUTINE AMBITO2        ! Ambito 2
      TYPE :: AMBITO3        ! Ambito 3
         ...                 ! Ambito 3
      END TYPE AMBITO3       ! Ambito 3
      INTERFACE              ! Ambito 2
        ...                  ! Ambito 4
      END INTERFACE          ! Ambito 2
      ...                    ! Ambito 2
   CONTAINS                  ! Ambito 2
```

```
          FUNCTION AMBITO5(...)  ! Ambito 5
          ...                    ! Ambito 5
          END FUNCTION AMBITO5   ! Ambito 5
       END SUBROUTINE AMBITO2    ! Ambito 2
    END MODULO AMBITO1           ! Ambito 1
```

9.14.3 Controle de acesso ao hospedeiro

Uma instrução `IMPORT` pode ser usada para estabelecer um controle de acesso ao hospedeiro para um subprograma interno, um submódulo (seção 9.13.7) ou um construto `BLOCK` (seção 9.15). Esta instrução já foi descrita na seção 9.5.1 para fornecer acesso no interior de um bloco de interfaces às entidades acessíveis na unidade hospedeira.

Agora, a instrução assume uma das três formas a seguir:

```
IMPORT, ONLY: <lista-nomes-import>
IMPORT, NONE
IMPORT, ALL
```

Estas formas têm as seguintes ações:

`IMPORT, ONLY` Somente as entidades na `<lista-nomes-import>` serão acessíveis por associação ao hospedeiro. Cada nome nesta lista deve ser o nome de uma entidade do hospedeiro e não deve aparecer em um contexto que torna uma dessas entidades inacessível, tal como ter o mesmo nome declarado como uma entidade local.

`IMPORT, NONE` Nenhuma entidade será acessível por associação ao hospedeiro e esta deve ser a única instrução `IMPORT`. Esta forma não é permitida no campo de declarações de um submódulo, exceto se estiver dentro de um bloco de interfaces.

`IMPORT, ALL` Todas as entidades do hospedeiro são acessíveis por associação ao hospedeiro e esta deve ser a única instrução `IMPORT`. Nenhuma entidade do hospedeiro deve aparecer em um contexto que a torna inacessível, tal como ter o mesmo nome declarado como uma entidade local.

O programa `rot_int_import_only` abaixo é uma nova versão do programa 9.1, agora com a função interna `calc_a_raiz` fazendo uso da instrução `IMPORT, ONLY: A` para importar somente uma das variáveis declaradas no programa principal.

```
program rot_int_import_only
implicit none
real :: x,a
print*, "Entre com o valor de x:"
read*, x
print*, "Entre com o valor de a:"
read*, a
print*, "O resultado de a + sqrt(x) é:"
print*, calc_a_raiz(x)
print*, "O resultado de a + sqrt(1+ x**2) é:"
print*, calc_a_raiz(1.0 + x**2)
CONTAINS
   function calc_a_raiz(y)
   import, only: a
   real              :: calc_a_raiz
   real, intent(in) :: y
   calc_a_raiz= a + sqrt(y)
   return
   end function calc_a_raiz
end program rot_int_import_only
```

9.15 CONSTRUTOS BLOCK

[EM CONSTRUÇÃO]

10

COMANDOS DE ENTRADA/SAÍDA DE DADOS

O Fortran possui um conjunto rico de instruções de entrada/saída (E/S) de dados. Entretanto, este capítulo irá apresentar apenas um conjunto de instruções que implementam um processo básico de E/S.

Muitos programas necessitam de dados iniciais para o seu processamento. Depois que os cálculos estiverem completos, os resultados naturalmente precisam ser impressos, mostrados graficamente ou salvos para uso posterior. Durante a execução de um programa, algumas vezes há uma quantidade grande de dados produzidos por uma parte do programa, os quais não podem ser todos armazenados na memória de acesso aleatório (RAM) do computador. Nesta situação também se usam os comandos de E/S da linguagem.

O Fortran possui muitos comandos de E/S. Os mais utilizados são:

```
OPEN    READ    WRITE
CLOSE   PRINT
```

Estes comandos, e outros que serão abordados neste capítulo, estão todos embutidos dentro da linguagem a qual possui, adicionalmente, recursos de formatação que instruem a maneira como os dados são lidos ou escritos.

Até este ponto, todos os exemplos abordados executaram operações de E/S de dados com teclado (entrada) e terminal (saída). Adicionalmente, o Fortran permite que diversos outros objetos, como arquivos de dados, estejam conectados a um programa para leitura e/ou escrita. Neste capítulo, serão abordados os processos de E/S em arquivos de dados, por ser este o uso mais frequente deste recurso. Outros tipos de usos incluem saída direta de dados em impressoras, *plotters*, programas gráficos, recursos de multimeios, *etc.*

Devido à variedade de usos dos comandos de E/S, é interessante que se faça inicialmente uma introdução simples ao assunto, abordando os usos mais frequentes dos processos com arquivos externos, para posteriormente entrar-se em maiores detalhes. Com este intuito, a seção 10.1 contém esta introdução rápida, ao passo que as seções posteriores (10.2 – 10.13) abordam o assunto de uma forma bem mais abrangente.

10.1 COMANDOS DE ENTRADA/SAÍDA: INTRODUÇÃO RÁPIDA

Na seção 2.6 já foram apresentadas a entrada (teclado) e a saída (tela do monitor) padrões do Fortran. Os processos de entrada/saída (E/S) também foram realizados na forma **lista-dirigida** (*list-directed*), na qual o formato é determinado pelo computador no momento em que o comando de E/S é executado e que depende tanto do tipo quanto da magnitude das quantidades envolvidas. Este tipo de processo também é denominado **formato livre** (*free format*) e é executado pelos comandos e na sintaxe já mencionados na seção 2.6, isto é, por

```
READ*,      READ(*,*)
PRINT*,     WRITE(*,*)
```

sendo que, no caso dos comandos `READ(*,*)` e `WRITE(*,*)`, o primeiro asterisco "*" se refere à entrada/saída padrão e o segundo ao formato livre.

Este tipo de E/S de dados é útil quando há um número pequeno de quantidades envolvidas ou para testes do código. Mas, usados nestas formas, os recursos são demasiado limitados. O programador possui controle muito restrito, por exemplo, sobre onde, na tela, o resultado deve ser apresentado e sobre o número de casas decimais que deve ser utilizado para representar o valor. Adicionalmente, o programador está limitado ao teclado para o fornecimento de parâmetros ao programa. Em diversos outros exemplos subsequentes, distribuídos ao longo dos capítulos anteriores desta Apostila, mostrou-se exemplos de operações de E/S formatadas, mas ainda nos dispositivos padrões.

Uma **especificação de formato** (*format specification*) consiste em uma string de caracteres contendo um certo número de **descritores de edição** (*edit descriptors*), os quais indicam como um determinado objeto de dados deve ser formatado a partir de sua representação interna (binária) no computador para uma variável de caractere contendo uma outra representação deste objeto. Dessa forma, a quantidade que na base numérica decimal é representada por -0.000450 pode ser formatada e impressa, dependendo do descritor de edição empregado, como `-0.00045` ou `-4.5E-04` ou ainda `-5E-04`. Os descritores de edição mais comuns serão mostrados nas tarefas abaixo.

Um uso mais desejável dos recursos de E/S do Fortran consiste na habilidade de ler e gravar dados, por exemplo, em arquivos residentes no disco rígido do computador ou em outra mídia à qual o mesmo tem acesso. Nesta seção, o acesso de E/S a arquivos será também brevemente discutido na forma de tarefas atribuídas ao programador.

TAREFA 1. SAÍDA FORMATADA NA TELA DO MONITOR

Considera-se inicialmente o seguinte programa que realiza saída no formato livre na tela:[1]

```
program F_out
use, intrinsic :: iso_fortran_env, only: dp => real64
implicit none
real(dp) :: a, b, c
a= 1.0_dp/7.0_dp
b= sqrt(2.0_dp)
c= 4*atan(1.0_dp)
print*, a, b, c
end program F_out
```

Gerando o executável com o compilador GFortran, os resultados deste programa são apresentados da seguinte maneira na tela do monitor:[2]

```
␣␣0.142857142857143␣␣␣␣␣␣␣␣1.41421356237310␣␣␣␣␣␣␣␣3.14159265358979
```

Observa-se que há um número grande de espaços em branco antes e entre os valores dos resultados e todos são apresentados com o número total de dígitos representáveis para uma variável real de dupla precisão.

O programa a seguir apresenta os mesmos resultados, porém de maneiras mais organizadas:

```
program F_out_formatado
use, intrinsic :: iso_fortran_env, only: dp => real64
implicit none
real(dp) :: a, b, c
a= 1.0_dp/7.0_dp
b= sqrt(2.0_dp)
c= 4*atan(1.0_dp)
print"(f7.5,1x,f9.7,1x,f11.9)", a, b, c
print"(f7.7,1x,f9.9,1x,e12.5)", a, b, c
print"('a= ',f7.5,2x,'b= ',f9.7,3x,'c= ',f11.9)", a, b, c
print'(3(a, g0, 2x))', 'a= ', a, 'b= ', b, 'c= ', c
end program F_out_formatado
```

[1] A instrução `use, intrinsic :: iso_fortran_env` acessa o módulo intrínseco `iso_fortran_env`, descrito na seção 9.13.6.

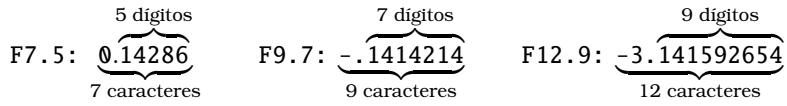
[2] Lembrando que o simbolo "␣" representa um espaço em branco.

Agora, os resultados são:

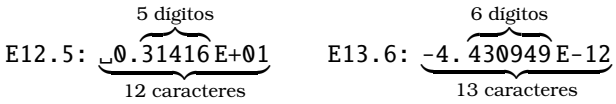
```
0.14286␣1.4142136␣3.141592654
*******␣*********␣␣0.31416E+01
a=␣0.14286␣␣b=␣1.4142136␣␣␣c=␣3.141592654
a=␣0.14285714285714285␣␣b=␣1.4142135623730951␣␣c=␣3.1415926535897931
```

Nos quatro resultados, os argumentos dos comandos `PRINT` são as especificações de formatação. A formatação dos dados de saída é determinada pela sequência de descritores de edição empregados (`X`, `F`, `E`, `G` e `A`), os quais têm os seguintes significados:

- Os descritores `1X`, `2X` e `3X` indicam quantos espaços em branco (respectivamente 1, 2 e 3) devem ser deixados na saída dos dados.
- Os descritores `F`, `E` e `G` são descritores de edição de dados e eles se aplicam ao dados na lista de variáveis (`A`, `B` e `C`) na mesma ordem em que aparecem. Ou seja, na saída 1, o descritor `F7.5` determina a edição da variável `A`, `F9.7` determina a adição de `B` e `F11.9` determina a edição de `C`.
- O descritor `F7.5` indica que a variável deve ser impressa no formato de ponto flutuante, com um total de 7 algarismos alfanuméricos, contando o ponto decimal e o sinal (se houver), destinando 5 dígitos para a parte fracionária da variável `A`. Exemplos de números de ponto flutuante válidos são:



- Na saída 2, o descritor `E12.5` determina o formato de saída da variável `C`. O descritor indica a saída na forma de ponto flutuante com parte exponencial. O resultado impresso (`␣0.31416E+01`) possui uma extensão total de 12 algarismos alfanuméricos, contando o sinal ("`-`" se for negativo, "`␣`" se for positivo), o ponto decimal, o símbolo da parte exponencial (`E`), o sinal da parte exponencial (`+`) e o seu valor, com duas casas decimais (`01`). Restam, então, somente 5 dígitos para a parte fracionária. Caso a extensão total do descritor, descontado o número de dígitos na parte fracionária, não for suficiente para imprimir a parte exponencial e mais o sinal, ocorre um *estouro de campo*, indicado pelos asteriscos nos resultados das variáveis `A` e `B`. Exemplos válidos de números no formato exponencial são:



- Na saída 3, as constantes de caractere `'A= '`, `'B= '` e `'C= '` que aparecem na formatação são impressas *ipsis literis* na saída padrão e na mesma ordem em que aparecem, relativamente a si próprias e às variáveis.
- Finalmente, na saída 4, a especificação global `'(3(A, G0, 2X))'` indica que deve haver `3` repetições da especificação `'(A, G0, 2X)'`, uma para cada par string + variável (como em `'a= ', a`). O descritor `A` indica a saída de uma string, a qual será impressa no seu comprimento total, enquanto que `G0` é um descritor de edição geral de dados numéricos onde o formato pode ser de ponto flutuante ou exponencial (dependendo da magnitude do dado) e a precisão do resultado é determinada pelo compilador. Em geral, neste formato o dado é convertido em uma representação decimal com a precisão máxima relacionada ao tipo e espécie da variável cujo valor está sendo impresso. Este tipo de formatação é bastante flexível e foi empregado em diversos exemplos na Apostila.

Uma exposição completa de todos os descritores é realizada na seção 10.10.

TAREFA 2. ENTRADA FORMATADA A PARTIR DO TECLADO

Entrada formatada a partir do teclado não é uma tarefa muito prática, pois uma instrução do tipo

```
READ(*,FMT='(1X,F7.5,5X,F9.6,3X,F11.8)')A, B, C
```

iria requerer a digitação dos valores das variáveis exatamente nas posições assinaladas e com as extensões e partes fracionárias exatamente como determinadas pelos descritores. Por exemplo, se `A= 2.6`, `B= -3.1` e `C= 10.8`, seria necessário digitar

```
␣␣2.60000␣␣␣␣␣-3.100000␣␣␣10.80000000
```

para preencher todos os dígitos e espaços determinados pelos descritores de formatação.

TAREFA 3. SAÍDA DE DADOS EM UM ARQUIVO

Da mesma maneira que um programa-fonte é gravado em um arquivo situado no disco rígido do computador, é possível gravar dados em arquivos a partir de um programa ou, de forma alternativa, ler dados contidos em arquivos situados no disco.

Para possibilitar acesso de E/S em um arquivo é necessário que o programador:

- Identifique o nome do arquivo, fornecendo o caminho completo, caso o arquivo resida em um diretório (pasta) distinto do programa executável.
- Informe ao sistema sobre o tipo de acesso e uso que será feito do arquivo.
- Associe instruções individuais de leitura ou gravação com o arquivo em uso. É possível existir mais de um arquivo simultaneamente acessível para E/S, além das interfaces já utilizadas (teclado e monitor).
- Quando as operações de E/S estiverem concluídas, é necessário instruir ao sistema que o acesso ao(s) arquivo(s) foi concluído.

O programa exemplo a seguir grava informações formatadas no arquivo `EX1.DAT`. A extensão `.DAT` simplesmente identifica o arquivo como sendo de dados. Se o arquivo não existie, este é criado e tornado acessível; se ele existe, então os dados previamente contidos nele serão substituídos por novos valores.

```
program Arq_Sai
use, intrinsic :: iso_fortran_env, only: dp => real64
implicit none
real(dp) :: a, b, c
a= 1.0_dp/7.0_dp
b= sqrt(2.0_dp)
c= 4*atan(1.0_dp)
open(10, file="ex1.dat")
write(10, fmt="(f7.5,1x,f9.7,1x,f11.9)") a, b, c
close(10)
end program Arq_Sai
```

Alterando os valores das variáveis `A`, `B` e `C`, altera-se os valores gravados no arquivo `EX1.DAT`.

As instruções relevantes neste exemplo são:

`OPEN(10, FILE="EX1.DAT").` O comando `OPEN` habilita acesso ao arquivo `EX1.DAT` para o programa executável. O arquivo está situado no mesmo diretório que o programa e este será criado caso não exista previamente, sendo possível então a gravação no mesmo, ou, caso ele já exista, terá acesso de leitura e/ou escrita e dados previamente existentes serão sobrescritos.

O número `10` é o **número da unidade lógica** que será acessada. Doravante, o arquivo `EX1.DAT` será sempre identificado pelo programa através deste número.

O modo como o acesso ao arquivo é feito pode ser alterado de diversas maneiras. A seção 10.6 descreve todos os especificadores possíveis do comando `OPEN`.

`WRITE(10, FMT="(F7.5,1X,F9.7,1X,F11.9)") A, B, C.` Este comando é aqui utilizado no lugar de `PRINT` para realizar um processo de escrita formatada no arquivo associado à unidade lógica `10`. A formatação é fornecida pela lista de descritores de formatação que seguem a palavra-chave `FMT=`. O papel destes descritores já foi esclarecido.

O comando essencialmente grava os valores das variáveis A, B e C de maneira formatada no arquivo associado à unidade 10.

CLOSE(10). Finalmente, este comando informa ao computador que não será mais necessário acessar o arquivo associado à unidade lógica 10 e que tal associação deve ser interrompida. Todas os dados remanescentes serão gravados no arquivo EX1.DAT e este será desconectado da unidade lógica 10.

TAREFA 4. LEITURA/ESCRITA DE DADOS EM ARQUIVOS. VERSÃO 1: NÚMERO CONHECIDO DE LINHAS

O programa a seguir lê os dados gravados no arquivo EX1.DAT, criado a partir do programa ARQ_SAI, e usa estes dados para definir valores de outros, os quais serão gravados no arquivo EX2.DAT.

```
program Arq_Sai_2
use, intrinsic :: iso_fortran_env, only: dp => real64
implicit none
real(dp) :: a, b, c ! Dados de entrada.
real(dp) :: d, e, f ! Dados de saida.
open(10, file="ex1.dat", status="old")
open(11, file="ex2.dat", status="new")
read(10, fmt="(f7.5,1x,f9.7,1x,f11.9)") a, b, c
print*, "Os valores lidos foram:"
print*, "a= ",a
print*, "b= ",b
print*, "c= ",c
d= a + b + c
e= b**2 + cos(c)
f= sqrt(a**2 + b**3 + c**5)
write(11, fmt="(3(e11.5,1x))")d, e, f
end program Arq_Sai_2
```

No programa ARQ_SAI_2 aparecem as seguintes instruções que ainda não foram discutidas:

STATUS="OLD"/"NEW". Estas cláusulas, que aparecem nos comandos OPEN, indicam o status exigido para o arquivo a ser acessado. STATUS="OLD" indica que o arquivo deve existir no momento de acesso. Caso isto não aconteça, ocorre uma mensagem de erro e o programa é interrompido. STATUS="NEW" indica que o arquivo não deve existir previamente e, então, deve ser criado. Novamente, se estas condições não se cumprirem, o programa é interrompido.

READ(10, FMT="(F7.5,1X,F9.7,1X,F11.9)") A, B, C. Observa-se aqui o uso de uma entrada formatada com o comando READ. A formatação das variáveis A, B e C já foi discutida. Deve-se enfatizar novamente que a formatação especificada pelo FMT= deve possibilitar a leitura correta dos dados no arquivo associado à unidade lógica 10.

WRITE(11, FMT="(3(E11.5,1X))") D, E, F. O aspecto ainda não abordado neste comando WRITE é o uso do *contador de repetição* 3, o qual indica que as variáveis D, E e F devem ser gravadas no arquivo associado à unidade 11 com a formatação dada por "(E11.5,1X)" repetida 3 vezes consecutivas.

Finalmente, deve-se mencionar que os comandos CLOSE(10) e CLOSE(11) não foram utilizados. Isto é permitido, uma vez que, neste caso, a associação das unidades lógicas aos arquivos será interrompida com o final da execução do programa. Contudo, em certas situações é necessário associar uma unidade lógica previamente empregada a um outro arquivo, o que neste caso obriga o programador a fazer uso do comando CLOSE.

Modificações posteriores no programa necessitarão também a eliminação do arquivo EX2.DAT, devido à cláusula STATUS="NEW".

TAREFA 5. LEITURA/ESCRITA DE DADOS EM ARQUIVOS. VERSÃO 2: NÚMERO DESCONHECIDO DE LINHAS

Quando o número de linhas no arquivo de entrada não é previamente conhecido, é necessário um procedimento diferente daquele adotado na Tarefa 4. O programa 10.1 ilustra dois possíveis métodos a ser adotados, os quais utilizam as opções `IOSTAT` ou `END` do comando `READ` (seção 10.7). Em ambos, aloca-se inicialmente uma matriz temporária com o posto adequado e com um número tal de elementos que seja sempre maior ou igual ao número máximo de linhas que podem estar contidas no arquivo de entrada. Após a leitura dos dados na matriz temporária, o número de linhas no arquivo tornou-se conhecido e aloca-se, então, uma matriz definitiva com o número correto de elementos. O segundo exemplo faz uso de um rótulo,[3] que embora não seja um procedimento recomendado em Fortran, é completamente equivalente ao primeiro exemplo.

Listagem 10.1: Programa que ilustra leitura de arquivo com número desconhecido de linhas.

```
! ***************************** PROGRAMA ARQ_SAI_3 ******************************
! Lê um arquivo de dados com um número conhecido de elementos por linha,
!    mas com um número desconhecido de linhas no formato livre, usando
!    matrizes temporárias.
! A leitura do arquivo é realizada de 2 maneiras distintas:
!    1. Usando a opção IOSTAT.
!    2. Usando a opção END.
! O programa então imprime na tela os dados e o número de linhas lido.
!
! Autor: Rudi Gaelzer
! Data:  Maio/2008
!
program Arq_Sai_3
use, intrinsic :: iso_fortran_env, only: dp => real64
implicit none
integer :: controle, i, npt1= 0, npt2= 0
character(len= 1) :: char
real(kind=dp), dimension(:,:), allocatable :: temp1, temp2, matriz1, matriz2
allocate(temp1(2,1000), temp2(2,1000)) ! Aloca matrizes temporárias.
                                       ! Estima um limite superior no número
                                       ! de linhas.
! Metodo 1: usando opcao IOSTAT.
open(10, file='ex3.dat', status='old')
do
   read(10, *, iostat= controle)temp1(1,npt1+1), temp1(2,npt1+1)
   if(controle < 0) exit               ! Final de arquivo detectado.
   npt1= npt1 + 1
end do
close(10)
allocate(matriz1(2,npt1))
matriz1= temp1(:,:npt1)
write(*, fmt="('Matriz 1 lida:',/)")
print'(2(e10.3,1x))', (matriz1(:,i), i= 1, npt1)
deallocate(temp1)                      ! Libera espaço de memoria ocupado por
                                       ! temp1.
print'(/,a)', 'Pressione ENTER/RETURN para continuar.'
read(*,'(a)')char
! Metodo 2: usando opcao END.
open(10, file='ex3.dat', status='old')
do
   read(10, *, end= 100)temp2(1,npt2+1), temp2(2,npt2+1)
   npt2= npt2 + 1
end do
100 continue                           ! Linha identificada com o rótulo 100.
```

[3] A descrição de um rótulo é realizada em Apostila de Fortran 90/95 (seção 5.1.1).

```
close(10)
allocate(matriz2(2,npt2))
matriz2= temp2(:,:npt2)
print'("Matriz 2 lida:",/)'
print'(2(e10.3,1x))', (matriz2(:,i), i= 1, npt1)
deallocate(temp2)                    ! Libera espaço de memoria ocupado por
                                     ! temp2.
print'(/,"O numero total de linhas lidas é':",/, &
      "Matriz 1: ", i4, 3x, "Matriz 2: ",i4)', npt1, npt2
end program Arq_Sai_3
```

TAREFA 6. LEITURA/ESCRITA DE DADOS EM ARQUIVOS. VERSÃO 3: NÚMERO DESCONHECIDO DE LINHAS USANDO LISTAS ENCADEADAS

Existem maneiras mais inteligentes de realizar esta tarefa, sem que seja necessário definir-se uma matriz temporária. Nesta tarefa isto será realizado com o uso de de uma lista encadeada (seção 7.4.1).

Uma lista encadeada é uma série de variáveis de um tipo derivado, o qual é definido com um componente de ponteiro (seção 7.2), do mesmo tipo derivado, o qual irá apontar para o elemento seguinte da lista. Cada elemento desta lista, denominado *nodo*, contém um conjunto de dados de diferentes tipos que devem ser armazenados e pelo menos um ponteiro que irá localizar o nodo seguinte. A estrutura básica do tipo derivado que pode ser usado para se criar uma lista encadeada é a seguinte:

```
TYPE :: LINK
   REAL :: DADO
   TYPE(LINK), POINTER :: NEXT => NULL()
END TYPE LINK
```

Neste exemplo, o tipo derivado `LINK` é definido. Seus componentes são uma variável real (`DADO`), a qual irá armazenar o valor lido e um ponteiro do tipo `LINK`, denominado `NEXT`. Este ponteiro está inicialmente com o status de desassociado, através do uso da função `NULL()` (seção 8.16), e a sua presença serve para localizar o próximo elemento da lista, funcionando neste caso como os elos de uma cadeia.

Usualmente, são declarados dois ponteiros do tipo derivado `LINK`, um denominado `RAIZ`, o qual irá servir como localização para o início da lista (ou o primeiro nodo) e o outro denominado `CORRENTE`, que irá servir para identificar o nodo presentemente em uso. Portanto, uma declaração semelhante a

```
TYPE(LINK), POINTER :: RAIZ, CORRENTE
```

se faz necessária. O programa 10.2 ilustra a aplicação de uma lista encadeada para este fim.

Listagem 10.2: Programa que ilustra uso de uma lista encadeada para a leitura de um número arbitrário de dados.

```
1  !*************************** PROGRAMA LINKED_LIST_IO ***********************
2  ! Lê um um número arbitrário de dados e os armazena em um vetor,
3  !  fazendo uso de uma lista encadeada.
4  ! Para interromper a entrada de dados, basta fornecer
5  !  um caractere não numérico.
6  ! Autor: Rudi Gaelzer - IF/UFRGS
7  ! Data:  Março/2011.
8  program linked_list_io
9  implicit none
10 type :: link
11    real :: dado
12    type(link), pointer :: next => null()
```

```
13 end type link
14 type(link), pointer :: raiz => null(), corrente => null()
15 integer :: conta= 0, i, fim
16 real :: temp
17 real, dimension(:), allocatable :: vetor
18 print*, 'Entre com os dados (digite caractere não numérico para encerrar):'
19 do !Lê dados da entrada padrão até final e conta.
20    write(*, fmt='("Valor= ")', advance='no')
21    read(*, fmt=*, iostat= fim) temp
22    if(fim /= 0)exit                  ! Dados encerrados.  Sai do laço
23    if(.not. associated(raiz))then ! Inicia a lista pela raiz
24       allocate(raiz)                 ! Aloca espaço para raiz
25       raiz%dado= temp                ! Grava dado lido na lista
26       corrente => raiz               ! Corrente aponta para raiz
27    else                           ! Inicia novo nodo
28       allocate(corrente%next)    ! Aloca espaço para o próximo nodo
29       corrente => corrente%next ! Corrente aponta para o novo nodo
30       corrente%dado= temp        ! Acumula valor no nodo
31    end if
32    conta= conta + 1
33 end do
34 print*, 'Número de dados lidos:', conta
35 allocate(vetor(conta))             !Aloca vetor.
36 corrente => raiz                   !Retorna ao início da lista.
37 do i= 1, conta !Atribui valores lidos ao vetor, na ordem de leitura.
38    vetor(i)= corrente%dado
39    corrente => corrente%next
40 end do
41 write(*, fmt='(/,"Vetor lido:")')
42 do i= 1, conta                      !Imprime vetor na tela.
43    print*, vetor(i)
44 end do
45 ! Grava lista lida em arquivo
46 open(10, file= 'linked_list_io.dat')
47 write(10, fmt= '(g0)') (vetor(i), i= 1, conta)
48 end program linked_list_io
```

Uma descrição mais longa para a definição e uso da lista encadeada pode ser vista na seção 7.4.1. Na linha 21, foi empregada a especificação `ADVANCE= 'NO'`, a qual executa uma **escrita sem avanço** (*non-advancing write*), isto é, o cursor se mantém na mesma linha após escrever `Valor=` na tela. Deve-se notar também que no programa 10.2 não foi tomada nenhuma providência para liberar o espaço de memória ocupado pela lista encadeada. Os perigos inerentes desta prática e como evitá-los estão descritos nos comentários adjacentes ao programa 7.8.

Usar uma lista encadeada para este tipo de tarefa é muito mais eficiente que o emprego de uma matriz temporária, como foi realizado na Tarefa 5, pois não é mais necessário cogitar-se o número máximo de valores a ser lidos. O número de nodos da lista encadeada pode crescer de forma arbitrária, sendo a quantitade total de memória o único limite existente. Desta forma, a lista encadeada é um objeto que pode ser empregado em um número arbitrário de situações distintas. De fato, listas simplesmente ou multiplamente encadeadas são instrumentos empregados com frequência em programação orientada a objeto.

A mesma tarefa poderia ter sido implementada em uma lista alocável, ao invés de uma lista encadeada. A seção 7.4.1.2 fornece uma descrição de listas alocáveis.

Os exemplos abordados nas Tarefas 1 — 6 constituem um conjunto pequeno, porém frequente, de usos dos comando de E/S. A descrição completa destes recursos pode ser obtida nas seções 10.4—10.13.

10.2 DECLARAÇÃO NAMELIST

Em certas situações, quando há por exemplo um número grande de parâmetros sendo transferidos em processos de E/S, pode ser útil definir-se uma lista rotulada de valores destinados à transferência. O Fortran introduziu, com este fim, a declaração NAMELIST, que define um ou mais *grupos* de variáveis referenciadas com o mesmo nome. Esta declaração é empregada em conjunto com operações de E/S executadas por comandos READ e WRITE.

A forma geral da declaração, que pode aparecer em qualquer unidade de programa, é:

```
NAMELIST /<nome-grupo-namelist>/ <lista-nomes-variáveis> &
         [[,] /<nome-grupo-namelist>/ <lista-nomes-variáveis>[,] ...]
```

Como se pode notar, é possível definir-se mais de um grupo NAMELIST na mesma declaração. O primeiro campo substituível: *<nome-grupo-namelist>*, é o nome do grupo para uso subsequente nos comandos de E/S. A *<lista-nomes-variáveis>*, como o nome indica, lista os nomes das variáveis que compõe o grupo. Não podem constar desta lista nomes de matrizes mudas de forma assumida, ou objetos automáticos, variáveis de caractere de extensão variável, matrizes alocáveis, pointers, ou ser um componente de qualquer profundidade de uma estrutura que seja um ponteiro ou que seja inacessível. Um exemplo válido de *<lista-nomes-variáveis>* é:

```
REAL :: TV, CARPETE
REAL, DIMENSION(10) :: CADEIRAS
...
NAMELIST /ITENS_DOMESTICOS/ CARPETE, TV, CADEIRAS
```

Outro exemplo válido:

```
NAMELIST /LISTA1/ A, B, C /LISTA2/ X, Y, Z
```

É possível também continuar a mesma *<lista-nomes-variáveis>* de um dado grupo em mais de uma declaração contidas no campo de declarações da unidade de programa. Assim,

```
NAMELIST /LISTA/ A, B, C
NAMELIST /LISTA/ D, E, F
```

é equivalente a uma única declaração NAMELIST com os 6 nomes de variáveis listados na mesma ordem acima. Além disso, um objeto de um grupo NAMELIST pode também pertencer a outros grupos.

Se o tipo, parâmetro de tipo ou forma de uma variável de um grupo é especificado em uma declaração na mesma unidade de programa , esta declaração deve constar antes da declaração NAMELIST, ou ser uma declaração de tipo implícito que regule o tipo e espécie das variáveis que iniciam com aquele caractere explicitado na declaração.

Um grupo namelist pode possuir o atributo público ou privado, concedido através das declarações PUBLIC ou PRIVATE. Contudo, se o grupo possuir o atributo público, nenhum membro seu pode ser declarado com o atributo privado ou possuir componentes privados.

Para se executar uma operação de E/S em algum registro (como um arquivo, por exemplo) que contenha um ou mais grupos namelist, o registro sempre irá iniciar com o caractere &, seguido, sem espaço, pelo *<nome-grupo-namelist>*, vindo então a lista dos nomes das variáveis, sendo cada nome seguido por sinais de igual e, então pelo valor da variável, podendo ser precedido ou seguido por espaços em branco. Constantes de caractere devem ser sempre delimitadas em registros de entrada. A lista de nomes e valores de um determinado grupo é sempre encerrada com o caractere / inserido fora de uma constante de caractere.

Os comandos READ e WRITE, ao serem empregados em conjunto com um namelist, não possuem uma lista explícita de variáveis ou constantes a ser transferidas, mas sim o nome do grupo NAMELIST como segundo parâmetro posicional ou com o especificador NML= contendo o nome do grupo, no lugar do especificador FMT=. Um exemplo de registro de entrada é:

```
INTEGER :: NO_DE_OVOS, LITROS_DE_LEITE, QUILOS_DE_ARROZ
NAMELIST /FOME/ NO_DE_OVOS, LITROS_DE_LEITE, QUILOS_DE_ARROZ
READ(5, NML=FOME)
```

O associado exemplo de registro (arquivo) de entrada é:

```
&FOME LITROS_DE_LEITE= 5, NO_DE_OVOS= 12 /
```

Nota-se no registro acima que a ordem de atribuição de valores aos nomes não é necessariamente a mesma da declaração NAMELIST. Além disso, a variável QUILOS_DE_ARROZ não tem seu valor atribuído no registro. Quando isso acontece com uma ou mais variáveis da lista o valor destas permanece inalterado.

Os valores de uma matriz podem ser atribuídos através do nome da mesma seguida por = e a lista de constantes que serão atribuídos aos elementos da matriz de acordo com o ordenamento padrão da linguagem, descrito na seção 6.6.2. Se a variável for de tipo derivado, a lista de valores deve também seguir a ordem e o tipo e espécie dos componentes constantes da declaração do tipo. É possível também definir-se o valor de um elemento isolado ou de uma seção de matriz, ou qualquer sub-objeto, sem alterar os valores dos demais elementos, usando um designador de sub-objeto, tal como um tripleto de subscritos. Neste caso, os elementos do tripleto devem todos ser constantes inteiras escalares, sem parâmetros de espécie de tipo

Na gravação (saída) do registro, todos os componentes do grupo são escritos no arquivo especificado, precedidos por & e o nome do grupo, na mesma ordem da declaração da *<lista-nomes-variáveis>*, com os nomes explicitamente escritos em letras maiúsculas. O registro é finalizado com o caractere "/". Um exemplo de saída de um namelist é:

```
INTEGER :: NUMERO, I
INTEGER, DIMENSION(10) :: LISTA= [14, (0, I= 1,9)]
NAMELIST /SAI/ NUMERO, LISTA
WRITE(6, NML=SAI)
```

o qual produz o seguinte registro no arquivo associado à unidade lógica 6:

```
&SAI NUMERO=1 , LISTA=14, 9*0 /
```

Nota-se que os 9 elementos 0 do vetor LISTA são registrados usando-se a notação compacta 9*0. Esta notação também é válida em registros de entrada.

Em um processo de leitura de registros (usando o comando READ, por exemplo), não é permitida a mistura de registros em um NAMELIST com registros que não pertencem a nenhum grupo, como em uma entrada formatada ou não-formatada. Contudo, em um processo de saída de registros (comando WRITE), pode-se misturar os tipos de registros.

Todos os nomes de grupos, nomes de objetos e nomes de componentes são interpretados sem considerar o caso de capitalização. A lista de valores não pode ter um número excedente de ítens, mas ela pode conter um número inferior, como já foi mencionado. Se o objeto é do tipo caractere e o valor tiver uma extensão menor que a declarada, espaços em branco preenchem o restante do campo. Matrizes ou objetos de tamanho zero (seção 6.5) não podem constar no registro de entrada de um NAMELIST. Se ocorrer uma múltipla atribuição de um mesmo objeto, o último valor é assumido.

Por fim, comentários podem ser incluídos no registro de entrada em seguida a um nome a uma vírgula que separa os nomes. O comentário é introduzido, como é costumeiro a partir do caractere !. Uma linha de comentário, iniciada com o caractere !, também é permitida, desde que esta não ocorra em um contexto de uma constante de caractere.

O programa Testa_NAMELIST abaixo ilustra praticamente todos os recursos discutidos nesta seção a respeito de listas NAMELIST.

```
program Testa_NAMELIST
use, intrinsic :: iso_fortran_env, only: dp => real64
implicit none
integer :: no_de_ovos, litros_de_leite, quilos_de_arroz= 12, numero
! Note que quilos_de_arroz foi inicializado.
integer, dimension(10) :: lista_int1, lista_int2
real, dimension(10) :: aleat, vet_tes= 1.0
real(dp) :: pi, e_euler, m_electron, c_vacuum
logical :: tes
character(len= 10) :: nome
namelist /fome/ nome, no_de_ovos, litros_de_leite, quilos_de_arroz
namelist /sai/ numero, lista_int1, lista_int2, vet_tes
namelist /ctes/ tes, pi, e_euler, m_electron, c_vacuum
```

```
open(10, file= "Dados.ent", status= "old")
open(11, file= "Dados.dat")
read(10, nml= fome)
read(10, nml= sai)
read(10, nml= ctes)
! Mostra na tela os valores lidos.
print'(a)', "Lista /fome/ lida:"
write(*, nml= fome)
print'(/a)', "Lista /sai/ lida:"
write(*, nml= sai)
print'(/a)', "Lista /ctes/ lida:"
write(*, nml= ctes)
print'(/a)', "Para confirmar:"
print'("Vetor vet_tes: ",10(f4.2,x))', vet_tes
print'(/,"lista 1:",10i3)', lista_int1
print'(/,"lista 2:",10i3)', lista_int2
! Altera alguns valores e grava as listas no arquivo.
call random_number(aleat)
print'(/,"Lista de No. aleatorios:",/,10(f7.5,x))', aleat
lista_int1= lista_int2**2
lista_int2= aleat*lista_int2
write(11, nml= sai)
write(11, nml= ctes)
write(11, nml= fome) ! Verifique Dados.dat.
end program Testa_NAMELIST
```

Um exemplo de arquivo de entrada `Dados.ent` que pode ser fornecido ao programa `Testa_NAMELIST` é o seguinte:

```
&fome
litros_de_leite= 10,
no_de_ovos= 24,
nome= "Rancho"
/

&sai
numero= 50,
lista_int1= 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
lista_int2= 3*1, 3*5, 4*8,
vet_tes(4:8)= 5*2.0
/

&ctes
tes= .true.,
pi=          3.14159265358979323846264338327950288419717d+00,
e_euler=     2.71828182845904523536028747135266249775725d+00, !Euler's number.
m_electron= 9.10938215d-31,        !Electron rest mass (kg).
c_vacuum=    2.99792458d+08        !Speed of light in vacuum (m/s).
/
```

10.3 INSTRUÇÃO INCLUDE

Em certas situações, durante a elaboração do código-fonte de uma determinada unidade de programa, pode ser útil incluir-se texto oriundo de um lugar distinto, tal como um outro arquivo que contém parte ou o todo de uma unidade de programa. Este recurso é possível fazendo-se uso da instrução `include`:

```
include <constante de caracteres>
```

onde <constante de caracteres> não pode conter um parâmetro de espécie do tipo de caracteres. Esta instrução deve ser inserida no código-fonte em uma linha própria e, no momento da compilação do código-fonte, esta linha será substituída pelo material identificado pela <constante de caracteres>, de uma maneira que depende do processador. O texto incluído pode, por sua vez, conter outras linhas `include`, as quais serão similarmente substituídas.

Uma linha `include` não pode fazer referência a si própria, direta ou indiretamente. Quando uma linha `include` é substituída, a primeira linha incluída não pode ser uma linha de continuação e a última linha não pode ser continuada. A linha de código contendo a instrução `include` pode ter um comentário ao final, mas não pode ser rotulada ou conter instruções incompletas.

A instrução `include` já estava disponível no Fortran 77 como uma extensão ao padrão da linguagem. Usualmente, a instrução era empregada para assegurar que as ocorrências dos blocos `common` fossem todas idênticas nas unidades que continham as linhas `include`. No Fortran , os blocos `common` são substituídos por módulos, com diversas vantagens, mas as instruções `include` podem ainda ser úteis quando se deseja compartilhar o mesmo bloco de instruções ou declarações (ou mesmo unidades de programa completas) entre distintos módulos, por exemplo. Contudo, cautela deve ser exercida para evitar a ocorrência de uma definição repetida de uma mesma rotina em diferentes módulos.

10.4 Unidades lógicas

Em Fortran, um arquivo está conectado a uma *unidade lógica* denotada por um número inteiro. Este número deve ser positivo e está frequentemente limitado entre 1 e 100, dependendo do compilador. Cada unidade lógica possui muitas propriedades. Por exemplo,

`FILE`. O nome do arquivo conectado à unidade. O nome é especificado pelo comando `OPEN`.

`ACTION`. Especifica o tipo de ação que pode ser executada sobre o arquivo. Estas compreendem leitura, escrita ou ambas. Se um arquivo é aberto para um tipo de ação e outro tipo é tentado, uma mensagem de erro será gerada. Por exemplo, não é permitida escrita sobre um arquivo aberto somente para leitura.

`STATUS`. Estabelece o status de um arquivo. Estes são: velho *(old)*, novo *(new)* substituição *(replace)*, entre outros. Caso um arquivo tente ser acessado com um status incorreto, uma mensagem de erro é gerada.

`ACCESS`. Forma de acesso a um arquivo: direto ou sequencial. Alguns arquivos pode ser acessados por ambos os métodos, como é o caso de arquivos em discos removíveis ou discos rígidos. Já outras mídias permite acesso somente sequencial, como é o caso de uma fita magnética.

- **Sequencial.** É o acesso usual. Os processos de leitura/escrita principiam no início do arquivo e seguem, linha a linha, até o seu final.
- **Direto.** Cada linha é acessada por um número, denominado *número de gravação (record number)* o qual deve ser especificado no comando de leitura/escrita.

O número máximo de arquivos que podem ser abertos simultaneamente para E/S também é especificado pelo manual do compilador.

10.5 Arquivos internos

Arquivos internos são um mecanismo no Fortran para a conversão de um objeto de dados para uma string ou *vice-versa*. Os usos principais deste recurso consistem na criação de nomes de arquivos, especificações de formatos ou linhas de saída, todos contendo uma mistura de strings com dados numéricos.

Como um exemplo do segundo uso, o programa abaixo declara a string `BUFFER`, contendo 30 caracteres, e a inicializada como uma sequência de caracteres numéricos.

```
1 program internal_files1
2 implicit none
```

```
3  character(len= 6), dimension(3) :: form= ['(30i1)', '(15i2)', '(10i3)']
4  character(len= 30) :: buffer= '283764398475304984298239769256'
5  integer :: key, i
6  integer, dimension(:), allocatable :: ival
7  print'(a)', 'Numerais lidos:'
8  do key= 1, size(form)
9     allocate(ival(30/key))
10    read(buffer, fmt= form(key)) (ival(i), i= 1, 30/key)
11    print'(a,g0,a,*(g0, x))', 'key= ', key, ': ', (ival(i), i= 1, 30/key)
12    deallocate(ival)
13 end do
14 end program internal_files1
```

A variável inteira KEY pode assumir valores 1, 2 ou 3. O valor de KEY determina qual elemento do vetor FORM será adotado como especificador de formatação de leitura do arquivo interno. O descritores 30I1, 15I2 e 10I3 determinam então se BUFFER será interpretada como uma sequência de 30 números inteiros (descritor I: números inteiros) de um dígito, 15 números de dois dígitos ou 10 números de três dígitos.

Na linha 10, a string BUFFER é tratada como um arquivo interno, o qual é lido sendo o seu conteúdo convertido para um vetor de números inteiros de acordo com o formato descrito pelo elemento FORM(KEY). Estes números são então atribuídos aos elementos do vetor alocável IVAL, o qual foi alocado com o número exato de elementos, de acordo com o valor de KEY.

Finalmente, na linha 11 os números inteiros lidos do arquivo interno são impressos na tela. Note que na formatação da saída padrão, o asterisco "*" indica a *repetição ilimitada de formato* (*unlimited format repetition*), significando que o subformato (G0, X) deve ser repetido sem limite pré-fixado enquanto existirem dados a serem transferidos em uma operação de E/S.

A execução do programa INTERNAL_FILES1 fornece a saída:

```
Numerais lidos:
key= 1: 2 8 3 7 6 4 3 9 8 4 7 5 3 0 4 9 8 4 2 9 8 2 3 9 7 6 9 2 5 6
key= 2: 28 37 64 39 84 75 30 49 84 29 82 39 76 92 56
key= 3: 283 764 398 475 304 984 298 239 769 256
```

A terceira tarefa para a qual pode-se empregar arquivos internos consiste na criação de uma string a partir de combinações de variáveis de caracteres com dados numéricos. Este tipo de tarefa é útil para a criação de nomes de arquivos ou cabeçalhos de arquivos de dados contendo informações a respeito dos mesmos. O programa abaixo mostra uma implementação simples desta tarefa.

```
1  program internal_files2
2  implicit none
3  integer, dimension(8) :: vls
4  character(len= 100) :: linha
5  call date_and_time(values= vls)
6  write(linha, fmt= '(6(a,g0))')'Neste momento são as ', vls(5), 'h', vls(6), &
7                                 'm', vls(7), 's do dia ', vls(3), '/', &
8                                vls(2), '/', vls(1)
9  print'(a)', trim(linha)
10 end program internal_files2
```

A sub-rotina intrínseca DATE_AND_TIME (seção 8.17.1) é invocada para fornecer a data e a hora em que o código é executado. Estes valores são depositados como números inteiros no vetor VLS. Então, na linha 6 a string LINHA é tratada como um arquivo interno no qual são gravadas na forma de uma string as informações de data e hora, de acordo com a formatação fornecida no comando WRITE. O valor de LINHA é então impresso na tela. A execução do programa INTERNAL_FILES2 resultou na seguinte saída:

```
Neste momento são as 11h37m40s do dia 22/10/2019
```

10.6 COMANDO OPEN

O comando OPEN é usado para conectar um dado arquivo a uma unidade lógica. Frequentemente é possível pré-conectar um arquivo antes que o programa comece a rodar; neste caso, não é necessário usar-se o comando OPEN. Contudo, isto depende de conhecimento prévio a respeito dos números-padrão das unidades lógicas para E/S, o que geralmente depende do processador e dos sistema operacional em uso. Para sistemas linux, as unidades pré-conectadas são, usualmente:

Uso	Designação	Número da unidade lógica
Mensagens de erro	stderr	0
Entrada padrão	stdin	5
Saída padrão	stdout	6

Sempre que possível, é recomendado evitar-se o uso de unidade pré-conectadas, o que torna o programa mais portável.

Sugestões de uso & estilo para programação

É possível que diferentes sistemas ou compiladores adotem diferentes números para as unidades lógicas pré-conectadas. Para evitar a ocorrência de erros, é sugerido empregar-se as constantes nomeadas error_unit, input_unit e output_unit fornecidas no módulo intrínseco iso_fortran_env (seção 9.13.6).

A sintaxe do comando é:

```
OPEN([UNIT=] <int-exp> [,<op-list>])
```

onde <int-exp> é uma expressão escalar inteira que especifica o número da unidade lógica externa e <op-list> é uma lista opcional de especificadores, formada por um conjunto de palavras-chave. Se a palavra-chave "UNIT=" for incluída, ela pode aparecer em qualquer posição no campo de argumentos do comando OPEN. Um especificador não pode aparecer mais de uma vez. Nos especificadores, todas as entidades são escalares e todos os caracteres são da espécie padrão. Em expressões de caractere, todos os brancos precedentes são ignorados e, exceto no caso do especificador "FILE=", quaisquer letras maiúsculas são convertidas às correspondentes letras minúsculas. Os especificadores mais comuns são:

FILE= <fln>, onde <fln> é uma expressão de caractere que fornece o nome do arquivo. O nome deve coincidir exatamente com o arquivo, inclusive com o caminho, caso o arquivo esteja em um diretório distinto do diretório de trabalho. Se este especificador é omitido e a unidade não é conectada a uma arquivo, o especificador "STATUS=" deve ser especificado com o valor SCRATCH e o arquivo conectado à unidade irá depender do sistema.

IOSTAT= <ios>, onde <ios> é uma variável inteira padrão que é fixada a zero se o comando executou corretamente e a um valor positivo em caso contrário.

IOMSG= <iom>, onde <iom> é uma variável de caracteres padrão que terá atribuída uma mensagem gerada pelo processador caso ocorra um erro na execução do comando.

STATUS= <status>, onde <status> é uma expressão de caracteres que fornece o status do arquivo. O status pode ser um dos seguintes:

- **'OLD'–** Arquivo deve existir.
- **'NEW'–** Arquivo não deve existir. O arquivo será criado pelo comando OPEN. A partir deste momento, o seu status se torna 'OLD'.
- **'REPLACE'–** Se o arquivo não existe, ele será criado. Se o arquivo já existe, este será eliminado e um novo arquivo é criado com o mesmo nome. Em ambos os casos, o status é então alterado para 'OLD'.
- **'SCRATCH'–** O arquivo é temporário e será deletado quando este for fechado com o comando CLOSE ou na saída da unidade de programa.
- **'UNKNOWN'–** Status do arquivo desconhecido; depende do sistema. Este é o valor padrão do especificador, caso este seja omitido.

O especificador "FILE=" deve estar presente se 'NEW' ou 'REPLACE' são especificados ou se 'OLD' é especificado e a unidade não está conectada.

Os demais especificadores fornecidos pelo padrão são:

ACCESS= <acc>, onde <acc> é uma expressão de caracteres que fornece o valor 'SEQUENTIAL', 'DIRECT' ou 'STREAM'. Para um arquivo que já exista, este valor deve ser uma opção válida. Se o arquivo ainda não existe, ele será criado com o método de acesso apropriado. Se o especificador é omitido, o valor 'SEQUENTIAL' é assumido. O significado das especificações é:

'DIRECT'– O arquivo consiste em registros acessados por um número de identificação. Neste caso, o tamanho do registro **deve** ser especificado por "RECL=". Registros individuais podem ser especificados e atualizados sem alterar o restante do arquivo.

'SEQUENTIAL'– O arquivo é escrito/lido sequencialmente, linha a linha.

'STREAM'- Indica acesso por *fluxo* (*stream*), no qual o arquivo é posicionado em **unidades de armazenagem de arquivo** (*file storage units*), normalmente bytes, começando na posição 1. Para um arquivo aberto com esta forma de acesso, o especificador POS= deve ser incluído nos comandos INQUIRE, READ ou WRITE.

ACTION= <act>, onde <act> é uma expressão de caracteres que fornece os valores 'READ', 'WRITE' ou 'READWRITE'.

'READ'– Se esta especificação é escolhida, os comandos WRITE, PRINT e ENDFILE não devem ser usados para esta conexão.

'WRITE'– Se esta especificação é escolhida, o comando **READ** não pode ser usado.

'READWRITE'– Se esta especificação é escolhida, não há restrição.

Se o especificador é omitido, o valor padrão depende do processador.

ASYNCHRONOUS= <asyn>, onde <asyn> pode ser 'YES' ou 'NO' (valor padrão). Indica se E/S assíncrona será permitida para a unidade lógica.

'YES'- Indica que E/S assíncrona será permitida para a unidade.

'NO'- Indica que E/S assíncrona não será permitida para a unidade.

BLANK= <bl>, onde <bl> é uma expressão de caracteres que fornece os valores 'NULL' ou 'ZERO'. A conexão deve ser com E/S formatada. Este especificador determina o padrão para a interpretação de brancos em campos de entrada numéricos.

'NULL'– Os brancos são ignorados, exceto que se o campo for completamente em branco, ele é interpretado como zero.

'ZERO'– Os brancos são interpretados como zeros.

Se o especificados é omitido, o valor padrão é 'NULL'.

DECIMAL= <dec>, onde <dec> é uma expressão de caracteres com os valores 'POINT' (valor padrão) ou 'COMMA'. O valor de <dec> especifica o modo de edição decimal.

'POINT'- Emprega o ponto como separador decimal.

'COMMA'- Emprega a vírgula como separador decimal.

DELIM= <del>, onde <del> é uma expressão de caracteres que fornece os valores 'APOSTROPHE', 'QUOTE' ou 'NONE'. Se 'APOSTROPHE' ou 'QUOTE' são especificados, o caractere correspondente será usado para delimitar constantes de caractere escritos com formatação de lista ou com o uso da declaração NAMELIST, e será duplicado onde ele aparecer dentro de tal constante de caractere. Além disso, caracteres fora do padrão serão precedidos por valores da espécie. Nenhum caractere delimitador será usado e nenhuma duplicação será feita se a especificação for 'NONE'. A especificação 'NONE' é o valor padrão se este especificador for omitido. O especificador somente pode aparecer em arquivos formatados.

ENCODING= <enc>, onde <enc> é uma expressão de caracteres que fornece os valores 'DEFAULT' (valor padrão) ou 'UTF-8'. Neste último, a *encodificação* (*encoding*) UTF-8 será empregada para o arquivo.

ERR= <rótulo-erro>, onde <rótulo-erro> é o rótulo de um comando na mesma unidade de âmbito, para o qual o controle do fluxo será transferido caso ocorra algum erro na execução do comando OPEN.[4]

FORM= <fm>, onde <fm> é uma expressão de caracteres que fornece os valores 'FORMATTED' ou 'UNFORMATTED', determinando se o arquivo deve ser conectado para E/S formatada ou não formatada. Para um arquivo que já exista, o valor deve ser uma opção válida. Se o arquivo ainda não existe, ele será criado com um conjunto de forma que incluem o forma especificada. Se o especificador é omitido, os valores-padrão são:

'FORMATTED'– Para acesso sequencial.

'UNFORMATTED'– Para conexão de acesso direto.

NEWUNIT= <nu>, onde <nu> é uma variável inteira cujo valor é o número inteiro negativo (diferente de -1) correspondente à unidade lógica. O processador irá determinar um valor para <nu> que não colida com nenhum outro valor empregado internamente. O emprego deste especificador evita confusões nos números das unidades lógicas ativas.

PAD= <pad>, onde <pad> é uma expressão de caracteres que fornece os valores 'YES' ou 'NO'.

'YES'– Um registro de entrada formatado será considerado preenchido por brancos sempre que uma lista de entrada e a formatação associada especificarem mais dados que aqueles que são lidos no registro.

'NO'– Neste caso, o tamanho do registro de entrada deve ser não menor que aquele especificado pela lista de entrada e pela formatação associada, exceto na presença de um especificador ADVANCE= 'NO' e uma das especificações "EOR=" ou "IOSTAT=".

O valor padrão se o especificador é omitido é 'YES'. Para caracteres não padronizados, o caractere de branco que preenche o espaço restante depende do processador.

POSITION= <pos>, onde <pos> é uma expressão de caracteres que fornece os valores 'ASIS', 'REWIND' ou 'APPEND'. O método de acesso deve ser sequencial e se o especificador é omitido, o valor padrão é 'ASIS'. Um arquivo novo é sempre posicionado no seu início. Para um arquivo que já existe e que já está conectado:

'ASIS'– O arquivo é aberto sem alterar a posição corrente de leitura/escrita no seu interior.

'REWIND'– O arquivo é posicionado no seu ponto inicial.

'APPEND'– O arquivo é posicionado logo após o registro de final de arquivo, possibilitando a inclusão de dados novos.

Para um arquivo que já existe mas que não está conectado, o efeito da especificação 'ASIS' no posicionamento do arquivo é indeterminado.

RECL= <rl>, onde <rl> é uma expressão inteira cujo valor deve ser positivo.

- Para arquivo de acesso direto, a opção especifica o tamanho dos registros e é obrigatório.
- Para arquivo sequencial, a opção especifica o tamanho máximo de um registro, e é opcional com um valor padrão que depende do processador.
  - Para arquivos formatados, o tamanho é o número de caracteres para registros que contenham somente caracteres-padrão.
  - Para arquivos não formatados, o tamanho depende do sistema, mas o comando INQUIRE (seção 10.12) pode ser usado para encontrar o tamanho de uma lista de E/S.

[4] O uso de um rótulo não é recomendado. Sua descrição pode ser vista em Apostila de Fortran 90/95 (seção 5.1.1).

Em qualquer situação, para um arquivo que já exista, o valor especificado deve ser permitido para o arquivo. Se o arquivo ainda não existe, ele é criado com um conjunto permitido de tamanhos de registros que incluem o valor especificado.

ROUND= <rnd>, onde <rnd> é uma expressão de caracteres que fornece os valores 'UP', 'DOWN', 'ZERO', 'NEAREST', 'COMPATIBLE' ou 'PROCESSOR_DEFINED' (valor padrão). O valor de <rnd> determina o modo de arredondamento para operações de E/S do arquivo, sendo:

'UP'- O menor valor representável que é maior que ou igual ao valor original.

'DOWN'- O maior valor representável que é menor que ou igual ao valor original.

'ZERO'- O valor mais próxima ao valor original, mas não maior em magnitude que o valor original.

'NEAREST'- O mais próximo dos dois valores mais próximos representáveis, se um destes estiver mais próximo que o outro.

'COMPATIBLE'- O mais próximo dos dois valores mais próximos representáveis. Se o valor estiver no meio do intervalo entre os valores representáveis, o escolhido é o mais distante de zero.

'PROCESSOR_DEFINED'- O valor é determinado pela configuração-padrão do processador, a qual pode ser um dos modos acima.

SIGN= <sgn>, onde <sgn> é uma expressão de caracteres que fornece os valores 'PLUS', 'SUPPRESS' ou 'PROCESSOR_DEFINED' (valor padrão). O valor de <sgn> controla se o caractere "+" será impresso na saída de valores numéricos, sendo que:

'PLUS'- Indica que o processador deve produzir o caractere "+" sempre que for aplicável.

'SUPPRESS'- Indica que o processador deve suprimir o caractere "+".

'PROCESSOR_DEFINED'- Indica que o processador determina se o caractere "+" será impresso ou não.

A seguir, temos um exemplo do uso deste comando:

```
OPEN(17, FILE= 'SAIDA.DAT', ERR= 10, STATUS= 'REPLACE', &
     ACCESS= 'SEQUENTIAL', ACTION= 'WRITE')
```

Neste exemplo, um arquivo SAIDA.DAT é aberto para escrita, é conectado ao número de unidade lógica 17, o arquivo é acessado linha a linha e ele já existe mas deve ser substituído. O rótulo 10 deve pertencer a um comando executável válido.

```
OPEN(14, FILE= 'ENTRADA.DAT', ERR= 10, STATUS= 'OLD',  &
     RECL=IEXP, ACCESS= 'DIRECT', ACTION= 'READ')
```

Aqui, o arquivo ENTRADA.DAT é aberto somente para leitura na unidade 14. O arquivo é diretamente acessado e deve ser um arquivo previamente existente.

10.7 COMANDO READ

Este é o comando que executa a leitura formatada de dados. Até agora, o comando foi utilizado para leitura na entrada padrão, que usualmente é o teclado:

```
READ*, <lista-input>
```

onde o asterisco "*" indica a entrada padrão e <lista-input> é a lista de variáveis cujo valor é lido do teclado.

Este comando será agora generalizado para uma forma com ou sem unidade lógica. Sem o número de unidade lógica o comando fica:

```
READ <format> [,<lista-input>]
```

e com o número de unidade fica:

```
READ(<lista-input-spec>) <lista-input>
```

Onde <lista-input-spec> é uma lista dos especificadores do comando READ, os quais são os seguintes:

[UNIT=] <u>, onde <u> é um número de unidade lógica válido ou "*" para a entrada padrão. Caso esta especificação seja o primeiro argumento do comando, a palavra-chave é opcional.

[FMT=] <format>, onde <format> é uma string de caracteres formatadores, um rótulo válido em Fortran ou "*" para formato livre. Caso esta especificação seja o segundo argumento do comando, a palavra-chave é opcional.

[NML=] <grp>, onde <grp> é o nome de um grupo NAMELIST (seção 10.2) previamente definido em uma declaração NAMELIST. O nome <grp> não é uma constante de caracteres e sim um nome válido em Fortran.

ADVANCE= <mode>, onde <mode> possui dois valores: 'YES' ou 'NO'. A opção 'NO' especifica que cada comando READ inicia um novo registro na mesma posição; isto é, implementa entrada de dados sem avanço (*non-advancing I/O*). A opção padrão é 'YES', isto é a cada leitura o arquivo é avançado em uma posição. Se a entrada sem avanço é usada, então o arquivo deve estar conectado para acesso sequencial e o formato deve ser explícito.

ASYNCHRONOUS= <asyn>, onde <asyn> pode ser 'YES' ou 'NO' (valor padrão). Indica se entrada assíncrona será permitida para a unidade lógica.

'YES'- Indica que entrada assíncrona será permitida para a unidade.

'NO'- Indica que entrada assíncrona não será permitida para a unidade.

BLANK= <bl>, onde <bl> é uma expressão de caracteres que fornece os valores 'NULL' (valor padrão) ou 'ZERO'. A conexão deve ser com entrada formatada. Este especificador determina o padrão para a interpretação de brancos em campos de entrada numéricos.

'NULL'- Os brancos são ignorados, exceto que se o campo for completamente em branco, ele é interpretado como zero.

'ZERO'- Os brancos são interpretados como zeros.

DECIMAL= <dec>, onde <dec> é uma expressão de caracteres com os valores 'POINT' (valor padrão) ou 'COMMA'. O valor de <dec> especifica o modo de edição decimal.

'POINT'- Emprega o ponto como separador decimal.

'COMMA'- Emprega a vírgula como separador decimal.

END= <end-label>, é um rótulo válido para onde o controle de fluxo é transferido se uma condição de final de arquivo é encontrada. Esta opção somente existe no comando READ com acesso sequencial.

EOR= <eor-label>, é um rótulo válido para onde o controle de fluxo é transferido se uma condição de final de registro é encontrada. Esta opção somente existe no comando READ com acesso sequencial e formatado e somente se a especificação ADVANCE= 'NO' também estiver presente.

ERR= <err-label>, é um rótulo válido para onde o controle de fluxo é transferido quando ocorre um erro de leitura.

ID= <id-var>, onde <id-var> é uma variável escalar inteira padrão que será usada como um identificador. Se ASYNCHRONOUS= 'YES' é especificado e a operação completa com sucesso, o identificador toma um valor dependente do processador que pode ser empregado um comandos WAIT ou INQUIRE posteriores para identificar uma operação de escrita em particular. Se um erro ocorrer, <id-var> se torna indefinido.

IOMSG= <iom>, onde <iom> é uma variável de caracteres padrão que terá atribuída uma mensagem gerada pelo processador caso ocorra um erro na execução do comando.

IOSTAT= <ios>, onde <ios> é uma variável inteira padrão que armazena o status do processo de leitura. Os valores possíveis são:

<ios> = 0, quando o comando é executado sem erros.

<ios> > 0, quando ocorre um erro na execução do comando.

<ios> < 0, quando uma condição de final de registro é detectada em entrada sem avanço ou quando uma condição de final de arquivo é detectada.

PAD= <pad>, onde <pad> é uma expressão de caracteres que fornece os valores 'YES' ou 'NO'.

'YES'– Um registro de entrada formatado será considerado preenchido por brancos sempre que uma lista de entrada e a formatação associada especificarem mais dados que aqueles que são lidos no registro.

'NO'– Neste caso, o tamanho do registro de entrada deve ser não menor que aquele especificado pela lista de entrada e pela formatação associada, exceto na presença de um especificador ADVANCE= 'NO' e uma das especificações "EOR=" ou "IOSTAT=".

O valor padrão se o especificador é omitido é 'YES'. Para caracteres não padronizados, o caractere de branco que preenche o espaço restante depende do processador.

POS= <pos>, onde <pos> é uma expressão escalar inteira padrão que especifica a posição no arquivo, em unidades de armazenagem de arquivo, em um arquivo com acesso de fluxo (*stream*). Somente pode ser empregado se o arquivo for aberto para acesso de fluxo. Se POS= for omitido, a entrada em fluxo ocorre partindo da próxima posição no arquivo a partir da posição corrente.

REC= <int-exp>, onde <int-exp> é uma expressão inteira escalar cujo valor é o número de índice do registro lido durante acesso direto. Se REC estiver presente, os especificadores END e NML e o formato livre "*" não podem ser também usados.

ROUND= <rnd>, onde <rnd> é uma expressão de caracteres que fornece os valores 'UP', 'DOWN', 'ZERO', 'NEAREST', 'COMPATIBLE' ou 'PROCESSOR_DEFINED' (valor padrão). O valor de <rnd> determina o modo de arredondamento para operações de entrada do arquivo, sendo:

'UP'- O menor valor representável que é maior que ou igual ao valor original.

'DOWN'- O maior valor representável que é menor que ou igual ao valor original.

'ZERO'- O valor mais próxima ao valor original, mas não maior em magnitude que o valor original.

'NEAREST'- O mais próximo dos dois valores mais próximos representáveis, se um destes estiver mais próximo que o outro.

'COMPATIBLE'- O mais próximo dos dois valores mais próximos representáveis. Se o valor estiver no meio do intervalo entre os valores representáveis, o escolhido é o mais distante de zero.

'PROCESSOR_DEFINED'- O valor é determinado pela configuração-padrão do processador, a qual pode ser um dos modos acima.

SIZE= <size>, onde <size> é uma variável escalar inteira padrão, a qual guarda o número de caracteres lidos. Este especificador somente existe no comando READ e somente se a especificação ADVANCE= 'NO' também estiver presente.

Dado o seguinte exemplo:

```
READ(14, FMT= '(3(F10.7,1X))', REC=IEXP) A, B, C
```

este comando especifica que o registro identificado pela variável inteira IEXP seja lido na unidade 14. O registro deve ser composto por 3 números reais designados pelos descritores F10.7, separados por um espaço em branco. Estes 3 números serão atribuídos às variáveis A, B e C. Outro exemplo:

```
READ(*, '(A)', ADVANCE= 'NO', EOR= 12, SIZE= NCH) STR
```

especifica que a leitura é sem avanço na unidade padrão; isto é, o cursor na tela do monitor irá permanecer na mesma linha há medida que a string STR será lida. O comando lê uma constante de caractere porque o descritor no formato é "A". Em uma situação normal, isto é, leitura com avanço, o cursor seria posicionado no início da próxima linha na tela. A variável NCH guarda o comprimento da string e o rótulo 12 indica o ponto onde o programa deve se encaminhar se uma condição de final de registro é encontrada.

10.8 COMANDOS PRINT E WRITE

Em comparação com o comando READ, o comando WRITE suporta os mesmos argumentos, com exceção do especificador SIZE.

Até agora, foi usada somente forma do comando PRINT para a saída padrão de dados, que usualmente é a tela do monitor:

```
PRINT*, <lista-output>
```

O mesmo comando, generalizado para saída formatada fica,

```
PRINT <format> [, <lista-output>]
```

Já a sintaxe mais geral do comando é:

```
WRITE(<lista-output-spec>) <lista>
```

Onde <lista-output-spec> é uma lista dos especificadores do comando WRITE, os quais são:

```
[UNIT=] <u>            DECIMAL= <dec>     IOSTAT= <ios>
[FMT=] <format>        DELIM= <dlm>       POS= <pos>
[NML=] <grp>           ERR= <err-label>   REC= <int-exp>
ADVANCE= <mode>        ID= <id-var>       ROUND= <rnd>
ASYNCHRONOUS= <asyn>   IOMSG= <iom>       SIGN= <sgn>
```

Os especificadores deste comando executam, na sua maioria, as mesmas funções executadas pelos especificadores do comando READ, com exceção dos seguintes, alguns dos quais especificam funções ligeiramente distintas:

[UNIT=] <u>, onde <u> é um número de unidade lógica válido ou "*" para a saída padrão. Caso esta especificação seja o primeiro argumento do comando, a palavra-chave é opcional.

ADVANCE= <mode>, onde <mode> possui dois valores: 'YES' ou 'NO'. A opção 'NO' especifica que cada comando WRITE inicia um novo registro na mesma posição; isto é, implementa saída de dados sem avanço (*non-advancing I/O*). A opção padrão é 'YES', isto é a cada escrita, o arquivo é avançado em uma posição. Se a saída sem avanço é usada, então o arquivo deve estar conectado para acesso sequencial e o formato deve ser explícito.

ASYNCHRONOUS= <asyn>, onde <asyn> pode ser 'YES' ou 'NO' (valor padrão). Indica se saída assíncrona será permitida para a unidade lógica.

DELIM= <del>, onde <del> é uma expressão de caracteres que fornece os valores 'APOSTROPHE', 'QUOTE' ou 'NONE' (valor padrão). Se o valor de <del> for 'APOSTROPHE', apóstrofes irão delimitar as constantes de caracteres. Se for 'QUOTE', aspas serão empregadas e se for 'NONE', as constantes de caracteres não terão delimitadores.

ERR= <err-label>, é um rótulo válido para onde o controle de fluxo é transferido quando ocorre um erro de escrita.

IOSTAT= <ios>, onde <ios> é uma variável inteira padrão que armazena o status do processo de escrita.

NML= <grp>, onde <grp> é o nome de um grupo NAMELIST previamente definido em uma declaração NAMELIST. O nome <grp> não é uma constante de caracteres e sim um nome válido em Fortran.

REC= <int-exp>, onde <int-exp> é uma expressão inteira escalar cujo valor é o número de índice do registro a ser escrito durante acesso direto. Se REC estiver presente, os especificadores END e NML e o formato livre "*" não podem ser também usados.

ROUND= <rnd>, onde <rnd> é uma expressão de caracteres que fornece os valores 'UP', 'DOWN', 'ZERO', 'NEAREST', 'COMPATIBLE' ou 'PROCESSOR_DEFINED' (valor padrão). O valor de <rnd> determina o modo de arredondamento para operações de saída do arquivo.

SIGN= <sgn>, onde <sgn> é uma expressão de caracteres que fornece os valores 'PLUS', 'SUPPRESS' ou 'PROCESSOR_DEFINED' (valor padrão). O valor de <sgn> controla se o caractere "+" será impresso na saída de valores numéricos (ver seção 10.6).

Considere o seguinte exemplo,

```
WRITE(17, FMT= '(I4)', IOSTAT= ISTAT, ERR= 10) IVAL
```

aqui, '(I4)' descreve format de dado inteiro com 4 dígitos, ISTAT é uma variável inteira que indica o status do processo de escrita; em caso de erro, o fluxo é transferido ao rótulo 10. IVAL é a variável cujo valor é escrito na unidade 17. Outro exemplo seria:

```
WRITE(*,'(A)', ADVANCE= 'NO') 'Entrada: '
```

Como a especificação de escrita sem avanço foi escolhida, o cursor irá permanecer na mesma linha que a string 'Entrada: '. Em circunstâncias normais, o cursor passaria ao início da linha seguinte na tela. Nota-se também o descritor de formato de constante de caracteres explícito.

Como exemplo do uso dos comandos de E/S apresentados nas seções 10.6 – 10.8, o programa a seguir abre um novo arquivo sequencial chamado Notas.dat. O programa então lê o nome de um estudante, seguido por 3 notas, as quais são introduzidas pelo teclado (entrada padrão), escrevendo, finalmente, as notas e a média final em Notas.dat e no monitor (saída padrão), na mesma linha do nome. O processo de leitura e escrita é repetido até que um estudante com o nome 'fim' é introduzido.

```
program Notas
implicit none
character(len= 20) :: nome
real :: mark1, mark2, mark3
open (unit=4,file='Notas.dat')
do
   read(*,*) nome, mark1, mark2, mark3
   if(nome == 'fim')exit
   write(unit= 4,fmt=*) nome, mark1, mark2, mark3, (mark1 + mark2 + mark3)/3.0
   write(unit= *,fmt=*) nome, mark1, mark2, mark3, (mark1 + mark2 + mark3)/3.0
end do
close(unit= 4)
end program Notas
```

10.9 COMANDO FORMAT E ESPECIFICADOR FMT=

O especificador FMT= em um comando READ ou WRITE pode conter ou o símbolo "*" identificando uma saída formatada em lista dirigida,[5] ou uma constante de caracteres que indica o formato ou um rótulo que indica a linha onde se encontra um comando FORMAT.

Dado o seguinte exemplo,

```
WRITE(17, FMT= '(2X,2I4,1X,"nome ",A7)')I, J, STR
```

este comando escreve no arquivo SAIDA.DAT, aberto em um exemplo anterior, as três variáveis I, J e STR de acordo com o formato especificado: "2 espaços em branco (2X), 2 objetos inteiros de 4 dígitos em espaços entre eles (2I4), um espaço em branco (1X) a constante de caractere 'nome ' e, finalmente, 7 caracteres de um objeto de caracteres (A7)". As variáveis a ser realmente escritas são tomadas da lista de E/S após a lista de especificações do comando.

Outro exemplo:

```
READ(14,*)X, Y
```

Este comando lê dos valores do arquivo ENTRADA.DAT usando formato livre e atribuindo os valores às variáveis X e Y.

Finalmente, no exemplo:

[5] Do inglês: *list-directed.* Este tipo de operação de E/S é muitas vezes denominado *formato livre.*

```
WRITE(*,FMT= 10)A, B
10 FORMAT('Valores: ',2(F15.6,2X))
```

O comando `WRITE` acima usa o formato especificado no rótulo `10` para escrever na saída padrão os valores das variáveis `A` e `B`. O formato de saída pode ser descrito como: "2 instâncias de: um objeto real com 15 colunas ao todo com uma precisão de 6 casas decimais `(F15.6)`, seguido por 2 espaços em branco `(2X)`".

O formato de E/S de dados é definido fazendo-se uso dos *descritores de edição*, abordados na próxima seção.

10.10 Descritores de edição

Nos exemplos apresentados nas seções anteriores, alguns descritores de edição já foram mencionados. Estes editores fornecem uma especificação precisa sobre como os valores devem ser convertidos em uma string escrita em um dispositivo externo (monitor, impressora, *etc*) ou em um arquivo interno (disco rígido, SSD, DVD, fita, *etc*), ou convertido de uma string em um dispositivo de entrada de dados (teclado, *etc*) ou arquivo interno para as diferentes representações de tipos de variáveis suportados pela linguagem.

Com algumas exceções, descritores de edição aparecem em uma lista separados por vírgulas e somente no caso em que a lista de E/S for vazia ou contiver somente matrizes de tamanho zero que os descritores podem estar totalmente ausentes. Em um processador que suporta tanto caracteres maiúsculos quanto minúsculos, os descritores de edição são interpretados de forma independente com relação ao caso.

Descritores de edição dividem-se em três classes:

Descritores de edição de dados: **`I`**, **`B`**, **`O`**, **`Z`**, **`F`**, **`E`**, **`EN`**, **`ES`**, **`EX`**, **`D`**, **`DT`**, **`G`**, **`L`**, **`A`**.

Descritores de controle: **`T`**, **`TL`**, **`TR`**, **`X`**, **`S`**, **`SP`**, **`SS`**, **`BN`**, **`BZ`**, **`RU`**, **`RD`**, **`RZ`**, **`RN`**, **`RC`**, **`RP`**, **`P`**, **`DC`**, **`DP`**, **`:`**, **`/`**.

Descritores de edição de strings: **`H`**, `'<string>'`, "<string>".

Estes descritores serão discutidos em mais detalhes a seguir.

10.10.1 Contadores de repetição

Os descritores de edição de dados podem ser precedidos por um **contador de repetição** (*repeat count*): um literal inteiro positivo (sem sinal), colocado à esquerda do descritor, como no exemplo

```
10F12.3
```

a qual indica `10` constantes reais com 12 posições ao todo e com precisão de 3 casas decimais cada. Dos descritores de edição restantes, somente a barra "/" pode ter um contador de repetição associado. Um contador de repetição também pode ser aplicado a um **subformato**, isto é, um grupo de descritores de edição delimitado por parênteses, como em:

```
PRINT '(4(I5,F8.2))', (I(J), A(J), J= 1,4)
```

no qual o subformato `(I5,F8.2)` é repetido `4` vezes, o que é equivalente a escrever:

```
PRINT '(I5,F8.2,I5,F8.2,I5,F8.2,I5,F8.2)', (I(J), A(J), J= 1,4)
```

Note também o `Do` implícito (seção 6.6),

```
(I(J), A(J), J= 1,4)
```

o que equivale a escrever

```
I(1), A(1), I(2), A(2), I(3), A(3), I(3), A(3)
```

Contadores de repetição como estes podem ser aninhados:

```
PRINT '(2(2I5,2F8.2))', (I(J),I(J+1),A(J),A(J+1), J=1,4,2)
```

Pode-se ver que esta notação possibilita compactar a formatação de E/S de dados.

Se uma especificação de formato é usada com uma lista de E/S que contém mais elementos que o número de descritores de formatação, incluindo os contadores de repetição, um novo registro irá começar e a especificação de formato será repetida até que todos os elementos na lista forem exauridos. Isto acontece, por exemplo, no comando abaixo:

```
PRINT '(10I8)', (I(J), J= 1, 100)
```

o qual irá imprimir na tela 10 linhas, cada uma contendo 10 números inteiros de 8 dígitos cada.

Para facilitar a gravação de dados no formato CSV (*comma-separated values*),[6] existe o recurso da **repetição ilimitada de formato** (*unlimited format repetition*), indicada pela sintaxe

```
*(<subformato>)
```

onde o asterisco "*" na posição de um contador de repetição indica que não há limite pré-fixado no número de vezes que o subformato será aplicado à lista de saída. O subformato será repetido enquanto houver dados na lista.

Este recurso somente pode aparecer no final de uma especificação de formato de saída. O descritor de controle ":" (seção 10.10.3) pode ser incluído no subformato para evitar que descritores de edição de strings sejam impressos após o último registro numérico. Por exemplo, o comando

```
PRINT'("Lista: ", *(G0,:,", "))', 10, 20, 30, 40
```

irá imprimir na tela:

```
Lista: 10, 20, 30, 40
```

10.10.2 DESCRITORES DE EDIÇÃO DE DADOS

Cada tipo intrínseco de variáveis em Fortran é convertido por um conjunto específico de descritores, com exceção do descritor `G`, que pode converter qualquer tipo intrínseco. A tabela 10.1 apresenta uma descrição das conversões.

Tabela 10.1: Descritores de edição que convertem os tipos intrínsecos.

Descritor	Tipo de dados convertidos
A	Caractere
B	Inteiro de/para base binária
D	Real
DT	Designa rotina específica para E/S de tipos derivados
E	Real com expoente
EN	Real com notação de engenharia
ES	Real com notação científica
EX	Significando hexadecimal
F	Real de ponto flutuante (sem expoente)
G	Todos os tipo intrínsecos
I	Inteiro
L	Lógico
O	Inteiro de/para base octal
Z	Inteiro de/para base hexadecimal

[6] Uma formatação onde os dados são gravados em arquivos de texto (ASCII), separados por vírgulas.

FORMATO DOS DESCRITORES DE EDIÇÃO DE DADOS.

Um descritor de edição de dados pode assumir uma das seguintes formas em uma especificação de formato de E/S:

```
[<r>]<c><w>
[<r>]<c><w>.<m>
[<r>]<c><w>.<d>
[<r>]<c><w>.<d>[E<e>]
```

onde os campos <r>, <w>, <d> e <e> devem ser todos constantes inteiras positivas (sem sinal). Parâmetros de espécie de tipo não podem ser especificados. O significado dos campos é o seguinte:

<r> é o contador de repetição. O seu valor pode variar entre `1` e `2147483647 (2**31 - 1)`. Se <r> for omitido, ele é assumido igual a 1.

<c> é um dos descritores: `I, B, O, Z, F, EN, ES, D, G, L, A`.

<w> é o número total de dígitos no campo (ou a *largura do campo*). Se <w> for omitido, é assumido um valor padrão que pode variar com o sistema. <w> pode ser inclusive igual a **0**.

<m> é o número mínimo de dígitos que devem aparecer no campo (incluindo zeros precedentes). O intervalo permitido para <m> inicia em **0** mas o valor final depende do sistema.

<d> é o número de dígitos à direita do ponto decimal (os *dígitos significativos*). O intervalo de <d> inicia em **0** mas o valor final depende do sistema. O número de dígitos significativos é afetado se um fator de escala é especificado para o descritor de edição.

E identifica um campo de expoente.

<e> é o número de dígitos no expoente. O intervalo de <e> inicia em 1 mas o valor final depende do sistema.

Os descritores de edição de dados têm as seguintes formas específicas:

Tipo de dado	Descritores de dados
Inteiro	`I<w>[.<m>], B<w>[.<m>], O<w>[.<m>], Z<w>[.<m>], G<w>.<d>[E<e>]`
Real ou Complexo	`F<w>.<d>, E<w>.<d>[E<e>], EN<w>.<d>[E<e>], ES<w>.<d>[E<e>], D<w>.<d>, G<w>.<d>[E<e>]`
Lógico	`L<w>, G<w>.<d>[E<e>]`
Caractere	`A[<w>], G<w>.<d>[E<e>]`

O campo <d> deve ser especificado com os descritores `F, E, D` e `G` mesmo se <d> for zero. O ponto decimal também é exigido. Para uma constante de caractere, o campo <w> é opcional, independente do tamanho da constante.

Para todos os descritores numéricos, se o campo de escrita for muito estreito para conter o número completo, de acordo com a descrição fornecida, o campo é preenchido por <w> asteriscos.

Na saída os números são geralmente escritos deslocados para a direita no espaço de campo especificado por <w>; isto é feito preenchendo de espaços em branco à esquerda, caso necessário. Valores negativos são sempre escritos com o sinal de menos.

Na entrada os números também deveriam ser lidos a partir da direita. Espaços em branco à esquerda são ignorados. Se o campo é todo composto por espaços em branco, o registro será lido como zero.

NÚMEROS INTEIROS (I<w>[.<m>], B<w>[.<m>], O<w>[.<m>], Z<w>[.<m>])

Na sua forma básica, `I<w>`, o inteiro será lido ou escrito no campo de largura <w> ajustado à direita. Representando um espaço em branco por "␣", o valor `-99` escrito sob o controle de `I5` irá ser aparecer como `␣␣-99`, com o sinal contando como uma posição no campo.

Para impressão, uma forma alternativa deste descritor permite que o número de dígitos a ser impressos sejam especificados exatamente, mesmo que alguns deles tenham que ser zeros. Trata-se da forma `I<w>.<m>`, onde <m> indica o número mínimo de dígitos a ser impressos. Neste caso, o valor `99` impresso sob o controle de `I5.3` aparecerá como `␣␣099`. O valor <m> pode ser

zero, em cuja situação o campo será preenchido por brancos se o número impresso for 0. Na entrada, I<w>.<m> é interpretado exatamente da mesma forma que I<w>.

Para flexibilizar a saída de dados inteiros e minimizar a quantidade de espaços em branco, o descritor pode ser escrito como I0, o que significa que a largura do campo será automaticamente a menor possível para conter o dado de saída.

Inteiros também podem ser convertidos pelos descritores B<w>[.<m>], O<w>[.<m>] e Z<w>[.<m>]. Estes são similares ao descritor I, porém destinados a representar o número inteiro nos sistemas numéricos binário, octal e hexadecimal, respectivamente.

NÚMEROS REAIS (F<w>.<d>, E<w>.<d>[E<e>], EN<w>.<d>[E<e>], ES<w>.<d>[E<e>], D<w>.<d>)

A forma básica, F<w>.<d>, gera um número real de ponto flutuante. O ponto decimal conta como uma posição no campo. Na entrada de dados, se a string possui um ponto decimal, o valor de <d> é ignorado. Por exemplo, a leitura da string ␣9.3729␣ com o descritor F8.3 causa a transferência do valor 9.3729. Todos os dígitos são usados, mas arredondamento pode ocorrer devido à representação finita de um número pelo computador.

Há outras duas formas de entrada de números aceitáveis pelo descritor F<w>.<d>:

1. Número sem ponto decimal. Neste caso, os <d> dígitos mais à direita serão tomados como a parte fracionário do número. Por exemplo, a string ␣-14629será lida pelo descritor F7.2 como -146.29.

2. Número real na forma padrão; a qual envolve uma parte exponencial (seção 3.1.2), como o número -14.629E-2 ou sua forma variante, onde o expoente sempre tem sinal e o indicador de expoente "E" é omitido: -14.629-2. Neste caso, o campo <d> será novamente ignorado e o número na forma exponencial será escrita na forma de ponto flutuante. Sob o controle do descritor F9.1, a string anterior será convertida ao número 0.14629.

Estas propriedades aplicam-se à entrada de dados.

Na saída de dados, os valores são arredondados seguindo as regras normais de aritmética. Assim, o valor 10.9336, sob o controle do descritor F8.3 irá aparecer escrito como ␣␣10.934 e sob o controle de F4.0 irá aparecer como ␣11. (note o ponto). Por outro lado, para a saída, se <w> = 0, como em F0.3, a largura do campo será a menor possível para conter o dado com <d> dígitos à direita do ponto decimal.

O descritor de edição E possui duas formas, E<w>.<d> e E<w>.<d>E<e> e é uma maneira mais apropriada de transferir números abaixo de 0.01 ou acima de 1000.

As regras para estes descritores na entrada de dados são idênticas às do descritor F<w>.<d>. Na saída com o descritor E<w>.<d>, uma string de caractere contendo a parte inteira com valor absoluto menor que um e quatro caracteres que consistem ou no E seguido por um sinal e dois dígitos ou de um sinal seguido por três dígitos. Assim, o número 1.234×10^{23}, convertido pelo descritor E10.4 vai gerar a string ␣.1234E+24 ou ␣.1234+024. A forma contendo a letra E não é usada se a magnitude do expoente exceder 99. Por exemplo, E10.4 irá imprimir o número 1.234×10^{-150} como ␣.1234-149. Alguns processadores põe um zero antes do ponto decimal. Note que agora o valor de <w> deve ser grande o suficiente para acomodar, além dos dígitos, o sinal (se for negativo), o ponto decimal, a letra E (caso possível), o sinal do expoente e o expoente. Assim, caso se tentasse imprimir o número anterior com o descritor F9.4, apareceria *********.

Na segunda forma do descritor, E<w>.<d>E<e>, o campo <e> determina o número de dígitos no expoente. Esta forma é obrigatória para números cujos expoentes têm valor absoluto maior que 999. Assim, o número 1.234×10^{1234}, com o descritor E12.4E4 é transferido como a string ␣.1234E+1235.

O descritor de edição EN implementa a *notação de engenharia*. Ele atua de forma semelhante ao descritor E, exceto que na saída o expoente decimal é múltiplo de 3, a parte inteira é maior ou igual a 1 e menor que 1000 e o fator de escala não tem efeito. Assim, o número 0.0217 transferido sob EN9.2 será convertido a 21.70E-03 ou 21.70-003.

O descritor de edição ES implementa a *notação científica*. Ele atua de forma semelhante ao descritor E, exceto que na saída o valor absoluto da parte inteira é maior ou igual a 1 e menor que 10 e o fator de escala não tem efeito. Assim, o número 0.0217 transferido sob ES9.2 será convertido a 2.17E-02 ou 2.17E-002.

NÚMEROS COMPLEXOS (F<w>.<d>, E<w>.<d>[E<e>], EN<w>.<d>[E<e>], ES<w>.<d>[E<e>], D<w>.<d>)

Números complexos podem ser editados sob o controle de pares dos mesmos descritores de números reais. Os dois descritores não necessitam ser idênticos. O valor complexo `(0.1,100.)`, convertido sob o controle de `F6.1,8.1` seria convertido a `␣␣␣0.1␣␣.1E+03`. Os dois descritores podem ser separados por uma constante de caracteres e descritores de controle de edição.

VALORES LÓGICOS (L<w>)

Valores lógicos podem ser editados usando o descritor `L<w>`. este define um campo de tamanho `<w>` o qual, na entrada consiste de brancos opcionais, opcionalmente seguidos por um ponto decimal, seguido por `T` ou `F`. Opcionalmente, este caractere pode ser seguido por caracteres adicionais. Assim, o descritor `L7` permite que as strings `.TRUE.` e `.FALSE.` sejam lidas com o seu significado correto.

Na saída, um dos caracteres `T` ou `F` irá aparecer na posição mais à direita do campo.

VARIÁVEIS DE CARACTERES A[<w>]

Valores de caracteres podem ser editados usando o descritor `A` com ou sem o campo `<w>`. Sem o campo, a largura do campo de entrada ou saída é determinado pelo tamanho real do ítem na lista de E/S, medido em termos do número de caracteres de qualquer espécie.

Por outro lado, usando-se o campo `<w>` pode-se determinar o tamanho desejado da string lida ou escrita. Por exemplo, dada a palavra `TEMPORARIO`, temos os seguintes resultados no processo de escrita:

Descritor	Registro escrito
A	TEMPORARIO
A11	␣TEMPORARIO
A8	TEMPORAR

Ou seja, quando o campo do descritor é menor que o tamanho real do registro, este é escrito com os caracteres mais à esquerda. Por outro lado, em caso de leitura da mesma string, as seguintes formas serão geradas:

Descritor	Registro lido
A	TEMPORARIO
A11	␣TEMPORARIO
A8	TEMPORAR␣␣

Todos os caracteres transferidos sob o controle de um descritor `A` ou `A<w>` têm a espécie do ítem na lista de E/S. Além disso, este descritor é o único que pode ser usado para transmitir outros tipos de caracteres, distintos dos caracteres padrões.

DESCRITOR GERAL G<w>.<d>[E<e>]

O descritor de edição geral pode ser usado para qualquer tipo intrínseco de dados. Quando usado para os tipos real ou complexo, a sua ação é idêntica à do descritor `E<w>.<d>[E<e>]`, exceto quando o formato `F` for considerado mais adequado.

A regra para decidir qual dos formatos `F` ou `E` será empregado é a seguinte: sejam N o número decimal obtido pelo arredondamento do valor interno do dado a `<d>` dígitos decimais significativos (de acordo com o modo de arredondamento para E/S em vigor) e s um inteiro positivo tal que $s = 1$ (se $N = 0$) ou senão tal que

$$10^{s-1} \leqslant N < 10^s.$$

Se $0 \leqslant s \leqslant$ `<d>`, então o formato

`F(<w> - `n`).(<d> - `s`)`

seguido por n espaços em branco será empregado, onde $n = 4$ para o formato G<w>.<d> e <e>+2 para o formato G<w>.<d>E<e>.

De acordo com esta regra, sob o controle do descritor G10.3, os números 0.01732, 1.732, 173.2 e 1732.0 serão impressos como 0.173E-01, 1.73, 173. e 0.173E04, respectivamente.

Esta forma é útil para escrever valores cujas magnitudes não são bem conhecidas previamente e quando a conversão do descritor F é preferível à do descritor E.

Quando o descritor G é usado para os tipos inteiro, lógico e de caracteres, ele segue as regras dos respectivos descritores de edição .

Para flexibilizar a saída no formato geral, particularmente para arquivos no formato CSV (*comma-separated values*), pode-se empregar o editor G0, o qual irá transferir os dados da seguinte maneira:

- Dados inteiros serão transferidos de acordo com o descritor I0.
- Dados reais ou complexos será transferidos de acordo com o descritor ES<w>.<d>E<e>, sendo que o compilador determina os valores de <w>, <d> e <e> de acordo com o valor a ser impresso.
- Dados lógicos são transferidos de acordo com o descritor L1.
- Dados de caracteres são transferidos de acordo com o descritor A.

Por exemplo, o comando

```
PRINT'(5(G0, :, "; "))', 17, 2.71828, .false., "Hello"
```

irá imprimir na tela algo semelhante a

```
17; 2.7183e+00; F; Hello
```

TIPOS DERIVADOS

Valores de tipos derivados são editados pela sequência apropriada de descritores de edição correspondentes aos tipos intrínsecos dos componentes do tipo derivado. Como exemplo,

```
TYPE :: STRING
   INTEGER            :: COMP
   CHARACTER(LEN= 20) :: PALAVRA
END TYPE STRING
TYPE(STRING) :: TEXTO
READ(*, '(I2,A)')TEXTO
```

DESCRITOR DE EDIÇÃO DE STRING DE CARACTERES

Uma constante de caracteres da espécie padrão, sem um parâmetro de espécie especificado, pode ser transferida a um arquivo de saída inserindo-a na especificação de formato, como no exemplo:

```
PRINT"('Este é um exemplo!')"
```

O qual vai gerar o registro de saída:

```
Este é um exemplo
```

cada vez que o comando seja executado. Descritores de edição de string de caracteres não podem ser usados em entrada de dados.

10.10.3 DESCRITORES DE CONTROLE DE EDIÇÃO

Um descritor de controle de edição age de duas formas: determinando como texto é organizado ou afetando as conversões realizadas por descritores de edição de dados subsequentes.

FORMATO DOS DESCRITORES DE CONTROLE DE EDIÇÃO

Um descritor de controle de edição pode assumir uma das seguintes formas em uma lista de formatação de E/S:

```
<c>
<c><n>
<n><c>
```

onde:

<c> é um dos seguintes códigos de formato: `T`, `TL`, `TR`, `X`, `S`, `SP`, `SS`, `BN`, `BZ`, `RU`, `RD`, `RZ`, `RN`, `RC`, `RP`, `P`, `DC`, `DP`, dois pontos (`:`) e a barra (`/`).

<n> é um número de posições de caracteres. Este número não pode ser uma variável; deve ser uma constante inteira positiva sem especificação de parâmetro de espécie de tipo. O intervalo de <n> inicia em 1 até um valor que depende do sistema. Em processadores Intel de 32 bits, por exemplo, o maior valor é <n>= `2**15 - 1`.

Em geral, descritores de controle de edição não são repetíveis. A única exceção é a barra (`/`), a qual pode também ser precedida por um contador de repetição.

Os descritores de controle possuem a seguintes formas específicas quanto às suas funções:

Tipo de controle	Descritores de controle
Posicional	`T<n>`, `TL<n>`, `TR<n>`, `<n>X`
Sinal	`S`, `SP`, `SS`
Interpretação de brancos	`BN`, `BZ`
Fator de escala	`<k>P`
Modo de arredondamento	`RU`, `RD`, `RZ`, `RN`, `RC`, `RP`
Ponto/vírgula decimal	`DC`, `DP`
Miscelâneo	`:`, `/`

O descritor `P` é uma exceção à sintaxe geral dos descritores. Ele é precedido por um fator de escala (`<k>`), em vez de um especificador de posição.

Descritores de controle de edição também podem ser agrupados em parênteses e precedidos por um contador de repetição do grupo.

EDIÇÃO POSICIONAL

Os descritores `T`, `TL`, `TR` e `X` especificam a posição onde o próximo caractere é transferido de ou para um registro.

Na saída, estes descritores não executam a conversão e transferência de caracteres propriamente dita, além de não afetarem o tamanho do registro. Se caracteres são transferidos a posições na posição especificada por um destes descritores, ou após esta, posições saltadas e que não eram previamente preenchidas por brancos passam a ser preenchidas. O resultado é como se o registro por completo estava inicialmente preenchido por brancos.

EDIÇÃO T

O descritor `T` especifica uma posição de caractere em um registro de E/S. Ele toma a seguinte forma:

```
T<n>
```

onde <n> é uma constante positiva que indica a posição de caractere do registro, relativa ao limite à esquerda do tabulador.

Na entrada, o descritor posiciona o registro externo a ser lido na posição de caractere especificada por <n>. Na saída, o descritor indica que a transferência de dados inicia na <n>-ésima posição de caractere do registro externo.

EXEMPLOS. Suponha que um arquivo possua um registro contendo o valor ABC␣␣␣XYZe o seguinte comando de leitura seja executado:

```
CHARACTER(LEN= 3) :: VALOR1, VALOR2
...
READ(11,'(T7,A3,T1,A3)')VALOR1, VALOR2
```

Os valores lidos serão VALOR1= 'XYZ' e VALOR2= 'ABC'.

Suponha agora que o seguinte comando seja executado:

```
PRINT 25
25 FORMAT(T51,'COLUNA 2',T21,'COLUNA 1')
```

a seguinte linha é impressa na tela do monitor (ou na saída padrão):

```
00000000011111111112222222222333333333344444444445555555556666666666777777777 8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
                    |                              |
                    COLUNA 1                       COLUNA 2
```

Deve-se notar que as constantes de caractere foram impressas iniciando nas colunas 20 e 50 e não nas colunas 21 e 51. Isto ocorreu porque o primeiro caractere do registro impresso foi reservado como um caractere de controle. Este é o comportamento padrão na saída de dados.

EDIÇÃO TL

O descritor TL especifica uma posição de caractere *à esquerda* da posição corrente no registro de E/S. O descritor toma a seguinte forma:

```
TL<n>
```

onde <n> é uma constante inteira positiva indicando a <n>-ésima posição à esquerda do caractere corrente. Se <n> é maior ou igual à posição corrente, o próximo caractere acessado é o primeiro caractere do registro.

EXEMPLO. No exemplo anterior, temos:

```
PRINT 25
25 FORMAT(T51,'COLUNA 2',T21,'COLUNA 1',TL20,'COLUNA 3')
```

o que gera na tela:

```
00000000011111111112222222222333333333344444444445555555556666666666777777777 8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
       |           |                              |
       COLUNA 3    COLUNA 1                       COLUNA 2
```

EDIÇÃO TR

O descritor TR especifica uma posição de caractere *à direita* da posição corrente no registro de E/S. O descritor toma a seguinte forma:

```
TR<n>
```

onde <n> é uma constante inteira positiva indicando a <n>-ésima posição à direita do caractere corrente.

EDIÇÃO X

O descritor X especifica uma posição de caractere à direita da posição corrente no registro de E/S. Este descritor é equivalente ao descritor TR. O descritor X toma a seguinte forma:

```
<n>X
```

onde <n> é uma constante inteira positiva indicando a <n>-ésima posição à direita do caractere corrente. Na saída, este descritor não gera a gravação de nenhum caractere quando ele se encontra no final de uma especificação de formato.

EDIÇÃO DE SINAL

Os descritores S, SP e SS controlam a saída do sinal de mais (+) opcional dentro de campos de saída numéricos. Estes descritores não têm efeito durante a execução de comandos de entrada de dados.

Dentro de uma especificação de formato, um descritor de edição de sinal afeta todos os descritores de dados subsequentes I, F, E, EN, ES, D e G até que outro descritor de edição de sinal seja incluído na formatação.

EDIÇÃO SP

O descritor SP força o processador a *produzir* um sinal de mais em qualquer posição subsequente onde ele seria opcional. O descritor toma a forma simples

```
SP
```

EDIÇÃO SS

O descritor SS força o processador a *suprimir* um sinal de mais em qualquer posição subsequente onde ele seria opcional. O descritor toma a forma simples

```
SS
```

EDIÇÃO S

O descritor S retorna o status do sinal de mais como opcional para todos os campos numéricos subsequentes. O descritor toma a forma simples

```
S
```

INTERPRETAÇÃO DE BRANCOS

Os descritores BN e BZ controlam a interpretação de brancos posteriores embebidos dentro de campos de entrada numéricos. Estes descritores não têm efeito durante a execução de comandos de saída de dados.

Dentro de uma especificação de formato, um descritor de edição de brancos afeta todos os descritores de dados subsequentes I, B, O, Z, F, E, EN, ES, D e G até que outro descritor de edição de brancos seja incluído na formatação.

Os descritores de edição de brancos sobrepõe-se ao efeito do especificador BLANK durante a execução de um comando de entrada de dados.

EDIÇÃO BN

O descritor BN força o processador a *ignorar* todos os brancos posteriores embebidos em campos de entrada numéricos. O descritor toma a forma simples:

```
BN
```

O campo de entrada é tratado como se todos os brancos tivessem sido removidos e o restante do campo é escrito na posição mais à direita. Um campo todo composto de brancos é tratado como zero.

EDIÇÃO BZ

O descritor BZ força o processador a *interpretar* todos os brancos posteriores embebidos em campos de entrada numéricos como zeros. O descritor toma a forma simples:

```
BN
```

EDIÇÃO DE FATOR DE ESCALA P

O descritor P especifica um fator de escala, o qual move a posição do ponto decimal em valores reais e das duas partes de valores complexos. O descritor toma a forma

```
<k>P
```

onde <k> é uma constante inteira com sinal (o sinal é opcional, se positiva) especificando o número de posições, para a esquerda ou para a direita, que o ponto decimal deve se mover (o fator de escala). O intervalo de valores de <k> é de -128 a 127.

No início de um comando de E/S formatado, o valor do fator de escala é zero. Se um descritor de escala é especificado, o fator de escala é fixado, o qual afeta todos os descritores de edição de dados reais subsequentes até que outro descritor de escala ocorra. Para redefinir a escala a zero, deve-se obrigatoriamente especificar 0P.

Na entrada, um fator de escala positivo move o ponto decimal para a esquerda e um fator negativo move o ponto decimal para direita. Quando um campo de entrada, usando os descritores F, E, D, EN, ES ou G, contém um expoente explícito, o fator de escala não tem efeito. Nos outros casos, o valor interno do dado lido na lista de E/S é igual ao campo externo multiplicado por $10^{-<k>}$. Por exemplo, um fator de escala 2P multiplica um valor de entrada por 0.01, movendo o ponto decimal duas posições *à esquerda*. Um fator de escala -2P multiplica o valor de entrada por 100, movendo o ponto decimal duas posições *à direita.*

A seguinte tabela apresenta alguns exemplos do uso do especificador <k>P na entrada de dados:

Formato	Campo de entrada	Valor interno
3PE10.5	␣␣37.614␣␣	.037614
3PE10.5	␣␣37.614E2	3761.4
-3PE10.5	␣␣␣37.614␣	37614.0

O fator de escala deve preceder o primeiro descritor de edição real associado com ele, mas não necessariamente deve preceder o descritor propriamente dito. Por exemplo, todos os seguintes formatos têm o mesmo efeito:

```
(3P,I6,F6.3,E8.1)
(I6,3P,F6.3,E8.1)
(I6,3PF6.3,E8.1)
```

Note que se o fator de escala precede imediatamente o descritor real associado, a vírgula é opcional.

Na saída, um fator de escala positivo move o ponto decimal para a direita e um fator de escala negativo move o ponto decimal para a esquerda. Neste caso, o efeito do fator de escala depende em que tipo de edição real está associada com o mesmo, como segue:

- Para edição F, o valor externo iguala o valor interno da lista de E/S multiplicado por $10^{<k>}$, alterando a magnitude do dado.
- Para edições E e D, o campo decimal externo da lista de E/S é multiplicado por $10^{<k>}$ e <k> é subtraído do expoente, alterando a magnitude do dado. Um fator de escala positivo diminui o expoente; um fator de escala negativo aumenta o expoente. Para fator de escala positivo, <k> deve ser menor que <d>+2 senão ocorrerá um erro no processo de conversão de saída.
- Para edição G, o fator de escala não tem efeito se a magnitude do dado a ser impresso está dentro do intervalo efetivo do descritor. Se a magnitude estiver fora do intervalo efetivo do descritor, edição E é usada e o fator de escala tem o mesmo efeito como neste caso.
- Para edições EN e ES, o fator de escala não tem efeito.

A seguir, alguns exemplos de saída de dados usando o descritor P:

Formato	Campo de entrada	Valor interno
1PE12.3	-270.139	␣␣-2.701E+02
1PE12.2	-270.139	␣␣␣-2.70E+02
-1PE12.2	-270.139	␣␣␣-0.03E+04

O exemplo a seguir também usa edição P:

```
REAL, DIMENSION(6) :: A= 25.0
WRITE(6,10) A
10 FORMAT(' ', F8.2,2PF8.2,F8.2)
```

resultando os seguintes valores gravados na unidade 6:

```
  25.00  2500.00  2500.00
2500.00  2500.00  2500.00
```

Modo de arredondamento em E/S

O modo de arredondamento dos dados numéricos durante E/S formatadas é globalmente determinado para uma unidade lógica pelo comando OPEN. Entretanto, o modo de arredondamento pode ser temporariamente alterado por comandos READ ou WRITE via os descritores RU, RD, RZ, RN, RC ou RP, os quais determinam, respectivamente, os modos UP, DOWN, ZERO, NEAREST, COMPATIBLE e PROCESSOR_DEFINED. As descrições destes modos de arredondamento são realizadas para o qualificador ROUND= do comando OPEN, na seção 10.6.

Determinação de ponto/vírgula decimal

Por padrão, o símbolo que separa a parte inteira da parte fracionária de um número real é o ponto decimal ".". Contudo, em certas situações, é necessário empregar-se a vírgura "," para realizar essa notação. Esta tarefa também é determinada globalmente pelo comando OPEN, mas a notação pode ser temporariamente alterada por comandos READ ou WRITE via os descritores DC ou DP, os quais determinam, respectivamente, COMMA (vírgula) ou POINT (ponto). A descrição da notação é realizada para o qualificador DECIMAL= do comando OPEN, na seção 10.6.

Edição com barra (/)

O descritor / termina a transferência de dados para o registro corrente e inicia a transferência de dados para um registro novo. Em um arquivo ou na tela, o descritor tem a ação de iniciar uma nova linha.

A forma que o descritor toma é:

```
[<r>]/
```

onde <r> é um contador de repetição. O intervalo de <r> inicia em 1 até um valor que depende do sistema. Em processadores intel de 32 bits, o valor máximo é <r>= 2**15-1.

Múltiplas barras forçam o sistema a pular registros de entrada ou a gerar registros brancos na saída, como segue:

- Quando n barras consecutivas aparecem entre dois descritores de edição, n - 1 registros são pulados na entrada ou n - 1 registros brancos são gerados na saída. A primeira barra termina o registro corrente. A segunda barra termina o primeiro pulo ou registro em branco e assim por diante.

- Quando n barras consecutivas aparecem no início ou final de uma especificação de formato, n registros são pulados ou n registros em branco são gerados na saída, porque o parênteses inicial e final da especificação de formato são, por si mesmos, iniciadores ou finalizadores de registros, respectivamente

Por exemplo, dado a seguinte formatação de saída:

```
WRITE(6,99)
99 FORMAT('1',T51,'Linha Cabeçalho'//T51,'Sublinha Cabeçalho'//)
```

irá gerar os seguintes registros no arquivo:

```
0000000001111111111222222222233333333334444444444555555555566666666667777777778
1234567890123456789012345678901234567890123456789012345678901234567890123456789
                                                  |
                                                  Linha Cabeçalho
<linha em branco>
                                                  Sublinha Cabeçalho
<linha em branco>
<linha em branco>
```

EDIÇÃO COM DOIS PONTOS (:)

O descritor ":" termina o controle de formato se não houver mais ítens na lista de E/S. Em particular, o descritor evita a impressão de constantes de caracteres adicionais caso não haja mais dados numéricos a serem impressos. Se houver ítens de E/S restantes, o descritor ":" não tem efeito.

Por exemplo, sendo `L(3)` um vetor inteiro, o comando

```
PRINT'(" l1= ", I5, :, " l2= ", I5, :, " l3= ", I5)', (L(I), I=1, N)
```

irá imprimir os três elementos do vetor caso `N = 3`, mas, se `N = 1` e `L(1) = 59`, o comando irá imprimir somente

```
l1= 59
```

10.10.4 DESCRITORES DE EDIÇÃO DE STRINGS

Descritores de edição de strings controlam a saída das constantes de string. Estes descritores são:

- Constantes de caracteres
- Descritor de edição `H` (eliminado no Fortran 95).

Um descritor de edição de strings não pode ser precedido por um contador de repetição; contudo, eles podem fazer parte de um grupo contido entre parênteses, o qual, por sua vez, é precedido por um contador de repetição.

EDIÇÃO DE CONSTANTES DE CARACTERES

O descritor de constante de caracteres provoca a saída de uma constante de string em um registro externo. Ele vem em duas formas:

```
'<string>'
"<string>"
```

onde `<string>` é uma constante de caracteres, sem especificação de espécie. O seu comprimento é o número de caracteres entre os delimitadores; dois delimitadores consecutivos são contados como um caractere.

Para incluir um apóstrofe (') em uma constante de caractere que é delimitada por apóstrofes, deve-se colocar duas apóstrofes consecutivas ('') na formatação. Por exemplo,

```
50 FORMAT('Today''s date is:  ',i2,'/',i2,'/',i2)
```

Da mesma forma, para incluir aspas (") em uma constante de caractere que é delimitada por aspas, deve-se colocar duas aspas consecutivas na formatação.

Como exemplo do uso dos comandos de E/S apresentados nas seções 10.6 – 10.10, o programa a seguir abre o arquivo pessoal.dat, o qual contém registros dos nomes de pessoas (até 15 caracteres), idade (inteiro de 3 dígitos), altura (em metros e centímetros) e número de telefone (inteiro de 8 dígitos).

```
program pessoais_dados
      implicit none
      character(len= 15) :: nome
      integer :: age, tel_no, stat
      real :: height
      open(unit= 8, file= 'pessoal.dat')
      write(*, 100)
  100 format(t24, 'Altura')
      write(*, 200)
  200 format(t4, 'Nome', t17, 'Idade', t23, '(metros)', t32, 'Tel. No.')
      write(*, 300)
  300 format(t4, '----', t17, '-----', t23, ' ------ ', t32, '--------')
      do
         read(8, fmt='(a15, 1x, i3, 1x, f4.2, 1x, i8)', iostat= stat) nome, &
                                                      age, height, tel_no
      if (stat < 0) exit                        ! testa final de arquivo
      write(*, 400) nome, age, height, tel_no
  400 format(a, t18, i3, t25, f4.2, t32, i8)
      end do
end program pessoais_dados
```

O programa lê estes dados contidos em pessoal.dat e os imprime no monitor no formato:

```
                          Altura
    Nome       Idade    (metros)   Tel.  No.
    ----       -----    ------     --------
P. A. Silva     45       1.80      33233454
J. C. Pedra     47       1.75      34931458
```

sendo que o arquivo pessoal.dat possui a seguinte formatação:

```
P. A. Silva     045 1.80 33233454
J. C. Pedra     047 1.75 34931458
```

Cabe ressaltar também que os comandos FORMAT utilizados nos rótulos 100, 200, 300 e 400 poderiam ser igualmente substituídos pelo especificador FMT=. Por exemplo, as linhas

```
      WRITE(*,200)
  200 FORMAT(T4,'Nome',T17,'Idade',T23,'(metros)',T32,'Tel. No.')
```

são equivalentes a

```
      WRITE(*, FMT='(T4,"Nome",T17,"Idade",T23,"(metros)",T32,"Tel.  No.")')
```

10.11 Comando CLOSE

O propósito do comando CLOSE é desconectar um arquivo de uma unidade. Sua forma é:

```
CLOSE([UNIT=]<u>[, IOSTAT=<ios>][, ERR= <err-label>][, STATUS=<status>])
```

onde <u>, <ios> e <err-label> têm os mesmos significados descritos para o comando OPEN (seção 10.6). Novamente, especificadores em palavras-chaves podem aparecer em qualquer ordem, mas o especificador de unidade deve ser o primeiro ser for usada a forma posicional.

A função do especificador STATUS= é de determinar o que acontecerá com o arquivo uma vez desconectado. O valor de <status>, a qual é uma expressão escalar de caracteres da espécie padrão, pode ser um dos valores 'KEEP' ou 'DELETE', ignorando qualquer branco posterior. Se o

valor é 'KEEP', um arquivo que existe continuará existindo após a execução do comando CLOSE, e pode ser posteriormente conectado novamente a uma unidade. Se o valor é 'DELETE', o arquivo não mais existirá após a execução do comando. Se o especificador for omitido, o valor padrão é 'KEEP', exceto se o arquivo tem o status 'SCRATCH', em cujo caso o valor padrão é 'DELETE'.

Em qualquer caso, a unidade fica livre para ser conectada novamente a um arquivo. O comando CLOSE pode aparecer em qualquer ponto no programa e se é executado para uma unidade não existente ou desconectada, nada acontece.

No final da execução de um programa, todas as unidades conectadas são fechadas, como se um comando CLOSE sem o qualificador STATUS= fosse aplicado a cada unidade conectada.

Como exemplo:

```
CLOSE(2, IOSTAT=IOS, ERR=99, STATUS='DELETE')
```

10.12 COMANDO INQUIRE

O status de um arquivo pode ser definido pelo sistema operacional antes da execução do programa ou pelo programa durante a sua execução, seja por um comando OPEN ou seja por alguma ação sobre um arquivo pré-conectado que o faz existir. Em qualquer momento, durante a execução do programa, é possível inquirir a respeito do status e atributos de um arquivo usando o comando INQUIRE.

Usando uma variante deste comando, é possível determinar-se o status de uma unidade; por exemplo, se o número de unidade existe para o sistema em questão, se o número está conectado a um arquivo e, em caso afirmativo, quais os atributos do arquivo. Outra variante do comando permite inquirir-se a respeito do tamanho de uma lista de saída quando usada para escrever um registro não formatado.

Alguns atributos que podem ser determinados pelo uso do comando INQUIRE são dependentes de outros. Por exemplo, se um arquivo não está conectado a uma unidade, não faz sentido inquirir-se a respeito da forma de acesso que está sendo usada para este arquivo. Se esta inquirição é feita, de qualquer forma, o especificador relevante fica indefinido.

As três variantes do comando são conhecidas como INQUIRE por arquivo, INQUIRE por unidade e INQUIRE por lista de saída. Na descrição feita a seguir, as primeiras duas variantes são descritas juntas. Suas formas são:

```
INQUIRE({[UNIT=]<u>|FILE=<arq>}, <lista-inquire>)
```

onde os especificadores entre chaves ({}) são excludentes; o primeiro deve ser usado no caso de INQUIRE por unidade e o segundo no caso de INQUIRE por arquivo. Neste último caso, <arq> é uma expressão escalar de caracteres cujo valor, ignorando quaisquer brancos posteriores, fornece o nome do arquivo envolvido, incluindo o caminho. A interpretação de <arq> depende do sistema. Em um sistema Unix/Linux, o nome depende do caso.

Também no comando, <lista-inquire> é uma lista de especificadores opcionais, os quais serão descritos abaixo. Um especificador não pode aparecer mais de uma vez na lista de especificadores opcionais. Todas as atribuições de valores aos especificadores seguem as regras usuais e todos os valores do tipo de caracteres, exceto no caso de NAME= são maiúsculos. Os especificadores, cujos valores são todos escalares e da espécie padrão são:

ACCESS= <acc>, onde <acc> é uma variável de caracteres à qual são atribuídos um dos valores 'SEQUENTIAL', 'DIRECT' ou 'STREAM', dependendo do método de acesso para um arquivo que está conectado e 'UNDEFINED' se não houver conexão.

ACTION= <act>, onde <act> é uma variável de caracteres à qual são atribuídos os valores 'READ', 'WRITE' ou 'READWRITE', conforme a conexão. Se não houver conexão, o valor é 'UNDEFINED'.

ASYNCHRONOUS= <asy>, onde <asy> é uma variável de caracteres à qual será atribuído o valor 'YES' se o arquivo está conectado e E/S assíncrona na unidade é possível; ou será atribuído o valor 'NO' se o arquivo está conectado mas E/S assíncrona não é permitida. Se não há conexão, <asy> recebe o valor 'UNDEFINED'.

BLANK= <bl>, onde <bl> é uma variável de caracteres à qual são atribuídos os valores 'NULL' ou 'ZERO', dependendo se os brancos em campos numéricos devem ser por padrão interpretados como campos de nulos ou de zeros, respectivamente, ou 'UNDEFINED' se ou não houver conexão ou se a conexão não for para E/S formatada.

DECIMAL= <dec>, onde <dec> é uma variável de caracteres à qual é atribuído o valor 'COMMA' ou 'POINT', correspondendo ao modo de edição decimal em efeito para uma conexão com E/S formatada. Se não há conexão, ou a conexão não é para E/S formatada, <dec> recebe o valor 'UNDEFINED'.

DELIM= <del>, onde <del> é uma variável de caracteres à qual são atribuídos os valores 'QUOTE', 'APOSTROPHE' ou 'NONE', conforme especificado pelo correspondente comando OPEN (ou pelo valor padrão). Se não houver conexão, ou se o arquivo não estiver conectado para E/S formatada, o valor é 'UNDEFINED'.

DIRECT= <dir>, SEQUENTIAL= <seq>, STREAM= <str>, onde <dir>, <seq> e <str> são variáveis de caracteres às quais são atribuídos os valores 'YES', 'NO' ou 'UNKNOWN', dependendo se o arquivo *puder* ser aberto para acesso direto, sequencial ou stream, respectivamente, ou se isto não pode ser determinado.

ENCODING= <enc>, onde <enc> é uma variável de caracteres à qual é atribuído o valor 'UTF-8' ou 'DEFAULT', correspondendo à encodificação (*encoding*) em uso.

ERR= <err-label>, tem o mesmo significado descrito no comando OPEN (seção 10.6).

EXIST= <ex>, onde <ex> é uma variável lógica. O valor .TRUE. é atribuído se o arquivo (ou unidade) existir e .FALSE. em caso contrário.

FORM= <frm>, onde <frm> é uma variável de caracteres à qual são atribuídos um dos valores 'FORMATTED' ou 'UNFORMATTED', dependendo na forma para a qual o arquivo é realmente conectado, ou 'UNDEFINED' se não houver conexão.

FORMATTED= <fmt>, UNFORMATTED= <unf>, onde <fmt> e <unf> são variáveis de caracteres às quais são atribuídos os valores 'YES', 'NO' ou 'UNKNOWN', dependendo se o arquivo *puder* ser aberto para acesso formatado ou não formatado, respectivamente, ou se isto não pode ser determinado.

ID= <id>, onde <id> é a variável inteira que identifica uma operação assíncrona de E/S em particular (ver PENDING=).

IOMSG= <msg>, onde <msg> é uma variável escalar de caracteres da espécie padrão na qual o processador coloca uma mensagem se uma condição de erro, final de arquivo ou final de registro ocorrer durante a execução do comando.

IOSTAT= <ios>, tem o mesmo significado descrito no comando OPEN (seção 10.6). A variável <ios> é a única definida se uma condição de erro ocorre durante a execução do INQUIRE.

NAMED= <nmd>, NAME= <nam>, onde <nmd> é uma variável lógica à qual o valor .TRUE. é atribuído se o arquivo tem um nome ou .FALSE. em caso contrário. Se o arquivo tem um nome, este será atribuído à variável de caracteres <nam>. Este valor não é necessariamente o mesmo que é dado no especificador FILE=, se usado, mas pode ser qualificado de alguma forma. Contudo, em qualquer situação é um nome válido para uso em um comando OPEN subsequente. Assim, o INQUIRE pode ser usado para determinar o valor real de um arquivo antes deste ser conectado. Dependendo do sistema, o nome depende do caso.

NEXTREC= <nr>, onde <nr> é uma variável inteira à qual é atribuído o valor do número do último registro lido ou escrito, mais um. Se nenhum registro foi ainda lido ou escrito, <nr>= 1. Se o arquivo não estiver conectado para acesso direto ou se a posição é indeterminada devido a um erro anterior, <nr> resulta indefinido.

NUMBER= <num>, onde <num> é uma variável inteira à qual é atribuído o número da unidade conectada ao arquivo, ou -1 se nenhuma unidade estiver conectada ao arquivo.

OPENED= <open>, onde <open> é uma variável lógica. O valor .TRUE. é atribuído se o arquivo (ou unidade) estiver conectado a uma unidade (ou arquivo) e .FALSE. em caso contrário.

PAD= <pad>, onde <pad> é uma variável de caracteres à qual são atribuídos os valores 'YES', caso assim especificado pelo correspondente comando OPEN (ou pelo valor padrão); em caso contrário, o valor é 'NO'.

PENDING= <pnd>, onde <pnd> é uma variável lógica. Se o especificador ID= estiver presente, <pnd> recebe o valor .TRUE. se aquela operação de E/S em particular está ainda pendente, ou .FALSE. em caso contrário.

POS= <pos>, onde <pos> é uma variável inteira escalar que recebe a posição corrente em um arquivo de acesso de stream.

POSITION= <pos>, onde <pos> é uma variável de caracteres à qual são atribuídos os valores 'REWIND', 'APPEND' ou 'ASIS', conforme especificado no correspondente comando OPEN, se o arquivo não foi reposicionado desde que foi aberto. Se não houver conexão ou se o arquivo está conectado para acesso direto, o valor é 'UNDEFINED'. Se o arquivo foi reposicionado desde que a conexão foi estabelecida, o valor depende do processador (mas não deve ser 'REWIND' ou 'APPEND' exceto se estes corresponderem à verdadeira posição).

READ= <rd>, onde <rd> é uma variável de caracteres à qual são atribuídos os valores 'YES', 'NO' ou 'UNKNOWN', dependendo se leitura é permitida, não permitida ou indeterminada para o arquivo.

READWRITE= <rw>, onde <rw> é uma variável de caracteres à qual são atribuídos os valores 'YES', 'NO' ou 'UNKNOWN', dependendo se leitura e escrita são permitidas, não permitidas ou indeterminadas para o arquivo.

RECL= <rec>, onde <rec> é uma variável inteira à qual é atribuído o valor do tamanho do registro de um arquivo conectado para acesso direto, ou o tamanho máximo do registro permitido para um arquivo conectado para acesso sequencial. O comprimento é o número de caracteres para registros formatados contendo somente caracteres do tipo padrão e dependente do sistema em caso contrário. Se não houver conexão, <rec> resulta indefinido.

ROUND= <rnd>, onde <rnd> é uma variável de caracteres que recebe os valores 'UP', 'DOWN', 'ZERO', 'NEAREST', 'COMPATIBLE' ou 'PROCESSOR_DEFINED', dependendo do modo de arredondamento em vigor (seção 10.6). Se não há conexão, ou se o arquivo não está conectado para E/S formatada, <rnd> recebe o valor 'UNDEFINED'.

SIGN= <sgn>, onde <sgn> é uma variável de caracteres que recebe os valores 'PLUS', 'SUPPRESS' ou 'PROCESSOR_DEFINED', dependendo do modo de sinal em vigor (seção 10.6). Se não há conexão, ou se o arquivo não está conectado para E/S formatada, <sgn> recebe o valor 'UNDEFINED'.

SIZE= <siz>, onde <siz> é uma variável inteira à qual é atribuído o tamanho do arquivo em unidades de armazenagem de arquivos. Se o tamanho não puder ser deteriminado, é atribuído <siz> = -1.

WRITE= <wr>, onde <wr> é uma variável de caracteres à qual são atribuídos os valores 'YES', 'NO' ou 'UNKNOWN', dependendo se escrita é permitida, não permitida ou indeterminada para o arquivo.

Uma variável que é a especificação em um comando INQUIRE, ou está associada com um, não deve aparecer em outra especificador no mesmo comando.

A terceira variante do comando INQUIRE, INQUIRE por lista de E/S, tem a forma:

```
INQUIRE(IOLENGTH=<comp>) <lista-saída>
```

onde <comp> é uma variável inteira escalar padrão que é usada para determinar o comprimento de uma <lista-saída> não formatada em unidades que dependem do processador. Esta variável pode ser usada para estabelecer se, por exemplo, uma lista de saída é muito grande para o tamanho do registro dado pelo especificador RECL= de um comando OPEN, ou ser usado como o valor do comprimento a ser determinado ao especificador RECL=.

Um exemplo de uso do comando INQUIRE é dado abaixo:

```
LOGICAL            :: EX, OP
CHARACTER(LEN= 11) :: NAM, ACC, SEQ, FRM
INTEGER            :: IREC, NR
...
OPEN(2, IOSTAT= IOS, ERR= 99, FILE= 'Cidades', STATUS= 'NEW', &
```

```
     ACCESS= 'DIRECT', RECL= 100)
INQUIRE(2, ERR= 99, EXIST= EX, OPENED= OP, NAME= NAM, ACCESS= ACC, &
        SEQUENTIAL= SEQ, FORM= FRM, RECL= IREC, NEXTREC= NR)
```

Após execução bem sucedida dos comandos OPEN e INQUIRE, as variáveis terão os seguintes valores:

```
EX= .TRUE.
OP= .TRUE.
NAM= 'Cidades␣␣␣␣'
ACC= 'DIRECT␣␣␣␣␣'
SEQ= 'NO␣␣␣␣␣␣␣␣'
FRM= 'UNFORMATTED'
IREC= 100
NR= 1
```

10.13 Outros comandos que operam sobre arquivos

Outros comandos que exercem funções de controle no arquivo, em adição à entrada e saída de dados, são fornecidos abaixo.

10.13.1 Comando BACKSPACE

Pode acontecer em um programa que uma série de registros sejam escritos e que, por alguma razão, o último registro escrito deve ser substituído por um novo; isto é, o último registro deve ser sobrescrito. De forma similar, durante a leitura dos registros, pode ser necessário reler o último registro, ou verificar por leitura o último registro escrito. Para estes propósitos, Fortran fornece o comando BACKSPACE, o qual tem a sintaxe

```
BACKSPACE([UNIT=]<u>[, IOSTAT=<ios>][, ERR=<err-label>])
```

onde <u> é uma expressão inteira escalar padrão que designa o número da unidade e os outros especificadores opcionais têm o mesmo significado que no comando READ (seção 10.7).

A ação deste comando é posicionar o arquivo antes do registro corrente, se houver. Se o comando for tentado quando o registro estiver no início do arquivo, nada ocorre. Se o arquivo estiver posicionado após um registro de final de arquivo, este resulta posicionado antes deste registro.

Não é possível solicitar BACKSPACE em um arquivo que não exista, nem sobre um registro escrito por um comando NAMELIST. Uma serie de comandos BACKSPACE resultará no retrocesso no número correspondente de registros.

10.13.2 Comando REWIND

De forma semelhante a uma releitura, re-escritura ou verificação por leitura de um registro, uma operação similar pode ser realizada sobre um arquivo completo. Com este propósito, o comando REWIND:

```
REWIND([UNIT=]<u>[, IOSTAT=<ios>][, ERR=<err-label>])
```

pode ser usado para reposicionar um arquivo, cujo número de unidade é especificado pela expressão escalar inteira <u>. Novamente, os demais especificadores opcionais têm o mesmo significado que no comando READ.

Se o arquivo já estiver no seu início, nada ocorre. O mesmo ocorre caso o arquivo não exista.

10.13.3 COMANDO ENDFILE

O final de um arquivo conectado para acesso sequencial é normalmente marcado por um registro especial, denominado *registro de final de arquivo*, o qual é assim identificado pelo computador.

Quando necessário, é possível escrever-se um registro de final de arquivo explicitamente, usando o comando ENDFILE:

```
ENDFILE([UNIT=]<u>[, IOSTAT=<ios>][, ERR=<err-label>])
```

onde todos os argumentos têm o mesmo significado que nos comandos anteriores.

O arquivo é então posicionado após o registro de final de arquivo. Este registro, se lido subsequentemente por um programa, deve ser manipulado usado a especificação END=<end-label> do comando READ senão a execução do programa irá normalmente terminar.

Antes da transferência de dados, um arquivo não pode estar posicionado após o registro de final de arquivo, mas é possível retroceder com os comandos BACKSPACE ou REWIND, o que permite a ocorrência de outras transferências de dados.

Um registro de final de arquivo é automaticamente escrito sempre quando ou uma operação de retrocesso parcial ou completo segue um operação de escrita como a operação seguinte na unidade; quando o arquivo é fechado por um comando CLOSE, por um novo comando OPEN na mesma unidade ou por encerramento normal do programa.

Se o arquivo também pode ser conectado para acesso direto, somente os registros além do registro de final de arquivo são considerados como escritos e somente estes podem ser lidos em uma conexão de acesso direto subsequente.

10.13.4 COMANDO FLUSH

A execução de um comando

```
FLUSH(<unit>)
```

sobre um arquivo externo identificado com o número de unidade lógica <unit> faz com que qualquer dado pendente em uma área virtual de memória seja gravado no arquivo, para que outros processos tenham acesso a esses dados. Se os dados no arquivo externo foram gravados por outros processos, o comando torna os mesmos acessíveis ao comando READ.

Em combinação com ADVANCE= 'NO' ou acesso de stream, o comando permite ao programa assegurar-se que dados escritos na unidade <unit> são gravados no arquivo antes de solicitar dados de entrada de outra unidade.

Referências Bibliográficas

[1] M. Metcalf, J. Reid, and M. Cohen. *Modern Fortran Explained: Incorporating Fortran 2018*. Numerical Mathematics and Scientific Computation. OUP Oxford, Oxford, 2018. 522 + xxii pp. doi:10.1093/oso/9780198811893.001.0001.

[2] S. J. Chapman. *Fortran for Scientists and Engineers*. McGraw-Hill, New York, 2017. 1024 + xxiv pp. URL: https://books.google.com.br/books?id=OQhBMQAACAAJ.

[3] Ian D. Chivers and Jane Sleightholme. Compiler Support for the Fortran 2003, 2008, TS29113, and 2018 Standards Revision 26. *SIGPLAN Fortran Forum*, 38(2):7–36, July 2019. doi:10.1145/3345502.3345505.

[4] P. Prinz and T. Crawford. *C in a Nutshell: The Definitive Reference*. O'Reilly Media, Boston, 2nd edition, 2015. URL: https://books.google.com.br/books?id=FgMsCwAAQBAJ.

[5] N. S. Clerman and W. Spector. *Modern Fortran: Style and Usage*. Cambridge University Press, New York, 2011. URL: https://books.google.com.br/books?id=5Qj2DieTHsYC.

[6] Intel® Fortran Compiler 19.0 Developer Guide and Reference. Acesso em setembro de 2019. URL: https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-language-reference.

[7] Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education, New York, 4th. edition, 2013. 1346 + xiv pp. URL: https://books.google.com.br/books?id=PSUNAAAAQBAJ.