UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE MATEMÁTICA
CADERNOS DE MATEMÁTICA E ESTATÍSTICA
SÉRIE B: TRABALHO DE APOIO DIDÁTICO

INTRODUÇÃO À PROGRAMAÇÃO ESTRUTURADA EM FORTRAN 90

RUDNEI DIAS DA CUNHA

SÉRIE B, N° 68
PORTO ALEGRE, ABRIL DE 2003

Rudnei Dias da Cunha é professor adjunto da UFRGS, desempenhando suas atividades junto ao Departamento de Matemática Pura e Aplicada, do Instituto de Matemática, desde 1994. É Bacharel em Ciências de Computação pela UFRGS (1988) e *Doctor of Philosophy in Computer Science* pela *University of Kent at Canterbury*, Reino Unido (1992). Exerceu as funções de programador de computadores e analista de sistemas no Centro de Processamento de Dados da UFRGS (1983-1994). Foi coordenador do Programa de Pós-Graduação em Matemática Aplicada da UFRGS (1999-2000) e atualmente ocupa o cargo de Vice-Diretor do Instituto de Matemática da UFRGS.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE MATEMÁTICA
DEPARTAMENTO DE MATEMÁTICA PURA E APLICADA

Introdução à Programação Estruturada com Fortran 90

Rudnei Dias da Cunha

Porto Alegre, março de 2003.

Sumário

Capítulo 1

Introdução

O surgimento dos computadores de uso geral, a partir de 1950, trouxe consigo a necessidade de se utilizá-los de forma mais eficiente. A dificuldade existente na programação de computadores até então, usando-se diretamente códigos binários – números escritos com os algarismos 0 e 1 – era um dos principais entraves na utilização dos computadores.

Com isso, surgiu o conceito de linguagens de programação de alto nível. A primeira a surgir foi a linguagem FORTRAN. Um grupo de pesquisadores da IBM, liderado por John Backus [3], desenvolveu a linguagem entre 1954 e 1957. O resultado foi uma linguagem que atendia às necessidades daqueles usuários que precisavam resolver problemas de cunho científico, com uma significativa parcela de cálculo envolvida. O próprio nome da linguagem já exibia essa característica: FORTRAN é uma corruptela de *"FORmula TRANslation"*.

Em 1966, quando a primeira versão padronizada da linguagem estava disponível, FORTRAN já era a linguagem de escolha da comunidade científica. Dentre as principais características que a linguagem exibia, destacam-se, como citado em [3]: ampla disponibilidade de compiladores para diferentes computadores; facilidade para ensiná-la; independência do computador; eficiência em muitas implementações; e demonstrava as vantagens do uso de subrotinas e de compilação independente. Além dessas, destacamos a simplicidade da linguagem, que colaborou para a sua disseminação na comunidade científica.

A linguagem foi modernizada a partir de 1970 e o novo padrão, FORTRAN 77, introduzia algumas pequenas modificações na estrutura dos comandos, como o uso de blocos `IF..THEN..ELSE`. No entanto, à essa época, existiam outras linguagens – como C, PASCAL, ADA e MODULA – que haviam introduzido novos conceitos ou sedimentado outros, como a tipagem explícita de variáveis; definição de novos tipos de dados, permitindo a criação de estruturas de dados mais adequadas para resolver problemas; alocação dinâmica de dados; subprogramas recursivos; controle de exceção (uma das principais características da linguagem ADA); e estabelecimento de módulos. FORTRAN 77 não oferecia qualquer desses recursos e, por isso, acabou sendo relegada quase que a segundo plano.

Por meados da década de 80, uma nova revisão da linguagem estava por vir. Batizada então de FORTRAN 8X, ela viria a incorporar praticamente todos aqueles conceitos citados acima (exceto a tipagem explícita e o controle de exceção). O processo de definição do novo padrão foi demorado, porém permitiu que muitas das idéias a serem inseridas no padrão fossem amadurecidas. Assim, surgiu a linguagem FORTRAN 90, que, na visão do autor, não perde em nada para linguagens como C e PASCAL, para os fins a que ela se destina. Mais do que isso, ela incorpora mecanismos para a manipulação de arranjos que não são oferecidos em qualquer outra linguagem e que em muito auxiliam no desenvolvimento de programas científicos.

Em 1996, foi feita mais uma revisão da linguagem, chamada de FORTRAN 95, a qual incorpora pequenas modificações à linguagem FORTRAN 90, motivada pela necessidade de aproximar o padrão à linguagem HIGH PERFORMANCE FORTRAN – HPF, a qual é voltada para o uso de computadores de arquiteturas avançadas, vetoriais e/ou paralelas. A linguagem HPF pode ser considerada uma extensão da linguagem FORTRAN 90 e, atualmente, muitos fabricantes oferecem

compiladores HPF baseados na linguagem FORTRAN 90.

Certamente, podemos afirmar que a linguagem FORTRAN exerceu uma grande influência no desenvolvimento das atividades científicas com necessidade de cálculos intensivos. A enorme quantidade de *"software"* disponível em FORTRAN 77 e FORTRAN 90 – grátis e de boa qualidade – certamente fará com que a mesma continue sendo a escolha da comunidade científica. Um dos grandes cientistas de computação, C.A.R. Hoare, disse, certa vez: *"Não sei como será a linguagem de programação do ano 2000, mas ela será chamada Fortran"*.

1.1 Notação

Ao longo desse texto, será utilizada uma notação para descrever os comandos da linguagem, conforme segue:

- Qualquer símbolo da linguagem – variável, tipo de dado e comando – aparece escrito em `teletipo`;
- Uma palavra englobada por < > indica o nome de uma *variável*, *expressão* ou *comando* da linguagem, que deve *obrigatoriamente* aparecer naquela situação;
- Uma ou mais palavras englobadas por [ ] indica que tais palavras são *opcionais* naquele contexto;
- Mais de uma palavra, englobadas por < >, separadas por |, indica que uma dessas palavras deve ser escolhida.
- O símbolo ::= indica que a palavra à sua esquerda receberá o valor da palavra ou expressão à sua direita.
- O símbolo ♭ representa o espaço em branco.

Capítulo 2

Estrutura de um programa

Um programa ou código-fonte escrito na linguagem FORTRAN 90 é composto por *blocos*, os quais definem as diferentes seções do mesmo. Um programa FORTRAN 90 tem, no mínimo, uma seção chamada de *programa principal*, a qual é composta por comandos não-executáveis (declarações e comentários) e executáveis, e é englobada pelos comandos `PROGRAM <nome-do-programa>` e `END PROGRAM <nome-do-programa>`. A estrutura básica um programa em FORTRAN 90 pode, então, ser descrita como:

```
PROGRAM <nome-do-programa>

[ <declarações> ]

[ <comandos-executáveis> ]

END PROGRAM <nome-do-programa>
```

Semântica:

1. `<declarações>` é um conjunto de *comandos não-executáveis* que definem os *dados* e *subprogramas* a serem usados no programa;
2. `<comandos-executáveis>` é um conjunto de comandos que serão executados, na ordem em que aparecem listados e atendendo a quaisquer desvios em seu *fluxo de execução*, conforme expresso pelo programador, valendo-se dos diferentes comandos de *controle* de fluxo de execução (ver capítulo 5).

Assim, o menor programa que pode existir em FORTRAN 90 é

Exemplo 2.1 *Menor programa em* FORTRAN 90

```
PROGRAM MENOR_PROGRAMA
END PROGRAM MENOR_PROGRAMA
```

Além do programa principal, pode-se definir *subprogramas* (`SUBROUTINE` e `FUNCTION`) e módulos (capítulo 7). Um subprograma é dito *interno*, quando o seu código *pertencer* a uma outra seção (isto é, o seu código-fonte é expresso após o comando `CONTAINS` dentro daquela seção), ou ele é *externo*, quando o seu código-fonte for descrito separadamente – em um outro arquivo ou dentro de um *módulo*. Um módulo (§7.5) é uma seção englobada por `MODULE <nome-do-módulo>` e `END MODULE <nome-do-módulo>`, o qual contém declarações de variáveis e tipos de dados e/ou um conjunto de subprogramas, necessariamente externos a outras seções (programa principal e outros módulos).

Um subprograma interno só pode ser invocado dentro da seção na qual ele é declarado; um subprograma externo pode ser invocado em qualquer seção que dele faça uso. Se um subprograma externo estiver contido em um módulo, então na seção onde ele for invocado deverá constar implicitamente um comando `USE <nome-do-módulo>`.

2.1 Formato livre e fixo

Existem dois tipos diferentes de formatos em que um programa pode se apresentar – livre e fixo. No formato *livre*, um comando pode iniciar em qualquer caracter da linha, a qual contém até 132 caracteres. Nesse formato, os comandos podem aparecer um em cada linha ou, então, mais de um comando numa mesma linha, quando então eles devem ser separados por ; (ponto-e-vírgula). Caso um comando não caiba numa linha, ele deve ser interrompido com o caracter &, e a continuação do comando deve aparecer na linha seguinte. Um comentário, identificado pelo caracter !, pode aparecer em qualquer posição numa linha. O exemplo 2.2 mostra o uso do formato livre.

Exemplo 2.2 *Um programa em* FORTRAN 90 *usando formato livre.*

```
PROGRAM TESTE
! este é um programa de teste
INTEGER :: I, J
        REAL, & ! este é um comentário
            DIMENSION(200,300) :: MATRIZ
        MATRIZ = 100.0
END PROGRAM TESTE
```

Já no formato *fixo*, herdado da linguagem Fortran 77, uma linha tem apenas 72 caracteres, e os comandos devem aparecer apenas a partir da coluna 7, um comando por linha. As colunas 1 a 5 são reservadas para se identificar um comando através de um rótulo numérico; e a coluna 6 é reservada para a colocação de um caracter de continuação (diferente de 0 ou espaço em branco). Comentários podem ser inseridos através do caracter C ou *, na coluna 1, ou através do caracter !, o qual só será considerado como início de comentário se aparecer nas colunas 1 a 5 ou 7 a 72. O exemplo 2.3 mostra como usar o formato fixo, com os mesmos comandos usados no exemplo 2.2.

Exemplo 2.3 *Um programa em* FORTRAN 90 *usando formato fixo.*

```
C       1         2         3         4         5         6         7
C2345678901234567890123456789012345678901234567890123456789012345678901234567890
      PROGRAM TESTE
* este é um programa de teste
      INTEGER :: I, J
      REAL, ! este é um comentário
     +DIMENSION(200,300) :: MATRIZ
      MATRIZ = 100.0
      END PROGRAM TESTE
```

Observe que a declaração do arranjo MATRIZ utiliza o caracter + como caracter de continuação, colocado na coluna 6 da linha subseqüente à linha onde aparece `REAL, ! este é um comentário`.

Os compiladores disponíveis para a linguagem FORTRAN 90 distinguem entre esses dois formatos através da extensão do nome do arquivo. Um programa gravado num arquivo `teste.F` ou `teste.f` será interpretado como sendo em formato fixo; se for utilizada a extensão `F90` ou `f90`, como em `teste.F90`, então o programa será considerado como sendo em formato livre.

2.2 Conjunto de caracteres da linguagem Fortran 90

Qualquer programa em FORTRAN 90 é escrito utilizando-se um conjunto de caracteres, incluindo as letras do alfabeto, os algarismos e outros caracteres especiais:

Letras: 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

Algarismos: 0 1 2 3 4 5 6 7 8 9

Caracteres especiais:		espaço em branco	:	dois pontos	;	ponto-e-vírgula
	=	igual	+	mais	-	menos
	*	asterisco	/	barra	(	abre parêntese
	)	fecha parêntese	,	vírgula	.	ponto
	'	apóstrofo	"	haspas	!	ponto de exclamação
	%	porcentual	&	e comercial	?	ponto de interrogação
	<	menor do que	>	maior do que	$	dólar
	_	sublinha				

O uso de letras minúsculas num programa FORTRAN 90 é dependente do compilador. Caso seu uso seja aceito, então as letras minúsculas são consideradas como se fossem maiúsculas, com exceção de seu uso em constantes literais (ver §4), em caracteres de edição (ver §6.2) e em parâmetros dos comandos `OPEN` e `INQUIRE` (ver §6.1).

Capítulo 3

Tipos e declarações de variáveis

A linguagem FORTRAN 90 apresenta um conjunto de diferentes *tipos* de variáveis, pré-definidos, capazes de armazenar números inteiros, reais e complexos ou, ainda, valores lógicos ou caracteres. Além disso, FORTRAN 90 oferece ao programador a possibilidade de criar novos tipos, permitindo criar diferentes estruturas de dados. Nesse capítulo, trataremos da declaração de variáveis e de tipos.

3.1 Tipos de dados intrínsecos

A linguagem FORTRAN 90 define seis tipos diferentes de dados intrínsecos, a saber:

`INTEGER` Define variáveis de tipo *inteiro*, isto é, capazes de armazenar valores inteiros negativos, positivos e zero.

`REAL` Define variáveis de tipo *real*, isto é, capazes de armazenar valores reais negativos, positivos e zero.

`DOUBLE PRECISION` Define variáveis de tipo *real*, isto é, capazes de armazenar valores reais negativos, positivos e zero, com uma precisão duas vezes superior àquela apresentada por variáveis de tipo `REAL`.

`COMPLEX` Define variáveis de tipo *complexo*, isto é, capazes de armazenar valores complexos onde cada parte – *real* e *imaginária* – é um dado de tipo `REAL`.

`LOGICAL` Define variáveis de tipo *lógico*, isto é, capazes de armazenar os valores `.TRUE.` (verdadeiro) e `.FALSE.` (falso).

`CHARACTER` Define variáveis de tipo *caracter*, isto é, capazes de armazenar um caracter ou um conjunto de caracteres (uma *"string"*).

3.2 Nomes

Um *nome* em FORTRAN 90 é definido como

```
<nome> ::= <caracter-inicial><caracteres>

<caracter-inicial> ::= 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

<caracteres> ::= <caracter-inicial> | <dígitos> | _

<dígitos> :: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Semântica:

1. Um <nome> só pode iniciar com uma letra do alfabeto (é admissível letras em minúsculas), seguido opcionalmente de outras letras, dígitos ou o caracter _.

Dessa forma, são nomes admissíveis na linguagem: A, A09B_HI, ABC__0_45Z (note a ocorrência de dois caracteres _ seguidos. Porém, _ABC e 0_456_B não são nomes admissíveis, pois nenhum começa com alguma das letras do alfabeto.

3.3 Declaração explícita de variáveis

A sintaxe de declaração de variáveis em FORTRAN 90 é a seguinte:

```
<tipo>[<característica>][ ,<lista-atributos> ] [::] <lista-nomes>

<característica> ::= ( [KIND=] <kind> ) | ( [LEN=] <tamanho> ) |
                       *( <kind> | <tamanho> )

<lista-atributos> ::= <atributo-1> [,... [,<atributo-n>] ]

<lista-nomes> ::= <nome-1> [,... [,<nome-n>] ]
```

Semântica:

1. <tipo> é o nome de um tipo intrínseco da linguagem ou definido pelo usuário;
2. <kind> é um valor inteiro que especifica a quantidade de *bytes* utilizada para representar o <tipo>, desde que este seja INTEGER, REAL ou COMPLEX;
3. <tamanho> é um valor inteiro que especifica o número de caracteres de uma variável de <tipo> CHARACTER;
4. <lista-atributos> é uma lista *opcional* de atributos da variável, separados por vírgulas, intrínsecos à linguagem; se algum atributo for especificado, então os símbolos :: devem ser usados, para separar a lista de atributos da lista de nomes de variáveis;
5. <lista-nomes> é uma lista de nomes de variáveis, com no mínimo um nome, separados por vírgulas;
6. Cada variável em <lista-nomes> pode ser inicializada com um *valor*, o qual deve ter o mesmo <tipo>; se a variável é um *arranjo*, então o seu valor é uma lista de valores do mesmo tipo, separados por vírgulas e englobados por (/ e /). Não é permitido, no entanto, utilizar como *valor* uma chamada a um intrínseco (ver §8), com exceção dos intrínsecos KIND, DIGITS, EPSILON, HUGE, MINEXPONENT, MAXEXPONENT, SELECTED_INT_KIND, SELECTED_REAL_KIND e TINY.

Num bloco PROGRAM...END PROGRAM, SUBROUTINE...END SUBROUTINE, FUNCTION...END FUNCTION ou MODULE...END MODULE podem existir inúmeras declarações de variáveis, de diferentes tipos, até que ocorra o primeiro *comando executável* no bloco; a partir daí, não pode mais haver declarações de variáveis, até o fechamento do bloco.

Exemplo 3.1 *Declarações de variáveis intrínsecas*

```
REAL :: X, Y, Z=-1.0
REAL A, B, C
REAL*8 :: U = 3.0   ! Uma variável REAL com 8 bytes de tamanho
DOUBLE PRECISION :: M=100.0D0, N
DOUBLE PRECISION A, B, C
INTEGER :: I=4, J, K
```

```
INTEGER F1,F2
INTEGER(2) :: DOIS ! Uma variável INTEGER com 2 bytes de tamanho
COMPLEX :: RHO = (0.0,3.0)
COMPLEX U, V, W
CHARACTER :: SIMBOLO = '%'
CHARACTER TIPO
CHARACTER(LEN=10) :: NOME = 'JOSÉ SILVA' ! Uma "string" com 10 caracteres
CHARACTER*13 :: NOME = 'MANOEL CARLOS'
LOGICAL :: ACHOU = .FALSE.
LOGICAL PROCURE, FIM
```

Note que as declarações acima não contém nenhum atributo, os quais serão apresentados posteriormente.

3.4 Declaração implícita de variáveis

Como salientado anteriormente, a linguagem FORTRAN 90 é uma evolução de versões anteriores, Fortran 66 e Fortran 77. Naquelas linguagens, era permitido utilizar-se variáveis sem que elas tivessem sido declaradas, o que é uma *má técnica de programação*, pois permite a ocorrência de erros colaterais devido ao uso indevido de variáveis (por exemplo, passando como argumento de uma subrotina uma variável `REAL` quando deveria ser uma variável `COMPLEX`, o que normalmente leva a um erro fatal do programa).

Feita essa advertência, é admitido em FORTRAN 90 que se utilize variáveis que não tenham sido declaradas, as quais seguem a seguinte regra:

Variável do tipo `INTEGER`: Uma variável cuja primeiro caracter de seu nome seja `I`, `J`, `K`, `L`, `M` ou `N`;

Variável do tipo `REAL`: Uma variável cujo primeiro caracter de seu nome não seja nenhum dos listados acima.

Observe como nessa regra percebe-se as origens matemáticas da linguagem, já que as letras *i, j, k, l, m* e *n* são, normalmente, associadas a índices inteiros.

A fim de evitar que tal regra seja utilizada pelo compilador da linguagem FORTRAN 90, pode-se incluir em qualquer bloco, antes da declaração de variáveis, o comando `IMPLICIT NONE`, o qual faz com que todas as variáveis utilizadas no bloco sejam declaradas, prevenindo a ocorrência de erros como o mencionado anteriormente.

3.5 Tipos de dados definidos pelo usuário

Para se manipular estruturas de dados mais complexas, de forma clara, a linguagem FORTRAN 90 oferece a possibilidade de se definir novos tipos de dados, através do comando `TYPE..END TYPE`, cuja sintaxe é a seguinte:

```
TYPE <nome-tipo>
    [PRIVATE]
    [SEQUENCE]
    <tipo-1>,[,<lista-atributos-1> ] [::] <lista-nomes-1>
    ...
    [ <tipo-k>,[,<lista-atributos-k> ] [::] <lista-nomes-k>
END TYPE <nome-tipo>
```

Semântica:

1. `<nome-tipo>` é o nome do tipo definido pelo usuário;
2. `PRIVATE` é um atributo que define as *partes* `<nome-1>, ..., <nome-s>` como *visíveis* apenas dentro de um *módulo* no qual o tipo é declarado; esse atributo só pode ser usado quando o tipo é declarado dentro de um módulo;

3. SEQUENCE é um atributo que faz com que as partes do tipo sejam alocadas em áreas contíguas da memória; deve ser usado quando variáveis desse tipo forem utilizadas em comandos COMMON e EQUIVALENCE (ver §3.7.1, 3.7.2);

4. <tipo-1>, <tipo-2>, ..., <tipo-k> são tipos intrínsecos da linguagem ou definidos pelo usuário;

5. <lista-nomes-1>, ..., <lista-nomes-k> contém os nomes das *partes* do tipo <nome-tipo>, as quais serão referenciadas por esses nomes quando forem acessadas, na forma <nome-tipo>%<nome-parte>.

A declaração de um tipo definido pelo usuário só é visível ao bloco no qual ele foi definido, e deve preceder qualquer declaração de variável daquele tipo.

3.5.1 Declaração de variáveis de tipo definido pelo usuário

Variáveis cujo tipo seja um tipo definido pelo usuário devem ser declaradas de acordo com a seguinte sintaxe:

```
TYPE (<nome-tipo>)[ ,<lista-atributos> ] [::] <nome-1>[,... [,<nome-m>] ]

<lista-atributos> ::= <atributo-1> [,... [,<atributo-n>] ]
```

Semântica:

1. <nome-tipo> é o nome do tipo definido pelo usuário;

2. <atributo-1>, ..., <atributo-n> é uma lista *opcional* de atributos da variável, separados por vírgulas, intrínsecos à linguagem;

3. <nome-1>, ..., <nome-m> é uma lista de nomes de variáveis, com no mínimo um nome, separados por vírgulas, todas do mesmo <tipo> e com os mesmos atributos, se houverem.

Exemplo 3.2 *Declarações de tipos definidos pelo usuário*

```
TYPE VETOR_3D
    REAL :: X,Y,Z
END TYPE VETOR_3D

TYPE PLANO
    TYPE (VETOR_3D) :: NORMAL
    REAL :: D
END TYPE PLANO

TYPE (VETOR_3D) :: U,V,W
TYPE (PLANO) :: FACE
```

Nesse exemplo, é definido um tipo chamado VETOR_3D, o qual tem três partes do tipo REAL, chamadas X, Y, Z. A seguir, é definido um tipo PLANO, o qual tem duas partes: uma chamada NORMAL, de tipo VETOR_3D e outra, chamada D, do tipo REAL. Após, são definidas três variáveis U, V, W do tipo VETOR_3D e uma variável FACE de tipo PLANO.

3.6 Comandos de especificação de atributos

Um *atributo* é uma característica apresentada por um dado. Por exemplo, um vetor de dados tem um atributo chamado DIMENSION; ou um argumento de um subprograma pode ser opcional, e para tanto declara-se tal argumento com o seu tipo e com o atributo OPTIONAL. Os atributos existentes na linguagem são apresentados a seguir.

3.6.1 KIND

Uma variável de tipo INTEGER, REAL, COMPLEX, DOUBLE PRECISION, LOGICAL ou CHARACTER tem associada a si um KIND (ou *gênero*), que especifica a quantidade de *bytes* usados para representar aquela variável. A declaração desse atributo é opcional e, caso não seja especificada, então será assumido o KIND *default* de cada tipo, dependente da implementação do compilador e do computador a ser utilizado; por exemplo, usando o compilador *Microsoft Fortran Powerstation 1.0* em um microcomputador dotado de um processador *Pentium 4*, o KIND *default* de cada tipo está listado na tabela 3.1.

tipo	KIND
INTEGER	4
REAL	4
COMPLEX	4
DOUBLE PRECISION	8
LOGICAL	4
CHARACTER	1

Tabela 3.1: KIND *default* de tipos no compilador MS Fortran Powerstation 1.0.

Note que uma variável do tipo DOUBLE PRECISION *não pode* ser declarada com o atributo KIND. Por outro lado, se uma variável do tipo REAL tiver, em um computador hipotético, KIND=4, então as declarações

```
REAL(KIND=8) :: A
REAL(8) :: B
DOUBLE PRECISION C
```

definem variáveis que terão a mesma capacidade de representação de números reais, em *precisão dupla*.

É possível, também, se utilizar do intrínseco KIND(<valor>) (ver capítulo 8) para se determinar o gênero de um <valor>.

3.6.2 PUBLIC

O atributo PUBLIC indica que uma variável declarada em um módulo é visível fora do módulo, isto é, pode ser lida e alterada. Esse atributo é *"default"* da linguagem; qualquer variável declarada em um módulo que não tenha o atributo PRIVATE (ver §3.6.3) é considerada como PUBLIC.

Exemplo 3.3 *Uso do atributo PUBLIC*

```
MODULE MODULO_A
    INTEGER, PUBLIC :: A, B = 3
    INTEGER C, D
    PUBLIC C
    CONTAINS
        SUBROUTINE SUB_1
            C = 2
            A = B+C
            D = 2*A
        END SUBROUTINE SUB_1
END MODULE MODULO_A

PROGRAM PRINCIPAL
USE MODULO_A
IMPLICIT NONE
```

```
    A = B+C
    CALL SUB_1
    D = 5+D

END PROGRAM PRINCIPAL
```

Note as duas formas de se definir uma variável como sendo PUBLIC; ou junto com o seu tipo, como na declaração de A e B, ou separadamente, como na declaração do atributo da variável C. Todas as variáveis, inclusive D, são públicas *e, portanto, podem ser lidas e alteradas dentro do programa principal, já que ele usa o módulo MODULO_A. Ao executarmos esse programa, B=3, C=2, A=5 e D=15.*

3.6.3 PRIVATE

O atributo PRIVATE indica que uma variável (ou partes de um tipo) é visível apenas dentro de um módulo.

Exemplo 3.4 *Uso do atributo PRIVATE*

```
MODULE MODULO_A
    INTEGER, PRIVATE :: A, B = 3
    INTEGER C, D
    PRIVATE C
    CONTAINS
        SUBROUTINE SUB_1
            C = 2
            A = B+C
            D = 2*A
        END SUBROUTINE SUB_1
END MODULE MODULO_A

PROGRAM PRINCIPAL
USE MODULO_A
IMPLICIT NONE

    A = B+C ! Erro, variáveis A, B e C só existem dentro do módulo MODULO_A
    CALL SUB_1
    D = 5+D ! Correto

END PROGRAM PRINCIPAL
```

Ao se submeter esse código ao compilador, ocorrerão três erros, pois as variáveis A, B e C não foram declaradas no programa principal; sua declaração no módulo MODULO_A, usando o atributo PRIVATE, "escondeas" daquele.

3.6.4 DATA

O atributo DATA serve para se inicializar variáveis. Tais variáveis contém os valores com as quais foram inicializadas já quando do início da execução do bloco no qual foram inicializadas; elas podem ter seus valores modificados subseqüentemente. Existem duas formas de sintaxe para esse atributo:

```
<tipo>[,<lista-atributos>] [::] <nome-1> [= <valor-nome-1> = <valor-nome-1> [,...
                                                [,<nome-m> = <valor-nome-m> ] ]

<tipo>[,<lista-atributos>] [::] <nome-1> [,... [,<nome-m> ] ]
DATA <nome-1> [,... [,<nome-m> ] ] / <valor-nome-1> [,... [,<valor-nome-m] ] /

<lista-atributos> ::= <atributo-1> [,... [,<atributo-n>] ]
```

Semântica:

1. <nome-1>, ..., <nome-m> é uma lista de nomes de variáveis, as quais serão inicializadas com os valores <valor-nome-1>, ..., <valor-nome-m>; se a variável é um *arranjo*, então o seu valor é uma lista de valores do mesmo tipo, separados por vírgulas e englobados por (/ e /).

Exemplo 3.5 *Uso do atributo DATA*

```
INTEGER, DIMENSION(4) :: U = (/ 1, 2, 9, 16 /)
REAL :: X = 1.0, Y, K, Z = 5.0
DATA Y, K / -1.0, 6.0 /
```

No exemplo, U é um arranjo com uma dimensão, com quatro elementos, os quais foram inicializados como U(1)=1, U(2)=2, U(3)=9, U(4)=16; X e Z são variáveis escalares, as quais foram inicializadas como X=1.0 e Z=5.0; e Y e K foram declaradas como do tipo REAL mas foram inicializadas posteriormente com o atributo DATA, recebendo os valores -1.0 e 6.0, respectivamente.

3.6.5 PARAMETER

O atributo PARAMETER indica que uma variável, a qual deve ser inicializada, utilizando o atributo DATA, não poderá ter seu valor alterado. Por exemplo,

Exemplo 3.6 *Uso do atributo PARAMETER*

```
INTEGER, PARAMETER :: M = 100, N = 20, NZ = M*N - M
REAL :: PI
PARAMETER (PI = 3.1415926)
```

Observe que a variável NZ é inicializada com uma expressão envolvendo M e N, cujos valores foram definidos anteriormente.

3.6.6 DIMENSION

O atributo DIMENSION define uma variável como sendo um *arranjo*, isto é, uma matriz ou vetor. Para definirmos um arranjo, especificamos os limites de indexação de cada uma de suas dimensões; dependendo do contexto, esses limites podem ou devem ser omitidos. Os limites de indexação de uma dimensão definem o *"rank"* de um arranjo, e o número de dimensões e de elementos de cada dimensão a *"shape"* do arranjo.

A sintaxe de uso do atributo pode ser expressa como

```
DIMENSION( <dimensão> [, ... [, <dimensão> ] )

<dimensão> ::= <nel> | [<inf>]:[<sup>] | *
```

Semântica:

1. <nel> é uma expressão do tipo INTEGER e representa o número de elementos naquela dimensão, a qual tem, como limites inferior e superior de indexação, os valores 1 e <nel>;
2. <inf> e <sup> são expressões do tipo INTEGER e representam os limites inferior e superior de indexação daquela dimensão, respectivamente;
3. No máximo sete dimensões podem ser especificadas;
4. Um arranjo com o atributo ALLOCATABLE (apresentado a seguir) deve usar apenas o símbolo : para declarar suas dimensões;

5. Na passagem de arranjos como argumentos de um subprograma, ou em expressões envolvendo arranjos, então a omissão de <inf> ou <sup> pode ocorrer ou, ainda, deve-se usar *, conforme descrito nos capítulos 4 e 7;

Exemplo 3.7 *Declaração de arranjos*

```
REAL, DIMENSION(10) :: A
DOUBLE PRECISION, DIMENSION(-1:2,2) :: B
COMPLEX. DIMENSION(20) :: B, X
CHARACTER, DIMENSION(4) :: CODIGO
```

Observe que o arranjo CODIGO não *é uma* "string" *de quatro caracteres; cada um dos quatro elementos do arranjo podem armazenar apenas* um *caracter.*

3.6.7 ALLOCATABLE

O atributo ALLOCATABLE indica que um arranjo será *alocado dinamicamente*, através do comando ALLOCATE.

Exemplo 3.8 *Uso do atributo ALLOCATABLE*

```
COMPLEX, DIMENSION(:,:), ALLOCATABLE :: PONTOS
REAL, DIMENSION(:), ALLOCATABLE :: U, V
INTEGER, DIMENSION(:) :: A
ALLOCATABLE A
```

3.6.8 EXTERNAL

O atributo EXTERNAL é usado para indicar que um nome é um subprograma externo, um subprograma passado como argumento de outro, ou ainda um bloco BLOCK DATA.

Exemplo 3.9 *Uso do atributo EXTERNAL*

```
REAL, EXTERNAL :: CALOR
INTEGER :: HORA
EXTERNAL HORA
```

3.6.9 INTRINSIC

O atributo INTRINSIC serve para indicar que um nome é uma função intrínseca da linguagem.

Exemplo 3.10 *Uso do atributo EXTERNAL*

```
REAL, INTRINSIC :: SIN
DOUBLE PRECISION, INTRINSIC :: DSIN
```

Em FORTRAN 90, *os intrínsecos (ver capítulo 8) têm nomes genéricos, independentes do tipo de argumento usado e do valor retornado (como SIN), ou nomes específicos, dependentes do tipo, como DSIN.*

3.6.10 SAVE

O atributo SAVE é usado para se preservar o valor de variáveis, declaradas em um subprograma.

Exemplo 3.11 *Uso do atributo EXTERNAL*

```
REAL, SAVE :: U, V, W
DOUBLE PRECISION :: A, B
SAVE A, B
```

3.6.11 INTENT

O atributo INTENT é usado para se especificar como é passado um argumento para um subprograma e como ele é retornado. A sintaxe é a seguinte:

```
INTENT ( IN | OUT | INOUT )
```

Semântica:

1. Se um argumento de um subprograma é declarado como INTENT(IN), então esse argumento é dito de entrada. Esse argumento pode ser usado em expressões e comandos da linguagem mas não pode ter seu valor alterado;
2. Se um argumento de um subprograma é declarado como INTENT(OUT), então esse argumento é dito de saída. Esse argumento só pode ser usado em expressões e comandos da linguagem depois de um valor ter sido atribuído a ele;
3. Se um argumento de um subprograma é declarado como INTENT(INOUT), então esse argumento é dito de entrada-e-saída. Esse argumento pode ser usado em expressões e comandos da linguagem e pode ter seu valor alterado.

Se um argumento de um subprograma não é declarado com o atributo INTENT, então ele é considerado como sendo de entrada-e-saída.

Exemplo 3.12 *Uso do atributo INTENT*

```
SUBROUTINE RETIRA_VALOR (A,B,C)
INTEGER, INTENT(IN) :: A
INTEGER, INTENT(OUT) :: B
INTEGER, INTENT(INOUT) :: C
INTEGER :: I, J

    I = 2*A
    J = 3*B - I ! Erro, B é um argumento OUT e não teve um valor
                ! atribuído a ele antes de ser usado
    C = I+J+C

END SUBROUTINE RETIRA_VALOR
```

3.6.12 OPTIONAL

Quando um subprograma é escrito de tal forma que nem todos seus argumentos são necessários, deve-se utilizar o atributo OPTIONAL para indicar os argumentos que podem ser omitidos.

Exemplo 3.13 *Uso do atributo OPTIONAL*

```
SUBROUTINE RETIRA_VALOR (A,B,C)
INTEGER, INTENT(IN) :: A
INTEGER, INTENT(OUT) :: B
INTEGER, INTENT(INOUT), OPTIONAL :: C
INTEGER :: I, J

    I = 2*A
    B = 3*A - I
    C = I+J+C

END SUBROUTINE RETIRA_VALOR

PROGRAM TESTE
INTEGER :: U = 1, V, W = 3.0
```

```
EXTERNAL RETIRA_VALOR

    CALL RETIRA_VALOR( A = U, B = V)

END PROGRAM TESTE
```

Quando se utilizam argumentos do tipo OPTIONAL, é necessário fazer uma associação entre o nome do argumento e o nome da variável transmitida ao subprograma, caso o número de argumentos transmitidos seja menor do que o número de argumentos declarados no subprograma.

3.6.13 POINTER

Um atributo POINTER indica que uma variável é um *ponteiro*, isto é, ela contém o endereço de uma outra variável de mesmo tipo. Uma variável do tipo POINTER não pode ser inicializada nem tampouco pode ter os atributos: ALLOCATABLE, EXTERNAL, INTENT, INTRINSIC, PARAMETER ou TARGET.

Exemplo 3.14 *Uso do atributo POINTER*

```
TYPE NODO_ARVORE_BINARIA
    INTEGER :: VALOR
    TYPE (NODO_ARVORE_BINARIA), POINTER :: NODO_FILHO_ESQUERDA, NODO_FILHO_DIREITA
END TYPE NODO_ARVORE_BINARIA

TYPE (NODO_ARVORE_BINARIA) :: RAIZ
POINTER RAIZ
```

Nesse exemplo, utiliza-se um tipo definido pelo programador, o qual representa nodos de uma árvore binária; cada nodo tem, além do valor armazenado nele, dois ponteiros para os seus nodos filhos. Também é declarado um ponteiro, de mesmo tipo, que será o nodo raiz *da árvore. Note, também, as duas formas de declaração usando esse atributo. O uso de ponteiros e estruturas de dados como essa são discutidos no capítulo 9.*

3.6.14 TARGET

O atributo TARGET serve para indicar que uma determinada variável poderá ser associada a uma variável com o atributo POINTER, a qual apontará para aquela.

Exemplo 3.15 *Uso do atributo TARGET*

```
TYPE (NODO_ARVORE_BINARIA), TARGET :: P

TYPE (NODO_ARVORE_BINARIA) :: Q
TARGET Q

TYPE (NODO_ARVORE_BINARIA), POINTER :: RAIZ
```

Com as declarações acima, é possível que o apontador RAIZ aponte para quaisquer uma das variáveis P e Q.

3.7 Comandos de especificação

Em FORTRAN 90 é possível designar algumas variáveis com características comuns, particularmente no tocante à maneira como as mesmas serão alocadas na memória do computador. Essas características são especificadas através dos comandos COMMON, EQUIVALENCE, IMPLICIT, NAMELIST e SEQUENCE.

3.7.1 COMMON

O comando COMMON especifica que variáveis declaradas em *diferentes blocos*, possivelmente com nomes diferentes, ocupam os *mesmos endereços na memória* do computador. Um exemplo típico é permitir que dados declarados no programa principal sejam acessados e/ou modificados por um subprograma.

A sintaxe do comando é a seguinte:

```
COMMON [/ <nome-do-bloco-common> /] <lista-nomes>

<lista-nomes> ::= <nome-1> [,... [,<nome-n>] ]
```

Semântica:

1. <nome-do-bloco-common> é um nome opcional, associado a uma lista de variáveis do bloco COMMON;
2. <nome-1> a <nome-m> são nomes de variáveis, as quais serão armazenadas em uma mesma área de memória do que outras variáveis, declaradas em outro bloco e que também, por sua vez, deverão ser listadas em comando COMMON no bloco em que foram declaradas.

Exemplo 3.16 *Uso do atributo COMMON*

```
PROGRAM TESTE
REAL :: A = 1.0, B = 2.0, C
COMMON /BLOCO1/ A, B

    CALL SUB

END PROGRAM TESTE

SUBROUTINE SUB
REAL :: U, V
COMMON /BLOCO1/ U, V

    PRINT *,'U=',U,' V=',V

END SUBROUTINE SUB
```

Um bloco COMMON, de nome BLOCO1, foi declarado no programa principal. Esse bloco contém duas variáveis do tipo REAL, A e B, as quais foram inicializadas com os valores 1.0 e 2.0, respectivamente. Na subrotina SUB, são declaradas duas variáveis REAL, chamadas U e V, as quais estão associadas ao mesmo bloco COMMON BLOCO1. Assim, A e U são dois nomes diferentes para o mesmo *endereço na memória; o mesmo ocorre com B e V. Logo, ao se executar esse código, serão impressos os valores U=1.0 e V=2.0.*

3.7.2 EQUIVALENCE

O comando EQUIVALENCE especifica que variáveis declaradas em um *mesmo bloco* ocupam os *mesmos endereços na memória* do computador.

A sintaxe do comando é a seguinte:

```
EQUIVALENCE ( <lista-nomes-1> ] ) [,... [, ( <lista-nomes-k> ) ] ]

<lista-nomes-1> ::= <nome-i> [,... [,<nome-j>] ]

<lista-nomes-k> ::= <nome-k> [,... [,<nome-l>] ]
```

Semântica:

1. <nome-i> a <nome-1> são nomes de variáveis, englobadas entre parênteses ( ), as quais ocupam o mesmo endereço na memória do computador.

Exemplo 3.17 *Uso do atributo EQUIVALENCE*

```
PROGRAM TESTE
REAL :: A = 1.0, B = 2.0, C, D, E
EQUIVALENCE ( A, C ), ( B, D )

    E = C + D

END PROGRAM TESTE
```

Nesse exemplo, as variáveis A e C ocupam o mesmo endereço de memória (assim como B e D ocupam outro endereço único). Dessa forma, ao referenciarmos C e D na expressão cujo resultado será atribuído à variável E, é como se estivéssemos referenciando, de forma equivalente, *as variáveis A e B.*

3.7.3 IMPLICIT

O comando IMPLICIT serve para especificarmos de forma implícita variáveis. Como salientado anteriormente, essa não é uma técnica recomendada de programação, mas é apresentada aqui a fim de não deixar lacunas no entendimento de programas já existentes.

A sintaxe do comando é a seguinte:

```
IMPLICIT [ <tipo> ( <lista-letras> ) [,... [ <tipo> ( <lista-letras> ) ] ] | NONE

<lista-letras> ::= <letra-1> [- <letra-2 ]
```

Semântica:

1. <tipo> é o tipo do dado, pré-definido ou definido na linguagem;
2. <letra-1> é uma letra que indica que variáveis iniciando com essa letra serão do tipo <tipo>;
3. se <letra-2> for especificada, significa que quaisquer variáveis iniciando com letras entre <letra-1> e <letra-2> serão daquele tipo;
4. se NONE for utilizado, então não haverá declaração implícita de variáveis dentro do bloco no qual IMPLICIT NONE foi declarado.

Exemplo 3.18 *Uso do atributo IMPLICIT*

```
PROGRAM TESTE
IMPLICIT COMPLEX (A-H), INTEGER (P-T)

    PONTO = 4
    AREA = (5.0,1.0)

END PROGRAM TESTE
```

Nesse exemplo, a variável PONTO é do tipo INTEGER, pois começa com a letra P; e AREA é do tipo COMPLEX, pois começa com A.

3.7.4 NAMELIST

O comando NAMELIST é usado para definir listas de variáveis que serão usadas, posteriormente, em operações de entrada e saída de dados. A sintaxe do comando é a seguinte

```
NAMELIST / <nome-lista> / <lista-nomes>

<lista-nomes> ::= <nome-1> [,... [,<nome-n>] ]
```

Semântica:

1. <nome-lista> é o nome da lista de variáveis, a qual será referenciada dentro de um comando de entrada e saída de dados;
2. <nome-1>, ..., <nome-n> são nomes das variáveis que serão lidas e/ou escritas através de um comando de entrada e saída.

Exemplo 3.19 *Uso do atributo NAMELIST*

```
INTEGER A, B
CHARACTER, DIMENSION(4) :: NOME
REAL X
NAMELIST /LISTA1/ NOME, A, B, X

READ (*, LISTA1)
```

Uma NAMELIST chamada LISTA1 foi declarada, a qual contém quatro variáveis, NOME, A, B e X. Ao se executar o comando de leitura READ, o computador esperará que sejam inseridos (através do teclado, usualmente), um conjunto de quatro caracteres, dois valores inteiros e um valor real, os quais serão armazenados nas variáveis listadas na NAMELIST.

3.7.5 SEQUENCE

O comando SEQUENCE é usado dentro de uma declaração de um tipo definido pelo usuário, e faz com que as partes componentes do tipo sejam alocadas na memória do computador na seqüência em que são declaradas.

Exemplo 3.20 *Uso do atributo SEQUENCE*

```
TYPE VETOR_3D
    SEQUENCE
    REAL :: X, Y, Z
END TYPE VETOR_3D
```

3.8 Modelos de representação numérica

Os valores numéricos admitidos na linguagem FORTRAN 90, tais como números inteiros, binários e reais (ponto-flutuante), são representados de acordo com um determinado modelo para cada tipo de número. Esses modelos são definidos com base em algumas quantidades e equações, as quais serão definidas a seguir para cada tipo.

3.8.1 Modelo de representação de números inteiros

Um número inteiro i é definido como

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1} \tag{3.1}$$

onde s é o *sinal* (igual a $+1$ ou -1), r é a *base* (um número inteiro, tal que $r > 1$), q é o *número máximo de dígitos* ($q > 0$) e cada w_k é um *dígito inteiro*, expresso na base r, i.e. $0 \leq w_k < r$. Os parâmetros r e q definem o modelo de representação de números inteiros; para uma expressão de tipo INTEGER (com qualquer KIND admissível), esses valores podem ser obtidos através dos intrínsecos RADIX e DIGITS; além disso, pode-se definir um KIND com o intrínseco SELECTED_INT_KIND (ver §8.4).

3.8.2 Modelo de representação de números inteiros em binário

Um número inteiro j em binário, sem sinal(i.e. positivo) é definido como

$$j = \sum_{k=0}^{n-1} w_k \times 2^k, \tag{3.2}$$

onde n é o *número máximo de dígitos* ($n > 0$) e cada w_k é um *dígito binário*, i.e. $0 \leq w_k < 2$. O parâmetro n define o modelo de representação de números inteiros binários; para uma expressão de tipo `INTEGER` (com qualquer `KIND` admissível), esse valor pode ser obtido através do intrínseco `BIT_SIZE` (ver §8.7). Observe que esse modelo só se aplica à utilização dos intrínsecos de manipulação de *"bits"* (ver §8.7).

3.8.3 Modelo de representação de números reais

Um número real x é definido como

$$x = \begin{cases} s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, & x \neq 0 \\ b^0 \times \sum_{k=1}^{p} 0 \times b^{-k}, & x = 0 \end{cases} \tag{3.3}$$

onde s é o *sinal* (igual a $+1$ ou -1), b é a *base* (um número inteiro, tal que $b > 1$), p é o *número máximo de dígitos* ($p > 1$), e é o expoente (um número inteiro, tal que $e_{\min} \leq e \leq e_{\max}$, $e_{\min}$ e $e_{\max}$ os menor e maior expoentes representáveis) e cada f_k é um *dígito inteiro*, expressos na base b, i.e. $0 \leq f_k < b$, com $f_1 > 0$. Os parâmetros b, p, $e_{\min}$ e $e_{\max}$ definem o modelo de representação de números reais; para uma expressão de tipo `REAL` (com qualquer `KIND` admissível), esses valores podem ser obtidos através dos intrínsecos `RADIX`, `DIGITS`, `MINEXPONENT`, `MAXEXPONENT` e `RANGE`; além disso, pode-se definir um `KIND` com o intrínseco `SELECTED_INT_REAL` (ver §8.4).

Capítulo 4

Literais, expressões e operadores

Neste capítulo abordaremos os aspectos relativos à sintaxe usada na construção de literais, expressões e operadores utilizada na linguagem FORTRAN 90.

4.1 Literais

O nome *literal* é dado àquelas constantes dos diferentes tipos pré-definidos na linguagem, como números inteiros, reais de precisão simples e dupla, complexos, valores lógicos e caracteres.

4.1.1 Números inteiros

Um número inteiro, de tipo INTEGER, é qualquer número na forma

```
[+|-]d[_<kind>]
```

Semântica:

1. o sinal, se omitido, indica um número positivo;
2. d é um conjunto de algarismos, entre 0 e 9;
3. <kind> representa o *"kind"* (gênero) do número inteiro, obtido com o uso dos intrínsecos KIND (§8.3.3) ou SELECTED_INT_KIND (§8.4.11).

Exemplo 4.1 *Números inteiros:*

```
-435656790
+32768
32768_INTEIRO2
+00021323
```

No exemplo, INTEIRO2 é o nome de uma variável que contém o KIND desejado.

4.1.2 Números reais de tipo REAL

Um número real, de tipo REAL, é qualquer número na forma

```
[+|-]i.[f][E|e[+|-]p][_<kind>]
```

Semântica:

1. o sinal, se omitido, indica um número positivo;
2. i é um conjunto de algarismos, entre 0 e 9, que representa a *parte inteira* do número;

3. `f` é um conjunto de algarismos, entre 0 e 9, que representa a *parte decimal* do número;
4. `E` ou `e` indicam que o número encontra-se representado em notação científica;
5. o sinal do expoente da potência de 10, se omitido, indica expoente positivo;
6. `p` é um conjunto de algarismos, entre 0 e 9, que representa o expoente da potência de 10 do número;
7. `<kind>` representa o *"kind"* do número real, obtido com o uso dos intrínsecos `KIND` (§8.3.3) ou `SELECTED_REAL_KIND` (§8.4.12).

Exemplo 4.2 *Números reais de tipo `REAL`:*

```
-3.1415926
+2.
0.9113
3.169E-8_FLUT
+1,660e-24
0.
```

No exemplo, `FLUT` é o nome de uma variável que contém o `KIND` desejado.

4.1.3 Números reais de tipo `DOUBLE PRECISION`

Um número real, de tipo `DOUBLE PRECISION`, é qualquer número na forma

`[+|-]i.[f][D|d[+|-]p][_<kind>]`

Semântica:

1. o sinal, se omitido, indica um número positivo;
2. `i` é um conjunto de algarismos, entre 0 e 9, que representa a *parte inteira* do número;
3. `f` é um conjunto de algarismos, entre 0 e 9, que representa a *parte decimal* do número;
4. `D` ou `d` indicam que o número encontra-se representado em notação científica;
5. o sinal do expoente da potência de 10, se omitido, indica expoente positivo;
6. `p` é um conjunto de algarismos, entre 0 e 9, que representa o expoente da potência de 10 do número;
7. `<kind>` representa o *"kind"* do número real, obtido com o uso do intrínseco `KIND` (§8.3.3).

Exemplo 4.3 *Números reais do tipo `DOUBLE PRECISION`:*

```
-3.1415926D0
+2.D0
0.9113D0
3.169D-8
+1,660d-24
0.D0
```

4.1.4 Literais lógicos

Um literal lógico só admite dois valores: `.TRUE.` (verdadeiro) ou `.FALSE.` (falso).

4.1.5 Caracteres

Um literal composto por caracteres é qualquer constante englobada por ' ou ".

Exemplo 4.4 *Literais compostos por caracteres*

```
'a'
'ABCdef'
'-3.1415926E0'
```

4.2 Operadores

Na linguagem FORTRAN 90, existem disponíveis operadores de *atribuição e apontamento*, *aritméticos*, *relacionais*, *lógicos* e de *caracteres*. Além disso, um programador pode definir outros operadores (ver §7.4.1).

4.2.1 Atribuição e apontamento

Como vimos no Capítulo 3, existem basicamente duas classes de variáveis na linguagem FORTRAN 90: aquelas que armazenam *dados* e outras que armazenam *endereços de variáveis*. Na primeira encontram-se as variáveis declaradas com os tipos pré-definidos na linguagem e aqueles definidos pelo usuário; e na segunda, aquelas que tenham sido declaradas com o atributo POINTER.

Essas duas classes de variáveis são diferentes entre si e, portanto, existem duas formas diferentes de se atribuir valores a elas. A *atribuição* do resultado de uma expressão a uma variável é feita com o símbolo =, cuja sintaxe é a seguinte:

```
<nome> = <expressão>
```

Semântica:

1. <nome> é o nome da variável que receberá o valor de <expressão>;
2. Se <nome> e <expressão> não são do mesmo tipo, então ou irá ocorrer um arredondamento, ou perda de informação, ou um erro.

A uma variável com atributo POINTER não pode, no entanto, ser atribuído um valor. Ela apenas pode *apontar* para um outro apontador. Nesse caso, utiliza-se o operador =>,

```
<nome-apontador-1> => <nome-apontador-2>
```

Semântica:

1. <nome-apontador-1> e <nome-apontador-2> são os nomes de variáveis de *mesmo tipo* e declaradas com o atributo POINTER; um apontamento faz com que o apontador <nome-apontador-1> contenha o *mesmo endereço de memória* do dado apontado por <nome-apontador-2>.

4.2.2 Operadores aritméticos

Em FORTRAN 90, existem os operadores aritméticos usuais, para realizar a soma (+), subtração e negação de sinal (-), multiplicação (*), divisão (/) e elevar um número a uma potência inteira (**). A sintaxe de utilização é

```
<expressão> ::= [ - ] <expressão-1>  + | - | * | / | **  [ - ] <expressão-2>
```

Semântica:

1. <expressão-1> e <expressão-2> são expressões aritméticas admissíveis na linguagem, envolvendo literais numéricos, variáveis e chamadas a funções e intrínsecos matemáticos;

2. Se o operador utilizado for **, então <expressão-2> deve ser de tipo INTEGER e, se o sinal - preceder <expressão-2>, então - <expressão-2> deve ser englobado por abre e fecha parênteses (,);

3. <expressão> é o resultado numérico.

4.2.3 Operadores relacionais

Os operadores relacionais são utilizados para se realizar a comparação lógica entre os valores de expressões aritméticas. Esses operadores são os seguintes:

igual: == ou .EQ.

diferente: /= ou .NE.

menor: < ou .LT.

maior: > ou .GT.

menor ou igual: <= ou .LE.

maior ou igual: >= ou .GE.

A sintaxe de utilização é

```
<expressão> ::= <expressão-1> <operador-relacional> <expressão-2>
```

Semântica:

1. <expressão-1> e <expressão-2> são expressões aritméticas admissíveis na linguagem, envolvendo literais numéricos, variáveis e chamadas a funções e intrínsecos matemáticos;

2. <expressão> é o resultado lógico, podendo assumir os valores .TRUE. e .FALSE.

4.2.4 Operadores lógicos

Os operadores lógicos permitem combinar expressões lógicas. Os operadores disponíveis são os seguintes:

negação lógica: .NOT.

multiplicação lógica: .AND.

adição lógica: .OR.

equivalência lógica: .EQV.

negação da equivalência lógica: .NEQV.

A sintaxe de utilização é a seguinte:

```
<expressão> ::= <operador-lógico-unário> <expressão-1> |
                <expressão-1> <operador-lógico-binário> <expressão-2>
```

Semântica:

1. <expressão-1> e <expressão-2> são expressões lógicas admissíveis na linguagem;

2. <operador-lógico-unário> é a negação lógica, .NOT.;

3. <operador-lógico-binário> é um dentre .AND., .OR., .EQV. ou .NEQV.;

4. <expressão> é o resultado lógico, podendo assumir os valores .TRUE. e .FALSE.

Os resultados possíveis da utilização desses operadores são mostrados na tabelas 4.1 e 4.2.

<expressão>	.NOT.
.TRUE.	.FALSE.
.FALSE.	.TRUE.

Tabela 4.1: Tabela verdade para .NOT.

<expressão-1>	<expressão-2>	.AND.	.OR.	.EQV.	.NEQV.
.FALSE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.TRUE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.

Tabela 4.2: Tabela verdade para os operadores lógicos .AND., .OR., .EQV. e .NEQV.

4.2.5 Operador de concatenação

Para se concatenar duas *"strings"* de caracteres, utiliza-se o operador //. A sintaxe de utilização é

```
<expressão> ::= <expressão-1> // <expressão-2>
```

Semântica:

1. <expressão-1> e <expressão-2> são expressões de caracteres;
2. <expressão> é uma *"string"* de caracteres, a qual é formada pelos caracteres presentes em <expressão-1>, seguidos dos caracteres em <expressão-2>; o tamanho de <expressão> é a soma dos tamanhos de <expressão-1> e <expressão-2>. Se <expressão> for atribuída a uma variável do tipo CHARACTER*<tamanho>, tal que <tamanho> for menor do que o tamanho de <expressão>, então apenas <tamanho> caracteres serão armazenados na variável.

4.2.6 Ordem de precedência de operadores

Ao se utilizar os operadores descritos nesse capítulo, bem como aqueles definidos pelo usuário (ver §7.4.1), é seguida uma ordem de precedência, definida pela linguagem. A tabela 4.3 apresenta essa ordem.

4.3 Alocação dinâmica de dados

A linguagem FORTRAN 90 introduziu mecanismos que permitem a alocação *dinâmica* de dados na memória. Esse recurso é extremamente importante para a manipulação de certas estruturas de dados, além de permitir a construção de bibliotecas de subprogramas que trabalham com a sua própria área de dados, sem necessitar a intervenção do usuário.

Em FORTRAN 90, dois tipos de variáveis podem ser alocadas dinamicamente: arranjos declarados com o atributo ALLOCATABLE e variáveis declaradas com atributo POINTER. Para se alocar na memória variáveis, utiliza-se o comando ALLOCATE, cuja sintaxe é

```
ALLOCATE( <variável> [ ( <seção> [,... [<seção>] ] ) ] )

<seção> ::= <número-de-elementos> | <índice-inicial>:<índice-final>
```

Semântica:

1. <variável> é o nome de uma variável declarada com o atributo POINTER, ou um arranjo declarado com o atributo ALLOCATABLE;
2. <seção> pode ser uma dentre duas opções:

Expressão	Operador	Operação	Ordem de precedência
Def. pelo usuário	Unário	–	1
Aritmética	**	potenciação	2
	*, /	multiplicação, divisão	3
	+, -	adição, subtração (negação numérica)	4
Caracter	//	concatenação	5
Relacional	==, .EQ.	igualdade	6
	/=, .NE.	desigualdade	6
	<, .LT.	menor do que	6
	>, .GT.	maior do que	6
	<=, .LE.	menor ou igual do que	6
	>=, .GE.	maior ou igual do que	6
Lógica	.NOT.	negação lógica	7
	.AND.	multiplicação lógica	8
	.OR.	adição lógica	9
	.EQV.	equivalência	10
	.NEQV.	não-equivalência	10
Def. pelo usuário	Binário	–	11

Tabela 4.3: Ordem de precedência de operadores; o nível 1 é o de maior precedência.

(a) `<número-de-elementos>` é um número inteiro, que indica o número de elementos de uma dimensão. Nesse caso, os elementos ao longo daquela dimensão serão acessados por índices entre 1 e `<número-de-elementos>`;

(b) `<índice-inicial>:<índice-final>` indica que os elementos de uma dimensão serão acessados por índices compreendidos entre `<índice-inicial>` e `<índice-final>`. O número de elementos daquela dimensão será igual a `<índice-final>` − `<índice-inicial>` + 1.

Observe que uma variável com o atributo `POINTER` pode simplesmente apontar para uma outra variável, não havendo a necessidade de se alocar um espaço de memória para tal variável. Porém, se se desejar atribuir valores a um apontador, então ele deve ser alocado anteriormente.

Sempre que uma variável tiver sido alocada dinamicamente, é interessante que ela seja removida da memória, após não ser mais necessário seu uso. Para isso, utiliza-se o comando `DEALLOCATE`, cuja sintaxe é

```
DEALLOCATE( <variável> )
```

Semântica:

1. `<variável>` é o nome de uma variável declarada com o atributo `POINTER`, ou um arranjo declarado com o atributo `ALLOCATABLE`.

Em certos computadores, pode acontecer um erro fatal no programa caso, ao terminar sua execução, existam áreas de memória, alocadas dinamicamente, que não tenham sido desalocadas.

Além desses dois comandos, a linguagem oferece ainda o intrínseco `ALLOCATED` (ver 8.3.1), o qual permite verificar se uma variável foi alocada. O exemplo abaixo ilustra o uso desses comandos.

Exemplo 4.5 *Utilizando alocação dinâmica de memória.*

```
PROGRAM TESTE
INTEGER :: M,N
REAL :: SOMA
    PRINT *,'M = ?, N = ?'
```

```
    READ *,M,N
    SOMA = CALCULA_SOMA(M,N)
    PRINT *,'SOMA = ',SOMA
CONTAINS
    INTEGER FUNCTION CALCULA_SOMA(M,N) RESULT (X)
    INTEGER, INTENT(IN) :: M, N
    REAL, DIMENSION(:,:), ALLOCATABLE :: ARR
    INTEGER :: I,J
        ALLOCATE(ARR(0:M-1,0:N-1))

        DO J=0,N-1
            DO I=0,M-1
                ARR(I,J) = I+J*M
            END DO
        END DO

        X = SUM(SQRT(ARR))

        DEALLOCATE(ARR)
    END FUNCTION CALCULA_SOMA
END PROGRAM TESTE
```

*O programa acima lê duas variáveis, M e N, as quais são transmitidas à função CALCULA_SOMA. Dentro dessa função, é alocado um arranjo ARR, com duas dimensões, através do comando ALLOCATE(ARR(0:M-1, 0:N-1)). A cada elemento do arranjo ARR é atribuído o resultado da expressão I+J*M, e X recebe o valor da soma das raízes quadradas de cada elemento do arranjo. Ao final da função, o arranjo ARR é desalocado. Se, durante a execução desse programa, os valores de M e N forem 4 e 6, respectivamente, então o programa produzirá a saída abaixo:*

```
SOMA =      75.00000
```

4.4 Operações envolvendo ponteiros

Uma variável declarada com o atributo POINTER – um *ponteiro* – pode tanto apontar para uma outra variável – ambas de mesmo tipo – ou, então, ser usada para alocar dinamicamente um espaço de memória (com os comandos ALLOCATE e DEALLOCATE). Nesse caso, o ponteiro não só contém o endereço daquele espaço de memória, mas também contém valores.

Devido à essa característica "dual" de ponteiros, é *extremamente* aconselhável que, antes de se utilizar um ponteiro, ele não aponte para qualquer posição de memória. Isso é feito através do comando NULLIFY, cuja sintaxe é

```
NULLIFY( <variável-ponteiro> )
```

Semântica:

1. <variável-ponteiro> é o nome de uma variável declarada com o atributo POINTER.

O exemplo abaixo mostra uma utilização de ponteiros.

Exemplo 4.6 *Utilizando ponteiros.*

```
PROGRAM TESTE
IMPLICIT NONE
INTEGER, POINTER :: P, Q
INTEGER, TARGET :: I = -10
        NULLIFY(Q)
        Q => I
        IF (ASSOCIATED(Q,TARGET=I)) THEN
                PRINT *,'Q APONTA PARA I, Q=',Q
```

```
        ELSE
                PRINT *,'Q NÃO APONTA PARA I'
        END IF

        NULLIFY(P)
        ALLOCATE(P)
        P = 5
        PRINT *,'P= ',P

        P => Q
        PRINT *,'P= ',P
END PROGRAM TESTE
```

O programa acima declara duas variáveis de tipo INTEGER, P e Q, com o atributo POINTER; além de uma variável do mesmo tipo, I, declarada com o atributo TARGET (o que indica que ela poderá ser "apontada" por uma variável de tipo INTEGER e com o atributo POINTER). A variável I é inicializada com o valor -10. Após, o ponteiro Q é anulado, através do comando NULLIFY(Q) e, na seqüência, ele passa a apontar para I, com o comando Q => I.

Em seguida, é feito um teste para se verificar se o ponteiro Q aponta para a variável I, através do uso do intrínseco ASSOCIATED(Q,TARGET=I) (ver §8.5); caso seja verdadeiro (e, obviamente, o é), é escrito o texto Q APONTA PARA I, Q=. Observe que foi utilizado o próprio ponteiro Q no comando PRINT. Como Q aponta para I, o valor a ser impresso é aquele armazenado em I, qual seja, -10.

Os comandos seguintes demonstram a característica dual de um ponteiro. Observe que, após P ser anulado, ele é utilizado como argumento do comando ALLOCATE, efetivamente alocando um espaço de memória com 32 "bits" *(já que variáveis de tipo INTEGER tem tamanho* "default" *de* 4 "bytes", *i.e. KIND=4). Uma vez feito isso, pode ser atribuído um valor a P, através do comando P = 5; se o comando ALLOCATE(P) não tivesse sido executado (ou não estivesse presente), essa atribuição não poderia ser feita (ocorrendo um erro de execução do tipo "acesso inválido à memória"*[1].

Após a impressão do valor de P, esse ponteiro passa a apontar para outra posição de memória, através do comando P => Q. São dois os efeitos desse comando:

1. *O valor anterior (5) armazenado em P não pode mais ser recuperado;*
2. *Como Q aponta para I, então também ponteiro P aponta para I.*

Assim, o valor impresso de P, ao final, é -10, que corresponde ao valor de I. Abaixo são mostrados os valores impressos pelo programa.

```
Q APONTA PARA I, Q=          -10
P=             5
P=           -10
```

4.5 Operações envolvendo arranjos

Um das novas características introduzidas pela linguagem FORTRAN 90 foi a possibilidade de se operar com arranjos como se fossem variáveis escalares, i.e., certas operações podem ser executadas sem a necessidade de se utilizar laços de repetição (ver §5.4). Isso aumenta a legibilidade de um programa em FORTRAN 90, torna-os menores (em termos do número de linhas de código fonte) e permite que compiladores otimizadores e/ou específicos para computadores de arquitetura vetorial e/ou paralela (como os fabricados pela CRAY, NEC, IBM e HP, dentre outros) possam gerar um código executável extremamente eficiente.

Não obstante tal nova característica, ainda deve-se tomar certos cuidados ao se acessar arranjos em FORTRAN 90, da mesma forma que na versão anterior (Fortran 77), como descrito na seção 4.5.1. Após, apresentaremos os mecanismos com os quais se podem operar com arranjos em FORTRAN 90.

[1] Em ambientes UNIX e assemelhados, o erro é *Segmentation fault (core dumped)*; em ambientes MS WINDOWS, o erro é *"Access violation"*.

4.5.1 Armazenamento de um arranjo na memória

Um arranjo bi-dimensional em FORTRAN 90 é organizado na memória do computador *por colunas*. Para arranjos multi-dimensionais, diz-se que a *primeira* dimensão varia *mais rapidamente* do que a *última*. Considere, por exemplo, a declaração do arranjo

```
REAL, DIMENSION(-1:2,2) :: B
```

o qual contém quatro linhas (cujos índices são -1, 0, 1 e 2) por duas colunas (com índices 1 e 2), totalizando 8 elementos. Então, os elementos estarão dispostos na memória da seguinte forma:

elemento	B(-1,1)	B(0,1)	B(1,1)	B(2,1)	B(-1,2)	B(0,2)	B(1,2)	B(2,2)
posição	i	$i+1$	$i+2$	$i+3$	$i+4$	$i+5$	$i+6$	$i+7$

onde i é o endereço inicial do arranjo na memória (determinado pelo compilador e/ou sistema de gerenciamento de memória do computador). Note que os primeiros quatro elementos correspondem à coluna 1; e os restantes quatro à coluna 2.

Devido à forma como os elementos de um arranjo são organizados na memória, em computadores que disponham de memória *"cache"*, é importante que se acesse um arranjo bi-dimensional pelas colunas (ou, se multi-dimensional, a partir da última dimensão), de forma a se minimizar os efeitos de *"cache-miss"*, que podem elevar consideravelmente o tempo de execução de um programa. Uma breve explanação do funcionamento da memória *"cache"* será feita a seguir, juntamente com um exemplo.

Basicamente, a memória *"cache"* é uma memória de pequena capacidade (não mais do que 4 *"megabytes"*) e de baixo tempo de acesso – e, portanto, mais cara – se comparada com a memória principal. O termo *"cache"* vem do francês e significa "escondido"; ela é uma memória que todo programa utiliza, porém o programador não tem controle sobre ela.

Estatisticamente, já se mostrou que a maior parte do tempo de execução de programas de cunho científico está concentrada na execução de laços de repetição (ver §5.4), acessando elementos de arranjos. A memória *"cache"* surgiu para se aumentar a velocidade de tais programas. Como, ao se acessar um elemento (i, j) de um arranjo, muito provavelmente iremos acessar o próximo elemento do mesmo arranjo, o que o computador faz é buscar da memória um conjunto de elementos a cada vez (por exemplo, 8 colunas a cada vez, para um programa em FORTRAN 90), armazenando-os na memória *"cache"*. Assim, se o acesso ao arranjo for feito por colunas, muito provavelmente o próximo elemento a ser acessado (por exemplo, $(i+1, j)$) já estará presente na memória *"cache"* e, portanto, será acessado mais rapidamente, já que essa memória tem um tempo de acesso menor.

Obviamente, quando se tenta acessar um elemento que não se encontra mais na *"cache"* – o que se chama de *"cache-miss"* – então aqueles elementos já transferidos da memória principal (em nosso exemplo, nossas 8 colunas) terão de ser desprezados, e um novo acesso de mais 8 colunas à memória principal será feito, substituindo aquelas na memória *"cache"*. Se os elementos na memória *"cache"* tiverem sido modificados, então os elementos a eles correspondentes, na memória principal, terão de ser substituídos pelos da *"cache"*.

É fácil perceber que, para um programa em FORTRAN 90, se não acessarmos um arranjo pelas suas colunas, então o número de *"cache-misses"* será alto, e o tempo de acesso aos elementos do arranjo irá ser excessivo. O exemplo abaixo demonstra esse efeito.

Exemplo 4.7 *O efeito da memória* "cache" *em um programa* FORTRAN 90.

```
PROGRAM TESTA_CACHE
INTEGER :: N,I,T0,T1,T_COLUNA,T_LINHA,USEC
REAL, DIMENSION(:,:), ALLOCATABLE :: A
EXTERNAL USEC
        PRINT *,'N=?'
        READ *,N
        ALLOCATE(A(N,N))
        T0 = USEC()
        DO J=1,N
```

```
            DO I=1,N
                    A(I,J) = LOG(REAL((J-1)*N+I))
            END DO
        END DO
        T1 = USEC()
        T_COLUNA = T1-T0
        DEALLOCATE(A)
        ALLOCATE(A(N,N))
        T0 = USEC()
        DO I=1,N
            DO J=1,N
                    A(I,J) = LOG(REAL((J-1)*N+I))
            END DO
        END DO
        T1 = USEC()
        T_LINHA = T1-T0
        DEALLOCATE(A)
        WRITE(*,'("T_COLUNA=",I32," microsegundos")')T_COLUNA
        WRITE(*,'("T_LINHA =",I32," microsegundos")')T_LINHA
END PROGRAM TESTA_CACHE
```

O programa acima acessa o arranjo A por colunas, inicialmente, e por linhas. O arranjo A é alocado dinamicamente, de forma a facilitar a execução do programa para qualquer tamanho de arranjo, N; ele também é desalocado ao final de cada conjunto de laços a fim de garantir que ele não exista mais na memória (nem tampouco na memória "cache"*). Os resultados obtidos com ele são tabulados na tabela 4.4.*

N	T_COLUNA (μs)	T_LINHA (μs)	T_COLUNA/T_LINHA%
1000	280000	330000	85%
2000	1180000	1620000	73%
4000	4570000	7580000	60%

Tabela 4.4: Efeitos da memória *"cache"* nos tempos de execução para acesso de um arranjo por coluna e por linha, em um computador AMD K6 500MHz, usando o compilador PGI *pgf90*, sem otimização.

Observe na tabela 4.4 que o tempo de acesso por coluna é sempre inferior ao tempo de acesso por linha e que, à medida que N cresce, maior é a perda com o acesso por linha – isso ocorre por duas razões: maior é o número de *"cache-misses"* e maior é a quantidade de dados desperdiçada a cada *"cache-miss"*.

4.5.2 Acessando seções de um arranjo

Em FORTRAN 90, é possível efetuar operações com *seções* de arranjos, através do uso do operador :. O caracter : já foi apresentado anteriormente, ao se apresentar a declaração um arranjo através do atributo DIMENSION (ver 3.6.6). Da mesma forma que se pode declarar os índices inicial e final de uma dimensão de um arranjo, também pode-se *acessar* uma seção de um arranjo. A sintaxe é a seguinte:

```
<arranjo>( <seção> [, ...[ <seção> ] ] )

<seção> ::= <índice> | <lista-de-índices> | : |
            <índice-inicial>:<índice-final> |
            <índice-inicial>:<índice-final>:<incremento>
```

Semântica:

1. <arranjo> é o nome de uma variável declarada com o atributo DIMENSION;
2. <seção> pode ser uma dentre quatro opções:
   (a) <índice> é um número inteiro, cujo valor situa-se dentro do intervalo declarado (implícita ou explicitamente) de variação dos índices numa das dimensões do arranjo. Apenas um elemento de uma dimensão será acessado;
   (b) <lista-de-índices> é um arranjo inteiro, cujos elementos situam-se dentro do intervalo declarado (implícita ou explicitamente) de variação dos índices numa das dimensões do arranjo. Serão acessados tantos elementos na dimensão quanto o número de elementos em <lista-de-índices>;
   (c) : indica que todos os elementos numa dimensão serão acessados;
   (d) <índice-inicial>:<índice-final> indica que os elementos cujos índices estiverem compreendidos entre <índice-inicial> e <índice-final>, numa dimensão, serão acessados;
   (e) <índice-inicial>:<índice-final>:<incremento> indica que os elementos <índice-inicial>, <índice-inicial>+<incremento>, <índice-inicial>+2×<incremento>, ..., <índice-final>, numa dimensão, serão acessados. Se <índice-final>−<índice-inicial> não for um múltiplo inteiro de <incremento>, então o último elemento a ser acessado naquela dimensão terá índice menor do que <índice-final>.

O exemplo abaixo apresenta o uso de seções de arranjos.

Exemplo 4.8 *Acessando seções de arranjos.*

```
PROGRAM TESTE
REAL, DIMENSION(:,:), ALLOCATABLE :: ARR
INTEGER :: N,I,J
    PRINT *,'N=?'
    READ *,N
    ALLOCATE(ARR(N,4))
    CALL RANDOM_NUMBER(ARR(:,1:3))   ! Atribui valores pseudo-aleatórios a ARR
    ARR(:,4) = 0.0

    DO J=1,3
        IF (PRODUCT(ARR(:,J))<=0.5) THEN
            ARR(:,4) = ARR(:,4) + ARR(:,J)
        END IF
    END DO

    DO I=1,N
        WRITE(*,*)(ARR(I,J),J=1,4)
    END DO

    DEALLOCATE(ARR)
END PROGRAM TESTE
```

O programa acima inicializa as colunas de 1 a 3 com números aleatórios, através do comando CALL RANDOM_NUMBER(ARR(:,1:3)). Observe que a primeira dimensão é acessada por :, o que significa todos os elementos daquela dimensão; e que a segunda dimensão é acessada na forma 1:3, indicando que as colunas 1, 2 e 3 serão acessadas. Após, o valor 0.0 é atribuído à quarta coluna. O programa procede verificando, para cada uma das três primeiras colunas, se o produto dos elementos de cada coluna é menor do que 0.5; se tal condição for satisfeita, então a coluna J é adicionada à coluna 4. Um possível resultado desse programa segue abaixo.

```
0.9079230       0.7973146       0.3206477       2.025885
0.1906921       0.6368300       0.4481761       1.275698
6.7165293E-02   0.5887827       0.6579502       1.313898
0.8000845       0.1590656       0.6075459       1.566696
```

4.5.3 Manipulação condicional de um arranjo (WHERE..END WHERE)

A linguagem FORTRAN 90 introduziu também a possibilidade de se efetuar a manipulação de um arranjo de forma condicional, sem que necessariamente se acessem elementos individuais do arranjo. Isso é feito usando-se o comando WHERE...END WHERE, cuja sintaxe é a seguinte:

```
WHERE ( <expressão-lógica-arranjo> )

    <comandos-verdadeiro>

[ ELSEWHERE

    <comandos-falso>
]
END WHERE
```

Semântica:

1. `<expressão-lógica-arranjo>` é uma expressão lógica envolvendo pelo menos um arranjo;
2. `<comandos-verdadeiro>` é um conjunto de comandos da linguagem, nos quais um ou mais arranjos serão possivelmente manipulados, desde que `<expressão-lógica-arranjo>` seja satisfeita;
3. ELSEWHERE é uma palavra-chave opcional. Caso esteja presente, os `<comandos-falso>` serão executados, desde que `<expressão-lógica-arranjo>` não seja satisfeita.

O exemplo abaixo ilustra o uso desse comando.

Exemplo 4.9 *Usando o comando WHERE...END WHERE*

```
PROGRAM TESTE
INTEGER, DIMENSION(0:3,0:3) :: MASCARA
REAL, DIMENSION(4,4) :: ARR
INTEGER :: N,I,J
    CALL RANDOM_NUMBER(ARR)   ! Atribui valores pseudo-aleatórios a ARR

    WRITE(*,'(" MATRIZ ALEATóRIA:")')
    DO I=1,4
        WRITE(*,'(4(F4.2,1X))')(ARR(I,J),J=1,4)
    END DO

    WHERE (ARR<0.5)
        MASCARA = 1
    ELSEWHERE
        MASCARA = 0
    END WHERE

    WRITE(*,'(/" MATRIZ BINÁRIA:")')
    DO I=0,3
        WRITE(*,'(4I2)')(MASCARA(I,J),J=0,3)
    END DO
END PROGRAM TESTE
```

O programa acima calcula uma matriz MASCARA, de tipo INTEGER, a partir dos valores da matriz ARR. Se algum elemento ARR(I,J) é menor do que 0.5, então o elemento correspondente MASCARA(I,J) receberá o valor 1; caso contrário, esse elemento receberá o valor 0. Observe que ambos o arranjo MASCARA deve ter o mesmo número de dimensões de A e o mesmo número de elementos por dimensão, apesar de que os índices podem ser diferentes; nesse caso, é feita uma associação elemento-a-elemento entre cada arranjo. Um possível resultado desse programa segue abaixo.

MATRIZ ALEATÓRIA:
0.91 0.80 0.32 0.76
0.19 0.64 0.45 0.44
0.07 0.59 0.66 0.46
0.80 0.16 0.61 0.70

MATRIZ BINÁRIA:
0 0 1 0
1 0 1 1
1 0 0 1
0 1 0 0

Capítulo 5

Controle de fluxo de execução

Neste capítulo, serão apresentados os comandos disponíveis na linguagem FORTRAN 90 que permitem controlar o fluxo de execução de um programa.

5.1 Desvio incondicional

5.1.1 GO TO

O comando `GO TO <rótulo>` faz com que o fluxo de execução do programa seja desviado incondicionalmente para o comando identificao por `<rótulo>`, o qual é um número inteiro, colocado à frente do comando e separado do mesmo por no mínimo um espaço em branco, para um código-fonte em formato livre, ou nas colunas 1 a 6 de uma linha, em formato fixo.

Exemplo 5.1 *Uso do `GO TO`*

```
      I = 1
      J = I+1
      K = I+J
      GO TO 100
      K = 1  ! Esse comando nunca será executado
  100 PRINT *,'K=',K
```

Nesse exemplo, K terá o valor 3, pois a atribuição K = 1 jamais será executada. Em situações como essa, compiladores otimizadores não irão gerar código executável para tal comando, uma vez que se garante que ele nunca será executado.

O comando `GO TO` jamais deve ser usado para se desviar o fluxo de execução de um programa, se tal desvio pode ser obtido com outras construções existentes na linguagem, como, por exemplo, comandos de repetição. É admitido, no entanto, que se utilize o `GO TO` em situações como tratamento de exceções (por exemplo, para evitar uma possível divisão por zero), ou para consistência de dados, desde que o fluxo de execução do programa seja desviado apenas para um *único* comando, *abaixo* do `GO TO`. Com essas regras, é possível utilizar o `GO TO` de forma a manter a legibilidade do programa. O exemplo abaixo ilustra o exposto:

Exemplo 5.2 *Uso do `GO TO` para tratamento de exceção*

```
PROGRAM EXEMPLO
    ...
    AR = MATMUL(A,R) ! Calcula o produto matriz-vetor entre a matriz A e o vetor R
    RR = DOT_PRODUCT(R,R) ! Calcula o produto interno de R com ele próprio
    RAR = DOT_PRODUCT(R,AR) ! Calcula o produto interno de R com AR
    IF (RAR == 0.0) THEN
        ERRONUM = 1
```

```
        GO TO 9999
      END IF
      ALPHA = RR/RAR
      X = X + ALPHA*D
      R = R - ALPHA*AR
      ...
      RRO = RR  ! Salva o valor anterior de RR em RRO
      RR = DOT_PRODUCT(R,R)
      IF (RR == 0.0) THEN
          ERRONUM = 2
          GO TO 9999
      END IF
      BETA = RR/RRO
      ...
9999  PRINT *,'X=',X,' ERRONUM=',ERRONUM
END PROGRAM EXEMPLO
```

Observe que ambos os comandos GO TO desviam o fluxo de execução do programa para apenas um comando, identificado pelo rótulo 9999, e esse comando encontra-se abaixo dos demais.

5.2 Parada incondicional

O comando STOP permite parar a execução de um programa, e pode aparecer inúmeras vezes, em diferentes seções de um programa, incluindo subprogramas. A sintaxe do comando é

```
STOP [ <constante-numérica> | <string-caracteres> ]
```

Semântica:

1. O comando STOP, ao ser executado, interrompe a execução do programa. Se <constante-numérica> ou <string-caracteres> estiverem presentes, então seus valores serão impressos antes de encerrar-se a execução do programa.

5.3 Decisão

5.3.1 IF...END IF

Um bloco IF...END IF permite que se desvie o fluxo de execução de um programa de forma *condicional*. Ele apresenta, em sua forma completa, duas seções de comandos, chamadas de *verdadeira* e *falsa*, as quais correspondem aos valores que a condição de decisão (lógica) pode assumir. A seção falsa pode ser omitida.

```
[<nome-if>:] IF (<expressão-lógica>) THEN
                 <comando-verdadeiro-1>
                         ...
                 <comando-verdadeiro-n>
             [ ELSE [nome-if]
                 <comando-falso-1>
                         ...
                 <comando-falso-n> ]
             [ ELSE IF (<expressão-lógica>) THEN [nome-if]
                 <comando-falso-1>
                         ...
                 <comando-falso-n> ]
             END IF [<nome-if>]
```

Semântica:

1. Caso <expressão-lógica> resulte em um valor .TRUE., então serão executados os comandos presentes na sua seção *verdadeira*, isto é, entre as palavras THEN e ELSE ou END IF;

2. Caso <expressão-lógica> resulte em um valor .FALSE., então:

   (a) Se a palavra ELSE existir, serão executados os comandos presentes presentes na sua seção *falsa*, isto é, entre as palavras ELSE e END IF;

   (b) Caso contrário, será executado o comando imediatamente posterior à palavra END IF.

5.3.1.1 Aninhamento e encadeamento de blocos IF...END IF

É possível que um bloco IF...END IF tenha como um ou mais dos comandos presentes nas suas seções *verdadeira* e *falsa* outros comandos IF...END IF:

Exemplo 5.3 *Aninhamento de blocos IF...END IF*

```
IF (I==J) THEN
   H=I+2
ELSE
    F=H+3
    IF (I<J) THEN
        H=J-2
        L=2*H+5
    END IF
END IF
```

Observe que o comando IF (I<J) THEN, presente na seção falsa do comando IF (I==J) THEN, é fechado por um comando END IF.

Um detalhe a ser considerado, no entanto, é quanto ao fechamento de blocos IF...END IF quando imediatamente após a palavra ELSE existir um outro IF...END IF, como no exemplo abaixo:

Exemplo 5.4 *Encadeamento de blocos IF...END IF*

```
IF (X==0) THEN
    SINAL = 0
ELSE IF (X<0) THEN
    SINAL = -1
ELSE
    SINAL = 1
END IF
```

Esse código calcula o valor da função sinal de X. Note que apenas um END IF deve ser usado, nesse caso, em contraste com o exemplo anterior.

5.3.2 IF lógico

Essa forma do comando IF, herdada das versões anteriores da linguagem, é equivalente a um bloco IF...END IF que contenha apenas a seção verdadeira. Sua sintaxe é a seguinte:

```
IF (<expressão-lógica>) <comando>
```

Semântica:

1. Caso <expressão-lógica> resulte em um valor .TRUE., então será executado <comando>, o qual pode ser qualquer comando da linguagem, com exceção de IF, END, END SUBROUTINE, END FUNCTION e END PROGRAM;

2. Caso <expressão-lógica> resulte em um valor .FALSE., então será executado o comando imediatamente posterior.

Observe que, se <expressão-lógica> contiver uma chamada a uma função que altera o valor de uma variável usada dentro de <comando>, então poderá ocorrer um efeito colateral.

Exemplo 5.5 *Uso do IF lógico*

```
IF ((K>KMAX).OR.(ABS(X)<1.0E-5)) FIM = .TRUE.
PRINT *,'K=',K
```

5.3.3 IF aritmético

Essa forma do comando IF foi herdada das versões anteriores da linguagem. Sua sintaxe é a seguinte:

```
IF (<expressão-aritmética>) <rótulo-menor>,<rótulo-igual>,<rótulo-maior>
```

Semântica:

1. Caso <expressão-aritmética> seja *menor do que 0*, o fluxo de execução será desviado para o comando identificado por <rótulo-menor>; se for *igual a 0*, o fluxo de execução será desviado para o comando identificado por <rótulo-igual>; e, se for *maior do que 0*, o fluxo de execução será desviado para o comando identificado por <rótulo-maior>.

Exemplo 5.6 *Uso do IF aritmético*

```
      IF (X) 10,20,30
10    SINAL = -1
      GO TO 40
20    SINAL = 0
      GO TO 40
30    SINAL = 1
40    PRINT *,'SINAL=',SINAL
```

O trecho de código acima calcula o valor da função sinal de X; observe que é necessário utilizar o comando GO TO para desviar para o comando de impressão, caso contrário o comando SINAL = 1 sempre seria executado.

O uso do IF aritmético não é recomendado, pois pode causar dificuldades na compreensão do programa. Ele sempre pode ser substituído por blocos IF...END IF em seqüência ou encadeados, como apresentado no exemplo 5.4. Além disso, esse comando é obsoleto, e não será mais permitido em versões posteriores da linguagem.

5.3.4 SELECT CASE...END SELECT

O comando SELECT CASE...END SELECT é equivalente a usarmos blocos IF...END IF encadeados; porém sua utilização favorece a legibilidade do código. Sua estrutura é a seguinte:

```
[<nome-select>:] SELECT CASE ( <expressão> )
                    CASE ( <valor-1> )
                        <comando-1-1>
                                ...
                        <comando-1-n>

                    CASE ( <valor-2> )
                        <comando-2-1>
                                ...
                        <comando-2-n>
```

```
                ...

            [ CASE DEFAULT
                <comando-default-1>
                        ...
                <comando-default-n> ]

    END SELECT [<nome-select>]
```

onde `<valor-1>`, `<valor-2>`, ... são quaisquer expressões admitidas na linguagem, de acordo com o *tipo* de `<expressão>`:

- um valor simples, como `CASE ( 'A' )`;
- um intervalo, como `CASE ( 1:10 )`;
- um conjunto de valores, separados por vírgula, como `CASE ( 'A', 'a' )`, `CASE ( 1:10 , 20:30 )`.

Semântica:

1. Caso `<expressão>` contenha um valor dentre `<valor-1>`, `<valor-2>`, ..., então serão executados os comandos correspondentes àquele valor; após esses comandos terem sido executados, o fluxo de execução é desviado para o próximo comando imediatamente posterior à palavra `END SELECT`;
2. Caso contrário:
   (a) se a cláusula `CASE DEFAULT` estiver presente, serão executados os comandos a ela correspondentes; após esses comandos terem sido executados, o fluxo de execução é desviado para o próximo comando imediatamente posterior à palavra `END SELECT`.
   (b) caso contrário, o fluxo de execução é desviado para o próximo comando imediatamente posterior à palavra `END SELECT`.

Exemplo 5.7 *Uso do SELECT...END SELECT*

```
CHARACTER :: TECLA
LOGICAL :: CONTINUA = .TRUE.

DO WHILE (CONTINUA)
    PRINT *,'(A)dicionar, (R)emover, (F)inalizar?'
    READ *,TECLA
    SELECT CASE (TECLA)
        CASE ( 'A', 'a' ) ! adicionar item
            PRINT *,'Código do item?'
            READ *,CODITEM
            ! processa adição do item
        CASE ( 'R', 'r' ) ! remover item
            PRINT *,'Código do item?'
            READ *,CODITEM
            ! processa remoção do item
        CASE ( 'F', 'f' ) ! finaliza operação
            PRINT *,'Sistema sendo encerrado, aguarde...'
            CONTINUA = .FALSE.
        CASE DEFAULT
            PRINT *,'Opção inválida!'
    END SELECT
END DO
```

O código acima é típico de um diálogo entre um sistema e um operador; enquanto CONTINUA for .TRUE., o programa pede que seja digitada uma letra, correspondente às opções possíveis. Se o caracter digitado for admitido (não importando se foi em letras maiúsculas ou minúsculas), então o código correspondente à opção é selecionado pelo SELECT...END SELECT. Se a letra digitada não é uma dentre A, a, R, r, F, f, então a cláusula CASE DEFAULT será ativada - no caso, será impressa uma mensagem de erro. Mais detalhes sobre o comando DO WHILE...END DO são dados em §5.4.2.

5.4 Repetição

5.4.1 DO...END DO finito

Nesta forma, o bloco de comandos DO...END DO permite que sejam feitas um número finito de *iterações*; a cada iteração o conjunto de comandos listado dentro do bloco será executado.

```
[<nome-do>:] DO <variável-de-controle> = <valor-inicial>, <valor-final> [,<valor-passo>]
               <comando-1>
               ...
               <comando-n>
             END DO [<nome-do>]
```

Semântica:

1. A <variável-de-controle> é inicializada com o valor da expressão <valor-inicial>;
2. Se
   (a) <variável-de-controle> for um valor contido entre <valor-inicial> e <valor-final>, então os comandos <comando-1> a <comando-n> são executados;
   (b) caso contrário, o próximo comando imediatamente posterior a END DO será executado;
3. O <valor-passo> (ou o valor 1, caso <valor-passo> seja omitido) é adicionado à <variável-de-controle>, e o fluxo de execução do programa é desviado para o teste acima.

Se um dos comandos executados na iteração for EXIT (§5.4.4) ou CYCLE (ver §5.4.5) , então o fluxo de execução será alterado.

Exemplo 5.8 *Uso do DO...END DO finito*

```
INTEGER, PARAMETER :: M = 6
INTEGER, PARAMETER :: N = 2
INTEGER :: I, J, K
REAL, DIMENSION(M,N) :: A
REAL, DIMENSION(M) :: U = 0.0
REAL, DIMENSION(N) :: V

K = 1
inicializa: DO J = 1, N
                V(J) = J
                DO I = 1, M
                    A(I,J) = K
                    K = K+1
                END DO
            END DO inicializa
PRINT *,'V=',V
PRINT *,'A='
imprime: DO I=1,M
            WRITE(*,*)(A(I,J),J=1,N)
         END DO imprime
```

```
DO J = 1, N
    altera: DO I = 1, M, 2
                U(I) = U(I) + A(I,J)*V(J)
            END DO altera
END DO
PRINT *,'U=',U
```

Nesse exemplo, o vetor V é inicializado como V(1)=1, V(2)=2 e a matriz A é inicializada como A(1,1)=1, A(2,1)=2, ..., A(6,1)=6, A(1,2)=7, A(2,2)=8, ..., A(6,2)=12. O vetor U é inicializado com 0.0, na sua declaração, e apenas os seus elementos U(1), U(3), U(5) serão modificados dentro do laço altera, uma vez que esse laço tem como <valor-passo>=2. Ao final, o vetor U terá os seguintes valores:

U(1)=15.0, U(2)=0.0, U(3)=21.0, U(4)=0.0, U(5)=27.0, U(6)=0.0.

Observe, também, o uso do laço imprime, o qual imprime os elementos da matriz, por linhas, combinado com um laço implícito no comando WRITE, para imprimir todas as colunas de uma linha, conforme apresentado em §6.1.3.

5.4.2 DO WHILE...END DO

O bloco de comandos DO WHILE...END DO permite que sejam feitas um número finito de iterações, *enquanto* uma *condição lógica* for atendida; a cada iteração o conjunto de comandos listado dentro do bloco será executado.

```
[<nome-do>:] DO WHILE ( <condição-lógica> )
                <comando-1>
                ...
                <comando-n>
             END DO [<nome-do>]
```

Semântica:

1. Se
   (a) <condição-lógica> for avaliada como .TRUE., então os comandos <comando-1> a <comando-n> são executados;
   (b) caso contrário, o próximo comando imediatamente posterior a END DO será executado.

Se um dos comandos executados na iteração for EXIT (§5.4.4) ou CYCLE (ver §5.4.5) , então o fluxo de execução será alterado.

Exemplo 5.9 *Uso do DO WHILE...END DO*

```
INTEGER, PARAMETER :: KMAX = 100
INTEGER :: K = 0
REAL :: A = 0.5
REAL :: X = 1.0
REAL :: XO
LOGICAL :: CONVERGIU = .FALSE.

repete: DO WHILE ((K<=KMAX).AND.(.NOT. CONVERGIU))
            XO = X
            X = X-A*X
            CONVERGIU = ABS(X-XO)<1.0E-5
            K = K+1
        END DO repete
```

O laço DO WHILE...END DO contém uma <condição-lógica> composta: os comandos contidos dentro do laço serão continuamente executados enquanto não se exceder o máximo de iterações permitidas e não for satisfeito o critério de convergência, ABS(X-XO)<1.0E-5. Ao se executar esse trecho de programa, teremos os seguintes valores para as variáveis envolvidas: K=17, CONVERGIU=.TRUE., XO=1.525879E-05, X=7.629395E-06.

5.4.3 DO...END DO infinito

Nesta forma, o bloco de comandos DO...END DO permite que sejam feitas infinitas iterações; a cada iteração o conjunto de comandos listado dentro do bloco será executado.

```
[<nome-do>:] DO
                <comando-1>
                ...
                <comando-n>
             END DO [<nome-do>]
```

Semântica:

1. Os comandos <comando-1> a <comando-n> são executados até que o fluxo de execução do programa seja desviado para fora do bloco DO...END DO, através de um comando EXIT ou RETURN.

Se um dos comandos executados na iteração for EXIT (§5.4.4) ou CYCLE (ver §5.4.5) , então o fluxo de execução será alterado.

Exemplo 5.10 *Uso do DO...END DO infinito*

```
INTEGER, PARAMETER :: KMAX = 100
INTEGER :: K = 0
REAL :: A = 0.5
REAL :: X = 1.0
REAL :: XO
LOGICAL :: CONVERGIU = .FALSE.

repete: DO
            IF ((K>KMAX).OR.(CONVERGIU)) EXIT
            XO = X
            X = X-A*X
            CONVERGIU = ABS(X-XO)<1.0E-5
            K = K+1
        END DO repete
```

O código acima é equivalente ao apresentado no exemplo 5.9. No entanto, a fim de obtermos a mesma resposta obtida naquele exemplo, observe que a condição de parada foi negada logicamente, i.e., invertemos os operadores lógicos (de <= para > e removemos o .NOT.) e usamos .OR. ao invés de .AND.; ao se executar esse trecho de programa, teremos os seguintes valores para as variáveis envolvidas: K=17, CONVERGIU=.TRUE., XO=1.525879E-05, X=7.629395E-06, idênticos aos do exemplo 5.9.

5.4.4 EXIT

O comando EXIT faz com que o fluxo de execução do programa seja desviado para fora do bloco DO...END DO ou DO WHILE...END DO no qual o comando foi executado, isto é, nenhum comando presente entre EXIT e END DO é executado, e as iterações do laço são terminadas.

5.4.5 CYCLE

O comando CYCLE faz com que o fluxo de execução do programa, dentro de um bloco DO...END DO ou DO WHILE...END DO seja desviado para o comando END DO, isto é, nenhum comando presente entre CYCLE e END DO é executado, e uma nova iteração do laço será (possivelmente) executada.

Capítulo 6

Entrada e saída de dados

A linguagem FORTRAN 90 oferece um conjunto de comandos que permitem a leitura e escrita de dados, de e sobre diferentes dispositivos de entrada e saída (E/S), tais como teclado, monitor de vídeo, impressora, discos e fitas magnéticas. Intimamente associado a um dispositivo é a noção de *arquivo de dados*. Além disso, os dados podem ser formatados para que a leitura e/ou a escrita seja feita de uma determinada forma.

Para a linguagem, existem dois tipos de arquivos de dados: *internos* e *externos*. Por arquivo interno entende-se uma variável do tipo `CHARACTER` que contenha uma *"string"* de caracteres, os quais serão lidos ou sobre a qual serão escritos valores. Já um arquivo externo é um conjunto de *registros*, o qual existe como entidade separada do programa e é visível ao sistema operacional.

6.1 Comandos de E/S

Existem três comandos de leitura e escrita – `READ` (leitura), `PRINT` e `WRITE` (escrita). Além desses, existem comandos para se manipular arquivos externos de dados, como `OPEN`, `CLOSE`, `INQUIRE`, `REWIND`, `BACKSPACE` e `ENDFILE`.

Para se definir de qual dispositivo será feita uma operação de E/S, é necessário que se especifique o mesmo. Em FORTRAN 90, existem pré-definidos um dispositivo padrão de entrada - o *teclado* - e um de saída - o *monitor de vídeo*.

6.1.1 PRINT

O comando `PRINT` permite que se escreva um conjunto de valores no dispositivo padrão de saída. Sua sintaxe é a seguinte:

```
PRINT  * | <formato> | <nome-namelist>  [, <lista-nomes> ]

<formato> ::= <identificador-FORMAT> | ('<expressão-formato>') | <nome>
```

Semântica:

1. `*` indica que os dados serão escritos no dispositivo padrão de saída (monitor de vídeo) em *formato livre* (dependente de cada implementação);
2. `<formato>` indica como os dados serão formatados, podendo ser:
   (a) O número de um comando `FORMAT` (ver §6.2);
   (b) Uma `<expressão-formato>`, i.e. uma *"string"* de *caracteres de edição* (ver §6.2), englobada por apóstrofos e delimitada por abre e fecha parênteses;
   (c) Um `<nome>` de uma variável do tipo `CHARACTER` que contém uma `<expressão-formato>`;

3. <nome-namelist> é o nome de uma variável declarada com o atributo NAMELIST (ver §3.7.4), que contém uma lista de nomes de variáveis;

4. <lista-nomes> é uma lista de nomes de variáveis, com no mínimo um nome, separados por vírgulas. A lista de nomes pode ser especificada de forma implícita através de um comando DO. Se <nome-namelist> for especificado, então <lista-nomes> deve ser *omitida*.

Exemplo 6.1 *Comando PRINT*

```
PROGRAM PRINCIPAL
IMPLICIT NONE
INTEGER :: A=12323, B=64652545
REAL, DIMENSION(2,2) :: X
NAMELIST /DADOS/ B,A

    X(1,:) = (/ 1.45656, 3.2367 /)
    X(2,:) = (/ -9.435, 3.456 /)
    PRINT *,'A=',A,' B=',B
    PRINT '(I5,I7)',A,B
    PRINT DADOS
    PRINT *, ((X(I,J),J=1,2),I=1,2)
    PRINT '(2F8.5)', ((X(I,J),J=1,2),I=1,2)

END PROGRAM PRINCIPAL
```

O programa acima, ao ser executado, produziria o seguinte resultado:

```
 A=        12323  B=     64652545
12323*******
 &DADOS
 B =     64652545,
 A =        12323
 /
   1.456560        3.236700        -9.435000        3.456000
 1.45656 3.23670
-9.43500 3.45600
```

*Note que, na segunda linha impressa, o valor da variável B foi substituído por 7 caracteres *; isso indica que o formato utilizado, I7, o qual especifica um tamanho de campo de apenas 7 dígitos inteiros, é pequeno demais para conter o valor 64652545. Observe, também, a maneira como a lista DADOS foi impressa. Os dois últimos comandos PRINT exemplificam o uso do comando DO para gerar uma lista implícita de valores a serem impressos. Observe que, ao não especificar um formato, os valores da matriz X são impressos seqüencialmente na mesma linha; a fim de formatarmos a saída, o último comando PRINT utiliza o formato 2F8.5, o que especifica que serão impressos apenas 2 valores reais por linha, cada um ocupando um campo de tamanho 8 caracteres (incluindo o ponto decimal) e com cinco casas decimais.*

6.1.2 READ

O comando READ permite que se leia um conjunto de valores de um dispositivo de entrada. Sua sintaxe é a seguinte:

```
READ   <formato> , ! <nome-namelist>  |
    ( [UNIT=] <unidade> [ ,  [FMT=] <formato> | [NML=] <nome-namelist>  ]
      [ , ADVANCE=  'YES' | 'NO' ] [, END=<rótulo>] [ , EOR=<rótulo>] [ , ERR=<rótulo>]
      [ , IOSTAT=<resultado>] [ , REC=<registro>] [ , SIZE=<tamanho>] ) <lista-nomes>

<formato> ::= <identificador-FORMAT> | ('<expressão-formato>') | <nome>
```

Semântica:

1. * indica que os dados serão lidos do dispositivo padrão de entrada (teclado) em *formato livre*, cada dado separado dos demais por um ou mais espaços em branco (dependente de cada implementação);
2. `<formato>` indica como os dados serão formatados, podendo ser:
   (a) O número de um comando `FORMAT` (ver §6.2);
   (b) O caracter *, indicando formato livre;
   (c) Uma `<expressão-formato>`, i.e. uma *"string"* de *caracteres de edição* (ver §6.2), englobada por apóstrofos e delimitada por abre e fecha parênteses;
   (d) Um `<nome>` de uma variável do tipo `CHARACTER` que contém uma `<expressão-formato>`;
3. `<nome-namelist>` é o nome de uma variável declarada com o atributo `NAMELIST` (ver §3.7.4), que contém uma lista de nomes de variáveis;
4. `<unidade>` é:
   (a) uma *"string"* de caracteres ou uma variável do tipo `CHARACTER`, indicando um arquivo interno;
   (b) um número inteiro que identifica um arquivo externo; a associação entre o nome do arquivo externo e `<unidade>` é feita através do comando `OPEN`;
5. `ADVANCE` indica se cada operação de leitura de um arquivo externo será feita registro a registro (*"default"*) ou não;
6. `END` indica que, ao se encontrar o fim de um arquivo, o fluxo de execução do programa será desviado para o comando identificado por `<rótulo>`;
7. `EOR` indica que, ao se encontrar o fim de um registro, o fluxo de execução do programa será desviado para o comando identificado por `<rótulo>`;
8. `ERR` indica que, caso ocorra um erro na leitura, o fluxo de execução do programa será desviado para o comando identificado por `<rótulo>`;
9. Ao se usar o atributo `IOSTAT`, o resultado do `READ` será armazenado na variável inteira `<resultado>`; o valor `-1` indica fim de arquivo foi encontrado, e qualquer outro valor é o número identificador de um erro, dependente da implementação;
10. Ao se usar o atributo `REC`, a leitura será feita no registro indicado pelo valor numérico inteiro `<registro>`. O arquivo associado à unidade sendo lida deve ter sido aberto em modo `DIRECT`. O uso desse atributo impede que se utilize o atributo `END=<rótulo>` e os dados a serem lidos não podem ser especificados por uma `NAMELIST`;
11. `SIZE` especifica o número de caracteres a serem transferidos durante a operação de leitura, através do valor inteiro `<tamanho>`. Só pode ser especificado quando `ADVANCE='NO'`;
12. `<lista-nomes>` é uma lista de nomes de variáveis, com no mínimo um nome, separados por vírgulas. A lista de nomes pode ser especificada de forma implícita através de um comando `DO`. Se `<nome-namelist>` for especificado, então `<lista-nomes>` deve ser *omitida.*

Exemplo 6.2 *Comando READ*

```
PROGRAM PRINCIPAL
IMPLICIT NONE
INTEGER :: A, B, C, I
CHARACTER(LEN=13) :: BUFFER = ' 9876 1234 99'
REAL, DIMENSION(4) :: RESULTADOS
```

```
NAMELIST /DADOS/ A,B,C

    READ (BUFFER,FMT='(2I5)')A,B
    READ (BUFFER,'I2')C
    PRINT DADOS
    READ (*,'(2F4.2)')(RESULTADOS(I),I=1,4)
    PRINT *,RESULTADOS

END PROGRAM PRINCIPAL
```

O programa acima, ao ser executado, produziria o seguinte resultado:

```
&DADOS
A =          9876,
B =          1234,
C =             9
/
```

ou seja, foram lidos três valores inteiros, os quais encontravam-se armazenados numa "string" *de caracteres BUFFER. Após, o programa aguarda que se digite 4 valores reais, dois a cada vez; cada valor tem um tamanho de 4 caracteres, com duas casas decimais. Assim, se digitarmos*

```
0.457.68
1.237.90
```

(após cada linha, tecla-se ENTER) o arranjo RESULTADOS terá os valores

```
0.4500000        7.680000        1.230000        7.900000
```

Observe que, devido ao formato especificado no comando READ, os números são digitados sem espaço em branco entre eles.

6.1.3 WRITE

O comando WRITE permite que se escreva um conjunto de valores num dispositivo de saída. Sua sintaxe é a seguinte:

```
WRITE ( [UNIT=] <unidade> [ , [ FMT=] <formato>  | [ NML=] <nome-namelist> ]
        [ , ADVANCE=  'YES' | 'NO'  ] [ , ERR=<rótulo>] [, IOSTAT=<resultado>]
        [ , REC=<registro>] ) <lista-nomes>

<formato> ::= <identificador-FORMAT> | ('<expressão-formato>') | <nome>
```

Semântica:

1. <unidade> é:
   (a) uma *"string"* de caracteres ou uma variável do tipo CHARACTER, indicando um arquivo interno;
   (b) um número inteiro que identifica um arquivo externo; a associação entre o nome do arquivo externo e <unidade> é feita através do comando OPEN;
2. <formato> indica como os dados serão formatados, podendo ser:
   (a) O número de um comando FORMAT (ver §6.2);
   (b) O caracter *, indicando formato livre;
   (c) Uma <expressão-formato>, i.e. uma *"string"* de *caracteres de edição* (ver §6.2), englobada por apóstrofos e delimitada por abre e fecha parênteses;
   (d) Um <nome> de uma variável do tipo CHARACTER que contém uma <expressão-formato>;

3. <nome-namelist> é o nome de uma variável declarada com o atributo NAMELIST (ver §3.7.4), que contém uma lista de nomes de variáveis;

4. ADVANCE indica se cada operação de escrita de um arquivo externo será feita registro a registro (*"default"*) ou não;

5. ERR indica que, caso ocorra um erro na escrita, o fluxo de execução do programa será desviado para o comando identificado por <rótulo>;

6. Ao se usar o atributo IOSTAT, o resultado do WRITE será armazenado na variável inteira <resultado>; o valor -1 indica fim de arquivo foi encontrado, e qualquer outro valor é o número identificador de um erro, dependente da implementação;

7. Ao se usar o atributo REC, a escrita será feita no registro indicado pelo valor numérico inteiro <registro>. O arquivo associado à unidade sendo escrita deve ter sido aberto em modo DIRECT. Os dados a serem escritos não podem ser especificados por uma NAMELIST;

8. <lista-nomes> é uma lista de nomes de variáveis, com no mínimo um nome, separados por vírgulas. A lista de nomes pode ser especificada de forma implícita através de um comando DO. Se <nome-namelist> for especificado, então <lista-nomes> deve ser *omitida.*

Exemplo 6.3 *Comando WRITE*

```
PROGRAM PRINCIPAL
IMPLICIT NONE
INTEGER :: A=12323, B=64652545, I, J
REAL, DIMENSION(2,2) :: X
NAMELIST /DADOS/ B,A

    X(1,:) = (/ 1.45656, 3.2367 /)
    X(2,:) = (/ -9.435, 3.456 /)
    WRITE (*,*)'A=',A,' B=',B
    WRITE (*,'(I5,I7)')A,B
    WRITE (*,NML=DADOS)
    WRITE (*,*)((X(I,J),J=1,2),I=1,2)
    WRITE (*,FMT='(2F8.5)')((X(I,J),J=1,2),I=1,2)

END PROGRAM PRINCIPAL
```

O programa acima, ao ser executado, produziria o seguinte resultado:

```
 A=        12323  B=     64652545
12323*******
 &DADOS
 B =     64652545,
 A =        12323
 /
   1.456560        3.236700       -9.435000        3.456000
 1.45656 3.23670
-9.43500 3.45600
```

Observe que esse programa produz o mesmo resultado do que aquele obtido usando o comando PRINT, mostrado no exemplo 6.1.

6.1.4 OPEN

O comando OPEN permite que se associe um arquivo externo a uma unidade, para se realizar uma operação de entrada, saída, ou ambos. Sua sintaxe é a seguinte:

```
OPEN ( [UNIT=] <unidade>
       [ , ACCESS= 'DIRECT' | 'SEQUENTIAL' ]
       [ , ACTION= 'READ' | 'WRITE' | 'READWRITE' ]
       [ , BLANK= 'NULL' | 'ZERO' ]
       [ , DELIM= 'APOSTROPHE' | 'QUOTE' | 'NONE' ]
       [ , ERR= <rótulo>]
       [ , FILE= <nome-arquivo>]
       [ , FORM= 'FORMATTED' | 'UNFORMATTED' ]
       [ , IOSTAT= <resultado> ]
       [ , PAD= 'YES' | 'NO' ]
       [ , POSITION= 'ASIS' | 'REWIND' | 'APPEND' ]
       [ , RECL= <tamanho-registro>]
       [ , STATUS= 'OLD' | 'NEW' | 'SCRATCH' | 'REPLACE' | 'UNKNOWN' ] )
```

Semântica:

1. `<unidade>` é um número inteiro maior ou igual a zero que identifica um arquivo externo;
2. `ACCESS` indica o tipo de acesso a um arquivo, seqüencial (*"default"*) ou direto (i.e., um comando `READ` ou `WRITE` pode ser usado com o atributo `REC`;
3. `ACTION` indica como o arquivo será aberto, se para leitura, escrita, ou para leitura e escrita;
4. `BLANK` indica como são tratados espaços em branco ao ser lido um dado. `BLANK = 'NULL'` é o *"default"* e significa que qualquer espaço em branco será desconsiderado; `BLANK = 'ZEROS'` indica que os espaços em branco devem ser considerados como zeros.
5. `DELIM` indica qual caracter é utilizado para separar os dados. `DELIM = 'APOSTROPHE'` indica o uso do caracter ', `DELIM = 'QUOTE'` indica o caracter " e `DELIM = 'NONE'` indica que nenhum caracter é usado (*"default"*);
6. `ERR` indica que, caso ocorra um erro, o fluxo de execução do programa será desviado para o comando identificado por `<rótulo>`;
7. `FILE` faz a associação entre o nome do arquivo externo e o número da unidade. O nome é o valor do *"string"* `<nome-arquivo>`, podendo ser uma variável ou uma constante do tipo `CHARACTER`. Caso não seja informado, serão utilizados os nomes *"default"* `UNIT0`, `UNIT1`, ...;
8. `FORM` indica se o arquivo é formatado ou não. Para arquivos com `ACCESS = 'SEQUENTIAL'`, o *"default"* é `FORM = 'FORMATTED'`; para arquivos com `ACESS = 'DIRECT'`, o *"default"* é `FORM = 'UNFORMATTED'`;
9. Ao se usar o atributo `IOSTAT`, o resultado do `OPEN` será armazenado na variável inteira `<resultado>`; o valor `-1` indica fim de arquivo foi encontrado, e qualquer outro valor é o número identificador de um erro, dependente da implementação;
10. `PAD` informa se os dados lidos serão completados com espaços em branco (`PAD = 'YES'`), caso o formato estabelecido para leitura tenha um tamanho maior do que o existente (*"default"*);
11. `POSITION` indica a posição do arquivo para acesso seqüencial:
    - `POSITION = 'ASIS'` é o *"default"*, i.e., a posição de um arquivo já associado a uma unidade não é alterado, enquanto arquivos novos sempre estão posicionados no início;
    - `POSITION = 'REWIND'` faz com que o arquivo seja reposicionado para o seu início;
    - `POSITION = 'APPEND'` faz com que o arquivo seja reposicionado para o seu fim, permitindo, por exemplo, que novos registros sejam adicionados ao arquivo.
12. `RECL = <tamanho-registro>` informa o tamanho do registro (em *"bytes"*) de um arquivo de acesso direto, ou o maior tamanho de um registro num arquivo de acesso seqüencial. Esse atributo é requerido para arquivos abertos com `ACCESS = 'DIRECT'`;

13. STATUS descreve como o arquivo é aberto:

    - STATUS = 'OLD' indica que o arquivo já existe (se não existir, ocorrerá um erro);
    - STATUS = 'NEW' indica que o arquivo será criado pelo programa (se já existir, ocorrerá um erro);
    - STATUS = 'SCRATCH' indica que o arquivo será criado pelo programa, porém de forma temporária. Ele será removido quando a unidade sofrer um comando CLOSE ou quando o programa terminar;
    - STATUS = 'REPLACE' indica que um arquivo já existente terá o seu conteúdo modificado pelo programa. Caso o arquivo não exista, ele será criado;
    - STATUS = 'UNKNOWN' indica que, se um arquivo já existir, ele será aberto; se não existir, um novo arquivo será criado.

6.1.5 CLOSE

O comando CLOSE permite que se desassocie um arquivo externo a uma unidade. Sua sintaxe é a seguinte:

```
CLOSE ( [UNIT=] <unidade>
        [ , ERR= <rótulo> ]
        [ , IOSTAT= <resultado>]
        [ , STATUS=  'KEEP' | 'DELETE'  ] )
```

Semântica:

1. <unidade> é um número inteiro maior ou igual a zero que identifica um arquivo externo;
2. ERR indica que, caso ocorra um erro, o fluxo de execução do programa será desviado para o comando identificado por <rótulo>;
3. Ao se usar o atributo IOSTAT, o resultado do OPEN será armazenado na variável inteira <resultado>; o valor -1 indica fim de arquivo foi encontrado, e qualquer outro valor é o número identificador de um erro, dependente da implementação;
4. STATUS descreve como o arquivo é fechado:
    - STATUS = 'KEEP' indica que o arquivo será mantido (não pode ser usado para arquivos abertos com STATUS = 'SCRATCH');
    - STATUS = 'DELETE' indica que o arquivo será removido.

6.1.6 INQUIRE

O comando INQUIRE permite que se obtenha informações sobre um arquivo externo.

```
INQUIRE (  [UNIT=] <unidade> | FILE= <nome-arquivo> | IOLENGTH= <iolength>
           [, ACCESS= <accesso> ]
           [, ACTION= <ação> ]
           [, BLANK= <espaços> ]
           [, DELIM= <delimitador> ]
           [, DIRECT= <acesso> ]
           [, ERR= <rótulo> ]
           [, EXIST= <situação> ]
           [, FORM= <forma> ]
           [, FORMATTED= <formatado> ]
           [, IOSTAT= <resultado> ]
           [, NAME= <nome> ]
           [, NAMED= <nomeado> ]
```

```
[, NEXTREC= <registro> ]
[, NUMBER= <número> ]
[, OPENED= <situação> ]
[, PAD= <situação> ]
[, POSITION= <posição> ]
[, READ= <situação> ]
[, READWRITE= <situação> ]
[, RECL= <tamanho> ]
[, SEQUENTIAL= <situação> ]
[, UNFORMATTED= <situação> ]
[, WRITE= <situação> ] )
```

Semântica:

1. <unidade> é um número inteiro maior ou igual a zero que identifica um arquivo externo, ou *;
2. FILE é o nome de um arquivo externo. O nome é o valor do *"string"* <nome-arquivo>, podendo ser uma variável ou uma constante do tipo CHARACTER. Caso não seja informado, serão utilizados os nomes *"default"* UNIT0, UNIT1, ...;
3. IOLENGTH é um número inteiro que indica o tamanho de uma lista de saída;
4. ACCESS retorna uma *"string"* informando se o acesso é seqüencial ('SEQUENTIAL') ou direto ('DIRECT'). Caso o arquivo ainda não tenha sido associado a uma unidade através do comando OPEN, o resultado é 'UNDEFINED';
5. ACTION retorna uma *"string"* informando se o arquivo foi aberto para leitura ('READ'), escrita ('WRITE') ou leitura-e-escrita ('READWRITE'). Caso o arquivo ainda não tenha sido associado a uma unidade através do comando OPEN, o resultado é 'UNDEFINED';
6. BLANK retorna uma *"string"* informando como são tratados os espaços em branco, se desconsiderados ('NULL') ou como zeros ('ZEROS'). Caso o arquivo ainda não tenha sido associado a uma unidade através do comando OPEN, ou ele não foi aberto para saída formatada, o resultado é 'UNDEFINED';
7. DELIM retorna uma *"string"* informando quais delimitadores são usados: nenhum ('NONE'), apóstrofos ('APOSTROPHE') ou aspas ('QUOTE'). Caso o arquivo ainda não tenha sido associado a uma unidade através do comando OPEN, o resultado é 'UNDEFINED';
8. DIRECT retorna uma *"string"* informando se o arquivo foi aberto para acesso direto ('YES'), ou não ('NO'). Caso o arquivo ainda não tenha sido associado a uma unidade através do comando OPEN, o resultado é 'UNKNOWN';
9. ERR indica que, caso ocorra um erro, o fluxo de execução do programa será desviado para o comando identificado por <rótulo>;
10. EXIST retorna um valor lógico informando se o arquivo existe (.TRUE.) ou não (.FALSE.);
11. FORM retorna uma *"string"* informando se o arquivo foi aberto para operações formatadas ('FORMATTED') ou não-formatada ('UNFORMATTED'). Caso o arquivo ainda não tenha sido associado a uma unidade através do comando OPEN, o resultado é 'UNDEFINED';
12. FORMATTED retorna uma *"string"* informando se o arquivo pode ser aberto para operações formatadas ('YES') ou não ('NO'). Caso o processador não possa determinar essa possibilidade, o resultado é 'UNKNOWN';
13. Ao se usar o atributo IOSTAT, o resultado do OPEN será armazenado na variável inteira <resultado>; o valor 0 indica que não ocorreu erro, ou fim de arquivo ou fim de registro, e qualquer outro valor é o número identificador de um erro, dependente da implementação;

14. NAME retorna uma *"string"* contendo o nome do arquivo associado a uma <unidade>, caso UNIT = <unidade> tenha sido especificado. Se FILE = <nome-arquivo> foi especificado, então retorna <nome-arquivo>. Caso o arquivo ainda não tenha sido conectado a uma unidade através de um comando OPEN, ou não tenha nome, o resultado é indefinido;

15. NAMED retorna uma variável lógica especificando se o arquivo tem um nome (.TRUE.) ou não (.FALSE.);

16. NEXTREC retorna uma variável inteira indicando qual o número do próximo registro num arquivo aberto para acesso direto (o primeiro registro é indicado pelo número 1);

17. NUMBER retorna uma variável inteira indicando qual a unidade associada ao arquivo. Não pode ser usado em conjunto com UNIT = <unidade>;

18. OPENED retorna uma variável lógica indicando se o arquivo indicado por FILE = <nome-arquivo> ou UNIT = <unidade> está aberto (.TRUE.) ou não (.FALSE.);

19. PAD retorna uma *"string"* contendo 'YES' se o arquivo foi aberto com PAD = 'YES', ou 'NO' caso contrário;

20. POSITION retorna uma *"string"* contendo ASIS se a posição de um arquivo já associado a uma unidade não é alterado, enquanto arquivos novos sempre estão posicionados no início; REWIND se o arquivo foi reposicionado para o seu início; APPEND se o arquivo foi reposicionado para o seu fim, permitindo, por exemplo, que novos registros sejam adicionados ao arquivo; e UNDEFINED se o arquivo ainda não foi aberto ou se foi aberto para acesso direto;

21. READ retorna uma *"string"* contendo 'YES' se o arquivo foi aberto para leitura, 'NO' se o arquivo não pode ser lido e 'UNKNOWN' se o processador não pode determinar essa condição;

22. READWRITE retorna uma *"string"* contendo 'YES' se o arquivo foi aberto para leitura-e-escrita, 'NO' se o arquivo não pode ser lido e 'UNKNOWN' se o processador não pode determinar essa condição;

23. RECL retorna numa variável inteira o tamanho de um registro, em *"bytes"*, para um arquivo de acesso direto, ou o tamanho do maior registro de um arquivo de acesso seqüencial;

24. SEQ retorna uma *"string"* contendo 'YES' se o arquivo foi aberto para acesso seqüencial, 'NO' se não e 'UNKNOWN' se o processador não pode determinar essa condição;

25. UNFORMATTED retorna uma *"string"* contendo 'YES' se o arquivo foi aberto para operações formatadas, 'NO' se não e 'UNKNOWN' se o processador não pode determinar essa condição;

26. WRITE retorna uma *"string"* contendo 'YES' se o arquivo foi aberto para escrita, 'NO' se o arquivo não pode ser lido e 'UNKNOWN' se o processador não pode determinar essa condição.

Exemplo 6.4 *Comandos CLOSE, INQUIRE e OPEN*

```
PROGRAM TESTE

LOGICAL :: EXISTE
CHARACTER*9 :: NOME = 'dados.dat'

    INQUIRE(FILE=NOME,EXIST=EXISTE)

    IF (EXISTE) THEN
        OPEN(UNIT=1,FILE=NOME,STATUS='OLD')
        CLOSE(1,STATUS='DELETE')
    END IF

END PROGRAM TESTE
```

O programa acima faz uma consulta para verificar se o arquivo dados.dat existe. Se a variável EXISTE contiver o valor .TRUE. após a execução do comando INQUIRE, então o arquivo será aberto com o atributo STATUS='OLD', indicando que ele já existe. Após, o comando CLOSE será executado com o atributo STATUS='DELETE', efetivamente removendo o arquivo. Observe as duas maneiras de especificar a unidade 1, nos comandos OPEN e CLOSE.

6.1.7 REWIND

O comando REWIND faz com que qualquer operação de E/S sobre um arquivo seja feita a partir do registro número 1.

```
REWIND ( [UNIT=] <unidade>
       [ , ERR= <rótulo> ]
       [ , IOSTAT= <resultado>] )
```

Semântica:

1. <unidade> é um número inteiro maior ou igual a zero que identifica um arquivo externo;
2. ERR indica que, caso ocorra um erro, o fluxo de execução do programa será desviado para o comando identificado por <rótulo>;
3. Ao se usar o atributo IOSTAT, o resultado do comando será armazenado na variável inteira <resultado>; o valor -1 indica fim de arquivo foi encontrado, e qualquer outro valor é o número identificador de um erro, dependente da implementação.

6.1.8 BACKSPACE

O comando BACKSPACE faz com que qualquer operação de E/S sobre um arquivo seja feita a partir do início do registro anterior àquele no qual o arquivo encontra-se posicionado.

```
BACKSPACE ( [UNIT=] <unidade>
       [ , ERR= <rótulo> ]
       [ , IOSTAT= <resultado>] )
```

Semântica:

1. <unidade> é um número inteiro maior ou igual a zero que identifica um arquivo externo;
2. ERR indica que, caso ocorra um erro, o fluxo de execução do programa será desviado para o comando identificado por <rótulo>;
3. Ao se usar o atributo IOSTAT, o resultado do comando será armazenado na variável inteira <resultado>; o valor -1 indica fim de arquivo foi encontrado, e qualquer outro valor é o número identificador de um erro, dependente da implementação.

6.1.9 ENDFILE

O comando ENDFILE escreve um registro num arquivo externo que caracteriza fim de arquivo.

```
ENDFILE ( [UNIT=] <unidade>
       [ , ERR= <rótulo> ]
       [ , IOSTAT= <resultado>] )
```

Semântica:

1. <unidade> é um número inteiro maior ou igual a zero que identifica um arquivo externo;

2. ERR indica que, caso ocorra um erro, o fluxo de execução do programa será desviado para o comando identificado por <rótulo>;

3. Ao se usar o atributo IOSTAT, o resultado do comando será armazenado na variável inteira <resultado>; o valor -1 indica fim de arquivo foi encontrado, e qualquer outro valor é o número identificador de um erro, dependente da implementação.

6.2 Formatação de E/S

Em alguns dos exemplos mostrados nas seções anteriores, já fizemos uso de alguns *caracteres de edição* para formatar um dado. Existem caracteres de edição *de dados* para formatar cada um dos tipos de dados pré-definidos na linguagem, e caracteres *de controle* de edição que agem sobre a operação de E/S. Esses caracteres de edição podem ser usados diretamente num comando READ, PRINT ou WRITE, ou ainda serem especificados dentro do comando FORMAT.

6.2.1 Comando FORMAT

A sintaxe do comando é a seguinte:

```
<rótulo> FORMAT ( <lista-de-caracteres-de-edição> )
```

Semântica:

1. <rótulo> é um número inteiro que identifica o comando FORMAT, através do qual ele será referenciado em um comando READ ou WRITE através do atributo FMT = <rótulo>;

2. <lista-de-caracteres-de-edição> é um conjunto de caracteres de edição.

6.2.2 Caracteres de edição de dados

Os caracteres de edição de dados definem como um valor, seja ele armazenado em uma variável ou uma constante de um dos tipos pré-definidos na linguagem, será lido e/ou escrito. A especificação de um caracter de edição trabalha com a noção de *tamanho de campo*, i.e., quantos caracteres serão utilizados para representar aquele valor (seja na leitura ou na escrita). Os caracteres de edição mais comumente usados serão descritos a seguir.

6.2.2.1 Caracter de edição rAw ou rA

Formata uma *"string"* num campo de tamanho w. Na leitura, será transferido para a variável sendo lida w caracteres; na escrita, será transferido para o dispositivo de saída apenas os primeiros w caracteres da expressão. O caracter de edição pode ser precedido de um número inteiro r, o qual indica que r dados serão escritos ou lidos.

A forma A formata uma *"string"* num campo de tamanho não-especificado. Na leitura, será transferido para a variável sendo lida apenas o número de caracteres correspondente ao tamanho com a qual foi declarada; na escrita, será transferido para o dispositivo de saída todo o conteúdo da expressão.

6.2.2.2 Caracter de edição rBw ou rBw.m

Formata um número inteiro num campo de tamanho w, em notação binária. O valor de w deve ser tal que o número inteiro possa ser representado na forma de uma *"string"*, incluindo um caracter para o sinal e os caracteres 0 e 1. O caracter de edição pode ser precedido de um número inteiro r, o qual indica que r dados serão escritos ou lidos.

Na escrita, se w não for grande o suficiente, serão impressos w caracteres *, indicando *"estouro de campo"*; na leitura, serão lidos w caracteres, inclusive espaços em branco, os quais serão

interpretados de acordo com o tratamento atribuído a eles, através dos caracteres de edição BN/BZ ou do uso do atributo BLANK em um comando OPEN.

A forma Bw.m só é válida em uma operação de escrita, sendo desconsiderada na leitura. Ela faz com que, se o número inteiro ocupar até m caracteres, então ele será escrito justificado à direita do campo, com w-m espaços em branco à sua esquerda.

6.2.2.3 Caracter de edição rDw.d

Formata um número real de tipo DOUBLE PRECISION num campo de tamanho w, em notação decimal, i.e. [-]i.f, onde - representa o sinal negativo, i a parte inteira e f a parte fracionária, expressas como *"strings"* contendo caracteres de 0 a 9. O caracter de edição pode ser precedido de um número inteiro r, o qual indica que r dados serão escritos ou lidos.

O valor de w deve ser tal que o número real possa ser representados na forma de uma *"string"*, incluindo um caracter para o sinal e um caracter para o ponto decimal. O valor de d especifica o tamanho do campo da parte fracionária.

Na escrita, se w não for grande o suficiente, serão impressos w caracteres *, indicando *"estouro de campo"*; na leitura, serão lidos w caracteres, inclusive espaços em branco, os quais serão interpretados de acordo com o tratamento atribuído a eles, através dos caracteres de edição BN/BZ ou do uso do atributo BLANK em um comando OPEN. Se d não for grande o suficiente para representar a parte fracionária, ocorrerá arredondamento, dependente da implementação.

6.2.2.4 Caracter de edição rEw.d ou rEw.dEe

Formata um número real ou complexo num campo de tamanho w, em notação científica, i.e. [-]i.fEse, onde - representa o sinal negativo, i a parte inteira, f a parte fracionária, E a potência 10, s o sinal do expoente e e o valor do expoente (um número inteiro); i, f e e são expressos como *"strings"* contendo caracteres de 0 a 9. O caracter de edição pode ser precedido de um número inteiro r, o qual indica que r dados serão escritos ou lidos.

O valor de w deve ser tal que o número real ou as partes real e imaginária de um número complexo possam ser representados na forma de uma *"string"*, incluindo um caracter para o sinal, um caracter para o ponto decimal, o caracter E e o sinal do expoente (o qual é impresso sempre, seja positivo ou negativo). O valor de d especifica o tamanho do campo da parte fracionária; se utilizada a forma Ew.dEe, então e indica o tamanho do campo representando o expoente.

Na escrita, se w não for grande o suficiente, serão impressos w caracteres *, indicando *"estouro de campo"*; na leitura, serão lidos w caracteres, inclusive espaços em branco, os quais serão interpretados de acordo com o tratamento atribuído a eles, através dos caracteres de edição BN/BZ ou do uso do atributo BLANK em um comando OPEN. Se d não for grande o suficiente para representar a parte fracionária, ocorrerá arredondamento, dependente da implementação.

6.2.2.5 Caracter de edição rIw ou rIw.m

Formata um número inteiro num campo de tamanho w. O valor de w deve ser tal que o número inteiro possa ser representado na forma de uma *"string"*, incluindo um caracter para o sinal e os caracteres 0 a 9. O caracter de edição pode ser precedido de um número inteiro r, o qual indica que r dados serão escritos ou lidos.

Na escrita, se w não for grande o suficiente, serão impressos w caracteres *, indicando *"estouro de campo"*; na leitura, serão lidos w caracteres, inclusive espaços em branco, os quais serão interpretados de acordo com o tratamento atribuído a eles, através dos caracteres de edição BN/BZ ou do uso do atributo BLANK em um comando OPEN.

A forma Iw.m só é válida em uma operação de escrita, sendo desconsiderada na leitura. Ela faz com que, se o número inteiro ocupar até m caracteres, então ele será escrito justificado à direita do campo, com w-m espaços em branco à sua esquerda.

6.2.2.6 Caracter de edição rFw.d

Formata um número real ou complexo num campo de tamanho w, em notação decimal, i.e. [-]i.f, onde - representa o sinal negativo, i a parte inteira e f a parte fracionária, expressas como *"strings"* contendo caracteres de 0 a 9. O caracter de edição pode ser precedido de um número inteiro r, o qual indica que r dados serão escritos ou lidos.

O valor de w deve ser tal que o número real ou as partes real e imaginária de um número complexo possam ser representados na forma de uma *"string"*, incluindo um caracter para o sinal e um caracter para o ponto decimal. O valor de d especifica o tamanho do campo da parte fracionária.

Na escrita, se w não for grande o suficiente, serão impressos w caracteres *, indicando *"estouro de campo"*; na leitura, serão lidos w caracteres, inclusive espaços em branco, os quais serão interpretados de acordo com o tratamento atribuído a eles, através dos caracteres de edição BN/BZ ou do uso do atributo BLANK em um comando OPEN. Se d não for grande o suficiente para representar a parte fracionária, ocorrerá arredondamento, dependente da implementação.

6.2.2.7 Caracter de edição rLw

Formata um valor lógico num campo de tamanho w. O caracter de edição pode ser precedido de um número inteiro r, o qual indica que r dados serão escritos ou lidos.

Na escrita, será impresso w-1 caracteres em branco, seguidos dos caracteres T ou F, indicando os valores .TRUE. e .FALSE. respectivamente. Na leitura, deve haver no mínimo um caracter T ou F (opcionalmente precedidos de espaços em branco e um ponto).

6.2.2.8 Caracter de edição wH

Indica o tamanho w de uma *"string"* de caracteres, expressa dentro de um comando FORMAT.

6.2.2.9 Caracter de edição rOw ou rOw.m

Formata um número inteiro num campo de tamanho w, em notação octal. O valor de w deve ser tal que o número inteiro possa ser representado na forma de uma *"string"*, incluindo um caracter para o sinal e os caracteres 0 a 7. O caracter de edição pode ser precedido de um número inteiro r, o qual indica que r dados serão escritos ou lidos.

Na escrita, se w não for grande o suficiente, serão impressos w caracteres *, indicando *"estouro de campo"*; na leitura, serão lidos w caracteres, inclusive espaços em branco, os quais serão interpretados de acordo com o tratamento atribuído a eles, através dos caracteres de edição BN/BZ ou do uso do atributo BLANK em um comando OPEN.

A forma Ow.m só é válida em uma operação de escrita, sendo desconsiderada na leitura. Ela faz com que, se o número inteiro ocupar até m caracteres, então ele será escrito justificado à direita do campo, com w-m espaços em branco à sua esquerda.

6.2.2.10 Caracter de edição rZw ou rZw.m

Formata um número inteiro num campo de tamanho w, em notação hexadecimal. O valor de w deve ser tal que o número inteiro possa ser representado na forma de uma *"string"*, incluindo um caracter para o sinal e os caracteres 0 a 9 e A a F. O caracter de edição pode ser precedido de um número inteiro r, o qual indica que r dados serão escritos ou lidos.

Na escrita, se w não for grande o suficiente, serão impressos w caracteres *, indicando *"estouro de campo"*; na leitura, serão lidos w caracteres, inclusive espaços em branco, os quais serão interpretados de acordo com o tratamento atribuído a eles, através dos caracteres de edição BN/BZ ou do uso do atributo BLANK em um comando OPEN.

A forma Zw.m só é válida em uma operação de escrita, sendo desconsiderada na leitura. Ela faz com que, se o número inteiro ocupar até m caracteres, então ele será escrito justificado à direita do campo, com w-m espaços em branco à sua esquerda.

6.2.3 Caracteres de controle de edição

Os caracteres de controle de edição permitem controlar a edição das operações de E/S.

6.2.3.1 Caracter de edição BN ou BZ

Indica como deve ser feito o tratamento dos espaços em branco. BN indica que um espaço em branco deve ser desconsiderado; BZ indica que um espaço em branco deve ser tratado como se fosse um zero.

6.2.3.2 Caracter de edição S, SP, SS

Indicam se o sinal de um número positivo deve ou não ser escrito. SP indica que o sinal + deve sempre ser impresso para os formatos subseqüentes na lista de impressão de um atributo FMT= ou comando FORMAT; SS indica que o sinal + não deve ser impresso para os formatos subseqüentes; e S retorna ao modo *"default"*, i.e., só imprime os sinais negativos.

6.2.3.3 Caracter de edição wX

Indica que w espaços em branco serão lidos ou escritos.

6.2.3.4 Caracter de edição :

Faz com que os caracteres de edição de dados ou *"strings"* sejam desconsiderados, caso não haja mais variáveis a serem lidas ou escritas.

6.2.3.5 Caracter de edição /

Indica que a transferência de dados deve ser terminada. Na leitura, permite desconsiderar dados em um registro; na escrita, faz com que o registro sendo escrito seja terminado e a próxima operação de E/S far-se-á sobre o próximo registro (por exemplo, imprimir linhas em branco).

6.2.4 Exemplos do uso de caracteres de edição

Exemplo 6.5 *Uso dos caracteres de edição A, Bw, Iw, wX e /.*

```
PROGRAM TESTE
IMPLICIT NONE
INTEGER :: A=8, B=1024, I
CHARACTER*10 :: TAB = '1234567890'

    WRITE(*,'8(9X,I1)/8A')(I,I=1,8),(TAB,I=1,8)
    WRITE(*,'B32')A,-A,B

END PROGRAM TESTE
```

O programa acima imprime valores inteiros usando os caracteres de edição Bw e Iw. Inicialmente, utiliza-se um comando WRITE com uma lista criada por dois DOs explícitos, usando primeiramente o formato 9X,I1 repetido 8 vezes, de forma a imprimir os inteiros de 1 a 8, separados por 9 espaços em branco; após, pula-se para uma nova linha, através do caracter de edição / e imprime-se oito vezes o conteúdo da variável TAB. Com isso, obtemos as duas primeiras linhas da saída do programa, conforme abaixo ilustrado:

```
         1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
                                1000
11111111111111111111111111111000
                      10000000000
```

Após, imprime-se o conteúdo da variável A, da expressão aritmética -A e da variável B, usando o formato B32. Com isso, os valores a serem impressos serão formatados em notação binária com 32 dígitos, já que as variáveis foram declaradas como sendo do tipo INTEGER, cujo tamanho "default" *é de 4* "bytes" *(8* "bits"*). Note que os valores são impressos um por linha, já que o formato B32 não continha um valor de repetição precedendo-o (por exemplo, 3B32 faria com que os três valores fossem impressos na mesma linha, ocupando 96 caracteres ao todo). Além disso, observe que o valor impresso para -A corresponde à representação, em* complemento-de-2, *do valor A=8.*

Exemplo 6.6 *Uso dos caracteres de edição Ew.d, Fw.d e SP*

```
PROGRAM TESTE
IMPLICIT NONE
TYPE VETOR_2D
    REAL :: X,Y
END TYPE VETOR_2D

TYPE (VETOR_2D) :: U
COMPLEX :: POLAR
INTEGER :: I
CHARACTER*10 :: TAB = '1234567890'

    POLAR = CMPLX(10.0, ACOS(SQRT(2.0)/2))
    U%X = REAL(POLAR)*COS(IMAG(POLAR))
    U%Y = REAL(POLAR)*SIN(IMAG(POLAR))

    WRITE(*,'8(9X,I1)/8A')(I,I=1,8),(TAB,I=1,8)
    WRITE(*,*)POLAR,U
    WRITE(*,'2F3.2/2F3.2')POLAR,U
    WRITE(*,'SP,2E12.4')U

END PROGRAM TESTE
```

*O programa acima imprime valores reais usando os caracteres de edição Ew.d, Fw.d e em formato livre, *. Após a impressão das duas primeiras linhas, obtidas da mesma forma que no exemplo 6.5, imprime-se em formato livre os valores das variáveis POLAR e U, obtendo-se a terceira linha da saída do programa, conforme abaixo ilustrado:*

```
        1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
 (10.00000,0.7853982)     7.071068         7.071068
***.79
******
 +0.7071E+01 +0.7071E+01
```

*Note que POLAR é uma variável do tipo COMPLEX e, em formato livre, o seu valor é impresso englobado por parênteses. U é uma variável de tipo definido pelo usuário, VETOR_2D, e o comando WRITE imprimiu em sequência os valores das partes X e Y de U. Após, tentou-se imprimir os valores de POLAR e U usando o formato 2F3.2/2F3.2; note que o formato F3.2 é muito pequeno para representar os números reais em questão, já que apenas números com parte inteira igual a 0 podem ser impressos. Nesse caso, apenas a parte imaginária de POLAR atende a esse formato, tendo sido impresso o valor .79 (observe o arredondamento do valor 0.7853982); os demais foram substituídos por ***, indicando que houve estouro de campo. Finalmente, imprime-se o valor da variável U usando o formato SP,2E12.4, exigindo que se imprima o sinal + à frente dos valores positivos, expressos em notação científica.*

Exemplo 6.7 *Uso dos caracteres de edição BZ e BN*

```
PROGRAM TESTE
IMPLICIT NONE
COMPLEX :: POLAR
INTEGER :: A, I
CHARACTER*10 :: TAB = '1234567890'
REAL :: V

    WRITE(*,'8(9X,I1)/8A')(I,I=1,8),(TAB,I=1,8)

    READ(*,100)A,V
    WRITE(*,'("A=",I5," V=",F6.4')A,V

100 FORMAT(BZ,I5,BN,F6.4)

END PROGRAM TESTE
```

O programa acima lê um valor inteiro, A e outro real, V. O formato identificado pelo rótulo 100 especifica que os espaços em branco devem ser tratados como zeros, através do caracter de controle de edição BZ. O valor inteiro deve ser lido com o formato I5 e o valor real com o formato F6.4 (isto é, no máximo um dígito antes e quatro depois do ponto decimal).

Antes de ser feita a leitura, imprime-se duas linhas contendo a tabulação, de forma que se identifique o tamanho dos campos; assim, se digitarmos ␣976␣4.23, como ilustrado abaixo, A conterá o valor 9760 (já que o primeiro espaço em branco é considerado como zero e, obviamente, é descartado), e V conterá o valor 4.23.

```
        1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
 976 4.23
A= 9760 V=4.2300
```

Capítulo 7

Subprogramas

Na linguagem FORTRAN 90, existem dois tipos de subprogramas: subrotinas (`SUBROUTINE`) e funções (`FUNCTION`).

Um subprograma tem um conjunto *opcional* de argumentos, os quais podem ser de *entrada*, de *saída* e de *entrada-e-saída* (ver §3.6.11). Um *argumento* é um nome que só pode ser referenciado dentro do subprograma. Cabe observar que nada impede que existam, em outras seções do programa, variáveis com o mesmo nome de argumentos de subprogramas; deve-se ressaltar, no entanto, que essas variáveis *não são* os argumentos. A associação entre um argumento e seu valor só é dada quando o subprograma é *invocado*.

A diferença principal entre `SUBROUTINE` e `FUNCTION` é que, na segunda, é possível se retornar um valor associado ao próprio nome da `FUNCTION`, além de, se necessário, retornar outros valores através de argumentos de saída e/ou de entrada-e-saída.

Um subprograma pode ser *interno* ou *externo* (ver Capítulo 2). No primeiro caso, um subprograma só pode ser invocado no bloco dentro do qual ele foi definido, após o comando `CONTAINS`; no segundo, um subprograma pode existir como um código separado do programa principal e de outros subprogramas, ou dentro de um `MODULE`.

A execução de um subprograma é feita até que se encontre o comando `END SUBROUTINE` ou `END FUNCTION`, ou seja executado o comando `RETURN`. Quando um subprograma termina sua execução, o fluxo de execução do programa retorna para o comando imediatamente posterior ao comando de invocação do subprograma.

7.1 Subrotinas

Uma subrotina em FORTRAN 90 é declarada de acordo com a seguinte sintaxe

```
[ RECURSIVE ] SUBROUTINE <nome-subrotina> [ ( <lista-argumentos> ) ]

   [ <declarações-argumentos> ]

   [ <declarações-internas> ]

   [ <comandos> ]

END SUBROUTINE <nome-subrotina>

<lista-argumentos> ::= <argumento-1> [, <argumento-2> [ ... [, <argumento-n> ] ] ]
```

Semântica:

1. `RECURSIVE` é um atributo opcional da subrotina e indica que ela é *recursiva*, isto é, pode existir uma invocação de `<nome-subrotina>` dentro dos `<comandos>`, inclusive de forma encadeada;

2. <argumento-1> a <argumento-n> é uma lista de argumentos, opcional, cada argumento separado por vírgulas;

3. <declarações-argumentos> é a declaração dos tipos de cada argumento, os quais podem ter o atributo INTENT (ver §3.6.11);

4. <declarações-internas> é a declaração das variáveis que existem *apenas* no escopo da subrotina;

5. <comandos> é uma seqüência de comandos da linguagem, os quais serão executados quando da invocação de <nome-subrotina>; se um desses comandos for o comando RETURN, a execução da subrotina é encerrada.

Os exemplos a seguir mostram a utilização de subrotinas.

Exemplo 7.1 *Passagem de argumentos a uma subrotina.*
Considere o programa abaixo:

```
PROGRAM TESTE
IMPLICIT NONE
INTEGER, PARAMETER :: I = -4, F = 6
INTEGER, DIMENSION(I:F) :: A
INTEGER :: J
    A = 0
    CALL SUB(A)
    WRITE(*,'(" A(",I2,")=",I4)')(J,A(J),J=I,F)
CONTAINS
    SUBROUTINE SUB(ARR)
    IMPLICIT NONE
    INTEGER, DIMENSION(:) :: ARR
    INTEGER :: N, I
        N = SIZE(ARR)
        DO I = 1,N
            ARR(I) = I
        END DO
    END SUBROUTINE SUB
END PROGRAM TESTE
```

A passagem de arranjos a subprogramas em FORTRAN 90 *é uma das situações em que mais ocorrem erros do tipo* acesso inválido a elementos do arranjo. *Pela estrutura da linguagem, é possível se transmitir arranjos de tamanho diferente daquele que um subprograma precisa utilizar; se o arranjo é de tamanho inferior ao necessário, ocorrerá um acesso a um ou mais elementos que não existem no arranjo, e o programa será interrompido de forma catastrófica.*

Nesse exemplo, mostramos como se pode transmitir um arranjo com uma dimensão a uma subrotina. Ao se executar a chamada CALL SUB(A), dentro da subrotina SUB será subentendido que o argumento ARR tem o mesmo "shape" e tamanho *do arranjo A, através da declaração INTEGER, DIMENSION(:) :: ARR. O comando N = SIZE(ARR) armazena em N o número de elementos de ARR (nesse caso, N terá o valor 11). Dentro da subrotina, os elementos do arranjo ARR são acessados pelos índices 1, 2, ..., 11 – e não pelos índices -4, -3, ..., 6 com os quais os elementos de A são acessados dentro do programa principal. No entanto, quando do retorno ao programa principal, verifica-se que os elementos de A contém os valores 1, 2, ..., 11, armazenados nos elementos -4, -3, ..., 6, como demonstra a saída do programa:*

```
A(-4)=   1
A(-3)=   2
A(-2)=   3
A(-1)=   4
A( 0)=   5
A( 1)=   6
```

```
A( 2)=   7
A( 3)=   8
A( 4)=   9
A( 5)=  10
A( 6)=  11
```

Exemplo 7.2 *Cálculo do fatorial de um número.*

Suponha que se deseja calcular o fatorial de um número de um número $n \in \mathbb{N}$, denotado por $n!$, definido como

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

Em FORTRAN 90, *não é oferecido um intrínseco que calcule tal valor. O programa abaixo permite calcular o fatorial chamando-se a subrotina* `FATORIAL`.

```
PROGRAM TESTE
IMPLICIT NONE
INTEGER :: F, N
    PRINT *,'N?'
    READ *,N
    IF (N>=0) THEN
            CALL FATORIAL(N,F)
            WRITE(*,'("FATORIAL(",I12,")=",I12)')N,F
    ELSE
            PRINT *,'VALOR INVÁLIDO PARA N.'
    END IF
CONTAINS
    SUBROUTINE FATORIAL(N,F)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: N
    INTEGER, INTENT(OUT) :: F = 1
    INTEGER :: I
            IF (N>0) THEN
                    DO I = 2,N
                        F = F*I
                    END DO
            ELSE IF (N==0) THEN
                    F = 1
            END IF
    END SUBROUTINE FATORIAL
END PROGRAM TESTE
```

7.2 Funções

Uma função em FORTRAN 90 é declarada de acordo com a seguinte sintaxe

```
[ RECURSIVE ] [<tipo>] FUNCTION <nome-função> [ ( <lista-argumentos> ) ] RESULT (<nome>)

   [ <declarações-argumentos> ]

   [ <declarações-internas> ]

   [ <comandos> ]

END FUNCTION <nome-função>

<lista-argumentos> ::= <argumento-1> [, <argumento-2> [ ... [, <argumento-n> ] ] ]
```

Semântica:

1. RECURSIVE é um atributo opcional da função e indica que ela é *recursiva*, isto é, pode existir uma invocação de <nome-função> dentro dos <comandos>, inclusive de forma encadeada;

2. <tipo> é um tipo pré-definido na linguagem ou definido pelo usuário, associado à função;

3. <argumento-1> a <argumento-n> é uma lista de argumentos, opcional, cada argumento separado por vírgulas;

4. <nome> é o nome de uma variável de tipo <tipo>, à qual será atribuído um valor em pelo menos um dos comandos existentes dentro da função. Se <tipo> foi omitido, então a variável <nome> deve ser declarada em <declarações-argumentos>;

5. <declarações-argumentos> é a declaração dos tipos de cada argumento, os quais podem ter o atributo INTENT (ver §3.6.11);

6. <declarações-internas> é a declaração das variáveis que existem *apenas* no escopo da função;

7. <comandos> é uma seqüência de comandos da linguagem, os quais serão executados quando da invocação de <nome-função>; se um desses comandos for o comando RETURN, a execução da função é encerrada.

7.3 Recursividade

Um subprograma em FORTRAN 90 pode ser *recursivo*, i.e. dentro do seu código pode haver uma chamada a si mesmo. É possível, também, ocorrer uma chamada recursiva dita *indireta*: um subprograma A chama um subprograma B o qual, por sua vez, chama A.

Podem existir inúmeros níveis de recursão, limitados apenas pela memória disponível no computador. É importante notar que, sempre que se invoca um subprograma, são alocadas na memória do computador todas as variáveis declaradas internamente ao subprograma. Em um subprograma recursivo, é possível ocorrer um erro de execução, causado pela inexistência de memória para alocar as variáveis internas ao subprograma; é importante que o algoritmo recursivo, contenha um critério de parada das chamadas recursivas.

Exemplo 7.3 *Cálculo do fatorial de um número de forma recursiva.*

No Exemplo 7.2, mostramos uma subrotina que calcula o fatorial n!, obtido através da multiplicação sucessiva dos inteiros 1, 2, ..., *n utilizando se o comando de repetição DO..END DO. No entanto, n! também pode ser definido de forma recursiva,*

$$n! = \begin{cases} n \cdot (n-1)!, & n > 0 \\ 1, & n = 0 \end{cases}$$

Note que a definição recursiva precisa estipular um valor de parada (i.e., para um determinado valor de n, o valor de n! é conhecido). Um programa que implementa a definição recursiva acima pode ser expresso como

```
PROGRAM TESTE
IMPLICIT NONE
INTEGER :: F, N
    PRINT *,'N?'
    READ *,N
    IF (N>=0) THEN
            F=FATORIAL(N)
            WRITE(*,'("FATORIAL(",I12,")=",I12)')N,F
    ELSE
            PRINT *,'VALOR INVÁLIDO PARA N.'
    END IF
```

```
    CONTAINS
        RECURSIVE INTEGER FUNCTION FATORIAL(N) RESULT (F)
        IMPLICIT NONE
        INTEGER, INTENT(IN) :: N
                IF (N>0) THEN
                        F = FATORIAL(N-1)*N
                        PRINT *,'N=',N,' F=',F
                ELSE IF (N==0) THEN
                        F = 1
                        PRINT *,'N=',N,' F=',F
                END IF
        END FUNCTION FATORIAL
    END PROGRAM TESTE
```

*O programa acima utiliza uma FUNCTION declarada internamente a ele. Essa FUNCTION, de tipo INTEGER, é declarada com o atributo RECURSIVE, de forma a permitir a chamada à ela mesma no comando F = FATORIAL(N-1)*N. Ao se digitar o valor 5, quando da leitura de N, no programa principal, teremos a seguinte saída:*

```
 N=           0  F=           1
 N=           1  F=           1
 N=           2  F=           2
 N=           3  F=           6
 N=           4  F=          24
 N=           5  F=         120
FATORIAL(           5)=         120
```

*Observe a seqüência de chamadas à FUNCTION: cada vez que ela é invocada, o comando IF é executado. Para N maior do que 0, exatamente N chamadas serão feitas dentro da função: FATORIAL(4), FATORIAL(3), FATORIAL(2), FATORIAL(1) e FATORIAL(0). Apenas quando essa última chamada é feita, a variável F recebe um valor, no caso, 1. A partir daí, é calculado F = FATORIAL(0)*1 = 1*1 = 1; F = FATORIAL(1)*2 = 1*2 = 2; F = FATORIAL(2)*3 = 2*3 = 6; F = FATORIAL(3)*4 = 6*4 = 24 e, finalmente, F = FATORIAL(4)*5 = 24*5 = 120.*

Observe que, apesar da função fatorial retornar valores inteiros, o uso do tipo INTEGER limita substancialmente o intervalo de números passíveis de utilização, devido ao intervalo de representação de números daquele tipo. Particularmente, o programa acima não consegue calcular valores de $n!$ para $n > 15$. A solução, nesse caso, seria declarar a FUNCTION como sendo do tipo DOUBLE PRECISION.

7.4 Definição de interface de subprogramas

A fim de que um subprograma em FORTRAN 90 possa ser invocado utilizando-se de recursos específicos da linguagem, tais como a utilização de argumentos opcionais, nomes genéricos associados a (mais de) um subprograma e operadores associados a subprogramas, é necessário que seja declarada uma *interface* para aquele subprograma. Por interface, entende-se a utilização de um bloco INTERFACE...END INTERFACE que informa a declaração de um subprograma, com os seus argumentos. A sintaxe desse bloco é a seguinte:

```
INTERFACE [ <nome-genérico> | OPERATOR (<símbolo>) | ASSIGNMENT (=) ]

    [ <declaração-subprograma> ]

    [ MODULE PROCEDURE <nome-subprograma> ]

END INTERFACE

<simbolo> ::= + | - | * | / | ** | .<nome>.
```

Semântica:

1. <nome-genérico> é um atributo opcional que especifica um nome pelo qual os subprogramas listados nesse bloco serão referenciados;

2. OPERATOR (<símbolo>) indica que os subprogramas listados nesse bloco serão invocados quando <símbolo> for encontrado em uma expressão. Se o subprograma pertence a um módulo (ver §7.5), então ele só pode ser referenciado na forma MODULE PROCEDURE <nome-subprograma>. Um novo operador é definido usando o símbolo .<nome>., onde <nome> é um nome definido segundo as regras da linguagem (ver §3.2). Regras específicas para a declaração de tais subprogramas são apresentadas em §7.4.1;

3. ASSIGNMENT (=) indica que os subprogramas listados nesse bloco serão invocados numa operação de atribuição, através do símbolo =. Regras específicas para a declaração de tais subprogramas são apresentadas em §7.4.1.

O exemplo a seguir mostra como definir a interface de subprogramas.

Exemplo 7.4 *Definição da interface de subprogramas.*

```
PROGRAM TESTE
INTERFACE F
        SUBROUTINE F_REAL(X)
        REAL, INTENT(INOUT) :: X
        END SUBROUTINE F_REAL

        SUBROUTINE F_DOUBLE(X)
        DOUBLE PRECISION, INTENT(INOUT) :: X
        END SUBROUTINE F_DOUBLE
END INTERFACE

REAL :: Y = -7.0
DOUBLE PRECISION :: Z = -10.0D0

        CALL F(Y) ! Após a chamada, Y = -0.8750000
        CALL F(Z) ! Após a chamada, Z = -0.9090909090909090

CONTAINS
        SUBROUTINE F_REAL(X)
        REAL, INTENT(INOUT) :: X
                X = X/(1.0+ABS(X))
        END SUBROUTINE F_REAL

        SUBROUTINE F_DOUBLE(X)
        DOUBLE PRECISION, INTENT(INOUT) :: X
                X = X/(1.0D0+DABS(X))
        END SUBROUTINE F_DOUBLE
END PROGRAM TESTE
```

O programa acima contém um bloco INTERFACE que define um nome genérico, F, pelo qual as subrotinas F_REAL e F_DOUBLE serão invocadas. Qual delas será invocada é decidido pelo compilador, comparando o tipo do argumento transmitido; na primeira chamada, como Y é do tipo REAL, então na chamada CALL F(Y) é invocada a subrotina F_REAL e, na segunda chamada, é invocada a subrotina F_DOUBLE, já que Z é do tipo DOUBLE PRECISION.

7.4.1 Operadores definidos pelo usuário

Em FORTRAN 90, é possível se definir novos operadores (por exemplo, a solução de um sistema de equações lineares pode ser expresso por um operador) e, também, se efetuar a *sobrecarga de*

operadores já existentes (por exemplo, que efetue a operação equivalente à soma sobre dados de um tipo definido pelo usuário). É possível, também, definir-se como é feita uma atribuição de um valor a uma variável.

Essas definições são feitas utilizando-se um bloco `INTERFACE...END INTERFACE`; nesse caso, deve-se fazer uso das palavras-chave `OPERATOR (<símbolo>)` ou `ASSIGNMENT (=)`. No entanto, existem regras específicas que um subprograma deve atender para que possa definir um operador, as quais são as seguintes:

Definição de operador: O subprograma só pode ser uma `FUNCTION` *externa*, com apenas *um* ou *dois* argumentos *não-opcionais*, declarados com o atributo `INTENT(IN)`. O resultado não pode ser uma *"string"* de tamanho assumido. Um operador que tenha um só argumento é dito *unário*; um operador com dois argumentos é dito *binário*.

Definição de atribuição: O subprograma só pode ser uma `SUBROUTINE` *externa*, com apenas *dois* argumentos *não-opcionais*. O primeiro argumento deve ser declarado com o atributo `INTENT(OUT)` ou `INTENT(INOUT)`; o segundo argumento deve ser declarado com o atributo `INTENT(IN)`.

Deve-se salientar que esses subprogramas também podem ser invocados da forma usual de invocação de subprogramas; porém, quando for encontrado `<símbolo>` numa expressão ou uma atribuição, eles serão invocados, desde que os tipos dos operadores sejam idênticos aos declarados nos subprogramas.

O exemplo abaixo ilustra a utilização de operadores.

Exemplo 7.5 *Definição e uso de operadores.*

```
PROGRAM TESTE
        INTERFACE OPERATOR (+)
                INTEGER FUNCTION SOMA(A,B) RESULT (C)
                        INTEGER, INTENT(IN) :: A, B
                END FUNCTION SOMA
        END INTERFACE

        INTERFACE ASSIGNMENT (=)
                SUBROUTINE ATRIBUI(A,B)
                        INTEGER, INTENT(OUT) :: A
                        INTEGER, INTENT(IN) :: B
                END SUBROUTINE ATRIBUI
        END INTERFACE

        INTEGER :: I, J, K
        REAL :: U

                U = -4.0     ! U = -4.0
                I = 1        ! I = 1
                J = -1       ! J = 1
                K = I+J      ! K = 2

END PROGRAM TESTE

SUBROUTINE ATRIBUI(A,B)
        INTEGER, INTENT(OUT) :: A
        INTEGER, INTENT(IN) :: B
                A = ABS(B)
END SUBROUTINE ATRIBUI

INTEGER FUNCTION SOMA(A,B) RESULT (C)
        INTEGER, INTENT(IN) :: A, B
                C = A*B + B*A
END FUNCTION SOMA
```

No exemplo acima, um único arquivo, contendo o código-fonte do programa TESTE e de dois subprogramas externos, ATRIBUI e SOMA, define um operador de atribuição e uma sobrecarga ao operador pré-definido para a adição. Note que a atribuição de um valor de tipo INTEGER a uma variável de mesmo tipo será sempre feita, nesse exemplo, tomando-se a parte absoluta do valor; além disso, a soma é feita de acordo com o definido na função SOMA. Observe que a atribuição U = -4.0 é feita de acordo com a definição da linguagem, já que o tipo da variável U e da expressão do lado direito do sinal de atribuição = não é INTEGER.

7.5 Módulos

Um módulo é uma das possíveis seções de código de um programa FORTRAN 90 (ver capítulo 2). O uso dos módulos permite uma melhor estruturação e controle de bibliotecas de subprogramas na linguagem.

Dentro de um módulo, pode-se declarar estruturas de dados, variáveis, subrotinas e funções; porém, não pode haver qualquer comando executável dentro de um módulo. A sintaxe de declaração de um módulo é a seguinte:

```
MODULE <nome-do-módulo>

    [ USE <nome-de-módulo> ]
    [ <declarações-de-tipos> ]
    [ <declarações-de-variáveis> ]

    [ CONTAINS

        [ <declaração-de-subprograma> ]

    ]

END MODULE <nome-do-módulo>
```

Semântica:

1. <nome-do-módulo> é o nome pelo qual ele será referenciado num comando USE;
2. <declarações-de-tipos> são declarações de tipos definidos pelo usuário, os quais serão visíveis por outras seções, desde que não tenham sido declaradas com o atributo PRIVATE (ver §3.6.3);
3. <declarações-de-variáveis> são declarações de variáveis, as quais serão visíveis por outras seções, desde que não tenham sido declaradas com o atributo PRIVATE (ver §3.6.3);
4. Se um módulo contém subprogramas, então eles devem ser declarados após a palavra-chave CONTAINS.

O exemplo a seguir ilustra como definir um módulo.

Exemplo 7.6 *Definição de um módulo, com seção declarativa de variáveis e de subprogramas.*

```
MODULE VETOR_3D_MODULO
   TYPE VETOR_3D
      REAL :: X, Y, Z
   END TYPE VETOR_3D

   CONTAINS
        SUBROUTINE ATRIBUI_VETOR(A,B)
            TYPE (VETOR_3D), INTENT(OUT) :: A
            REAL, DIMENSION(3) , INTENT(IN) :: B
```

```
            A%X = B(1)
            A%Y = B(2)
            A%Z = B(3)
        END SUBROUTINE ATRIBUI_VETOR

        TYPE (VETOR_3D) FUNCTION ADICAO_VETORIAL(A,B) RESULT (C)
            TYPE (VETOR_3D), INTENT(IN) :: A, B
            C%X = A%X + B%X
            C%Y = A%Y + B%Y
            C%Z = A%Z + B%Z
        END FUNCTION ADICAO_VETORIAL

        TYPE (VETOR_3D) FUNCTION SUBTRACAO_VETORIAL(A,B) RESULT (C)
            TYPE (VETOR_3D), INTENT(IN) :: A, B
            C%X = A%X - B%X
            C%Y = A%Y - B%Y
            C%Z = A%Z - B%Z
        END FUNCTION SUBTRACAO_VETORIAL

        TYPE (VETOR_3D) FUNCTION INVERTE_VETOR(A) RESULT (B)
            TYPE (VETOR_3D), INTENT(IN) :: A
            B%X = -A%X
            B%Y = -A%Y
            B%Z = -A%Z
        END FUNCTION INVERTE_VETOR

        REAL FUNCTION PRODUTO_ESCALAR(A,B) RESULT (ALPHA)
            TYPE (VETOR_3D), INTENT(IN) :: A, B
            ALPHA = A%X*B%X+A%Y*B%Y+A%Z*B%Z
        END FUNCTION PRODUTO_ESCALAR

        TYPE (VETOR_3D) FUNCTION PRODUTO_VETORIAL(A,B) RESULT (C)
            TYPE (VETOR_3D), INTENT(IN) :: A, B
            C%X = A%Y*B%Z - A%Z*B%Y
            C%Y = A%Z*B%X - A%X*B%Z
            C%Z = A%X*B%Y - A%Y*B%X
        END FUNCTION PRODUTO_VETORIAL
END MODULE VETOR_3D_MODULO
```

O módulo VETOR_3D_MODULO define um tipo derivado VETOR_3D, o qual pode ser utilizado dentro do módulo e, também, por um programa principal, subprograma e módulo que se refiram a esse módulo através do comando USE VETOR_3D_MODULO.

O comando USE permite que se faça referência a tipos de dados, dados (que não tenham sido declarados com o atributo PRIVATE – ver §3.6.3) e subprogramas definidos em um módulo. Sua sintaxe é a seguinte:

```
USE <nome-módulo> [,  <lista-nomes-locais> | ONLY : [ <seleção> ]  ]

<lista-nomes-locais> ::= <nome-local> => <nome-no-módulo> [,
                         [ ... [, <nome-local> => <nome-no-módulo> ] ] ]

<seleção> ::= <lista-nomes-no-módulo> | <lista-nomes-locais> |
              OPERATOR (<operador>) | ASSIGNMENT(=)

<lista-nomes-no-módulo> ::= <nome-no-módulo> [, [ ... [, <nome-no-módulo> ] ] ]
```

Semântica:

1. <nome-módulo> é o nome de um módulo, declarado num comando MODULE...END MODULE;

2. Se for utilizada a palavra-chave ONLY, então dentro da seção de código onde o comando USE foi usado só poderá fazer referências a subprogramas declarados em <nome-módulo> cujos nomes estejam listados em <seleção>;

3. <lista-nomes-locais> é uma lista contendo os nomes locais pelos quais subprogramas declarados num módulo serão acessados;

4. <lista-nomes-no-módulo> é uma lista contendo os nomes de subprogramas declarados num módulo.

O próximo exemplo mostra como utilizar definições de um módulo dentro de um programa.

Exemplo 7.7 *Utilização de um módulo dentro de um programa.*

```
PROGRAM TESTE_VETOR_3D
    USE VETOR_3D_MODULO, ONLY : VETOR_3D, ATRIBUI_VETOR, &
                                ESCALAR => PRODUTO_ESCALAR, PRODUTO_VETORIAL, &
                                ADICAO_VETORIAL
    IMPLICIT NONE
    TYPE (VETOR_3D) :: U,V,W
    REAL :: ALPHA, BETA
    INTERFACE OPERATOR (+)
            MODULE PROCEDURE ADICAO_VETORIAL
    END INTERFACE
    INTERFACE OPERATOR (.o.)
            MODULE PROCEDURE ESCALAR
    END INTERFACE
    INTERFACE OPERATOR (.x.)
            MODULE PROCEDURE PRODUTO_VETORIAL
    END INTERFACE
    INTERFACE ASSIGNMENT (=)
            MODULE PROCEDURE ATRIBUI_VETOR
    END INTERFACE

        U = (/ 1.0,  1.0, 1.0 /)
        V = (/ 0.0, -1.0, 1.0 /)
        ALPHA = ESCALAR(U,V)          ! ALPHA = 0.0
        BETA = U .o. V                ! BETA = 0.0
        W = U .x. (U+V)               ! W = (/ 2.0, -1.0, -1.0 /)

END PROGRAM TESTE_VETOR_3D
```

O programa TESTE_VETOR_3D pode fazer uso das definições de tipo e de subprogramas no módulo VETOR_3D_MODULO através do comando USE VETOR_3D_MODULO. A lista após a palavra-chave ONLY informa que, daquele módulo, apenas a definição de tipo VETOR_3D e os subprogramas ATRIBUI_VETOR, ADICAO_VETORIAL, PRODUTO_ESCALAR e PRODUTO_VETORIAL poderão ser utilizados dentro do programa TESTE_VETOR_3D; além disso, o subprograma PRODUTO_ESCALAR só pode ser invocado através do nome local ESCALAR. Quaisquer outras referências a subprogramas presentes no módulo VETOR_3D_MODULO incorrerão em erro de compilação.

São definidos, também, quatro operadores: um deles é uma sobrecarga do operador +, que efetuará a operação de soma vetorial, componente a componente; um corresponde à atribuição de valores a um vetor, expressos como um arranjo uni-dimensional de três elementos; e os outros dois são novos operadores, definidos como .o. e .x., correspondendo aos produtos escalar e vetorial. Note que, como foi definido um nome local para o subprograma PRODUTO_ESCALAR, então o seu nome local é que deve ser utilizado na definição do operador .o. Além disso, observe que, conforme citado em §7.4.1, pode-se fazer referência a subprogramas associados a operadores da forma usual, como no comando ALPHA = ESCALAR(U,V).

7.6 Inicialização de blocos COMMON nomeados

Blocos COMMON nomeados (ver §3.7) podem ser inicializados através do comando não-executável BLOCKDATA...END BLOCKDATA. O comando BLOCKDATA...END BLOCKDATA deve ser encarado como se fosse um subprograma e, como tal, deve aparecer apenas dentro de um PROGRAM...END PROGRAM (como um subprograma interno, após o comando CONTAINS) ou MODULE...END MODULE, ou ainda como um trecho de código separado, como se fosse um subprograma externo. Antes de se iniciar a execução do programa, os dados listados nos blocos COMMON nomeados serão inicializados; apenas um comando BLOCKDATA...END BLOCKDATA pode existir por seção de código.

A sintaxe do comando é a seguinte:

```
BLOCKDATA [<nome>]

    [ <declarações> ]

END [ BLOCKDATA [<nome>] ]
```

Semântica:

1. <nome> é o nome opcional do comando. Se for especificado, então o comando deve ser terminado como END BLOCKDATA <nome>;
2. <declarações> é um conjunto de declarações de variáveis, blocos COMMON nomeados, comandos DATA (ver §3.6.4), USE; não pode haver um comando INTERFACE e não podem ser utilizados os seguintes atributos de declaração: ALLOCATABLE, EXTERNAL, INTENT, OPTIONAL, PRIVATE, PUBLIC.

O exemplo abaixo mostra a utilização do comando.

Exemplo 7.8 *Utilização do comando BLOCKDATA*

```
PROGRAM TESTE
        INTEGER, DIMENSION(3) :: A
        REAL, DIMENSION(4) :: B
        COMMON /ARRANJOS/ A, B
                CALL SUB_A
                WRITE(*,*)A
                WRITE(*,*)B
        CONTAINS
                SUBROUTINE SUB_A
                        INTEGER, DIMENSION(3) :: X
                        REAL, DIMENSION(4) :: Y
                        COMMON /ARRANJOS/ X, Y
                                Y(1:2) = 2*X(1:2)
                                Y(3:4) = 2*X(2:3)
                END SUBROUTINE SUB_A

                BLOCKDATA DADOS
                        INTEGER, DIMENSION(3) :: A
                        REAL, DIMENSION(4) :: B
                        COMMON /ARRANJOS/ A, B
                        DATA A / 1, 2, 4 /
                        DATA B / 1.0, -1.0, 2.0, -2.0 /
                END BLOCKDATA DADOS
END PROGRAM TESTE
```

Note que o bloco COMMON de nome ARRANJOS está sendo usado com a finalidade de receber e transmitir dados do programa principal para a subrotina SUB_A. Ao final da execução desse programa, o arranjo A conterá os valores 1, 2, 4 (sem sofrer alteração dos valores atribuídos a ele dentro do comando BLOCKDATA) e o arranjo B conterá os valores 1.0, 2.0, 2.0, 4.0, tendo sido modificados dentro da subrotina SUB_A.

Capítulo 8

Intrínsecos

Intrínsecos são subprogramas disponibilizados ao programador pela linguagem. Os intrínsecos permitem, por exemplo, inquirir sobre determinadas características da linguagem, calcular valores de funções matemáticas típicas (como o seno) e manipular *"strings"* de caracteres, dentre outras. A seguir, descreveremos os intrínsecos disponíveis na linguagem FORTRAN 90.

8.1 Intrínsecos matemáticos

8.1.1 ACOS

Sintaxe: `ACOS(X)`

Sinopse: Retorna o valor do arco-coseno de `X`;

Restrições: O argumento `X` deve satisfazer $|\texttt{X}| \leq 1$; o valor retornado é $0 \leq \texttt{ACOS(X)} \leq \pi$;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos:

Nome	Tipo do argumento
DACOS	DOUBLE PRECISION

8.1.2 ASIN

Sintaxe: `ASIN(X)`

Sinopse: Retorna o valor do arco-seno de `X`;

Restrições: O argumento `X` deve satisfazer $|\texttt{X}| \leq 1$; o valor retornado é $-\pi/2 \leq \texttt{ASIN(X)} \leq \pi/2$;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos:

Nome	Tipo do argumento
DASIN	DOUBLE PRECISION

8.1.3 ATAN

Sintaxe: `ATAN(X)`

Sinopse: Retorna o valor do arco-tangente de `X`;

Restrições: O valor retornado é $-\pi/2 \leq$ `ACOS(X)` $\leq \pi/2$;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos:

Nome	Tipo do argumento
DATAN	DOUBLE PRECISION

8.1.4 ATAN2

Sintaxe: `ATAN2(Y,X)`

Sinopse: Retorna o valor principal do número complexo (`X`, `Y`), expresso em radianos;

Restrições: O argumento `X` deve atender a `X` $\neq 0$; o valor retornado é $-\pi \leq$ `ATAN2(Y,X)` $\leq \pi$;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; ambos os argumentos devem ser do mesmo tipo;

Tipo do valor retornado: Igual ao tipo dos argumentos;

Nomes específicos:

Nome	Tipo do argumento
DATAN2	DOUBLE PRECISION

8.1.5 CONJG

Sintaxe: `CONJG(Z)`

Sinopse: Retorna o conjugado do número complexo `Z`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `COMPLEX`;

Tipo do valor retornado: `COMPLEX`;

Nomes específicos: Nenhum.

8.1.6 COS

Sintaxe: `COS(X)`

Sinopse: Retorna o valor do coseno de `X`;

Restrições: O argumento `X` é o ângulo em radianos; o valor retornado é $-1 \leq$ `COS(X)` ≤ 1;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`, `COMPLEX`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos:

Nome	Tipo do argumento
CCOS	COMPLEX
DCOS	DOUBLE PRECISION

8.1.7 COSH

Sintaxe: `COSH(X)`

Sinopse: Retorna o valor do coseno hiperbólico de `X`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos:

Nome	Tipo do argumento
DCOSH	DOUBLE PRECISION

8.1.8 DIM

Sintaxe: `DIM(X,Y)`

Sinopse: Retorna o valor $\max(\mathtt{X} - \mathtt{Y}, 0)$;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos: Nenhum.

8.1.9 DOT_PRODUCT

Sintaxe: `DOT_PRODUCT(A,B)`

Sinopse: Retorna o valor do produto escalar entre os arranjos `A` e `B`;

Restrições: O argumento `B` é opcional. Se `A` e `B` são valores numéricos e não são do tipo `COMPLEX`, então o valor retornado é `SUM(A*B)`. Se são valores numéricos do tipo `COMPLEX`, o resultado é `SUM(CONJG(A)*B)`. Se são valores lógicos, então o resultado é `ANY(A .AND. B)`;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `LOGICAL`; devem ser arranjos uni-dimensionais de mesmo tamanho e do mesmo tipo;

Tipo do valor retornado: Um escalar, igual aos tipos dos argumentos;

Nomes específicos: Nenhum.

8.1.10 DPROD

Sintaxe: `DPROD(X,Y)`

Sinopse: Retorna o valor do produto entre `X` e `Y`, expresso em precisão dupla;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; podem ser arranjos;

Tipo do valor retornado: Igual aos tipos dos argumentos;

Nomes específicos: Nenhum.

8.1.11 EXP

Sintaxe: `EXP(X)`

Sinopse: Retorna o valor de $e^{\texttt{X}}$;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`, `COMPLEX`; podem ser arranjos;

Tipo do valor retornado: Igual aos tipos dos argumentos;

Nomes específicos:

Nome	Tipo do argumento
CEXP	COMPLEX
DEXP	DOUBLE PRECISION

8.1.12 LOG

Sintaxe: `LOG(X)`

Sinopse: Retorna o valor de $ln(\texttt{X})$;

Restrições: O argumento deve satisfazer $\texttt{X} > 0$; se o argumento for do tipo `COMPLEX`, o resultado é um número complexo cuja parte real é o valor principal de `X` e a parte imaginária é um número ω tal que $-\pi < \omega < \pi$;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`, `COMPLEX`; podem ser arranjos;

Tipo do valor retornado: Igual aos tipos dos argumentos;

Nomes específicos:

Nome	Tipo do argumento
ALOG	REAL
CLOG	COMPLEX
DLOG	DOUBLE PRECISION

8.1.13 LOG10

Sintaxe: LOG10(X)

Sinopse: Retorna o valor de $\log_{10}$(X);

Restrições: O argumento deve satisfazer X > 0;

Tipo de intrínseco: Função;

Tipo de argumento: REAL, DOUBLE PRECISION; podem ser arranjos;

Tipo do valor retornado: Igual aos tipos dos argumentos;

Nomes específicos:

Nome	Tipo do argumento
ALOG10	REAL
DLOG10	DOUBLE PRECISION

8.1.14 MATMUL

Sintaxe: MATMUL(A,B)

Sinopse: Retorna o valor do produto matricial entre os arranjos A e B;

Restrições: Os argumentos podem ser arranjos uni- ou bi-dimensionais. Se A for uni-dimensional, então B deve ser bi-dimensional; se A for bi-dimensional, então B deve ser uni-dimensional. A segunda (ou única) dimensão de A deve ser igual à primeira (ou única) dimensão de B.

A forma e tamanho do arranjo resultado segue as regras usuais da multiplicação de matrizes. Se A é um arranjo de tamanho DIMENSION(M,K) e B é de tamanho DIMENSION(K,N), então o resultado é um arranjo de tamanho DIMENSION(M,N); se A é um arranjo de tamanho DIMENSION(M) e B é de tamanho DIMENSION(M,N), então o resultado é um arranjo de tamanho DIMENSION(N); se A é um arranjo de tamanho DIMENSION(M,N) e B é de tamanho DIMENSION(N), então o resultado é um arranjo de tamanho DIMENSION(M).

Se A e B são numéricos, então o elemento (i,j) do resultado é SUM(A(i,:)*B(:,j)), SUM(A(:)*B(:,j)) ou SUM(A(i,:)*B(:)); se A e B são arranjos do tipo LOGICAL, então o elemento (i,j) do resultado é ANY(A(i,:) .AND. B(:,j)), ANY(A(:) .AND. B(:,j)) ou ANY(A(i,:) .AND. B(:))

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL; devem ser arranjos do mesmo tipo;

Tipo do valor retornado: Igual aos tipos dos argumentos;

Nomes específicos: Nenhum.

8.1.15 MOD

Sintaxe: MOD(A,B)

Sinopse: Retorna o valor do resto da divisão de A por B;

Restrições: Se B $\neq$ 0, o resultado é A-INT(A/B)*B; se B=0, o resultado é dependente da implementação;

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER, REAL, DOUBLE PRECISION; podem ser arranjos;

Tipo do valor retornado: Igual ao tipo de A;

Nomes específicos:

Nome	Tipo do argumento
AMOD	REAL
DMOD	DOUBLE PRECISION

8.1.16 MODULO

Sintaxe: `MODULO(A,B)`

Sinopse: Retorna o valor do resto da divisão de `A` por `B`;

Restrições: Se `B` $\neq 0$, o resultado é `A-FLOOR(A/B)*B`; se `B=0`, o resultado é dependente da implementação;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`; podem ser arranjos;

Tipo do valor retornado: Igual ao tipo de `A`;

Nomes específicos: Nenhum.

8.1.17 PRODUCT

Sintaxe: `PRODUCT(A,<dimensão>,<expressão-lógica>)`

Sinopse: Retorna o produto dos elementos do arranjo `A` que atendam a `<expressão-lógica>` (caso seja especificada) e, opcionalmente, ao longo de uma dimensão, se o argumento `<dimensão>` for especificado;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `<expressão-lógica>` é do tipo `LOGICAL`; `<dimensão>` é do tipo `INTEGER`;

Tipo do valor retornado: Igual ao tipo de `A`;

Nomes específicos: Nenhum;

Exemplo de uso: `PRODUCT(A,1,A>1.0)`

8.1.18 RANDOM_NUMBER

Sintaxe: `CALL RANDOM_NUMBER(A)`

Sinopse: Retorna um número pseudo-aleatório `A`;

Restrições: O valor retornado satisfaz $0 \leq$ `A` < 1;

Tipo de intrínseco: Subrotina;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um arranjo;

Tipo do valor retornado: Igual ao tipo de `A`;

Nomes específicos: Nenhum.

8.1.19 RANDOM_SEED

Sintaxe: `CALL RANDOM_SEED(N,S,C)`

Sinopse: Inicializa ou interroga o gerador de números pseudo-aleatórios. Os argumentos são todos opcionais; caso se chame o intrínseco sem especificar qualquer argumento, o gerador pseudo-aleatório é inicializado com um valor dependente da implementação. O argumento de saída `N`, de tipo `INTEGER`, retorna o tamanho do arranjo inicializador; o argumento de entrada `S`, de tipo `INTEGER`, é um arranjo uni-dimensional de tamanho maior ou igual a `N`; o argumento de saída `C`, de tipo `INTEGER`, é um arranjo uni-dimensional de tamanho maior ou igual a `N` e contém o valor atual da semente geradora.

Restrições: Nenhuma;

Tipo de intrínseco: Subrotina;

Tipo de argumento: `INTEGER`;

Tipo do valor retornado: De acordo com a sinopse;

Nomes específicos: Nenhum.

8.1.20 SIGN

Sintaxe: `SIGN(A,B)`

Sinopse: Retorna o valor absoluto de `A` multiplicado pelo sinal de `B`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`; podem ser arranjos;

Tipo do valor retornado: Igual ao tipo de `A`;

Nomes específicos:

Nome	Tipo do argumento
ISIGN	INTEGER
DSIGN	DOUBLE PRECISION

8.1.21 SIN

Sintaxe: `SIN(X)`

Sinopse: Retorna o valor do seno de `X`;

Restrições: O argumento `X` é o ângulo em radianos; o valor retornado é $-1 \leq \texttt{SIN(X)} \leq 1$;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`, `COMPLEX`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos:

Nome	Tipo do argumento
CSIN	COMPLEX
DSIN	DOUBLE PRECISION

8.1.22 SINH

Sintaxe: `SINH(X)`

Sinopse: Retorna o valor do seno hiperbólico de `X`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos:

Nome	Tipo do argumento
DSINH	DOUBLE PRECISION

8.1.23 SUM

Sintaxe: `SUM(A,<dimensão>,<expressão-lógica>`

Sinopse: Retorna a soma dos elementos do arranjo `A` que atendam a `<expressão-lógica>` (caso seja especificada) e, opcionalmente, ao longo de uma dimensão, se o argumento `<dimensão>` for especificado;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `<expressão-lógica>` é do tipo `LOGICAL`; `<dimensão>` é do tipo `INTEGER`;

Tipo do valor retornado: Igual ao tipo de `A`;

Nomes específicos: Nenhum;

Exemplo de uso: `SUM(A,1,A>1.0)`

8.1.24 SQRT

Sintaxe: `SQRT(X)`

Sinopse: Retorna o valor da raiz quadrada de `X`;

Restrições: Se `X` é do tipo `REAL` ou `DOUBLE PRECISION`, então `X` > 0; se `X` é do tipo `COMPLEX`, então o resultado é o valor principal, com parte real maior ou igual a zero (nesse último caso, a parte imaginária do resultado é maior ou igual a zero);

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`, `COMPLEX`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos:

Nome	Tipo do argumento
CSQRT	COMPLEX
DSQRT	DOUBLE PRECISION

8.1.25 TAN

Sintaxe: `TAN(X)`

Sinopse: Retorna o valor da tangente de `X`;

Restrições: O argumento `X` é o ângulo em radianos;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos:

Nome	Tipo do argumento
DTAN	DOUBLE PRECISION

8.1.26 TANH

Sintaxe: `TANH(X)`

Sinopse: Retorna o valor da tangente hiperbólica de `X`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos:

Nome	Tipo do argumento
DTANH	DOUBLE PRECISION

8.2 Intrínsecos de manipulação de valores numéricos e lógicos

8.2.1 ABS

Sintaxe: `ABS(X)`

Sinopse: Retorna o valor absoluto do argumento;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Se o argumento for `INTEGER`, `REAL` ou `DOUBLE PRECISION`, igual ao tipo do argumento; se o argumento for `COMPLEX`, o resultado é o valor (dependente da implementação) de $\sqrt{\Re(X) + \Im(X)}$;

Nomes específicos:

Nome	Tipo do argumento
IABS	INTEGER
DABS	DOUBLE PRECISION
CABS	COMPLEX

8.2.2 AIMAG

Sintaxe: `AIMAG(Z)`

Sinopse: Retorna a parte imaginária do número complexo `Z`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `COMPLEX`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: `REAL`;

Nomes específicos: Nenhum.

8.2.3 AINT

Sintaxe: `AINT(X,KIND)`

Sinopse: Trunca o valor número real `X`; o argumento `KIND`, do tipo `INTEGER`, se presente, faz com que o resultado tenha o gênero especificado por `KIND`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: `REAL`;

Nomes específicos: Nenhum.

8.2.4 DINT

Sintaxe: `DINT(X,KIND)`

Sinopse: Trunca o valor número real `X`; o argumento `KIND`, do tipo `INTEGER`, se presente, faz com que o resultado tenha o gênero especificado por `KIND`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: `DOUBLE PRECISION`;

Nomes específicos: Nenhum.

8.2.5 ANINT

Sintaxe: `ANINT(X,KIND)`

Sinopse: Arredonda o valor número real `X`, i.e., o resultado é `AINT(X)+1` se a parte fracionária de `X` é maior do que 0.5, ou `AINT(X)` caso contrário. O argumento `KIND`, do tipo `INTEGER`, se presente, faz com que o resultado tenha o gênero especificado por `KIND`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: REAL, DOUBLE PRECISION; pode ser um escalar ou um arranjo;

Tipo do valor retornado: REAL;

Nomes específicos: Nenhum.

8.2.6 DNINT

Sintaxe: DNINT(X,KIND)

Sinopse: Arredonda o valor número real X, i.e., o resultado é DINT(X)+1 se a parte fracionária de X é maior do que 0.5, ou DINT(X) caso contrário. O argumento KIND, do tipo INTEGER, se presente, faz com que o resultado tenha o gênero especificado por KIND;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: REAL, DOUBLE PRECISION; pode ser um escalar ou um arranjo;

Tipo do valor retornado: DOUBLE PRECISION;

Nomes específicos: Nenhum.

8.2.7 CEILING

Sintaxe: CEILING(X)

Sinopse: Retorna o menor valor inteiro maior do que X;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: REAL, DOUBLE PRECISION; pode ser um escalar ou um arranjo;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum.

8.2.8 CMPLX

Sintaxe: CMPLX(X,Y,KIND)

Sinopse: Retorna um valor complexo cujas partes real e imaginária são X e Y, respectivamente. O argumento Y é opcional e, nesse caso, o número complexo terá parte imaginária igual a zero;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: REAL, DOUBLE PRECISION; pode ser um escalar ou um arranjo;

Tipo do valor retornado: COMPLEX;

Nomes específicos: Nenhum.

8.2.9 DBLE

Sintaxe: `DBLE(X)`

Sinopse: Converte para o tipo `DOUBLE PRECISION` o valor de `X`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: `DOUBLE PRECISION`;

Nomes específicos: Nenhum.

8.2.10 FLOOR

Sintaxe: `FLOOR(X)`

Sinopse: Retorna o maior valor inteiro menor do que `X`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: `INTEGER`;

Nomes específicos: Nenhum.

8.2.11 IAND

Sintaxe: `IAND(I,J)`

Sinopse: Retorna um número inteiro cujos *"bits"* são formados pela aplicação da multiplicação lógica (`.AND.`) a cada *"bit"* de `I` e o *"bit"* correspondente de `J`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: `INTEGER`;

Nomes específicos: Nenhum.

8.2.12 IEOR

Sintaxe: `IEOR(I,J)`

Sinopse: Retorna um número inteiro cujos *"bits"* são formados pela aplicação da operação lógica "ou exclusivo" (`.EQV.`) a cada *"bit"* de `I` e o *"bit"* correspondente de `J`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: `INTEGER`;

Nomes específicos: Nenhum.

8.2.13 INT

Sintaxe: `INT(X,KIND)`

Sinopse: Converte para o tipo `INTEGER` o valor de `X`; se `X` é do tipo `REAL` ou `DOUBLE PRECISION`, o resultado será apenas a parte inteira de `X`. O argumento opcional `KIND`, do tipo `INTEGER`, se presente, faz com que o valor retornado tenha o gênero especificado por `KIND`.

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: `INTEGER`;

Nomes específicos: Nenhum.

8.2.14 IOR

Sintaxe: `IOR(I,J)`

Sinopse: Retorna um número inteiro cujos *"bits"* são formados pela aplicação da adição lógica (`.OR.`) a cada *"bit"* de `I` e o *"bit"* correspondente de `J`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: `INTEGER`;

Nomes específicos: Nenhum.

8.2.15 LOGICAL

Sintaxe: `LOGICAL(L[,KIND=<gênero>])`

Sinopse: Converte o `KIND` do argumento `L`, de tipo `LOGICAL`, para o `KIND` *"default"* (4). Se o argumento `KIND=<gênero>` for especificado, então o `KIND` do argumento `L` passará a ser `<gênero>`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `L : LOGICAL; <gênero> : INTEGER`

Tipo do valor retornado: `LOGICAL`;

Nomes específicos: Nenhum.

8.2.16 MAX

Sintaxe: `MAX(<lista-de-argumentos>)`

Sinopse: Retorna o maior valor entre aqueles listados em `<lista-de-argumentos>`, cada valor separado por uma vírgula;

Restrições: Deve haver no mínimo dois argumentos em `<lista-de-argumentos>`;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`; são escalares e devem ter o mesmo tipo;

Tipo do valor retornado: O mesmo dos argumentos;

Nomes específicos: Nenhum.

8.2.17 MIN

Sintaxe: `MIN(<lista-de-argumentos>)`

Sinopse: Retorna o menor valor entre aqueles listados em `<lista-de-argumentos>`, cada valor separado por uma vírgula;

Restrições: Deve haver no mínimo dois argumentos em `<lista-de-argumentos>`;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`; são escalares e devem ter o mesmo tipo;

Tipo do valor retornado: O mesmo dos argumentos;

Nomes específicos: Nenhum.

8.2.18 NINT

Sintaxe: `NINT(X,KIND)`

Sinopse: Retorna o número de tipo `INTEGER` mais próximo do número real `X`; o argumento `KIND`, do tipo `INTEGER`, se presente, faz com que o resultado tenha o gênero especificado por `KIND`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um escalar ou um arranjo;

Tipo do valor retornado: `INTEGER`;

Nomes específicos: Nenhum.

8.2.19 IDNINT

Sintaxe: IDNINT(X,KIND)

Sinopse: Retorna o número de tipo INTEGER mais próximo do número real X; o argumento KIND, do tipo INTEGER, se presente, faz com que o resultado tenha o gênero especificado por KIND;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: DOUBLE PRECISION; pode ser um escalar ou um arranjo;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum.

8.2.20 REAL

Sintaxe: REAL(X,KIND)

Sinopse: Converte para o tipo REAL o valor de X. Se X é do tipo INTEGER ou DOUBLE PRECISION, o resultado será uma aproximação de X, dependente da implementação; se X é do tipo COMPLEX, o resultado é a parte real do argumento. O argumento opcional KIND, do tipo INTEGER, se presente, faz com que o valor retornado tenha o gênero especificado por KIND.

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER, REAL, DOUBLE PRECISION; pode ser um escalar ou um arranjo;

Tipo do valor retornado: INTEGER;

Nomes específicos:

Nome	Tipo do argumento
FLOAT	REAL
SNGL	DOUBLE PRECISION

8.2.21 SCALE

Sintaxe: SCALE(X,I)

Sinopse: Multiplica o número X por b^I, onde b é a base do modelo de operação aritmética associado ao tipo de X;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: REAL, DOUBLE PRECISION; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos: Nenhum.

8.2.22 TRANSFER

Sintaxe: TRANSFER(A,B[,SIZE=<tamanho>]

Sinopse: Transfere a representação binária interna do argumento A para a representação binária interna do tipo do argumento B. A e B podem ser arranjos.

Se B é um escalar e SIZE=<tamanho> não é especificado, então o resultado é um escalar.

Se B é um arranjo e SIZE=<tamanho> não é especificado, então o resultado é um arranjo tão pequeno quanto for possível, para se utilizar os *"bits"* de A.

Se SIZE=<tamanho> é especificado, o resultado é um arranjo de dimensão <tamanho>.

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: Qualquer tipo; pode ser um escalar ou um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento B;

Nomes específicos: Nenhum.

Exemplo de uso:

```
COMPLEX :: A
    A = TRANSFER((/3.0,-2.0/),(0.0,0.0)) ! A é o número complexo 3.0-2.0i
```

8.3 Intrínsecos de inquirição sobre dados

8.3.1 ALLOCATED

Sintaxe: ALLOCATED(A)

Sinopse: Retorna .TRUE. ou .FALSE. se o arranjo A encontra-se alocado ou não, respectivamente;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: um arranjo de qualquer tipo, declarado com o atributo ALLOCATABLE, ou uma variável declarada com o atributo POINTER;

Tipo do valor retornado: LOGICAL;

Nomes específicos: Nenhum.

8.3.2 PRESENT

Sintaxe: PRESENT(A)

Sinopse: Retorna .TRUE. ou .FALSE. se o argumento A de um subprograma, declarado com o atributo OPTIONAL, foi utilizado ou não na chamada ao subprograma, respectivamente;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: um argumento de qualquer tipo, de um subprograma, declarado com o atributo OPTIONAL;

Tipo do valor retornado: LOGICAL;

Nomes específicos: Nenhum.

Exemplo de uso:

```
SUBROUTINE INFORMA(NOME,CODIGO)
INTEGER, INTENT(IN) :: NOME
INTEGER, INTENT(IN), OPTIONAL :: CODIGO
    IF (PRESENT(CODIGO)) THEN
        PRINT *,CODIGO,"/",NOME
    ELSE
        PRINT *,NOME
    END IF
END SUBROUTINE INFORMA

    ! essa chamada imprimirá apenas CARLOS ADOLFO
    CALL INFORMA('CARLOS ADOLFO')

    ! essa chamada imprimirá QAX-178-PRT/MANOEL CARLOS
    CALL INFORMA('MANOEL CARLOS',CODIGO='QAX-178-PRT')
```

8.3.3 KIND

Sintaxe: KIND(X)

Sinopse: Retorna um valor do tipo INTEGER que indica o KIND associado ao tipo do argumento X;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: Uma variável ou expressão de qualquer tipo pré-definido na linguagem;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum.

Exemplo de uso:

```
INTEGER :: I
INTEGER(2) :: J
REAL :: K
REAL(4) :: L
REAL(8) :: M
LOGICAL :: N
CHARACTER :: O
DOUBLE PRECISION :: P
INTEGER, PARAMETER :: GENERO_INTEGER = KIND(I)             ! = 4
INTEGER, PARAMETER :: GENERO_INTEGER2 = KIND(J)            ! = 2
INTEGER, PARAMETER :: GENERO_REAL = KIND(K)                ! = 4
INTEGER, PARAMETER :: GENERO_REAL4 = KIND(L)               ! = 4
INTEGER, PARAMETER :: GENERO_REAL8 = KIND(M)               ! = 8
INTEGER, PARAMETER :: GENERO_LOGICAL = KIND(N)             ! = 4
INTEGER, PARAMETER :: GENERO_CHARACTER = KIND(O)           ! = 1
INTEGER, PARAMETER :: GENERO_DOUBLE_PRECISION = KIND(P)    ! = 8
```

8.4 Intrínsecos de manipulação do modelo de operação aritmética

8.4.1 DIGITS

Sintaxe: `DIGITS(X)`

Sinopse: Retorna o valor do número de dígitos no modelo de operação aritmética associado ao tipo de `X`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`; pode ser um arranjo;

Tipo do valor retornado: `INTEGER`;

Nomes específicos: Nenhum.

8.4.2 EPSILON

Sintaxe: `EPSILON(X)`

Sinopse: Retorna o valor ε no modelo de operação aritmética de ponto-flutuante associado ao tipo de `X`, tal que $1 + \varepsilon \neq 1$;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos: Nenhum.

8.4.3 FRACTION

Sintaxe: `FRACTION(X)`

Sinopse: Retorna $\texttt{X} \times b^{-e}$ no modelo de operação aritmética de ponto-flutuante associado ao tipo de `X`, onde b é a base do modelo e e é o expoente de `X`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos: Nenhum.

8.4.4 HUGE

Sintaxe: HUGE(X)

Sinopse: Retorna o maior número representável no modelo de operação aritmética associado ao tipo de X. Se X é de tipo INTEGER, retorna o valor $r^q - 1$, onde r é a base e q é o maior expoente representável; se X é do tipo REAL ou DOUBLE PRECISION, retorna o valor $(1 - b^{-p})b^{e_{max}}$, onde b é a base, p é o número máximo de dígitos e e_{max} é o maior expoente;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER, REAL, DOUBLE PRECISION; pode ser um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos: Nenhum.

8.4.5 MAXXEPONENT

Sintaxe: MAXEXPONENT(X)

Sinopse: Retorna o maior expoente no modelo de operação aritmética associado ao tipo de X;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: REAL, DOUBLE PRECISION; pode ser um arranjo;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum.

8.4.6 MINEXPONENT

Sintaxe: MINEXPONENT(X)

Sinopse: Retorna o menor expoente no modelo de operação aritmética associado ao tipo de X;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: REAL, DOUBLE PRECISION; pode ser um arranjo;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum.

8.4.7 NEAREST

Sintaxe: NEAREST(X,S)

Sinopse: Retorna o número mais próximo de X, no modelo de operação aritmética de ponto-flutuante associado ao tipo de X; se $S < 0$, então o número retornado é menor do que X, e maior se S>0;

Restrições: $S \neq 0$;

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos: Nenhum.

8.4.8 RADIX

Sintaxe: `RADIX(X)`

Sinopse: Retorna o valor da base do modelo de operação aritmética associado ao tipo de `X`;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`; pode ser um arranjo;

Tipo do valor retornado: `INTEGER`;

Nomes específicos: Nenhum.

8.4.9 RANGE

Sintaxe: `RANGE(X)`

Sinopse: Retorna o intervalo de variação do expoente do modelo de operação aritmética associado ao tipo de `X`, expresso por um único número e, tal que o intervalo é $[-e, e]$;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`; pode ser um arranjo;

Tipo do valor retornado: `INTEGER`;

Nomes específicos: Nenhum.

8.4.10 RRSPACING

Sintaxe: `RRSPACING(X)`

Sinopse: Retorna o recíproco do espaçamento relativo dos números próximos a `X`, no modelo de operação aritmética de ponto-flutuante associado ao tipo de `X`, i.e. $|\texttt{X} \times b^{-e}| \times b^{p}$, onde b é a base do modelo, e é o expoente de `X` e p é o número máximo de dígitos no modelo;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos: Nenhum.

8.4.11 SELECTED_INT_KIND

Sintaxe: `SELECTED_INT_KIND(R)`

Sinopse: Retorna o `KIND` de um tipo de dado `INTEGER` que permite a representação de números inteiros no intervalo $[-10^R, 10^R]$. Se não existir tal `KIND`, então o resultado retornado é `-1`; se existir mais do que um `KIND` para representar tais inteiros, então o `KIND` retornado é aquele que corresponde aos inteiros com o menor intervalo possível.

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`;

Tipo do valor retornado: `INTEGER, PARAMETER`;

Nomes específicos: Nenhum.

Exemplo de uso:

```
INTEGER, PARAMETER :: GENERO16 = SELECTED_INT_KIND(16) ! GENERO16 = -1 (não suportado)
INTEGER, PARAMETER :: GENERO4 = SELECTED_INT_KIND(4) ! GENERO4 = 2
```

8.4.12 SELECTED_REAL_KIND

Sintaxe: `SELECTED_REAL_KIND([P] [,R])`

Sinopse: Retorna o `KIND` de um tipo de dado `REAL` que permite a representação de números reais com `P` dígitos (de acordo com o retornado pelo intrínseco `PRECISION`) e com expoente no intervalo $[-10^R, 10^R]$, no mínimo, de acordo com o retornado pelo intrínseco `RANGE`. Se não existir tal `KIND`, então o resultado retornado é `-1`, se a precisão `P` não é suportada; `-2`, se o intervalo do expoente não é suportado; e `-3` se ambos não são suportados. Se existir mais do que um `KIND` para representar tais reais, então o `KIND` retornado é aquele que corresponda aos reais com a menor precisão possível.

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`;

Tipo do valor retornado: `INTEGER, PARAMETER`;

Nomes específicos: Nenhum.

Exemplo de uso:

```
INTEGER, PARAMETER :: GENERO1 = SELECTED_REAL_KIND(3,15)    ! GENERO1 =  4
INTEGER, PARAMETER :: GENERO2 = SELECTED_REAL_KIND(6,38)    ! GENERO2 =  8
INTEGER, PARAMETER :: GENERO3 = SELECTED_REAL_KIND(15,307) ! GENERO3 =  8
INTEGER, PARAMETER :: GENERO4 = SELECTED_REAL_KIND(20,500) ! GENERO4 = -3
```

8.4.13 SET_EXPONENT

Sintaxe: `SET_EXPONENT(X,I)`

Sinopse: Retorna o valor um número, no modelo de operação aritmética de ponto-flutuante associado ao tipo de `X`, cuja parte fracionária é a mesma de `X` e o expoente é `I`, i.e. $\texttt{X} \times b^{\texttt{I}-e}$, onde b é a base do modelo e e é o expoente de `X`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `X` pode ser `REAL` ou `DOUBLE PRECISION` e `I` deve ser de tipo `INTEGER`; `X` pode ser um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento `X`;

Nomes específicos: Nenhum.

8.4.14 SPACING

Sintaxe: `SPACING(X)`

Sinopse: Retorna o espaçamento absoluto dos números próximos a `X`, no modelo de operação aritmética de ponto-flutuante associado ao tipo de `X`. Se $\texttt{X} \neq 0$, o resultado é b^{e-p}, onde b é a base do modelo, e é o expoente de `X` e p é o número máximo de dígitos no modelo; se $\texttt{X} = 0$, então o resultado é `TINY(X)`;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos: Nenhum.

8.4.15 TINY

Sintaxe: `TINY(X)`

Sinopse: Retorna o menor número representável no modelo de operação aritmética de ponto-flutuante associado ao tipo de `X`, i.e. $b^{e_{\min}-1}$, onde b é a base do modelo e e_{min} é o menor expoente no modelo;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `REAL`, `DOUBLE PRECISION`; pode ser um arranjo;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos: Nenhum.

Exemplo 8.1 *Utilização dos intrínsecos referentes a modelos de operação aritmética*

```
PROGRAM NUMERICO

    INTEGER, PARAMETER :: NO_DECIMAL_PLACES = 15
    INTEGER, PARAMETER :: MAXIMUM_EXPONENT = 307
    INTEGER, PARAMETER :: FLOAT = SELECTED_REAL_KIND(NO_DECIMAL_PLACES, &
                                                     MAXIMUM_EXPONENT)
    INTEGER, PARAMETER :: GENERO = SELECTED_INT_KIND(2)
    INTEGER, PARAMETER :: INTEIRO = 1_GENERO
    REAL, PARAMETER :: PF = 1.0_FLOAT

        WRITE(*,'(" MODELO DE INTEIROS:")')
        WRITE(*,*)'KIND=',KIND(INTEIRO)
        WRITE(*,*)'RADIX=',RADIX(INTEIRO)
        WRITE(*,*)'EXPOENTE=',RANGE(INTEIRO)
        WRITE(*,*)'HUGE=',HUGE(INTEIRO)

        WRITE(*,'(/" MODELO DE PONTO-FLUTUANTE:")')
        WRITE(*,*)'KIND=',KIND(PF)
        WRITE(*,*)'RADIX=',RADIX(PF)
        WRITE(*,*)'PRECISION=',PRECISION(PF)
        WRITE(*,*)'DIGITS=',DIGITS(PF)
        WRITE(*,*)'EPSILON=',EPSILON(PF)
        WRITE(*,*)'MINEXPONENT=',MINEXPONENT(PF)
        WRITE(*,*)'MAXEXPONENT=',MAXEXPONENT(PF)

        WRITE(*,*)'TINY=',TINY(PF)
        WRITE(*,*)'FRACTION(TINY)=',FRACTION(TINY(PF))
        WRITE(*,*)'EXPONENT(TINY)=',EXPONENT(TINY(PF))
        WRITE(*,*)'SPACING(TINY)=',SPACING(TINY(PF))
        WRITE(*,*)'NEAREST<TINY=',NEAREST(TINY(PF),-PF)
        WRITE(*,*)'NEAREST>TINY=',NEAREST(TINY(PF),PF)
        WRITE(*,*)'HUGE=',HUGE(PF)
        WRITE(*,*)'FRACTION(HUGE)=',FRACTION(HUGE(PF))
        WRITE(*,*)'EXPONENT(HUGE)=',EXPONENT(HUGE(PF))
        WRITE(*,*)'SPACING(HUGE)=',SPACING(HUGE(PF))
        WRITE(*,*)'NEAREST<HUGE=',NEAREST(HUGE(PF),-PF)

END PROGRAM NUMERICO
```

Após compilar o programa acima com o compilador MICROSOFT POWERSTATION 4.0, *e executando o código objeto em um computador* INTEL PENTIUM 4, *obteve-se a saída abaixo:*

```
MODELO DE INTEIROS:
KIND=           4
RADIX=          2
PRECISION=          6
EXPOENTE=           9
HUGE= 2147483647

MODELO DE PONTO-FLUTUANTE:
KIND=           4
RADIX=          2
PRECISION=          6
DIGITS=         24
EPSILON=1.192093E-07
MINEXPONENT=        -125
MAXEXPONENT=         128
TINY=   1.175494E-38
FRACTION(TINY)= 5.000000E-01
```

```
EXPONENT(TINY)=        -125
SPACING(TINY)=    1.401298E-45
NEAREST<TINY=     1.175494E-38
NEAREST>TINY= 1.175494E-38
HUGE=    3.402823E+38
FRACTION(HUGE)=    9.999999E-01
EXPONENT(HUGE)=         128
SPACING(HUGE)= 2.028241E+31
NEAREST<HUGE=     3.402823E+38
```

8.5 Intrínsecos de manipulação de ponteiros

8.5.1 ASSOCIATED

Sintaxe: `ASSOCIATED(P [,TARGET=Q])`

Sinopse: Retorna `.TRUE.` ou `.FALSE.` se a variável `P`, declarada com o atributo `POINTER`, aponta para alguma variável ou não, respectivamente. Se o argumento `TARGET=Q` foi especificado, então o valor retornado será `.TRUE.` se `P` aponta para `Q` (uma variável que pode ser declarada com o atributo `POINTER` ou `TARGET`);

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `P` é uma variável de qualquer tipo, declarada com o atributo `POINTER`; `Q` é uma variável de qualquer tipo, declarada com o atributo `POINTER` ou `TARGET`;

Tipo do valor retornado: `LOGICAL`;

Nomes específicos: Nenhum.

8.6 Intrínsecos de manipulação de arranjos

8.6.1 ALL

Sintaxe: `ALL(<expressão-lógica> [,DIM=<dimensão>])`

Sinopse: Verifica se `<expressão-lógica>` é satisfeita para todos os elementos de um arranjo ou ao longo de uma dimensão, se o argumento `DIM=<dimensão>` for especificado. Retorna `.TRUE.` ou `.FALSE.` se `<expressão-lógica>` for um arranjo uni-dimensional ou `DIM=<dimensão>` não for especificado, ou um arranjo lógico, uma dimensão menor do que `<expressão-lógica>`;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `<expressão-lógica>` é um arranjo do tipo `LOGICAL`; `<dimensão>` é do tipo `INTEGER`;

Tipo do valor retornado: `LOGICAL`; pode ser um arranjo;

Nomes específicos: Nenhum.

Exemplo de uso:

```
REAL, DIMENSION(3,2) :: A
    A = 0.0
    IF (ALL(A==0.0)) THEN !     | 1.0 0.0 |
           A(:,1)=1.0     ! A = | 1.0 0.0 |
    END IF                !     | 1.0 0.0 |
```

8.6.2 ANY

Sintaxe: ANY(<expressão-lógica> [,DIM=<dimensão>])

Sinopse: Verifica se <expressão-lógica> é satisfeita para algum dos elementos de um arranjo ou ao longo de uma dimensão, se o argumento DIM=<dimensão> for especificado. Retorna .TRUE. ou .FALSE. se <expressão-lógica> for um arranjo uni-dimensional ou DIM=<dimensão> não for especificado, ou um arranjo lógico, uma dimensão menor do que <expressão-lógica>;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: <expressão-lógica> é um arranjo do tipo LOGICAL; <dimensão> é do tipo INTEGER;

Tipo do valor retornado: LOGICAL; pode ser um arranjo;

Nomes específicos: Nenhum.

Exemplo de uso:

```
REAL, DIMENSION(3,2) :: A
    A = 0.0
    A(1,1) = 1.0
    IF (ANY(A==1.0)) THEN !     ! 0.0 0.0 |
           A(:,1)=0.0     ! A = | 0.0 0.0 |
    END IF                !     | 0.0 0.0 |
```

8.6.3 COUNT

Sintaxe: COUNT(<expressão-lógica> [,DIM=<dimensão>])

Sinopse: Retorna o número de elementos de um arranjo, ou ao longo de uma dimensão, se o argumento DIM=<dimensão> for especificado, que atendam a <expressão-lógica>;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: <expressão-lógica> é um arranjo do tipo LOGICAL; <dimensão> é do tipo INTEGER;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum.

Exemplo de uso:

```
REAL, DIMENSION(3,2) :: A
    A = 0.0
    IF (ALL(A==0.0)) THEN !     | 1.0 0.0 |
           A(:,1)=1.0     ! A = | 1.0 0.0 |
    END IF                !     | 1.0 0.0 |
    ZEROS = COUNT(A==0.0) ! ZEROS = 3
```

8.6.4 CSHIFT

Sintaxe: `CSHIFT(A,<deslocamento> [,DIM=<dimensão>])`

Sinopse: Executa um deslocamento circular dos elementos do arranjo `A`, de acordo com o valor de `<deslocamento>`, o qual pode ser um número inteiro ou um arranjo. Se `<deslocamento>` é negativo, significa que os elementos serão deslocados em direção à direita; se positivo, à esquerda; se zero, não ocorrerá deslocamento. Se `DIM=<dimensão>]` for especificado, então o deslocamento dar-se-á naquela dimensão;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `<deslocamento>` é um escalar ou um arranjo do tipo `INTEGER`; `<dimensão>` é do tipo `INTEGER`;

Tipo do valor retornado: um arranjo de tipo igual ao de `A`;

Nomes específicos: Nenhum.

Exemplo de uso:

```
REAL, DIMENSION(3,3) :: A, B
INTEGER :: I,J
    DO J=1,3
            DO I=1,3                          !     | 1.0 2.0 3.0 |
                    A(I,J) = (J-1)*3+I        ! A = | 4.0 5.0 6.0 |
            END DO                            !     | 7.0 8.0 9.0 |
    END DO
    B = A
    B(2:3,2:3) = CSHIFT(A(2:3,2:3),1)
    !     | 1.0 2.0 3.0 |
    ! B = | 4.0 8.0 9.0 |
    !     | 7.0 5.0 6.0 |

    B = CSHIFT(A,1,DIM=1)
    !     | 4.0 5.0 6.0 |
    ! B = | 7.0 8.0 9.0 |
    !     | 1.0 2.0 3.0 |

    B = CSHIFT(A,1,DIM=2)
    !     | 2.0 3.0 1.0 |
    ! B = | 5.0 6.0 4.0 |
    !     | 8.0 9.0 7.0 |

    B = CSHIFT(A,(/1, 0, -1/),DIM=2)
    !     | 2.0 3.0 1.0 |
    ! B = | 4.0 5.0 6.0 |
    !     | 9.0 7.0 8.0 |
```

No primeiro `CSHIFT`, apenas os elementos do bloco `A(2:3,2:3)` são deslocados de uma unidade. No segundo, é feito um deslocamento ao longo das linhas, deslocando-as uma linha para cima. Já no terceiro, é feito um deslocamento de uma unidade ao longo das colunas. Por último, são feitos deslocamentos diferentes ao longo das colunas, sendo que a segunda linha fica inalterada, já que o deslocamento correspondente é 0.

8.6.5 EOSHIFT

Sintaxe: EOSHIFT(A,<deslocamento> [,BOUNDARY=<substitutos>] [,DIM=<dimensão>])

Sinopse: Executa um deslocamento dos elementos do arranjo A, de acordo com o valor de <deslocamento>, o qual pode ser um número inteiro ou um arranjo. Se <deslocamento> é negativo, significa que os elementos serão deslocados em direção à direita; se positivo, à esquerda; se zero, não ocorrerá deslocamento. Se BOUNDARY=<substitutos> é especificado, então os elementos presentes no arranjo <substitutos> (de tipo igual ao do arranjo A serão utilizados para ocupar os elementos tornados vagos com o deslocamento; Se DIM=<dimensão>] for especificado, então o deslocamento dar-se-á naquela dimensão;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: <deslocamento> é um escalar ou um arranjo do tipo INTEGER; <substitutos> é um arranjo de tipo igual ao de A; <dimensão> é do tipo INTEGER;

Tipo do valor retornado: um arranjo de tipo igual ao de A;

Nomes específicos: Nenhum.

Exemplo de uso:

```
REAL, DIMENSION(3,3) :: A, B
INTEGER :: I,J
    DO J=1,3
           DO I=1,3                                !     | 1.0 2.0 3.0 |
                  A(I,J) = (J-1)*3+I               ! A = | 4.0 5.0 6.0 |
           END DO                                  !     | 7.0 8.0 9.0 |
    END DO
    B = A
    B(2:3,2:3) = EOSHIFT(A(2:3,2:3),1)
    !     | 1.0  2.0  3.0 |
    ! B = | 4.0  8.0  9.0 |
    !     | 7.0  0.0  0.0 |

    B = EOSHIFT(A,1,DIM=1)
    !     | 4.0  5.0  6.0 |
    ! B = | 7.0  8.0  9.0 |
    !     | 0.0  0.0  0.0 |

    B = EOSHIFT(A,1,BOUNDARY=(/ -1.0, -2.0 /),DIM=2)
    !     | 2.0  3.0 -1.0 |
    ! B = | 5.0  6.0 -2.0 |
    !     | 8.0  9.0  0.0 |

    B = EOSHIFT(A,(/1, 0, -1/),BOUNDARY=(/ -1.0, -2.0, -3.0 /),DIM=2)
    !     | 2.0  3.0 -1.0 |
    ! B = | 4.0  5.0  6.0 |
    !     |-3.0  7.0  8.0 |
```

No primeiro EOSHIFT, apenas os elementos do bloco A(2:3,2:3) são deslocados de uma unidade e, com isso, são inseridos zeros na segunda linha do bloco. No segundo, é feito um deslocamento ao longo das linhas, deslocando-as uma linha para cima. Já no terceiro, é feito um deslocamento de uma unidade ao longo das colunas; como o argumento BOUNDARY=(/ -1.0, -2.0 /) foi especificado, foram inseridos os valores -1.0 e -2.0, na ordem em que foram listados em BOUNDARY=. Por último, são feitos deslocamentos diferentes ao longo das colunas, sendo que a segunda linha fica inalterada, já que o deslocamento correspondente é 0; observe que os elementos listados em BOUNDARY= são tomados de acordo com os deslocamentos especificados.

8.6.6 LBOUND

Sintaxe: LBOUND(A [,DIM=<dimensão>])

Sinopse: Retorna um arranjo do tipo INTEGER contendo os índices inferiores de cada dimensão do arranjo A ou ao longo de uma dimensão, se o argumento <dimensão> for especificado;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER; <dimensão> é do tipo INTEGER;

Tipo do valor retornado: arranjo do tipo INTEGER, com número de elementos igual ao número de dimensões do arranjo A;

Nomes específicos: Nenhum;

Exemplo de uso:

```
REAL, DIMENSION(-1:1,3,4:10) :: A
INTEGER, DIMENSION(3) :: IND_INF,IND_SUP
    IND_INF = LBOUND(A) ! IND_INF = | -1 1 4 |; note que a
                        ! segunda dimensão foi declarada
                        ! apenas com o tamanho, de forma que
                        ! o seu índice inferior é 1
```

8.6.7 MAXVAL

Sintaxe: MAXVAL(A [,DIM=<dimensão>] [,MASK=<expressão-lógica>])

Sinopse: Retorna o maior dos elementos de um arranjo A, ou ao longo de uma dimensão, se o argumento DIM=<dimensão> for especificado, que, opcionalmente, atendam a MASK=<expressão-lógica>;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER, REAL, DOUBLE PRECISION; <expressão-lógica> é do tipo LOGICAL; <dimensão> é do tipo INTEGER;

Tipo do valor retornado: LOGICAL;

Nomes específicos: Nenhum;

Exemplo de uso:

```
REAL, DIMENSION(3,2) :: A
REAL, DIMENSION(2) :: MAXIMO_COLUNA
REAL :: MAXIMO
INTEGER :: I,J
    DO J=1,2
            DO I=1,3                          !     | 1.0 2.0 |
                    A(I,J) = (I-1)*2+J        ! A = | 3.0 4.0 |
            END DO                            !     | 5.0 6.0 |
    END DO
    DO J=1,2
            MAXIMO_COLUNA(J) = MAXVAL(A(:,J), MASK = A(:,J)>=3.0)
    END DO
    ! MAXIMO_COLUNA = | 5.0 6.0 |
    MAXIMO = MAXVAL(A)  !   MAXIMO = 6.0
```

8.6.8 MAXLOC

Sintaxe: MAXLOC(A [,DIM=<dimensão>] [,MASK=<expressão-lógica>])

Sinopse: Retorna o(s) índice(s) do maior dos elementos de um arranjo A, que, opcionalmente, atendam a MASK=<expressão-lógica>;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER, REAL, DOUBLE PRECISION;
<expressão-lógica> é do tipo LOGICAL;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum;

Exemplo de uso:

```
REAL, DIMENSION(3,2) :: A
INTEGER, DIMENSION(2) :: IND_MAX
    DO J=1,2
            DO I=1,3                              !     | 1.0 2.0 |
                    A(I,J) = (I-1)*2+J            ! A = | 3.0 4.0 |
            END DO                                !     | 5.0 6.0 |
    END DO
    IND_MAX = MAXLOC(A) ! IND_MAX = | 3 2 |
```

8.6.9 MERGE

Sintaxe: MERGE(A,B,<expressão-lógica>)

Sinopse: Retorna um arranjo com elementos tomados dos arranjos A e B, de acordo com o valor armazenado no arranjo <expressão-lógica>. Os arranjos A, B e <expressão-lógica> devem ter a mesma dimensão e tamanho. Se um elemento de <expressão-lógica> é .TRUE., então o elemento correspondente de A será copiado para o elemento correspondente do resultado, e de B caso o elemento correspondente de <expressão-lógica> seja .FALSE.;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: A pode ser do tipo INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL ou CHARACTER; <forma> é um arranjo uni-dimensional do tipo INTEGER; <preenchimento> é um arranjo uni-dimensional de tipo idêntico ao de A; <ordem> é um arranjo do tipo INTEGER com número de elementos igual ao tamanho de <forma>;

Tipo do valor retornado: Igual ao tipo de A;

Nomes específicos: Nenhum;

Exemplo de uso:

```
REAL, DIMENSION(3,2) :: A, B = 0.0, C
LOGICAL, DIMENSION(3,2) :: MASCARA
INTEGER :: I,J
    DO J=1,2
            DO I=1,3                              !     | 1.0 2.0 |
                    A(I,J) = (I-1)*2+J            ! A = | 3.0 4.0 |
            END DO                                !     | 5.0 6.0 |
```

```
END DO
                          !           | .FALSE. .FALSE. |
MASCARA = A>3.0           ! MASCARA = | .FALSE.  .TRUE. |
                          !           |  .TRUE.  .TRUE. |

                               !     | 0.0 0.0 |
C = MERGE(A,B,MASCARA)         ! C = | 0.0 4.0 |
                               !     | 5.0 6.0 |
```

8.6.10 MINVAL

Sintaxe: `MINVAL(A [,DIM=<dimensão>] [,MASK=<expressão-lógica>])`

Sinopse: Retorna o menor dos elementos de um arranjo `A`, ou ao longo de uma dimensão, se o argumento `DIM=<dimensão>` for especificado, que, opcionalmente, atendam a `MASK=<expressão-lógica>`;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`; `<expressão-lógica>` é do tipo `LOGICAL`; `<dimensão>` é do tipo `INTEGER`;

Tipo do valor retornado: `LOGICAL`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
REAL, DIMENSION(3,2) :: A
REAL, DIMENSION(2) :: MINIMO_COLUNA
REAL :: MINIMO
INTEGER :: I,J
    DO J=1,2
            DO I=1,3                          !     | 1.0 2.0 |
                    A(I,J) = (I-1)*2+J        ! A = | 3.0 4.0 |
            END DO                            !     | 5.0 6.0 |
    END DO
    DO J=1,2
            MINIMO_COLUNA(J) = MINVAL(A(:,J), MASK = A(:,J)<=3.0)
    END DO
    ! MINIMO_COLUNA = | 1.0 2.0 |
    MINIMO = MINVAL(A)  !   MINIMO = 1.0
```

8.6.11 MINLOC

Sintaxe: `MINLOC(A [,DIM=<dimensão>] [,MASK=<expressão-lógica>])`

Sinopse: Retorna o(s) índice(s) do menor dos elementos de um arranjo `A`, que, opcionalmente, atendam a `MASK=<expressão-lógica>`;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`;
`<expressão-lógica>` é do tipo `LOGICAL`;

Tipo do valor retornado: `INTEGER`;

Nomes específicos: Nenhum;

Exemplo de uso: MINLOC(A,A>0.0)

```
REAL, DIMENSION(3,2) :: A
INTEGER, DIMENSION(2) :: IND_MIN
   DO J=1,2
          DO I=1,3                                  !     | 1.0 2.0 |
                 A(I,J) = (I-1)*2+J                 ! A = | 3.0 4.0 |
          END DO                                    !     | 5.0 6.0 |
   END DO
   IND_MIN = MINLOC(A, MASK = A>=3.0) ! IND_MIN = | 2 1 |
```

8.6.12 PACK

Sintaxe: PACK(A,<expressão-lógica> [,VECTOR=<vetor>])

Sinopse: Retorna um arranjo uni-dimensional com elementos tomados do arranjo A, de acordo com o valor armazenado no arranjo <expressão-lógica>. Os arranjos A e <expressão-<expressãológica> devem ter a mesma dimensão e tamanho.

Se um elemento de <expressão-lógica> é .TRUE., então o elemento correspondente de A será copiado para o resultado. Se um número menor de elementos de A satisfizer a <expressão-lógica>, então serão mantidos inalterados demais valores do arranjo resultado.

Se VECTOR=<vetor> for especificado, então serão copiados tantos elementos do arranjo uni-dimensional <vetor> quantos for possível, e o valor zero, .FALSE. ou '' (dependendo do tipo de A) para completar os elementos do arranjo resultado.

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: A pode ser do tipo INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL ou CHARACTER; <forma> é um arranjo uni-dimensional do tipo INTEGER; <preenchimento> é um arranjo uni-dimensional de tipo idêntico ao de A; <ordem> é um arranjo do tipo INTEGER com número de elementos igual ao tamanho de <forma>;

Tipo do valor retornado: Igual ao tipo de A;

Nomes específicos: Nenhum;

Exemplo de uso:

```
REAL, DIMENSION(3,3) :: A
REAL, DIMENSION(3) :: COLUNA
REAL, DIMENSION(6) :: VET = 0.0
INTEGER :: I,J
    DO J=1,3
           DO I=1,3                                 !     | 1.0 4.0 7.0 |
                  A(I,J) = (J-1)*3+I                ! A = | 2.0 5.0 8.0 |
           END DO                                   !     | 3.0 6.0 9.0 |
    END DO
    COLUNA = A(:,2) ; A(:,2) = A(:,1) ; A(:,1) = COLUNA ! troca a coluna 1
                                                        ! com a 2

    !     | 4.0 1.0 7.0 |
    ! A = | 5.0 2.0 8.0 |
    !     | 6.0 3.0 9.0 |

    VET = PACK(A, A>3.0) ! VET = | 4.0 5.0 6.0 7.0 8.0 9.0 |
```

A `<expressão-logica> A>3.0` acaba por selecionar os elementos das colunas 1 e 3 de A; dessa forma, tomando esses elementos coluna a coluna, de acordo com a forma de armazenamento de arranjos em FORTRAN 90, *obtemos o arranjo uni-dimensional `VET = (/ 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 /)`.*

8.6.13 RESHAPE

Sintaxe: `RESHAPE(A,<forma> [,PAD=<preenchimento>] [,ORDER=<ordem>])`

Sinopse: Retorna um arranjo cujo *"shape"* é especificado pelo arranjo `<forma>`, armazenados de acordo com a ordem das dimensões listada no arranjo `<ordem>`, opcionalmente preenchido com os elementos contidos no arranjo uni-dimensional `<preenchimento>`;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `A` pode ser do tipo `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `LOGICAL` ou `CHARACTER`; `<forma>` é um arranjo uni-dimensional do tipo `INTEGER`; `<preenchimento>` é um arranjo uni-dimensional de tipo idêntico ao de `A`; `<ordem>` é um arranjo do tipo `INTEGER` com número de elementos igual ao tamanho de `<forma>`;

Tipo do valor retornado: Igual ao tipo de `A`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
REAL, DIMENSION(3,2) :: A
REAL, DIMENSION(2,4) :: B
INTEGER :: I,J
    DO J=1,2
           DO I=1,3                          !     | 1.0 2.0 |
                  A(I,J) = (I-1)*2+J         ! A = | 3.0 4.0 |
           END DO                            !     | 5.0 6.0 |
    END DO
    B = RESHAPE(A, (/ 2, 4 /), &
                PAD = (/ -1.0, -2.0, -3.0, -4.0 /), &
                ORDER = (/ 2, 1 /) )

    ! B = |  1.0  3.0  5.0  2.0 |
    !     |  4.0  6.0 -1.0 -2.0 |
```

Observe que, em FORTRAN 90, *uma matriz é armazenada por colunas. O comando `RESHAPE`, acima, faz com que os elementos de `A` sejam transferidos para uma matriz `B` com 2 linhas e 4 colunas, armazenados por linha (já que `ORDER=(/ 2, 1 /)`, o que significa que a primeira dimensão de `A` deve ser armazenada na segunda dimensão de `B`). Assim, a primeira coluna de `A`, contendo os elementos `1.0, 3.0, 5.0` será armazenada nos três primeiros elementos da primeira linha de `B`; a segunda coluna de `A`, contendo os elementos `2.0, 4.0, 6.0`, será armazenada a partir do quarto elemento da primeira linha de `B`, até o segundo elemento da segunda linha de `B`. Como, agora, não há mais elementos em `A` a serem transferidos, são tomados os dois primeiros elementos listados em `PAD = (/ -1.0, -2.0, -3.0, -4.0 /)`, os quais são armazenados nos elementos `B(2,3)` e `B(2,4)`.*

8.6.14 SHAPE

Sintaxe: `SHAPE(A)`

Sinopse: Retorna um arranjo do tipo `INTEGER` com o mesmo número de dimensões de `A`;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER;
<dimensão> é do tipo INTEGER;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum;

Exemplo de uso:

```
REAL, DIMENSION(3,2) :: A
REAL, DIMENSION(2,4) :: B
INTEGER, DIMENSION(2) :: FORMA
    FORMA = SHAPE(A) ! FORMA = | 3 2 |
    FORMA = SHAPE(B) ! FORMA = | 2 4 |
```

8.6.15 SIZE

Sintaxe: SIZE(A [,DIM=<dimensão>])

Sinopse: Retorna um número inteiro contendo o tamanho de cada dimensão do arranjo A ou ao longo de uma dimensão, se o argumento DIM=<dimensão> for especificado;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER;
<dimensão> é do tipo INTEGER;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum;

Exemplo de uso:

```
REAL, DIMENSION(3,2) :: A
REAL, DIMENSION(2,4) :: B
INTEGER :: TAMANHO
    TAMANHO = SIZE(A)        ! TAMANHO = 6
    TAMANHO = SIZE(B, DIM=1) ! TAMANHO = 2
```

8.6.16 SPREAD

Sintaxe: SPREAD(A,<dimensão>,<cópias>)

Sinopse: Retorna um arranjo do tipo INTEGER contendo a forma de A;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER;
<dimensão> é do tipo INTEGER;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum;

Exemplo de uso:

```
REAL, DIMENSION(3,2) :: A = 0.0
    A = SPREAD( (/ 1.0, 2.0 /), 1, 2)   !     | 1.0 2.0 |
                                        ! A = | 1.0 2.0 |
                                        !     | 1.0 2.0 |
```

8.6.17 TRANSPOSE

Sintaxe: `TRANSPOSE(A)`

Sinopse: Retorna a matriz transposta de `A`; se a `A` é da forma `DIMENSION(M,N)`, então o resultado é da forma `DIMENSION(N,M)`;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `LOGICAL`, `CHARACTER`;

Tipo do valor retornado: Igual ao tipo do argumento;

Nomes específicos: Nenhum;

Exemplo de uso: `TRANSPOSE(A)`

8.6.18 UBOUND

Sintaxe: `UBOUND(A [,DIM=<dimensão>])`

Sinopse: Retorna um arranjo do tipo `INTEGER` contendo os índices superiores de cada dimensão do arranjo `A` ou ao longo de uma dimensão, se o argumento `DIM=<dimensão>` for especificado;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `LOGICAL`, `CHARACTER`; `<dimensão>` é do tipo `INTEGER`;

Tipo do valor retornado: arranjo do tipo `INTEGER`, com número de elementos igual ao número de dimensões do arranjo `A`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
REAL, DIMENSION(-1:1,3,4:10) :: A
INTEGER, DIMENSION(3) :: IND_SUP
    IND_SUP = UBOUND(A) ! IND_INF = | 1 3 10 |; note que a
                        ! segunda dimensão foi declarada
                        ! apenas com o tamanho, de forma que
                        ! o seu índice superior é 3
```

8.7 Intrínsecos de manipulação de *"bits"*

8.7.1 BIT_SIZE

Sintaxe: `BIT_SIZE(I)`

Sinopse: Retorna um escalar do tipo `INTEGER` contendo o número de *"bits"* associados ao `KIND` do argumento `I`, de tipo `INTEGER`;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`;

Tipo do valor retornado: `INTEGER`, de mesmo `KIND` que o argumento;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER(1) :: A, NBITS_A
INTEGER(2) :: B, NBITS_B
INTEGER(4) :: C, NBITS_C
INTEGER(8) :: D, NBITS_D
    NBITS_A = BIT_SIZE(A) ! NBITS_A = 8
    NBITS_B = BIT_SIZE(B) ! NBITS_B = 16
    NBITS_C = BIT_SIZE(C) ! NBITS_C = 32
    NBITS_D = BIT_SIZE(D) ! NBITS_D = 64
```

8.7.2 BTEST

Sintaxe: `BTEST(I,P)`

Sinopse: Retorna um escalar ou arranjo do tipo `LOGICAL` indicando se o(s) *"bit(s)"* na posição indicada no escalar ou arranjo `P`, de tipo `INTEGER`, está(ão) ativo(s) (i.e., iguais a 1). A posição de um *"bit"* é um número entre 0 e `BIT_SIZE(I)-1`; a ordem de enumeração dos *"bits"* é dependente da implementação;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`;

Tipo do valor retornado: `LOGICAL`; pode ser um arranjo;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER(1) :: A = -88 ! Em binário, 10101000
LOGICAL, DIMENSION(2) :: BIT_LIGADO
    BIT_LIGADO = BTEST(A,(/4,5/)) ! BIT_LIGADO = | .FALSE. .TRUE. |
```

8.7.3 IBCLR

Sintaxe: `IBCLR(I,P)`

Sinopse: Coloca o valor 0 no(s) *"bit(s)"* do argumento `I` de tipo `INTEGER`, indicados pela(s) posição(ões) indicada(s) pelo escalar ou arranjo `P`, de tipo `INTEGER`. A posição de um *"bit"* é um número entre 0 e `BIT_SIZE(I)-1`; o *"bit"* menos significativo (mais à direita) ocupa a posição 0;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`;

Tipo do valor retornado: `INTEGER` com o mesmo `KIND` do argumento `I`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER(1) :: A = -88        ! Em binário, 10101000
INTEGER(1) :: B
    B = IBCLR(A,5) ! B = -120 (em binário, 10001000)
```

8.7.4 IBITS

Sintaxe: `IBITS(I,P,L)`

Sinopse: Extrai o(s) *"bit(s)"* do argumento `I`, de tipo `INTEGER`, contados a partir da(s) posição(ões) indicada(s) pelo escalar ou arranjo `P`, de tipo `INTEGER`; o argumento `L` é um escalar ou arranjo de tipo `INTEGER`, que indica quantos *"bits"* devem ser extraídos. A posição de um *"bit"* é um número entre 1 e `BIT_SIZE(I)`; o *"bit"* mais significativo (mais à esquerda) ocupa a posição 1;

Restrições: A soma `P+L` não deve exceder `BIT_SIZE(I)`;

Tipo de intrínseco: Função;

Tipo de argumento: `INTEGER`;

Tipo do valor retornado: `INTEGER` com o mesmo `KIND` do argumento `I`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER(1) :: A = -88        ! Em binário, 10101000
INTEGER(1) :: B
    B = IBITS(A,2,5) ! B = 10 (em binário, 00001010)
```

8.7.5 IBSET

Sintaxe: `IBSET(I,P)`

Sinopse: Coloca o valor 1 no(s) *"bit(s)"* do argumento `I` de tipo `INTEGER`, indicados pela(s) posição(ões) indicada(s) pelo escalar ou arranjo `P`, de tipo `INTEGER`. A posição de um *"bit"* é um número entre 0 e `BIT_SIZE(I)-1`; o *"bit"* menos significativo (mais à direita) ocupa a posição 0;

Restrições: Nenhuma

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER;

Tipo do valor retornado: INTEGER com o mesmo KIND do argumento I;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER(1) :: A = -88        ! Em binário, 10101000
INTEGER(1) :: B
    B = IBSET(A,2) ! B = -84 (em binário, 10101100)
```

8.7.6 ISHFT

Sintaxe: ISHFT(I,D)

Sinopse: Faz um deslocamento dos *"bit(s)"* do argumento I, de tipo INTEGER, tantos quantos forem indicado(s) pelo escalar ou arranjo D, de tipo INTEGER. Valores negativos em D correspondem a deslocamentos à direita (i.e., divisões inteiras por 2); positivos correspondem a deslocamentos à esquerda (i.e. multiplicações por 2);

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER;

Tipo do valor retornado: INTEGER com o mesmo KIND do argumento I;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER(2), DIMENSION(2) :: A = (/ 5, -4, 3 /)
    ! Em binário, (/  101, 1111111111111100, 11 /)

    A = ISHFT(A,(/1,2,-1/)) ! A = (/ 10, -16, 1 /)
    ! Em binário, (/ 1010, 1111111111110000,  1 /)
```

8.7.7 ISHFTC

Sintaxe: ISHFTC(I,D [,SIZE=<tamanho>])

Sinopse: Faz um deslocamento circular dos *"bit(s)"* do argumento I, de tipo INTEGER, tantos quantos forem indicado(s) pelo escalar ou arranjo D, de tipo INTEGER. Valores negativos em D correspondem a deslocamentos à direita (i.e., divisões inteiras por 2); positivos correspondem a deslocamentos à esquerda (i.e. multiplicações por 2). O argumento SIZE=<tamanho>, de tipo INTEGER, se especificado, indica quantos *"bits"* consecutivos, contados a partir do *bit* menos significativo (mais à direita), sofrerão a rotação.

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: INTEGER;

Tipo do valor retornado: INTEGER com o mesmo KIND do argumento I;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER(2) :: A = -20           ! Em binário, 1111111111101100
    A = ISHFTC(A,1)  ! A = -39   (em binário, 1111111111011001)

INTEGER(2) :: B = 125           ! Em binário, 0000000001111101
    B = ISHFTC(B,-2) ! B = 16415 (em binário, 0100000000011111)

INTEGER(2) :: C = 24567         ! Em binário, 0101111111110111
    C = ISHFTC(C,-2) ! C = 24573 (em binário, 0101111111111101)
```

8.7.8 MVBITS

Sintaxe: `MVBITS(I,P,L,J,Q)`

Sinopse: Copia um conjunto de `L` *"bits"* de `I`, a partir da posição `P`, e armazena-os a partir da posição `Q` do argumento `J`. Todos os argumentos são do tipo `INTEGER` e devem ter o mesmo `KIND`; se forem arranjos, devem ter o mesmo *"shape"*. O *"bit"* menos significativo (mais à direita) está na posição 0;

Restrições: Nenhuma;

Tipo de intrínseco: Subrotina;

Tipo de argumento: `INTEGER`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER(2) :: A = -20               ! Em binário, 1111111111101100
INTEGER(2) :: B = 125               ! Em binário, 0000000001111101
    CALL MVBITS(A,2,4,B,8) ! B = 2941 (em binário, 0000101101111101)
```

8.8 Intrínsecos de manipulação de *"strings"*

8.8.1 ADJUSTL

Sintaxe: `ADJUSTL(C)`

Sinopse: Remove os primeiros caracteres em branco da *"string"* `C`, e insere um número equivalente de espaços em branco ao final;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `CHARACTER*(*)`, pode ser um arranjo;

Tipo do valor retornado: `CHARACTER*(*)`, pode ser um arranjo;

Nomes específicos: Nenhum;

Exemplo de uso:

```
CHARACTER*(11) :: A
    A = ADJUSTL('  AU-10-BR2') ! A = 'AU-10-BR2  '
```

8.8.2 ADJUSTR

Sintaxe: ADJUSTR(C)

Sinopse: Remove os últimos caracteres em branco da *"string"* C, e insere um número equivalente de espaços em branco no início;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: CHARACTER*(*), pode ser um arranjo;

Tipo do valor retornado: CHARACTER*(*), pode ser um arranjo;

Nomes específicos: Nenhum;

Exemplo de uso:

```
CHARACTER*(11) :: A
    A = ADJUSTR('  AU-10-BR2') ! A = '  AU-10-BR2'
```

8.8.3 IACHAR

Sintaxe: IACHAR(C)

Sinopse: Retorna um escalar ou arranjo, de tipo INTEGER, contendo o(s) código(s) numérico(s), na tabela de caracteres ASCII, correspondente(s) ao(s) caracter(es) contido(s) no escalar ou arranjo C, de tipo CHARACTER;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: CHARACTER; pode ser um arranjo;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER :: I
    I = IACHAR('W') ! I = 87
```

8.8.4 ICHAR

Sintaxe: ICHAR(C)

Sinopse: Retorna um escalar ou arranjo, de tipo INTEGER, contendo o(s) código(s) numérico(s), na tabela de caracteres específica ao processador, correspondente(s) ao(s) caracter(es) contido(s) no escalar ou arranjo C, de tipo CHARACTER. As entradas de 0 a 127 da tabela específica do processador correspondem às 128 entradas da tabela ASCII;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: CHARACTER; pode ser um arranjo;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER :: I
    I = ICHAR('W') ! I = 87
```

8.8.5 INDEX

Sintaxe: `INDEX(C,D [,BACK=<expressão-lógica>])`

Sinopse: Retorna um escalar, de tipo `INTEGER`, correspondente à posição onde a *"string"* de caracteres `D` encontra-se na *"string"* de caracteres `C`. O caracter mais à esquerda encontra-se na posição 1. O tamanho de `C` pode ser obtido com o intrínseco `LEN(C)`.

Se `BACK=<expressão-lógica>` for especificado, então a busca dar-se-á da direita para a esquerda, se `<expressão-lógica>=.TRUE.`; e da esquerda para a direita, se `<expressão-lógica>=.FALSE.` (*"default"*);

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `CHARACTER*(*)`, pode ser um arranjo;

Tipo do valor retornado: `INTEGER`, pode ser um arranjo;

Nomes específicos: Nenhum;

Exemplo de uso:

```
CHARACTER*(10) :: CODIGO_PRODUTO = 'ABX-567-BR1'
INTEGER :: POS
    POS = INDEX(CODIGO_PRODUTO,'BR') ! POS = 9
```

8.8.6 LEN

Sintaxe: `LEN(C)`

Sinopse: Retorna um escalar, de tipo `INTEGER`, contendo o número de caracteres na *"string"* `C`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `CHARACTER`, pode ser um arranjo;

Tipo do valor retornado: `INTEGER`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
CHARACTER*(10) :: CODIGO_PRODUTO = '  ABX      '
                                  !1234567890
INTEGER :: TAMANHO
    TAMANHO = LEN(CODIGO_PRODUTO) ! TAMANHO = 10
```

8.8.7 LEN_TRIM

Sintaxe: LEN_TRIM(C)

Sinopse: Retorna um escalar, de tipo INTEGER, contendo o número de caracteres *"string"* C. Não são considerados os últimos espaços em branco em C;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: CHARACTER*(*), pode ser um arranjo;

Tipo do valor retornado: INTEGER;

Nomes específicos: Nenhum;

Exemplo de uso:

```
CHARACTER*(10) :: CODIGO_PRODUTO = '  ABX      '
                                   !1234567890
INTEGER :: TAMANHO
    TAMANHO = LEN_TRIM(CODIGO_PRODUTO) ! TAMANHO = 5
```

8.8.8 LGE

Sintaxe: LGE(A,B)

Sinopse: Retorna um escalar, de tipo LOGICAL, contendo o valor .TRUE se a *"string"* A tem caracteres cujos códigos ASCII são iguais ou superiores aos caracteres da *"string"* B. A comparação é efetuada da esquerda para a direita; se as *"strings"* não tem o mesmo tamanho, então a de menor tamanho é completada com caracteres em branco;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: CHARACTER*(*), podem ser arranjos (de mesmo *"shape"*;

Tipo do valor retornado: LOGICAL;

Nomes específicos: Nenhum;

Exemplo de uso:

```
LOGICAL :: RESULTADO
    RESULTADO = LGE('ABC','AAC')     ! RESULTADO = .TRUE.
    RESULTADO = LGE('ABBCED','ABC') ! RESULTADO = .FALSE.
```

8.8.9 LGT

Sintaxe: LGT(A,B)

Sinopse: Retorna um escalar, de tipo LOGICAL, contendo o valor .TRUE se a *"string"* A tem caracteres cujos códigos ASCII são superiores aos caracteres da *"string"* B. A comparação é efetuada da esquerda para a direita; se as *"strings"* não tem o mesmo tamanho, então a de menor tamanho é completada com caracteres em branco;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `CHARACTER*(*)`, podem ser arranjos (de mesmo *"shape"*);

Tipo do valor retornado: `LOGICAL`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
LOGICAL :: RESULTADO
    RESULTADO = LGT('ABC','AAC')     ! RESULTADO = .TRUE.
    RESULTADO = LGT('ABBCED','ABC') ! RESULTADO = .FALSE.
```

8.8.10 LLE

Sintaxe: `LLE(A,B)`

Sinopse: Retorna um escalar, de tipo `LOGICAL`, contendo o valor `.TRUE` se a *"string"* `A` tem caracteres cujos códigos ASCII são iguais ou menores aos caracteres da *"string"* `B`. A comparação é efetuada da esquerda para a direita; se as *"strings"* não tem o mesmo tamanho, então a de menor tamanho é completada com caracteres em branco;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `CHARACTER*(*)`, podem ser arranjos (de mesmo *"shape"*);

Tipo do valor retornado: `LOGICAL`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
LOGICAL :: RESULTADO
    RESULTADO = LLE('ABC','AAC')     ! RESULTADO = .FALSE.
    RESULTADO = LLE('ABBCED','ABC') ! RESULTADO = .TRUE.
```

8.8.11 LLT

Sintaxe: `LLT(A,B)`

Sinopse: Retorna um escalar, de tipo `LOGICAL`, contendo o valor `.TRUE` se a *"string"* `A` tem caracteres cujos códigos ASCII são menores aos caracteres da *"string"* `B`. A comparação é efetuada da esquerda para a direita; se as *"strings"* não tem o mesmo tamanho, então a de menor tamanho é completada com caracteres em branco;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `CHARACTER*(*)`, podem ser arranjos (de mesmo *"shape"*);

Tipo do valor retornado: `LOGICAL`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
LOGICAL :: RESULTADO
    RESULTADO = LLE('ABC','AAC')     ! RESULTADO = .FALSE.
    RESULTADO = LLE('ABBCED','ABC') ! RESULTADO = .TRUE.
```

8.8.12 REPEAT

Sintaxe: `REPEAT(A,N)`

Sinopse: Retorna uma *"string"* contendo os caracteres na *"string"* `A`, repetidos `N` vezes, onde `N` é um argumento de tipo `INTEGER`;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `CHARACTER*(*)`;

Tipo do valor retornado: `CHARACTER*(*)`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
CHARACTER*(3) :: GRITO = 'AI'
CHARACTER*(9) :: GRITO_MAIOR
    GRITO_MAIOR = REPEAT(GRITO,3) ! GRITO_MAIOR = 'AIAIAI'
```

8.8.13 SCAN

Sintaxe: `SCAN(C,D [,BACK=<expressão-lógica>])`

Sinopse: Retorna um escalar, de tipo `INTEGER`, correspondente à posição onde algum dos caracteres da *"string"* `D` encontra-se na *"string"* `C`. O caracter mais à esquerda encontra-se na posição 1. O tamanho de `C` pode ser obtido com o intrínseco `LEN(C)`.

Se `BACK=<expressão-lógica>` for especificado, então a busca dar-se-á da direita para a esquerda, se `<expressão-lógica>=.TRUE.`; e da esquerda para a direita, se `<expressão-lógica>=.FALSE.` (*"default"*);

Restrições: Se `C` e `D` forem arranjos, então eles devem ser de mesmo tamanho;

Tipo de intrínseco: Função;

Tipo de argumento: `CHARACTER*(*)`, pode ser um arranjo; `LOGICAL`

Tipo do valor retornado: `INTEGER`, pode ser um arranjo;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER, DIMENSION(2) :: I
    I = SCAN(('AXL-589-BR1', 'AU-10-US2  '),('BR','US'))
            ! I = (/ 9, 2 /)
    I = SCAN(('AXL-589-BR1', 'AU-10-US2  '),('BR','US'),BACK=.TRUE.)
            ! I = (/ 10, 8 /)
```

8.8.14 TRIM

Sintaxe: `TRIM(C)`

Sinopse: Retorna uma *"string"* com os caracteres da *"string"* `C`, menos os últimos espaços em branco;

Restrições: Nenhuma;

Tipo de intrínseco: Função;

Tipo de argumento: `CHARACTER*(*)`;

Tipo do valor retornado: `CHARACTER*(*)`;

Nomes específicos: Nenhum;

Exemplo de uso:

```
CHARACTER*(11) :: A
    A = TRIM('AU-10-US2  ') ! A = 'AU-10-US2'
```

8.8.15 VERIFY

Sintaxe: `VERIFY(C,D [,BACK=<expressão-lógica>])`

Sinopse: Retorna um escalar, de tipo `INTEGER`, correspondente à primeira posição onde exista algum dos caracteres da *"string"* `C` que não se encontra na *"string"* `D`. O caracter mais à esquerda encontra-se na posição 1. O tamanho de `C` pode ser obtido com o intrínseco `LEN(C)`.

Se `BACK=<expressão-lógica>` for especificado, e `<expressão-lógica>=.TRUE.`, então a busca será feita da direita para a esquerda.

Se `BACK=<expressão-lógica>` for especificado, e `<expressão-lógica>=.FALSE.`, então a busca será feita da esquerda para a direita (*"default"*);

Restrições: Se `C` e `D` forem arranjos, então eles devem ser de mesmo tamanho;

Tipo de intrínseco: Função;

Tipo de argumento: `CHARACTER*(*)`, pode ser um arranjo; `LOGICAL`

Tipo do valor retornado: `INTEGER`, pode ser um arranjo;

Nomes específicos: Nenhum;

Exemplo de uso:

```
INTEGER, DIMENSION(2) :: I
    I = VERIFY((/'AXL-589-BR1', 'AU-10-US2  '/),(/'BR','US'/))
        ! I = (/ 1, 1 /)
    I = VERIFY((/'AXL-589-BR1', 'AU-10-US2  '/),(/'BR','US'/),BACK=.TRUE.)
        ! I = (/ 11, 11 /)
```

8.9 Intrínsecos de tempo e hora

8.9.1 DATE_AND_TIME

Sintaxe: `CALL DATE_AND_TIME([DATE=<data>] [,TIME=<horário>] [,ZONE=<zona>] [,VALUES=<valores>])`

Sinopse: Os argumentos são todos opcionais e de saída, ao menos um deles deve ser especificado na chamada. Se `DATE=<data>` é especificado, então os oito primeiros caracteres da *"string"* `<data>` conterão o século, o ano, o mês e o dia, no formato `CCAAMMDD`. Se `TIME=<horário>` é especificado, então os dez primeiros caracteres da *"string"* `<horário>` conterão as horas, minutos, segundos e milissegundos, no formato `HHMMSS.SSS`. Se `ZONE=<zona>` é especificado, então os primeiros cinco caracteres da *"string"* `<zona>` conterão a diferença em horas e minutos correspondente ao meridiano de Greenwich, no formato `[+|-]HHMM`. Se `VALUES=<valores>`

é especificado, então os oito primeiros elementos do arranjo uni-dimensional <valores>, de tipo INTEGER, conterão:

<valores>(1) = Ano
<valores>(2) = Mês
<valores>(3) = Dia
<valores>(4) = Diferença em minutos em relação ao meridiano de Greenwich
<valores>(5) = Horas
<valores>(6) = Minutos
<valores>(7) = Segundos
<valores>(8) = Milissegundos

Se algum dos valores acima não puder ser devolvido, então o valor -HUGE(0) é retornado.

Restrições: Nenhum;

Tipo de intrínseco: Subrotina;

Tipo de argumento: <data> : CHARACTER*(8); <horário> : CHARACTER*(10); <zona> : CHARACTER*(5); <valores> : INTEGER, DIMENSION(8)

Nomes específicos: Nenhum;

Exemplo de uso:

```
CHARACTER*(8) :: DATA
CHARACTER*(10) :: HORARIO

    CALL DATE_AND_TIME(DATE=DATA, TIME=HORARIO)
```

8.9.2 SYSTEM_CLOCK

Sintaxe: CALL SYSTEM_CLOCK([COUNT=<contador>] [,COUNT_RATE=<taxa>] [,COUNT_MAX=<valor>])

Sinopse: Os argumentos são todos opcionais e de saída, ao menos um deles deve ser especificado na chamada. Se COUNT=<contador> é especificado, o valor de tipo INTEGER, armazenado no relógio do processador, será retornado, ou -HUGE(0) caso não haja relógio disponível. Se COUNT_RATE=<taxa> é especificado, será retornado em <taxa> um valor de tipo INTEGER que indica quantas vezes o relógio marca por segundo, ou 0 caso não haja relógio disponível. Se COUNT_MAX=<valor> é especificado, então o valor máximo admitido pelo relógio será retornado em <valor>, de tipo INTEGER, ou 0 caso não haja relógio disponível.

Restrições: Nenhum;

Tipo de intrínseco: Subrotina;

Tipo de argumento: <contador>, taxa, valor : INTEGER;

Nomes específicos: Nenhum;

Capítulo 9

Estudos de caso

Neste capítulo, abordaremos inicialmente a portabilidade de um programa FORTRAN 90 e como deve ser feito o controle de erros de execução. Após, são apresentados alguns exemplos típicos que demonstram a potencialidade da linguagem.

9.1 Portabilidade

A *portabilidade* de um programa consiste na capacidade de compilá-lo e/ou executá-lo em diferentes computadores, possivelmente sob diferentes sistemas operacionais e com diferentes compiladores, de tal forma que os resultados obtidos com o programa, nesses diferentes computadores, sejam iguais ou compatíveis.

Apesar de existir um padrão para a linguagem FORTRAN 90, nem todos os compiladores disponíveis aderem ao padrão. Nesse caso, ou se modifica o código fonte, para adequá-lo ao compilador (o que não é indicado mas, às vezes, é a única alternativa possível), ou se adquire um compilador de melhor qualidade.

Por outro lado, a linguagem FORTRAN 90 oferece uma série de mecanismos embutidos que melhoram, em muito, a portabilidade de programas escritos nessa linguagem. Considere, por exemplo, o problema de se localizar o maior elemento dentre os valores armazenados em um arranjo unidimensional `A` com `N` posições; o código abaixo é típico de uma implementação em FORTRAN 77:

```
MAIOR = -1.0E50
DO I=1,N
    MAIOR = MAX(MAIOR,A(I))
END DO
```

Esse código *não é portável*. Suponha que, no computador no qual ele foi testado, `-1.0E50` seja o menor valor possível de armazenamento mas, num outro computador, esse valor seja `-1.0E499`; se todos os valores em `A` forem porventura menores do que `-1.0E50`, esse código não selecionará o maior valor existente no arranjo. Em FORTRAN 90, pode-se - e *deve-se* - fazer, simplesmente:

```
MAIOR = MAXVAL(A)
```

pois o intrínseco `MAXVAL` (vide §8.6) deve, pelo padrão, necessariamente retornar o maior valor do arranjo.

Pode-se dizer, via de regra, que sempre que puder ser utilizado um intrínseco da linguagem, isso aumentará a portabilidade do programa.

9.2 Controle de erros de execução

A linguagem FORTRAN 90 não oferece um mecanismo de controle de erros de execução como, por exemplo, a linguagem Ada. Dessa forma, fica a cargo do programador especificar qual ação a ser tomada caso haja algum erro de execução, o que leva à interrupção da execução do programa.

Dentre os erros mais comuns, encontram-se a *divisão por zero*, o uso de argumentos *fora do intervalo de definição* de uma função e o *acesso a posições inválidas de arranjos*. Nos dois primeiros, o programador deve embutir um trecho de código que evite a ocorrência do erro. Por exemplo, suponha o cálculo da normalização de um vetor $v \in \mathbb{R}^n$, i.e.

$$\tilde{v} = \frac{1}{||v||_1} v, \quad ||v||_1 = \max_{i=1}^{n} |v_i|.$$

É possível que o arranjo V contenha valores nulos, de tal forma que $||v||_1 = 0$, e um programa de boa qualidade deve incluir o tratamento desse (possível) erro, como no trecho de código abaixo:

```
NORMA_1 = MAXVAL(ABS(V))
IF (NORMA_1==0.0) THEN
    V_TILDE = 0.0
ELSE
    V_TILDE = 1.0/NORMA_1 * V
END IF
```

Já o acesso a posições inválidas de arranjos é um erro que pode ser evitado se utilizarmos alguns dos recursos de declaração de arranjos. Suponha, por exemplo, que se deseja transmitir um arranjo unidimensional de tamanho N a um subprograma, como no exemplo abaixo:

```
PROGRAM TESTE
    REAL, DIMENSION(-4:4) :: A
        CALL SUB(A)
    CONTAINS
        SUBROUTINE SUB(X)
        REAL, DIMENSION(:), INTENT(INOUT) :: X
            ...
        END SUBROUTINE SUB
END PROGRAM TESTE
```

Para se acessar os elementos do argumento X, é necessário que se conheça o *tamanho* do arranjo transmitido à subrotina SUB quando da sua chamada. Note que os elementos de X são numerados como 1, 2, ..., 8, 9 e *não* como -4, -3, ..., 2, 3. Por isso, deve-se usar o intrínseco SIZE (vide §8.6), que permite obter o tamanho do arranjo passado como argumento e acessá-lo convenientemente. Além disso, é interessante que, enquanto se está desenvolvendo o programa, se permita que o compilador detecte tais erros de execução[1].

9.3 Manipulação de matrizes esparsas

Um dos problemas mais comumente enfrentados em computação científica é a necessidade de se operar com matrizes *esparsas*. Essas são matrizes com um grande número de linhas e colunas (tipicamente, mais de 10.000) e, por outro lado, apenas um pequeno número de seus elementos (em torno de 5%) são diferentes de zero. Obviamente, matrizes de certo tamanho não poderão ser armazenadas na forma usual de linhas e colunas, pois não caberão na memória disponível num dado computador; dessa forma, é necessário se utilizar *estruturas de dados* específicas para tais matrizes. Além disso, os algoritmos que manipulam essas matrizes devem ser eficientes, operando *apenas* com os elementos não-nulos da matriz.

[1] O anexo A apresenta como fazer isso em um ambiente UN*X.

Uma das estruturas mais utilizadas para se representar estruturas de dados é o chamado *"Compressed Column Storage" - CCS* [2], na qual se armazenam os elementos não-nulos de uma matriz esparsa, tomados coluna a coluna[2]. Para tanto, são utilizados três arranjos, aqui denominados de `COLUNAS`, `LINHAS` e `VALORES`. O primeiro armazena o *índice inicial* de cada coluna nos outros dois arranjos; o arranjo `LINHAS` contém o número da linha de cada elemento, cujo valor é armazenado na mesma posição no arranjo `VALORES`. O arranjo `COLUNAS` tem `N+1` elementos, onde `N` é o número de colunas da matriz esparsa; já os arranjos `LINHAS` e `VALORES` tem `NZ` elementos, onde `NZ` é o número de elementos não-nulos a serem armazenados.

Como exemplo, suponha a matriz $A \in \Re^{n \times n}$,

$$A = \begin{bmatrix} 4 & -1 & -1 & -1 & -1 \\ -1 & 4 & & & \\ -1 & & 4 & & \\ -1 & & & 4 & \\ -1 & & & & 4 \end{bmatrix} \tag{9.1}$$

a qual tem $n = 5$ colunas e um total de $nz = 13$ elementos não-nulos (obviamente, não é esparsa, mas a usamos apenas para fins de explanação do esquema de armazenamento). Observe que a coluna 1 tem 5 elementos; todas as demais tem apenas 2 elementos; essa característica poderá ser facilmente verificada no esquema CCS, cujos arranjos são mostrados na figura 9.1.

Observe que a primeira coluna ocupa as primeiras cinco posições dos arranjos `LINHAS` e `VALORES` e, portanto, o elemento `COLUNAS(1)=1`. A segunda coluna, a qual contém apenas dois elementos, começa a partir da posição 6 nos arranjos `LINHAS` e `VALORES`, de onde `COLUNAS(2)=6`. O mesmo processo é repetido para as demais colunas; observe que é necessário armazenar na posição `N+1` do arranjo `COLUNAS` o valor `NZ+1`. Por inspeção, pode-se concluir que cada coluna `J` está contida nas posições `COLUNAS(J)` a `COLUNAS(J+1)-1` dos arranjos `LINHAS` e `VALORES`.

	COLUNAS	1		6		8		10		12		14	
	posição	1		2		3		4		5		6	
LINHAS	1	2	3	4	5	1	2	1	3	1	4	1	5
VALORES	4	-1	-1	-1	-1	-1	4	-1	4	-1	4	-1	4
posição	1	2	3	4	5	6	7	8	9	10	11	12	13

Figura 9.1: O esquema de armazenamento CCS.

Para implementar o esquema CCS em FORTRAN 90, podemos fazer uso de tipos definidos pelo usuário, criando um estrutura de dados que armazene todas as informações necessárias: os valores de `N` e `NZ` e os três arranjos. Isso é feito, por exemplo, com os comandos

```
TYPE ESPARSA
    INTEGER :: N, NZ
    INTEGER, DIMENSION(:), ALLOCATABLE :: COLUNAS
    INTEGER, DIMENSION(:), ALLOCATABLE :: LINHAS
    REAL, DIMENSION(:), ALLOCATABLE :: VALORES
END TYPE ESPARSA
```

o qual cria um tipo de dado chamado `ESPARSA`; uma matriz esparsa armazenada no esquema CCS poderá, agora, ser declarada como sendo desse tipo. O uso da estrutura de dados não é obrigatório: poderíamos usar duas variáveis inteiras `N` e `NZ` e os três arranjos como entidades separadas. No entanto, se não usamos a estrutura de dados, é possível que erros sejam cometidos ao se desenvolver um programa que manipule tais dados.

Suponha, agora, que se deseja escrever um conjunto de subprogramas que permitam ler de um arquivo uma matriz esparsa, armazenando-a no esquema CCS, bem como calcular os produtos

[2]Tal formato de armazenamento serve também para representar um grafo, através da matriz de adjacência do grafo.

matriz-vetor e matriz-transposta-vetor (operações típicas de utilização de matrizes esparsas). Para tanto, criamos um módulo, conforme mostrado no exemplo 9.1: nele estão presentes as declarações do tipo ESPARSA, bem como as subrotinas LE_MATRIZ, PRODUTO_MATRIZ_VETOR e PRODUTO_MATRIZ_TRANSPOSTA_VETOR.

A subrotina LE_MATRIZ lê um arquivo conforme o formato existente em vários repositórios universais de matrizes esparsas, como o MATRIX MARKET [9]. O uso desse formato é proposital, de forma que o leitor tome contato com rotinas de uso comum nesse tipo de aplicação. Os detalhes de como o arquivo é montado podem ser obtidos em [9].

A subrotina PRODUTO_MATRIZ_VETOR é uma implementação do produto matriz-vetor para matrizes armazenadas no esquema CCS. Se $u, v \in \mathbb{R}^n$, então o produto $v = Au$ pode ser escrito como

$$v = \sum_{j=1}^{n} u_j A_{:,j} \tag{9.2}$$

onde o símbolo : indica todas as linhas de uma coluna, de forma semelhante ao operador : da linguagem FORTRAN 90. Particularizando a equação 9.2 para o esquema CCS, podemos escrever

$$\texttt{V} = 0 \tag{9.3}$$

$$\texttt{V(LINHAS(I))} = \texttt{V(LINHAS(I))} + \texttt{U(J)} \times \texttt{VALORES(I)}, \tag{9.4}$$
$$\texttt{I} = \texttt{COLUNAS(J)}, \ldots, \texttt{COLUNAS(J+1)-1}, \quad \texttt{J} = 1, \ldots, \texttt{N}$$

De forma semelhante, o produto matriz-transposta-vetor $v = A^T u$ pode ser escrito – sem que se obtenha explicitamente a matriz transposta – através de

$$\texttt{V} = 0 \tag{9.5}$$

$$\texttt{V(J)} = \texttt{V(J)} + \texttt{U(LINHAS(I))} \times \texttt{VALORES(I)}, \tag{9.6}$$
$$\texttt{I} = \texttt{COLUNAS(J)}, \ldots, \texttt{COLUNAS(J+1)-1}, \quad \texttt{J} = 1, \ldots, \texttt{N}$$

Exemplo 9.1 *Manipulação de matrizes esparsas*

```
MODULE MATRIZ_ESPARSA
TYPE ESPARSA
    INTEGER :: N, NZ
    INTEGER, DIMENSION(:), ALLOCATABLE :: COLUNAS
    INTEGER, DIMENSION(:), ALLOCATABLE :: LINHAS
    REAL, DIMENSION(:), ALLOCATABLE :: VALORES
END TYPE ESPARSA
CONTAINS
    SUBROUTINE LE_MATRIZ(MATRIZ,ARQUIVO)
    IMPLICIT NONE
    TYPE (ESPARSA), INTENT(OUT) :: MATRIZ
    CHARACTER*(*), INTENT(IN) :: ARQUIVO
    ! Le uma matriz no formato Harwell
    ! Codigo baseado no exemplo dado em
    ! Duff, I.S., Grimes, R.G. e Lewis, J.G.,
    ! Users' Guide for the Harwell-Boeing
    ! Sparse Matrix Collection (Release I),
    ! Report TR/PA/92/86, CERFACS, pp. 14-16
    INTEGER :: INDCRD,MAXIT,NCOL,NELTVL,NNZERO,NRHS,NROW,NRSHIX, &
                    PTRCRD,RHSCRD,TOTCRD,VALCRD,I,J
    LOGICAL :: EXISTE
    CHARACTER*3 :: MXTYPE,RHSTYP
    CHARACTER*8 :: KEY
    CHARACTER*16 :: INDFMT,PTRFMT
    CHARACTER*20 :: RHSFMT,VALFMT
    CHARACTER*72 :: TITLE
```

```
    INQUIRE(FILE=ARQUIVO,EXIST=EXISTE)
    IF (.NOT.EXISTE) THEN
            PRINT *,'Arquivo não-encontrado.'
        RETURN
    END IF

    OPEN(UNIT=1,FILE=ARQUIVO,STATUS='old')
    ! Le cabecalho
    READ(UNIT=1,FMT=9010)TITLE,KEY,TOTCRD,PTRCRD,INDCRD,VALCRD, &
                RHSCRD,MXTYPE,NROW,NCOL,NNZERO,NELTVL,PTRFMT,INDFMT,VALFMT, &
                RHSFMT

    IF (RHSCRD>0) THEN
        READ(UNIT=1,FMT=9020) RHSTYP,NRHS,NRSHIX
    END IF
    ! Le estrutura da matriz
    MATRIZ%N = NCOL
    MATRIZ%NZ = NNZERO
    ALLOCATE(MATRIZ%COLUNAS(MATRIZ%N+1))
    ALLOCATE(MATRIZ%LINHAS(MATRIZ%NZ))
    ALLOCATE(MATRIZ%VALORES(MATRIZ%NZ))
    READ(UNIT=1,FMT=PTRFMT)(MATRIZ%COLUNAS(I),I=1,MATRIZ%N+1)
    READ(UNIT=1,FMT=INDFMT)(MATRIZ%LINHAS(I),I=1,MATRIZ%NZ)
    IF (VALCRD>0) THEN
        ! Le valores da matriz
        IF (MXTYPE(3:3)=='A') THEN
                READ(UNIT=1,FMT=VALFMT)(MATRIZ%VALORES(I),I=1,MATRIZ%NZ)
            ELSE
                PRINT *,'Erro no formato do arquivo'
            CLOSE(UNIT=1)
            RETURN
        END IF
        END IF

    CLOSE(UNIT=1)
    9000 FORMAT(A,/,4 (E16.10,1X))
    9010 FORMAT(A72,A8,/,5I14,/,A3,11X,4I14,/,2A16,2A20)
    9020 FORMAT(A3,11X,2I14)
END SUBROUTINE LE_MATRIZ

SUBROUTINE PRODUTO_MATRIZ_VETOR(MATRIZ,U,V)
IMPLICIT NONE
TYPE (ESPARSA), INTENT(IN) :: MATRIZ
REAL, DIMENSION(*), INTENT(IN) :: U
REAL, DIMENSION(*), INTENT(OUT) :: V
INTEGER :: I,II,J,N
    V(1:MATRIZ%N)=0.0
    DO J=1,MATRIZ%N
        DO I=MATRIZ%COLUNAS(J),MATRIZ%COLUNAS(J+1)-1
            II=MATRIZ%LINHAS(I)
            V(II)=V(II)+U(J)*MATRIZ%VALORES(I)
        END DO
    END DO
END SUBROUTINE PRODUTO_MATRIZ_VETOR

SUBROUTINE PRODUTO_MATRIZ_TRANSPOSTA_VETOR(MATRIZ,U,V)
IMPLICIT NONE
TYPE (ESPARSA), INTENT(IN) :: MATRIZ
```

```
        REAL, DIMENSION(*), INTENT(IN) :: U
        REAL, DIMENSION(*), INTENT(OUT) :: V
        INTEGER :: I,II,J,N
            V(1:MATRIZ%N)=0.0
            DO J=1,MATRIZ%N
                DO I=MATRIZ%COLUNAS(J),MATRIZ%COLUNAS(J+1)-1
                    II=MATRIZ%LINHAS(I)
                    V(J)=V(J)+U(II)*MATRIZ%VALORES(I)
                END DO
            END DO
        END SUBROUTINE PRODUTO_MATRIZ_TRANSPOSTA_VETOR
    END MODULE MATRIZ_ESPARSA
```

Exemplo 9.2 *Arquivo contendo a matriz esparsa 9.1*

```
1MATRIZ ESPARSA DE TESTE                                                ESPARSA6
             5           1           1           3           0
RSA                      5           5          13           0
(6I5)           (13I5)          (5E16.8)
    1    6    8   10   12   14
    1    2    3    4    5    1    2    1    3    1    4    1    5
   .40000000+001  -.10000000+001  -.10000000+001  -.10000000+001  -.10000000+001
  -.20000000+001   .40000000+001  -.20000000+001   .40000000+001  -.20000000+001
   .40000000+001  -.20000000+001   .40000000+001
```

O exemplo 9.3 mostra como utilizar o módulo MATRIZ_ESPARSA; o arquivo mostrado no exemplo 9.2 é chamado de esparsa6.

Exemplo 9.3 *Manipulação de matrizes esparsas*

```
    PROGRAM TESTE
    USE MATRIZ_ESPARSA
    IMPLICIT NONE
    TYPE (ESPARSA) :: A
    REAL, DIMENSION(:), ALLOCATABLE :: U, V
    INTEGER :: I
        CALL LE_MATRIZ(A,'esparsa6')
        PRINT *,'A='
        PRINT *,A%N, A%NZ
        PRINT *,A%COLUNAS
        PRINT *,A%LINHAS
        PRINT *,A%VALORES
        ALLOCATE(U(A%N))
        ALLOCATE(V(A%N))
        DO I=1,A%N
            U(I) = I
        END DO
        CALL PRODUTO_MATRIZ_VETOR(A,U,V)
        PRINT *,V
        CALL PRODUTO_MATRIZ_TRANSPOSTA_VETOR(A,U,V)
        PRINT *,V
    END PROGRAM TESTE
```

9.4 Usando a biblioteca LAPACK

Em muitas aplicações de cunho científico é necessário trabalhar com operações envolvendo matrizes e/ou vetores [6], como calcular a solução de um sistema de equações lineares, obter os autovalores e autovetores de uma matriz, dentre outras. Existem inúmeras bibliotecas de subprogramas que

implementam tais operações. Algumas são comerciais, como as bibliotecas NAG e IMSL; outras são domínio público, como o "Linear Algebra PACKage" (LAPACK) [1]. O LAPACKé escrito em Fortran 77 (com interfaces disponíveis nas linguagens FORTRAN 90 e C/C++) e apresenta subprogramas de alta qualidade, escritos por alguns dos melhores analistas numéricos, estando disponíveis para dados *reais* ou *complexos*, em precisão *simples* ou *dupla*. Além disso, pode ser otimizada de forma a se alcançar desempenhos próximos do máximo possível em um determinado computador, em termos do número de operações aritméticas de ponto-flutuante realizadas por segundo (MFLOPS); para tanto, os subprogramas presentes no LAPACKpodem utilizar operações em bloco, o que é indicado em certos computadores. Mais detalhes podem ser encontrados em [1, pág. 114–116], [6] e [4].

O exemplo 9.4 mostra como utilizar duas subrotinas do LAPACK, em precisão simples, a fim de se calcular a inversa de uma matriz $A \in \mathbb{R}^{n\times n}$ (tal operação pode ser necessária em algumas situações, apesar de não ser recomendada para se obter a solução de um sistema de equações lineares). Para se obter tal inversa, inicialmente iremos calcular a fatoração $A = PLU$, onde $L \in \mathbb{R}^{n\times n}$ e $U \in \mathbb{R}^{n\times n}$ são os fatores triangulares inferior e superior de A; e $P \in \mathbb{R}^{n\times n}$ é uma matriz de permutação (a qual pode ser armazenada como um vetor, de tal forma que, se $p_i = j$, isso significa que a linha j foi trocada com a linha i – essa é a forma que o LAPACKütiliza). Essa fatoração é obtida com uma chamada à subrotina `SGETRF` (a letra inicial `S` indica que os dados são reais em precisão simples; outras opções são `D`, para reais em precisão dupla; `C`, para complexos em precisão simples; e `Z`, para complexos em precisão dupla).

Uma vez obtida a fatoração PLU de A, podemos calcular a matriz inversa de A, através da relação $A^{-1} = U^{-1}L^{-1}$, de onde $A^{-1}L = U^{-1}$. Resolve-se então esse sistema, obtendo-se A^{-1}; isso é feito através da subrotina `SGETRI`. Essa subrotina, assim como outras no LAPACK, exige que se forneça como um de seus argumentos um arranjo auxiliar de trabalho, cujo *tamanho ótimo* é definido como o produto entre o número de linhas da matriz e o tamanho ótimo de um *bloco*, conforme definido durante a instalação da biblioteca LAPACKno computador em uso. Por tamanho ótimo, entende-se aquele tamanho que permite que a subrotina seja executada no menor tempo possível. Cabe ressaltar que cada subprograma tem seus requerimentos específicos de tamanhos de arranjos auxiliares, os quais estão especificados no guia do usuário [1].

Exemplo 9.4 *Calculando a matriz inversa usando o* LAPACK

```
PROGRAM USANDO_LAPACK
IMPLICIT NONE
REAL, DIMENSION(:,:), ALLOCATABLE :: MAT
REAL, DIMENSION(:), ALLOCATABLE :: PIVOS, TRABALHO
INTEGER :: I, J, N, RESULTADO, ILAENV, NB
EXTERNAL ILAENV
    ! Lê a ordem da matriz e aloca a matriz MAT e o vetor PIVOS
    PRINT *,'N=?'
    READ *,N
    ALLOCATE(MAT(N,N))
    ALLOCATE(PIVOS(N))

    ! Armazena valores na matriz MAT
    DO J=1,N
        DO I=1,N
            IF (I==J) THEN
                MAT(I,J) = REAL(2*N)
            ELSE
                MAT(I,J) = -1.0
            END IF
        END DO
    END DO

    ! Calcula a fatoração LU da matriz MAT usando a rotina SGETRF
    CALL SGETRF(N,N,MAT,N,PIVOS,RESULTADO)
```

```
WRITE(*,'(1X,A,": RESULTADO=",I2)')"SGETRF",RESULTADO

! Consulta o tamanho do bloco para a rotina SGETRI
NB = ILAENV(1,'SGETRI','',N,N,N,N)

! Aloca o vetor TRABALHO com o tamanho recomendado
ALLOCATE(TRABALHO(N*NB))

! Calcula a inversa usando a rotina SGETRI
CALL SGETRI(N,MAT,N,PIVOS,TRABALHO,N,RESULTADO)
WRITE(*,'(1X,A,": RESULTADO=",I2)')"SGETRI",RESULTADO

! Desaloca os arranjos utilizados
DEALLOCATE(TRABALHO)
DEALLOCATE(PIVOS)
DEALLOCATE(MAT)
END PROGRAM USANDO_LAPACK
```

9.5 Solução de uma equação diferencial parcial

Uma das principais aplicações em que se utiliza a linguagem FORTRAN 90 é na solução de uma equação diferencial parcial. Aqui, vamos apresentar um caso simples, mas que ilustra como utilizar os recursos disponíveis na linguagem para se escrever um programa que resolva tal problema.

Suponha o problema de se calcular a distribuição de temperatura ao longo de uma barra de metal, isolada termicamente, cujas extremidades encontram-se ligadas a blocos de gelo a uma temperatura $u = 0$. A barra, por sua vez, encontra-se inicialmente (no tempo $t = 0$) a uma temperatura $u = 1000$. A equação diferencial parcial com a qual modelamos esse fenômeno é (ver [10])

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{9.7}$$

sujeita às condições iniciais

$$u = \begin{cases} 0, & x = 0, \quad x = 1 \\ 1000, & 0 < x < 1 \end{cases} \qquad t = 0 \tag{9.8}$$

e condições de fronteira

$$u = \begin{cases} 0, & x = 0 \\ 0, & x = 1 \end{cases}, \quad t > 0 \tag{9.9}$$

Para aproximar a equação (9.7) por diferenças finitas (ver [10]), sobrepomos uma malha com $n_x + 2$ células sobre a barra (onde a primeira e última célula correspondem à fronteira). A aproximação ao longo do tempo dá-se supondo que a malha é repetida n_t vezes. Em cada uma dessas células, iremos aproximar (9.7), substituindo as derivadas parciais por aproximações em diferenças finitas de primeira e segunda ordem, obtendo

$$\frac{u_{i+1,j} - u_{i,j}}{h_t} = \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_x^2}, \quad h_t = \frac{1}{n_t + 1}, \quad h_x = \frac{1}{n_x + 1} \tag{9.10}$$

de onde, isolando $u_{i+1,j}$, vem

$$u_{i+1,j} = au_{i,j} + r(u_{i,j-1} + u_{i,j+1}), \quad a = 1 - 2r, \quad r = \frac{h_t}{h_x^2} \tag{9.11}$$

A expressão $u_{i+1,j}$ representa a temperatura em uma célula j da barra no tempo $i+1$. A equação (9.11) será utilizada dentro do programa para calcular os valores de u em cada célula.

Note que, como h_x foi definido, então a célula $j = 0$ da malha para u corresponde a $x = 0$; e a célula $j = n_x + 1$ corresponde a $x = 1$. Logo, a malha para u deve ser um arranjo U com uma

dimensão, cujos índices são 0 e $NX + 1$, respectivamente; observe a declaração de U no programa 9.5. O uso do atributo ALLOCATABLE em sua declaração é apenas para tornar o programa mais genérico em termos de utilização.

Agora, como implementar a equação (9.11)? Observe que, para se calcular o valor da célula u_j no tempo $i + 1$, necessitamos dos valores das células u_j, u_{j-1} e u_{j+1} no tempo i. Ora, como isso é válido para todas as células, podemos notar que exatamente $3n_x$ células podem ser operadas simultaneamente, sendo que dois conjuntos de n_x células são defasadas de 1 célula à direita e à esquerda. Evidentemente, poderíamos traduzir para FORTRAN 90 da forma tradicional, usando um laço DO...END DO:

```
DO I=1,NX
    U(I)=A*U(I)+R*(U(I-1)+U(I+1))
END DO
```

porém, em FORTRAN 90, podemos escrever de forma mais compacta, com o uso do operador : (ver §4.5.2),

```
U(1:NX) = A*U(1:NX)+R*(U(0:NX-1)+U(2:NX+1))
```

onde deve-se ressaltar que cada operando no lado direito da atribuição – U(1:NX), U(0:NX-1) e U(2:NX+1) – tem o mesmo número de elementos, assim como a variável que irá receber o resultado, U(1:NX). O programa 9.5 apresenta uma implementação em FORTRAN 90 para se resolver o problema em questão usando as idéias aqui apresentadas.

Exemplo 9.5 *Calculando a temperatura numa barra.*

```
PROGRAM CALOR
INTEGER :: NT, NX, T, I
REAL(8), DIMENSION(:), ALLOCATABLE :: U
REAL(8) :: A, R
    PRINT *,'NT=?, NX=?'
    READ *,NT,NX
    ! Calcula constantes necessárias
    HT = 1.0/(NT+1)
    H = 1.0/(NX+1)
    R = HT/(H**2)
    IF (R<0.5) THEN
        ! Condição inicial: (t=0)
        ! U=0    em x=0
        ! U=1000 em 0<x<1
        ! U=0    em x=1
        ALLOCATE(U(0:NX+1))
        U(0) = 0.0
        U(1:NX) = 1000.0
        U(NX+1) = 0.0
        ! Inicia iterações ao longo do tempo
        A = 1.0-2.0*R
        DO T=1,NT
            ! Aplica equação em diferenças-finitas para
            ! células interiores de U
            U(1:NX) = A*U(1:NX) + R*(U(2:NX+1) + U(0:NX-1))

            ! Força condição de fronteira:
            ! U=0 em x=0 e x=1
            U(0) = 0.0
            U(NX+1) = 0.0
        END DO
        WRITE(*,'(" U(T=",I7,")=")')NT
```

```
        WRITE(*,'(4(F16.14,1X))')(U(I),I=0,NX+1)
        DEALLOCATE(U)
    ELSE
        PRINT *,'CONDIÇÃO R<0.5 VIOLADA, PARANDO!'
    END IF
END PROGRAM CALOR
```

9.6 Listas encadeadas

[11]Em certas situações, é interessante utilizar-se uma *lista encadeada*, ao invés de um arranjo, para armazenar valores. Tais situações caracterizam-se por apresentarem um grande número de inserções e remoções de valores da estrutura de dados; ao usarmos uma lista encadeada, armazenamos apenas os valores realmente existentes, o que permite um melhor controle da memória.

Uma lista encadeada é uma estrutura de dados composta por *nodos*, ligados entre si através de ponteiros (ver §3.6.13 e §4.4). Num nodo, armazenamos não só os valores desejados (por exemplo, o código de um produto), os quais podem ser armazenados, por sua vez, em estruturas de dados mais complexas, mas, também, um ou mais ponteiros que ligam esse nodo ao(s) seu(s) nodo(s) vizinho(s). As listas encadeadas podem ser de diferentes tipos: *lineares* ou *circulares*, com encadeamento *simples* ou *duplo*. Nessa seção, consideraremos um lista encadeada linear com encadeamento simples; listas de outros tipos podem ser implementadas com poucas modificações sobre os códigos aqui apresentados.

Como qualquer estrutura de dados, necessitamos definir como serão efetuadas algumas operações básicas sobre a lista, a saber: *inserção*, *remoção*, *visitação* e *localização*. Além dessas, é necessário definir como identificar se a lista está *vazia*. Para tanto, vamos definir um nodo da lista como contendo um valor escalar inteiro, chamado de `VALOR`, e um ponteiro chamado de `PROXIMO`, o qual apontará para o nodo sucessor a ele. Em FORTRAN 90, podemos definir tal nodo como

```
TYPE NODO
    INTEGER :: VALOR
    TYPE (NODO), POINTER :: PROXIMO
END TYPE NODO
```

Observe que o ponteiro `PROXIMO` aponta para uma estrutura de dados do tipo `NODO`. A lista, por sua vez, apresenta também dois ponteiros – `INICIO` e `FIM` – que apontarão para o primeiro e último nodos inseridos, sendo definida como

```
TYPE LISTA
    TYPE (NODO), POINTER :: INICIO, FIM
END TYPE LISTA
```

Quando a lista estiver vazia, os ponteiros `INICIO` e `FIM` serão nulos; dessa forma, antes de manipularmos a lista, devemos anulá-los com o comando `NULLIFY` (ver 4.4).

Definimos, agora, que a operação de *inserção* será feita sempre ao *final* da lista; isto significa que, a cada inserção efetuada, o ponteiro `FIM` irá apontar para o novo nodo inserido (como se estivesse "movendo-se" ao longo da lista). Uma vez alocada uma área de dados do tipo nodo – cuja parte `VALOR` deve armazenar o valor que se deseja inserir na lista e o ponteiro `PROXIMO` deve ser anulado – pode-se proceder à inserção desse nodo na lista. Para a inserção, existem duas situações possíveis: ou a lista é vazia, ou já existem nodos na lista. No primeiro caso, basta fazer os ponteiros `INICIO` e `FIM` apontarem para o nodo alocado; já no segundo, deve-se fazer com que o ponteiro `PROXIMO` do nodo apontado por `FIM` aponte para o nodo alocado e, após, se aponte `FIM` para o nodo alocado. A operação de inserção é implementada pela subrotina `INSERE`.

A operação de *localização* de um nodo é feita com base em um valor de busca. Para tal, basta percorrer a lista, usando um ponteiro para "caminhar" na lista – chamado de *ponteiro de varredura*; além disso, já que a operação de localização será utilizada dentro da operação de remoção, é necessário que se tenha um outro ponteiro, que aponta para o nodo *antecessor* do

nodo apontado pelo ponteiro de varredura da lista. Quando o conteúdo da parte VALOR do nodo apontado pelo ponteiro de varredura for igual ao valor de busca, então a busca é encerrada, e um argumento de tipo lógico ACHOU é retornada com o valor .TRUE.; se o valor de busca não for encontrado, então ACHOU conterá o valor .FALSE. Essa operação é implementada pela subrotina LOCALIZA.

A operação de visitação da lista (implementada na subrotina MOSTRA) é semelhante à localização de um nodo, com a diferença que se utiliza apenas o ponteiro de varredura; a cada nodo visitado, o conteúdo da parte VALOR é escrito.

A operação de *remoção* pode ser definida de diferentes maneiras: por exemplo, poderíamos remover sempre o nodo inicial da lista, apontado por INICIO. Aqui, no entanto, iremos remover um nodo cuja parte VALOR seja igual a um dado valor de busca. Para remover tal nodo, necessitamos primeiro localizá-lo, usando para isso a subrotina LOCALIZA. Se o nodo foi localizado (conforme indicado pelo valor do argumento ACHOU), então deve-se proceder à remoção do nodo. Observe que, nesse caso, o nodo a ser removido é apontado por um apontador, digamos, Q, e o nodo anterior a ele por outro (Q_ANTERIOR). Existem, então, três situações possíveis:

1. Q aponta para INICIO: nesse caso, basta fazer com que INICIO aponte para o próximo nodona lista, i.e. aquele apontado por INICIO%PROXIMO;

2. Q aponta para FIM: aqui, o ponteiro FIM deverá apontar para o nodo antecessor, Q_ANTERIOR;

3. Q aponta para um nodo que não está nem no início nem no fim da lista: nesse caso, é necessário que o ponteiro PROXIMO do nodo apontado por Q_ANTERIOR passe a apontar para o nodo apontado pelo ponteiro PROXIMO do nodo apontado por Q.

Os códigos em FORTRAN 90 que implementam as operações descritas acima são apresentados no exemplo 9.6. Nesse módulo, existe uma outra subrotina, EXISTE, a qual simplesmente mostra o resultado da operação de localização, sem que num programa principal haja a necessidade de se declarar dois ponteiros para serem usados como argumentos da subrotina LOCALIZA. Um programa que utiliza tais operações, inserindo, removendo, localizando e mostrando o conteúdo da lista é apresentado no exemplo 9.7; observe que é necessário que se chame explicitamente a subrotina INICIALIZA no programa principal, antes de se começar a operar com a lista.

Exemplo 9.6 *Módulo para manipulação de listas encadeadas.*

```
MODULE MODULO_LISTA
TYPE NODO
    INTEGER :: VALOR
    TYPE (NODO), POINTER :: PROXIMO
END TYPE NODO

TYPE LISTA
    TYPE (NODO), POINTER :: INICIO, FIM
END TYPE LISTA

CONTAINS
    SUBROUTINE INICIALIZA(L)
    TYPE (LISTA) :: L
        NULLIFY(L%INICIO)
        NULLIFY(L%FIM)
    END SUBROUTINE INICIALIZA

    SUBROUTINE INSERE(L,V)
    TYPE (LISTA) :: L
    TYPE (NODO), POINTER :: Q
    INTEGER, INTENT(IN) :: V
        NULLIFY(Q)
        ALLOCATE(Q)
```

```
    Q%VALOR = V
    NULLIFY(Q%PROXIMO)
    IF (ASSOCIATED(L%INICIO).AND.ASSOCIATED(L%FIM)) THEN
        L%FIM%PROXIMO => Q
    ELSE
        L%INICIO => Q
    END IF
    L%FIM => Q
END SUBROUTINE INSERE

SUBROUTINE LOCALIZA(L,V,ACHOU,Q,Q_ANTERIOR)
TYPE (LISTA) :: L
TYPE (NODO), POINTER :: Q,Q_ANTERIOR
INTEGER, INTENT(IN) :: V
LOGICAL, INTENT(OUT) :: ACHOU
    ACHOU = .FALSE.
    NULLIFY(Q_ANTERIOR)
    Q => L%INICIO
    DO WHILE (ASSOCIATED(Q).AND.(.NOT.ACHOU))
        IF (Q%VALOR == V) THEN
            ACHOU = .TRUE.
        ELSE
            Q_ANTERIOR => Q
            Q => Q%PROXIMO
        END IF
    END DO
END SUBROUTINE LOCALIZA

SUBROUTINE EXISTE(L,V,ACHOU)
TYPE (LISTA) :: L
TYPE (NODO), POINTER :: Q,Q_ANTERIOR
INTEGER, INTENT(IN) :: V
LOGICAL, INTENT(OUT) :: ACHOU
    CALL LOCALIZA(L,V,ACHOU,Q,Q_ANTERIOR)
END SUBROUTINE EXISTE

SUBROUTINE MOSTRA(L)
TYPE (LISTA) :: L
TYPE (NODO), POINTER :: Q
    Q => L%INICIO
    IF (ASSOCIATED(Q)) THEN
        PRINT *,'LISTA:'
        DO WHILE (ASSOCIATED(Q))
            WRITE(*,'(I10)',ADVANCE='NO')Q%VALOR
            IF (ASSOCIATED(Q,TARGET=L%INICIO)) THEN
                WRITE(*,'("<- INICIO")',ADVANCE='NO')
            END IF
            IF (ASSOCIATED(Q,TARGET=L%FIM)) THEN
                WRITE(*,'("<- FIM")',ADVANCE='NO')
            END IF
            WRITE(*,*)
            Q => Q%PROXIMO
        END DO
    ELSE
        PRINT *,'LISTA VAZIA'
    END IF
END SUBROUTINE MOSTRA
```

```
    SUBROUTINE REMOVE(L,V)
    TYPE (LISTA) :: L
    TYPE (NODO), POINTER :: Q,Q_ANTERIOR
    INTEGER, INTENT(IN) :: V
    LOGICAL :: ACHOU
        CALL LOCALIZA(L,V,ACHOU,Q,Q_ANTERIOR)
        IF (ACHOU) THEN
            IF (ASSOCIATED(L%INICIO,TARGET=L%FIM)) THEN ! INÍCIO DA LISTA
                NULLIFY(L%INICIO)
                NULLIFY(L%FIM)
            ELSE IF (ASSOCIATED(Q,TARGET=L%FIM)) THEN ! FIM DA LISTA
                L%FIM => Q_ANTERIOR
                NULLIFY(L%FIM%PROXIMO)
            ELSE
                Q_ANTERIOR%PROXIMO => Q%PROXIMO ! MEIO DA LISTA
            END IF
            DEALLOCATE(Q)
        END IF
    END SUBROUTINE REMOVE

    SUBROUTINE DESTROI(L)
    TYPE (LISTA) :: L
    TYPE (NODO), POINTER :: Q,R
        Q => L%INICIO
        DO WHILE (ASSOCIATED(Q))
            R => Q
            Q => Q%PROXIMO
            DEALLOCATE(R)
        END DO
        CALL INICIALIZA(L)
    END SUBROUTINE DESTROI
END MODULE MODULO_LISTA
```

Exemplo 9.7 *Programa principal com manipulação de listas encadeadas.*

```
PROGRAM USA_LISTA
USE MODULO_LISTA
TYPE (LISTA) :: MINHA_LISTA
INTEGER :: VALOR
LOGICAL :: CONTINUA = .TRUE.
CHARACTER :: ESCOLHA
LOGICAL :: ACHOU
    CALL INICIALIZA(MINHA_LISTA)
    DO WHILE (CONTINUA)
        PRINT *,'I(NSERE), R(EMOVE), M(OSTRA), L(OCALIZA), T(ERMINA)?'
        READ(*,'(A1)')ESCOLHA
        SELECT CASE (ESCOLHA)
            CASE ('I','i')
                PRINT *,'VALOR=?'
                READ(*,'(I10)',ERR=999)VALOR
                CALL INSERE(MINHA_LISTA,VALOR)
            CASE ('R','r')
                PRINT *,'VALOR=?'
                READ(*,'(I10)',ERR=999)VALOR
                CALL REMOVE(MINHA_LISTA,VALOR)
            CASE ('M','m')
                CALL MOSTRA(MINHA_LISTA)
            CASE ('L','l')
                PRINT *,'VALOR=?'
```

```
            READ(*,'(I10)',ERR=999)VALOR
            CALL EXISTE(MINHA_LISTA,VALOR,ACHOU)
            IF (ACHOU) THEN
                PRINT *,'VALOR ',VALOR,' EXISTE NA LISTA'
            ELSE
                PRINT *,'VALOR ',VALOR,' NÃO EXISTE NA LISTA'
            END IF
        CASE ('T','t')
            CALL DESTROI(MINHA_LISTA)
            CONTINUA = .FALSE.
        CASE DEFAULT
            PRINT *,'***OPCAO INVÁLIDA!'
    END SELECT
END DO
STOP
! CASO HAJA ERRO NA LEITURA
999 PRINT *,'***ERRO NA LEITURA, TERMINANDO!'
    CALL DESTROI(MINHA_LISTA)
END PROGRAM USA_LISTA
```

Anexo A

Compilação de um programa em Fortran 90

Nesse capítulo, apresentaremos de forma sucinta como compilar um programa escrito em FORTRAN 90, em ambiente UN*X, já que cada compilador apresenta um conjunto diferente de opções que podem ser selecionadas pelo usuário. Mais detalhes devem ser verificados no manual de utilização do compilador.

Em ambientes UN*X (como o Linux, por exemplo), o comando para se ativar o compilador FORTRAN 90 é, normalmente, `f90`[1].

Suponha, então, que exista um arquivo chamado `principal.f90`, que contém um programa em FORTRAN 90, i.e. uma seção de código `PROGRAM...END PROGRAM`, escrito em formato livre (de acordo com a extensão `.f90`). Para se obter um código executável correspondente a esse programa, deve-se digitar, após o *"prompt"* `$` do UN*X

```
$ f90 principal.f90
```

o que fará com que o compilador analise o código fonte presente naquele arquivo. Se não houver erro de sintaxe, então o compilador gerará o código executável, o qual será armazenado no nome *"default"* `a.out`. Para executá-lo, basta digitar, após o *"prompt"* `$` do UN*X,

```
$ ./a.out
```

onde os caracteres `./` indicam que o arquivo `a.out` encontra-se no diretório atual.

Como salientado anteriormente, cada compilador oferece um sem-número de opções – chamadas de *"flags"* – que podem ser utilizadas pelo programador para controlar como o código executável é gerado. Pode-se, por exemplo, exigir que o compilador insira código que verifique se o acesso a elementos de arranjos é válido; que o código executável seja otimizado, i.e., execute de forma mais rápida; ou, ainda, que o código executável seja armazenado num arquivo com outro nome que não `a.out`. Por exemplo, se digitássemos

```
$ f90 -g -C principal.f90 -o objeto
```

isso indicaria que o código a ser gerado pelo compilador deverá permitir depuração (`-g`); controlar o acesso a arranjos (`-C`); e o código executável será gravado no arquivo `objeto` (especificado após a *"flag"* `-o`). Outras opções devem ser verificadas junto ao manual do usuário ou através do comando UN*X

```
$ man f90
```

[1] Mas pode ser outro; consulte o responsável pelo seu computador para mais detalhes.

o qual mostra como utilizar o compilador e suas diferentes opções.

Caso se utilize módulos ou subprogramas independentes, deve-se compilá-los separadamente. Por exemplo, se o arquivo `modulo.f90` contiver uma seção de código `MODULE...END MODULE`, e o programa `principal.f90` fizer uso do módulo, então pode-se fazer a compilação através dos comandos

```
$ f90 -c modulo.f90
$ f90 principal.f90 modulo.o
```

O primeiro comando faz com que seja efetuada a compilação em separado (através da *"flag"* `-c`). Se não houver erro de sintaxe no módulo `modulo.f90`, então serão gerados dois arquivos: um, contendo um código relocável – `modulo.o` – o qual contém um código intermediário (não executável), correspondente aos subprogramas presentes no módulo, e outro contendo informações sobre o módulo – `modulo.MOD` – o qual será usado pelo compilador no segundo comando, quando se solicita compilar o programa principal presente no arquivo `principal.f90`, e usar o código relocável presente em `modulo.o`.

A.1 Uso de `Makefile`

Um dos bons recursos disponíveis em ambientes UN*X é o da compilação através de um arquivo de comandos, chamado de `Makefile`[2]. Esse arquivo pode conter uma série de cláusulas que especificam como deve ser efetuada a compilação de vários códigos fonte, gerando um ou mais códigos executáveis. O `Makefile` é lido pelo comando `make` (presente em qualquer sistema UN*X)[3].

O exemplo de `Makefile` a seguir irá compilar um conjunto de três diferentes códigos fonte, armazenados nos arquivos `a.f90`, `b.f90` e `principal.f90` (este último contendo o programa principal, o qual chama subprogramas contidos nos arquivos `a.f90` e `b.f90`), e gerará (caso não haja qualquer erro de sintaxe nos três arquivos), um executável chamado `principal`. O programa `principal.f90` também chama subprogramas presentes na biblioteca LAPACK e que encontram-se armazenados num arquivo chamado `liblapack.a`, presente na área de bibliotecas gerais do computador (isto é, acessível por qualquer usuário).

Antes de explicar o significado de cada cláusula, salientamos que o comando `make` é bastante sensível à formatação do `Makefile`; por exemplo, a identação que deve ser incluída em cada linha posterior a uma cláusula *deve* ser feita com a tecla `TAB` (ou `->|`).

Inicialmente, definimos uma série de símbolos. O primeiro a ser (re-)definido é o símbolo reservado `SUFFIXES`, ao qual adicionamos o sufixo `.f90`. Isso deve ser feito de forma que o `make` entenda que arquivos com esse sufixo devam ser tratados por ele (infelizmente, isso ainda não foi incluído nas distribuições UN*X). A seguir, definimos os símbolos `PROG`, `SRCS` e `OBJS`, os quais contém, respectivamente: o nome do código executável, os nomes dos arquivos contendo os códigos fonte, e os nomes dos arquivos contendo código intermediário (que serão gerados pelo compilador).

O símbolo `LIBS` contém a expressão `-llapack`; o símbolo `-l` é um *"flag"* do compilador e diz que os códigos objeto devem ser ligados com a biblioteca indicada a seguir, no caso, `lapack` – isso faz com que o compilador procure pela biblioteca `liblapack.a` nas áreas de armazenamento de bibliotecas do sistema (normalmente, `/usr/lib` e `/usr/local/lib`). O compilador a ser usado é definido atribuindo-se o nome do comando que o ativa ao símbolo `F90` (no caso, `pgf90`), bem como quaisquer *"flags"* de compilação no símbolo `F90FLAGS` (nesse exemplo, estipulou-se `-O`).

Uma vez definidos esses símbolos, passa-se a definir as cláusulas. A primeira é a cláusula `all`; da forma como está definida, i.e. `all: $(PROG)`, significa que, sempre que for digitado o comando `make` na linha de comando UN*X, serão executados os comandos associados a todas as cláusulas necessárias para se produzir o executável `principal` (já que `$(PROG)` contém apenas o nome `principal`). Nesse caso, observe que a cláusula `$(PROG): $(OBJS)` é equivalente a

[2]Esse arquivo também pode ser chamado de `makefile`; se houver dois arquivos, um chamado de `Makefile` e outro de `makefile`, serão executados os comandos listados no primeiro.

[3]Mais detalhes podem ser obtidos através da página de ajuda do comando `make`, digitando-se `man make`.

```
.SUFFIXES: $(SUFFIXES) .f90
PROG =  principal
SRCS =  a.f90 b.f90 principal.f90
OBJS =  a.o b.o principal.o
LIBS = -llapack
F90 = pgf90
F90FLAGS = -O

all: $(PROG)

$(PROG): $(OBJS)
    $(F90) -o $@ $(OBJS) $(LIBS)

clean:
    rm -f $(PROG) $(OBJS)

.f90.o:
    $(F90) $(F90FLAGS) -c $*.f90
```

Figura A.1: Exemplo de `Makefile`.

```
principal: a.o b.o principal.o
```

Isso indica que o código objeto `principal.o` *depende* dos códigos objetos `a.o`, `b.o` e `principal.o`. Dessa forma, será necessário que se produzam esses três códigos objeto: isso é feito através da cláusula

```
.f90.o:
    $(F90) $(F90FLAGS) -c $*.f90
```

a qual indica como devem ser gerados códigos objeto (com o sufixo `.o`) a partir de códigos fonte, através do uso do compilador `F90`. A expressão `$*.f90` indica que essa compilação será executada para cada arquivo com o sufixo `.f90` presente no mesmo diretório onde reside o arquivo `Makefile`. Observe que a *"flag"* de compilação `-c` é incluída explicitamente, de forma a garantir que sejam gerados apenas códigos objeto e não códigos executáveis.

Uma vez que os três códigos objeto – `a.o`, `b.o` e `principal.o` – tenham sido gerados, então será executado o comando associado à cláusula `$(PROG): $(OBJS)`,

```
    $(F90) -o $@ $(OBJS) $(LIBS)
```

o qual gera o código executável `principal` (indicado pela expressão `-o $@`), através da ligação dos códigos objeto `a.o`, `b.o` e `principal.o` (e resolvendo quaisquer chamadas a subprogramas da biblioteca LAPACK).

Por fim, a cláusula `clean:` permite que sejam removidos os códigos objeto e o executável, desde que se peça explicitamente que ela seja executada, através do comando `make clean`.

O arquivo mostrado na figura A.1 pode ser utilizado para compilar outros programas, bastando para isso modificar convenientemente os valores dos símbolos `PROG`, `SRCS`, `OBJS` e `LIBS`, sem que se modifique quaisquer outras linhas do arquivo.

Bibliografia

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, 1992.

[2] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donald, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods.* SIAM, Philadelphia, 1993.

[3] I. Chivers and J. Sleightholme. *Introducing Fortran 95.* Springer-Verlag, London, 2000.

[4] K. Dowd and C. Severance. *High Performance Computing.* O'Reilly, Sebastopol, 2nd edition edition, 1998.

[5] W. Gehrke. *Fortran 90 Language Guide.* Springer-Verlag, London, 1995.

[6] G.H. Golub and C.F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

[7] J.F. Kerrigan. *Migrating to Fortran 90.* O'Reilly, Sebastopol, 1993.

[8] D.R. Kincaid and W. Cheney. *Numerical Analysis.* Brooks/Cole, Pacific Grove, 1991.

[9] Mathematical and Computational Sciences Division, Information Technology Laboratory, National Institute of Standards and Technology. Matrix Market: A visual repository of test data for use in comparative studies of algorithms for numerical linear algebra. http://math.nist.gov/MatrixMarket/.

[10] G.D. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods.* Oxford University Press, Oxford, 3rd edition, 1985.

[11] P.A.S. Veloso, C.S. dos Santos, A.L. Furtado, and P.A. Azeredo. *Estruturas de Dados.* Editora Campus, Rio de Janeiro, 1983.

Índice