Introdução ao Fortran 90/95

Apostila preparada para a disciplina de Modelos Computacionais da Física I, ministrada para o Curso de Licenciatura em Física do Departamento de Física, Instituto de Física e Matemática, Fundação Universidade Federal de Pelotas, Pelotas - RS.

Início: Janeiro de 2005.

Versão: 10 de setembro de 2012

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

Sumário

Referências Bibliográficas

[1] Intel® Fortran Compiler for Linux. http://www.intel.com/software/products/compilers/flin/docs/manuals.htm. Acesso em: 01 jun. 2005.

[2] ADAMS, J.C., BRAINERD, W.S., HENDRICKSON, R.A. *The Fortran 2003 handbook: the complete syntax, features and procedures.* Springer, 2009, 712 + xii pp.

[3] CHAPMAN, STEPHEN J. *Fortran 95/2003 for Scientists and Engineers.* McGraw-Hill, 2007, xxvi + 976 pp., 3rd. Edio.

[4] MARSHALL, A. C. Fortran 90 Course Notes. http://www.liv.ac.uk/HPC/HTMLFrontPageF90.html, 1996. Acesso em: 01 jun. 2005.

[5] METCALF, M., REID, J.K., COHEN, M. *Fortran 95/2003 explained.* Oxford University Press, 2004, 416 + xviii pp. (Numerical mathematics and scientific computation).

[6] METCALF, MICHAEL, REID, JOHN. *Fortran 90/95 Explained.* New York : Oxford University Press, 1996, 345 + xv pp.

[7] PAGE, CLIVE G. Professional Programer's Guide to Fortran77. http://www.star.le.ac.uk/ cgp/-prof77.pdf, Leicester, 2001. Acesso em: 01 jun. 2005.

[8] RAMSDEN, S., LIN, F., PETTIPHER, M. A., NOLAND, G. S., BROOKE, J. M. Fortran 90. A Conversion Course for Fortran 77 Programmers. http://www.hpctec.mcc.ac.uk/hpctec/courses/Fortran90/F90course.html, 1995. Acesso em: 01 jun. 2005.

[9] REID, JOHN. The New Features of Fortran 2003. Publicado em: ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1648.pdf, 2004. Acesso em: 02 de março de 2011.

Capítulo 1

Introdução

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

Esta apostila destina-se ao aprendizado da **Linguagem de Programação Fortran 95.**

1.1 As origens da Linguagem Fortran

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

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

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

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

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

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

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

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

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

1.2 O padrão Fortran 90

Após trinta anos de existência, Fortran não mais era a única linguagem de programação disponível para os programadores. Ao longo do tempo, novas linguagens foram desenvolvidas e, onde elas se mostraram mais adequadas para um tipo particular de aplicação, foram adotadas em seu lugar. A superioridade do Fortran sempre esteve na área de aplicações numéricas, científicas, técnicas e de engenharia. A comunidade de usuários do Fortran realizou um investimento gigantesco em códigos, com muitos programas em uso freqüente, alguns com centenas de milhares ou milhões de linhas de código. Isto não significava, contudo, que a comunidade estivesse completamente satisfeita com a linguagem. Vários programadores passaram a migrar seus códigos para linguagens tais como Pascal, C e C++. Para levar a cabo mais uma modernização da linguagem, o comitê técnico X3J3, aprovado pela ANSI, trabalhando como o corpo de desenvolvimento do comitê da ISO (*International Standards Organization,* Organização Internacional de Padrões) ISO/IEC JTC1/SC22/WG5 (doravante conhecido como WG5), preparou um novo padrão, inicialmente conhecido como Fortran 8x, e agora como Fortran 90.

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

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

uma linguagem mais eficiente em supercomputadores. O segundo possibilita aos programadores a expressão de seus problemas em termos de tipos de dados que reproduzem exatamente os conceitos utilizados nas sua elaborações.

1.2.1 Recursos novos do Fortran 90

Um resumo dos novos recursos é dado a seguir:

- O Fortran 90 possui uma maneira para rotular alguns recursos antigos como *em obsolescência* (isto é, tornando-se obsoletos).
- Operações de matrizes.
- Ponteiros.
- Recursos avançados para computação numérica usando um conjunto de funções inquisidoras numéricas.
- Parametrização dos tipos intrínsecos, permitindo o suporte a inteiros curtos, conjuntos de caracteres muito grandes, mais de duas precisões para variáveis reais e complexas e variáveis lógicas agrupadas.
- Tipos de dados derivados, definidos pelo programador, compostos por estruturas de dados arbitrárias e de operações sobre estas estruturas.
- Facilidades na definição de coletâneas denominadas *módulos*, úteis para definições globais de dados e para bibliotecas de subprogramas.
- Exigência que o compilador detecte o uso de construções que não se conformam com a linguagem ou que estejam em obsolescência.
- Um novo formato de fonte, adequado para usar em um terminal.
- Novas estruturas de controle, tais como `SELECT CASE` e uma nova forma para os laços `DO`.
- A habilidade de escrever subprogramas internos e subprogramas recursivos e de chamar subprogramas com argumentos opcionais.
- Alocação dinâmica de dados (matrizes automáticas, matrizes alocáveis e ponteiros).
- Melhoramentos nos recursos de entrada/saída de dados.
- Vários novos subprogramas intrínsecos.

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

Os procedimentos de trabalho adotados pelo comitê X3J3 estabelecem um período de aviso prévio antes que qualquer recurso existente seja removido da linguagem. Isto implica, na prática, um ciclo de revisão, que para o Fortran é de cerca de cinco anos. A necessidade de remoção de alguns recursos é evidente; se a única ação adotada pelo X3J3 fosse de adicionar novos recursos, a linguagem se tornaria grotescamente ampla, com muitos ítens redundantes e sobrepostos. A solução finalmente adotada pelo comitê foi de publicar como uma apêndice ao padrão um conjunto de duas listas mostrando quais os ítens que foram removidos e quais são os candidatos para uma eventual remoção.

A primeira lista contém os *recursos removidos* (*deleted features*). A segunda lista contém os *recursos em obsolescência* (*obsolescent features*), os quais são considerados obsoletos e redundantes, sendo assim candidatos à remoção na próxima revisão da linguagem.

1.2.2 Recursos em obsolescência do Fortran 90

Os recursos em obsolescência do Fortran 90 são:

- `IF` aritmético;
- desvio para uma declaração `END IF` a partir de um ponto fora de seu bloco;
- variáveis reais e de dupla precisão nas expressões de controle de um comando `DO`;

- finalização compartilhada de blocos `DO`, bem como finalização por uma declaração ou comando distintos de um `CONTINUE` ou de um `END DO`;
- declaração `ASSIGN` e comando `GO TO` atribuído;
- `RETURN` alternativo;
- comando `PAUSE;`
- especificadores `FORMAT` atribuídos;
- descritor de edição `H`.

1.2.3 Recursos removidos do Fortran 90

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

1.3 Uma revisão menor: Fortran 95

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

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

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

À medida que os comitês X3J3 e WG5 trabalhavam, este comunicavam-se regularmente com o HPFF. Era evidente que, para evitar o surgimento de dialetos divergentes de Fortran, havia a necessidade de incorporar a sintaxe nova desenvolvida pelo HPFF no novo padrão da linguagem. De fato, os recursos do HPF constituem as novidades mais importantes do Fortran 95. As outras mudanças consistem em correções, clarificações e interpretações do novo padrão. Estas se tornaram prementes quando os novos compiladores de Fortran 90 foram lançados no mercado e utilizados; notou-se uma série de erros e detalhes obscuros que demandavam reparações. Todas estas mudanças foram incluídas no novo padrão Fortran 95, que teve a sua versão inicial lançada no próprio ano de 1995.[2]

O Fortran 95 é compatível com o Fortran 90, exceto por uma pequena alteração na função intrínseca `SIGN` (seção 7.4.2) e a eliminação de recursos típicos do Fortran 77, declarados em obsolescência no Fortran 90. Os detalhes do Fortran 95 foram finalizados em novembro de 1995 e o novo padrão ISO foi finalmente publicado em outubro de 1996.

1.3.1 Recursos novos do Fortran 95

Os novos recursos do Fortran 95 estão discutidos ao longo desta apostila. Em relação ao Fortran 90, foram introduzidos os seguintes recursos:

- Concordância aprimorada com o padrão de aritmética de ponto flutuante binária da IEEE (IEEE 754 ou IEC 559-1989).
- Rotinas *(procedures)* puras (seção 8.2.16).
- Rotinas *(procedures)* elementais (seção 8.2.17).

[2] Fortran 95, Commitee Draft, May 1995. *ACM Fortran Forum*, v. 12, n. 2, June 1995.

- Dealocação automática de matrizes alocáveis (página 65).
- Comando e construto `FORALL` (seção 6.10).
- Extensões do construto `WHERE` (página 62).
- Funções especificadoras.
- Inicialização de ponteiro e a função `NULL` (seção 7.16).
- Inicialização de componentes de tipo derivado (página 27).
- Funções elementares `CEILING`, `FLOOR` e `SIGN` (seções 7.4.1 e 7.4.2).
- Funções transformacionais (7.15).
- Subrotina intrínseca `CPU_TIME` (seção 7.17.2).
- Comentário na entrada de uma lista `NAMELIST` (página 133).
- Descritores de edição com largura de campo mínimo.
- Especificação genérica na cláusula `END INTERFACE`.

1.3.2 Recursos em obsolescência do Fortran 95

Os recursos abaixo entraram na lista em obsolescência do Fortran 95 e, portanto, pouco ou nada foram comentados ao longo desta Apostila.

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

1.3.3 Recursos removidos do Fortran 95

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

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

1.4 O Fortran no Século XXI: Fortran 2003

O Fortran 2003 é novamente uma revisão grande do padrão anterior: Fortran 95. A versão curta *(draft)* final do novo padrão foi divulgada em maio de 2004.[3] Maiores informações podem ser obtidas na página Wiki do Fortran 2003 em: http://fortranwiki.org/fortran/show/Fortran+2003. Embora o Fortran 2003 tenha se tornado o novo padrão da linguagem após a publicação final dos padrões da linguagem, embora até o presente momento somente um compilador apresenta suporte completo do novo padrão. Uma relação dos recursos do Fortran 2003 suportados por diferentes compiladores, tanto livres (ou gratuítos), quanto proprietários, pode ser também obtida na página Wiki em: http://fortranwiki.org/fortran/show/Fortran+2003+status.

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

Esta Apostila não irá abordar os novos recursos introduzidos pelo Fortran 2003, limitando-se a listá-los.

1.4.1 Recursos novos do Fortran 2003

Os principais recursos introduzidos pelo padrão são:

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

1.4.2 Recursos em obsolescência do Fortran 2003

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

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

[3] Ver o *draft* em: http://www.j3-fortran.org/doc/2003_Committee_Draft/04-007.pdf.

- `RETURN` alternativo.
- Comando `GO TO` computado.
- Funções definidas em uma linha *(statement functions)*.
- Declarações `DATA` entre comandos executáveis.
- Funções de caractere de extensão assumida.
- Formato fixo da fonte.
- Forma `CHARACTER*` da declaração `CHARACTER`.

1.4.3 Recursos removidos do Fortran 2003

Um recurso é removido do novo padrão se este for considerado redundante e praticamente sem uso pela comunidade, devido a novos recursos muito mais úteis na construção do código.

A lista de recursos removidos no Fortran 2003, divulgada no *draft*, repete os recursos removidos do Fortran 95. Portanto, nenhum outro recurso foi removido.

1.5 O novo padrão: Fortran 2008

Embora poucos compiladores já suportem completamente o padrão estabelecido pelo Fortran 2003, os comitês X3J3/WG5 já divulgaram, em 07 de junho de 2010, a resenha final do novo padrão da linguagem: o Fortran 2008. O texto final do novo padrão pode ser obtido em `ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1830.pdf`. Uma relação dos recursos do Fortran 2008 suportados por diferentes compiladores, tanto livres (ou gratuítos), quanto proprietários, pode ser também obtida na página Wiki em: `http://fortranwiki.org/fortran/show/Fortran+2008+status`.

1.5.1 Recursos novos do Fortran 2008

O Fortran 2008, embora pelo procedimento adotado pelos comitês X3J3/WG5 devesse ser uma revisão menor do Fortran 2003, apresenta algumas novidades substanciais, tais como as co-matrizes *(co-arrays)* e sub-módulos. Maiores informações podem ser obtidos na página Wiki do Fortran 2008: `http://fortranwiki.org/fortran/show/Fortran+2008`.

Os novos recursos disponíveis na linguagem são:

- Submodules.
- Coarrays.
- do concurrent construct.
- contiguous attribute.
- block construct.
- exit statement.
- error stop statement.
- Internal procedures can be passed as actual arguments.
- Procedure pointers can point to an internal procedure.
- Maximum rank increased to 15.
- newunit= in open statement.
- g0 edit descriptor.
- Unlimited format item.

As seguintes mudanças foram realizadas em rotinas intrínsecas:

- acos, asin, atan, cosh, sinh, tan, and tanh agora aceitam argumentos complexos.

- atan2 pode ser agora chamada simplesmente como atan.
- lge, lgt, lle, and llt agora aceitam argumentos da espécie ASCII.
- maxloc and minloc possuem agora um argumento opcional back=.
- selected_real_kind possui agora um argumento radix=.

Novas rotinas intrínsecas:

- Special functions:
  - Inverse hyperbolic trigonometric functions: acosh, asinh, and atanh.
  - Bessel functions: bessel_j0, bessel_j1, bessel_jn, bessel_y0, bessel_y1, and bessel_yn.
  - Error function: erf, erfc, and erfc_scaled.
  - Gamma function: gamma and log_gamma.
  - Euclidean distance: hypot.
  - L 2 norm: norm2.
- Bitwise operations:
  - Bit sequence comparisons: bge, bgt, ble, and blt.
  - Combined shifting: dshiftl and dshiftr.
  - Counting bits: leadz, trailz, popcnt and poppar.
  - Masking bits: maskl and maskr.
  - Shifting bits: shifta, shiftl and shiftr.
  - Merging bits: merge_bits.
  - Bit transformational functions: iall?, iany?, and iparity?.
- Coarray intrinsics:
  - Convert a cosubscript to an image index: image_index?.
  - Cobounds of a coarray: lcobound? and ucobound?.
  - Number of images: num_images?.
  - Image index or cosubscripts: this_image.
- Other:
  - Test for the contiguous attribute: is_contiguous?.
  - Size of an element in bits: storage_size?.
  - Tests for the number of true values being odd: parity.
  - Search for a value in an array: findloc?.
  - Shell commands: execute_command_line.
  - Define and reference variables atomically: atomic_define? and atomic_ref?.

Adições a módulos intrínsecos:

- iso_fortran_env:
  - Information about the compiler: compiler_version and compiler_options.
  - Named constants for selecting kind values.
- ieee_arithmetic:
  - ieee_selected_real_kind now has a radix= argument.
- iso_c_binding:
  - c_sizeof returns the size of an array element in bytes.

1.5.2 Recursos em obsolescência do Fortran 2008

- `IF` aritmético
- Término de laços `DO` compartilhados e término de laços `DO` por outras instruções exceto `END DO` ou `CONTINUE`.
- `RETURN` alternativo.
- GO TO computado.
- Statement functions.
- Declarações `DATA` entre instruções executáveis.
- Funções de caractere de tamanho assumido.
- Formato de fonte fixo (7ł coluna).
- Forma `CHARACTER*` para declarações de variáveis de caractere.
- Instrução `ENTRY`.

1.5.3 Recursos removidos do Fortran 2008

Em adição aos recursos do Fortran 77 já removidos pelo Fortran 95, os seguintes recursos foram também excluídos do padrão:

- Formato de controle vertical.

1.6 Comentários sobre a bibliografia

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

- Para informações a respeito do padrão existente na linguagem Fortran 77, a qual foi substituída pelo Fortran 90/95, utilizei freqüentemente o livro de Clive Page: Professional Programer's Guide to Fortran 77 [7].
- A principal fonte de informação sobre o padrão do Fortran 90/95 foi o amplo livro de Michael Metcalf e John Reid: Fortran 90/95 Explained [6].
- O curso virtual de Fortran 90 oferecido pela Universidade de Liverpool [4].
- O curso virtual de Fortran 90 para programadores que já conhecem o Fortran 77, oferecido pela Universidade de Manchester [8].
- O manual de referência à Linguagem Fortran 90/95 que acompanha o compilador intel também foi freqüentemente consultado [1].
- Informações divulgadas sobre o Fortran 2003 podem obtidas a partir da internet no livreto de John Reid (2004): The New Features of Fortran 2003 [9]. Uma descrição completa da linguagem pode ser consultada nos livros de Michael Metcalf *et al.* (2004): Fortran 95/2003 Explained [5], Stephen Chapman (2007): Fortran 95/2003 for Scientists and Engineers [3] e Jeanne Adams *et al.* (2009): The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures [2].

1.7 Observações sobre a apostila e agradecimentos

Por se tratar de uma obra em constante revisão, esta apostila pode conter (e conterá, invariavelmente) uma série de erros de ortografia, pontuação, acentuação, *etc.* Em particular, o texto não foi ainda revisado para se conformar com o novo acordo ortográfico da Lingua Portuguesa. Certos pontos poderiam também ser melhor discutidos e podem conter até informações não completamente corretas.

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

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

Capítulo 2

Formato do Código-Fonte

Neste capítulo estuda-se, inicialmente, um formato básico para o programa-fonte em Fortran 90/95 e os tipos de variáveis e suas características básicas.

2.1 Formato do programa-fonte

Em Fortran há três formatos básicos de arquivos que são utilizados:

Programa-Fonte. Trata-se do programa e/ou dos subprogramas escritos pelo programador, usando algum tipo de editor de texto, de acordo com as regras definidas pela linguagem de programação de alto nível.

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

Programa executável. Após a fase de compilação, onde os programas objetos são criados, o agente de compilação aciona o *linker*, o qual consiste em um programa especial que agrupa os programas objetos de forma a criar um arquivo final, o programa executável, o qual pode ser então executado pelo programador.

Nesta seção será apresentada a estrutura básica de um programa-fonte em Fortran 90/95.

O Fortran 90/95 suporta duas formas de código-fonte: o formato antigo do Fortran 77, agora denominado *formato fixo* e o novo *formato livre*.

Fortran 77: O formato fixo foi utilizado pelo Fortran desde a primeira versão até o Fortran 77. Este formato foi determinado pelas restrições impostas pelos cartões perfurados, que eram a única opção para entrar com o código fonte. A estrutura do formato fixo é a seguinte:

Colunas 1-5: rótulos (*labels*) para desvio de processamento do programa.

Coluna 6: caractere de continuação de linha.

Colunas 7-72: código fonte.

Há, ainda, a possibilidade de incluir o caractere "C" ou "c" na coluna 1, o que instrui o compilador a ignorar tudo que segue neste linha. Esta á a forma encontrada para se colocar comentários dentro do código fonte.

Fortran 90/95: Nestas revisões, a linguagem passou a suportar o formato livre, com as seguintes características.

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

- Mais de uma instrução pode ser colocada na mesma linha. O separador de linhas padrão é o ponto-e-vírgula ";". Múltiplos ";" em uma linha, com ou sem brancos, são considerados como um separador simples. Desta forma, a seguinte linha de código é interpretada como 3 linhas em seqüência:

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

- O caractere *ampersand* "&" é a *marca de* continuação, isto é, ele indica que a linha com instruções imediatamente posterior é continuação da linha onde o "&" foi digitado. Desta forma, são permitidas até 39 linhas adicionais de código. Como exemplo, a seguinte linha de código:

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

  também pode ser escrita usando o "&":

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

- Para entrar com comentários em qualquer ponto do código-fonte, o usuário deve digitar o ponto de exclamação "!" em qualquer coluna de uma linha. Todo o restante da linha será desprezado pelo compilador. Exemplo:

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

O programa "Alô Mamãe".

Como primeiro exemplo de programa em Fortran 90/95, considere o seguinte código-fonte:

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

A forma mais simples de um programa em Fortran 90/95 é a seguinte:

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

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

```
PROGRAM primeiro
```

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

Em seguida vêm as `declarações de nomes de variáveis.` Neste ponto, são definidos os nomes das variáveis a ser usadas pelo programa. Caso não seja necessário declarar variáveis, como no programa `primeiro`, é recomendável incluir, pelo menos, a instrução `implicit none.` Esta instrui o compilador a exigir que todas as variáveis usadas pelo programa tenham o seu tipo explicitamente definido.[1]

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

```
print *, "Alô Mamãe"
```

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

Finalmente, o programa é encerrado com a instrução

```
end program primeiro
```

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

[1] Os tipos de variáveis suportados pelo Fortran 95 são discutidos no capítulo 3.

2.2 Nomes em Fortran 90/95

Um programa em Fortran 90/95 faz referência a muitas entidades distintas, tais como programas, sub-programas, módulos, variáveis, etc. Os nomes destas entidades devem consistir em caracteres alfanuméricos, de 1 a 31 caracteres. Os caracteres alfanuméricos válidos são:

- Letras (a - z; A - Z).
- Numerais (0 - 9).
- O caractere *underscore* "_".

A única restrição é que o primeiro caractere seja uma letra. Os seguintes exemplos são válidos:

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

já estes exemplos **não** são válidos:

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

2.3 Entrada e saída padrões

O Fortran 90 possui três comandos de entrada/saída de dados. A maneira mais direta de definir valores de variáveis ou de exibir os valores destas é através dos dispositivos de entrada/saída padrões.

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

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

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

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

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

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

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

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

Esta seção apresentou somente um uso muito simples dos recursos de Entrada/Saída de dados. Uma descrição mais completa destes recursos será realizada no capítulo 9 na página 125.

Tabela 2.1: Caracteres especiais do Fortran 90/95

Caractere	Nome/Função	Caractere	Nome/Função
=	Igual		Espaço em branco
+	Soma	!	Exclamação
-	Subtração	“	Aspas
*	Multiplicação	%	Porcentagem
/	Divisão	&	E comercial (ampersand)
(	Parênteses esquerdo	;	Ponto e vírgula
)	Parênteses direito	?	Ponto de interrogação
,	Vírgula	**	Potência
.	Ponto decimal	’	Apóstrofe
$	Cifrão	<	Menor que
:	Dois pontos	>	Maior que

2.4 Conjunto de caracteres aceitos

No Fortran 90/95, os elementos básicos, denominados de *tokens*, são os caracteres alfanuméricos, os 10 dígitos arábicos, o *underscore* “_” e o conjunto de caracteres especiais apresentados na tabela 2.1. Dentro da sintaxe da linguagem, não existem diferenças entre letras maiúsculas e minúsculas.

Capítulo 3

Tipos de Variáveis

Em Fortran, há cinco tipos intrínsecos de variáveis: três tipos numéricos, *inteiros, reais* e *complexos* (em inglês `INTEGER, REAL` e `COMPLEX`) e dois tipos não numéricos, *caracteres* e *lógicos* (em inglês `CHARACTER` e `LOGICAL`). O primeiro grupo de variáveis é utilizado em operações matemáticas e é o mais utilizado. O segundo grupo é utilizado para operações que envolvem manipulações de texto ou operações lógicas.

Em Fortran 90/95, cada um dos cinco tipos intrínsecos possui um valor inteiro não negativo denominado *parâmetro de espécie do tipo* (em inglês, *kind type parameter*). A norma padrão da linguagem determina que qualquer processador deve suportar pelo menos duas espécies para os tipos `REAL` e `COMPLEX` e pelo menos uma espécie para os tipos `INTEGER, CHARACTER` e `LOGICAL`.

Um outro avanço do Fortran 90 sobre o predecessor Fortran 77 consiste na habilidade em definir novos tipos de variáveis, baseados nos tipos intrínsecos. Estas são os chamados *tipos derivados* ou *estruturas*, os quais consistem em combinações de partes compostas por tipos intrínsecos e/ou por outros tipos derivados, permitindo gerar objetos de complexidade crescente.

Neste capítulo, vamos inicialmente definir o uso básicos dos tipos intrínsecos de variáveis, seguindo pela definição das espécies de variáveis mais comuns, terminando por definir os tipos derivados.

3.1 Declaração de tipo de variável

Como já foi colocado anteriormente, é uma boa praxe de programação iniciar o setor de declarações de variáveis com a declaração

```
IMPLICIT NONE
```

a qual impede a possibilidade de haver nomes de variáveis não definidos, os quais possuem o seu tipo implícito, uma prática corriqueira em Fortran 77.

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

```
<tipo>[([KIND=]<parâmetro de espécie>)][,<lista de atributos>] ::  <lista de entidades>
```

onde `<tipo>` especifica o tipo de variável, `<parâmetro de espécie>` especifica a espécie da variável, `<lista de atributos>` é um dos seguintes:

```
PARAMETER       DIMENSION(<lista de extensões>)
PUBLIC          INTENT(<inout>)
PRIVATE         OPTIONAL
POINTER         SAVE
TARGET          EXTERNAL
ALLOCATABLE     INTRINSIC
```

e cada entidade é

```
<nome do objeto> [(lista de extensões)] [*char-len] [= expressão de inicialização]
```

ou

```
<nome da função> [*char-len]
```

onde `<nome do objeto>` é o nome da variável, seguindo a regra de nomes válidos definida no capítulo 2. Os objetos restantes, em particular `<parâmetro de espécie>` e `<lista de atributos>`, serão estudados ao longo do desenvolvimento desta apostila.

3.2 Variáveis do tipo INTEGER

Este tipo de variável armazena apenas a parte inteira de um número, exemplos de números inteiros válidos, também denominados *literais* são:

123, 89312, 5.

As declarações básicas de nomes de variáveis de tipo inteiro são:

Fortran 77: `INTEGER <lista de nomes de variáveis>`

Fortran 90/95: `INTEGER :: <lista de nomes de variáveis>`

O tipo de dados inteiro possui valores que pertencem ao conjunto dos números inteiros.

Programa exemplo:

```
program inteiro
implicit none
integer :: x
! O valor digitado não pode conter ponto (.) Caso isto
! aconteça, vai gerar um erro de execução no programa,
! abortando o mesmo.
read *, x
print*, "Valor lido:",x
end program inteiro
```

3.3 Variáveis do tipo REAL

O tipo de variável real é composto de quatro partes, assim dispostas:

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

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

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

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

As declarações básicas do tipo real são:

Fortran 77: `REAL <lista de nomes de variáveis>`

Fortran 90/95: `REAL :: <lista de nomes de variáveis>`

Programa exemplo:

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

3.4 Variáveis do tipo COMPLEX

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

`(<parte real>,<parte imaginária>)`

Exemplos de literais complexos são:

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

Uma outra grande vantagem do Fortran é que toda a álgebra de números complexos já está implementada a nível de compilador. Assim, se for realizado o produto de dois números complexos `(x1,y1)` e `(x2,y2)`, o resultado será o literal complexo dado por `(x1*x2 - y1*y2,x1*y2 + x2*y1)`. O mesmo acontecendo com as outras operações algébricas.

As declarações básicas do tipo complexo são:

Fortran 77: `COMPLEX <lista de nomes de variáveis>`

Fortran 90/95: `COMPLEX :: <lista de nomes de variáveis>`

Programa exemplo:

```
program var_complexa
implicit none
complex :: a= (5,-5),b,c ! Variável a é inicializada a (5,-5).
print *, "Valor de b:"
! O valor de b deve ser entrado como um literal complexo.
! Exemplo: (-1.5,2.5)
read *, b
c= a*b
print *, "O valor de c:", c
! Verifique o resultado no papel.
end program var_complexa
```

3.5 Variáveis do tipo CHARACTER

O tipo padrão consiste em um conjunto de caracteres contidos em um par de apóstrofes ou aspas. Os caracteres não estão restritos ao conjunto de caracteres padrão definidos na seção 2.4. Qualquer caractere que possa ser representado pelo processador é aceito, exceto os caracteres de controle tais como o *return*. Os apóstrofes ou aspas servem como delimitadores dos literais de caractere e **não** são considerados parte integrante do conjunto. Ao contrário das normas usuais, um espaço em branco é diferente de dois ou mais.

Exemplos de literais de caractere:

```
'bom Dia'
'bomDia'
'BRASIL'
"Fortran 90"
```

As declarações mais utilizadas para o tipo caractere são:

Fortran 77: `character*<comprimento> <lista de nomes de variáveis>`
ou
`character <nome 1>*<comp. 1>, [<nome 2>*<comp. 2>, ...]`

Fortran 90/95: `character(len=<comprimento>) :: <lista de nomes de variáveis>`

onde `<comprimento>` indica a quantidade de caracteres contidos nas variáveis. Todas as variáveis definidas por esta declaração têm o mesmo número de caracteres. Se for informado um literal maior que `<comprimento>`, este será truncado; se for informado um literal menor que o declarado, o processador irá preencher o espaço restante à direita com espaços em branco.

Programa exemplo:

```
program le_caractere
implicit none
character(len=10) :: str_read
print *, "Entre com texto":
read '(a)', str_read
print *, "Texto lido:", str_read
end program le_caractere
```

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

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

Como exemplo temos:

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

3.6 Variáveis do tipo LOGICAL

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

- `.TRUE.` ⇒ Verdadeiro

- `.FALSE.` ⇒Falso.

As declarações do tipo lógico são:

Fortran 77: `logical <lista de nomes de variáveis>`

Fortran 90/95: `logical :: <lista de nomes de variáveis>`

Programa exemplo:

```
program logico
implicit none
logical :: a= .true.
if (a) then
print*, "A variável é verdadeira."
end if
end program logico
```

3.7 O conceito de espécie *(kind)*

Em Fortran, além dos 5 tipos de variáveis definidos nas seções 3.2 a 3.6, é possível definir extensões a um determinado tipo, cujas declarações dependem da versão do Fortran utilizada e do processador no qual o programa será executado.

3.7.1 Fortran 77

Em Fortran 77 as extensões mais comuns e as correspondentes declarações são:

- variáveis reais de precisão dupla: `real*8 <lista de nomes de variáveis>` ou `double precision <lista de nomes de variáveis>`.
- variáveis reais de precisão estendida (ou quádrupla): `real*16 <lista de nomes de variáveis>`.
- variáveis complexas de precisão dupla: `complex*16 <lista de nomes de variáveis>`.

As diferenças entre estas extensões e os correspondentes tipos originais serão ilustradas a seguir.

3.7.2 Fortran 90/95

Em Fortran 90/95, cada um dos cinco tipos intrínsecos, `INTEGER, REAL, COMPLEX, CHARACTER` e `LOGICAL` possui associado um valor inteiro não negativo denominado *parâmetro de espécie do tipo (kind type parameter)*. Por exigência do padrão, um processador deve suportar, no mínimo, duas espécies para os tipos `REAL` e `COMPLEX` e uma espécie para os tipos `INTEGER, CHARACTER` e `LOGICAL.`

Os valores da espécie são dependentes do processador e/ou do compilador empregado. Contudo, há funções intrínsecas fornecidas pelo compilador que verificam as precisões suportadas pelo processador e que podem ser usadas para definir o valor do parâmetro `KIND`, possibilitando assim a portabilidade do código, isto é, a possibilidade deste rodar em diferentes arquiteturas usando uma precisão mínima especificada pelo programador.

Para demonstrar como diferentes compiladores implementam e usam o parâmetro de espécie, serão considerados os compiladores `Intel® Fortran Compiler for linux` (versão 9.1), `gfortran` e o compilador `F`, todos gratuítos.

3.7.2.1 Compilador `Intel® Fortran`

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

Tipo Inteiro. Há 04 parâmetros de espécie para o tipo inteiro.
Declaração: `INTEGER([KIND=]<n>) [::] <lista de nomes de variáveis>`
Sendo `<n>` das espécies 1, 2, 4 ou 8. Se o parâmetro de espécie é explicitado, as variáveis na `<lista de nomes de variáveis>` serão da espécie escolhida. Em caso contrário, a espécie será o *inteiro padrão:* `INTEGER(KIND=4)`; ou seja, a declaração `INTEGER :: <lista de nomes de variáveis>` equivale a `INTEGER(KIND=4) :: <lista de nomes de variáveis>`.

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

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

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

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

Tipo Real. Há 03 parâmetros de espécie para o tipo real.
Declaração: `REAL([KIND=]<n>) [::] <lista de nomes de variáveis>`
Sendo `<n>` igual a 4, 8 ou 16. Caso o parâmetro de espécie não seja especificado, a espécie será o *real padrão:* `REAL(KIND= 4)`.

Tipo Complexo. Há 03 parâmetros de espécie para o tipo complexo.
Declaração: `COMPLEX([KIND=]<n>) [::] <lista de nomes de variáveis>`
Sendo `<n>` igual a 4, 8 ou 16. Caso o parâmetro de espécie não seja explicitado, a espécie será o *complexo padrão:* `COMPLEX(KIND= 4)`.

Tipo Lógico. Há 04 parâmetros de espécie para o tipo lógico.
Declaração: `LOGICAL([KIND=]<n>) [::] <lista de nomes de variáveis>`
Sendo `<n>` igual a 1, 2, 4 ou 8.

Tipo Caractere. Há somente uma espécie do tipo caractere.
Declaração: `CHARACTER([KIND=1],[LEN=]<comprimento>) [::] <lista de nomes de variáveis>`

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

O programa 3.1 a seguir ilustra do uso e as diferenças de algumas opções de espécies de tipos de variáveis. O mesmo programa também indica o uso de alguns atributos na declaração de variáveis, tais como na declaração

```
INTEGER, PARAMETER ::  DP= 2
```

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

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

O atributo `PARAMETER` indica que as variáveis declaradas nesta sentença devem se comportar como constantes matemáticas, isto é, não podem ser alteradas no programa por nenhuma atribuição de variáveis (ver capítulo 4). Na mesma declaração, já está sendo inicializado o valor do parâmetro `DP`, sendo este igual a 2.

O mesmo exemplo também ilustra o uso da função implícita `SQRT(X)`:

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

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

3.7.2.2 Compilador `gfortran`

`Gfortran` é o compilador Fortran 95 da GNU (Fundação *Gnu is Not Unix.*), inicialmente desenvolvido como alternativa ao compilador `f95` distribuído pelas versões comerciais do Unix. Atualmente, o `gfortran` é parte integrante da plataforma de desenvolvimento de software `GCC` (`GNU Compiler Collection`), que é composta por compiladores de diversas linguagens distintas, tais como Fortran 95, C/C++, Java, Ada, entre outros. O comando `gfortran` consiste simplesmente em um *script* que invoca o programa `f951`, o qual traduz o código-fonte para assembler, invocando em seguida o linkador e as bibliotecas comuns do pacote `GCC`.

No `gfortran`, os parâmetros de espécie são determinados de forma semelhante ao compilador `Intel®Fortran`, discutido na seção 3.7.2.1, ou seja, o parâmetro indica o número de bytes necessários para armazenar cada variável da respectiva espécie de tipo. As espécies suportadas pelo `gfortran` são descritas na tabela 3.2.

O programa `testa_kind_gfortran` a seguir (programa 3.2) ilustra o uso e as diferenças entre as diversas espécies de tipos de dados no compilador `gfortran`.

3.7.2.3 Compilador `F`

Como exemplo do uso do parâmetro de espécie, a tabela 3.3 ilustra os valores suportados pelo compilador F, conforme fornecidos pelo guia do usuário[2]. Há duas possibilidades para os números da espécie:

1. o modo padrão de operação, também denominado seqüencial, o qual pode, porém, ser especificado explicitamente no momento da compilação com a chave `-kind=sequential`;

2. o esquema de numeração bytes, o qual deve ser especificado no momento da compilação com a chave `-kind=byte`:
   `alunos|fulano>F -kind=byte <nome programa>.f90 -o <nome programa>`

O exemplo a seguir, programa `testa_kind_F`, ilustra do uso e as diferenças de algumas opções de espécies de tipos de variáveis.

```
program testa_kind_F
implicit none
integer, parameter :: dp= 2
real :: r_simples
real(kind= dp) :: r_dupla
complex :: c_simples
complex(kind= dp) :: c_dupla
```

[1] Em plataformas de 64 bits, tais como a família de processadores Intel® Core™ 2.
[2] The F Compiler and Tools (`http://www.fortran.com/imagine1/ftools.pdf`).

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

```
program testa_kind_gfortran
implicit none
integer, parameter :: i1= 1, i2= 2, i4= 4, i8= 8, i16= 16
integer, parameter :: dp= 8, qp= 10
integer(kind= i1) :: vi1
integer(kind= i2) :: vi2
integer(kind= i4) :: vi4
integer(kind= i8) :: vi8
integer(kind= i16) :: vi16
real :: r_simples
real(kind= dp) :: r_dupla
real(kind= qp) :: r_quad
complex :: c_simples
complex(kind= dp) :: c_dupla
complex(kind= qp) :: c_quad
!
!Mostra maiores numeros representaveis do tipo inteiro.
vi1= huge(1_i1)
vi2= huge(1_i2)
vi4= huge(1_i4)
vi8= huge(1_i8)
vi16= huge(1_i16)
print*, 'Especies Inteiras:'
print*, vi1, vi2, vi4
print*, vi8
print*, vi16
!Mostra maiores numeros representaveis do tipo real.
r_simples= huge(1.0)   ! Precisao simples
r_dupla= huge(1.0_dp) ! Precisao dupla
r_quad= huge(1.0_qp)   ! Precisao estendida
!
!Calcula a raiz quadrada de (2,2) em diversas precisoes.
c_simples= sqrt((2.0,2.0))
c_dupla=   sqrt((2.0_dp,2.0_dp))
c_quad=    sqrt((2.0_qp,2.0_qp))
!
print*,
print*, 'Especies Reais e Complexas:'
print *, r_simples
print *, r_dupla
print *, r_quad
print*, c_simples
print*, c_dupla
print*, c_quad
!
end program testa_kind_gfortran
```

Tabela 3.3: Valores de espécies *(kind)* de tipos de variáveis suportados pelo compilador F.

Tipo	Número do *Kind* (Seqüencial)	Número do *Kind* (Byte)	Descrição
`Real`	1	4	Real precisão simples (padrão)
`Real`	2	8	Real precisão dupla
`Real`[3]	3	16	Real precisão quádrupla
`Complex`	1	4	Complexo precisão simples (padrão)
`Complex`	2	8	Complexo precisão dupla
`Complex`	3	16	Complexo precisão quádrupla
`Logical`	1	1	Lógico 1 byte
`Logical`	2	2	Lógico 2 bytes
`Logical`	3	4	Lógico 4 bytes (padrão)
`Logical`	4	8	Lógico 8 bytes
`Integer`	1	1	Inteiro 1 byte
`Integer`	2	2	Inteiro 2 bytes
`Integer`	3	4	Inteiro 4 bytes (padrão)
`Integer`	4	8	Inteiro 8 bytes
`Character`	1	1	Caractere, 1 byte por caractere

```
!
!Calcula a raiz quadrada de 2 em diversas precises.
r_simples= sqrt(2.0)    ! Preciso simples
r_dupla= sqrt(2.0)      ! Preciso dupla
!
!Nmeros complexos: parte real: raiz de 2.  Parte imaginria: raiz de 3.
c_simples= cmplx(sqrt(2.0),sqrt(3.0))
c_dupla= cmplx(sqrt(2.0),sqrt(3.0))
!
!Imprime resultados na tela.
print *, r_simples
print *, r_dupla
print *, c_simples
print *, c_dupla
!
end program testa_kind_F
```

3.7.2.4 Literais de diferentes espécies

Para distingüir as espécies dos literais (ou constantes) dentre diferentes números fornecidos ao compilador, utiliza-se o sufixo `_<k>`, sendo `<k>` o parâmetro da espécie do tipo:

Literais inteiros. Constante inteiras, incluindo a espécie, são especificadas por:
`[<s>]<nnn...>[_<k>]`
onde: `<s>` é um sinal (+ ou −); obrigatório se negativo, opcional se positivo. `<nnn...>` é um conjunto de dígitos (0 a 9); quaisquer zeros no início são ignorados. `_<k>` é um dos parâmetros de espécie do tipo: 1, 2, 4 ou 8; esta opção explicita a espécie do tipo à qual o literal pertence.

Literais reais. Constantes reais são especificadas de diferentes maneiras, dependendo se possuem ou não parte exponencial. A regra básica para a especificação de um literal real já foi definida na seção 3.3. Para explicitar a espécie do tipo real à qual o literal pertence, deve-se incluir o sufixo `_<k>`, onde `<k>` é um dos parâmetros de espécie do tipo: 4, 8 ou 16. Por exemplo:
`2.0_8`: para indicar tipo real, espécie 8.

Literais complexos. A regra básica para a especificação de um literal complexo já foi definida na seção 3.4. Para explicitar a espécie do tipo à qual o literal pertence, deve-se incluir o sufixo `_<k>`, onde `<k>` é um dos parâmetros de espécie do tipo: 4, 8 ou 16, em cada uma das partes real e imaginária do

[3] Variáveis reais e complexas de precisão quádrupla não são suportadas por todas as versões do compilador F.

literal complexo. Por exemplo:
`(1.0_8,3.5345_8)`: para indicar tipo complexo, espécie 8.

Literais lógicos. Uma constante lógica pode tomar uma das seguintes formas:
`.TRUE.[_<k>]`
`.FALSE.[_<k>]`
onde `<k>` é um dos parâmetros de espécie do tipo: 1, 2, 4 ou 8.

3.7.3 Funções intrínsecas associadas à espécie

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

As funções intrínsecas descritas nesta e nas subsequentes seções serão novamente abordadas no capítulo 7, onde serão apresentadas todas as rotinas intrínsecas fornecidas pelo padrão da linguagem Fortran 95.

3.7.3.1 `KIND(X)`

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

```
program tes_fun_kind
implicit none
integer :: i,j
integer, parameter :: dp= 2
real :: y
real(kind= dp) :: x
!
i= kind(x) ! i= 2
j= kind(y) ! Depende do sistema (j=1 para compilador F).
print*, i
print*, j
end program tes_fun_kind
```

Outros exemplos:

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

3.7.3.2 `SELECTED_REAL_KIND(P,R)`

A função intrínseca `SELECTED_REAL_KIND(P,R)` tem dois argumentos opcionais: `P` e `R`. A variável `P` especifica a precisão (número de dígitos decimais) mínima requerida e `R` especifica o intervalo de variação mínimo da parte exponencial da variável.

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

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

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

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

```
A= 2.0_I10
```

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

```
D= 1.0E201_I10
```

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

3.7.3.3 `SELECTED_INT_KIND(R)`

A função intrínseca `SELECTED_INT_KIND(R)` é usada de maneira similar à função `SELECTED_REAL_KIND`. Agora, a função tem um único argumento `R`, o qual especifica o intervalo de números inteiros requerido. Assim, `SELECTED_INT_KIND(r)` retorna o valor da espécie que representa, no mínimo, valores inteiros no intervalo -10^r a $+10^r$. Se mais de uma espécie satisfaz o requerimento, o valor retornado é aquele com o menor intervalo no expoente r. Se o intervalo não for disponível, o valor -1 é retornado.

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

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

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

3.8 Tipos derivados

Uma das maiores vantagens do Fortran 90/95 sobre seus antecessores está na disponibilidade ao programador de definir seus próprios tipos de variáveis. Estas são os chamados *tipos derivados*, também freqüentemente denominados de *estruturas*.

A forma geral da declaração de um tipo derivado é:

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

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

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

onde aqui o <*tipo*> pode ser um tipo de variável intrínseca ou outro tipo derivado. A declaração de uma lista de variáveis do tipo derivado <*nome do tipo*> é feita através da linha:

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

Para tentar-se entender o uso de uma variável de tipo derivado, vamos definir um novo tipo: `PONTO`, o qual será construído a partir de três valores reais, representando os valores das coordenadas x, y e z do ponto no espaço cartesiano:

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

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

Como foi mostrado na atribuição `APICE%X= 0.0`, cada componente da variável de tipo derivado pode ser referenciada individualmente por meio do *caractere seletor de componente,* "`%`". Já a variável `CENTRO` teve os valores de suas componentes definidos pelo *construtor de estrutura*:

```
CENTRO= PONTO(0.0,0.0,0.0)
```

ou seja, `CENTRO%X= 0.0`, `CENTRO%Y= 0.0` e `CENTRO%Z= 0.0`.

Estruturas definidas desta forma podem ser interessantes quando o programador quer classificar determinados objetos caracterizados por parâmetros e/ou qualificadores representados por variáveis de diferentes tipos. É possível construirem-se estruturas progressivamente mais complicadas definindo-se novos tipos derivados que englobam aqueles previamente definidos. Por exemplo,

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

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

da DIAGONAL_PRINCIPAL. Note, por outro lado, o uso do construtor de estrutura para definir a posição do ponto P2, como componente da diagonal principal.

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

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

Em Fortran 95, tornou-se possível inicializar os valores dos componentes dos tipos derivados. O valor deve ser especificado quando o componente é declarado como parte da definição do tipo. Se o componente não for um ponteiro, a inicialização é feita de forma usual, com um sinal de igual seguido da expressão de inicialização. Se o componente for um ponteiro, a única inicialização admitida é o símbolo de atribuição de um ponteiro (=>), seguida pela função intrínseca NULL() (seção 7.16). Em ambos os casos os caracteres :: são exigidos.

Uma inicialização não deve necessariamente se aplicar a todos os componentes de um tipo derivado. Um exemplo válido seria:

```
TYPE :: ENTRY
   REAL                  :: VALOR= 2.0
   INTEGER                  INDEX
   TYPE(ENTRY), POINTER :: NEXT => NULL()
END TYPE ENTRY
```

Dada uma declaração de matriz tal como

```
TYPE(ENTRY), DIMENSION(100) :: MATRIZ
```

os sub-objetos tais como MATRIZ(3)%VALOR terão automaticamente o valor 2.0 e a seguinte operação, que usa uma função intrínseca ASSOCIATED(MATRIX(3)%NEXT) vai retornar o valor .FALSE. (ver seção 7.3).

Se as declarações de tipos derivados estiverem aninhadas, as inicializações associadas a componentes são reconhecidas em todos os níveis. Assim,

```
TYPE :: NODO
   INTEGER     CONTA
   TYPE(ENTRY) ELEMENTO
END TYPE NODO
TYPE(NODO) N
```

ocasiona que o componente N%ELEMENTO%VALOR terá automaticamente o valor 2.0.

Os componentes do tipo derivado continuam podendo ser inicializados em declarações de variáveis do tipo, tais como

```
TYPE(ENTRY), DIMENSION(100) :: MATRIZ= ENTRY(HUGE(0.0), HUGE(0), NULL())
```

em cuja situação a inicialização inicial é ignorada.

Capítulo 4

Expressões e Atribuições Escalares

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

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

4.1 Regras básicas para expressões escalares

Uma expressão em Fortran 90/95 é formada de *operandos* e *operadores*, combinados de tal forma que estes seguem as regras de sintaxe. Os operandos podem ser constantes, variáveis ou funções e uma expressão, por si mesma, também pode ser usada como operando. A sentença

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

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

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

```
operador operando
```

um exemplo sendo

```
-Y
```

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

```
operando operador operando
```

um exemplo sendo

```
X + Y
```

Uma expressão mais complicada seria

```
operando operador operando operador operando
```

onde operandos consecutivos são separados por um único operador.

Cada operando deve ter valor definido e o resultados da expressão deve ser matematicamente definido; por exemplo, divisão por zero não é permitido, gerando o que se denomina uma *exceção de ponto flutuante.*[1]

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

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

[1] Do inglês *floating point exception.*

a subexpressão entre parênteses será desenvolvida primeiro e o resultado usado como um operando para o primeiro operador, quando então a expressão completa será desenvolvida usando a norma padrão, ou seja, da esquerda para a direita.

Se uma expressão ou subexpressão não é contida entre parênteses, é permitido ao processador desenvolver uma expressão equivalente, a qual é uma expressão que resultará no mesmo valor, exceto por erros de arredondamento numérico. As duas operações a seguir são, em princípio, equivalentes:

`A/B/C` ou `A/(B*C)`

Possíveis diferenças nos resultados das operações acima consistem em diferentes erros de arredondamento e/ou tempo de processamento, caso o processador consiga multiplicar mais rapidamente que dividir.

Se dois operadores seguem-se imediatamente, como em

```
operando operador operador operando
```

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

4.2 Expressões numéricas escalares

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

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

Na ausência de parênteses, ou dentro destes, no caso de subexpressões, a operação com a maior precedência é o cálculo das funções, seguidos das exponenciações que serão realizadas antes de multiplicações ou divisões e estas, por sua vez, antes de adições ou subtrações.

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

- Os operadores de subtração "`-`" e de adição "`+`" podem ser usados como operadores unitários, como em `-VELOCIDADE`
- Uma vez que não é permitido na notação matemática ordinária, uma subtração ou adição unitária não pode seguir imediatamente após outro operador. Quando isto é necessário, deve-se fazer uso de parênteses. Por exemplo, as operações matemáticas a seguir:
  x^{-y} deve ser digitada: `X**(-Y);`
  $x(-y)$ deve ser digitada : `X*(-Y).`
  Como já foi mencionado na seção 4.1, um operador binário também não pode seguir outro operador.
- Os parênteses devem indicar os agrupamentos, como se escreve em uma expressão matemática. Os parênteses indicam que as operações dentro deles devem ser executadas prioritariamente:
  $(a+b)\left[(x+y)^2+w^2\right]^3$ deve ser digitada: `(A+B)*((X+Y)**2 + W**2)**3` ou `(A+B)*(((X+Y)**2 + W**2)**3)`
- Os parênteses não podem ser usados para indicar multiplicação, sendo sempre necessário o uso do operador "`*`". Qualquer expressão pode ser envolvida por parênteses exteriores, que não a afetam:
  `X + Y` é equivalente a `((X) + Y)` ou equivalente a `((X + Y))`
  Contudo, o número de parênteses à esquerda deve sempre ser igual ao número de parênteses à direita.

Tabela 4.1: Operadores numéricos escalares

Operador	Operação
**	Potenciação
*	Multiplicação
/	Divisão
+	Adição
-	Subtração

- Nenhum operador pode ser deixado implícito:
  `5*T` ou `T*5`: correto
  `5T, 5(T)` ou `T5`: incorreto.

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

- A exceção à regra esquerda-para-direita para operadores de igual precedência ocorre apenas para a potenciação "`**`". A expressão
  `A**B**C`
  é válida e é desenvolvida da direita para a esquerda como
  `A**(B**C).`

- Para dados inteiros, o resultado de qualquer divisão será truncado para zero, isto é, para o valor inteiro cuja magnitude é igual ou logo inferior à magnitude do valor exato. Assim, o resultado de:
  6/3 é 2
  8/3 é 2
  -8/3 é -2.
  Este fato deve sempre ser levado em conta quando operações com inteiros estão sendo realizadas. Por isto, o valor de `2**3` é `8`, enquanto que o valor de `2**(-3)` é `0`.

- A regra padrão do Fortran 90 também permite que uma expressão numérica contenha operandos numéricos de diferentes tipos ou espécies. Esta é uma *expressão de modo misto*. Exceto quando elevando um valor real ou complexo a uma potência inteira, o objeto do tipo mais fraco (ou mais simples) de variável dos dois tipos envolvidos em uma expressão será convertido, ou *coagido*, para o tipo mais forte. O resultado será também do tipo mais forte. Por exemplo, se `A` é real e `I` é inteiro, a expressão `A*I` tem, inicialmente, `I` sendo convertido a real antes que a multiplicação seja efetuada e o resultado da mesma é do tipo real. As tabelas 4.2 e 4.3 ilustram os resultados de diferentes operações numéricas escalares.

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

4.3 Atribuições numéricas escalares

A forma geral de uma atribuição numérica escalar é

```
<nome variável> = <expressão>
```

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

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

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

são, respectivamente, `7` e `4.01935`.

4.4 Operadores relacionais

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

Tabela 4.2: Tipos de resultados de `A .op. B`, onde `.op.` é `+`, `-`, `*` ou /. As funções intrínsecas `REAL()` e `CMPLX()` são discutidas na seção 7.4.1

Tipo de A	Tipo de B	Valor de A usado na .op.	Valor de B usado na .op.	Tipo de resultado
I	I	A	B	I
I	R	REAL(A,KIND(B))	B	R
I	C	CMPLX(A,0,KIND(B))	B	C
R	I	A	REAL(B,KIND(A))	R
R	R	A	B	R
R	C	CMPLX(A,0,KIND(B))	B	C
C	I	A	CMPLX(B,0,KIND(A))	C
C	R	A	CMPLX(B,0,KIND(A))	C
C	C	A	B	C

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

Tipo de A	Tipo de B	Valor de A usado na .op.	Valor de B usado na .op.	Tipo de resultado
I	I	A	B	I
I	R	REAL(A,KIND(B))	B	R
I	C	CMPLX(A,0,KIND(B))	B	C
R	I	A	B	R
R	R	A	B	R
R	C	CMPLX(A,0,KIND(B))	B	C
C	I	A	B	C
C	R	A	CMPLX(B,0,KIND(A))	C
C	C	A	B	C

Fortran 77: somente a forma apresentada na coluna 2 é válida.

Fortran 90/95: qualquer uma das duas formas é válida; porém, a forma apresentada na coluna 3 é a mais moderna e recomendável, sendo a outra considerada obsoleta.

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

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

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

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

Para comparações de caracteres, as espécies devem ser as mesmas e as as letras (ou números ou caracteres especiais) são comparados da esquerda para a direita até que uma diferença seja encontrada ou ambos os

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

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

Tabela 4.5: Operadores relacionais em Fortran 90/95

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

caracteres sejam idênticos. Se os comprimentos diferem, a variável mais curta é suposta preenchida por brancos à direita.

4.5 Expressões e atribuições lógicas escalares

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

Operador unitário:
`.NOT.` (negação lógica)
Operadores binários:
`.AND.` (intersecção lógica, E lógico)
`.OR.` (união lógica, OU lógico)
`.EQV.` (equivalência lógica)
`.NEQV.` (não-equivalência lógica).

Dada então uma declaração de variáveis lógicas do tipo

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

as seguintes expressões lógicas são válidas:

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

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

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

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

O resultado de uma expressão lógica envolvendo duas variáveis lógicas `A` e `B`, por exemplo, pode ser inferido facilmente através da consulta às Tabelas-Verdade 4.6 – 4.8.

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

Tabela 4.6: Tabela-Verdade `.NOT.`

A	.NOT. A
T	F
F	T

Tabela 4.7: Tabela-Verdade `.AND.`

A	B	A .AND. B
T	T	T
T	F	F
F	T	F
F	F	F

Tabela 4.8: Tabela-Verdade `.OR.`

A	B	A .OR. B
T	T	T
T	F	T
F	T	T
F	F	F

```
FLAG= .TRUE.
```

ou no corpo de declarações de variáveis:

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

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

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

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

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

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

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

4.6 Expressões e atribuições de caracteres escalares

O único operador intrínseco para expressões de caracteres é o *operador de concatenação* "`//`", o qual tem o efeito de combinar dois operandos de caracteres em um único caractere resultante, de extensão igual à soma das extensões dos operandos originais. Por exemplo, o resultado da concatenação das constantes de caractere `'AB'` e `'CD'`, escrita como

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

é a constante `'ABCD'`.

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

Substrings de caracteres.

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

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

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

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

onde I e J são variáveis inteiras, as quais localizam explicitamente os caracteres de I a J em HINO. Os dois pontos ":" são usados para separar os dois índices da substring e são sempre obrigatórios, mesmo que se queira isolar somente um caractere da variável. Alguns exemplos de substrings da variável HINO são:

```
HINO(8:13)          !Correspondente a 'aurora'
HINO(60:79)         !Correspondente a 'Foi o 20 de setembro'
HINO(140:140)       !Correspondente a 'â'
```

As constantes de caracteres resultantes das substrings podem ser então atribuídas a outras variáveis de caracteres. Há valores padrão para os índices das substrings. Se o índice inferior é omitido, o valor 1 é assumido; se o valor superior é omitido, um valor correspondente ao comprimento da variável é assumido. Assim,

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

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

Pode-se também fazer concatenações com substrings:

`HINO(8:13)//HINO(69:79)` gera a constante `'aurora de setembro'`

`TROCADILHO= HINO(153:157)//'s'//HINO(191:199)` atribui à `TROCADILHO` o valor `'ímpias façanhas'`.

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

```
CHARACTER(LEN= 10) :: PARTE1, PARTE2
PARTE1= HINO(178:183)    ! Resulta em PARTE1= 'Sirvam   '
```

Ao passo que se o resultado da expressão foi maior que o tamanho da variável, esta será preenchida completamente e o restante do valor será truncado:

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

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

```
HINO(50:113)= 'AAAAAAAAAAAARRRRRRRRRRR-
RGGGGGGGGGGGGGHHHHHHHHHHHH!!!!!$%#$%%&%#%$#'
```

de tal forma que resulta,

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

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

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

resulta em `PARTE2= 'ComoComaur'`.

Comparações e uso de operadores relacionais com variáveis/constantes de caracteres são possíveis entre caracteres únicos, inclusive com os operadores > e <. Neste caso, não se trata de testar qual caractere é *"maior"* ou *"menor"* que outro. Em um sistema computacional, caracteres possuem uma propriedade denominada *sequência de intercalação*,[2] a qual ordena o armazenamento destes caracteres pelo sistema.

O Fortran 90/95 determina que a sequência de intercalação para qualquer arquitetura deve satisfazer as seguintes condições:

- A precede (é menor) que B, que precede C ... precede Y, que precede Z.
- 0 precede 1, que precede 2 ... precede 8, que precede 9.

[2] Do inglês *collating sequence.*

- Espaço em branco precede `A` e `Z` precede `0`; ou branco precede `0` e `9` precede `A`.

Se letras minúsculas são disponíveis, então

- `a` precede `b`, que precede `c`, ... precede `y`, que precede `z`.
- Espaço em branco precede `a` e `z` precede `0`; ou branco precede `0` e `9` precede `a`.

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

De acordo com estas regras, as expressões relacionais

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

fornecem ambas o resultado `T` *(true).*

Fortran 90/95 fornece acesso também à sequência de intercalação original da tabela ASCII[3] através de funções intrínsecas (seção 7.6.1). O ASCII consiste em uma padronização para um encodeamento de caracteres[4] de 7 bits (originalmente), baseado no alfabeto inglês. Códigos ASCII representam textos em computadores, equipamentos de comunicação e outros dispositivos eletrônicos que trabalham com texto. A tabela 4.9 mostra os caracteres ASCII originais, juntamente com os seus identificadores nos sistemas numéricos decimal, hexadecimal e octal, bem como o código html correspondente. Os primeiros 32 (0 – 31) caracteres mais o caractere 127 são caracteres de controle, destinados à comunicação com periféricos; estes não geram caracteres que podem ser impressos. Já os caracteres restantes (32 – 126) compõe-se de caracteres alfanuméricos mais caracteres especiais.

[3] *American Standard Code for Information Interchange* (ASCII).
[4] *Character encoding.*

Tabela 4.9: Tabela de códigos ASCII de 7 bits.

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

Capítulo 5

Comandos e Construtos de Controle de Fluxo

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

Para estes propósitos, recursos são disponíveis em Fortran que possibilitam o controle do fluxo lógico através dos comandos no programa. Os recursos contidos em Fortran 90 correspondem aos que agora são geralmente reconhecidos como os mais apropriados para uma linguagem de programação moderna. Sua forma geral é a de um *construto de bloco* (do inglês *block construct*), o qual é um construto (uma construção ou estrutura) que começa com uma palavra-chave inicial, pode ter palavras-chave intermediárias e que termina com uma palavra-chave final que identifica a palavra inicial. Cada seqüência de comandos entre as palavras-chave é chamada de um *bloco*. Um bloco pode ser vazio, embora tais casos sejam raros.

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

Os primeiros comandos de controle de fluxo a ser mencionados serão aqueles que foram definidos tanto no Fortran 77 quanto no Fortran 90/95, seguidos daqueles que somente existem no Fortran 90/95.

5.1 Comandos obsoletos do Fortran 77

Os comandos descritos nesta seção fazem parte do padrão do Fortran 77 mas que são considerados obsoletos no Fortran 90/95. O seu uso em um programa escrito nas últimas versões da linguagem é fortemente desencorajado e, para alguns compiladores, proibido. A razão para este status de obsoleto é porque estes comandos facilmente geram dificuldades e dão margem a confusão durante a leitura do programa. Algumas razões para o status de obsoleto para estes comandos são apresentadas abaixo.

5.1.1 Rótulos (*statement labels)*

Um *rótulo* consiste em um número inteiro, de 1 a 99999, inserido entre as colunas 1 e 5 de um programa ou sub-programa em Fortran 77 escrito, portanto, no formato fixo. Este rótulo enfatiza o ponto do código onde ele se encontra. Há três razões para se usar um rótulo:

1. O final de um laço `DO` é especificado por um rótulo determinado no início do mesmo laço.
2. Todo comando `FORMAT` deve possuir um rótulo, pois esta é a maneira como os comandos `READ` e `WRITE` fazem referência ao mesmo.
3. Qualquer comando executável pode possuir um rótulo afixado, de tal forma que o fluxo do código pode ser transferido ao mesmo. Isto é realizado, por exemplo, pelo comando `GO TO`.

Exemplo:

```
c (F77) Lê números de um arquivo de entrada até que ele termine, então os soma.
      SUMA= 0.0
100   READ(UNIT= IN, FMT= 200, END= 9999) VALOR
200   FORMAT(F20.0)
      SOMA= SOMA + VALOR
      GO TO 100
9999  CONTINUE
      WRITE(UNIT= *, FMT=*)'SOMA dos valores é:', SOMA
```

O comando `CONTINUE` significa, literalmente, "continue", isto é, ele apenas instrui o programa a continuar a partir daquele ponto.

5.1.2 Comando `GO TO` incondicional

O comando `GO TO` incondicional simplesmente transfere o fluxo do código para um comando rotulado em algum outro ponto do programa. Sua forma geral é:

```
GO TO <rótulo>
```

Um exemplo de aplicação deste comando pode ser visto acima. A única exceção para seu uso é a proibição de transferir o fluxo *para dentro* de um bloco `IF` ou laço `DO`; entretanto, é permitido transferir o fluxo com o comando `GO TO`, ou outro, *para fora* de um bloco `IF` ou laço `DO`.

O uso do `GO TO` incondicional torna possível escrever-se programas com uma estrutura bastante indisciplinada. Tais programas são usualmente difíceis de ser compreendidos e corrigidos. Bons programadores usam este comando raras vezes ou nunca. Contudo, no Fortran 77 em algumas situações não era possível evitá-lo, devido a falta de alternativas de estruturas de controle.

5.1.3 Comando `GO TO` computado

O comando `GO TO` computado é uma alternativa para um bloco `IF` para as situações em que um número grande de opções devem ser consideradas e elas podem ser selecionadas por uma expressão de valores inteiros. A forma geral do comando é:

```
GO TO (<rótulo 1>, <rótulo 2>, ..., <rótulo N>)[,] <expressão inteira>
```

A `<expressão inteira>` é desenvolvida; se o seu resultado é 1 (um), o controle é transferido ao comando afixado ao `<rótulo 1>`; se o resultado é 2 (dois), o controle é transferido ao comando afixado ao `<rótulo 2>` e assim por diante. Se o resultado da expressão é menor que um ou maior que N (havendo N rótulos na lista), o comando não tem efeito e o fluxo do programa continua com o próximo comando em seqüência. O mesmo rótulo pode estar presente mais de uma vez na lista.

O `GO TO` computado sofre das mesmas desvantagens que o `GO TO` incondicional, uma vez que se suas ramificações forem usadas sem cuidado, o programa se torna ilegível de tão confuso.

5.1.4 Comando `IF` aritmético

A forma geral do comando `IF` aritmético é:

```
IF (<expressão aritmética>) <rótulo 1>,<rótulo 2>,<rótulo 3>
```

Este comando geralmente fornece uma ramificação com três possibilidades, embora dois dos rótulos possam ser iguais, o que o torna uma ramificação de duas possibilidades. A `<expressão aritmética>` pode ser inteira, real ou real de precisão dupla. O controle do fluxo é transferido ao comando afixado ao `<rótulo 1>` se o valor for negativo, ao `<rótulo 2>` se for zero e ao `<rótulo 3>` se for positivo.

5.1.5 Comandos `ASSIGN` e `GO TO` atribuído

Estes dois comandos são normalmente usados juntos. O comando `ASSIGN` atribui o valor de um rótulo a uma variável inteira. Quando isto é realizado, a variável não mais possui um valor aritmético, mas se torna o próprio rótulo. Se o rótulo é afixado a um comando executável, a variável somente pode ser usada em um comando `GO TO` atribuído; se é afixado a um comando `FORMAT`, a variável somente pode ser usada em um comando `READ` ou `WRITE`. As formas gerais destes comandos são:

```
ASSIGN <rótulo> TO <variável inteira>
GO TO <variável inteira> [,] [(<rótulo>,<rótulo>,...,<rótulo>)]
```

Um `GO TO` atribuído pode ser usado para fornecer uma ligação para e a partir de uma seção de um programa ou sub-programa, atuando assim como um subrotina interna.

5.1.6 Laços `DO` rotulados

O comando `DO` controla um bloco de comandos os quais são executados repetidas vezes, uma vez para cada valor de uma variável denominada *variável de controle do laço (loop-control variable)*. O número de iterações depende dos parâmetros do comando `DO` no cabeçalho do laço.

A primeira versão de um bloco ou laço `DO` contém um rótulo que indica o último comando do laço. A forma geral do cabeçalho de um laço `DO` rotulado pode ter uma das duas formas seguintes:

```
DO <rótulo> [,] <variável> = <início>, <limite>, <passo>
DO <rótulo> [,] <variável> = <início>, <limite>
```

Na segunda forma, o tamanho do passo é implicitamente igual a um. O `<rótulo>` marca o último comando do laço. Ele deve estar afixado a um comando executável em alguma linha posterior do programa ou sub-programa. A regra permite que este comando seja qualquer comando executável, exceto outro comando de controle (como o `IF`, por exemplo), mas é recomendável que se use o comando `CONTINUE`, cuja função foi exemplificada na seção 5.1.1.

A `<variável>` é a variável de controle do laço ou índice do laço; ela deve ser uma variável escalar (não um elemento de matriz) e pode ser dos tipos inteiro, real ou dupla precisão.

Os valores de `<início>, <limite>` e `<passo>` podem ser expressões também dos tipos inteiro, real ou precisão dupla. Se o valor do `<passo>` estiver presente, este não pode ser zero; se for omitido, o seu valor é assumido igual a um. O número de iterações a ser realizadas é calculado antes do início da primeira iteração e é dado pela fórmula:

```
iterações = MAX(0,INT((limite - início + passo)/passo))
```

onde a função implícita `INT` toma a parte inteira do argumento por truncagem e a função `MAX` toma o maior valor dos seus dois argumentos. Nota-se que se o valor de `limite` é menor que `início`, o número de iterações é zero, exceto no case de `passo` ser negativo. Um número nulo de iterações é permitido, mas isto significa que o conteúdo do laço não será executado e o controle do fluxo é transferido ao primeiro comando após o `rótulo`. A variável de controle do laço não necessariamente assume o valor `limite`, especialmente se o tamanho do `passo` for maior que um e/ou do tipo real com valor fracionário.

Comandos dentro do laço podem alterar o valor das expressões usadas para o `início, limite` ou `passo`, mas o número de iterações não é alterado, uma vez que este é determinado antes do início da primeira iteração. A variável de controle do laço pode ser usada em expressões mas um novo valor não pode ser atribuído a ela dentro do laço.

Dois exemplos de aplicações de laços `DO` rotulados:

```
c Soma dos quadrados dos N primeiros elementos da matriz X.
      SOMA= 0.0
      DO 15, I= 1,N
         SOMA= SOMA + X(I)**2
15    CONTINUE
-------------------------------
C Inicializa as componentes da matriz FIELD iguais a zero.
c Exemplo de laços DO encadeados.
      REAL FIELD(NX, NY)
      DO 50, IY= 1,NY
         DO 40, IX= 1,NX
            FIELD(IX, IY)= 0.0
40       CONTINUE
50    CONTINUE
```

5.2 Comando e construto `IF`

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

5.2.1 Comando IF

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

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

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

```
IF (FLAG) GO TO 6
IF(X-Y > 0.0) X= 0.0
IF(COND .OR. P < Q .AND. R <= 1.0) S(I,J)= T(J,I)
```

5.2.2 Construto IF

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

```
[<nome>:] IF (<expressão relacional e/ou lógica>) THEN
   <bloco>
END IF [<nome>]
```

onde <bloco> denota uma seqüência de linhas de comandos executáveis. O bloco é executado somente se o resultado da <expressão relacional e/ou lógica> for verdadeiro. O construto IF pode ter um <nome>, o qual deve ser um nome válido em Fortran 90/95. O <nome> é opcional, mas se for definido no cabeçalho do construto, ele deve ser também empregado no final, denotado pela declaração END IF <nome>. Como exemplo, temos:

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

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

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

```
[<nome>:] IF (<expressão relacional e/ou lógica>) THEN
   <bloco 1>
ELSE [<nome>]
   <bloco 2>
END IF [<nome>]
```

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

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

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

A terceira e final versão usa a instrução ELSE IF para realizar uma série de testes independentes, cada um dos quais possui um bloco de comandos associado. Os testes são realizados um após o outro até que um

deles seja satisfeito, em cujo caso o bloco associado é executado, enquanto que os outros são solenemente ignorados. Após, o controle do fluxo é transferido para a instrução `END IF`. Caso nenhuma condição seja satisfeita, nenhum bloco é executado, exceto se houver uma instrução `ELSE` final, que abarca quaisquer possibilidades não satisfeitas nos testes realizados no construto. A forma geral é:

```
[<nome>:] IF (<expressão relacional e/ou lógica>) THEN
   <bloco>
[ELSE IF (<expressão relacional e/ou lógica>) THEN [<nome>]
   <bloco>]
...
[ELSE [<nome>]
   <bloco>]
END IF [<nome>]
```

Pode haver qualquer número (inclusive zero) de instruções `ELSE IF` e, no máximo, uma instrução `ELSE`. Novamente, o `<nome>` é opcional, mas se for adotado no cabeçalho do construto, então deve ser mencionado em todas as circunstâncias ilustradas acima. O exemplo a seguir ilustra o encadeamento de construtos `IF`. Estruturas ainda mais complicadas são possíveis, mas neste caso é recomendável adotar `nomes` para os construtos, como forma de facilitar a leitura e compreensão do programa.

```
IF (I < 0) THEN
   IF (J < 0) THEN
      X= 0.0
      Y= 0.0
   ELSE
      Z= 0.0
   END IF
ELSE IF (K < 0) THEN
   Z= 1.0
ELSE
   X= 1.0
   Y= 1.0
END IF
```

O programa-exemplo a seguir (programa 5.1) faz uso de construtos `IF` para implementar o cálculo do fatorial de um número natural, já utilizando o construto `DO`, abordado na seção 5.3.

5.3 Construto `DO`

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

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

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

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

Esta forma do laço `DO` não usa rótulos, como a forma definida na seção 5.1.6. A forma geral de um construto `DO` é a seguinte:

Programa 5.1: Programa que utiliza os construtos `IF` e `DO`.

```
program if_fat
!Calcula o fatorial de um nmero natural.
implicit none
integer :: i, fat, j
!
print *, "Entre com valor:"
read *, i
if (i < 0) then
   print *, "No possvel calcular o fatorial."
else if (i == 0) then
   print *, "fat(",i,")=",1
else
   fat= 1
   do j= 1, i
      fat= fat*j
   end do
   print *, "fat(",i,")=",fat
end if
end program if_fat
```

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

onde `<variável>` é uma variável inteira e as três expressões contidas no cabeçalho do construto devem ser do mesmo tipo. No cabeçalho acima, cada uma das expressões indica:

`<expressão 1>`: o valor inicial da `<variável>`;

`<expressão 2>`: o valor máximo, ou limite, da `<variável>` (não necessariamente deve ser o último valor assumido pela variável);

`<expressão 3>`: o passo, ou incremento, da variável em cada nova iteração.

Este construto também é aceito no Fortran 77, exceto pela possibilidade de se definir um `<nome>`. Como no caso de um laço `DO` rotulado, o número de iterações é definido antes de se executar o primeiro comando do `<bloco>` e é dado pelo resultado da conta

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

Novamente, se o resultado do primeiro argumento da função `MAX` acima for negativo, o laço não será executado. Isto pode acontecer, por exemplo, se `<expressão 2>` for menor que `<expressão 1>` e o passo `<expressão 3>` for positivo. Novamente, se o passo `<expressão 3>` for omitido, este será igual a 1. O exemplo abaixo ilustra o uso de expressões no cabeçalho do construto:

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

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

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

Programa 5.2: Exemplo de uso do construto DO.

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

Como no caso do DO rotulado, caso alguma das expressões envolvam o uso de variáveis, estas podem ser modificadas no bloco do laço, inclusive o passo das iterações. Entretanto, o número de iterações e o passo a ser realmente tomado não são modificados, uma vez que foram pré-determinados antes do início das iterações. O programa abaixo (programa 5.2) exemplifica este fato:

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

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

Neste exemplo, o número de iterações será

```
MAX(N-2+1,0) = MAX(N-1,0).
```

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

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

5.3.1 Construto DO ilimitado

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

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

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

5.3.2 Instrução EXIT

Esta instrução permite a saída, por exemplo, de um laço sem limite, mas o seu uso não está restrito a este caso particular, podendo ser usada em um construto DO geral. A forma geral desta instrução é:

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

onde o <nome> é opcional, mas deve ser usado caso ele exista e, neste caso, ele serve para denotar de qual construto a saída deve ser feita. Isto ocorre, por exemplo, no caso de construtos encadeados. Execução de um EXIT transfere controle ao primeiro comando executável após o END DO [<nome>] correspondente.

Como exemplo, temos:

```
DO
   ...
   I= I + 1
   IF(I == J) EXIT
   ...
END DO
```

5.3.3 Instrução CYCLE

A forma geral desta instrução é:

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

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

```
program do_cycle
implicit none
integer :: index= 1
do
   index= index + 1
   if(index == 20) cycle
   if(index == 30) exit
   print *, "Valor do ndice:",index
end do
end program do_cycle
```

5.4 Construto CASE

Fortran 90/95 fornece uma outra alternativa para selecionar uma de diversas opções: trata-se do construto CASE. A principal diferença entre este construto e um bloco IF está no fato de somente uma expressão ser calculada para decidir o fluxo e esta pode ter uma série de resultados pré-definidos. A forma geral do construto CASE é:

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

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

```
CASE (1)
```

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

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

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

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

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

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

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

```
CASE DEFAULT
```

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

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

No exemplo acima, os caracteres `'C', 'D', 'R'` e todos os caracteres após o último indicam nomes de variáveis do tipo de caractere. Os caracteres `'I'` e `'N'` indicam variáveis do tipo inteiro, e todos os outros caracteres alfabéticos indicam variáveis reais.

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

```
program case_string
implicit none
character(len= 5) :: nome
print *, "Entre com o nome (5 caracteres):"
read "(a5)", nome
select case (nome)
case ("a":"z")          ! Seleciona nome que comea com letras minsculas.
   print *, "Palavra inicia com Letra minscula."
case ("A":"Z")          ! Seleciona nome que comea com letras maisculas.
   print *, "Palavra inicia com letras maiscula."
case ("0":"9")          ! Seleciona nmeros.
   print *, "Palavra inicia com nmeros!!!"
case default            ! Outros tipos de caracteres.
   print *, "Palavra inicia com Caracteres especiais!!!"
end select
end program case_string
```

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

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

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

```
! Imprime uma tabela de converso das escalas Celsius e Fahrenheit
!   entre limites de temperatura especificados.
program conv_temp
implicit none
character(len= 1) :: scale
integer :: low_temp,high_temp,temp
real :: celsius,farenheit
!
read_loop: do
! L escala e limites.
   print *, "Escala de temperaturas (C/F):"
   read "(a)", scale
! Confere validade dos dados.
   if (scale /= "C" .and. scale /= "F")then
      print *, "Escala no vlida!"
      exit read_loop
   end if
   print *, "Limites (temp. inferior , temp. superior):"
   read *, low_temp,high_temp
!
! Lao sobre os limites de temperatura.
   do temp= low_temp, high_temp
! Escolhe frmula de converso
      select case (scale)
      case ("C")
         celsius= temp
         farenheit= 9*celsius/5.0 + 32.0
      case ("F")
         farenheit= temp
         celsius= 5*(farenheit - 32)/9.0
      end select
! Imprime tabela
      print *, celsius,"graus C correspondem a",farenheit,"graus F."
   end do
end do read_loop
!
! Finalizao.
print *, "Final dos dados vlidos."
end program conv_temp
```

Capítulo 6

Processamento de Matrizes

A definição e o processamento de matrizes e vetores sempre foi um recurso presente em todas as linguagens de programação, inclusive no Fortran 77. Uma novidade importante introduzida no Fortran 90/95 é a capacidade estendida de processamento das mesmas. Agora é possível trabalhar diretamente com a matriz completa, ou com seções da mesma, sem ser necessário o uso de laços `DO`. Novas funções intrínsecas agora atuam de forma *elemental* (em todos os elementos) em matrizes e funções podem retornar valores na forma de matrizes. Também estão disponíveis as possibilidades de matrizes alocáveis, matrizes de forma assumida e matrizes dinâmicas.

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

6.1 Terminologia e especificações de matrizes

Uma *matriz* ou *vetor*[1] é um outro tipo de objeto composto suportado pelo Fortran 77/90/95. Uma matriz consiste de um conjunto retangular de elementos, todos do mesmo tipo e espécie do tipo. Uma outra definição equivalente seria: uma matriz é um grupo de posições na memória do computador as quais são acessadas por intermédio de um único nome, fazendo-se uso dos *subscritos* da matriz. Este tipo de objeto é útil quando for necessário se fazer referência a um número grande, porém a princípio desconhecido, de variáveis do tipo intrínseco ou outras estruturas, sem que seja necessário definir um nome para cada variável.

O Fortran 77/90/95 permite que uma matriz tenha até sete subscritos, cada um relacionado com uma dimensão da matriz. As dimensões de uma matriz podem ser especificadas usando-se tanto o atributo `DIMENSION` quanto a declaração `DIMENSION`.

Os índices de cada subscrito da matriz são constantes ou variáveis inteiras e, por convenção, eles começam em 1, exceto quando um intervalo distinto de valores é especificado, através do fornecimento de um limite inferior e um limite superior.

A declaração de matrizes difere ligeiramente entre o Fortran 77 e o Fortran 90/95.

Fortran 77. Uma matriz pode ser definida tanto na declaração de tipo intrínseco quanto com o uso da declaração `DIMENSION`:

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

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

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

Fortran 90/95. As formas de declarações de matrizes em Fortran 77 são aceitas no Fortran 90/95. Porém, é recomendável que estas sejam declaradas na forma de *atributos* de tipos de variáveis. Por exemplo,

```
REAL, DIMENSION(50) :: W
REAL, DIMENSION(5:54) :: X
CHARACTER(LEN= 25), DIMENSION(5) :: COLUMN
CHARACTER(LEN= 30), DIMENSION(10) :: ROW
```

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

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

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

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

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

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

Por exemplo, as seguintes matrizes:

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

A matriz `A` possui:

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

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

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

```
<tipo>[[, DIMENSION(<lista de extensões>)] [, < outros atributos>] ::]  <lista de nomes>
```

Entretanto, a forma recomendável da declaração é a seguinte:

```
<tipo>, DIMENSION(<lista de extensões>) [, <outros atributos>] :: <lista de no-
mes>
```

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

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

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

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

Os atributos contidos na lista acima serão abordados ao longo deste e dos próximos capítulos. Os atributos em vermelho: `POINTER` e `TARGET` consistem em recursos avançados do Fortran 90/95 que serão discutidos em separado.

Finalmente, segue a `<lista de nomes>` válidos no Fortran, onde os mesmos são atribuídos às matrizes. Os seguintes exemplos mostram a forma da declaração de diversos tipos diferentes de matrizes, alguns dos quais são novos no Fortran 90/95 e serão abordados adiante.

1. Inicialização de vetores contendo 3 elementos:

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

2. Declaração da matriz automática `LOGB`. Aqui, `LOGA` é uma matriz qualquer ("muda" ou "dummy") e `SIZE` é uma função intrínseca que retorna um escalar inteiro correspondente ao tamanho do seu argumento:

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

3. Declaração das matrizes dinâmicas, ou alocáveis, de duas dimensões `A` e `B`. A forma das matrizes será definida *a posteriori* por um comando `ALLOCATE`:

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

4. Declaração das matrizes de forma assumida de três dimensões `A` e `B`. A forma das matrizes será assumida a partir das informações transferidas pela rotina que aciona o sub-programa onde esta declaração é feita.

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

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

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

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

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

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

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

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

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

```
!  Declara um vetor, define valores aos elementos do mesmo e imprime
!   estes valores na tela.
program ex1_array
implicit none
integer, parameter :: dp= selected_real_kind(10,200)
integer :: i
real, dimension(10) :: vr
real(kind= dp), dimension(10) :: vd
!
do i= 1,10
   vr(i)= sqrt(real(i))
   vd(i)= sqrt(real(i))
end do
print *, "Raiz quadrada dos 10 primeiros inteiros, em precisao simples:"
print *, vr       ! Imprime todos os componentes do vetor.
print *, " "
print *, "Raiz quadrada dos 10 primeiros inteiros, em precisao dupla:"
print *, vd
end program ex1_array
```

O segundo programa-exemplo, a seguir, está baseado no programa `alunos`, descrito na seção 3.8. Agora, será criado um vetor para armazenar os dados de um número fixo de alunos e os resultados somente serão impressos após a aquisição de todos os dados.

```
!Dados acerca de alunos usando tipo derivado.
program alunos_vet
implicit none
integer :: i,ndisc= 5 !Mude este valor, caso seja maior.
type:: aluno
   character(len= 20):: nome
   integer:: codigo
   real:: n1,n2,n3,mf
end type aluno
type(aluno), dimension(5):: disc
!
do i= 1,ndisc
   print*, "Nome:"
   read "(a)", disc(i)%nome
   print*, "cdigo:"
   read*,disc(i)%codigo
   print*, "Notas: N1,N2,N3:"
   read*,disc(i)%n1,disc(i)%n2,disc(i)%n3
   disc(i)%mf= (disc(i)%n1 + disc(i)%n2 + disc(i)%n3)/3.0
end do
do i= 1,ndisc
   print*, " "
   print*, "---------> ",disc(i)%nome," (",disc(i)%codigo,") <---------"
   print*, "          Mdia final: ",disc(i)%mf
end do
end program alunos_vet
```

6.2 Expressões e atribuições envolvendo matrizes

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

Um desenvolvimento novo, introduzido no Fortran 90, é a sua habilidade de realizar operações envolvendo a matriz na sua totalidade, possibilitando o tratamento de uma matriz como um objeto único, o que, no mínimo, facilita enormemente a construção, leitura e interpretação do código. Uma outra vantagem, ainda mais importante, resulta deste novo modo de encarar matrizes. Com o desenvolvimento dos processadores e das arquiteturas de computadores, entraram em linha, recentemente, sistemas compostos por mais de um processador, os quais fazem uso da idéia de *processamento distribuído* ou, em outras palavras, *processamento paralelo*. As normas definidas pelo comitê X3J3 para o padrão da linguagem Fortran 90/95 supõe que compiladores usados em sistemas distribuídos devem se encarregar de distribuir automaticamente os processos numéricos envolvidos nas expressões com matrizes de forma equilibrada entre os diversos processadores que compõe a arquitetura. A evidente vantagem nesta estratégia consiste no fato de que as mesmas operações numéricas são realizadas de forma simultânea em diversos componentes distintos das matrizes, acelerando substancialmente a eficiência do processamento. Com a filosofia das operações sobre matrizes inteiras, a tarefa de implantar a paralelização do código numérico fica, essencialmente, a cargo do compilador e não do programador. Uma outra vantagem deste enfoque consiste na manutenção da portabilidade dos códigos numéricos.

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

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

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

o resultado da adição de `A` por `B` é:

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

o resultado da multiplicação é:

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

e o resultado da divisão é:

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

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

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

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

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

```
A= A + B
```

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

As vantagens do processamento de matrizes inteiras, presente no Fortran 90/95 mas ausente no Fortran 77, podem ser contempladas através da comparação de alguns exemplos de códigos desenvolvidos em ambas as revisões da linguagem. Nestes códigos serão também apresentadas algumas funções intrínsecas do Fortran 90/95 que operam em matrizes, elemento a elemento. Uma descrição completa das rotinas intrínsecas é apresentada no capítulo 7.

1. Considere três vetores, `A`, `B` e `C`, todos do mesmo comprimento. Inicialize todos os elementos de `A` a zero e realize as atribuições `A(I)= A(I)/3.1 + B(I)*SQRT(C(I))` para todos os valores de `I`

   Solução Fortran 77.

```
      REAL A(20), B(20), C(20)
...
      DO 10 I= 1, 20
         A(I)= 0.0
10    CONTINUE
...
      DO 20 I= 1, 20
         A(I)= A(I)/3.1 + B(I)*SQRT(C(I))
20    CONTINUE
```

   Solução Fortran 90/95.

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

   Note como o código ficou mais reduzido e fácil de ser lido. Além disso, a função intrínseca `SQRT` opera sobre cada elemento do vetor `C`.

2. Considere três matrizes bi-dimensionais com a mesma forma. Multiplique duas matrizes entre si, elemento a elemento, e atribua o resultado a uma terceira matriz.

   Solução Fortran 77.

```
      REAL A(5,5), B(5,5), C(5,5)
...
      DO 20 I= 1, 5
         DO 10 J= 1, 5
            C(J,I)= A(J,I) + B(J,I)
10       CONTINUE
20    CONTINUE
```

   Solução Fortran 90/95.

```
REAL, DIMENSION(5,5) :: A, B, C
...
C= A + B
```

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

   Solução Fortran 77.

```
      REAL A(5,5,5)
      REAL VAL_MAX
...
      VAL_MAX= 0.0
      DO 30 K= 1, 5
         DO 20 J= 1, 5
            DO 10 I= 1, 5
               IF((A(I,J,K) .GT. VAL_MAX) .AND.
     &                      (A(I,J,K) .LT. 1000.0))VAL_MAX= A(I,J,K)
10          CONTINUE
20       CONTINUE
30    CONTINUE
```

Solução Fortran 90/95.

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

Note que no Fortran 95 conseguiu-se fazer em uma linha o que necessitou de 8 linhas no Fortran 77. A função intrínseca `MAXVAL` devolve o valor máximo entre os elementos de uma matriz. O argumento opcional `MASK=(...)` estabelece uma *máscara,* isto é, uma expressão lógica envolvendo a(s) matriz(es). Em `MASK=(A<1000.0)`, somente aqueles elementos de `A` que satisfazem a condição de ser menores que 1000 são levados em consideração.

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

Solução Fortran 77.

```
      REAL A(5,5,5)
      REAL MEDIA,ACUM
      INTEGER CONTA
...
      ACUM= 0.0
      CONTA= 0
      DO 30 K= 1, 5
         DO 20 J= 1, 5
            DO 10 I= 1, 5
               IF(A(I,J,K) .GT. 3000.0)THEN
                  ACUM= ACUM + A(I,J,K)
                  CONTA= CONTA + 1
               END IF
10          CONTINUE
20       CONTINUE
30    CONTINUE
      MEDIA= ACUM/REAL(CONTA)
```

Solução Fortran 90/95.

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

Agora, conseguiu-se realizar em uma linha de código em Fortran 90 o que necessitou de 13 linhas em Fortran 77.

Os últimos dois exemplos fizeram uso das seguintes funções intrínsecas (ver seção 7.12):

`MAXVAL` - retorna o valor máximo de uma matriz.

`SUM` - retorna a soma de todos os elementos de uma matriz.

`COUNT` - retorna o número de elementos da matriz que satisfazem a máscara.

6.3 Seções de matrizes

Uma *sub-matriz*, também chamada *seção de matriz*, pode ser acessada através da especificação de um intervalo de valores de subscritos da matriz. Uma seção de matriz pode ser acessada e operada da mesma forma que a matriz completa, mas não é possível fazer-se referência direta a elementos individuais ou a subseções da seção.

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

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

Estes recursos serão descritos a seguir.

6.3.1 Subscritos simples

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

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

6.3.2 Tripleto de subscritos

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

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

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

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

$$\texttt{RA =} \begin{bmatrix} 0\,0\,0\,0\,0 \\ 0\,\texttt{X}\,0\,0\,0 \\ 0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0 \end{bmatrix} \Longrightarrow \texttt{RA(2:2,2:2) = X;}$$ Elemento simples, escalar, forma: `(/1/).`

$$\texttt{RA =} \begin{bmatrix} 0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0 \\ 0\,0\,\texttt{X}\,\texttt{X}\,\texttt{X} \\ 0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0 \end{bmatrix} \Longrightarrow \texttt{RA(3,3:5);}$$ seção de linha da matriz, forma: `(/3/).`

$$\texttt{RA =} \begin{bmatrix} 0\,0\,\texttt{X}\,0\,0 \\ 0\,0\,\texttt{X}\,0\,0 \\ 0\,0\,\texttt{X}\,0\,0 \\ 0\,0\,\texttt{X}\,0\,0 \\ 0\,0\,\texttt{X}\,0\,0 \end{bmatrix} \Longrightarrow \texttt{RA(:,3);}$$ coluna inteira, forma: `(/5/).`

$$\texttt{RA =} \begin{bmatrix} 0\,\texttt{X}\,\texttt{X}\,\texttt{X}\,0 \\ 0\,0\,0\,0\,0 \\ 0\,\texttt{X}\,\texttt{X}\,\texttt{X}\,0 \\ 0\,0\,0\,0\,0 \\ 0\,\texttt{X}\,\texttt{X}\,\texttt{X}\,0 \end{bmatrix} \Longrightarrow \texttt{RA(1::2,2:4);}$$ seções de linhas com passo 2, forma: `(/3,3/).`

6.3.3 Vetores de subscritos

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

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

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

```
! Resulta:
!RB= (/ RA(1), RA(3), RA(5) /), ou
!RB= (/ 1.2, 3.0, 1.0 /).
```

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

```
IV= (/ 1, 3, 5 /)
RA(IV)= (/ 1.2, 3.4, 5.6 /) !Atribuições dos elementos 1, 3 e 5 de RA.
! Resulta:
! RA(1)= 1.2; RA(3)= 3.4; RA(5)= 5.6
! Os elementos 2, 4 e 6 de RA não foram definidos.
```

6.4 Atribuições de matrizes e sub-matrizes

Tanto matrizes inteiras quanto seções de matrizes podem ser usadas como operandos (isto é, podem estar tanto no lado esquerdo quanto do lado direito de uma atribuição) desde que todos os operandos sejam conformáveis (página 50). Por exemplo,

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

Um outro exemplo, acompanhado de figura, torna a operação com sub-matrizes conformáveis mais clara.

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

A figura 6.1 ilustra como as duas sub-matrizes são conformáveis neste caso e o resultado da soma das duas seções será atribuído à matriz `C`, a qual possui a mesma forma.

O programa-exemplo 6.1 mostra algumas operações e atribuições básicas de matrizes:

6.5 Matrizes de tamanho zero

Matrizes de tamanho zero, ou *matrizes nulas* também são permitidas em Fortran 90/95. A noção de uma matriz nula é útil quando se quer contar com a possibilidade de existência de uma matriz sem nenhum elemento, o que pode simplificar a programação do código em certas situações. Uma matriz é nula quando o limite inferior de uma ou mais de suas dimensões é maior que o limite superior.

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

```
DO I= 1, N
   X(I)= B(I)/A(I,I)
   B(I+1:N)= B(I+1:N) - A(I+1:N,I)*X(I)
END DO
```

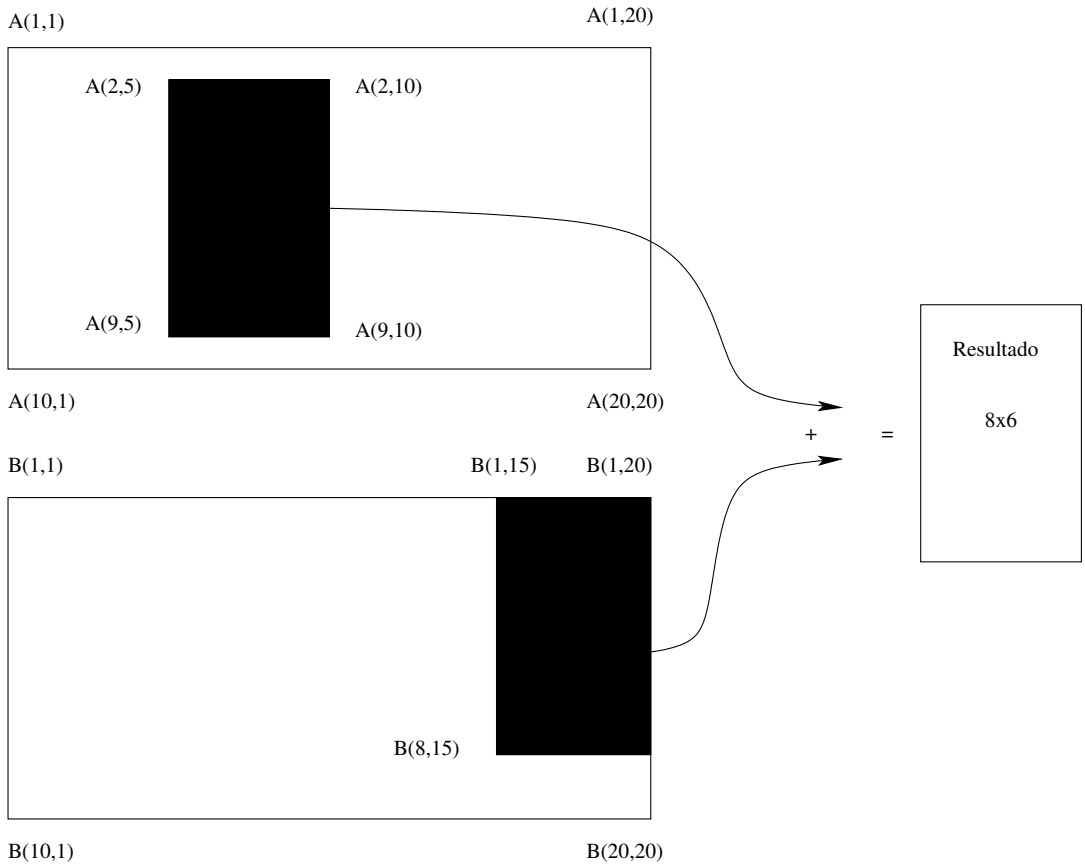


Figura 6.1: A soma de duas seções de matrizes conformáveis.

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

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

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

Programa 6.1: Expressões e atribuições envolvendo matrizes.

```
program testa_atr_matr
implicit none
real, dimension(3,3) :: a
real, dimension(2,2) :: b
integer :: i, j
!
do i= 1,3
   do j= 1,3
      a(i,j)= sin(real(i)) + cos(real(j))
   end do
end do
b= a(1:2,1:3:2)
print*, "Matriz A:"
print"(3(f12.5))", ((a(i,j), j= 1,3), i= 1,3)
Print*, "Matriz B:"
print"(2(f12.5))", ((b(i,j), j= 1,2), i= 1,2)
end program testa_atr_matr
```

é sempre válida.

6.6 Construtores de matrizes

Um construtor de matrizes cria um vetor (matriz de posto 1) contendo valores constantes. Estes construtores servem, por exemplo, para inicializar os elementos de um vetor, como foi exemplificado na página 51. Outro exemplo de uso dos construtores de matrizes está na definição de um vetor de subscritos, como foi abordado na seção 6.3.3.

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

```
(/ <lista de valores do construtor> /)
```

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

```
IV= (/ 1, 3, 5 /)
```

como pode ser um conjunto de expressões numéricas:

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

ou ainda um comando `DO` implícito no construtor, cuja forma geral é a seguinte:

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

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

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

sendo a `<variável>` substituída por `<exp1>`, `<exp1> + <exp3>`, ..., `<exp2>`, como acontece no construto `DO` (seção 5.3). Também é possível encadear-se `DO`'s implícitos. A seguir, alguns exemplos são apresentados:

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


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


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


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


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

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

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

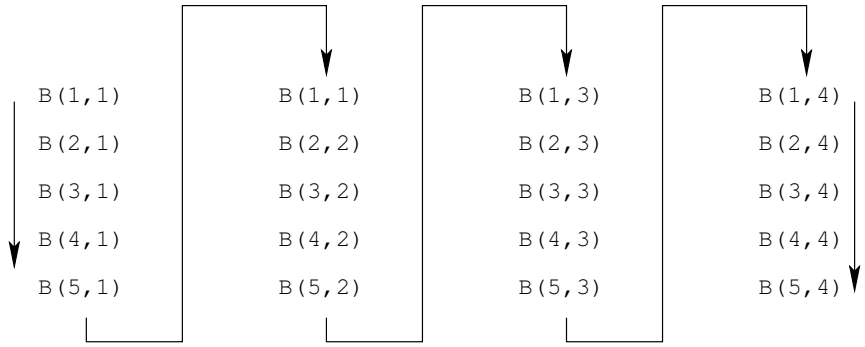


Figura 6.2: O ordenamento dos elementos da matriz `B(5,4)`.

6.6.1 A função intrínseca `RESHAPE`.

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

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

gera a matriz de posto 2:

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

a qual é uma matriz de forma `(/ 2, 3 /)`, isto é, 2 linhas e 3 colunas. Um outro exemplo seria:

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

de onde resulta

$$\texttt{RA} = \begin{bmatrix} 2\,3 \\ 3\,4 \\ 4\,5 \end{bmatrix}$$

Usando a função `RESHAPE`, o programa-exemplo da página 57 apresenta uma forma mais concisa:

```
program testa_atr_matr
implicit none
real, dimension(3,3) :: a
real, dimension(2,2) :: b
integer :: i, j
!
a= reshape(source= (/((sin(real(i))+cos(real(j)), i= 1,3), j= 1,3)/), &
           shape= (/3,3/))
b= a(1:2,1:3:2)
print*, "Matriz A:"
print"(3(f12.5))", ((a(i,j), j= 1,3), i= 1,3)
Print*, "Matriz B:"
print"(2(f12.5))", ((b(i,j), j= 1,2), i= 1,2)
end program testa_atr_matr
```

6.6.2 A ordem dos elementos de matrizes

A maneira como a função `RESHAPE` organizou os elementos das matrizes na seção 6.6.1 seguiu a denominada *ordem dos elementos de matriz*, a qual é a maneira como a maioria dos compiladores de Fortran armazena os elementos de matrizes em espaços contíguos de memória. Este ordenamento é obtido variando-se inicialmente o índice da primeira dimensão da matriz, depois variando-se o índice da segunda dimensão e assim por diante. Em uma matriz com 2 dimensões isto é obtido variando-se inicialmente as linhas e depois as colunas. A figura 6.2 ilustra este procedimento com a matriz `B(5,4)`.

Em uma matriz com mais de 2 dimensões, o ordenamento é realizado da mesma maneira. Assim, dada a matriz

```
REAL, DIMENSION(-10:5, -20:1, 0:1, -1:0, 2, 2, 2) :: G
```

o ordenamento segue a ordem:

G(-10,-20,0,-1,1,1,1) → G(-9,-20,0,-1,1,1,1) → G(-8,-20,0,-1,1,1,1) →...→ G(5,-20,0,-1,1,1,1) →
G(-10,-19,0,-1,1,1,1) → G(-9,-19,0,-1,1,1,1) → G(-8,-19,0,-1,1,1,1) →...→ G(5,-19,0,-1,1,1,1) →
G(-10,-18,0,-1,1,1,1) → ... → G(-8,-15,0,-1,1,1,1) →...→ G(5,1,1,0,2,2,1) →
G(-10,1,1,0,2,2,2) → G(-9,1,1,0,2,2,2) → ... →...→ G(5,1,1,0,2,2,2)

Em muitas situações, é mais rápido escrever-se um código que processa matrizes seguindo o ordenamento dos elementos. As funções intrínsecas do Fortran 90/95 que manipulam matrizes inteiras foram concebidas de forma a levar em conta este fato.

6.7 Rotinas intrínsecas elementais aplicáveis a matrizes

O Fortran 90/95 permite a existência de rotinas (funções ou subrotinas) intrínsecas *elementais*, isto é, rotinas que se aplicam a cada elemento de uma matriz. Matrizes podem, então, ser usadas como argumentos de rotinas intrínsecas, da mesma forma que escalares. Este tipo de recurso não existem no Fortran 77. As operações definidas na rotina intrínseca serão aplicadas a cada elemento da matriz separadamente. Novamente, caso mais de uma matriz apareça no argumento da rotina, estas devem ser conformáveis.

A seguir, alguns exemplos de rotinas intrínsecas elementais. A lista completa destas rotinas pode ser obtida no capítulo 7.

1. Calcule as raízes quadradas de todos os elementos da matriz `A`. O resultado será atribuído à matriz `B`, a qual é conformável com `A`.

```
REAL, DIMENSION(10,10) :: A, B
...
B= SQRT(A)
```

2. Calcule a exponencial de todos os argumentos da matriz `A`. Novamente, o resultado será atribuído a `B`.

```
COMPLEX, DIMENSION(5,-5:15, 25:125) :: A, B
...
B= EXP(A)
```

3. Encontre o comprimento da string (variável de caractere) excluindo brancos no final da variável para todos os elementos da matriz `CH`.

```
CHARACTER(LEN= 80), DIMENSION(10) :: CH
INTEGER :: COMP
...
COMP= LEN_TRIM(CH)
```

6.8 Comando e construto `WHERE`

Em muitas situações, é desejável realizar-se operações somente para alguns elementos de uma matriz; por exemplo, somente para aqueles elementos que são positivos.

6.8.1 Comando `WHERE`

O comando `WHERE` fornece esta possibilidade. Um exemplo simples é:

```
REAL, DIMENSION(10,10) :: A
...
WHERE (A > 0.0) A= 1.0/A
```

o qual fornece a recíproca (inversa) de todos os elementos positivos de `A`, deixando os demais inalterados. A forma geral do comando é

```
WHERE (<expressão lógica matriz>) <variável matriz>= <expressão matriz>
```

A `<expressão lógica matriz>` deve ter a mesma forma que a `<variável matriz>`. A expressão lógica é desenvolvida inicialmente e então somente aqueles elementos da `<variável matriz>` que satisfazem o teste lógico são operados pela `<expressão matriz>`. Os restantes permanecem inalterados.

6.8.2 Construto WHERE

Uma única expressão lógica de matriz pode ser usada para determinar uma seqüência de operações e atribuições em matrizes, todas com a mesma forma. A sintaxe deste construto é:

```
WHERE (<expressão lógica matriz>)
   <operações atribuições matrizes>
END WHERE
```

Inicialmente, a `<expressão lógica matriz>` é desenvolvida em cada elemento da matriz, resultando em uma matriz lógica temporária, cujos elementos são os resultados da `<expressão lógica matriz>`. Então, cada operação e atribuição de matriz no bloco do construto é executada sob o controle da *máscara* determinada pela matriz lógica temporária; isto é, as `<operações atribuições matrizes>` serão realizadas em cada elemento das matrizes do bloco que corresponda ao valor `.TRUE.` da matriz lógica temporária.

Existe também uma forma mais geral do construto `WHERE` que permite a execução de atribuições a elementos de matrizes que **não** satisfazem o teste lógico no cabeçalho:

```
WHERE (<expressão lógica matriz>)
   <operações atribuições matrizes 1>
ELSEWHERE
   <operações atribuições matrizes 2>
END WHERE
```

O bloco `<operações atribuições matrizes 1>` é executado novamente sob o controle da máscara definida na `<expressão lógica matriz>` e somente elementos que satisfazem esta máscara são afetados. Em seguida, as `<operações atribuições matrizes 2>` são executadas sob o controle da máscara definida por `.NOT. <expressão lógica matriz>`, isto é, novas atribuições são realizadas sobre elementos que não satisfazem o teste lógico definido no cabeçalho.

Um exemplo simples do construto `WHERE` é:

```
WHERE (PRESSURE <= 1.0)
   PRESSURE= PRESSURE + INC_PRESSURE
   TEMP= TEMP + 5.0
ELSEWHERE
   RAINING= .TRUE.
END WHERE
```

Neste exemplo, `PRESSURE`, `INC_PRESSURE`, `TEMP` e `RAINING` são todas matrizes com a mesma forma, embora não do mesmo tipo.

O programa-exemplo 6.2 mostra outra aplicação do construto `WHERE`.

Um recurso ausente no Fortran 90 mas incorporado no Fortran 95 é o mascaramento na instrução `ELSEWHERE`, juntamente com a possibilidade de conter qualquer número de instruções `ELSEWHERE` mascaradas mas, no máximo, uma instrução `ELSEWHERE` sem máscara, a qual deve ser a última. Todas as expressões lógicas que definem as máscaras devem ter a mesma forma. Adicionalmente, os construtos `WHERE` podem ser encadeados e nomeados; neste caso, as condições lógicas de todas as máscaras devem ter a mesma forma. O seguinte exemplo ilustra este recurso:

```
ATRIB1: WHERE (<cond 1>)
           <corpo 1>          !Mascarado por <cond 1>
        ELSEWHERE (<cond 2>) ATRIB1
           <corpo 2>          !Masc. por <cond 2> .AND. .NOT. <cond 1>
ATRIB2:    WHERE (<cond 4>)
              <corpo 4>       !Masc. por <cond 2> .AND. .NOT. <cond 1> .AND. <cond 4>
           ELSEWHERE ATRIB2
```

Programa 6.2: Exemplo do construto `WHERE`.

```
program testa_where
implicit none
real, dimension(3,3) :: a
integer :: i, j
!
a= reshape(source= (/ ((sin(real(i+j)), i= 1,3), j= 1,3) /), shape= (/3,3/))
print*, "Matriz A original:"
print*, a(1,:)
print*, a(2,:)
print*, a(3,:)
!
where (a >= 0.0)
   a= sqrt(a)
elsewhere
   a= a**2
end where
print*, "Matriz A modificada:"
print*, a(1,:)
print*, a(2,:)
print*, a(3,:)
end program testa_where
```

```
      <corpo 5>       !Masc. por <cond 2> .AND. .NOT. <cond 1>
      ...             !    .AND. .NOT. <cond 4>
   END WHERE ATRIB2
   ...
ELSEWHERE (<cond 3>) ATRIB1
   <corpo 3>          !Masc. por <cond 3> .AND. .NOT. <cond 1>
   ...                !    .AND. .NOT. <cond 2>
ELSEWHERE ATRIB1
   <corpo 6>          !Masc. por .NOT. <cond 1> .AND. .NOT. <cond 2>
   ...                !    .AND. .NOT. <cond 3>
END WHERE ATRIB1
```

6.9 Matrizes alocáveis

Uma novidade importante introduzida no Fortran 90/95 é a habilidade de se declarar variáveis dinâmicas; em particular, matrizes dinâmicas. Fortran 90 fornece tanto matrizes alocáveis quanto matrizes automáticas, ambos os tipos sendo matrizes dinâmicas. Usando matrizes alocáveis, é possível alocar e de-alocar espaço de memória conforme necessário. O recurso de matrizes automáticas permite que matrizes locais em uma função ou subrotina tenham forma e tamanho diferentes cada vez que a rotina é invocada. Matrizes automáticas são discutidas no capítulo 8.

Matrizes alocáveis permitem que grandes frações da memória do computador sejam usadas somente quando requerido e, posteriormente, liberadas, quando não mais necessárias. Este recurso oferece um uso de memória muito mais eficiente que o Fortran 77, o qual oferecia somente alocação estática (fixa) de memória. Além disso, o código torna-se muito mais robusto, pois a forma e o tamanho das matrizes podem ser decididos durante o processamento do código.

Uma matriz alocável é declarada na linha de declaração de tipo de variável com o atributo `ALLOCATABLE`. O posto da matriz deve também ser declarado com a inclusão dos símbolos de dois pontos "`:`", um para cada dimensão da matriz. Por exemplo, a matriz de duas dimensões `A` é declarada como alocável através da declaração:

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

Esta forma de declaração não aloca espaço de memória imediatamente à matriz, como acontece com as declarações usuais de matrizes. O status da matriz nesta situação é *not currently allocated*, isto é, corren-

temente não alocada. Espaço de memória é dinamicamente alocado durante a execução do programa, logo antes da matriz ser utilizada, usando-se o comando `ALLOCATE`. Este comando especifica os limites da matriz, seguindo as mesmas regras definidas na seção 6.1. A seguir são dados dois exemplos de uso deste comando:

```
ALLOCATE (A(0:N,M))
```

ou usando expressões inteiras:

```
ALLOCATE (A(0:N+1,M))
```

Como se pode ver, esta estratégia confere uma flexibilidade grande na definição de matrizes ao programador.

O espaço alocado à matriz com o comando `ALLOCATE` pode, mais tarde, ser liberado com o comando `DEALLOCATE`. Este comando requer somente nome da matriz previamente alocada. Por exemplo, para liberar o espaço na memória reservado para a matriz `A`, o comando fica

```
DEALLOCATE (A)
```

Tanto os comandos `ALLOCATE` e `DEALLOCATE` possuem o especificador opcional `STAT`, o qual retorna o status do comando de alocação ou de-alocação. Neste caso, a forma geral do comando é:

```
ALLOCATE (<lista de objetos alocados> [, STAT= <status>])
DEALLOCATE (<lista de objetos alocados> [, STAT= <status>])
```

onde `<status>` é uma variável inteira escalar. Se `STAT=` está presente no comando, `<status>` recebe o valor zero se o procedimento do comando `ALLOCATE/DEALLOCATE` foi bem sucedido ou um valor positivo se houve um erro no processo. Se o especificador `STAT=` não estiver presente e ocorra um erro no processo, o programa é abortado. Finalmente, é possível alocar-se ou de-alocar-se mais de uma matriz simultaneamente, como indica a `<lista de objetos alocados>`.

Um breve exemplo do uso destes comandos seria:

```
REAL, DIMENSION(:), ALLOCATABLE :: A, B
REAL, DIMENSION(:,:), ALLOCATABLE :: C
INTEGER :: N
...
ALLOCATE (A(N), B(2*N), C(N,2*N))
B(:N)= A
B(N+1:)= SQRT(A)
DO I= 1,N
   C(I,:)= B
END DO
DEALLOCATE (A, B, C)
```

Matrizes alocáveis tornam possível o requerimento freqüente de declarar uma matriz tendo um número variável de elementos. Por exemplo, pode ser necessário ler variáveis, digamos `tam1` e `tam2` e então declarar uma matriz com `tam1` $\times$ `tam2` elementos:

```
INTEGER :: TAM1, TAM2
REAL, DIMENSION (:,:), ALLOCATABLE :: A
INTEGER :: STATUS
...
READ*, TAM1, TAM2
ALLOCATE (A(TAM1,TAM2), STAT= STATUS)
IF (STATUS > 0)THEN
... ! Comandos de processamento de erro.
END IF
... ! Uso da matriz A.
DEALLOCATE (A)
...
```

No exemplo acima, o uso do especificador `STAT=` permite que se tome providências caso não seja possível alocar a matriz, por alguma razão.

Como foi mencionado anteriormente, uma matriz alocável possui um *status de alocação*. Quando a matriz é declarada, mas ainda não alocada, o seu status é *unallocated* ou *not currently allocated*. Quando a matriz aparece no comando `ALLOCATE`, o seu status passa a *allocated*; uma vez que ela é de-alocada, o seu status retorna a *not currently allocated*. Assim, o comando `ALLOCATE` somente pode ser usado em matrizes não correntemente alocadas, ao passo que o comando `DEALLOCATE` somente pode ser usado em matrizes alocadas; caso contrário, ocorre um erro.

É possível verificar se uma matriz está ou não correntemente alocada usando-se a função intrínseca `ALLOCATED`. Esta é uma função lógica com um argumento, o qual deve ser o nome de uma matriz alocável. Usando-se esta função, comandos como os seguintes são possíveis:

```
IF (ALLOCATED(A)) DEALLOCATE (A)
```

ou

```
IF (.NOT. ALLOCATED(A)) ALLOCATE (A(5,20))
```

Um terceiro status de alocação, possível no Fortran 90 mas não no Fortran 95, é o *undefined*, o qual ocorria quando uma matriz era alocada dentro de uma rotina e não era de-alocada antes de retornar ao programa que chamou a rotina. O que ocorria era que em subseqüentes chamadas desta rotina a matriz com o status *undefined* não mais podia ser utilizada. No Fortran 95, todas as matrizes alocáveis definidas em rotinas são automaticamente colocadas no status *unallocated* quando da saída da rotina. Contudo, um boa prática de programação consiste em sempre de-alocar todas as matrizes alocadas dentro de uma rotina.

Finalmente, há três restrições no uso de matrizes alocáveis:

1. Matrizes alocáveis não podem ser argumentos mudos de uma rotina e devem, portanto, ser alocadas e de-alocadas dentro da mesma unidade de programa (ver capítulo 8).

2. O resultado de uma função não pode ser uma matriz alocável (embora possa ser uma matriz).

3. Matrizes alocáveis não podem ser usadas na definição de um tipo derivado (esta restrição foi retirada no Fortran 2003.

O programa-exemplo a seguir ilustra o uso de matrizes alocáveis.

```
program testa_aloc
implicit none
integer, dimension(:,:), allocatable :: b
integer :: i,j,n= 2
!
print*, "Valor inicial de n:",n
allocate (b(n,n))
b= n
print*, "Valor inicial de B:"
print"(2(i2))", ((b(i,j), j= 1,n), i= 1,n)
deallocate (b)
n= n + 1
print*, "Segundo valor de n:",n
allocate (b(n,n))
b= n
print*, "Segundo valor de B:"
print"(3(i2))", ((b(i,j), j= 1,n), i= 1,n)
deallocate (b)
n= n + 1
print*, "Terceiro valor de n:",n
allocate (b(n+1,n+1))
b= n + 1
print*, "Terceiro valor de B:"
print"(5(i2))", ((b(i,j), j= 1,n+1), i= 1,n+1)
end program testa_aloc
```

6.10 Comando e construto FORALL

Quando um construto DO como o seguinte:

```
DO I= 1, N
   A(I,I)= X(I)   !A tem posto 2 e X tem posto 1.
END DO
```

é executado, o processador deve realizar cada iteração sucessiva em ordem, com uma atribuição após a outra. Isto representa um impedimento severo na otimização do código em uma plataforma paralelizada. Este problema foi enfocado com a construção do HPF (*High Performance Fortran*), o qual consiste em uma versão do Fortran 90 destinada a sistemas de computação paralela. Posteriormente, alguns avanços realizados pelo HPF, em relação ao Fortran 90, foram incorporados do padrão do Fortran 95; entre eles, a instrução FORALL. A idéia é oferecer ao programador uma estrutura que possibilite os mesmos recursos obtidos com laços DO, porém que possam ser automaticamente paralelizáveis, quando o programa estiver rodando em uma plataforma paralela.

6.10.1 Comando FORALL

As instruções do exemplo acima são implementadas através de um comando FORALL da seguinte maneira:

```
FORALL (I= 1:N) A(I,I)= X(I)
```

a qual especifica que as atribuições individuais sejam realizadas em qualquer ordem, inclusive simultaneamente, caso o programa rode em uma plataforma paralela. O comando FORALL pode ser visto como uma atribuição de matrizes expressa com a ajuda de índices. Neste exemplo em particular, deve-se notar que as atribuições não poderiam ser representadas de maneira simples através de uma atribuição de matrizes inteiras, tal como A= X, porque as matrizes A e X não têm a mesma forma. Outros exemplos do comando são:

```
FORALL (I= 1:N, J= 1:M) A(I,J)= I + J
FORALL (I= 1:N, J= 1:N, Y(I,J) /= 0.0) X(J,I)= 1.0/Y(I,J)
```

O primeiro exemplo poderia ser implementado com o uso da função intrínseca RESHAPE, mas o uso do FORALL possibilita a paralelização do programa executável. Já o segundo exemplo não poderia ser implementado com operação de matriz inteira (tal como X= 1.0/Y) porque, em primeiro lugar, a matriz X têm elementos transpostos em relação à matriz Y. Em segundo lugar, a máscara Y(I,J) /= 0.0 não está presente na operação de matriz inteira; somente é possível introduzir esta máscara com o uso do FORALL.

A forma geral do comando FORALL é a seguinte:

```
FORALL (<índice>= <menor>:<maior>[:<passo>] [,<índice>= <me-
nor>:<maior>[:<passo>]]  &
        [...] [, <expressão escalar lógica>]) <operações atribuições matrizes>
```

as condições impostas a <índice>, <menor>, <maior>, <passo> e <expressão escalar lógica> são as mesmas impostas ao construto FORALL e serão explicadas na descrição da forma geral do mesmo.

6.10.2 Construto FORALL

O construto FORALL também existe. Ele permite que diversas atribuições sejam executadas em ordem. Por exemplo, dadas as seguintes operações com sub-matrizes:

```
A(2:N-1, 2:N-1)= A(2:N-1, 1:N-2) + A(2:N-1, 3:N) &
               + A(1:N-2, 2:N-1) + A(3:N, 2:N-1)
B(2:N-1, 2:N-1)= A(2:N-1, 2:N-1)
```

também podem ser implementadas com o uso do construto FORALL:

```
FORALL (I= 2:N-1, J= 2:N-1)
   A(I,J)= A(I,J-1) + A(I, J+1) + A(I-1, J) + A(I+1, J)
   B(I,J)= A(I,J)
END FORALL
```

Neste exemplo, cada elemento de `A` é igualado à soma dos quatro vizinhos mais próximos e é feita então uma cópia em `B`. A versão com o `FORALL` é melhor legível e pode ser executado em qualquer ordem, inclusive simultaneamente. Devido a isto, quando o corpo do construto possui mais de uma instrução, o processador executa todas expressões e atribuições da primeira linha antes de passar a executar a instrução da linha seguinte. Não ocorre erro na atribuição, porque inicialmente as expressões são desenvolvidas, em qualquer ordem, seus resultados guardados em local de armazenagem temporária na memória e só no final as atribuições são feitas, também em qualquer ordem. A instrução em uma determinada linha é executada para todos os elementos e somente então o processamento passa para a linha seguinte.

Construtos `FORALL` podem ser encadeados. A seqüência

```
FORALL (I= 1:N-1)
   FORALL (J= I+1:N)
      A(I,J)= A(J,I)
   END FORALL
END FORALL
```

atribui a transposta do triângulo inferior de `A` ao triângulo superior de `A`.

Um construto `FORALL` pode conter um comando ou construto `WHERE`. Cada comando no corpo de um construto `WHERE` é executado em seqüência. Por exemplo,

```
FORALL (I= 1:N)
   WHERE (A(I,:) == 0) A(I,:)= I
   B(I,:)= I/A(I,:)
END FORALL
```

Aqui, cada elemento nulo de `A` é substituído pelo valor do índice das linhas e, seguindo a operação completa, os elementos das linhas de `B` são definidos como as recíprocas dos correspondentes elementos de `A` multiplicados pelo respectivo índice.

A sintaxe mais geral do construto `FORALL` é:

```
[<nome>:] FORALL (<índice>= <menor>:<maior>[:<passo>], &
          [, <índice>= <menor>:<maior>[:<passo>]] [...] &
          [, <expressão lógica escalar>])
             <corpo>
          END FORALL [<nome>]
```

onde `<índice>` é uma variável inteira escalar, a qual permanece válida somente dentro do `<corpo>` do construto, ou seja, outras variáveis podem ter o mesmo nome do `<índice>` mas estão separadas e não são acessíveis de dentro do `FORALL`.[2] O `<índice>` não pode ser redefinido dentro do construto. As expressões `<menor>`, `<maior>` e `<passo>` devem ser expressões escalares inteiras e formar uma seqüência de valores como para uma seção de matriz (seção 6.3); eles não podem fazer referência a um índice no mesmo cabeçalho, mas podem fazer referência a um índice de um `FORALL` mais externo. Uma vez tendo as expressões inteiras sido desenvolvidas, a `<expressão lógica escalar>`, se presente, é desenvolvida para cada combinação dos valores dos índices. Aqueles para os quais o resultado é `.TRUE.` são ativados em cada comando no `<corpo>` do construto.

O `<corpo>` consiste em um ou mais dos seguintes: expressões e atribuições de elementos de matrizes, comandos ou construtos `WHERE` e outros comandos ou construtos `FORALL`. Usualmente, o sub-objeto (sub-matriz, por exemplo) no lado esquerdo de uma atribuição no `<corpo>` fará referência a cada `<índice>` dos construtos nos quais ele está imerso, como parte da identificação do sub-objeto. Nenhum dos comandos no `<corpo>` pode conter uma ramificação ou ser acessível através de algum comando de desvio de fluxo, tal como o `GO TO`. Rotinas externas podem ser acessadas de dentro do `<corpo>` ou de dentro do cabeçalho do construto (na máscara, por exemplo). Todas as rotinas invocadas dentro desta estrutura devem ser puras (seção 8.2.16).

O programa-exemplo 6.3 (programa `tesforall`) ilustra uma aplicação simples do construto `FORALL`, enquanto que o programa 6.4 (programa `tesforall2`) calcula a transposta de uma matriz. Pode-se notar aqui que o comando `FORALL` não acarreta em erro, uma vez que as atribuições são inicialmente armazenadas em espaço temporário e somente no final são transferidas aos componentes da matriz `A`.

[2] O que é denominado de *âmbito (scope)*.

Programa 6.3: Exemplo simples do construto FORALL.

```
program tesforall
implicit none
integer, dimension(30) :: a,b
integer :: n
!
forall(n=1:30)
   a(n)=  n
   b(n)=  a(30-n+1)
end forall
print*, 'Vetor a:'
print*, a
print*, ''
print*, 'Vetor b:'
print*, b
end program tesforall
```

Programa 6.4: Calcula a transposta de uma matriz utilizando o comando FORALL.

```
program tesforall2
implicit none
integer, dimension(3,3) :: a
integer :: i,j
a= reshape(source= (/(i, i= 1,9)/), shape= (/3,3/))
print*, "Matriz A:"
print*, a(1,:)
print*, a(2,:)
print*, a(3,:)
forall (i= 1:3, j= 1:3) a(i,j)= a(j,i)
print*, "Transposta de A:"
print*, a(1,:)
print*, a(2,:)
print*, a(3,:)
end program tesforall2
```

Capítulo 7

Rotinas Intrínsecas

Em uma linguagem de programação destinada a aplicações científicas há uma exigência óbvia para que as funções matemáticas mais requisitadas sejam oferecidas como parte da própria linguagem, ao invés de terem de ser todas implementadas pelo programador. Além disso, ao serem fornecidas como parte do compilador, espera-se que estas funções tenham sido altamente otimizadas e testadas.

A eficiência das rotinas intrínsecas quando estas manipulam matrizes em computadores vetoriais ou paralelos deve ser particularmente marcante, porque uma única chamada à rotina deve causar um número grande de operações individuais sendo feitas simultaneamente, sendo o código que as implementa otimizado para levar em conta todos os recursos de hardware disponíveis.

No padrão do Fortran 90/95 há mais de cem rotinas intrínsecas ao todo. Elas se dividem em grupos distintos, os quais serão descritos em certo detalhe. Certos compiladores em particular poderão oferecer ainda mais rotinas, além das oferecidas pelo padrão da linguagem; contudo, o preço a pagar é a falta de portabilidade de programas que usam rotinas fora do grupo-padrão. Além disso, rotinas extras somente poderão ser oferecidas ao usuário através de *módulos* (capítulo 8).

Ao longo deste capítulo, as rotinas intrínsecas serão naturalmente divididas em funções e subrotinas. A distinção entre estas duas classes de rotinas será discutida com mais detalhes no capítulo 8.

7.1 Categorias de rotinas intrínsecas

Há quatro categorias de rotinas intrínsecas:

Rotinas elementais. São especificadas para argumentos escalares, mas podem também ser aplicadas a matrizes conformáveis. No caso de uma função elemental, cada elemento da matriz resultante, caso exista, será obtido como se a função tivesse sido aplicada a cada elemento individual da matriz que foi usada como argumento desta função. No caso de uma subrotina elemental com um argumento matricial, cada argumento com intento `IN` ou `INOUT` deve ser uma matriz e cada elemento resultante será obtido como se a subrotina tivesse sido aplicada a cada elemento da matriz que é passada como argumento à subrotina.

Funções inquisidoras. Retornam propriedades dos seus argumentos principais que não dependem dos seus valores, somente do seu tipo e/ou espécie.

Funções transformacionais. São funções que não são nem elementais nem inquisidoras. Elas usualmente têm argumentos matriciais e o resultado também é uma matriz cujos elementos dependem dos elementos dos argumentos.

Rotinas não-elementais. Rotinas que não se enquadram em nenhum dos tipos acima.

As rotinas intrínsecas serão descritas posteriormente com base nesta categorização.

7.2 Declaração e atributo `INTRINSIC`

Um nome pode ser especificado como o nome de uma rotina intrínseca com a declaração `INTRINSIC`, o qual possui a forma geral:

```
INTRINSIC <lista de nomes intrínsecos>
```

onde `<lista de nomes intrínsecos>` é uma lista de nomes de rotinas intrínsecas. O uso desta declaração é recomendado quando se quer enfatizar ao compilador que certos nomes são destinados a rotinas intrínsecas. Isto pode ser útil quando o programador está estendendo a definição de uma rotina intrínseca, em um procedimento denominado *sobrecarga de operador (operator overloading)*, o qual faz uso de *interfaces genéricas (generic interfaces)*. Para mais detalhes nestes recursos avançados, ver Metcalf & Reid (1996) [6]. Alternativamente, uma ou mais funções intrínsecas podem ser declaradas com o atributo `INTRINSIC`, no lugar da declaração.

O atributo ou declaração `INTRINSIC` são excludentes em relação ao atributo ou declaração `EXTERNAL` (seção 8.2.12).

7.3 Funções inquisidoras de qualquer tipo

As seguintes são funções inquisidoras, com argumentos que podem ser de qualquer tipo:

`ASSOCIATED(POINTER[,TARGET]).` Quanto `TARGET` está ausente, a função retorna o valor `.TRUE.` se o *ponteiro (pointer)* está associado com um *alvo (target)* e retorna o valor `.FALSE.` em caso contrário. O status de associação de ponteiro de `POINTER` não deve ser indefinido. Se `TARGET` estiver presente, este deve ter o mesmo tipo, espécie e posto de `POINTER`. O valor da função é `.TRUE.` se `POINTER` estiver associado a `TARGET` ou `.FALSE.` em caso contrário. No caso de matrizes, `.TRUE.` é obtido somente se as formas são idênticas e elementos de matriz correspondentes, na ordem de elementos de matriz, estão associadas umas com as outras. Se o comprimento do caractere ou tamanho da matriz são iguais a zero, `.FALSO.` é obtido. Um limite diferente, como no caso de `ASSOCIATED(P,A)` seguindo a atribuição de ponteiro `P => A(:)` quando `LBOUND(A)= 0`, é insuficiente para causar um resultado `.FALSE.`. `TARGET` pode ser também um ponteiro, em cujo caso seu alvo é comparado com o alvo de `POINTER`; o status de associação de ponteiro de `TARGET` não deve ser indefinido e se ou `POINTER` ou `TARGET` estão dissociados, o resultado é `.FALSE.`

`PRESENT(A).` Pode ser chamada em um sub-programa que possui um argumento mudo opcional `A` (ver seção 8.2.9) ou acesse este argumento mudo a partir de seu hospedeiro. A função retorna o valor `.TRUE.` se o argumento está presente na chamada do sub-programa ou `.FALSE.` em caso contrário. Se um argumento mudo opcional é usado como um argumento real na chamada de outra rotina, ele é considerado também ausente pela rotina chamada.

`KIND(X).` Tem o argumento `X` de qualquer tipo e espécie e o seu resultado tem o tipo inteiro padrão e de valor igual ao valor do parâmetro de espécie de `X`.

7.4 Funções elementais numéricas

Há 17 funções elementais destinadas à realização de operações numéricas simples, muitas das quais realizam conversão de tipo de variáveis para alguns ou todos os argumentos.

7.4.1 Funções elementais que podem converter

Se o especificador `KIND` estiver presente nas funções elementais seguintes, este deve ser uma expressão inteira e fornecer um parâmetro de espécie de tipo que seja suportado pelo processador.

`ABS(A).` Retorna o valor absoluto de um argumento dos tipos inteiro, real ou complexo. O resultado é do tipo inteiro se `A` for inteiro e é real nos demais casos. O resultado é da mesma espécie que `A`.

`AIMAG(Z).` Retorna a parte imaginária do valor complexo `Z`. O tipo do resultado é real e a espécie é a mesma que `Z`.

`AINT(A[,KIND]).` Trunca um valor real `A` em sentido ao zero para produzir um valor real cujos valores decimais são iguais a zero. Se o argumento `KIND` estiver presente, o resultado é da espécie dada por este; caso contrário, retorna o resultado na espécie padrão do tipo real.

`ANINT(A[,KIND]).` Retorna um real cujo valor é o número completo (sem parte fracionária) mais próximo de `A`. Se `KIND` estiver presente, o resultado é do tipo real da espécie definida; caso contrário, o resultado será real da espécie padrão.

`CEILING(A)`. Retorna o menor inteiro maior ou igual ao seu argumento real. Se `KIND` estiver presente, o resultado será do tipo inteiro com a espécie definida pelo parâmetro; caso contrário, o resultado será inteiro da espécie padrão.

`CMPLX(X[,Y][,KIND])`. Converte `X` ou `(X,Y)` ao tipo complexo com a espécie definida pelo argumento `KIND`, caso presente, ou na espécie padrão do tipo complexo em caso contrário. Se `Y` for ausente, `X` pode ser dos tipos inteiro, real ou complexo. Se `Y` estiver presente, tanto `X` quanto `Y` devem ser dos tipos inteiro ou real.

`FLOOR(A[,KIND])`. Retorna o maior inteiro menor ou igual a seu argumento real `A`. Se `KIND` estiver presente, o resultado é do tipo inteiro da espécie definida; caso contrário, o resultado é inteiro da espécie padrão.

`INT(A[,KIND])`. Converte ao tipo inteiro da espécie padrão ou dada pelo argumento `KIND`. O argumento `A` pode ser:

- inteiro, em cujo caso `INT(A)= A;`
- real, em cujo caso o valor é truncado em sentido ao zero, ou
- complexo, em cujo caso somente a parte real é considerada e esta é truncada em sentido ao zero.

`NINT(A[,KIND])`. Retorna o valor inteiro mais próximo do real `A`. Se `KIND` estiver presente, o resultado é do tipo inteiro da espécie definida; caso contrário, o resultado será inteiro da espécie padrão.

`REAL(A[,KIND])`. Converte ao tipo real da espécie dada pelo argumento `KIND`. O argumento `A` pode ser dos tipos inteiro, real ou complexo. Se `A` for complexo, a parte imaginária é ignorada. Se o argumento `KIND` estiver ausente, o resultado será da espécie padrão do tipo real se `A` for inteiro ou real e da mesma espécie de `A` se este for complexo.

7.4.2 Funções elementais que não convertem

As seguintes são funções elementais cujos resultados são do tipo e da espécie iguais aos do primeiro ou único argumento. Para aquelas que possuem mais de um argumento, todos devem ser do mesmo tipo e espécie.

`CONJG(Z)`. Retorna o conjugado do valor complexo `Z`.

`DIM(X,Y)`. Retorna `MAX(X-Y,0)` para argumentos que são ambos inteiros ou ambos reais.

`MAX(A1,A2[,A3,...])`. Retorna o máximo (maior valor) entre dois ou mais valores inteiros ou reais.

`MIN(A1,A2[,A3,...])`. Retorna o mínimo (menor valor) entre dois ou mais valores inteiros ou reais.

`MOD(A,P)`. Retorna o restante de `A` módulo `P`, ou seja, `A-INT(A/P)*P`. Se `P=0`, o resultado depende do processador. `A` e `P` devem ambos ser inteiros ou ambos reais.

`MODULO(A,P)`. Retorna `A` módulo `P` quando `A` e `P` são ambos inteiros ou ambos reais; isto é, `A-FLOOR(A/P)*P` no caso real ou `A-FLOOR(A÷P)*P` no caso inteiro, onde ÷ representa divisão matemática ordinária. Se `P=0`, o resultado depende do processador.

`SIGN(A,B)`. Retorna o valor absoluto de `A` vezes o sinal de `B`. `A` e `B` devem ser ambos inteiros ou ambos reais. Se `B= 0` e é inteiro, seu sinal é assumido positivo. Se o processador não tem a capacidade de distinguir entre zeros reais positivos ou negativos então, para `B= 0.0` o sinal é assumido positivo.

7.5 Funções elementais matemáticas

As seguintes são funções elementais que calculam o conjunto imagem de funções matemáticas elementares. O tipo e espécie do resultado são iguais aos do primeiro argumento, o qual, usualmente, é o único.

`ACOS(X)`. Retorna a função arco cosseno (ou $\cos^{-1}$) para valores reais do argumento `X` ($|\texttt{X}| \leq 1$). O resultado é obtido em radianos no intervalo $0 \leq \texttt{ACOS(X)} \leq \pi$.

`ASIN(X)`. Retorna a função arco seno (ou $\sin^{-1}$) para valores reais do argumento `X` ($|\texttt{X}| \leq 1$). O resultado é obtido em radianos no intervalo $-\pi/2 \leq \texttt{ASIN(X)} \leq \pi/2$.

`ATAN(X)`. Retorna a função arco tangente (ou $\tan^{-1}$) para valores reais do argumento `X` ($-\infty < \texttt{X} < \infty$). O resultado é obtido em radianos no intervalo $-\pi/2 \leq \texttt{ATAN(X)} \leq \pi/2$.

`ATAN2(Y,X)`. Retorna a função arco tangente (ou $\tan^{-1}$) para pares de argumentos reais, `X` e `Y`, ambos do mesmo tipo e espécie. O resultado é o valor principal do argumento do número complexo `(X,Y)`, expresso em radianos no intervalo $-\pi < \texttt{ATAN2(Y,X)} \leq \pi$. Os valores de `X` e `Y` não devem ser simultaneamente nulos.

`COS(X)`. Retorna o valor da função cosseno para um argumento dos tipos real ou complexo. A unidade do argumento `X` é suposta ser radianos.

`COSH(X)`. Retorna o valor da função cosseno hiperbólico para um argumento `X` real.

`EXP(X)`. Retorna o valor da função exponencial para um argumento `X` real ou complexo.

`LOG(X)`. Retorna o valor da função logaritmo natural para um argumento `X` real ou complexo. No caso real, `X` deve ser positivo. No caso complexo, `X` não pode ser nulo e a parte imaginária do resultado ($\texttt{Z}_i$) está no intervalo $-\pi < \texttt{Z}_i \leq \pi$.

`LOG10(X)`. Retorna o valor da função logaritmo de base 10 para um argumento `X` real e positivo.

`SIN(X)`. Retorna o valor da função seno para um argumento dos tipos real ou complexo. A unidade do argumento `X` é suposta ser radianos.

`SINH(X)`. Retorna o valor da função seno hiperbólico para um argumento `X` real.

`SQRT(X)`. Retorna o valor da função raiz quadrada para um argumento `X` real ou complexo. No caso real, `X` não pode ser negativo. No caso complexo, o resultado consiste na raiz principal, ou seja, a parte real do resultado é positiva ou nula. Quando a parte real do resultado for nula, a parte imaginária é positiva ou nula.

`TAN(X)`. Retorna o valor da função tangente para um argumento `X` real. A unidade do argumento é suposta ser radianos.

`TANH(X)`. Retorna o valor da função tangente hiperbólica para um argumento `X` real.

7.6 Funções elementais lógicas e de caracteres

7.6.1 Conversões caractere-inteiro

As seguintes são funções elementais para converter um único caractere em um inteiro e vice-versa.

`ACHAR(I)`. O resultado é do tipo de caractere padrão de comprimento um e retorna o caractere que está na posição especificada pelo valor inteiro `I` na tabela ASCII de caracteres (tabela 4.9). `I` deve estar no intervalo $0 \leq \texttt{I} \leq 127$; caso contrário, o resultado depende do processador.

`CHAR(I[,KIND])`. O resultado é do tipo caractere de comprimento um. A espécie é dada pelo argumento opcional `KIND`, se presente ou, em caso contrário, será a espécie padrão. A função retorna o caractere na posição `I` (tipo inteiro) da sequência de caracteres interna do processador associada com o parâmetro de espécie relevante. `I` deve estar no intervalo $0 \leq \texttt{I} \leq n-1$, onde n é o número de caracteres na sequência interna do processador.

`IACHAR(C)`. O resultado é do tipo inteiro padrão e retorna a posição do caractere `C` na tabela ASCII. Se `C` não está na tabela, o resultado depende do processador.

`ICHAR(C)`. O resultado é do tipo inteiro padrão e retorna a posição do caractere `C` na sequência interna do processador associada com a espécie de `C`.

7.6.2 Funções de comparação léxica

As seguintes funções elementais aceitam strings de caracteres da espécie padrão, realizam uma comparação léxica baseada na tabela ASCII e retornam um resultado lógico da espécie padrão. Se as strings tiverem comprimentos distintos, a mais curta é complementada com espaços em branco.

`LGE(STRING_A,STRING_B).` Retorna o valor `.TRUE.` se `STRING_A` segue `STRING_B` na sequência estabelecida pela tabela ASCII, ou a iguala. O resultado é `.FALSE.` em caso contrário.

`LGT(STRING_A,STRING_B).` Retorna o valor `.TRUE.` se `STRING_A` segue `STRING_B` na sequência estabelecida pela tabela ASCII. O resultado é `.FALSE.` em caso contrário.

`LLE(STRING_A,STRING_B).` Retorna o valor `.TRUE.` se `STRING_A` precede `STRING_B` na sequência estabelecida pela tabela ASCII, ou a iguala. O resultado é `.FALSE.` em caso contrário.

`LLT(STRING_A,STRING_B).` Retorna o valor `.TRUE.` se `STRING_A` precede `STRING_B` na sequência estabelecida pela tabela ASCII. O resultado é `.FALSE.` em caso contrário.

7.6.3 Funções elementais para manipulações de strings

As seguintes são funções elementais que manipulam strings. Os argumentos `STRING`, `SUBSTRING` e `SET` são sempre do tipo de caractere e, quando dois estão presentes, ambos devem ser da mesma espécie. O resultado é da mesma espécie de `STRING`.

`ADJUSTL(STRING).` Ajusta uma string à esquerda. Ou seja, remove os espaços em branco no início da string e coloca o mesmo número de brancos no final da string.

`ADJUSTR(STRING).` Ajusta uma string à direita. Ou seja, remove os espaços em branco no final da string e coloca o mesmo número de brancos no início da string.

`INDEX(STRING,SUBSTRING[,BACK]).` Retorna um valor do tipo inteiro padrão, o qual consiste na posição inicial de SUBSTRING como uma substring de `STRING`, ou zero se tal substring não existe. Se `BACK` (variável lógica) está ausente, ou se está presente com o valor `.FALSE.`, a posição inicial da primeira das substrings é retornada; o valor 1 é retornado se `SUBSTRING` tem comprimento zero. Se `BACK` está presente com valor `.TRUE.`, a posição inicial da última das substrings é retornada; o valor `LEN(STRING)+1` é retornado se `SUBSTRING` tem comprimento zero.

`LEN_TRIM(STRING).` Retorna um valor inteiro padrão correspondente ao comprimento de `STRING`, ignorando espaços em branco no final do argumento.

`SCAN(STRING,SET[,BACK]).` Retorna um valor inteiro padrão correspondente à posição de um caractere de `STRING` que esteja em `SET`, ou zero se não houver tal caractere. Se a variável lógica `BACK` está ausente, ou presente com valor `.FALSE.`, a posição mais à esquerda deste caractere é retornada. Se `BACK` está presente com valor `.TRUE.`, a posição mais à direita deste caractere é retornada.

`VERIFY(STRING,SET[,BACK]).` Retorna o valor inteiro padrão 0 se cada caractere em `STRING` aparece em `SET`, ou a posição de um caractere de `STRING` que não esteja em `SET`. Se a variável lógica `BACK` está ausente, ou presente com valor `.FALSE.`, a posição mais à esquerda de tal caractere é retornada. Se `BACK` está presente com valor `.TRUE.`, a posição mais à direita de tal caractere é retornada.

7.6.4 Conversão lógica

A função elemental a seguir converte de uma espécie do tipo lógico em outra.

`LOGICAL(L[,KIND]).` Retorna o valor lógico idêntico ao valor de `L`. A espécie do resultado é definida pelo argumento opcional `KIND`, caso esteja presente, ou é da espécie padrão em caso contrário.

7.7 Funções não-elementais para manipulação de strings

7.7.1 Função inquisidora para manipulação de strings

`LEN(STRING).` Trata-se de uma função inquisidora que retorna um valor inteiro padrão escalar que corresponde ao número de caracteres em `STRING` se esta é escalar ou de um elemento de `STRING` se a mesma é uma matriz. O valor de `STRING` não precisa estar definido.

7.7.2 Funções transformacionais para manipulação de strings

Existem duas funções que não são elementais porque o comprimento do resultado depende do valor de um argumento.

`REPEAT(STRING,NCOPIES).` Forma a string que consiste na concatenação de NCOPIES cópias de STRING, onde `NCOPIES` é do tipo inteiro e seu valor não deve ser negativo. Ambos argumentos devem ser escalares.

`TRIM(STRING).` Retorna `STRING` com todos os espaços em branco no final da variável removidos. `STRING` deve ser escalar.

7.8 Funções inquisidoras e de manipulações numéricas

7.8.1 Modelos para dados inteiros e reais

As funções inquisidoras e de manipulações numéricas são definidas em termos de modelos de representação de números inteiro e reais para cada espécie suportada pelo processador.

Para cada espécie do tipo inteiro, o modelo é:

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}, \tag{7.1}$$

onde $s = \pm 1$, q é um inteiro positivo, r é um inteiro maior que um (usualmente 2) e cada valor de w_k é um inteiro no intervalo $0 \leq w_k < r$.

Para cada espécie do tipo real, o modelo é:

$$x = 0$$

e

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, \tag{7.2}$$

onde $s = \pm 1$, p e b são inteiros maiores que um, e é um inteiro no intervalo $e_{\min} \leq e \leq e_{\max}$ e cada f_k é um inteiro no intervalo $0 \leq f_k < b$, exceto f_1, que é não nulo.

Os valores de todos os parâmetros nestes modelos são escolhidos para o processador de tal forma que o modelo melhor se ajuste ao hardware, desde que todos os números sejam representáveis.

7.8.2 Funções numéricas inquisidoras

Há nove funções inquisidoras que retornam valores para os modelos de dados associados com seus argumentos. Cada função tem um único argumento que pode ser escalar ou matricial e todas retornam um valor escalar. O valor *per se* do argumento não precisa ser definido, somente o seu tipo e espécie.

`DIGITS(X).` Para `X` real ou inteiro, retorna o inteiro padrão cujo valor é o número de dígitos significantes no modelo que inclui `X`; isto é, retorna p para `X` real ou q para `X` inteiro.

`EPSILON(X).` Para `X` real, retorna um resultado real com o mesmo tipo de `X` e que é quase indistinguível do valor `1.0` no modelo que inclui `X`. Ou seja, a função calcula b^{1-p}.

`HUGE(X).` Para `X` real ou inteiro, retorna o maior valor representável no modelo que inclui `X`. O resultado possui o mesmo tipo e espécie de `X`. O valor é

$$\left(1 - b^{-p}\right) b^{e_{\max}}$$

para `X` real ou

$$r^q - 1$$

para `X` inteiro.

`MAXEXPONENT(X).` Para `X` real, retorna o inteiro padrão $e_{\max}$, ou seja, o expoente máximo no modelo que inclui `X`.

`MINEXPONENT(X)`. Para `X` real, retorna o inteiro padrão $e_{\min}$, ou seja, o expoente mínimo no modelo que inclui `X`.

`PRECISION(X)`. Para `X` real ou complexo, retorna um inteiro padrão que contém a precisão decimal equivalente no modelo que representa números reais da mesma espécie de `X`. O valor da função é $\texttt{INT}((p-1)) * \texttt{LOG10}(b))+k$, onde k é 1 se b é uma potência inteira de 10 ou 0 em outro caso.

`RADIX(X)`. Para `X` real ou inteiro, retorna o inteiro padrão que é a base no modelo que inclui `X`. Isto é, retorna b para `X` real ou r para `X` inteiro.

`RANGE(X)`. Para `X` inteiro, real ou complexo, retorna o inteiro padrão que contém o intervalo de expoentes decimais nos modelos representando números reais ou inteiro da mesma espécie de `X`. O valor da função é $\texttt{INT}(\texttt{LOG10}(huge))$ para `X` inteiro e

$$\texttt{INT}(\texttt{MIN}(\texttt{LOG10}(huge), -\texttt{LOG10}(tiny)))$$

para `X` real, onde *huge* e *tiny* são, respectivamente, o maior e o menor números positivos nos modelos.

`TINY(X)`. Para `X` real, retorna o menor número positivo,

$$b^{e_{\min}-1}$$

no modelo que inclui `X`. O resultado é do mesmo tipo e espécie de `X`.

7.8.3 Funções elementais que manipulam quantidades reais

Há sete funções elementais cujo primeiro ou único argumento é do tipo real e que retorna valores relacionados aos componentes dos modelos de valores associados ao valor do argumento.

`EXPONENT(X)`. Retorna o inteiro padrão cujo valor é a parte de expoente e de `X` quando representado como um número de modelo. Se `X=0`, o resultado também é nulo.

`FRACTION(X)`. Retorna uma quantidade real da mesma espécie que `X` e cujo valor é a parte fracionária de `X` quando representado como um número de modelo; isto é, a função retorna $\texttt{X}b^{-e}$.

`NEAREST(X,S)`. Retorna uma quantidade real da mesma espécie que `X` e cujo valor é o número distinto mais próximo de `X` no sentido dado pelo sinal da quantidade real `S`. O valor de `S` não pode ser nulo.

`RRSPACING(X)`. Retorna uma quantidade real da mesma espécie que `X` cujo valor é a recíproca do espaçamento relativo dos números de modelo próximos a `X`; isto é, a função retorna $|\texttt{X}b^{-e}|\, b^{p}$.

`SCALE(X,I)`. Retorna uma quantidade real da mesma espécie que `X` cujo valor é $\texttt{X}b^{\texttt{I}}$, onde b é a base do modelo para `X` e `I` é do tipo inteiro.

`SET_EXPONENT(X,I)`. Retorna uma quantidade real da mesma espécie que `X` cuja parte fracionária é a parte fracionária da representação de `X` e cuja parte exponencial é `I`; isto é, a função retorna $\texttt{X}b^{1-e}$.

`SPACING(X)`. Retorna uma quantidade real da mesma espécie que `X` cujo valor é o espaçamento absoluto do modelo de números próximos a `X`. O resultado é b^{e-p} se `X` é não nulo e este resultado está dentro do intervalo de números representáveis. Caso contrário, a função retorna `TINY(X)`.

7.8.4 Funções transformacionais para valores de espécie *(kind)*

Há duas funções que retornam o menor valor do parâmetro de espécie que irá satisfazer um dado requerimento numérico. As funções têm argumentos e resultados escalares; porém são classificadas como transformacionais. Estas funções já foram discutidas na seção 3.7.3.

`SELECTED_INT_KIND(R)`. Retorna o inteiro escalar padrão que é o valor do parâmetro da espécie para um dado do tipo inteiro capaz de representar todos os valor inteiros n no intervalo $-10^{\texttt{R}} < n < 10^{\texttt{R}}$, onde `R` é um inteiro escalar. Se mais de uma espécie for disponível, a espécie com menor intervalo exponencial é escolhida. Se nenhuma espécie é disponível, o resultado é `-1`.

`SELECTED_REAL_KIND([P][,R])`. Retorna o inteiro escalar padrão que é o valor do parâmetro da espécie para um dado do tipo real com precisão decimal (conforme retornada pela função `PRECISION`) no mínimo igual a `P` e intervalo de expoente decimal (conforme retornado pela função `RANGE`) no mínimo igual a `R`. Se mais de uma espécie for disponível, a espécie com a menor precisão decimal é escolhida. Tanto `P` quanto `R` são inteiros escalares; pelo menos um destes deve estar presente. Se não houver espécie disponível, o resultado é:

`-1`: se a precisão requerida não for disponível;

`-2`: se um intervalo de expoente grande o suficiente não for disponível;

`-3`: se ambos não forem disponíveis.

7.9 Rotinas de manipulação de bits

Há onze rotinas para manipular bits contidos em quantidades inteiras. Estas rotinas são elementais, quando apropriado. As rotinas são baseadas em um modelo no qual um valor inteiro é representado por s bits w_k, com $k = 0, 1, \ldots, s-1$, em uma sequência da direita para a esquerda, baseada no valor não-negativo

$$\sum_{k=0}^{s-1} w_k \times 2^k.$$

Este modelo é válido somente no contexto destas rotinas intrínsecas e é idêntico ao modelo de números inteiros (7.1) quando r é uma potência inteira de 2 e $w_{s-1} = 0$; mas quando $w_{s-1} = 1$ os modelos diferem.

7.9.1 Função inquisidora

`BIT_SIZE(I)`. Retorna o número de bits no modelo para bits dentro de um inteiro da mesma espécie que `I`. O resultado é um inteiro escalar da mesma espécie que `I`.

7.9.2 Funções elementais

`BTEST(I,POS)`. Retorna o valor lógico da espécie padrão `.TRUE.` se o bit `POS` do inteiro `I` tem valor 1 e retorna `.FALSE.` em outra situação. `POS` deve ser um inteiro com valor no intervalo $0 \leq$ `POS` $<$ `BIT_SIZE(I)`.

`IAND(I,J)`. Retorna o lógico `AND` de todos os bits em `I` e bits correspondentes em `J`, de acordo com a tabela-verdade

`I`	1 1 0 0
`J`	1 0 1 0
`I AND(I, J)`	1 0 0 0

`I` e `J` devem ser do mesmo tipo; o resultado também será do mesmo tipo.

`IBCLR(I,POS)`. Retorna um inteiro do mesmo tipo que `I` e valor igual ao de `I` exceto que o bit `POS` é levado a 0. `POS` deve ser um inteiro com valor no intervalo $0 \leq$ `POS` $<$ `BIT_SIZE(I)`.

`IBITS(I,POS,LEN)`. Retorna um inteiro do mesmo tipo que `I` e valor igual aos `LEN` bits de `I` iniciando no bit `POS` ajustado à direita e com todos os outros bits iguais a 0. `POS` e `LEN` devem ser inteiros positivos tais que `POS` $+$ `LEN` $\leq$ `BIT_SIZE(I)`.

`IBSET(I,POS)`. Retorna um inteiro do mesmo tipo que `I` e valor igual ao de `I` exceto que o bit `POS` é levado a 1. `POS` deve ser um inteiro com valor no intervalo $0 \leq$ `POS` $<$ `BIT_SIZE(I)`.

`IEOR(I,J)`. Retorna o `OU` lógico exclusivo de todos os bits de `I` e correspondentes bits em `J`, de acordo com a tabela-verdade

`I`	1 1 0 0
`J`	1 0 1 0
`I EOR(I, J)`	0 1 1 0

`I` e `J` devem ser do mesmo tipo; o resultado também será do mesmo tipo.

`IOR(I,J)`. Retorna o `OU` lógico inclusivo de todos os bits de `I` e correspondentes bits em `J`, de acordo com a tabela-verdade

`I`	1 1 0 0
`J`	1 0 1 0
`IOR(I,J)`	1 1 1 0

`I` e `J` devem ser do mesmo tipo; o resultado também será do mesmo tipo.

`ISHFT(I,SHIFT)`. Retorna um inteiro do mesmo tipo que `I` e valor igual ao de `I` exceto que os bits são deslocados `SHIFT` posições para a esquerda (`-ISHIFT` desloca para a direita se `SHIFT` for positivo). Zeros são inseridos a partir da extremidade oposta. `SHIFT` deve ser um inteiro com valor que satisfaz a inegualdade $|\texttt{SHIFT}| \leq \texttt{BIT\_SIZE(I)}$.

`ISHFTC(I,SHIFT[,SIZE])`. Retorna um inteiro do mesmo tipo que `I` e valor igual ao de `I` exceto que os `SIZE` bits mais à direita (ou todos os bits se `SIZE` for ausente) são deslocados de forma circular `SHIFT` posições para a esquerda (`-ISHIFT` desloca para a direita se `SHIFT` for positivo). Zeros são inseridos a partir da extremidade oposta. `SHIFT` deve ser um inteiro com valor que não excede o valor de `SIZE` ou `BIT_SIZE(I)`, se `SIZE` estiver ausente.

`NOT(I)`. Retorna o complemento lógico de todos os bits em `I`, de acordo com a tabela-verdade

`I`	0 1
`NOT(I)`	1 0

7.9.3 Subrotina elemental

`MVBITS(FROM,FROMPOS,LEN,TO,TOPOS)`. Copia a sequência de bits em `FROM` que inicia na posição `FROMPOS` e tem comprimento `LEN` para `TO`, iniciando na posição `TOPOS`. Os outros bits de `TO` permanecem inalterados. `FROM`, `FROMPOS`, `LEN` e `TOPOS` são todos inteiros com intent `IN` e devem ter valores que satisfazem as inegualdades: $\texttt{LEN} \geq 0$, $\texttt{FROMPOS+LEN} \leq \texttt{BIT\_SIZE(FROM)}$, $\texttt{FROMPOS} \geq 0$, $\texttt{TOPOS} \geq 0$ e $\texttt{TOPOS+LEN} \leq \texttt{BIT\_SIZE(TO)}$. `TO` é um inteiro com intent `INOUT`; ele deve ser da mesma espécie que `FROM`. A mesma variável pode ser especificada para `FROM` e `TO`.

7.10 Função de transferência

A função de transferência permite que dados de um tipo sejam transferidos a outro tipo sem que a representação física dos mesmos seja alterada. Esta função pode ser útil, por exemplo, ao se escrever um sistema de armazenamento e recuperação de dados; o sistema pode ser escrito para uma dado tipo, por exemplo, inteiro, e os outros tipos são obtidos através de transferências de e para este tipo.

`TRANSFER(SOURCE,MOLD[,SIZE])`. Retorna um valor do tipo e espécie de `MOLD`. Quando `SIZE` estiver ausente, o resultado é escalar se `MOLD` for escalar e é de posto 1 e tamanho suficiente para armazenar toda a `SOURCE` se `MOLD` for uma matriz. Quando `SIZE` estiver presente, o resultado é de posto 1 e tamanho `SIZE`. Se a representação física do resultado é tão ou mais longa que o tamanho de `SOURCE`, o resultado contém `SOURCE` como sua parte inicial e o resto é indefinido; em outros casos, o resultado é a parte inicial de `SOURCE`. Como o posto do resultado depende se `SIZE` é ou não especificado, o argumento em si não pode ser um argumento opcional.

7.11 Funções de multiplicação vetorial ou matricial

Há duas funções transformacionais que realizam multiplicações vetorial ou matricial. Elas possuem dois argumentos que são ambos de um tipo numérico (inteiro, real ou complexo) ou ambas do tipo lógico. O resultado é do mesmo tipo e espécie como seria resultante das operações de multiplicação ou `.AND.` entre dois escalares. As funções `SUM` e `ANY`, usadas nas definições abaixo, são definidas na seção 7.12.1 abaixo.

`DOT_PRODUCT(VETOR_A,VETOR_B)`. Requer dois argumentos, ambos de posto 1 e do mesmo tamanho. Se `VETOR_A` é dos tipos inteiro ou real, a função retorna `SUM(VETOR_A*VETOR_B)`; se `VETOR_A` é do tipo complexo, a função retorna `SUM(CONJG(VETOR_A)*VETOR_B)`; e se `VETOR_A` é do tipo lógico, a função retorna `ANY(VETOR_A .AND. VETOR_B)`.

`MATMUL(MATRIZ_A,MATRIZ_B)`. Realiza a multiplicação escalar. Para argumentos numéricos, três casos são possíveis:

1. `MATRIZ_A` tem forma `(/n,m/)` e `MATRIZ_B` tem forma `(/m,k/)`. O resultado tem forma `(/n,k/)` e o elemento (i, j) tem o valor dado por `SUM(MATRIZ_A(I,:)*MATRIZ_B(:,J))`.
2. `MATRIZ_A` tem forma `(/m/)` e `MATRIZ_B` tem forma `(/m,k/)`. O resultado tem forma `(/k/)` e o elemento j tem o valor dado por `SUM(MATRI_A*MATRIZ_B(:,J))`.
3. `MATRIZ_A` tem forma `(/n,m/)` e `MATRIZ_B` tem forma `(/m/)`. O resultado tem forma `(/n/)` e o elemento i tem o valor dado por `SUM(MATRIZ_A(I,:)*MATRIZ_B)`.

Para argumentos lógicos, as formas são as mesmas que para argumentos numéricos e os valores são determinados pela substituição da função `SUM` e do operador "`*`" pela função `ANY` e pelo operador `.AND.`.

7.12 Funções transformacionais que reduzem matrizes

Há sete funções transformacionais que executam operações em matrizes, tais como somar seus elementos.

7.12.1 Caso de argumento único

Nas suas formas mais simples, estas funções têm um único argumento matricial e o resultado é um valor escalar. Todas, exceto `COUNT` tem o resultado do mesmo tipo e espécie que o argumento.

`ALL(MASK).` Retorna o valor `.TRUE.` se todos os elementos da matriz lógica `MASK` forem verdadeiros ou se `MASK` for nula; caso contrário, retorna o valor `.FALSE.`

`ANY(MASK).` Retorna o valor `.TRUE.` se algum dos elementos da matriz lógica `MASK` for verdadeiro e o valor `.FALSE.` se todos os elementos forem falsos ou se a matriz for nula.

`COUNT(MASK).` Retorna o valor inteiro padrão igual ao número de elementos da matriz `MASK` que têm o valor `.TRUE.`

`MAXVAL(ARRAY).` Retorna o valor máximo dentre os elementos da matriz `ARRAY`, a qual pode ser inteira ou real. Se `ARRAY` for nula, a função retorna o valor negativo de maior magnitude suportado pelo processador.

`MINVAL(ARRAY).` Retorna o valor mínimo dentre os elementos da matriz `ARRAY`, a qual pode ser inteira ou real. Se `ARRAY` for nula, a função retorna o valor positivo de maior magnitude suportado pelo processador.

`PRODUCT(ARRAY).` Retorna o produto de todos os elementos de uma matriz inteira, real ou complexa. A função retorna o valor 1 se `ARRAY` for nula.

`SUM(ARRAY).` Retorna a soma de todos os elementos de uma matriz inteira, real ou complexa. A função retorna o valor 0 se `ARRAY` for nula.

7.12.2 Argumento opcional `DIM`

Todas as funções acima têm um segundo argumento opcional `DIM`, o qual é um inteiro escalar. Se este argumento está presente, a operação é aplicada a todas as seções de posto 1 que varrem através da dimensão `DIM` para produzir uma matriz de posto reduzido por um e extensões iguais às extensões nas outras dimensões. Por exemplo, se `A` é uma matriz real de forma `(/4,5,6/)`, `SUM(A,DIM=2)` é uma matriz real de forma `(/4,6/)` e o seu elemento (i, j) tem o valor dado por `SUM(A(I,:,J))`.

7.12.3 Argumento opcional `MASK`

As funções `MAXVAL`, `MINVAL`, `PRODUCT` e `SUM` têm um terceiro argumento opcional; uma matriz lógica `MASK`. Se esta matriz estiver presente, ela deve ter a mesma forma que o primeiro argumento e a operação é aplicada aos elementos do primeiro argumento correspondentes aos elementos verdadeiros de `MASK`. Por exemplo, `SUM(A, MASK= A>0)` soma todos os elementos positivos da matriz `A`. O argumento `MASK` afeta somente o valor da função e não afeta o desenvolvimento de argumentos que são expressões matriciais.

7.13 Funções inquisidoras de matrizes

Há cinco funções que inquirem sobre os limites, forma, tamanho e status de alocação de uma matriz de qualquer tipo. Uma vez que o resultado depende somente nas propriedades da matriz, o valor desta não necessita ser definido.

7.13.1 Status de alocação

`ALLOCATED(ARRAY).` Retorna, quando a matriz alocável `ARRAY` está correntemente alocada, o valor `.TRUE.`; na outra situação, o valor `.FALSE.` é retornado. Se os status da alocação é indefinido, o resultado também é indefinido.

7.13.2 Limites, forma e tamanho

As funções a seguir inquirem sobre as propriedades de uma matriz. No caso de uma matriz alocável, esta deve estar alocada. No caso de um *ponteiro (pointer)*, ele deve estar associado a um *alvo (target)*. Uma seção de matriz ou uma expressão matricial é assumida ter limite inferior 1 e limite superior igual às extensões.

`LBOUND(ARRAY[,DIM]).` Quando `DIM` é ausente, retorna uma matriz inteira padrão de posto um, a qual contém os limites inferiores. Quando `DIM` é presente, este deve ser um inteiro escalar e o resultado é um escalar inteiro padrão com o valor do limite inferior na dimensão `DIM`. Como o posto do resultado depende da presença de `DIM`, o argumento da função não pode ser, por sua vez, um argumento mudo opcional.

`SHAPE(SOURCE).` Retorna um vetor inteiro padrão contendo a forma da matriz ou escalar `SOURCE`. No caso de um escalar, o resultado tem tamanho zero.

`SIZE(ARRAY[,DIM]).` Retorna um escalar inteiro padrão igual ao tamanho da matriz `ARRAY` ou a extensão ao longo da dimensão `DIM`, caso este argumento esteja presente.

`UBOUND(ARRAY[,DIM]).` Similar a `LBOUND` exceto que retorna limites superiores.

7.14 Funções de construção e manipulação de matrizes

Há oito funções que constroem ou manipulam matrizes de qualquer tipo.

7.14.1 Função elemental `MERGE`

`MERGE(TSOURCE,FSOURCE,MASK).` Trata-se de uma função elemental. `TSOURCE` pode ter qualquer tipo e `FSOURCE` deve ser do mesmo tipo e espécie. `MASK` deve ser do tipo lógico. O resultado é `TSOURCE` se `MASK` é verdadeiro ou `FSOURCE` no contrário.

7.14.2 Agrupando e desagrupando matrizes

A função transformacional `PACK` agrupa dentro de um vetor aqueles elemento de uma matriz que são selecionados por uma matriz lógica conforme e a função transformacional `UNPACK` executa a operação reversa. Os elementos são tomados na ordem dos elementos das matrizes.

`PACK(ARRAY,MASK[,VECTOR]).` Quando `VECTOR` é ausente, retorna um vetor contendo os elementos de `ARRAY` correspondentes aos valores verdadeiros de `MASK` na ordem dos elementos das matrizes. `MASK` pode ser um escalar de valor `.TRUE.`; em cujo caso, todos os elementos são selecionados. Se `VECTOR` é presente, este deve ser um vetor do mesmo tipo e espécie de `ARRAY` e tamanho no mínimo igual ao número t de elementos selecionados. O resultado, neste caso, tem tamanho igual ao tamanho n do `VECTOR`; se $t < n$, elementos i do resultado para $i > t$ são os elementos correspondentes de `VECTOR`.

`UNPACK(VECTOR,MASK,FIELD).` Retorna uma matriz do tipo e espécie de `VECTOR` e da forma de `MASK`. `MASK` deve ser uma matriz lógica e `VECTOR` deve ser um vetor de tamanho no mínimo igual ao número de elementos verdadeiros de `MASK`. `FIELD` deve ser do mesmo tipo e espécie de `VECTOR` e deve ou ser escalar ou ter a mesma forma que `MASK`. O elemento do resultado correspondendo ao i-ésimo elemento verdadeiro de `MASK`, na ordem dos elementos da matriz, é o i-ésimo elemento de `VECTOR`; todos os outros são iguais aos correspondentes elementos de `FIELD` se este for uma matriz ou igual a `FIELD` se este for um escalar.

7.14.3 Alterando a forma de uma matriz

A função transformacional RESHAPE permite que a forma de uma matriz seja alterada, com a possível permutação dos índices.

RESHAPE(SOURCE,SHAPE[,PAD][,ORDER]). Retorna uma matriz com forma dada pelo vetor inteiro SHAPE e tipo e espécie iguais aos da matriz SOURCE. O tamanho de SHAPE deve ser constante e seus elementos não podem ser negativos. Se PAD está presente, esta deve ser uma matriz do mesmo tipo e espécie de SOURCE. Se PAD estiver ausente ou for uma matriz de tamanho zero, o tamanho do resultado não deve exceder o tamanho de SOURCE. Se ORDER estiver ausente, os elementos da matriz resultante, arranjados na ordem de elementos de matrizes (seção 6.6.2), são os elementos de SOURCE, seguidos por cópias de PAD, também na ordem de elementos de matrizes. Se ORDER estiver presente, este deve ser um vetor inteiro com um valor que é uma permutação de (1, 2, ..., n); os elementos R($s_1, \ldots, s_n$) do resultado, tomados na ordem dos índices para a matriz tendo elementos R($s_{\text{ordem}(1)}, \ldots, s_{\text{ordem}(n)}$), são aqueles de SOURCE na ordem de elementos de matriz seguidos por cópias de PAD, também na ordem de elementos de matriz. Por exemplo, se ORDER tem o valor (/3,1,2/), os elementos R(1,1,1), R(1,1,2), ..., R(1,1,k), R(2,1,1), R(2,1,2), ... correspondem aos elementos de SOURCE e PAD na ordem de elementos de matriz.

7.14.4 Função transformacional para duplicação

SPREAD(SOURCE,DIM,NCOPIES). Retorna uma matriz do tipo e espécie de SOURCE e de posto acrescido por um. SOURCE pode ser escalar ou matriz. DIM e NCOPIES são inteiros escalares. O resultado contém MAX(NCOPIES,0) cópias de SOURCE e o elemento $(r_1, \ldots, r_{n+1})$ do resultado é SOURCE($s_1, \ldots, s_n$), onde $(s_1, \ldots, s_n)$ é $(r_1, \ldots, r_{n+1})$ com subscrito DIM omitido (ou a própria SOURCE se esta for um escalar).

7.14.5 Funções de deslocamento matricial

CSHIFT(ARRAY,SHIFT[,DIM]). Retorna uma matriz do mesmo tipo, espécie e forma de ARRAY. DIM é um escalar inteiro. Se DIM for omitido, o seu valor será assumido igual a 1. SHIFT é do tipo inteiro e deve ser um escalar se ARRAY tiver posto um. Se SHIFT é escalar, o resultado é obtido deslocando-se toda seção vetorial que se estende ao longo da dimensão DIM de forma circular SHIFT vezes. O sentido do deslocamento depende do sinal de SHIFT e pode ser determinado a partir do caso com SHIFT= 1 e ARRAY de posto um e tamanho m, quando o elemento i do resultado é ARRAY(i + 1), i= 1, 2, ..., m-1 e o elemento m é ARRAY(1). Se SHIFT for uma matriz, esta deve ter a forma de ARRAY com a dimensão DIM omitida; desta forma, SHIFT oferece um valor distinto para cada deslocamento.

EOSHIFT(ARRAY,SHIFT[,BOUNDARY][,DIM]). É idêntica a CSHIFT, exceto que um deslocamento final é executado e valores de borda são inseridos nas lacunas assim criadas. BOUNDARY pode ser omitida quando ARRAY tiver tipo intrínsecos, em cujo caso o valor zero é inserido para os casos inteiro, real e complexo; .FALSE. no caso lógico e brancos no caso de caracteres. Se BOUNDARY estiver presente, deve ser do mesmo tipo e espécie de ARRAY; ele pode ser um escalar e prover todos os valores necessários ou pode ser uma matriz cuja forma é a de ARRAY com dimensão DIM omitida e prover um valor separado para cada deslocamento.

7.14.6 Transposta de uma matriz

A função TRANSPOSE executa a transposição matricial para qualquer matriz de posto dois.

TRANSPOSE(MATRIX). Retorna uma matriz do mesmo tipo e espécie da matriz de posto dois MATRIX. Elemento (i,j) do resultado é igual a MATRIX(i,j).

7.15 Funções transformacionais para localização geométrica

Há duas funções transformacionais que localizam as posições dos valores máximos e mínimos de uma matriz inteira ou real.

MAXLOC(ARRAY[,MASK]). Retorna um vetor inteiro padrão de tamanho igual ao posto de ARRAY. Seu valor é a sequência de subscritos de um elemento de valor máximo (dentre aqueles correspondentes aos elementos de valor .TRUE. da matriz lógica MASK, caso esta exista), como se todos os limites inferiores de ARRAY fossem iguais a 1. Se houver mais de um elemento com o mesmo valor máximo, o primeiro na ordem de elementos de matriz é assumido.

`MAXLOC(ARRAY,DIM[,MASK])`. Retorna um vetor inteiro padrão de forma igual à forma de `ARRAY` com a dimensão `DIM` omitida, sendo `DIM` um inteiro escalar de valor no intervalo $1 \leqslant$ `DIM` $\leqslant$ `RANK(ARRAY)`. O valor de cada elemento do resultado é a posição do primeiro elemento de valor máximo na correspondente seção vetorial de `ARRAY` variando ao longo da dimensão `DIM`, dentre aqueles correspondentes aos elementos de valor `.TRUE.` da matriz lógica `MASK`, caso esta exista. O compilador pode distinguir sem auxílio caso o segundo argumento seja `DIM` ou `MASK` pelo fato de que o tipo de variável é distinto.

`MINLOC(ARRAY[,MASK])`. Idêntica a `MAXLOC(ARRAY [, MASK])`, exceto que agora é obtida a posição do elemento de menor valor.

`MINLOC(ARRAY,DIM[,MASK])`. Idêntica a `MAXLOC(ARRAY, DIM [, MASK])`, exceto que agora são obtidas as posições dos elementos de valor mínimo.

7.16 Função transformacional para dissociação de ponteiro

Em Fortran 95, a função `NULL` está disponível para fornecer status de dissociado a ponteiros.

`NULL([MOLDE])`. Retorna um pointer dissociado. O argumento `MOLDE` é um ponteiro de qualquer tipo e que pode ter qualquer status, inclusive de indefinido. O tipo, espécie e posto do resultado são aqueles de `MOLDE` se ele está presente; em caso contrário, são aqueles do objeto com o qual o ponteiro está associado.

7.17 Subrotinas intrínsecas não-elementais

Há cinco subrotinas intrínsecas não elementais, as quais foram escolhidas ser subrotinas no lugar de funções devido à necessidade destas retornarem diversas informações através de seus argumentos.

7.17.1 Relógio de tempo real

Há duas subrotinas que retornam informação acerca do relógio de tempo real; a primeiro é baseada no padrão ISO 8601 (Representação de datas e tempos). Assume-se que exista um relógio básico no sistema que é incrementado por um para cada contagem de tempo até que um valor máximo `COUNT_MAX` é alcançado e daí, na próxima contagem de tempo, este é zerado. Valores padronizados são retornados em sistemas sem relógio. Todos os argumentos têm intenção `OUT` (seção 8.2.5). Adicionalmente, há uma subrotina que acessa o relógio interno do processador.

Acesso a qualquer argumento opcional pode ser realizado através do uso das palavras-chave (seções 8.2.8 e 8.2.9).

`CALL DATE_AND_TIME([DATE][,TIME][,ZONE][,VALUES])`. Retorna os seguintes valores (com valores-padrão nulos ou `-HUGE(0)`, conforme apropriado, caso não haja relógio):

`DATE` é uma variável escalar de caractere da espécie padrão que contém a data na forma *ccyymmdd*, correspondendo ao século, ano, mês e dia, respectivamente.

`TIME` é uma variável escalar de caractere padrão que contém o tempo na forma *hhmmss.sss*, correspondendo a horas, minutos, segundos e milisegundos, respectivamente.

`ZONE` é uma variável escalar de caractere padrão que é fixada como a diferença entre o tempo local e o Tempo Coordenado Universal (UTC, de *Coordinated Universal Time*, também conhecido como Tempo Médio de Greenwich, ou *Greenwich Mean Time*) na forma *Shhmm*, correspondendo ao sinal, horas e minutos, respectivamente. Por exemplo, a hora local de Brasília corresponde a `UTC=-0300`.

`VALUES` é um vetor inteiro padrão que contém a seguinte sequência de valores: o ano, o mês do ano, o dia do mês, a diferença temporal em minutos com relação ao UTC, a hora do dia, os minutos da hora, os segundos do minuto e os milisegundos do segundo.

`CALL SYSTEM_CLOCK([COUNT][,COUNT_RATE][,COUNT_MAX])`. Retorna o seguinte:

`COUNT` é uma variável escalar inteira padrão que contém o valor, dependente do processador, que correspondem ao valor corrente do relógio do processador, ou `-HUGE(0)` caso não haja relógio. Na primeira chamada da subrotina, o processador pode fixar um valor inicial igual a zero.

`COUNT_RATE` é uma variável escalar inteira padrão que contém o número de contagens por segundo do relógio, ou zero caso não haja relógio.

`COUNT_MAX` é uma variável escalar inteira padrão que contém o valor máximo que `COUNT` pode assumir, ou zero caso não haja relógio.

7.17.2 Tempo da CPU

Em Fortran 95, há uma subrotina intrínseca não-elemental que retorna o tempo do processador.

`CALL CPU_TIME(TIME).` Retorna o seguinte:

`TIME` é uma variável real escalar padrão à qual é atribuída uma aproximação (dependente de processador) do tempo do processador em segundos, ou um valor negativo (dependente de processador) caso não haja relógio.

7.17.3 Números aleatórios

Uma sequência de números pseudo-aleatórios é gerada a partir de uma semente que é fornecida como um vetor de números inteiros. A subrotina `RANDOM_NUMBER` retorna os números pseudo-aleatórios e a subrotina `RANDOM_SEED` permite que uma inquirição seja feita a respeito do tamanho ou valor do vetor-semente e, então, redefinir o mesmo. As subrotinas fornecem uma interface a uma sequência que depende do processador.

`CALL RANDOM_NUMBER(COLHE)` Retorna um número pseudo-aleatório a partir da distribuição uniforme de números no intervalo $0 \leq x < 1$ ou um vetor destes números. `COLHE` tem intenção `OUT`, pode ser um escalar ou vetor e deve ser do tipo real.

`CALL RANDOM_SEED([SIZE][,PUT][,GET]).` Tem os argumentos abaixo. Não mais de um argumento pode ser especificado; caso nenhum argumento seja fornecido, a semente é fixada a um valor dependente do processador.

`SIZE` tem intenção `OUT` e é um inteiro escalar padrão que o processador fixa com o tamanho n do vetor-semente.

`PUT` tem intenção `IN` e é um vetor inteiro padrão de tamanho n que é usado pelo processador para fixar uma nova semente. Um processador pode fixar o mesmo valor de semente para mais de um valor de `PUT`.

`GET` tem intenção `OUT` e é um vetor inteiro padrão de tamanho n que o processador fixa como o valor atual da semente.

Capítulo 8

Sub-Programas e Módulos

Como foi visto nos capítulos anteriores, é possível escrever um programa completo em Fortran em um único arquivo, ou como uma unidade simples. Contudo, se o código é suficientemente complexo, pode ser necessário que um determinado conjunto de instruções seja realizado repetidas vezes, em pontos distintos do programa. Um exemplo desta situação seria a definição de uma função *extrínseca*, ou seja, não contida no conjunto de funções intrínsecas discutidas no capítulo 7. Neste caso, é melhor quebrar o programa em unidades distintas.

Cada uma das *unidades de programa* corresponde a um conjunto completo e consistente de tarefas que podem ser, idealmente, escritas, compiladas e testadas individualmente, sendo posteriormente incluídas no programa principal para gerar um arquivo executável. Em Fortran há dois tipos de estruturas que se encaixam nesta categoria: *subrotinas* e *funções* (externas ou extrínsecas).

Em Fortran 77, havia somente dois tipos de unidades de programas distintas:

- Programa principal.
- Rotinas externas.

Em Fortran 90/95, existem, ao todo, três unidades distintas de programa:

- Programa principal.
- Rotinas externas.
- Módulos.

Cada uma destas unidades de programas serão discutidas nas seções seguintes.

Neste capítulo (e nos posteriores), os termos *rotina (procedure)* e *sub-programa* serão usados alternadamente para indicar de forma genérica tanto subrotinas quanto funções. Outros textos podem vir a tratar estes dois termos como representando entidades ligeiramente distintas.

8.1 Unidades de programa

Tanto em Fortran 77 quanto em Fortran 90/95, um código executável é criado a partir de um e somente um *programa principal*, o qual pode invocar *rotinas externas* e, no caso do Fortran 90/95, pode usar também *módulos*. A única *unidade de programa* que deve necessariamente existir sempre é o programa principal.

Com a introdução do Fortran 90/95, quaisquer uma destas três unidades de programas podem também invocar *rotinas internas*, as quais têm estrutura semelhante às rotinas externas, porém não podem ser testadas isoladamente. Um diagrama ilustrativo das três unidades de programas existentes no Fortran 90/95 pode ser visto na figura 8.1.

Uma descrição mais completa das unidades de programas é dada a seguir.

8.1.1 Programa principal

Todo código executável deve ser composto a partir de um, e somente um, programa principal. Opcionalmente, este pode invocar sub-programas. Um programa principal possui a seguinte forma:

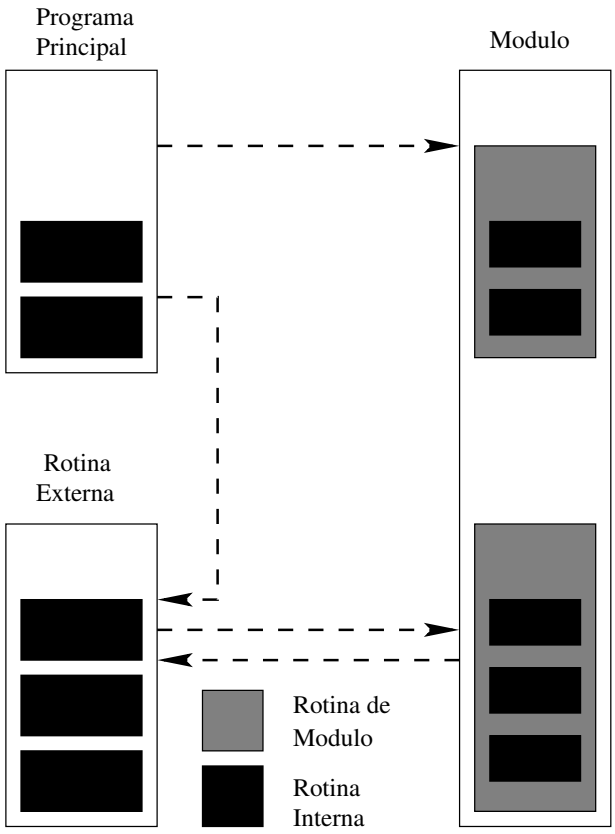


Figura 8.1: As três unidades de programa, as quais podem possuir também rotinas internas.

```
[PROGRAM <nome do programa>
   [<declarações de variáveis>]
   [<comandos executáveis>]
[CONTAINS
   <sub-programas internos>
END [PROGRAM [<nome do programa>]]
```

A declaração `PROGRAM` é opcional, porém o seu uso é recomendado.[1] O único campo não opcional na estrutura de um programa, na definição do padrão da linguagem, é a instrução `END`, a qual possui dois propósitos: indicar ao compilador que o programa chegou ao fim e, quando da execução do código, provoca a parada do mesmo.

Nos capítulos anteriores, já foram dados vários exemplos da estrutura do programa principal e, portanto, maiores detalhes não são necessários aqui.

A declaração `CONTAINS` indica a presença de um ou mais sub-programas internos. Estes serão descritos em mais detalhes na seção 8.2.2. Se a execução do último comando anterior a `CONTAINS` não provoca um desvio de percurso na execução do programa, através de uma instrução `GO TO`, por exemplo, o controle passa por sobre os sub-programas internos ao comando `END` e o programa pára.

O comando `END` pode ser rotulado e ser então atingido por um desvio de execução, como no exemplo

```
...
GO TO 100
...
100 END PROGRAM
```

em cuja situação o programa também termina. Contudo, este estilo de programação é típico do Fortran 77, mas não é recomendado pelo padrão do Fortran 90/95.

O comando `STOP`. Outra forma de interromper a execução do programa é através do comando `STOP`, o qual pode ser rotulado, pode ser parte de um comando ou construto `IF` e é um comando que pode aparecer no programa principal ou em um sub-programa.

[1] No caso do compilador F, é obrigatório.

Caso haja vários comandos `STOP` em uma mesma unidade, estes podem ser distingüidos por um *código de acesso*, que se trata de uma constante de caractere ou uma string de até 5 dígitos. Isto originará uma mensagem na saída padrão indicando o ponto onde a execução foi terminada. A seguir, dois exemplos deste caso:

```
STOP 'Dados incompletos.  Programa interrompido.'
STOP 12345
```

8.1.2 Rotinas externas

Rotinas externas são chamadas de dentro de um programa principal ou de outra unidade, usualmente com o intuito de executar uma tarefa bem definida dentro dos objetivos cumpridos pelo programa completo. Rotinas externas são escritas em arquivos distintos, separados do arquivo do programa principal.

Durante o processo de criação do programa executável, estas rotinas podem ser compiladas conjuntamente com o programa principal ou em separado. Nesta última situação, os programas objetos criados durante compilações prévias das rotinas externas são linkados junto com o programa objeto do programa principal. Embora seja possível colocar-se mais de uma rotina externa no mesmo arquivo, esta prática não é recomendada.

8.1.3 Módulos

O terceiro tipo de unidade de programa, exclusivo no Fortran 90/95, é o módulo. Trata-se de uma maneira de se agrupar dados globais, tipos derivados e suas operações associadas, blocos de interface, grupos `NAMELIST` (seção 9.2) e rotinas internas. Tudo associado com uma determinada tarefa pode ser coletado dentro de um módulo e acessado quando necessário. Aquelas partes que são restritas ao funcionamento interno da tarefa e não são do interesse direto do programador podem ser feitas "invisíveis" a este, o que permite que o funcionamento interno do módulo possa ser alterado sem a necessidade de alterar o programa que o usa, prevenindo-se assim alteração acidental dos dados.

Um módulo tem a seguinte forma:

```
MODULE <nome módulo>
   <declarações de variáveis>
[CONTAINS
   <rotinas de módulo>
END [MODULE [<nome módulo>]]
```

Assim como para `END PROGRAM`, é recomendado o uso da forma completa do comando `END`.

Uma discussão mais extensa acerca dos módulos em Fortran 90/95 será feita na seção 8.3.

8.2 Sub-programas

sub-programas podem ser *subrotinas* ou *funções*. Como já foi mencionado, tarefas que podem ser delimitadas por um número finito de operações e que devem ser realizadas repetidas vezes durante a execução de um programa devem ser escritas como sub-programas.

Uma função retorna um único valor, o qual pode ser um escalar ou uma matriz, e esta usualmente não altera os valores de seus argumentos[2]. Neste sentido, uma função em Fortran age como uma função em análise matemática. Já uma subrotina pode executar uma tarefa mais complexa e retornar diversos valores através de seus argumentos, os quais podem ser modificados ao longo da computação da subrotina.

8.2.1 Funções e subrotinas

Exceto pela declaração inicial, os sub-programas apresentam uma forma semelhante a de um programa principal. Um sub-programa pode ser uma subrotina:

```
[PURE] [ELEMENTAL] [RECURSIVE] SUBROUTINE <nome subrotina> [(<lista argumen-
tos mudos>)]
   [<declarações de variáveis>]
```

[2] Situações onde este requisito pode ser relaxado e como estabelecer salvaguardas são discutidas na seção 8.2.16.

```
   [<comandos executáveis>]
[CONTAINS
   <sub-programas internos>]
END [SUBROUTINE [<nome subrotina>]]
```

ou pode ser uma função:

```
[PURE] [ELEMENTAL] [<tipo>] [RECURSIVE] FUNCTION <nome função> &
                   (<lista argumentos mudos>)[RESULT (<nome resultado>)]
   [<declarações de variáveis>]
   [<comandos executáveis>]
[CONTAINS
   <sub-programas internos>]
END [FUNCTION [<nome função>]]
```

Os diversos constituintes das formas gerais apresentadas acima serão discutidos em detalhes nas seções seguintes. Por enquanto, serão apresentados alguns dos atributos usualmente empregados no campo de `<declarações de variáveis>` de um sub-programa. Além dos atributos já vistos, relacionados com as declarações de tipo e espécie de variáveis, dos atributos `DIMENSION` e `ALLOCATABLE`, os seguintes atributos podem ser usados:

```
INTENT([IN][OUT][INOUT])
OPTIONAL
SAVE
EXTERNAL
INTRINSIC
POINTER
TARGET
```

A declaração `CONTAINS` cumpre exatamente o mesmo papel cumprido dentro do programa principal. O efeito do comando `END` em um sub-programa consiste em retornar o controle à unidade que o chamou, ao invés de interromper a execução do programa. Aqui também recomenda-se o uso da forma completa do comando para deixar claro ao compilador e ao programador qual parte do programa está sendo terminada.

Uma função é ativada ou chamada de forma semelhante como se usa uma função em análise matemática. Por exemplo, dada a função `BESSELJ(n,x)`, a qual calcula $J_n(x)$, o valor da função de Bessel do primeiro tipo de ordem n no ponto x. Esta função pode ser chamada para atribuir seu valor a uma variável escalar ou a um elemento de matriz:

```
y= BESSELJ(n,x)
```

sendo que o tipo e espécie de `y` devem, em princípio, concordar com o tipo e espécie do resultado de `BESSELJ(n,x)`. Contudo, caso isto não ocorra, valem as regras de conversão definidas no capítulo 4. Uma função pode também fazer o papel de um operando em uma expressão:

```
y= BESSELJ(n,x) + 2*BESSELJ(n,x**2)
```

ou ainda servir de argumento para uma outra rotina.

Uma subrotina, devido ao fato de esta retornar, em geral, mais de um valor em cada chamada, não pode ser operada como uma função em análise matemática mas deve, isto sim, ser *chamada* através da instrução `CALL`. Supondo que exista a subrotina `BASCARA`, a qual calcula as raízes de um polinômio de segundo grau, `x1` e `x2`, estas serão obtidas através da chamada:

```
CALL BASCARA(A0,A1,A2,X1,X2)
```

onde os argumentos da subrotina serão discutidos na seção 8.2.3 e posteriores. Por exemplo, é possível usar-se uma função como argumento da subrotina:

```
CALL BASCARA(A0,A1,BESSELJ(n,x),x1,x2)
```

8.2.2 Rotinas internas

Já foi observado nas seções anteriores que rotinas internas podem ser definidas dentro de quaisquer unidades de programa. Uma rotina interna possui a seguinte estrutura:

```
[RECURSIVE] SUBROUTINE <nome subrotina> [(<lista argumentos mudos>)]
   [<declarações de variáveis>]
   [<comandos executáveis>]
END SUBROUTINE [<nome subrotina>]]
```

ou

```
[<tipo>] [RECURSIVE] FUNCTION <nome função> (<lista argumentos mudos>) &
                                              [RESULT (<nome resultado>)]
   [<declarações de variáveis>]
   [<comandos executáveis>]
END FUNCTION [<nome função>]]
```

Nota-se agora que as declarações `FUNCTION` e `SUBROUTINE` agora são obrigatórias no comando `END`. Uma determinada unidade de programa pode conter qualquer número de rotinas internas, porém estas, por sua vez, não podem conter outras rotinas internas.

Cabe aqui uma menção a respeito da diferença entre uma rotina interna e uma rotina de módulo. Esta última também é incluída depois da palavra-chave `CONTAINS`, em um módulo, mas pode conter rotinas internas a ela.

Uma rotina interna automaticamente tem acesso a todas as entidades do hospedeiro, tais como variáveis, e possuem também a habilidade de chamar as outras rotinas deste. Contudo, uma determinada rotina interna não possui acesso às entidades de outra rotina interna.

Um exemplo simples do uso de uma função interna é apresentado a seguir. Note que a variável `a` é declarada no programa principal, tornando-se assim uma variável declarada também dentro do âmbito da função `calc_a_raiz`:

```
program rot_int
implicit none
real :: x,a
!
print*, "Entre com o valor de x:"
read*, x
print*, "Entre com o valor de a:"
read*, a
print*, "O resultado de a + sqrt(x) :"
print*, calc_a_raiz(x)
print*, "O resultado de a + sqrt(1+ x**2) :"
print*, calc_a_raiz(1.0 + x**2)
CONTAINS
   function calc_a_raiz(y)
   real              :: calc_a_raiz
   real, intent(in) :: y
   calc_a_raiz= a + sqrt(y)
   return
   end function calc_a_raiz
end program rot_int
```

No restante deste capítulo, serão descritas várias propriedades que se aplicam a sub-programas internos, externos e a rotinas de módulos.

8.2.3 Argumentos de sub-programas

Argumentos de sub-programas fornecem um meio alternativo para que duas unidades de programa compartilhem os mesmos dados. Estes consistem em uma lista de variáveis escalares ou matriciais, constantes, expressões escalares ou matriciais e até mesmo os nomes de outras rotinas. Os argumentos de um sub-programa são muitas vezes denominados *variáveis mudas (dummy variables)*. Estes devem coincidir com

os argumentos transferidos a partir da unidade que chama a rotina em tipo, espécie e forma. Contudo, os nomes não necessitam ser os mesmos, como se pode ver no exemplo acima. Com base no mesmo exemplo, poder-se-ia realizar duas chamadas da função `calc_a_raiz`:

```
...
read*, a,x
print*, calc_a_raiz(x)
print*, calc_a_raiz(a + x**2)
...
```

Sempre que possível, é recomendado que o nome do argumento transferido ao sub-programa seja igual ao nome da variável muda. O ponto importante é que sub-programas podem ser escritos de forma independente uns dos outros. A associação das variáveis mudas com os argumentos verdadeiros ocorrem somente quando a rotina é chamada. Desta forma, bibliotecas de sub-programas podem ser construídos, possibilitando que estes sejam usados em diferentes programas (ver seção 8.2.6).

O programa-exemplo `bascara1`, apresentado na página 89, implementa, de forma ingênua, o cálculo das raízes de um polinômio de segundo grau através da Fórmula de Báscara usando uma subrotina interna. Exemplos semelhantes a este serão usados para ilustrar o uso de rotinas em outras unidades de programa.

8.2.4 Comando `RETURN`

No padrão da linguagem, um sub-programa pode ser encerrado pelo comando `END`, sendo o controle do fluxo retornado à unidade que chamou este sub-programa. Contudo, o meio recomendado de se retornar controle a partir de um sub-programa consiste na utilização do comando `RETURN`, o qual pode surgir em mais de um ponto da rotina, ao contrário do `END`.

Como no caso dos comandos `STOP` e `END`, o `RETURN` pode ser rotulado, pode fazer parte de um construto como o `IF` e é um comando executável. O `RETURN` não pode aparecer entre os comandos executáveis do programa principal.

8.2.5 Atributo e declaração `INTENT`

No programa `bascara1`, (programa 8.1) as variáveis mudas `a0`, `a1` e `a2` foram utilizadas para transferir à subrotina informação acerca dos coeficientes do polinômio de 2ž grau. Por isto, estes nomes foram declarados ter a *intenção* `INTENT(IN)`. Por outro lado, a subrotina devolveu ao programa os valores das raízes reais (caso existam) através das variáveis mudas `r1` e `r2`, as quais foram declaradas ter a *intenção* `INTENT(OUT)`. Uma terceira possibilidade seria a existência de uma variável cujo valor é inicialmente transferido **para dentro** da subrotina, modificado por esta e, então transferido **para fora** da subrotina. Neste caso, esta variável deveria ser declarada com a *intenção* `INTENT(INOUT)`.

Se uma variável muda é especificada com atributo `INTENT(IN)`, o seu valor não pode ser alterado pela rotina, seja através de atribuição de valor a partir de uma expressão, ou através de sua transferência para uma outra rotina que, por sua vez, alteraria este valor.

Se a variável é especificada com `INTENT(OUT)`, o seu valor é indefinido na chamada da rotina, sendo este então definido durante a execução da mesma e finalmente transferido à unidade que chamou a rotina.

Se a variável é especificada com `INTENT(INOUT)`, esta tem o seu valor inicial transferido na chamada da rotina, o valor pode ser alterado e, então, o novo valor pode ser devolvido à unidade que chamou a rotina.

É recomendado que a intenção de todas as variáveis mudas seja declarada. Em funções, variáveis mudas somente podem ser declaradas com `INTENT(IN)`.

Como alternativa ao uso do atributo `INTENT(INOUT)`, pode-se declarar a intenção de nomes de argumentos mudos de rotinas, que não sejam sub-programas mudos, através da declaração `INTENT(INOUT)`, a qual tem a forma geral:

```
INTENT(<inout>) [::] <lista nomes argumentos mudos>
```

Por exemplo,

```
SUBROUTINE SOLVE(A, B, C, X, Y, Z)
REAL :: A, B, C, X, Y, Z
INTENT(IN)  :: A, B, C
INTENT(OUT) :: X, Y, Z
...
```

Programa 8.1: Exemplo de emprego de rotinas internas.

```
program bascara1
implicit none
logical :: controle
real :: a, b, c
real :: x1, x2 ! Raizes reais.
!
do
   print*, "Entre com os valores dos coeficientes (a,b,c),"
   print*, "onde a*x**2 + b*x + c."
   read*, a,b,c
   call bascara(a,b,c,controle,x1,x2)
   if(controle)then
      print*, "As raizes (reais)  sao:"
      print*, "x1=",x1
      print*, "x2=",x2
      exit
   else
      print*, "As raizes sao complexas.  Tente novamente."
   end if
end do
CONTAINS
   subroutine bascara(a2,a1,a0,raizes_reais,r1,r2)
! Variaveis mudas:
   real, intent(in)     :: a0, a1, a2
   logical, intent(out) :: raizes_reais
   real, intent(out)    :: r1, r2
! Variaveis locais:
   real :: disc
!
   raizes_reais= testa_disc(a2,a1,a0)
   if(.not. raizes_reais)return
   disc= a1*a1 - 4*a0*a2
   r1= 0.5*(-a1 - sqrt(disc))/a2
   r2= 0.5*(-a1 + sqrt(disc))/a2
   return
   end subroutine bascara
!
   function testa_disc(c2,c1,c0)
   logical :: testa_disc
! Variaveis mudas:
   real, intent(in) :: c0, c1, c2
   if(c1*c1 - 4*c0*c2 >= 0.0)then
      testa_disc= .true.
   else
      testa_disc= .false.
   end if
   return
   end function testa_disc
end program bascara1
```

8.2.6 Rotinas externas e bibliotecas

Em muitas situações, uma determinada rotina é escrita com o intuito de ser utilizada em diversas aplicações distintas. Um exemplo seria uma rotina genérica que calcula a integral de uma função qualquer entre dois pontos reais. Nesta situação, é mais interessante manter a rotina como uma entidade distinta do programa principal, para que esta possa ser utilizada por outras unidades de programas.

Por isto, uma ou mais rotinas podem ser escritas em um arquivo em separado, constituindo uma unidade de programa própria, já mencionada na seção 8.1.2. Esta unidade pode então ser compilada em separado, quando será então gerado o programa-objeto correspondente a esta parte do código. Este programa-objeto é então acessado pelo programa linkador para gerar o programa executável, ligando a(s) rotina(s) contida(s) nesta unidade com as outras unidades que fazem referência à(s) mesma(s). O procedimento para realizar esta linkagem é realizado, usualmente, de duas maneiras distintas:

1. O(s) programa-objeto(s) é(são) linkado(s) diretamente com as demais unidades de programa. Neste caso, o compilador e o linkador identificam a existência do programa-objeto pela sua extensão, fornecida na linha de comando que demanda a criação do código executável. A extensão de um programa-objeto é usualmente `*.o` em sistemas operacionais Linux/Unix ou `*.obj` em sistemas DOS/Windows.

2. O programa-objeto recém criado é armazenado, inicialmente, em uma biblioteca de programas-objeto criados anteriormente. Então, no momento de se gerar o código executável a partir de um programa-fonte, o linkador é instruído com o nome e o endereço da biblioteca que contém o(s) programa-objeto(s) da(s) rotina(s) chamadas pelo programa principal e/ou por outras unidades de programas. O linkador procura nesta biblioteca pelos nomes das rotinas invocadas e incorpora o código destas (e somente destas) ao programa executável.

Com base nesta filosofia, o programa `bascara1`, apresentado na página 8.1, pode ser então desmembrado em três arquivos distintos, dois contendo a subrotina `bascara` e a função `testa_disc` e um contendo o programa principal `bascara2`, apresentados na página 91. Note o uso do atributo `EXTERNAL` para indicar que o nome `TESTA_DISC` consiste em uma função externa do tipo lógico.

8.2.7 Interfaces implícitas e explícitas

No exemplo apresentado na página 91, o nome `testa_disc` é identificado como pertencente a uma função lógica pelo atributo `EXTERNAL`:

```
...
LOGICAL, EXTERNAL :: TESTA_DISC
...
```

Caso esta declaração não fosse realizada, o compilador iria gerar uma mensagem de erro indicando que este nome não foi declarado.

Uma outra maneira de declarar um nome como pertencente a uma função externa é usando a declaração `EXTERNAL`, em vez do atributo. Esta é a forma utilizada no Fortran 77. Desta maneira, o mesmo resultado seria obtido através das seguintes linhas:

```
...
LOGICAL :: TESTA_DISC
EXTERNAL :: TESTA_DISC
...
```

A única diferença entre o exemplo acima e o mesmo tipo de declaração em um programa Fortran 77 consiste na ausência dos (`::`) no último caso.

Em ambas as opções anteriores, somente é fornecido ao compilador o nome da função. Nenhum controle existe sobre os argumentos da função `testa_disc` e da subrotina `bascara`. Por isto, seria possível o programa `bascara2` chamar a subrotina `bascara` tentando transferir variáveis complexas, por exemplo, quando os argumentos mudos da subrotina esperam variáveis reais. Esta falta de controle, não somente quanto ao tipo e espécie das variáveis transferidas a rotinas externas, mas também quanto ao número e a ordem dos argumentos, é inerente ao Fortran 77; nenhum mecanismo de controle existe para auxiliar o compilador e o linkador na tarefa de determinar a coerência entre as diferentes unidades de programas.

Para gerar chamadas a sub-programas de forma correta, o compilador necessita conhecer certos detalhes destes sub-programas, como os seus nomes e o número, tipo e espécie dos argumentos. Em Fortran 90/95, esta tarefa é desempenhada pelos *blocos de interface*.

```
! Chamada pela subrotina bascara para testar se razes so reais.
!
function testa_disc(c2,c1,c0)
implicit none
logical :: testa_disc
! Variveis mudas:
real, intent(in) :: c0, c1, c2
if(c1*c1 - 4*c0*c2 >= 0.0)then
   testa_disc= .true.
else
   testa_disc= .false.
end if
return
end function testa_disc
```

```
! Calcula as razes reais, caso existam, de um polinnio de grau 2.
! Usa funo testa_disc.
!
subroutine bascara(a2,a1,a0,raizes_reais,r1,r2)
implicit none
! Variveis mudas:
real, intent(in)     :: a0, a1, a2
logical, intent(out) :: raizes_reais
real, intent(out)    :: r1, r2
! Funo externa:
logical, external :: testa_disc
! Variveis locais:
real :: disc
!
raizes_reais= testa_disc(a2,a1,a0)
if(.not. raizes_reais)return
disc= a1*a1 - 4*a0*a2
r1= 0.5*(-a1 - sqrt(disc))/a2
r2= 0.5*(-a1 + sqrt(disc))/a2
return
end subroutine bascara
```

```
!Calcula as razes reais de um polinmio de grau 2.
program bascara2
implicit none
logical :: controle
real :: a, b, c
real :: x1, x2 ! Razes reais.
!
do
   print*, "Entre com os valores dos coeficientes (a,b,c),"
   print*, "onde a*x**2 + b*x + c."
   read*, a,b,c
   call bascara(a,b,c,controle,x1,x2)
   if(controle)then
      print*, "As razes (reais)  so:"
      print*, "x1=",x1
      print*, "x2=",x2
      exit
   else
      print*, "As razes so complexas.  Tente novamente."
   end if
end do
end program bascara2
```

No caso de rotinas intrínsecas, sub-programas internos e módulos, esta informação é sempre conhecida pelo compilador; por isso, diz-se que as interfaces são *explícitas*. Contudo, quando o compilador chama uma rotina externa, esta informação não é conhecida de antemão e daí a interface é dita *implícita*. Um bloco de interface, então, fornece a informação necessária ao compilador. A forma geral de um bloco de interface é:

```
INTERFACE
  <corpo interface>
END INTERFACE
```

Nesta forma, este bloco de interface **não** pode ser nomeado. Assim, a interface deixa de ser implícita e se torna explícita.

O `<corpo interface>` consiste das declarações `FUNCTION ou SUBROUTINE`, declarações dos tipos e espécies dos argumentos dos sub-programas e do comando `END FUNCTION` ou `END SUBROUTINE`. Em outras palavras, consiste, normalmente, em uma cópia exata do sub-programa sem seus comandos executáveis ou rotinas internas, cujas interfaces sempre são explícitas. Por exemplo,

```
INTERFACE
   FUNCTION FUNC(X)
   REAL              :: FUNC
   REAL, INTENT(IN) :: X
   END FUNCTION FUNC
END INTERFACE
```

Neste exemplo, o bloco fornece ao compilador a interface da função `FUNC`.

Existe uma certa flexibilidade na definição de um bloco de interfaces. Os nomes (mas não os tipos e espécies) das variáveis podem ser distintos dos nomes mudos definidos na rotina ou usados na chamada da mesma e outras especificações podem ser incluídas, como por exemplo para uma variável local à rotina. Contudo, rotinas internas e comandos `DATA` ou FORMAT não podem ser incluídos.

Um bloco de interfaces pode conter as interfaces de mais de um sub-programa externo e este bloco pode ser colocado na unidade de programa que chama as rotinas externas ou através do uso de módulos, como será discutido na seção 8.3. Em qualquer unidade de programa, o bloco de interfaces é sempre colocado após as declarações de variáveis mudas ou de variáveis locais.

Em certas situações, é necessário fornecer a uma rotina externa o nome de outra rotina externa como um de seus argumentos. Isto pode ser realizado de duas maneiras:

1. Usando o atributo ou declaração `EXTERNAL`, como foi realizado na subrotina `bascara`, na página ??.

2. Com o uso de um bloco de interface. Neste caso, a interface indicará ao compilador que um determinado nome consiste, na verdade, no nome de um outro sub-programa.

As situações onde isto pode ocorrer são discutidas na seção 8.2.12

Como exemplo do uso de interfaces explícitas será apresentado o programa `bascara3`, o qual chama novamente as rotinas `bascara` e `testa_disc`. Este programa está listado nas páginas 93 e 94.

8.2.8 Argumentos com palavras-chave

Argumentos com palavras-chave *(keyword arguments)* são um recurso já utilizado em comandos de Entrada/Saída (E/S) no Fortran 77:

```
READ(UNIT= 5, FMT= 101, END= 9000) X,Y,Z
```

onde `UNIT`, `FMT` e `END` são as palavras-chave. Elas indicam, respectivamente, o número da unidade de acesso para leitura, o rótulo do comando de formato e o rótulo para onde o controle de fluxo deve ser desviado quando se atingir o ponto final de entrada dos dados.

Em Fortran 90/95, argumentos com palavras-chave podem ser utilizados não somente em comandos de E/S, mas também em rotinas. Quando uma rotina tem diversos argumentos, palavras-chave são um excelente recurso para evitar confusão entre os mesmos. A grande vantagem no uso de palavras-chave está em que não é necessário lembrar a ordem dos argumentos. Contudo, é necessário conhecer os nomes dos argumentos mudos definidos no campo de declarações da rotina.

Por exemplo, dada a seguinte função interna:

```
! Chamada pela subrotina bascara para testar se razes so reais.
!
function testa_disc(c2,c1,c0)
implicit none
logical :: testa_disc
! Variveis mudas:
real, intent(in) :: c0, c1, c2
if(c1*c1 - 4*c0*c2 >= 0.0)then
   testa_disc= .true.
else
   testa_disc= .false.
end if
return
end function testa_disc
```

```
! Calcula as razes reais, caso existam, de um polinnio de grau 2.
! Usa funo testa_disc.
!
subroutine bascara(a2,a1,a0,raizes_reais,r1,r2)
implicit none
! Variveis mudas:
real, intent(in)     :: a0, a1, a2
logical, intent(out) :: raizes_reais
real, intent(out)    :: r1, r2
! Variveis locais:
real :: disc
INTERFACE
   function testa_disc(c2,c1,c0)
   logical :: testa_disc
   real, intent(in) :: c0, c1, c2
   end function testa_disc
END INTERFACE
!
raizes_reais= testa_disc(a2,a1,a0)
if(.not. raizes_reais)return
disc= a1*a1 - 4*a0*a2
r1= 0.5*(-a1 - sqrt(disc))/a2
r2= 0.5*(-a1 + sqrt(disc))/a2
return
end subroutine bascara
```

```
!Calcula as razes reais de um polinmio de grau 2.
program bascara3
implicit none
logical :: controle
real :: a, b, c
real :: x1, x2 ! Razes reais.
INTERFACE
   subroutine bascara(a2,a1,a0,raizes_reais,r1,r2)
   real, intent(in)     :: a0, a1, a2
   logical, intent(out) :: raizes_reais
   real, intent(out)    :: r1, r2
   end subroutine bascara
END INTERFACE
!
do
   print*, "Entre com os valores dos coeficientes (a,b,c),"
   print*, "onde a*x**2 + b*x + c."
   read*, a,b,c
   call bascara(a,b,c,controle,x1,x2)
   if(controle)then
      print*, "As razes (reais)  so:"
      print*, "x1=",x1
      print*, "x2=",x2
      exit
   else
      print*, "As razes so complexas.  Tente novamente."
   end if
end do
end program bascara3
```

```
...
CONTAINS
   FUNCTION AREA(INICIO, FINAL, TOL)
   REAL             :: AREA
   REAL, INTENT(IN) :: INICIO, FINAL, TOL
   ...
   END FUNCTION AREA
```

esta pode ser chamada por quaisquer uma das três seguintes atribuições:

```
A= AREA(0.0, 100.0, 1.0E-5)
B= AREA(INICIO= 0.0, TOL= 1.0E-5, FINAL= 100.0)
C= AREA(0.0, TOL= 1.0E-5, FINAL= 100.0)
VALOR= 100.0
ERRO= 1.0E-5
D= AREA(0.0, TOL= ERRO, FINAL= VALOR)
```

onde `A`, `B`, `C` e `D` são variáveis declaradas previamente como reais. Todos os argumentos antes da primeira palavra-chave devem estar ordenados na ordem definida pela função. Contudo, se uma palavra-chave é utilizada, esta não necessita estar na ordem da definição. Além disso, o valor atribuído ao argumento mudo pode ser uma constante, uma variável ou uma expressão, desde que a interface seja obedecida. Depois que uma palavra-chave é utilizada pela primeira vez, todos os argumentos restantes também devem ser transferidos através de palavras-chave. Conseqüentemente, a seguinte chamada não é válida:

```
C= AREA(0.0, TOL= 1.0E-5, 100.0) ! Não é válido.
```

O exemplo acima não é válido porque tentou-se transferir o valor `100.0` sem haver a informação da palavra-chave.

Uma vez que o compilador não será capaz de realizar as associações apropriadas exceto se este conhecer as palavras-chave (nomes dos argumentos mudos), a interface do sub-programa deve ser explícita caso palavras-chave sejam utilizadas. No caso de rotinas internas ou rotinas de módulo, a interface já é explícita. No caso de rotinas externas, faz-se necessário o uso de um bloco de interfaces.

8.2.9 Argumentos opcionais

Em algumas situações, nem todos os argumentos de um sub-programa necessitam ser transferidos durante uma chamada do mesmo. Um argumento que não necessita ser transferido em todas as chamadas possíveis de um sub-programa é denominado *opcional*. Estes argumentos opcionais podem ser declarados pelo atributo `OPTIONAL` na declaração do tipo de variáveis.

Mantendo o exemplo da função `AREA` acima, a seguinte definição pode ser feita:

...

```
CONTAINS
   FUNCTION AREA(INICIO, FINAL, TOL)
   REAL                          :: AREA
   REAL, INTENT(IN), OPTIONAL :: INICIO, FINAL, TOL
   ...
   END FUNCTION AREA
```

a qual agora tem as seguintes chamadas válidas, entre outras:

```
A= AREA(0.0, 100.0, 1.0E-2)
B= AREA(INICIO= 0.0, FINAL= 100.0, TOL= 1.0E-2)
C= AREA(0.0)
D= AREA(0.0, TOL= 1.0E-2)
```

onde se fez uso tanto da associação posicional para os valores dos argumentos, quanto da associação via o uso de palavras-chave.

Um argumento obrigatório (que não é declarado opcional) **deve** aparecer exatamente uma vez na lista de argumentos da chamada de uma rotina, ou na ordem posicional ou na lista de palavras-chave. Já um argumento opcional **pode** aparecer, no máximo uma vez, ou na lista posicional de argumentos ou na lista de palavras-chave.

Da mesma forma como na seção 8.2.8, a lista de palavras-chave pode aparecer em qualquer ordem; porém, depois que o primeiro argumento é transferido via uma palavra-chave, todos os restantes, obrigatórios ou opcionais, devem ser transferidos da mesma forma.

A rotina necessita de algum mecanismo para detectar se um argumento opcional foi transferido na sua chamada, para que esta possa tomar a medida adequada no caso de presença ou de ausência. Este mecanismo é fornecido pela função intrínseca `PRESENT` (seção 7.3). Por exemplo, na função `AREA` acima, pode ser necessário verificar a existência da variável `TOL` para definir a tolerância do cálculo ou usar um valor padrão caso ela não tenha sido transferida:

```
...
REAL :: TTOL
...
IF(PRESENT(TOL))THEN
   TTOL= TOL
ELSE
   TTOL= 1.0E-3
END IF
...
```

A variável `TTOL` é utilizada aqui porque ela pode ser redefinida, ao passo que a variável `TOL` não, uma vez que ela foi declarada com `INTENT(IN)`.

Como no caso dos argumento com palavras-chave, se a rotina é **externa** e possui algum argumento opcional, um bloco de interface **deve** ser fornecido em algum momento. Este não é o caso da função `AREA`, uma vez que ela é interna. Contudo se externa ela o fosse, dever-se-ia declarar o seguinte bloco de interface:

```
INTERFACE
   FUNCTION AREA(INICIO, FINAL, TOL)
```

```
      REAL                          :: AREA
      REAL, INTENT(IN), OPTIONAL :: INICIO, FINAL, TOL
      END FUNCTION AREA
   END INTERFACE
```

8.2.10 Tipos derivados como argumentos de rotinas

Argumentos de rotinas podem ser do tipo derivado se este está definido em somente uma única unidade de programa. Isto pode ser obtido de duas maneiras:

1. A rotina é interna à unidade de programa na qual o tipo derivado é definido.

2. O tipo derivado é definido em um módulo o qual é acessado pela rotina.

8.2.11 Matrizes como argumentos de rotinas

Um outro recurso importante em Fortran é a capacidade de usar matrizes como argumentos mudos de sub-programas. Se um argumento mudo de uma rotina é uma matriz, então o argumento real pode ser:

- o nome da matriz (sem subscritos);

- um elemento de matriz.

A primeira forma transfere a matriz inteira; a segunda forma, a qual transfere somente uma seção que inicia no elemento especificado, é descrita em mais detalhes a seguir.

8.2.11.1 Matrizes como argumentos em Fortran 77

O Fortran 77 já possuía diversos recursos para realizar a transferência de matrizes entre sub-programas. Estes recursos foram posteriormente estendidos com a definição do novo padrão com o Fortran 90/95.

O uso mais simples e comum de argumentos mudos matriciais consiste em tornar disponível o conteúdo completo de uma matriz em uma rotina. Se as matrizes usadas como argumentos reais serão todas do mesmo tamanho, então as matrizes mudas na rotina podem usar limites fixos (alocação estática). Por exemplo:

```
      SUBROUTINE PROD(X, Y, Z)
C Calcula o produto dos vetores X e Y, com 100 elementos cada,
C  resultando no vetor Z do mesmo tamanho.
      REAL X(100), Y(100), Z(100)
      DO 10, I= 1,100
         Z(I)= X(I)*Y(I)
15    CONTINUE
      END
```

Esta subrotina pode ser chamada por um programa semelhante a este:

```
      PROGRAM CHAMA_PROD
      REAL A(100), B(100), C(100)
      READ(UNIT=*, FMT=*) A,B
      CALL PROD(A, B, C)
      WRITE(UNIT=*, FMT=*)C
      END
```

Este uso de matrizes como argumentos mudos de sub-programas é perfeitamente legítimo, porém inflexível, porque não permite o uso de matrizes de qualquer outro tamanho. Para possibilitar uma flexibilidade maior, o Fortran 77 introduziu dois mecanismos que possibilitaram uma alocação dinâmica no tamanho de matrizes usadas como argumentos mudos de sub-programas. Estes mecanismos são as *matrizes ajustáveis* e as *matrizes de tamanho assumido.* A alocação dinâmica, entretanto, é parcial, porque as matrizes reais transferidas aos sub-programas deveriam ter seu tamanho definido por alocação estática em algum ponto dentro das unidades de programas que chamavam as rotinas.

Matrizes ajustáveis

Consiste na generalização dos argumentos matriciais de uma rotina para permitir o uso de matrizes de qualquer tamanho. Neste caso, as expressões inteiras que definem os limites em cada dimensão das matrizes mudas são incluídas nas declarações destas dentro da rotina, usando variáveis inteiras que são transferidas junto com as matrizes nos argumentos da rotina ou através do uso de um bloco `COMMON` (ver seção 8.3.1). O exemplo a seguir mostra como isto é realizado:

```
SUBROUTINE PROD(NPTS, X, Y, Z)
REAL X(NPTS), Y(NPTS), Z(NPTS)
DO 15, I= 1, NPTS
...
END
```

Esta subrotina pode ser invocada pela seguinte linha no programa principal ou em outra unidade de programa:

```
CALL PROD(100, A, B, C)
```

as matrizes `A`, `B` e `C` devem ter o seu tamanho alocado de forma estática nesta ou em alguma outra unidade de programa anterior, conforme já foi mencionado.

O recurso das matrizes ajustáveis pode ser estendido de forma trivial para cobrir o caso de matrizes multi-dimensionais, com limites inferior e superior distintos. Por exemplo,

```
SUBROUTINE MULTI(MAP, K1, L1, K2, L2, TRACO)
DOUBLE PRECISION MAP(K1:L1, K2:L2)
REAL TRACO(L1-K1+1)
```

Como o exemplo mostra, os limites das dimensões das matrizes mudas pode ser expressões inteiras envolvendo não somente constantes mas também variáveis inteiras transferidas à rotina ou na lista de argumentos ou através de um bloco `COMMON`.

O mecanismo de matrizes ajustáveis pode ser usado para matrizes de qualquer tipo. Uma matriz ajustável também pode ser transferida a uma outra rotina como um argumento real com, se necessário, os limites das dimensões sendo passados de forma concomitante.

Matrizes de tamanho assumido

Podem haver circunstâncias onde seja impraticável o uso tanto de matrizes fixas quanto de matrizes ajustáveis como argumentos mudos de um sub-programa. A circunstância mais freqüente ocorre quando o tamanho real da matriz é desconhecido quando a rotina é inicialmente chamada. Neste caso, uma matriz de tamanho assumido é a alternativa existente em Fortran 77.

Estas matrizes também somente são permitidas como argumentos mudos de sub-programas, mas neste caso a matriz é, efetivamente, declarada como tendo tamanho desconhecido. Por exemplo:

```
REAL FUNCTION SOMADOIS(TABELA, ANGULO)
REAL TABELA(*)
N= MAX(1, NINT(SIN(ANGULO)*500.0))
SOMADOIS= TABELA(N) + TABELA(N+1)
END
```

Neste caso, a função somente sabe que a matriz `TABELA` é unidimensional, com um limite inferior igual a um. Isto é tudo que a função necessita saber para acessar os elementos apropriados `N` e `N + 1`. Ao executar a função, é responsabilidade do programador assegurar que o valor do `ANGULO` nunca irá resultar em um subscrito fora do intervalo correto. Este é sempre um problema quando se usa matrizes de tamanho assumido, porque o compilador não tem nenhuma informação a respeito do limite superior da matriz.

Uma matriz de tamanho assumido somente pode ter o limite superior de sua última dimensão especificado por um asterisco; todas as outras dimensões devem ter seus limites definidos pelas regras usuais ou serem definidos na forma de matrizes ajustáveis.

Seções de matrizes

As regras do Fortran requerem que as extensões de uma matriz, usada como argumento real de uma rotina, sejam, no mínimo, iguais às respectivas extensões da matriz muda declarada na rotina. Entretanto, a matriz real pode ter extensões maiores e as regras permitem também uma discordância entre os limites inferior e superior das matrizes real e muda. Por exemplo:

```
      PROGRAM CONFUS
      REAL X(-1:50), Y(10:1000)
      READ(UNIT=*, FMT=*)X, Y
      CALL SAIDA(X)
      CALL SAIDA(Y)
      END
C
      SUBROUTINE SAIDA(MATRIZ)
      REAL MATRIZ(50)
      WRITE(UNIT=*,FMT=*)MATRIZ
      END
```

O resultado deste programa será escrever na saída padrão inicialmente os elementos `X(-1)` a `X(48)`, uma vez que este último coresponde à posição na memória de `MATRIZ(50)`. Em seguida, o programa escreve `Y(10)` a `Y(59)`. Esta subrotina irá funcionar de forma semelhante na seção de uma matriz bi-dimensional:

```
      PROGRAM DOIS_D
      REAL D(100,20)
      ...
      NFATIA= 15
      CALL SAIDA(D(1,NFATIA))
      ...
```

Neste exemplo, a seção da matriz dos elementos `D(1,15)` a `D(50,15)`ser escrita na saída padrão. Isto ocorre através da variação dos índices das linhas, porque esta é a ordem dos elementos de matrizes em Fortran (ver seção 6.6.2).

O uso de um elemento de matriz como argumento real, quando o argumento mudo da rotina é uma matriz completa consiste uma prática que facilmente incorre em erro e, portanto, deve ser evitada.

8.2.11.2 Matrizes como argumentos em Fortran 90/95

Com a definição do novo padrão Fortran 90/95, os recursos para uso de matrizes como argumentos mudos de sub-programas foi ampliado em relação aos mecanismos já existentes no Fortran 77.

No Fortran 90/95, como já foi visto na seção 6.9, a alocação do tamanho das matrizes pode ser realizada de maneira inteiramente dinâmica, sendo o espaço na memória da CPU alocado em tempo real de processamento. Para fazer uso deste novo recurso, o Fortran 90/95 definiu dois novos mecanismos para a definição de matrizes mudas variáveis como argumentos de sub-programas: as *matrizes de forma assumida*, as quais generalizam as matrizes de tamanho assumido do Fortran 77 e os *objetos automáticos*, os quais não são argumentos da rotina, mas tem suas extensões definidas pelas matrizes mudas desta.

Matrizes de forma assumida

Novamente como no caso de matrizes ajustáveis ou de tamanho assumido, as formas dos argumentos reais e mudos ainda devem concordar. Contudo, agora uma matriz muda pode assumir a forma (não somente o tamanho) da matriz real usada no argumento da rotina. Tal matriz muda é denominada *matriz de forma assumida*. Quando a forma é declarada com a especificação `DIMENSION`, cada dimensão tem a sintaxe:

```
[<lim inferior>]:
```

isto é, pode-se definir o limite inferior da dimensão onde `<lim inferior>`, no caso mais geral, é uma expressão inteira que pode depender dos outros argumentos da rotina ou de variáveis globais definidas em módulos (ou em blocos `COMMON`, mas o seu uso é desaconselhado no Fortran 90/95). Se `<lim inferior>` é omitido, o valor padrão é 1. Deve-se notar que é a forma da matriz que é transferida, não os seus limites. Por exemplo:

```
PROGRAM TESTA_MATRIZ
IMPLICIT NONE
REAL, DIMENSION(0:10, 0:20) :: A
...
CALL SUB_MATRIZ(A)
...
CONTAINS
   SUBROUTINE SUB_MATRIZ(DA)
   REAL, DIMENSION(:,:), INTENT(INOUT) :: DA
   ...
   END SUBROUTINE SUB_MATRIZ
END PROGRAM TESTA_MATRIZ
```

A subrotina `SUB_MATRIZ` declara a matriz de forma assumida `DA`. A correspondência entre os elementos da matriz real `A` e a matriz muda `DA` é: `A(0,0)`↔`DA(1,1)` ... `A(I,J)`↔`DA(I+1,J+1)` ... `A(10,20)`↔`DA(11,21)`. Para que houvesse a mesma correspondência entre os elemento de `A` e `DA`, seria necessário declarar esta última como:

```
REAL, DIMENSION(0:,0:), INTENT(INOUT) :: DA
```

Neste caso, a correspondência seria a desejada: `A(0,0)`↔`DA(0,0)` ... `A(I,J)`↔`DA(I,J)` ... `A(10,20)`↔`DA(10,20)`.

Para se definir matrizes de forma assumida, é necessário que a interface seja explícita na unidade que chama a rotina. A matriz real pode ter sido declarada, na unidade que chama a rotina ou em outra unidade anterior, como uma matriz alocável. Como exemplo, considere a seguinte rotina externa, seguida do programa que a chama, programa 8.2, listado na página 100:

```
subroutine sub(ra, rb, rc, max_a)
implicit none
real, dimension(:,:), intent(in)    :: ra, rb
real, dimension(:,:), intent(out)   :: rc
real, intent(out)                   :: max_a
!
max_a= maxval(ra)
rc= 0.5*rb
return
end subroutine sub
```

Objetos automáticos

Uma rotina com argumentos mudos que são matrizes cujos tamanhos variam de uma chamada a outra pode também precisar de matrizes locais (não mudas) cujos tamanhos variem ao sabor dos tamanhos das matrizes mudas. Um exemplo simples é a matriz `TEMP` na subrotina abaixo, destinada a trocar os elementos entre duas matrizes:

```
SUBROUTINE TROCA(A,B)
IMPLICIT NONE
REAL, DIMENSION(:), INTENT(INOUT) :: A, B
REAL, DIMENSION(SIZE(A)) :: TEMP   !A função SIZE fornece o tamanho de TEMP.
TEMP= A
A= B
B= TEMP
RETURN
END SUBROUTINE TROCA
```

Matrizes como a `TEMP`, cujas extensões variam da maneira apresentada são chamadas *matrizes automáticas* e são exemplos de *objetos automáticos de dados.* Estes são objetos de dados cujas declarações dependem do valor de expressões não-constantes e que não são argumentos mudos de rotinas.

Um outro objeto automático, relacionado com variáveis de caracteres, surge através do comprimento variável de um caractere:

Programa 8.2: Exemplifica o uso de matrizes de forma assumida.

```
program prog_sub
implicit none
real, dimension(0:9,10) :: a
real, dimension(5,5)    :: c
real     :: valor_max
integer  :: i
INTERFACE
   subroutine sub(ra, rb, rc, max_a)
   real, dimension(:,:), intent(in)  :: ra, rb
   real, dimension(:,:), intent(out) :: rc
   real, intent(out)                 :: max_a
   end subroutine sub
END INTERFACE
!
a= reshape(source=(/(cos(real(i)), i= 1,100)/), shape= (/10,10/))
call sub(a, a(0:4,:5), c, valor_max)
print*, "A matriz a:"
do i= 0, 9
   print*, a(i,:)
end do
print*, ""
print*, "O maior valor em a:", valor_max
print*, ""
print*, "A matriz c:"
do i= 1, 5
   print*, c(i,:)
end do
end program prog_sub
```

```
SUBROUTINE EXEMPLO(PALAVRA1)
IMPLICIT NONE
CHARACTER(LEN= *)             :: PALAVRA1
CHARACTER(LEN= LEN(PALAVRA1)) :: PALAVRA2
...
```

como um exemplo. Se este objeto consiste em uma função de caractere de tamanho variável, a interface deve ser explícita, como no programa-exemplo `loren` abaixo.

A definição dos limites das dimensões de um objeto automático pode ser realizada também de argumentos mudos ou de variáveis definidas por *associação por uso* ou por *associação ao hospedeiro. Associação por uso* ocorre quando variáveis globais públicas declaradas no corpo de um módulo são disponibilizadas à rotina

```
program loren
implicit none
character(len= *), parameter :: a= "S um pequeno exemplo."
print*, dobro(a)
CONTAINS
   function dobro(a)
   character(len= *), intent(in) :: a
   character(len= 2*len(a)+2)    :: dobro
   dobro= a // "  " // a
   return
   end function dobro
end program loren
```

através da instrução USE; enquanto que *associação ao hospedeiro* ocorre quando variáveis declaradas em uma unidade de programa são disponibilizadas às suas rotinas internas.

Matrizes automáticas são automaticamente criadas (alocadas) na entrada da rotina e automaticamente dealocadas na saída. Assim, o tamanho das matrizes automáticas pode variar em diferentes chamadas da rotina. Note que não há mecanismo para verificar a disponibilidade de memória para a criação de matrizes automáticas. Caso não haja memória suficiente, o programa é interrompido. Além disso, uma matriz automática não pode aparecer em uma declaração SAVE (seção 8.2.15) ou NAMELIST, ou possuir o atributo SAVE na sua declaração. Além disso, a matriz não pode ser inicializada na sua declaração.

O seguinte programa-exemplo usa matrizes alocáveis, de forma assumida e automáticas:

```
subroutine sub_mat(a,res)
implicit none
real, dimension(:,:), intent(in) :: a !Matriz de forma assumida
real, intent(out) :: res
real, dimension(size(a,1),size(a,2)) :: temp !Matriz automtica
!
temp= sin(a)
res= minval(a+temp)
return
end subroutine sub_mat
```

```
program matriz_aut
implicit none
real, dimension(:,:), allocatable :: a
real     :: res
integer :: n, m, i
INTERFACE
   subroutine sub_mat(a,res)
   real, dimension(:,:), intent(in) :: a
   real, intent(out) :: res
   real, dimension(size(a,1),size(a,2)) :: temp
   end subroutine sub_mat
END INTERFACE
print*, "Entre com dimenses da matriz:"
read*, n,m
allocate(a(n,m))
a= reshape(source=(/(tan(real(i)), i= 1,n*m)/), shape= (/n,m/))
print*, "Matriz a:"
do i= 1, n
   print*, a(i,:)
end do
call sub_mat(a,res)
print*, ""
print*, "O menor valor de a + sin(a) :",res
end program matriz_aut
```

8.2.12 sub-programas como argumentos de rotinas

Até este ponto, os argumentos de uma rotina forma supostos ser variáveis ou expressões. Contudo, uma outra possibilidade é um ou mais argumentos sendo nomes de outras rotinas que serão invocadas. Um exemplo de situação onde é necessário mencionar o nome de uma rotina como argumento de outra é quando a segunda executa a integração numérica da primeira. Desta forma, é possível escrever-se um integrador numérico genérico, que integra qualquer função que satisfaça a interface imposta a ela.

Um exemplo de uso do nome de uma rotina como argumento ocorre abaixo. a função MINIMO calcula o mínimo (menor valor) da função FUNC em um intervalo:

```
FUNCTION MINIMO(A, B, FUNC)
! Calcula o mínimo de FUNC no intervalo [A,B].
REAL :: MINIMO
```

```
REAL, INTENT(IN) :: A, B
INTERFACE
   FUNCTION FUNC(X)
   REAL :: FUNC
   REAL, INTENT(IN) :: X
   END FUNCTION FUNC
END INTERFACE
REAL :: F,X
...
F= FUNC(X) !Invocação da função.
...
END FUNCTION MINIMO
```

Note o uso de um bloco de interface para indicar ao compilador que o nome `FUNC` corresponde a uma função definida pelo usuário.

Em Fortran 77, como não havia o recurso de blocos de interface, a indicação que o nome `FUNC` corresponde a uma função externa é realizada através da declaração `EXTERNAL`. Assim, no exemplo acima, no lugar da interface apareceria:

```
...
REAL FUNC
EXTERNAL FUNC
...
```

para indicar a função externa. A mesma alternativa continua válida em Fortran 90/95, onde, além da declaração, pode-se usar também a palavra-chave `EXTERNAL` como atributo de um nome. Contudo, como já foi mencionado na seção 8.2.7, o simples uso da declaração ou atributo `EXTERNAL` não possibilita o controle nos argumentos desta função, ao contrário do que ocorre quando a interface é explícita.

Um exemplo de programa que chama a função `MINIMO` seria o seguinte:

```
PROGRAM MIN
IMPLICIT NONE
REAL :: MENOR
INTERFACE
   FUNCTION FUN(X)
   REAL :: FUN
   REAL, INTENT(IN) :: X
   END FUNCTION FUN
END INTERFACE
...
MENOR= MINIMO(1.0, 2.0, FUN)
...
END PROGRAM MIN
```

O uso de um bloco de interface não é necessário se a interface de `FUN` for explícita, como no caso de uma rotina de módulo.

No exemplo acima, a função `FUN` não pode ser interna, porque estas não são admitidas quando os seus nomes são usados como argumentos de outras rotinas. Ou seja, somente rotinas externas ou de módulos são aceitas quando os seus nomes são transferidos a outras rotinas como argumentos.

O programa-exemplo das páginas 103 e 104 descreve uma subrotina que gera uma matriz de pontos a partir de uma função fornecida pelo programador.

8.2.13 Funções de valor matricial

Funções, além de fornecer os resultados usuais na forma de valores dos tipos intrínsecos inteiro, real, complexo, lógico e de caractere, podem também fornecer como resultado valores de tipo derivado, matrizes e ponteiros. No caso de funções de valor matricial, o tamanho do resultado pode ser determinado de forma semelhante à maneira como matrizes automáticas são declaradas.

Considere o seguinte exemplo de uma função de valor matricial:

```
! Usa subrotina plota e chama a funo externa Meu_F.
!
program tes_plota
implicit none
integer :: pts, j
real    :: yi, yf
real, dimension(:,:), allocatable :: xy
INTERFACE
   subroutine plota(f, xi, xf, npt, plot)
   integer, intent(in) :: npt
   real, intent(in)    :: xi, xf
   real, dimension(2,npt), intent(out) :: plot
   INTERFACE
      function f(x)
      real :: f
      real, intent(in) :: x
      end function f
   END INTERFACE
   end subroutine plota
!
   function Meu_F(y)
   real :: Meu_F
   real, intent(in) :: y
   end function Meu_F
END INTERFACE
!
print*, "Entre com o nmero de pontos:"
read*, pts
print*, "Entre com os limites inferior e superior:"
read*, yi, yf
allocate(xy(2,pts))
!
call plota(Meu_F, yi, yf, pts, xy)
!
do j= 1, pts
   print*, xy(:,j)
end do
end program tes_plota
```

```
! Funo sen(exp(x)).
function Meu_F(x)
real :: Meu_F
real, intent(in) :: x
!
Meu_F= sin(exp(x))
return
end function Meu_F
```

```
! Gera uma matriz de pontos de uma funo qualquer destinada  plotagem
!  do grfico desta funo.
! Parmetros:
!  f: Funo externa a ser plotada.
!  xi: Ponto inicial (entrada).
!  xf: Ponto final (entrada).
!  npt: Nmero de pontos a ser gerados (entrada)
!  plot: matriz de forma (/ npt, 2 /) contendo as abcissas e ordenadas dos
!        pontos (sada).
!
subroutine plota(f, xi, xf, npt, plot)
implicit none
integer, intent(in) :: npt
real, intent(in)    :: xi, xf
real, dimension(2,npt), intent(out) :: plot
INTERFACE
   function f(x)
   real :: f
   real, intent(in) :: x
   end function f
END INTERFACE
! Variveis locais:
integer :: i
real    :: passo, x
!
passo= (xf - xi)/real(npt - 1)
x= xi
do i= 1, npt
   plot(1,i)= x
   plot(2,i)= f(x)
   x= x + passo
end do
return
end subroutine plota
```

```
PROGRAM TES_VAL_MAT
IMPLICIT NONE
INTEGER, PARAMETER :: M= 6
INTEGER, DIMENSION(M,M) :: IM1, IM2
...
IM2= FUN_VAL_MAT(IM1,1) !Chama função matricial.
...
CONTAINS
   FUNCTION FUN_VAL_MAT(IMA, SCAL)
   INTEGER, DIMENSION(:,:), INTENT(IN) :: IMA
   INTEGER, INTENT(IN) :: SCAL
   INTEGER, DIMENSION(SIZE(IMA,1),SIZE(IMA,2)) :: FUN_VAL_MAT
   FUN_VAL_MAT= IMA*SCAL
   END FUNCTION FUN_VAL_MAT
END PROGRAM TES_VAL_MAT
```

Neste exemplo, o limites das dimensões de `FUN_VAL_MAT` são herdadas do argumento verdadeiro transferido à função e usadas para determinar o tamanho da matriz resultante.

Para que o compilador conheça a forma da matriz resultante, a interface de uma função de valor matricial deve ser explícita em todas as unidades onde esta função é invocada.

8.2.14 Recursividade e rotinas recursivas

Recursividade ocorre quando rotinas chamam a si mesma, seja de forma direta ou indireta. Por exemplo, sejam as rotinas `A` e `B`. Recursividade pode ocorrer de duas formas:

Recursividade direta: `A` invoca `A` diretamente.

Recursividade indireta: `A` invoca `B`, a qual invoca `A`.

Qualquer cadeia de invocações de rotinas com um componente circular (isto é, invocação direta ou indireta) exibe recursividade. Embora recursividade seja uma técnica bonita e sucinta para implementar uma grande variedade de problemas, o uso incorreto da mesma pode provocar uma perda na eficiência da computação do código.

Fortran 77

Em Fortran 77, recursividade direta não é permitida. Qualquer tentativa de forçar uma rotina a chamar a si própria resultará em erro de compilação. Em grande parte dos textos, afirma-se que recursividade indireta também não é possível[3]. Contudo, recursividade pode ser simulada em Fortran 77 quando uma rotina chama a si própria usando não o seu nome real mas através de um argumento mudo, como no exemplo abaixo:[4]

```
      PROGRAM MAIN
      INTEGER N, X
      EXTERNAL SUB1
      COMMON /GLOBALS/ N
      X= 0
      PRINT*, 'Entre número de repetições:'
      READ(*,*) N
      CALL SUB1(X, SUB1)
      END PROGRAM MAIN
------------------------------------------
      SUBROUTINE SUB1(X, SUBMUDO)
      INTEGER N, X
      EXTERNAL SUBMUDO
      COMMON /GLOBALS/ N
      IF(X .LT. N)THEN
         X= X + 1
         PRINT*, 'X=', X
         CALL SUBMUDO(X, SUBMUDO)
      END IF
      RETURN
      END SUBROUTINE SUB1
```

Como a subrotina `SUB1` não sabe que ao invocar o argumento mudo `SUBMUDO`, o qual é o nome de uma rotina externa, vai estar na verdade invocando a si própria, o compilador não irá gerar mensagem de erro e o código irá funcionar em grande parte das plataformas. Por exemplo, se o valor de `N` for definido igual a 5, o resultado será

```
X = 1
X = 2
...
X = 5
```

o que demonstra que é possível simular-se recursividade mesmo em Fortran 77.

Fortran 90/95

Em Fortran 90/95, por outro lado, recursividade é suportada como um recurso explícito da linguagem.

[3] Ver, por exemplo, o texto de Clive Page [7, seção 4.3].
[4] Referência: Fortran Examples http://www.esm.psu.edu/~ajm138/fortranexamples.html, consultada em 07/06/2005.

Recursividade direta. Para fins de eficiência na execução do código, rotinas recursivas devem ser explicitamente declaradas usando-se a palavra-chave

```
RECURSIVE SUBROUTINE ...
```

ou

```
RECURSIVE FUNCTION ...
```

A declaração de uma função recursiva é realizada com uma sintaxe um pouco distinta da até então abordada: uma função explicitamente declarada recursiva deve conter também a palavra-chave `RESULT`, a qual especifica o nome de uma variável à qual o valor do resultado do desenvolvimento da função deve ser atribuído, em lugar do nome da função propriamente dito.

A palavra-chave `RESULT` é necessária, uma vez que não é possível usar-se o nome da função para retornar o resultado, pois isso implicaria em perda de eficiência no código. De fato, o nome da palavra-chave deve ser usado tanto na declaração do tipo e espécie do resultado da função quanto para atribuição de resultados e em expressões escalares ou matriciais. Esta palavra-chave pode também ser usada para funções não recursivas.

Cada vez que uma rotina recursiva é invocada, um conjunto novo de objetos de dados locais é criado, o qual é eliminado na saída da rotina. Este conjunto consiste de todos os objetos definidos no campo de declarações da rotina ou declarados implicitamente, exceto aqueles com atributos `DATA` ou `SAVE` (ver seção 8.2.15). Funções de valor matricial recursivas são permitidas e, em algumas situações, a chamada de uma função recursiva deste tipo é indistingüível de uma referência a matrizes.

O exemplo tradicional do uso de rotinas recursivas consiste na implementação do cálculo do fatorial de um inteiro positivo. A implementação é realizada a partir das seguintes propriedades do fatorial:

$$0! = 1; \quad N! = N(N-1)!$$

A seguir, o cálculo do fatorial é implementado tanto na forma de uma função quanto na forma de uma subrotina recursivas:

```
RECURSIVE FUNCTION FAT(N) RESULT (N_FAT)
IMPLICIT NONE
INTEGER :: N_FAT !Define também o tipo de FAT.
INTEGER, INTENT(IN) :: N
IF(N == 0)THEN
   N_FAT= 1
ELSE
   N_FAT= N*FAT(N-1)
END IF
RETURN
END FUNCTION FAT
-----------------------------------------------
RECURSIVE SUBROUTINE FAT(N, N_FAT)
IMPLICIT NONE
INTEGER, INTENT(IN)  :: N
INTEGER, INTENT(INOUT) :: N_FAT
IF(N == 0)THEN
   N_FAT= 1
ELSE
   CALL FAT(N - 1, N_FAT)
   N_FAT= N*N_FAT
END IF
RETURN
END SUBROUTINE FAT
```

Recursividade indireta. Uma rotina também pode ser invocada por recursividade indireta, isto é, uma rotina chama outra a qual, por sua vez, invoca a primeira. Para ilustrar a utilidade deste recurso, supõe-se que se queira realizar uma integração numérica bi-dimensional quando se dispõe somente de um código que executa integração unidimensional. Um exemplo de função que implementaria integração unidimensional é dado a seguir. O exemplo usa um módulo, o qual é uma unidade de programa que será discutido na seção 8.3:

```
FUNCTION INTEGRA(F, LIMITES)
! Integra F(x) de LIMITES(1) a LIMITES(2).
implicit none
REAL, DIMENSION(2), INTENT(IN) :: LIMITES
INTERFACE
   FUNCTION F(X)
   REAL :: F
   REAL, INTENT(IN) :: X
   END FUNCTION F
END INTERFACE
...
INTEGRA= <implementação integração numérica>
RETURN
END FUNCTION INTEGRA
-------------------------------------------
MODULE FUNC
IMPLICIT NONE
REAL :: YVAL
REAL, DIMENSION(2) :: LIMX, LIMY
CONTAINS
   FUNCTION F(XVAL)
   REAL :: F
   REAL, INTENT(IN) :: XVAL
   F= <expressão envolvendo XVAL e YVAL>
   RETURN
   END FUNCTION F
END MODULE FUNC
----------------------------------------
FUNCTION FY(Y)
! Integra em X, para um valor fixo de Y.
USE FUNC
REAL :: FY
REAL, INTENT(IN) :: Y
YVAL= Y
FY= INTEGRA(F, LIMX)
RETURN
END FUNCTION FY
```

Com base nestas três unidades de programa, um programa pode calcular a integral sobre o retângulo no plano (x, y) através da chamada:

```
AREA= INTEGRA(FY, LIMY)
```

8.2.15 Atributo e declaração SAVE

A declaração ou o atributo `SAVE` são utilizados quando se deseja manter o valor de uma variável local em um sub-programa após a saída. Desta forma, o valor anterior desta variável está acessível quando o sub-programa é invocado novamente.

Como exemplo, a subrotina abaixo contém as variáveis locais `CONTA`, a qual conta o número de chamadas da subrotina e é inicializada a zero e a variável `A`, que é somada ao valor do argumento.

```
SUBROUTINE CONTA_SOMA(X)
IMPICIT NONE
REAL, INTENT(IN) :: X
REAL, SAVE :: A
INTEGER :: CONTA= 0 !Inicializa o contador.  Mantém o valor da variável.
...
CONTA= CONTA + 1
IF(CONTA == 1)THEN
   A= 0.0
```

```
ELSE
   A= A + X
END IF
...
RETURN
END SUBROUTINE CONTA_SOMA
```

Neste exemplo, tanto a variável `A` quanto `CONTA` têm o seu valor mantido quando a subrotina é abandonada. Isto é garantido porque a variável `A` é declarada com o atributo `SAVE`. Já a variável `CONTA` não precisa ser declarada com o mesmo atributo porque ela tem o seu valor inicializado na declaração. De acordo com o padrão da linguagem, as variáveis que têm o seu valor inicializado no momento da declaração automaticamente adquirem o atributo `SAVE`.

Alternativamente ao uso do atributo, pode-se declarar uma lista de nomes de variáveis com a mesma propriedade através da declaração `SAVE`. Desta forma, a variável `A`, no exemplo acima, podia ser declarada através das linhas:

```
...
REAL :: A
SAVE :: A
...
```

A forma geral desta declaração é:

```
SAVE [[::] <lista nomes variáveis>]
```

onde uma declaração `SAVE` sem a subseqüente `<lista nomes variáveis>` equivale à mesma declaração aplicada a todos os nomes da rotina, em cujo caso nenhuma outra variável pode ser declarada com o atributo `SAVE`.

O atributo `SAVE` não pode ser especificado para um argumento mudo, um resultado de função ou um objeto automático. O atributo pode ser especificado para um ponteiro, em cuja situação seu status de associação é preservado. Pode ser aplicado também a uma matriz alocável, em cuja situação o status de alocação e valores são preservados. Uma variável preservada em um sub-programa recursivo é compartilhada por todas as instâncias da rotina.

O comando ou atributo `SAVE` podem aparecer no campo de declarações de um programa principal, mas neste caso o seu efeito é nulo. O uso prático deste recurso está restrito às outras unidades de programa.

8.2.16 Funções de efeito lateral e rotinas puras

Na seção 8.2, foi mencionado que funções normalmente não alteram o valor de seus argumentos. Entretanto, em certas situações esta ação é permitida. Funções que alteram o valor de seus argumentos são denominadas *funções de efeito lateral (side-effect functions)*.

Para auxiliar na otimização do código, o padrão da linguagem estabelece uma proibição no uso de funções de efeito lateral quando o uso de tal função em uma expressão altera o valor de um outro operando na mesma expressão, seja este operando uma variável ou um argumento de uma outra função. Caso este tipo de função fosse possível neste caso, a atribuição

```
RES= FUN1(A,B,C) + FUN2(A,B,C)
```

onde ou `FUN1` ou `FUN2`, ou ambas, alterassem o valor de um ou mais argumentos, a ordem de execução desta expressão seria importante; o valor de `RES` seria diferente caso `FUN1` fosse calculada antes de `FUN2` do que seria caso a ordem de cálculo fosse invertida. Exemplos de funções de efeito lateral são dados a seguir:

```
FUNCTION FUN1(A,B,C)
INTEGER :: FUN1
REAL, INTENT(INOUT) :: A
REAL, INTENT(IN)    :: B,C
A= A*A
FUN1= A/B
RETURN
END FUNCTION FUN1
-----------------------------------
```

```
FUNCTION FUN2(A,B,C)
INTEGER :: FUN2
REAL, INTENT(INOUT) :: A
REAL, INTENT(IN)    :: B,C
A= 2*A
FUN2= A/C
RETURN
END FUNCTION FUN2
```

Nota-se que ambas as funções alteram o valor de `A`; portanto, o valor de `RES` é totalmente dependente na ordem de execução da expressão.

Como o padrão da linguagem não estabelece uma ordem para a execução das operações e desenvolvimentos em uma expressão, este efeito lateral é proibido neste caso. Isto possibilita que os compiladores executem as operações na ordem que otimiza a execução do código.

O uso de funções de efeito lateral em comandos ou construtos `FORALL` (seção 6.10), por exemplo, acarretaria em um impedimento severo na otimização da execução do comando em um processador paralelo, efetivamente anulando o efeito desejado pela definição deste recurso.

Para controlar esta situação, o programador pode assegurar ao compilador que uma determinada rotina (não somente funções) não possui efeitos laterais ao incluir a palavra-chave `PURE` à declaração `SUBROUTINE` ou `FUNCTION`:

```
PURE SUBROUTINE ...
---------------------------
PURE FUNCTION ...
```

Em termos práticos, esta palavra-chave assegura ao compilador que:

- se a rotina é uma função, esta não altera os seus argumentos mudos;
- a rotina não altera nenhuma parte de uma variável acessada por associação ao hospedeiro (rotina interna) ou associação por uso (módulos);
- a rotina não possui nenhuma variável local com o atributo `SAVE`;
- a rotina não executa operações em um arquivo externo;
- a rotina não contém um comando `STOP`.

Para assegurar que estes requerimentos sejam cumpridos e que o compilador possa facilmente verificar o seu cumprimento, as seguintes regras adicionais são impostas:

- qualquer argumento mudo que seja o nome de uma rotina e qualquer rotina invocada devem também ser puras e ter a interface explícita;
- as intenções de um argumento mudo qualquer devem ser declaradas, exceto se este seja uma rotina ou um ponteiro, e a intenção deve sempre ser `IN` no caso de uma função;
- qualquer rotina interna de uma rotina pura também deve ser pura;
- uma variável que é acessada por associação ao hospedeiro ou por uso ou é um argumento mudo de intenção `IN` não pode ser o alvo de uma atribuição de ponteiro; se a variável é do tipo derivado com uma componente de ponteiro, ela não pode estar no lado direito de uma atribuição e ela não pode estar associada como o argumento real de um argumento mudo que seja um ponteiro ou que tenha intenções `OUT` ou `INOUT`.

Esta última regra assegura que um ponteiro local não pode causar um efeito lateral.

A principal razão para se permitir subrotinas puras está na possibilidade de seu uso em construtos `FORALL`. Porém, ao contrário de funções, uma subrotina pura pode ter argumentos mudos com intenções `OUT` ou `INOUT` ou atributo `POINTER`. A sua existência também oferece a possibilidade de se fazer chamadas a subrotinas de dentro de funções puras.

Uma rotina externa ou muda que seja usada como rotina pura deve possuir uma interface explícita que a caracterize inequivocamente como tal. Contudo, a rotina pode ser usada em outros contextos sejam o uso de um bloco de interface ou com uma interface que não a caracterize como pura. Isto permite que rotinas em bibliotecas sejam escritas como puras sem que elas sejam obrigatoriamente usadas como tal.

Todas as funções intrínsecas (capítulo 7) são puras e, portanto, podem ser chamadas livremente de dentro de qualquer rotina pura. Adicionalmente, a subrotina intrínseca elemental `MVBITS` (seção 7.9.3) também é pura.

O atributo `PURE` é dado automaticamente a qualquer rotina que seja definida com o atributo `ELEMENTAL` (seção 8.2.17).

8.2.17 Rotinas elementais

Na seção 6.7 já foi introduzida a noção de rotinas intrínsecas elementais, as quais são rotinas com argumentos mudos escalares que podem ser invocadas com argumentos reais matriciais, desde que os argumentos matriciais tenham todos a mesma forma (isto é, que sejam conformáveis). Para uma função, a forma do resultado é a forma dos argumentos matriciais.

O Fortran 95 estende este conceito a rotinas não-intrínsecas. Uma rotina elemental criada pelo programador deve ter um dos seguintes cabeçalhos:

```
ELEMENTAL SUBROUTINE ...
-------------------------
ELEMENTAL FUNCTION ...
```

Um exemplo é fornecido abaixo. Dado o tipo derivado `INTERVALO`:

```
TYPE :: INTERVALO
   REAL :: INF, SUP
END TYPE INTERVALO
```

pode-se definir a seguinte função elemental:

```
ELEMENTAL FUNCTION SOMA_INTERVALOS(A,B)
INTRINSIC NONE
TYPE(INTERVALO)              :: SOMA_INTERVALOS
TYPE(INTERVALO), INTENT(IN) :: A, B
SOMA_INTERVALOS%INF= A%INF + B%INF
SOMA_INTERVALOS%SUP= A%SUP + B%SUP
RETURN
END FUNCTION SOMA_INTERVALOS
```

a qual soma dois intervalos de valores, entre um limite inferior e um limite superior. Nota-se que os argumentos mudos são escritos como escalares, mas a especificação `ELEMENTAL` possibilita o uso de matrizes conformáveis como argumentos reais. Neste caso, as operações de soma dos intervalos são realizadas componente a componente das matrizes `A` e `B` da maneira mais eficiente possível.

Uma rotina não pode ser ao mesmo tempo elemental e recursiva.

Uma rotina elemental deve satisfazer todos os requisitos de uma rotina pura; de fato, ela já possui automaticamente o atributo `PURE`. Adicionalmente, todos os argumentos mudos e resultados de função devem ser variáveis escalares sem o atributo `POINTER`. Se o valor de um argumento mudo é usado em uma declaração, especificando o tamanho ou outra propriedade de alguma variável, como no exemplo

```
ELEMENTAL FUNCTION BRANCO(N)
IMPLICIT NONE
INTEGER, INTENT(IN) :: N
CHARACTER(LEN= N)   :: BRANCO
BRANCO= ' '
RETURN
END FUNCTION BRANCO
```

esta função *deve* ser chamada com argumento real *escalar*, uma vez que um argumento real matricial resultaria em uma matriz cujos elementos seriam caracteres de comprimentos variáveis.

Uma rotina externa elemental deve possuir uma interface explícita que sempre a caracterize de forma inequívoca como elemental. Isto é exigido para que o compilador possa determinar o mecanismo de chamada que acomode os elementos de matriz de forma mais eficiente. Isto contrasta com o caso de uma rotina pura, onde a interface nem sempre necessita caracterizar a rotina como pura.

No caso de uma subrotina elemental, se algum argumento real é matriz, todos os argumentos com intenções `OUT` ou `INOUT` devem ser matrizes. No exemplo abaixo,

```
ELEMENTAL SUBROUTINE TROCA(A,B)
IMPLICIT NONE
REAL, INTENT(INOUT) :: A, B
REAL :: TEMP
TEMP= A
A= B
B= TEMP
RETURN
END SUBROUTINE TROCA
```

chamar esta subrotina com um argumento escalar e um matricial está obviamente errado, uma vez que o mecanismo de distribuir o valor da variável escalar por uma matriz conformável com o outro argumento não pode ser aplicado aqui.

Se a referência a uma rotina genérica (seção 8.3.4) é consistente tanto com uma rotina elemental quanto com uma não-elemental, a segunda versão é invocada, pois espera-se que a versão elemental de uma rotina rode, em geral, mais lentamente.

Finalmente, uma rotina elemental não pode ser usada como um argumento real de outra rotina.

8.3 Módulos

Uma das novidades mais importantes introduzidas pelo Fortran 90 é um novo tipo de unidade de programa denominada *módulo*. Um módulo é um recurso muito poderoso para transferir dados entre subprogramas e para organizar a arquitetura global de um programa grande e complexo.

A funcionalidade de um módulo pode ser explorada por qualquer unidade de programa que deseja fazer uso (através de uma instrução `USE`) dos recursos disponibilizados por este. Os recursos que podem ser incluídos em um módulo são os seguintes:

Declaração de objetos globais. Módulos devem ser empregados no lugar das declarações `COMMON` e `INCLUDE`, comuns no Fortran 77. Se dados globais são necessários, por exemplo para transferir valores entre rotinas, então estes são são tornados visíveis sempre que o módulo é usado. Objetos em um módulo podem ser inicializados com valores estáticos, de tal forma que estes mantêm seu valor em diferentes unidades que usam o mesmo módulo.

Blocos de interfaces. Na maior parte das situações é vantajoso congregar todos os blocos de interfaces em um ou poucos módulos e então usar o módulo sempre que uma interface explícita é necessária. Isto pode realizado em conjunto com a cláusula `ONLY`.

Rotinas de módulos. Rotinas podem ser definidas internamente em um módulo, sendo estas tornadas acessíveis a qualquer unidade de programa que `USE` o módulo. Esta estratégia é mais vantajosa que usar uma rotina externa porque em um módulo a interface das rotinas internas é sempre explícita.

Acesso controlado a objetos. Variáveis, rotinas e declarações de operadores podem ter a sua visibilidade controlada por declarações ou atributos de acesso dentro de um módulo. Desta forma, é possível especificar que um determinado objeto seja visível somente no interior do módulo. Este recurso é freqüentemente utilizado para impedir que um usuário possa alterar o valor de um objeto de âmbito puramente interno ao módulo.

Interfaces genéricas. Um módulo pode ser usado para definir um nome genérico para um conjunto de rotinas, com interfaces distintas, mas que executem todas a mesma função. Interfaces genéricas podem ser usadas também para estender a funcionalidade de rotinas intrínsecas.

Sobrecarga de operadores. Um módulo pode ser usado também para redefinir a operação executada pelos operadores intrínsecos da linguagem (`= + - * / **`) ou para definir novos tipos de operadores.

Extensão semântica. Um módulo de extensão semântica é uma coleção de definições de tipos derivados, rotinas e operadores sobrecarregados tais que, quando agregados a uma unidade de programa, permitem que o usuário use esta funcionalidade estendida como se fizesse parte da linguagem padrão.

A forma geral de um módulo é:

```
MODULE <nome módulo>
<declarações variáveis>
```

```
<comandos executáveis>
[CONTAINS
   <rotinas de módulo>]
END [MODULE [<nome módulo>]]
```

8.3.1 Dados globais

Em Fortran, variáveis são usualmente entidades locais. Contudo, em muitas situações é necessário que alguns objetos tenham o seu valor compartilhado entre diferentes unidades de programa. A maneira mais usual de se realizar este compartilhamento é através da lista de argumentos de uma rotina. Entretanto, facilmente surge uma situação onde uma variável não pode ser compartilhada como argumento de um subprograma. Neste momento, é necessária uma outra estratégia para se definir dados globais. Esta necessidade já era corriqueira no Fortran 77.

Fortran 77

A maneira mais usada no Fortran 77 para compartilhar dados globais é através de *blocos common*. Um bloco common é uma lista de variáveis armazenadas em uma área da memória do computador que é identificada através de um nome e que pode ser acessada diretamente em mais de uma unidade de programa.

A necessidade de se criar um bloco common surge, muitas vezes, com o uso de rotinas de bibliotecas de aplicação generalizada, as quais são definidas com um conjunto restrito de argumentos. Muitas vezes, faz-se necessário transferir informação entre estas rotinas ou entre rotinas que usam as bibliotecas de maneira não prevista na lista de argumentos.

Por exemplo, seja a subrotina `INTEG`, a qual realiza a integração numérica unidimensional de uma função `F(x)` qualquer, no intervalo `(A,B)`:

```
      SUBROUTINE INTEG(F,A,B,RES)
      REAL*8 A, B, F, RES, TEMP
      EXTERNAL F
      ...
      TEMP= F(X)
      ...
      RES= <resultado>
      END SUBROUTINE INTEG
```

Caso o programador queira integrar a função $g(x) = \cos(\alpha x)$ no intervalo $[0, 1]$, ele não poderá informar ao integrador o valor de α na lista de argumentos da subrotina `INTEG`. A solução está no uso de um bloco common:

```
      REAL*8 FUNCTION G(X)
      REAL*8 X, ALFA
      COMMON /VAR/ ALFA
      G= COS(ALFA*X)
      RETURN
      END FUNCTION G
```

onde `VAR` é o nome que identifica o espaço de memória onde o valor de `ALFA` será armazenado. Assim, o programa pode realizar a integração usando o mesmo bloco common:

```
      PROGRAM INTEGRA
      REAL*8 G, A, INT
      EXTERNAL G
      COMMON /VAR/ A
      ...
      PRINT*, 'Entre com o valor de alfa:'
      READ(*,*) A
C
      CALL INTEG(G,0.0D0,1.0D0,INT)
C
      ...
      END PROGRAM INTEGRA
```

Note que o bloco VAR aparece tanto no programa quanto na função; porém não há necessidade de ele aparecer na subrotina INTEG.

A forma geral de um bloco common é:

```
COMMON /<nome common>/ <lista variáveis>
```

onde <nome common> é qualquer nome válido em Fortran e <lista variáveis> é uma lista de variáveis que podem ser escalares, elementos de matrizes, seções de matrizes ou matrizes completas. Neste último caso, somente se utiliza o nome da matriz. É claro que a informação acerca da natureza e tipo das variáveis listadas no bloco common deve ser fornecida por declarações adequadas de variáveis.

A grande desvantagem desta forma descentralizada de definir-se dados globais está no fato de que em todas as unidades de programa que utilizam um dado bloco common, as variáveis listadas nele devem concordar em número, ordem, natureza e tipo. Caso o programador esteja desenvolvendo um código complexo, composto por um número grande de unidades que usam o mesmo bloco common, e ele sinta a necessidade de alterar de alguma forma a lista de variáveis em uma determinada unidade, ele deverá buscar em todas unidades restantes que fazem referência ao mesmo bloco e realizar a mesma alteração; caso contrário, mesmo que o programa compile e linke sem erro, é quase certo que as variáveis serão compartilhadas de forma incoerente entre as diferentes unidade de programa e este acabe gerando resultados incorretos. É fácil perceber que este controle torna-se sucessivamente mais difícil à medida que a complexidade do programa aumenta.

Os blocos common são ainda aceitos em Fortran 90/95; porém, o seu uso não é recomendado, pela razão exposta acima.

Fortran 90/95

O uso de um módulo fornece um mecanismo centralizado de definição de objetos globais. Por exemplo, suponha que se queira ter acesso às variáveis inteiras I, J e K e às variáveis reais A, B e C em diferentes unidades de programa. Um módulo que permitira o compartilhamento destas variáveis é o seguinte:

```
MODULE GLOBAIS
IMPLICIT NONE
INTEGER :: I, J, K
REAL    :: A, B, C
END MODULE GLOBAIS
```

Estas variáveis se tornam acessíveis a outras unidades de programa (inclusive outros módulos) através da instrução USE, isto é:

```
USE GLOBAIS
```

A instrução USE é não executável e deve ser inserida logo após o cabeçalho da unidade de programa (PROGRAM, FUNCTION, SUBROUTINE ou MODULE) e antes de qualquer outra instrução não executável, tal como uma declaração de variáveis. Uma unidade de programa pode invocar um número arbitrário de módulos usando uma série de instruções USE. Um módulo pode usar outros módulos, porém um módulo não pode usar a si próprio, seja de forma direta ou indireta.

Assim, o módulo GLOBAIS pode ser usado da seguinte forma:

```
FUNCTION USA_MOD(X)
USE GLOBAIS
IMPLICIT NONE
REAL             :: USA_MOD
REAL, INTENT(IN) :: X
USA_MOD= I*A - J*B + K*C - X
RETURN
END FUNCTION USA_MOD
```

O vantagem de um módulo sobre um bloco COMMON para compartilhar objetos globais é evidente. A definição dos nomes, tipos e espécies das variáveis é realizada em uma única unidade de programa, a qual é simplesmente usada por outras, estabelecendo assim um controle centralizado sobre os objetos.

O uso de variáveis de um módulo pode causar problemas se o mesmo nome é usado para diferentes variáveis em partes diferentes de um programa. A declaração USE pode evitar este problema ao permitir a especificação de um nome local distinto daquele definido no módulo, porém que ainda permita o acesso aos dados globais. Por exemplo,

```
...
USE GLOBAIS, R => A, S => B
...
USA_MOD= I*R - J*S + K*C - X
...
```

Aqui, os nomes `R` e `S` são usado para acessar as variáveis globais `A` e `B`, definidas no módulo, caso estes nomes últimos sejam usados para diferentes variáveis na mesma unidade de programa. Os símbolos "`=>`" promovem a ligação do nome local com o nome no módulo. Tudo se passa como se `R` fosse o apelido de `A` e `S` o apelido de B nesta unidade de programa. Os ponteiros em Fortran 90/95 também atuam como apelidos, porém, neste caso, pode-se usar tanto o nome verdadeiro de uma variável quanto o nome de um ponteiro que aponta a ela.

Existe também uma forma da declaração `USE` que limita o acesso somente a certos objetos dentro de um módulo. Este recurso não é exatamente igual ao uso dos atributos `PUBLIC` ou `PRIVATE` (seção 8.3.3), uma vez que ele atua somente na unidade que acessa o módulo, e não sobre o módulo em geral. O recurso requer o uso do qualificador `ONLY`, seguido de dois pontos "`:`" e uma `<lista only>`:

```
USE <nome módulo>, ONLY: <lista only>
```

Por exemplo para tornar somente as variáveis `A` e `C` do módulo `GLOBAIS` acessíveis em uma dada unidade de programa, a declaração fica:

```
...
USE GLOBAIS, ONLY: A, C
...
```

Os dois últimos recursos podem ser também combinados:

```
...
USE GLOBAIS, ONLY: R => A
...
```

para tornar somente a variável `A` acessível, porém com o nome `R`. Uma unidade de programa pode ter mais de uma declaração `USE` referindo-se ao mesmo módulo. Por conseguinte, deve-se notar que que um `USE` com o qualificador `ONLY` não cancela uma declaração `USE` menos restritiva.

Um uso frequente de módulos para armazenar dados globais consiste em definições de parâmetros de espécie de tipo, constantes universais matemáticas e/ou físicas e outros objetos estáticos, como está apresentado no programa-exemplo `mod1.f90` (programa 8.3), o qual ilustra o uso de um módulo para armazenar dados globais.

Torna-se possível, portanto, a criação de módulos destinados a armazenar valores de constantes fundamentais, tanto matemáticas quanto físicas. O módulo `Const_Fund.f90` (programa 8.4) consiste em um exemplo que pode ser empregado em diversas aplicações distintas.

8.3.2 Rotinas de módulos

Rotinas podem ser definidas em módulos e estas são denominadas *rotinas de módulos*. Estas podem ser tanto subrotinas quanto funções e ter a mesma forma de rotinas internas definidas dentro de outras unidades de programa. O número de rotinas de módulo é arbitrário.

Rotinas de módulo podem ser chamadas usando o comando `CALL` usual ou fazendo referência ao nome de uma função. Contudo, estas somente são acessíveis a unidades de programa que fazem uso do módulo através da instrução `USE`.

Uma rotina de módulo pode invocar outras rotinas de módulo contidas no mesmo módulo. As variáveis declaradas no módulo antes da palavra-chave `CONTAINS` são diretamente acessíveis a todas as rotinas deste, por associação ao hospedeiro. Contudo, variáveis declaradas localmente em um determinado sub-programa de módulo são opacas aos outros sub-programas. Caso o módulo invoque outro módulo com uma instrução `USE` antes da palavra-chave `CONTAINS`, estes objetos também se tornam acessíveis a todas as rotinas de módulo. Por outro lado, uma determinada rotina de módulo pode usar localmente outro módulo (não o hospedeiro), em cuja situação os objetos somente são acessíveis localmente. Em suma, uma rotina de módulo possui todas as propriedades de rotinas internas em outras unidades de programa, exceto que uma rotina de módulo pode conter, por sua vez, rotinas internas a ela. Por exemplo, o seguinte módulo é válido:

Programa 8.3: Ilustra uso de módulo para armazenar dados globais.

```
MODULE modu1
implicit none
real, parameter :: pi= 3.1415926536
real, parameter :: euler_e= 2.718281828
END MODULE modu1
!**************************************
program mod1
use modu1
implicit none
real :: x
do
   print*, "Entre com o valor de x:"
   read*, x
   print*, "sen(pi*x)=",sen()
   print*, "ln(e*x)=",ln()
end do
CONTAINS
   function sen()
   real :: sen
   sen= sin(pi*x)
   return
   end function sen
!
   function ln()
   real :: ln
   ln= log(euler_e*x)
   return
   end function ln
end program mod1
```

```
MODULE INTEG
IMPLICIT NONE
REAL :: ALFA  ! Variável global.
CONTAINS
   FUNCTION FF(X)
   REAL :: FF
   REAL, INTENT(IN) :: X
   FF= EXP(-ALFA*X*X)
   FF= FF*X2(X)
   RETURN
   CONTAINS
      FUNCTION X2(Y)
      REAL :: X2
      REAL, INTENT(IN) :: Y
      X2= Y**2
      RETURN
      END FUNCTION X2
   END FUNCTION FF
END MODULE INTEG
```

A outra grande diferença está no fato de que as interfaces das rotinas de módulo são explícitas no âmbito deste. Todavia, quando uma outra unidade de programa usa o módulo, todas os objetos públicos nele contidos se tornam acessíveis a esta, resultando que as interfaces das rotinas de módulo são automaticamente explícitas também para a unidade de programa que o invoca. Portanto, um módulo é considerado a unidade ideal para armazenar grupos de sub-programas criados para desempenhar uma determinada tarefa, juntamente com os objetos globais associados a estes sub-programas.

Programa 8.4: Módulo contendo constantes matemáticas e físicas universais.

```
!*** Module Const_Fund ***
!  Define espcies de tipo padres e constantes universais.
!
MODULE Const_Fund
!
integer, parameter :: i4b = selected_int_kind(9)
integer, parameter :: i2b = selected_int_kind(4)
integer, parameter :: i1b = selected_int_kind(2)
integer, parameter :: sp= kind(1.0)
integer, parameter :: dp= selected_real_kind(2*precision(1.0_sp))
integer, parameter :: qp= selected_real_kind(2*precision(1.0_dp))
complex(kind= dp), parameter :: z1= (1.0_dp,0.0_dp), zi= (0.0_dp,1.0_dp)
real(dp), parameter :: pi=       3.14159265358979323846264338327950288419717_dp
real(dp), parameter :: pid2=     1.57079632679489661923132169163975144209858_dp
real(dp), parameter :: twopi=    6.28318530717958647692528676655900576839434_dp
real(dp), parameter :: rtpi=     1.77245385090551602729816748334114518279755_dp
real(dp), parameter :: sqrt2=    1.41421356237309504880168872420969807856967_dp
real(dp), parameter :: euler_e= 2.71828182845904523536028747135266249775725_dp
real(sp), parameter :: pi_s=     3.14159265358979323846264338327950288419717_sp
real(sp), parameter :: pid2_s=   1.57079632679489661923132169163975144209858_sp
real(sp), parameter :: twopi_s= 6.28318530717958647692528676655900576839434_sp
real(sp), parameter :: rtpi_s=   1.77245385090551602729816748334114518279755_sp
real(sp), parameter :: sqrt2_s= 1.41421356237309504880168872420969807856967_sp
!
!  Constantes fsicas fundamentais (SI), medidas pelo NIST:
!   http://physics.nist.gov/cuu/Constants/index.html
real(dp), parameter :: me= 9.10938215e-31_dp  !Massa de repouso do eltron.
real(dp), parameter :: mp= 1.672621637e-27_dp !Massa de repouso do prton.
real(dp), parameter :: e = 1.602176487e-19_dp !Carga eltrica fundamental.
!
END MODULE Const_Fund
```

Ao se compilar um arquivo contendo um módulo, os compiladores automaticamente geram um novo tipo de arquivo, geralmente com a extensão `*.mod`, onde as informações contidas no módulo, tais como variáveis globais e interfaces, são armazenadas. Quando uma outra unidade de programa faz referência a este módulo, o compilador busca estas informações no arquivo `.mod` . Desta forma, cria-se um mecanismo para verificar se as variáveis e as interfaces estão sendo corretamente utilizadas pela unidade que chama o módulo. Quando o módulo não define um rotina interna, não é necessário guardar o arquivo objeto (`*.o` ou `*.obj`) associado, bastando guardar o arquivo `*.mod`. Por esta razão, muitos compiladores oferecem uma chave extra de compilação através da qual nenhum arquivo objeto é criado, mas somente o arquivo de módulo.

Rotinas de módulo pode ser úteis por diversas razões. Por exemplo, um módulo que define a estrutura de um conjunto particular de dados pode também incluir rotinas especiais, necessárias para operar com estes dados; ou um módulo pode ser usado para conter uma biblioteca de rotinas relacionadas entre si.

Como exemplo, um módulo pode ser usado para "adicionar" variáveis de tipo derivado:

```
MODULE MOD_PONTO
IMPLICIT NONE
TYPE :: PONTO
   REAL :: X, Y
END TYPE PONTO
CONTAINS
   FUNCTION ADPONTOS(P,Q)
   TYPE(POINT) :: ADPONTOS
   TYPE(POINT), INTENT(IN) :: P, Q
   ADPONTOS%X= P%X + Q%X
```

Programa 8.5: Ilustra o uso de rotinas de módulo.

```
MODULE modu2
 implicit none
 real, parameter :: pi= 3.1415926536
 real, parameter :: euler_e= 2.718281828
 real :: x
CONTAINS
    function sen()
    real :: sen
    sen= sin(pi*x)
    return
    end function sen
 !
    function ln()
    real :: ln
    ln= log(euler_e*x)
    return
    end function ln
END MODULE modu2
 !*************************************
program mod2
use modu2
implicit none
do
    print*, "Entre com o valor de x:"
    read*, x
    print*, "sen(pi*x)=",sen()
    print*, "ln(e*x)=",ln()
end do
end program mod2
```

```
    ADPONTOS%Y= P%Y + Q%Y
    RETURN
    END FUNCTION ADPONTOS
END MODULE MOD_PONTO
```

Neste caso, o programa principal usa este módulo:

```
PROGRAM P_PONTO
USE MOD_PONTO
IMPLICIT NONE
TYPE(POINT) :: PX, PY, PZ
...
PZ= ADPONTO(PX,PY)
...
END PROGRAM P_PONTO
```

O recurso avançado de *sobrecarga de operador (operator overloading)* permite redefinir a operação de adição (`+`), por exemplo, dentro do âmbito do módulo `M_PONTO`, de tal forma que o processo de adição de duas variáveis do tipo `PONTO` automaticamente iria chamar a função `ADPONTO`. Desta forma, ao invés do programa chamar esta função, bastaria realizar a operação:

```
PZ= PX + PY
```

Entretanto, este recurso não será discutido aqui.

O programa-exemplo `mod2.f90` (programa 8.5) é igual ao `mod1.f90`, porém usando rotinas de módulo.

8.3.3 Atributos e declarações PUBLIC e PRIVATE

Usualmente, todas as entidades em um módulo estão disponíveis para qualquer unidade de programa que chame este módulo com a instrução `USE`. Contudo, em certas situações é recomendável proibir o uso de certas entidades (objetos globais e/ou rotinas) contidas no módulo para forçar o usuário a invocar as rotinas de módulo que realmente deveriam ser acessíveis a este, ou para permitir flexibilidade no aperfeiçoamento das entidades contidas no módulo sem haver a necessidade de informar o usuário a respeito destes aperfeiçoamentos.

Este controle é exercido através dos atributos ou declarações `PUBLIC` ou `PRIVATE`. Por exemplo, abaixo temos a declarações de variáveis com dois atributos distintos:

```
REAL, PUBLIC :: X, Y, Z
INTEGER, PRIVATE :: U, V, W
```

Dentro deste conjunto de variáveis, somente `X`, `Y` e `Z` são acessíveis à unidade de programa que acessa este módulo.

Outra maneira de se estabelecer o controle de acesso é através de declarações, as quais listam os nomes dos objetos que são públicos ou privados:

```
PUBLIC :: X, Y, Z
PRIVATE :: U, V, W
```

A forma geral da declaração é:

```
PUBLIC [[::] <lista acesso>]
PRIVATE [[::] <lista acesso>]
```

Caso nenhum controle é estabelecido em um módulo, seja através de um atributo ou de uma declaração, todas as entidades têm o atributo `PUBLIC.`

Se uma declaração `PUBLIC` ou `PRIVATE` não possui uma lista de entidades, esta confirma ou altera o acesso padrão. Assim, a declaração

```
PUBLIC
```

confirma o acesso padrão, ao passo que a declaração

```
PRIVATE
```

altera o acesso padrão. Desta forma, uma seqüência de declarações como as abaixo:

```
...
PRIVATE
PUBLIC <lista acesso>
...
```

confere aos nomes na `<lista acesso>` o atributo `PUBLIC` enquanto que todas as entidades restantes no módulo são privadas, podendo ser acessadas somente dentro do âmbito do módulo.

8.3.4 Interfaces e rotinas genéricas

Outro recurso poderoso introduzido pelo Fortran 90 é a habilidade do programador definir suas próprias *rotinas genéricas*, de tal forma que um único nome é suficiente para invocar uma determinada rotina, enquanto que a ação que realmente é executada quando este nome é usado depende do tipo de seus argumentos. Embora possam ser declaradas em quaisquer unidades de programa, rotinas genéricas são usualmente definidas em módulos.

Uma rotina genérica é definido usando-se um bloco interfaces e um nome genérico é usado para todas as rotinas definidas dentro deste bloco de interfaces. Assim, a forma geral é:

```
INTERFACE <nome genérico>
   <bloco interface rotina específica 1>
   <bloco interface rotina específica 2>
   ...
END INTERFACE <nome genérico>
```

onde `<bloco interface rotina específica 1>`, etc, são os blocos das interfaces das rotinas específicas, isto é, que se referem a um dado conjunto de tipos de variáveis, e que são acessadas através do `<nome genérico>`. As rotinas *per se* podem se encontrar em outras unidades; por exemplo, elas podem ser rotinas externas. O uso do `<nome genérico>` no final do bloco somente é permitido a partir do Fortran 95.

Como um módulo é a unidade de programa ideal para armazenar todas estas rotinas específicas relacionadas entre si e como as interfaces das rotinas de módulo são sempre explícitas, estas rotinas genéricas são, em geral, definidas dentro de módulos. Neste caso, a declaração

```
MODULE PROCEDURE <lista nomes rotinas>
```

é incluída no bloco de interface para nomear as rotinas de módulo que são referidas através do nome genérico. Assim, a declaração geral é:

```
INTERFACE <nome genérico>
   [<blocos interfaces>]
   [MODULE PROCEDURE <lista nomes rotinas>]
! Em Fortran 95 blocos de interfaces e declarações MODULE PROCEDURE
!   podem aparecer em qualquer ordem.
END INTERFACE [<nome genérico>]
! Somente em Fortran 95 o nome genérico é aceito aqui.
```

Deve-se notar que todos os nomes na `<lista nomes rotinas>` devem ser de rotinas de módulo acessíveis; portanto, elas não necessariamente devem estar definidas no mesmo módulo onde a interface genérica é estabelecida, bastando que elas sejam acessíveis por associação de uso.

Para demonstrar o poderio de uma interface genérica, será utilizada novamente a subrotina `TROCA`, a qual foi definida primeiramente na página 99 e depois, na forma de uma subrotina elemental, na seção 8.2.17. O módulo `gentroca` na página 120 define o nome genérico de uma série de subrotinas elementais `TROCA` associadas a variáveis dos tipos real, inteiro, lógico e do tipo derivado `ponto`, o qual é definido no mesmo módulo.

Este módulo é utilizado então em duas situações distintas. Na primeira vez, o módulo será utilizado para trocar o valor de duas variáveis **escalares** do tipo `ponto`, como no programa usa_gentroca.f90 na página 121.

Posteriormente, o mesmo módulo será utilizado para trocar os elementos de duas **matrizes** inteiras, como no programa `usa_gt_mat.f90`, também listado na página 121.

Uma declaração `MODULE PROCEDURE` somente é permitida se um `<nome genérico>` é fornecido. Porém, o nome genérico pode ser igual ao nome de uma das rotinas declaradas na `<lista nomes rotinas>`. Em conjunto com esta propriedade, o nome genérico definido em um módulo pode ser o mesmo de outro nome genérico acessível ao módulo, inclusive no caso onde o nome genérico corresponde ao de uma rotina intrínseca. Neste caso, o módulo estará estendendo o intervalo de aplicação de uma rotina intrínseca.

As rotinas às quais são dadas um certo nome genérico devem ser todas ou subrotinas ou funções, incluindo as intrínsecas quando uma rotina intrínseca é estendida. Quaisquer duas rotinas não-intrínsecas associadas ao mesmo nome genérico devem ter argumentos que diferem de tal forma que qualquer invocação é feita de forma inequívoca. As regras são que:

1. uma delas tenha mais argumentos obrigatórios mudos de um tipo, espécie e posto particulares que a outra ou

2. que ao menos uma delas tenha um argumento mudo obrigatório tal que

   (a) corresponda por posição na lista de argumentos a um argumento mudo que não esteja presente na outra, ou esteja presente com um tipo e/ou espécie distinta ou com posto distinto, e

   (b) corresponda por nome a um argumento mudo que não esteja presente na outra, ou presente com tipo e/ou espécie diferente ou com posto diferente.

Para o caso (2), ambas as regras são necessárias para descartar a possibilidade de invocação ambígua por uso de palavras-chave. Como exemplo onde haverá ambigüidade, o exemplo abaixo:

```
!Exemplo de definição ambígua de nome genérico
INTERFACE F
```

```
! Define nome genrico para troca de duas variveis quaisquer.
MODULE gentroca
implicit none
type :: ponto
   real :: x,y
end type ponto
!
INTERFACE troca
   MODULE PROCEDURE troca_ponto, troca_real, troca_int, troca_log
END INTERFACE troca
!
CONTAINS
   elemental subroutine troca_ponto(a,b)
   type(ponto), intent(inout) :: a, b
   type(ponto) :: temp
   temp= a
   a= b
   b= temp
   end subroutine troca_ponto
!
   elemental subroutine troca_real(a,b)
   real, intent(inout) :: a, b
   real :: temp
   temp= a
   a= b
   b= temp
   end subroutine troca_real
!
   elemental subroutine troca_int(a,b)
   integer, intent(inout) :: a, b
   integer :: temp
   temp= a
   a= b
   b= temp
   end subroutine troca_int
!
   elemental subroutine troca_log(a,b)
   logical, intent(inout) :: a, b
   logical :: temp
   temp= a
   a= b
   b= temp
   end subroutine troca_log
END MODULE gentroca
```

```
program usa_gentroca
use gentroca
type(ponto) :: b= ponto(1.0,0.0), c= ponto(1.0,1.0)
print*, 'Valores originais:'
print*, b,c
call troca(b,c)
print*, 'Novos valores:'
print*, b,c
end program usa_gentroca
```

```
program usa_gt_mat
use gentroca
integer, dimension(2,2) :: b, c
b= reshape(source= (/ ((i+j, i= 1,2), j= 1,2) /), shape= (/2,2/))
c= reshape(source= (/ ((i+j, i= 3,4), j= 3,4) /), shape= (/2,2/))
print*, 'Valores originais:'
do i= 1, 2
   print*, b(i,:),"      ",c(i,:)
end do
call troca(b,c)
print*, 'Novos valores:'
do i= 1, 2
   print*, b(i,:),"      ",c(i,:)
end do
end program usa_gt_mat
```

```
   MODULE PROCEDURE FXI, FIX
END INTERFACE F
CONTAINS
   FUNCTION FXI(X,I)
   REAL :: FXI
   REAL, INTENT(IN) :: X
   INTEGER, INTENT(IN) :: I
   ...
   END FUNCTION FXI
!
   FUNCTION FIX(I,X)
   REAL :: FIX
   REAL, INTEN(IN) :: X
   INTEGER, INTENT(IN) :: I
   ...
   END FUNCTION FIX
```

Neste caso, a chamada ao nome genérico `F` será não-ambígua no caso de argumentos posicionais:

```
A= F(INT,VAR)
```

porém será ambígua para chamada usando palavras-chave:

```
A= F(I= INT, X= VAR)
```

Se uma invocação genérica é ambígua entre uma rotina intrínseca e uma não-intrínseca, esta última é sempre invocada.

8.3.5 Estendendo rotinas intrínsecas *via* blocos de interface genéricos

Como já foi mencionado, uma rotina intrínseca pode ser estendida ou também redefinida. Uma rotina intrínseca *estendida* suplementa as rotinas intrínsecas específicas já existentes. Uma rotina intrínseca *redefinida* substitui uma rotina intrínseca específica existente. Desta forma é possível ampliar um determinado

cálculo de uma função, por exemplo, para interfaces não previstas pelo padrão da linguagem (*extensão*) e/ou substituir o processo de cálculo do valor da função por um código distinto daquele implementado no compilador (*substituição*).

Quando um nome genérico é igual ao nome genérico de uma rotina intrínseca e o nome é declarado com o atributo ou declaração `INTRINSIC` (ou aparece em um contexto intrínseco), a interface genérica estende a rotina genérica intrínseca.

Quando um nome genérico é igual ao nome genérico de uma rotina intrínseca e este nome não possui o atributo `INTRINSIC` (nem possui este atributo pelo contexto), a rotina genérica redefine a rotina genérica intrínseca.

Como exemplo, o módulo `EXT_SINH` abaixo estende o cálculo da função seno hiperbólico para um argumento escalar complexo, enquanto que o padrão da linguagem somente considera argumentos reais (seção 7.5):

```
MODULE EXT_SINH
IMPLICIT NONE
INTRINSIC :: SINH
!
INTERFACE SINH
   MODULE PROCEDURE SINH_C
END INTERFACE SINH
!
CONTAINS
   FUNCTION SINH_C(Z)
   COMPLEX             :: SINH_C
   COMPLEX, INTENT(IN) :: Z
   REAL :: X,Y
   X= REAL(Z)
   Y= AIMAG(Z)
   SINH_C= CMPLX(SINH(X)*COS(Y),COSH(X)*SIN(Y))
   RETURN
   END FUNCTION SINH_C
END MODULE EXT_SINH
```

8.4 Âmbito *(Scope)*

Já foi mencionado neste texto, em diversos lugares, o *âmbito* de um certo nome de variável. O âmbito de um objeto nomeado ou de um rótulo é o conjunto de *unidades de âmbito*, que não se sobrepõe, onde o nome deste objeto pode ser usado sem ambigüidade.

Uma unidade de âmbito é qualquer um dos seguintes:

- uma definição de tipo derivado;
- o corpo de um bloco de interfaces de rotinas, excluindo quaisquer definições de tipo derivado e blocos de interfaces contidos dentro deste, ou
- uma unidade de programa ou sub-programa, excluindo definições de tipo derivado, blocos de interfaces e rotinas internas contidas dentro desta unidade.

8.4.1 Âmbito dos rótulos

Toda unidade de programa ou sub-programa, interno ou externo, tem seu conjunto independente de rótulos. Portanto, o mesmo rótulo pode ser usado em um programa principal e em seus sub-programas internos sem ambigüidade.

Assim, o âmbito de um rótulo é um programa principal ou sub-programa, excluindo quaisquer sub-programas internos que eles contenham. O rótulo pode ser usado sem ambigüidade em qualquer ponto entre os comandos executáveis de seu âmbito.

8.4.2 Âmbito dos nomes

O âmbito de um nome declarado em uma unidade de programa estende-se do cabeçalho da unidade de programa ao seu comando `END`. O âmbito de um nome declarado em um programa principal ou rotina externa estende-se a todos os sub-programas que eles contêm, exceto quando o nome é redeclarado no sub-programa.

O âmbito de um nome declarado em uma rotina interna é somente a própria rotina, e não os outros sub-programas internos. O âmbito do nome de um sub-programa interno e do número e tipos dos seus argumentos estende-se por toda a unidade de programa que o contém, inclusive por todos os outros sub-programas internos.

O âmbito de um nome declarado em um módulo estende-se a todas as unidades de programa que usam este módulo, exceto no caso em que a entidade em questão tenha o atributo `PRIVATE`, ou é renomeada na unidade de programa que usa o módulo ou quando a instrução `USE` apresenta o qualificador `ONLY` e a entidade em questão não esteja na `<lista only>`. O âmbito de um nome declarado em um módulo estende-se a todos os sub-programas internos, excluindo aqueles onde o nome é redeclarado.

Considerando a definição de unidade de âmbito acima,

- Entidades declaradas em diferentes unidades de âmbito são sempre distintas, mesmo que elas tenham o mesmo nome e propriedades.
- Dentro de uma unidade de âmbito, cada entidade nomeada deve possuir um nome distinto, com a exceção de nomes genéricos de rotinas.
- Os nomes de unidades de programas são globais; assim, cada nome deve ser distinto dos outros e distinto de quaisquer entidades locais na unidade de programa.
- O âmbito do nome de uma rotina interna estende-se somente por toda a unidade de programa que a contém.
- O âmbito de um nome declarado em uma rotina interna é esta rotina interna.

Este conjunto de regras resume a definição de âmbito de um nome.

Nomes de entidades são acessíveis por *associação ao hospedeiro* ou *associação por uso* quando:

Associação ao hospedeiro. O âmbito de um nome declarado em uma unidade de programa estende-se do cabeçalho da unidade de programa ao comando `END.`

Associação por uso. O âmbito de um nome declarado em um módulo, o qual não possui o atributo `PRIVATE`, estende-se a qualquer unidade de programa que usa o módulo.

Nota-se que ambos os tipos de associação não se estende a quaisquer rotinas externas que possam ser invocadas e não incluem quaisquer rotinas internas onde o nome é redeclarado.

Um exemplo contendo 5 unidades de âmbito é ilustrado a seguir:

```
MODULE AMBITO1                ! Ambito 1
...                           ! Ambito 1
CONTAINS                      ! Ambito 1
   SUBROUTINE AMBITO2         ! Ambito 2
      TYPE :: AMBITO3         ! Ambito 3
         ...                  ! Ambito 3
      END TYPE AMBITO3        ! Ambito 3
      INTERFACE               ! Ambito 2
        ...                   ! Ambito 4
      END INTERFACE           ! Ambito 2
      ...                     ! Ambito 2
   CONTAINS                   ! Ambito 2
      FUNCTION AMBITO5(...)   ! Ambito 5
      ...                     ! Ambito 5
      END FUNCTION AMBITO5    ! Ambito 5
   END SUBROUTINE AMBITO2     ! Ambito 2
END MODULO AMBITO1            ! Ambito 1
```

Capítulo 9

Comandos de Entrada/Saída de Dados

O Fortran 90/95 possui um conjunto rico de instruções de entrada/saída (E/S) de dados. Entretanto, este capítulo irá apresentar apenas um conjunto de instruções que implementam um processo básico de E/S.

Muitos programas necessitam de dados iniciais para o seu processamento. Depois que os cálculos estiverem completos, os resultados naturalmente precisam ser impressos, mostrados graficamente ou salvos para uso posterior. Durante a execução de um programa, algumas vezes há uma quantidade grande de dados produzidos por uma parte do programa, os quais não podem ser todos armazenados na memória de acesso aleatório (RAM) do computador. Nesta situação também se usam os comandos de E/S da linguagem.

O Fortran 90/95 possui muitos comandos de E/S. Os mais utilizados são:

- `OPEN`
- `READ`
- `WRITE`
- `PRINT`
- `CLOSE`
- `REWIND`
- `BACKSPACE`

estes comandos estão todos embutidos dentro da linguagem a qual possui, adicionalmente, recursos de formatação que instruem a maneira como os dados são lidos ou escritos.

Até este ponto, todos os exemplos abordados executaram operações de E/S de dados com teclado (entrada) e terminal (saída). Adicionalmente, Fortran 90/95 permite que diversos outros objetos, como arquivos de dados, estejam conectados a um programa para leitura e/ou escrita. Neste capítulo, serão abordados os processos de E/S em arquivos de dados, por ser este o uso mais frequente deste recurso. Outros tipos de usos incluem saída direta de dados em impressoras, *plotters*, programas gráficos, recursos de multimeios, *etc.*

Devido à variedade de usos dos comandos de E/S, é interessante que se faça inicialmente uma introdução simples ao assunto, abordando os usos mais frequentes dos processos com arquivos externos, para posteriormente entrar-se em maiores detalhes. Com este intuito, a seção 9.1 contém esta introdução rápida, ao passo que as seções posteriores (9.4 – 9.12) abordam o assunto de uma forma bem mais abrangente.

9.1 Comandos de Entrada/Saída: introdução rápida

Na seção 2.3 na página 13 já foram apresentadas a entrada (teclado) e a saída (tela do monitor) padrões do Fortran. Usado desta forma, o recurso é demasiado limitado. O programador possui controle muito restrito, por exemplo, sobre onde, na tela, o resultado deve ser apresentado e sobre o número de casas decimais que deve ser utilizado para representar o valor. Adicionalmente, o programador está limitado ao teclado para o fornecimento de parâmetros ao programa.

Um uso mais desejável dos recursos de E/S do Fortran consiste na habilidade de ler e gravar dados, por exemplo, em arquivos residentes no disco rígido do computador ou em outra mídia à qual o mesmo tem

acesso. Nesta seção, o acesso de E/S a arquivos será brevemente discutido na forma de tarefas atribuídas ao programador.

Tarefa 1. Saída formatada na tela do monitor

Considera-se inicialmente o seguinte programa que realiza saída no formato livre na tela:

```
program F_out
implicit none
integer, parameter :: dp= SELECTED_REAL_KIND(15,300)
real(dp) :: a, b, c
a= 1.0_dp/7.0_dp
b= sqrt(2.0_dp)
c= 4*atan(1.0_dp)
print*, a, b, c
end program F_out
```

Os resultados deste programa são apresentados da seguinte maneira na tela do monitor:

```
␣␣0.142857142857143␣␣␣␣␣␣␣␣1.41421356237310␣␣␣␣␣␣␣␣3.14159265358979
```

Observa-se que há um número grande de espaços em branco antes e entre os valores dos resultados e todos são apresentados com o número total de dígitos representáveis para uma variável real de dupla precisão.

O programa a seguir apresenta os mesmos resultados, porém de maneiras mais organizadas:

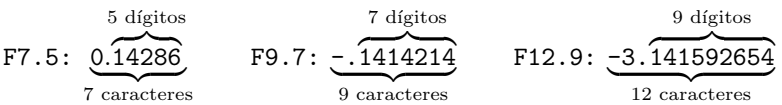
```
program F_out_formatado
implicit none
integer, parameter :: dp= SELECTED_REAL_KIND(15,300)
real(dp) :: a, b, c
a= 1.0_dp/7.0_dp
b= sqrt(2.0_dp)
c= 4*atan(1.0_dp)
print"(f7.5,1x,f9.7,1x,f11.9)", a, b, c
print"(f7.7,1x,f9.9,1x,e11.5)", a, b, c
print"('a= ',f7.5,2x,'b= ',f9.7,3x,'c= ',f11.9)", a, b, c
end program F_out_formatado
```

Agora, os resultados são:

```
␣0.14286␣1.4142136␣3.141592654
␣*******␣*********␣0.31416E+01
␣a=␣0.14286␣␣␣b=␣1.4142136␣␣␣␣c=␣3.141592654
```

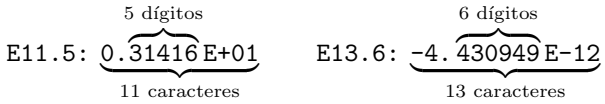
Nos três resultados, o segundo argumento dos comandos `WRITE` indica a formatação na saída, conforme determinada pelo especificador `FMT=`,[1] o qual não é obrigatório se o formato dos dados aparecer como segundo elemento de um comando de E/S. A formatação dos dados de saída é determinada pela sequência de descritores de edição (`X`, `F` e `E`), os quais têm os seguintes significados:

- Os descritores `1X`, `2X` e `3X` indicam quantos espaços em branco (respectivamente 1, 2 e 3) devem ser deixados na saída dos dados.
- Os descritores `F` e `E` são descritores de edição de dados e eles se aplicam ao dados na lista de variáveis (`A`, `B` e `C`) na mesma ordem em que aparecem, ou seja, na saída 1, `F7.5` determina a edição da variável `A`, `F9.7` determina a adição de `B` e `F11.9` determina a edição de `C`.
- O descritor `F7.5` indica que a variável deve ser impressa no formato de ponto flutuante, com um total de 7 algarismos alfanuméricos, contando o ponto decimal e o sinal (se houver), destinando 5 dígitos para a parte fracionária da variável `A`. Exemplos de números de ponto flutuante válidos são:



[1] Discutido em detalhes na seção 9.8 na página 141.

- O descritor `E11.5`, que aparece na saída 2 determinando o formato de saída da variável `C`, indica a saída na forma de ponto flutuante com parte exponencial. O resultado impresso (`0.31416E+01`) possui uma extensão total de 11 algarismos alfanuméricos, contando o ponto decimal, o sinal (se houver), o símbolo da parte exponencial (`E`), o sinal da parte exponencial (`+`) e o seu valor, com duas casas decimais (`01`). Restam, então, somente 5 dígitos para a parte fracionária. Caso a extensão total do descritor, descontado o número de dígitos na parte fracionária, não for suficiente para imprimir a parte exponencial e mais o sinal, ocorre um *estouro de campo*, indicado pelos asteriscos nos resultados das variáveis `A` e `B`. Exemplos válidos de números no formato exponencial são:



- Finalmente, as constantes de caractere `'A= '`, `'B= '` e `'C= '` que aparecem na formatação da saída 3 são impressas *ipsis literis* na saída padrão e na mesma ordem em que aparecem, relativamente a si próprias e às variáveis.

Uma exposição completa de todos os descritores é realizada na seção 9.9 na página 141.

Tarefa 2. Entrada formatada a partir do teclado

Entrada formatada a partir do teclado não é uma tarefa muito prática, pois uma instrução do tipo

```
READ(*,FMT='(1X,F7.5,5X,F9.6,3X,F11.8)')A, B, C
```

iria requerer a digitação dos valores das variáveis exatamente nas posições assinaladas e com as extensões e partes fracionárias exatamente como determinadas pelos descritores. Por exemplo, se `A= 2.6`, `B= -3.1` e `C= 10.8`, seria necessário digitar

`␣␣2.60000␣␣␣␣␣-3.100000␣␣␣10.80000000`

para preencher todos os dígitos e espaços determinados pelos descritores de formatação.

Tarefa 3. Saída de dados em um arquivo

Da mesma maneira que um programa-fonte é gravado em um arquivo situado no disco rígido do computador, é possível gravar dados em arquivos a partir de um programa ou, de forma alternativa, ler dados contidos em arquivos situados no disco.

Para possibilitar acesso de E/S em um arquivo é necessário que o programador:

- Identifique o nome do arquivo, fornecendo o caminho completo, caso o arquivo resida em um diretório (pasta) distinto do programa executável.
- Informe ao sistema sobre o tipo de acesso e uso que será feito do arquivo.
- Associe instruções individuais de leitura ou gravação com o arquivo em uso. É possível existir mais de um arquivo simultaneamente acessível para E/S, além das interfaces já utilizadas (teclado e monitor).
- Quando as operações de E/S estiverem concluídas, é necessário instruir ao sistema não é mais necessário acessar o(s) arquivo(s).

O programa exemplo a seguir grava informações formatadas ao arquivo `EX1.DAT`. A extensão `.DAT` simplesmente identifica o arquivo como sendo de dados. Se o arquivo não existir, este é criado e tornado acessível; se ele existe, então os dados previamente contidos nele podem ser substituídos por novos.

```
program Arq_Sai
implicit none
integer, parameter :: dp= SELECTED_REAL_KIND(15,300)
real(dp) :: a, b, c
a= 1.0_dp/7.0_dp
b= sqrt(2.0_dp)
c= 4*atan(1.0_dp)
open(10, file="ex1.dat")
write(10, fmt="(f7.5,1x,f9.7,1x,f11.9)") a, b, c
close(10)
end program Arq_Sai
```

Alterando os valores das variáveis `A`, `B` e `C`, altera-se os valores gravados no arquivo `EX1.DAT`.

Os comandos-chave neste exemplo são:

`OPEN(10, FILE="EX1.DAT")`. O comando `OPEN` habilita acesso ao arquivo `EX1.DAT` para o sistema. O arquivo está situado no mesmo diretório que o programa e este será criado caso não exista previamente, sendo possível então a gravação no mesmo, ou, caso ele já exista, terá acesso de leitura e/ou escrita.

O número `11` indica a unidade lógica que será acessada. Doravante, o arquivo `EX1.DAT` será sempre identificado pelo programa através deste número.

O modo como o acesso ao arquivo é feito pode ser alterado de diversas maneiras. A seção 9.5 descreve todos os especificadores possíveis do comando `OPEN`.

`WRITE(10, FMT="(F7.5,1X,F9.7,1X,F11.9)") A, B, C`. Este comando é aqui utilizado no lugar de `PRINT` para realizar um processo de escrita formatada no arquivo associado à unidade lógica `10`. A formatação é fornecida pela lista de descritores de formatação que seguem a palavra-chave `FMT=`. O papel destes descritores já foi esclarecido.

O comando essencialmente grava os valores das variáveis `A`, `B` e `C` de maneira formatada no arquivo associado à unidade `10`.

`CLOSE(10)`. Finalmente, este comando informa ao computador que não será mais necessário acessar o arquivo associado à unidade lógica `10` e que tal associação deve ser interrompida.

Tarefa 4. Leitura/escrita de dados em arquivos. Versão 1: número conhecido de linhas

O programa a seguir lê os dados gravados no arquivo `EX1.DAT`, criado a partir do programa `ARQ_SAI`, e usa estes dados para definir valores de outros, os quais serão gravados no arquivo `EX2.DAT`.

```
program Arq_Sai_2
implicit none
integer, parameter :: dp= SELECTED_REAL_KIND(15,300)
real(dp) :: a, b, c ! Dados de entrada.
real(dp) :: d, e, f ! Dados de saida.
open(10, file="ex1.dat", status="old")
open(11, file="ex2.dat", status="new")
read(10, fmt="(f7.5,1x,f9.7,1x,f11.9)") a, b, c
print*, "Os valores lidos foram:"
print*, "a= ",a
print*, "b= ",b
print*, "c= ",c
d= a + b + c
e= b**2 + cos(c)
f= sqrt(a**2 + b**3 + c**5)
write(11, fmt="(3(e11.5,1x))")d, e, f
end program Arq_Sai_2
```

No programa `ARQ_SAI_2` aparecem as seguintes instruções que ainda não foram discutidas:

`STATUS="OLD/NEW"`. Estas cláusulas, que aparecem nos comandos `OPEN`, indicam o status exigido para o arquivo a ser acessado. `STATUS="OLD"` indica que o arquivo deve existir no momento de acesso. Caso isto não aconteça, ocorre uma mensagem de erro e o programa é interrompido. `STATUS="NEW"` indica que o arquivo não deve existir previamente e, então, deve ser criado. Novamente, se estas condições não se cumprirem, o programa pára.

`READ(10, FMT="(F7.5,1X,F9.7,1X,F11.9)") A, B, C`. Observa-se aqui o uso de uma entrada formatada com o comando `READ`. A formatação das variáveis `A, B` e `C` já foi discutida. Deve-se enfatizar novamente que a formatação especificada pelo `FMT=` deve possibilitar a leitura correta dos dados no arquivo associado à unidade lógica `10`.

`WRITE(11, FMT="(3(E11.5,1X))") D, E, F`. O aspecto ainda não abordado neste comando `WRITE` é o uso do *contador de repetição* `3`, o qual indica que as variáveis `D`, `E` e `F` devem ser gravado no arquivo associado à unidade `11` com a formatação dada por `"E11.5,1X"` repetido 3 vezes consecutivas.

Finalmente, deve-se mencionar que os comandos `CLOSE(10)` e `CLOSE(11)` não foram utilizados. Isto é permitido, uma vez que, neste caso, a associação das unidades lógicas aos arquivos será interrompida com o final da execução do programa. Contudo, em certas situações é necessário associar uma unidade lógica previamente empregada a um outro arquivo, o que neste caso obriga o programador a fazer uso do comando `CLOSE`.

Modificações posteriores no programa necessitarão também a eliminação do arquivo `EX2.DAT`, devido à cláusula `STATUS="NEW"`.

Tarefa 5. Leitura/escrita de dados em arquivos. Versão 2: número desconhecido de linhas

Quando o número de linhas no arquivo de entrada não é previamente conhecido, é necessário um procedimento diferente daquele adotado na Tarefa 4. O programa 9.1 ilustra dois possíveis métodos a ser adotados, os quais utilizam as opções `IOSTAT` ou `END` do comando `READ` (seção 9.6). Em ambos, aloca-se inicialmente uma matriz temporária com o posto adequado e com um número tal de elementos que seja sempre maior ou igual ao número máximo de linhas que podem estar contidas no arquivo de entrada. Após a leitura dos dados na matriz temporária, o número de linhas no arquivo tornou-se conhecido e aloca-se, então, uma matriz definitiva com o número correto de elementos. O segundo exemplo faz uso de um rótulo (seção 5.1.1), que embora não seja um procedimento recomendado em Fortran 90/95, é completamente equivalente ao primeiro exemplo.

Tarefa 6. Leitura/escrita de dados em arquivos. Versão 3: número desconhecido de linhas usando listas encadeadas

Existem maneiras mais inteligentes de realizar esta tarefa, sem que seja necessário definir-se uma matriz temporária. Porém, estas maneiras envolvem o uso de ponteiros e listas encadeadas *(linked lists)*, os quais são recursos mais avançados e que não foram ainda abordados nesta apostila. Nesta seção será apresentado o uso básico de uma lista encadeada para executar esta tarefa.

Uma lista encadeada é uma série de variáveis de um tipo derivado, o qual é definido com um componente de ponteiro, do mesmo tipo derivado, o qual irá apontar para o elemento seguinte da lista. Cada elemento desta lista, denominado *nodo*, contém um conjunto de dados de diferentes tipos que devem ser armazenados e pelo menos um ponteiro que irá localizar o nodo seguinte. A estrutura básica do tipo derivado que pode ser usado para se criar uma lista encadeada é a seguinte:

```
TYPE :: LINK
   REAL :: DADO
   TYPE(LINK), POINTER :: NEXT => NULL()
END TYPE LINK
```

Neste exemplo, o tipo derivado `link` é declarado. Seus componentes são uma variável real (`dado`), a qual irá armazenar o valor lido e um ponteiro do tipo `link`, denominado `next` (próximo). Este ponteiro está inicialmente com o status de dissociado, através do uso da função `null()` (seção 7.16), e a sua presença serve para localizar o próximo elemento da lista, funcionando neste caso como os elos de uma cadeia.

Usualmente, são declarados dois ponteiros do tipo derivado `link`, um denominado *raiz*, o qual irá servir para identificar-se o início da lista (ou o primeiro nodo) e o outro denominado *corrente*, que irá servir para identificar o nodo presentemente em uso. Portanto, uma declaração semelhante a

```
TYPE(LINK), POINTER :: RAIZ, CORRENTE
```

se faz necessária. O programa 9.2 ilustra a aplicação de uma lista encadeada para este fim.

Deve-se notar aqui que no programa 9.2 não foi tomada nenhuma providência para liberar o espaço de memória ocupado pela lista encadeada. Esta tarefa pode ser executada pela chamada de uma subrotina recursiva que irá percorrer todos os elos da cadeia até seu final, dealocando consecutivamente, do final para o início, o espaço ocupado pela mesma. Este tipo de providência é importante se este recurso for empregado diversas vezes ou se a memória ocupada pela lista for uma fração substancial da memória disponível. Este tipo de situação leva ao que se denomina de *retenção de memória*,[2] isto é, a perda de memória livre por falta de liberação correta de espaço previamente ocupado e que não se faz mais necessário.

Usar uma lista encadeada para este tipo de tarefa é muito mais eficiente que o emprego de uma matriz temporária, como foi realizado na Tarefa 5, pois não é mais necessário cogitar-se o número máximo de valores

[2] Do inglês: *memory leak*.

Programa 9.1: Programa que ilustra leitura de arquivo com número desconhecido de linhas.

```
!************************** PROGRAMA ARQ_SAI_3 ****************************
! Le um arquivo de dados com um numero conhecido de elementos por linha,
!   mas com um numero desconhecido de linhas no formato livre, usando
!   matrizes temporarias.
! A leitura do arquivo e realizada de 2 maneiras distintas:
!   1. Usando a opcao IOSTAT.
!   2. Usando a opcao END.
! O programa entao imprime na tela os dados e o numero de linhas lido.
!
! Autor: Rudi Gaelzer
! Data:  Maio/2008
!
program Arq_Sai_3
implicit none
integer, parameter :: dp= SELECTED_REAL_KIND(15,300)
integer :: controle, i, npt1= 0, npt2= 0
character(len= 1) :: char
real(kind=dp), dimension(:,:), allocatable :: temp1, temp2, matriz1, matriz2
allocate(temp1(2,1000), temp2(2,1000)) ! Aloca matrizes temporarias.
                                       ! Estima um limite superior no numero
                                       !   de linhas.
! Metodo 1: usando opcao IOSTAT.
open(10, file='ex3.dat', status='old')
do
   read(10, *, iostat= controle)temp1(1,npt1+1), temp1(2,npt1+1)
   if(controle < 0) exit               ! Final de arquivo detectado.
   npt1= npt1 + 1
end do
close(10)
allocate(matriz1(2,npt1))
matriz1= temp1(:,:npt1)
write(*, fmt="('Matriz 1 lida:',/)")
print'(2(e10.3,1x))', (matriz1(:,i), i= 1, npt1)
deallocate(temp1)                      ! Libera espaco de memoria ocupado por
                                       !   temp1.
print'(/,a)', 'Pressione ENTER/RETURN para continuar.'
read(*,'(a)')char
! Metodo 2: usando opcao END.
open(10, file='ex3.dat', status='old')
do
   read(10, *, end= 100)temp2(1,npt2+1), temp2(2,npt2+1)
   npt2= npt2 + 1
end do
100 continue                           ! Linha identificada com o rotulo 100.
close(10)
allocate(matriz2(2,npt2))
matriz2= temp2(:,:npt2)
print'("Matriz 2 lida:",/)'
print'(2(e10.3,1x))', (matriz2(:,i), i= 1, npt1)
deallocate(temp2)                      ! Libera espaco de memoria ocupado por
                                       !   temp2.
print'(/,"O numero total de linhas lidas e'':",/, &
      "Matriz 1: ", i4, 3x, "Matriz 2: ",i4)', npt1, npt2
end program Arq_Sai_3
```

Programa 9.2: Programa que ilustra uso de uma lista encadeada para a leitura de um número arbitrário de dados.

```
!************************** PROGRAMA LISTA_ENCADEADA **********************
! Le um um numero arbitrario de dados e os armazena em um vetor,
!  fazendo uso de uma lista encadeada.
! Para interromper a entrada de dados, basta fornecer
!  um caractere nao numerico.
!
! Autor: Rudi Gaelzer - IFM/UFPel
! Data:  Marco/2011.
!
program lista_encadeada
implicit none
type :: link
   real :: dado
   type(link), pointer :: next => null()
end type link
type(link), pointer :: raiz, corrente
integer :: conta= 0, i, fim
real :: temp
real, dimension(:), allocatable :: vetor
!
allocate(raiz)     !Aloca inicio da lista encadeada.
corrente => raiz   !Localiza a posicao da raiz (inicio) da lista.
print*, 'Entre com os dados (digite caractere nao numerico para encerrar):'
do                                  !Le dados do arq. ate final e conta.
   write(*, fmt='("Valor= ")', advance='no')
   read(*, fmt=*, iostat= fim) temp
   if(fim /= 0)then                 !Dados encerrados.  Sair do laco.
      nullify(corrente%next)        !Dissocia nodo posterior.
      exit
   end if
   corrente%dado= temp              !Grava dado lido na lista.
   allocate(corrente%next)          !Aloca espaco para o nodo seguinte.
   corrente => corrente%next        !Aponta para proximo nodo da lista.
   conta= conta + 1
end do
print*, 'Numero de dados lidos:', conta
allocate(vetor(conta))              !Aloca vetor.
corrente => raiz                   !Retorna ao inicio da lista.
do i= 1, conta !Atribui valores lidos ao vetor, na ordem de leitura.
   vetor(i)= corrente%dado
   corrente => corrente%next
end do
write(*, fmt='(/,"Vetor lido:")')
do i= 1, conta                      !Imprime vetor na tela.
   print*, vetor(i)
end do
end program lista_encadeada
```

a ser lidos. O número de nodos da lista encadeada pode crescer de forma arbitrária, sendo a quantitade total de memória o único limite existente. Desta forma, a lista encadeada é um objeto que pode ser empregado em um número arbitrário de situações distintas. De fato, listas simplesmente ou multiplamente encadeadas são instrumentos empregados com frequência em programação orientada a objeto.

Os exemplos abordados nas Tarefas 1 — 6 constituem um conjunto pequeno, porém frequente, de usos dos comando de E/S. A descrição completa destes recursos pode ser obtida nas seções 9.4—9.12 posteriores.

9.2 Declaração NAMELIST

Em certas situações, quando há por exemplo um número grande de parâmetros sendo transferidos em processos de E/S, pode ser útil definir-se uma lista rotulada de valores destinados à transferência. O Fortran 90 introduziu, com este fim, a declaração `NAMELIST`, que define um ou mais *grupos* de variáveis referenciadas com o mesmo nome. Esta declaração é empregada em conjunto com operações de E/S executadas por comandos `READ` e `WRITE`.

A forma geral da declaração, que pode aparecer em qualquer unidade de programa, é:

```
NAMELIST /<nome-grupo-namelist>/ <lista-nomes-variáveis> &
         [[,] /<nome-grupo-namelist>/ <lista-nomes-variáveis>[,] ...]
```

Como se pode notar, é possível definir-se mais de um grupo `NAMELIST` na mesma declaração. O primeiro campo substituível: *<nome-grupo-namelist>*, é o nome do grupo para uso subsequente nos comandos de E/S. A *<lista-nomes-variáveis>*, como o nome indica, lista os nomes das variáveis que compõe o grupo. Não podem constar desta lista nomes de matrizes mudas de forma assumida, ou objetos automáticos, variáveis de caractere de extensão variável, matrizes alocáveis, pointers, ou ser um componente de qualquer profundidade de uma estrutura que seja um ponteiro ou que seja inacessível. Um exemplo válido de *<lista-nomes-variáveis>* é:

```
REAL :: TV, CARPETE
REAL, DIMENSION(10) :: CADEIRAS
...
NAMELIST /ITENS_DOMESTICOS/ CARPETE, TV, CADEIRAS
```

Outro exemplo válido:

```
NAMELIST /LISTA1/ A, B, C /LISTA2/ X, Y, Z
```

É possível também continuar a mesma *<lista-nomes-variáveis>* de um dado grupo em mais de uma declaração contidas no campo de declarações da unidade de programa. Assim,

```
NAMELIST /LISTA/ A, B, C
NAMELIST /LISTA/ D, E, F
```

é equivalente a uma única declaração `NAMELIST` com os 6 nomes de variáveis listados na mesma ordem acima. Além disso, um objeto de um grupo `NAMELIST` pode também pertencer a outros grupos.

Se o tipo, parâmetro de tipo ou forma de uma variável de um grupo é especificado em uma declaração na mesma unidade de programa , esta declaração deve constar antes da declaração `NAMELIST`, ou ser uma declaração de tipo implícito que regule o tipo e espécie das variáveis que iniciam com aquele caractere explicitado na declaração.

Um grupo namelist pode possuir o atributo público ou privado, concedido através das declarações `PUBLIC` ou `PRIVATE`. Contudo, se o grupo possuir o atributo público, nenhum membro seu pode ser declarado com o atributo privado ou possuir componentes privados.

Para se executar uma operação de E/S em algum registro (como um arquivo, por exemplo) que contenha um ou mais grupos namelist, o registro sempre irá iniciar com o caractere `&`, seguido, sem espaço, pelo <*nome-grupo-namelist*>, vindo então a lista dos nomes das variáveis, sendo cada nome seguido por sinais de igual e, então pelo valor da variável, podendo ser precedido ou seguido por espaços em branco. Constantes de caractere devem ser sempre delimitadas em registros de entrada. A lista de nomes e valores de um determinado grupo é sempre encerrada com o caractere `/` inserido fora de uma constante de caractere.

Os comandos `READ` e `WRITE`, ao serem empregados em conjunto com um namelist, não possuem uma lista explícita de variáveis ou constantes a ser transferidas, mas sim o nome do grupo `NAMELIST` como segundo parâmetro posicional ou com o especificador `NML=` contendo o nome do grupo, no lugar do especificador `FMT=`. Um exemplo de registro de entrada é:

```
INTEGER :: NO_DE_OVOS, LITROS_DE_LEITE, QUILOS_DE_ARROZ
NAMELIST /FOME/ NO_DE_OVOS, LITROS_DE_LEITE, QUILOS_DE_ARROZ
READ(5, NML=FOME)
```

O associado exemplo de registro (arquivo) de entrada é:

```
&FOME LITROS_DE_LEITE= 5, NO_DE_OVOS= 12 /
```

Nota-se no registro acima que a ordem de atribuição de valores aos nomes não é necessariamente a mesma da declaração `NAMELIST`. Além disso, a variável `QUILOS_DE_ARROZ` não tem seu valor atribuído no registro. Quando isso acontece com uma ou mais variáveis da lista o valor destas permanece inalterado.

Os valores de uma matriz podem ser atribuídos através do nome da mesma seguida por `=` e a lista de constantes que serão atribuídos aos elementos da matriz de acordo com o ordenamento padrão da linguagem, descrito na seção 6.6.2. Se a variável for de tipo derivado, a lista de valores deve também seguir a ordem e o tipo e espécie dos componentes constantes da declaração do tipo. É possível também definir-se o valor de um elemento isolado ou de uma seção de matriz, ou qualquer sub-objeto, sem alterar os valores dos demais elementos, usando um designador de sub-objeto, tal como um tripleto de subscritos. Neste caso, os elementos do tripleto devem todos ser constantes inteiras escalares, sem parâmetros de espécie de tipo

Na gravação (saída) do registro, todos os componentes do grupo são escritos no arquivo especificado, precedidos por `&` e o nome do grupo, na mesma ordem da declaração da *<lista-nomes-variáveis>*, com os nomes explicitamente escritos em letras maiúsculas. O registro é finalizado com o caractere `/`. Um exemplo de saída de um namelist é:

```
INTEGER :: NUMERO, I
INTEGER, DIMENSION(10) :: LISTA= (/ 14, (0, I= 1,9) /)
NAMELIST /SAI/ NUMERO, LISTA
WRITE(6, NML=SAI)
```

o qual produz o seguinte registro no arquivo associado à unidade lógica `6`:

```
&SAI NUMERO=1 , LISTA=14, 9*0 /
```

Nota-se que os 9 elementos `0` do vetor `LISTA` são registrados usando-se a notação compacta `9*0`. Esta notação também é válida em registros de entrada.

Em um processo de leitura de registros (usando o comando `READ`, por exemplo), não é permitida a mistura de registros em um `NAMELIST` com registros que não pertencem a nenhum grupo, como em uma entrada formatada ou não-formatada. Contudo, em um processo de saída de registros (comando `WRITE`), pode-se misturar os tipos de registros.

Todos os nomes de grupos, nomes de objetos e nomes de componentes são interpretados sem considerar o caso de capitalização. A lista de valores não pode ter um número excedente de ítens, mas ela pode conter um número inferior, como já foi mencionado. Se o objeto é do tipo caractere e o valor tiver uma extensão menor que a declarada, espaços em branco preenchem o restante do campo. Matrizes ou objetos de tamanho zero (seção 6.5) não podem constar no registro de entrada de um `NAMELIST`. Se ocorrer uma múltipla atribuição de um mesmo objeto, o último valor é assumido.

Por fim, comentários podem ser incluídos no registro de entrada em seguida a um nome a uma vírgula que separa os nomes. O comentário é introduzido, como é costumeiro a partir do caractere `!`. Uma linha de comentário, iniciada com o caractere `!`, também é permitida, desde que esta não ocorra em um contexto de uma constante de caractere.

O programa `Testa_NAMELIST` abaixo ilustra praticamente todos os recursos discutidos nesta seção a respeito de listas `NAMELIST`.

```
program Testa_NAMELIST
implicit none
integer, parameter :: dp= SELECTED_REAL_KIND(15,300)
integer :: no_de_ovos, litros_de_leite, quilos_de_arroz= 12, numero
! Note que quilos_de_arroz foi inicializado.
integer, dimension(10) :: lista_int1, lista_int2
real, dimension(10) :: aleat, vet_tes= 1.0
real(dp) :: pi, e_euler, m_electron, c_vacuum
logical :: tes
character(len= 10) :: nome
```

```
namelist /fome/ nome, no_de_ovos, litros_de_leite, quilos_de_arroz
namelist /sai/ numero, lista_int1, lista_int2, vet_tes
namelist /ctes/ tes, pi, e_euler, m_electron, c_vacuum
!
open(10, file= "Dados.ent", status= "old")
open(11, file= "Dados.dat")
read(10, nml= fome)
read(10, nml= sai)
read(10, nml= ctes)
! Mostra na tela os valores lidos.
print'(a)', "Lista /fome/ lida:"
write(*, nml= fome)
print'(/a)', "Lista /sai/ lida:"
write(*, nml= sai)
print'(/a)', "Lista /ctes/ lida:"
write(*, nml= ctes)
print'(/a)', "Para confirmar:"
print'("Vetor vet_tes: ",10(f4.2,x))', vet_tes
print'(/,"lista 1:",10i3)', lista_int1
print'(/,"lista 2:",10i3)', lista_int2
! Altera alguns valores e grava as listas no arquivo.
call random_number(aleat)
print'(/,"Lista de No. aleatorios:",/,10(f7.5,x))', aleat
lista_int1= lista_int2**2
lista_int2= aleat*lista_int2
write(11, nml= sai)
write(11, nml= ctes)
write(11, nml= fome) ! Verifique Dados.dat.
end program Testa_NAMELIST
```

Um exemplo de arquivo de entrada `Dados.ent` que pode ser fornecido ao programa `Testa_NAMELIST` é o seguinte:

```
&fome
litros_de_leite= 10,
no_de_ovos= 24,
nome= "Rancho"
/

&sai
numero= 50,
lista_int1= 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
lista_int2= 3*1, 3*5, 4*8,
vet_tes(4:8)= 5*2.0
/

&ctes
tes= .true.,
PI=         3.14159265358979323846264338327950288419717,
e_euler=    2.71828182845904523536028747135266249775725, !Euler's number.
m_electron= 9.10938215e-28,      !Electron rest mass (g).
c_vacuum=   2.99792458e10        !Speed of light in vacuum (cm/s).
/
```

9.3 Instrução INCLUDE

Em certas situações, durante a elaboração do código-fonte de uma determinada unidade de programa, pode ser útil incluir-se texto oriundo de um lugar distinto, tal como um outro arquivo que contém parte ou o todo de uma unidade de programa. Este recurso é possível fazendo-se uso da instrução `include`:

```
include <constante de caracteres>
```

onde `<constante de caracteres>` não pode conter um parâmetro de espécie do tipo de caracteres. Esta instrução deve ser inserida no código-fonte em uma linha própria e, no momento da compilação do código-fonte, esta linha será substituída pelo material identificado pela `<constante de caracteres>`, de uma maneira que depende do processador. O texto incluído pode, por sua vez, conter outras linhas `include`, as quais serão similarmente substituídas.

Uma linha `include` não pode fazer referência a si própria, direta ou indiretamente. Quando uma linha `include` é substituída, a primeira linha incluída não pode ser uma linha de continuação e a última linha não pode ser continuada. A linha de código contendo a instrução `include` pode ter um comentário ao final, mas não pode ser rotulada ou conter instruções incompletas.

A instrução `include` já estava disponível no Fortran 77 como uma extensão ao padrão da linguagem. Usualmente, a instrução era empregada para assegurar que as ocorrências dos blocos `common` fossem todas idênticas nas unidades que continham as linhas `include`. No Fortran 90/95, os blocos `common` são substituídos por módulos, com diversas vantagens, mas as instruções `include` podem ainda ser úteis quando se deseja compartilhar o mesmo bloco de instruções ou declarações (ou mesmo unidades de programa completas) entre distintos módulos, por exemplo. Contudo, cautela deve ser exercida para evitar a ocorrência de uma definição repetida de uma mesma rotina em diferentes módulos.

9.4 Unidades lógicas

Em Fortran, um arquivo está conectado a uma *unidade lógica* denotada por um número inteiro. Este número deve ser positivo e está frequentemente limitado entre 1 e 100, dependendo do compilador. Cada unidade lógica possui muitas propriedades. Por exemplo,

`FILE`. O nome do arquivo conectado à unidade. O nome é especificado pelo comando `OPEN`.

`ACTION`. Especifica o tipo de ação que pode ser executada sobre o arquivo. Estas compreendem leitura, escrita ou ambas. Se um arquivo é aberto para um tipo de ação e outro tipo é tentado, uma mensagem de erro será gerada. Por exemplo, não é permitida escrita sobre um arquivo aberto somente para leitura.

`STATUS`. Estabelece o status de um arquivo. Estes são: velho *(old)*, novo *(new)* substituição *(replace)*, entre outros. Caso um arquivo tente ser acessado com um status incorreto, uma mensagem de erro é gerada.

`ACCESS`. Forma de acesso a um arquivo: direto ou sequencial. Alguns arquivos pode ser acessados por ambos os métodos, como é o caso de arquivos em discos removíveis ou discos rígidos. Já outras mídias permite acesso somente sequencial, como é o caso de uma fita magnética.

Sequencial. É o acesso usual. Os processos de leitura/escrita principiam no início do arquivo e seguem, linha a linha, até o seu final.

Direto. Cada linha é acessada por um número, denominado *número de gravação (record number)* o qual deve ser especificado no comando de leitura/escrita.

O número máximo de arquivos que podem ser abertos simultaneamente para E/S também é especificado pelo manual do compilador.

9.5 Comando `OPEN`

O comando `OPEN` é usado para conectar um dado arquivo a uma unidade lógica. Frequentemente é possível pré-conectar um arquivo antes que o programa comece a rodar; neste caso, não é necessário usar-se o comando `OPEN`. Contudo, isto depende de conhecimento prévio a respeito dos números-padrão das unidades lógicas para E/S, o que geralmente depende do processador e dos sistema operacional em uso. Para sistemas linux, as unidades pré-conectadas são, usualmente:

Uso	Designação	Número da unidade lógica
Mensagens de erro	stderr	0
Entrada padrão	stdin	5
Saída padrão	stdout	6

Sempre que possível, é recomendado evitar-se o uso de unidade pré-conectadas, o que torna o programa mais portável.

A sintaxe do comando é:

```
OPEN([UNIT=] <int-exp> [,<op-list>])
```

onde <int-exp> é uma expressão escalar inteira que especifica o número da unidade lógica externa e <op-list> é uma lista de especificadores opcional, formada por um conjunto de palavras-chave. Se a palavra-chave "UNIT=" for incluída, ela pode aparecer em qualquer posição no campo de argumentos do comando OPEN. Um especificador não pode aparecer mais de uma vez. Nos especificadores, todas as entidades são escalares e todos os caracteres são da espécie padrão. Em expressões de caractere, todos os brancos precedentes são ignorados e, exceto no caso do especificador "FILE=", quaisquer letras maiúsculas são convertidas às correspondentes letras minúsculas. Os especificadores são:

FILE= <fln>, onde <fln> é uma expressão de caractere que fornece o nome do arquivo. O nome deve coincidir exatamente com o arquivo, inclusive com o caminho, caso o arquivo esteja em um diretório distinto do diretório de trabalho. Se este especificador é omitido e a unidade não é conectada a uma arquivo, o especificador "STATUS=" deve ser especificado com o valor SCRATCH e o arquivo conectado à unidade irá depender do sistema.

IOSTAT= <ios>, onde <ios> é uma variável inteira padrão que é fixada a zero se o comando executou corretamente e a um valor positivo em caso contrário.

ERR= <rótulo-erro>, onde <rótulo-erro> é o rótulo de um comando na mesma unidade de âmbito, para o qual o controle do fluxo será transferido caso ocorra algum erro na execução do comando OPEN.

STATUS= <status>, onde <status> é uma expressão de caracteres que fornece o status do arquivo. O status pode ser um dos seguintes:

'OLD'– Arquivo deve existir.

'NEW'– Arquivo não deve existir. O arquivo será criado pelo comando OPEN. A partir deste momento, o seu status se torna 'OLD'.

'REPLACE'– Se o arquivo não existe, ele será criado. Se o arquivo já existe, este será eliminado e um novo arquivo é criado com o mesmo nome. Em ambos os casos, o status é então alterado para 'OLD'.

'SCRATCH'– O arquivo é temporário e será deletado quando este for fechado com o comando CLOSE ou na saída da unidade de programa.

'UNKNOWN'– Status do arquivo desconhecido; depende do sistema. Este é o valor padrão do especificador, caso este seja omitido.

O especificador "FILE=" deve estar presente se 'NEW' ou 'REPLACE' são especificados ou se 'OLD' é especificado e a unidade não está conectada.

ACCESS= <acc>, onde <acc> é uma expressão de caracteres que fornece o valor 'SEQUENTIAL' ou 'DIRECT'. Para um arquivo que já exista, este valor deve ser uma opção válida. Se o arquivo ainda não existe, ele será criado com o método de acesso apropriado. Se o especificador é omitido, o valor 'SEQUENTIAL' é assumido. O significado das especificações é:

'DIRECT'– O arquivo consiste em registros acessados por um número de identificação. Neste caso, o tamanho do registro **deve** ser especificado por "RECL=". Registros individuais podem ser especificados e atualizados sem alterar o restante do arquivo.

'SEQUENTIAL'– O arquivo é escrito/lido sequencialmente, linha a linha.

FORM= <fm>, onde <fm> é uma expressão de caracteres que fornece os valores 'FORMATTED' ou 'UNFORMATTED', determinando se o arquivo deve ser conectado para E/S formatada ou não formatada. Para um arquivo que já exista, o valor deve ser uma opção válida. Se o arquivo ainda não existe, ele será criado com um conjunto de forma que incluem o forma especificada. Se o especificador é omitido, os valores-padrão são:

'FORMATTED'– Para acesso sequencial.

'UNFORMATTED'– Para conexão de acesso direto.

`RECL= <rl>,` onde `<rl>` é uma expressão inteira cujo valor deve ser positivo.

- Para arquivo de acesso direto, a opção especifica o tamanho dos registros e é obrigatório.
- Para arquivo sequencial, a opção especifica o tamanho máximo de um registro, e é opcional com um valor padrão que depende do processador.
  - Para arquivos formatados, o tamanho é o número de caracteres para registros que contenham somente caracteres-padrão.
  - Para arquivos não formatados, o tamanho depende do sistema, mas o comando `INQUIRE` (seção 9.11) pode ser usado para encontrar o tamanho de uma lista de E/S.

Em qualquer situação, para um arquivo que já exista, o valor especificado deve ser permitido para o arquivo. Se o arquivo ainda não existe, ele é criado com um conjunto permitido de tamanhos de registros que incluem o valor especificado.

`BLANK= <bl>,` onde `<bl>` é uma expressão de caracteres que fornece os valores `'NULL'` ou `'ZERO'`. A conexão deve ser com E/S formatada. Este especificador determina o padrão para a interpretação de brancos em campos de entrada numéricos.

`'NULL'`– Os brancos são ignorados, exceto que se o campo for completamente em branco, ele é interpretado como zero.

`'ZERO'`– Os brancos são interpretados como zeros.

Se o especificados é omitido, o valor padrão é `'NULL'`.

`POSITION= <pos>,` onde `<pos>` é uma expressão de caracteres que fornece os valores `'ASIS'`, `'REWIND'` ou `'APPEND'`. O método de acesso deve ser sequencial e se o especificador é omitido, o valor padrão é `'ASIS'`. Um arquivo novo é sempre posicionado no seu início. Para um arquivo que já existe e que já está conectado:

`'ASIS'`– O arquivo é aberto sem alterar a posição corrente de leitura/escrita no seu interior.

`'REWIND'`– O arquivo é posicionado no seu ponto inicial.

`'APPEND'`– O arquivo é posicionado logo após o registro de final de arquivo, possibilitando a inclusão de dados novos.

Para um arquivo que já existe mas que não está conectado, o efeito da especificação `'ASIS'` no posicionamento do arquivo é indeterminado.

`ACTION= <act>,` onde `<act>` é uma expressão de caracteres que fornece os valores `'READ'`, `'WRITE'` ou `'READWRITE'`.

`'READ'`– Se esta especificação é escolhida, os comandos `WRITE`, `PRINT` e `ENDFILE` não devem ser usados para esta conexão.

`'WRITE'`– Se esta especificação é escolhida, o comando `READ` não pode ser usado.

`'READWRITE'`– Se esta especificação é escolhida, não há restrição.

Se o especificador é omitido, o valor padrão depende do processador.

`DELIM= <del>,` onde `<del>` é uma expressão de caracteres que fornece os valores `'APOSTROPHE'`, `'QUOTE'` ou `'NONE'`. Se `'APOSTROPHE'` ou `'QUOTE'` são especificados, o caractere correspondente será usado para delimitar constantes de caractere escritos com formatação de lista ou com o uso da declaração `NAMELIST`, e será duplicado onde ele aparecer dentro de tal constante de caractere. Além disso, caracteres fora do padrão serão precedidos por valores da espécie. Nenhum caractere delimitador será usado e nenhuma duplicação será feita se a especificação for `'NONE'`. A especificação `'NONE'` é o valor padrão se este especificador for omitido. O especificador somente pode aparecer em arquivos formatados.

`PAD= <pad>,` onde `<pad>` é uma expressão de caracteres que fornece os valores `'YES'` ou `'NO'`.

`'YES'`– Um registro de entrada formatado será considerado preenchido por brancos sempre que uma lista de entrada e a formatação associada especificarem mais dados que aqueles que são lidos no registro.

'NO'– Neste caso, o tamanho do registro de entrada deve ser não menor que aquele especificado pela lista de entrada e pela formatação associada, exceto na presença de um especificador ADVANCE= 'NO' e uma das especificações "EOR=" ou "IOSTAT=".

O valor padrão se o especificador é omitido é 'YES'. Para caracteres não padronizados, o caractere de branco que preenche o espaço restante depende do processador.

A seguir, temos um exemplo do uso deste comando:

```
OPEN(17, FILE= 'SAIDA.DAT', ERR= 10, STATUS= 'REPLACE', &
     ACCESS= 'SEQUENTIAL', ACTION= 'WRITE')
```

Neste exemplo, um arquivo SAIDA.DAT é aberto para escrita, é conectado ao número de unidade lógica 17, o arquivo é acessado linha a linha e ele já existe mas deve ser substituído. O rótulo 10 deve pertencer a um comando executável válido.

```
OPEN(14, FILE= 'ENTRADA.DAT', ERR= 10, STATUS= 'OLD',  &
     RECL=IEXP, ACCESS= 'DIRECT', ACTION= 'READ')
```

Aqui, o arquivo ENTRADA.DAT é aberto somente para leitura na unidade 14. O arquivo é diretamente acessado e deve ser um arquivo previamente existente.

9.6 Comando READ

Este é o comando que executa a leitura formatada de dados. Até agora, o comando foi utilizado para leitura na entrada padrão, que usualmente é o teclado:

```
READ*, <lista>
```

onde o asterisco "*" indica a entrada padrão e <lista> é a lista de variáveis cujo valor é lido do teclado.

Este comando será agora generalizado para uma forma com ou sem unidade lógica. Sem o número de unidade lógica o comando fica:

```
READ <format> [,<lista>]
```

e com o número de unidade fica:

```
READ([UNIT=]<u>, [FMT=] <format>[, IOSTAT= <ios>] &
     [, ERR= <err-label>][, END= <end-label>]     &
     [, EOR= <eor-label>][, ADVANCE= <mode>]      &
     [, REC= <int-exp>][, SIZE= <size>][, NML= <grp>]) <lista>
```

Os argumentos do comando são os seguintes:

UNIT= <u>, onde <u> é um número de unidade lógica válido ou "*" para a entrada padrão. Caso esta especificação seja o primeiro argumento do comando, a palavra-chave é opcional.

FMT= <format>, onde <format> é uma string de caracteres formatadores, um rótulo válido em Fortran ou "*" para formato livre. Caso esta especificação seja o segundo argumento do comando, a palavra-chave é opcional.

IOSTAT= <ios>, onde <ios> é uma variável inteira padrão que armazena o status do processo de leitura. Os valores possíveis são:

<ios> = 0, quando o comando é executado sem erros.

<ios> > 0, quando ocorre um erro na execução do comando.

<ios> < 0, quando uma condição de final de registro é detectada em entrada sem avanço ou quando uma condição de final de arquivo é detectada.

ERR= <err-label>, é um rótulo válido para onde o controle de fluxo é transferido quando ocorre um erro de leitura.

`END= <end-label>`, é um rótulo válido para onde o controle de fluxo é transferido se uma condição de final de arquivo é encontrada. Esta opção somente existe no comando `READ` com acesso sequencial.

`EOR= <eor-label>`, é um rótulo válido para onde o controle de fluxo é transferido se uma condição de final de registro é encontrada. Esta opção somente existe no comando `READ` com acesso sequencial e formatado e somente se a especificação `ADVANCE= 'NO'` também estiver presente.

`ADVANCE= <mode>`, onde `<mode>` possui dois valores: `'YES'` ou `'NO'`. A opção `'NO'` especifica que cada comando `READ` inicia um novo registro na mesma posição; isto é, implementa entrada de dados sem avanço (*non-advancing I/O*). A opção padrão é `'YES'`, isto é a cada leitura o arquivo é avançado em uma posição. Se a entrada sem avanço é usada, então o arquivo deve estar conectado para acesso sequencial e o formato deve ser explícito.

`REC= <int-exp>`, onde `<int-exp>` é uma expressão inteira escalar cujo valor é o número de índice do registro lido durante acesso direto. Se `REC` estiver presente, os especificadores `END` e `NML` e o formato livre "`*`" não podem ser também usados.

`SIZE= <size>`, onde `<size>` é uma variável escalar inteira padrão, a qual guarda o número de caracteres lidos. Este especificador somente existe no comando `READ` e somente se a especificação `ADVANCE= 'NO'` também estiver presente.

`NML= <grp>`, onde `<grp>` é o nome de um grupo `NAMELIST` previamente definido em uma declaração `NAMELIST`. O nome `<grp>` não é uma constante de caracteres e sim um nome válido em Fortran[3].

Dado o seguinte exemplo:

```
READ(14, FMT= '(3(F10.7,1X))', REC=IEXP) A, B, C
```

este comando especifica que o registro identificado pela variável inteira `IEXP` seja lido na unidade `14`. O registro deve ser composto por 3 números reais designados pelos descritores `F10.7`, separados por um espaço em branco. Estes 3 números serão atribuídos às variáveis `A`, `B` e `C`. Outro exemplo:

```
READ(*, '(A)', ADVANCE= 'NO', EOR= 12, SIZE= NCH) STR
```

especifica que a leitura é sem avanço na unidade padrão; isto é, o cursor na tela do monitor irá permanecer na mesma linha há medida que a string `STR` será lida. O comando lê uma constante de caractere porque o descritor no formato é "`A`". Em uma situação normal, isto é, leitura com avanço, o cursor seria posicionado no início da próxima linha na tela. A variável `NCH` guarda o comprimento da string e o rótulo `12` indica o ponto onde o programa deve se encaminhar se uma condição de final de registro é encontrada.

9.7 Comandos `PRINT` e `WRITE`

Em comparação com o comando `READ`, o comando `WRITE` suporta os mesmos argumentos, com exceção do especificador `SIZE`.

Até agora, foi usada somente forma do comando `PRINT` para a saída padrão de dados, que usualmente é a tela do monitor:

```
PRINT*, <lista>
```

O mesmo comando, generalizado para saída formatada fica,

```
PRINT <format> [, <lista>]
```

Já a sintaxe mais geral do comando é:

```
WRITE([UNIT=]<u>, [FMT=] <format>[, IOSTAT= <ios>] &
      [, ERR= <err-label>][, ADVANCE= <mode>]      &
      [, REC= <int-exp>][, NML= <grp>]) <lista>
```

Os especificadores deste comando executam, na sua maioria, as mesmas funções executadas pelos especificadores do comando `READ`, com exceção dos seguintes, os quais especificam funções ligeiramente distintas:

[3] Ver seção 9.2.

`UNIT= <u>`, onde `<u>` é um número de unidade lógica válido ou "`*`" para a saída padrão. Caso esta especificação seja o primeiro argumento do comando, a palavra-chave é opcional.

`IOSTAT= <ios>`, onde `<ios>` é uma variável inteira padrão que armazena o status do processo de escrita. Os valores possíveis são:

`<ios> = 0`, quando o comando é executado sem erros.

`<ios> > 0`, quando ocorre um erro na execução do comando.

`<ios> < 0`, quando uma condição de final de registro é detectada em escrita sem avanço ou quando uma condição de final de arquivo é detectada.

`ERR= <err-label>`, é um rótulo válido para onde o controle de fluxo é transferido quando ocorre um erro de escrita.

`ADVANCE= <mode>`, onde `<mode>` possui dois valores: `'YES'` ou `'NO'`. A opção `'NO'` especifica que cada comando `WRITE` inicia um novo registro na mesma posição; isto é, implementa saída de dados sem avanço (*non-advancing I/O*). A opção padrão é `'YES'`, isto é a cada escrita, o arquivo é avançado em uma posição. Se a saída sem avanço é usada, então o arquivo deve estar conectado para acesso sequencial e o formato deve ser explícito.

`REC= <int-exp>`, onde `<int-exp>` é uma expressão inteira escalar cujo valor é o número de índice do registro a ser escrito durante acesso direto. Se `REC` estiver presente, os especificadores `END` e `NML` e o formato livre "`*`" não podem ser também usados.

`NML= <grp>`, onde `<grp>` é o nome de um grupo `NAMELIST` previamente definido em uma declaração `NAMELIST`. O nome `<grp>` não é uma constante de caracteres e sim um nome válido em Fortran[4].

Considere o seguinte exemplo,

```
WRITE(17, FMT= '(I4)', IOSTAT= ISTAT, ERR= 10) IVAL
```

aqui, `'(I4)'` descreve format de dado inteiro com 4 dígitos, `ISTAT` é uma variável inteira que indica o status do processo de escrita; em caso de erro, o fluxo é transferido ao rótulo `10`. `IVAL` é a variável cujo valor é escrito na unidade `17`. Outro exemplo seria:

```
WRITE(*,'(A)', ADVANCE= 'NO') 'Entrada: '
```

Como a especificação de escrita sem avanço foi escolhida, o cursor irá permanecer na mesma linha que a string `'Entrada: '`. Em circunstâncias normais, o cursor passaria ao início da linha seguinte na tela. Nota-se também o descritor de formato de constante de caracteres explícito.

Como exemplo do uso dos comandos de E/S apresentados nas seções 9.5 – 9.7, o programa a seguir abre um novo arquivo sequencial chamado `notas.dat`. O programa então lê o nome de um estudante, seguido por 3 notas, as quais são introduzidas pelo teclado (entrada padrão), escrevendo, finalmente, as notas e a média final em `NOTAS.DAT` e no monitor (saída padrão), na mesma linha do nome. O processo de leitura e escrita é repetido até que um estudante com o nome `'Fim'` é introduzido.

```
PROGRAM Notas
    IMPLICIT NONE
    CHARACTER(LEN=20) :: name
    REAL :: mark1, mark2, mark3
     OPEN (UNIT=4,FILE='NOTAS.DAT')
     DO
        READ(*,*) name, mark1, mark2, mark3
        IF(name == 'Fim')EXIT
        WRITE(UNIT=4,FMT=*) name, mark1, mark2, mark3, (mark1 + mark2 + mark3)/3.0
        WRITE(UNIT=*,FMT=*) name, mark1, mark2, mark3, (mark1 + mark2 + mark3)/3.0
     END DO
     CLOSE(UNIT=4)
  END PROGRAM Notas
```

[4] Ver seção 9.2.

9.8 Comando FORMAT e especificador FMT=

O especificador FMT= em um comando READ ou WRITE pode conter ou o símbolo de formato livre "*", ou uma constante de caracteres que indica o formato ou um rótulo que indica a linha onde se encontra um comando FORMAT.

Dado o seguinte exemplo,

```
WRITE(17, FMT= '(2X,2I4,1X,"nome ",A7)')I, J, STR
```

este comando escreve no arquivo SAIDA.DAT, aberto em um exemplo anterior, as três variáveis I, J e STR de acordo com o formato especificado: "2 espaços em branco (2X), 2 objetos inteiros de 4 dígitos em espaços entre eles (2I4), um espaço em branco (1X) a constante de caractere 'nome ' e, finalmente, 7 caracteres de um objeto de caracteres (A7)". As variáveis a ser realmente escritas são tomadas da lista de E/S após a lista de especificações do comando.

Outro exemplo:

```
READ(14,*)X, Y
```

Este comando lê dos valores do arquivo ENTRADA.DAT usando formato livre e atribuindo os valores às variáveis X e Y.

Finalmente, no exemplo:

```
   WRITE(*,FMT= 10)A, B
10 FORMAT('Valores: ',2(F15.6,2X))
```

O comando WRITE acima usa o formato especificado no rótulo 10 para escrever na saída padrão os valores das variáveis A e B. O formato de saída pode ser descrito como: "2 instâncias de: um objeto real com 15 colunas ao todo com uma precisão de 6 casas decimais (F15.6), seguido por 2 espaços em branco (2X)".

O formato de E/S de dados é definido fazendo-se uso dos *descritores de edição*, abordados na próxima seção.

9.9 Descritores de edição

Nos exemplos apresentados nas seções anteriores, alguns descritores de edição já foram mencionados. Estes editores fornecem uma especificação precisa sobre como os valores devem ser convertidos em uma string escrita em um dispositivo externo (monitor, impressora, etc) ou em um arquivo interno (disco rígido, CD-RW, fita, etc), ou convertido de uma string em um dispositivo de entrada de dados (teclado, etc) ou arquivo interno para as diferentes representações de tipos de variáveis suportados pela linguagem.

Com algumas exceções, descritores de edição aparecem em uma lista separados por vírgulas e somente no caso em que a lista de E/S for vazia ou contiver somente matrizes de tamanho zero que os descritores podem estar totalmente ausentes.

Em um processador que suporta tanto caracteres maiúsculos quanto minúsculos, os descritores de edição são interpretados de forma independente com relação ao caso.

Descritores de edição dividem-se em três classes:

Descritores de edição de dados: I, B, O, Z, F, E, EN, ES, D, G, L, A.

Descritores de controle: T, TL, TR, X, S, SP, SS, BN, BZ, P, :, /.

Descritores de edição de strings: H, 'c', "c" (onde c é uma constante de caractere).

Estes descritores serão discutidos em mais detalhes a seguir.

9.9.1 Contadores de repetição

Os descritores de edição de dados podem ser precedidos por um contador de repetição: uma constante inteira positiva (sem sinal), como no exemplo

```
10F12.3
```

a qual indica 10 constantes reais com 12 posições ao todo e com precisão de 3 casas decimais cada. Dos descritores de edição restantes, somente a barra "/" pode ter um contador de repetição associado. Um contador de repetição também pode ser aplicado a um grupo de descritores de edição, delimitado por parênteses:

Tabela 9.1: Descritores de edição que convertem os tipos intrínsecos.

Descritor	Tipo de dados convertidos
A	Caractere
B	Inteiro de/para base binária
D	Real
E	Real com expoente
EN	Real com notação de engenharia
ES	Real com notação científica
F	Real de ponto flutuante (sem expoente)
G	Todos os tipo intrínsecos
I	Inteiro
L	Lógico
O	Inteiro de/para base octal
Z	Inteiro de/para base hexadecimal

```
PRINT '(4(I5,F8.2))', (I(J), A(J), J= 1,4)
```

o que é equivalente a escrever:

```
PRINT '(I5,F8.2,I5,F8.2,I5,F8.2,I5,F8.2)', (I(J), A(J), J= 1,4)
```

Note também o *laço DO implicado (implied-Do list)*

```
(I(J), A(J), J= 1,4)
```

o que equivale a escrever

```
I(1), A(1), I(2), A(2), I(3), A(3), I(3), A(3)
```

Contadores de repetição como estes podem ser encadeados:

```
PRINT '(2(2I5,2F8.2))', (I(J),I(J+1),A(J),A(J+1), J=1,4,2)
```

Pode-se ver que esta notação possibilita compactar a formatação de E/S de dados.

Se uma especificação de formato é usada com uma lista de E/S que contém mais elementos que o número de descritores de formatação, incluindo os contadores de repetição, um novo registro irá começar e a especificação de formato será repetida até que todos os elementos forem cobertos. Isto acontece, por exemplo, no comando abaixo:

```
PRINT '(10I8)', (I(J), J= 1, 100)
```

9.9.2 Descritores de edição de dados

Cada tipo intrínseco de variáveis em Fortran é convertido por um conjunto específico de descritores, com exceção do descritor `G`, que pode converter qualquer tipo intrínseco. A tabela 9.1 apresenta uma descrição das conversões.

Formato dos descritores de edição de dados.

Um descritor de edição de dados pode assumir uma das seguintes formas em uma lista de formatação de E/S:

```
[<r>]<c><w>
[<r>]<c><w>.<m>
[<r>]<c><w>.<d>
[<r>]<c><w>.<d>[E<e>]
```

onde os campos `<r>`, `<w>`, `<d>` e `<e>` devem ser todos constantes inteiras positivas (sem sinal). Parâmetros de espécie de tipo não podem ser especificados. O significado dos campos é o seguinte:

`<r>` é o contador de repetição. O seu valor pode variar entre `1` e `2147483647 (2**31 - 1)`. Se `<r>` for omitido, ele é assumido igual a 1.

<c> é um dos descritores: I, B, O, Z, F, EN, ES, D, G, L, A.

<w> é o número total de dígitos no campo (ou a *largura do campo*). Se <w> for omitido, é assumido um valor padrão que pode variar com o sistema, porém <w> não pode ser nulo.

<m> é o número mínimo de dígitos que devem aparecer no campo (incluindo zeros precedentes). O intervalo permitido para <m> inicia em 0 mas o valor final depende do sistema.

<d> é o número de dígitos à direita do ponto decimal (os *dígitos significativos*). O intervalo de <d> inicia em 0 mas o valor final depende do sistema. O número de dígitos significativos é afetado se um fator de escala é especificado para o descritor de edição.

E identifica um campo de expoente.

<e> é o número de dígitos no expoente. O intervalo de <e> inicia em 1 mas o valor final depende do sistema.

Os descritores de edição de dados têm as seguintes formas específicas:

Tipo de dado	Descritores de dado
Inteiro	I<w>[.<m>], B<w>[.<m>], O<w>[.<m>], Z<w>[.<m>], G<w>.<d>[E<e>]
Real e Complexo	F<w>.<d>, E<w>.<d>[E<e>], EN<w>.<d>[E<e>], ES<w>.<d>[E<e>], D<w>.<d>, G<w>.<d>[E<e>]
Lógico	L<w>, G<w>.<d>[E<e>]
Caractere	A[<w>], G<w>.<d>[E<e>]

O campo <d> deve ser especificado com os descritores F, E, D e G mesmo se <d> for zero. O ponto decimal também é exigido. Para uma constante de caractere, o campo <w> é opcional, independente do tamanho da constante.

Para todos os descritores numéricos, se o campo de escrita for muito estreito para conter o número completo, de acordo com a descrição fornecida, o campo é preenchido por <w> asteriscos.

Na saída os números são geralmente escritos deslocados para a direita no espaço de campo especificado por <w>; isto é feito preenchendo de espaços em branco à esquerda, caso necessário. Valores negativos são sempre escritos com o sinal de menos.

Na entrada os números também deveriam ser lidos a partir da direita. Espaços em branco à esquerda são ignorados. Se o campo é todo composto por espaços em branco, o registro será lido como zero.

Números inteiros (I<w>[.<m>], B<w>[.<m>], O<w>[.<m>], Z<w>[.<m>])

Na sua forma básica, I<w>, o inteiro será lido ou escrito no campo de largura <w> ajustado à direita. Representando um espaço em branco por ␣, o valor -99 escrito sob o controle de I5 irá ser aparecer como ␣␣-99, com o sinal contando como uma posição no campo.

Para impressão, uma forma alternativa deste descritor permite que o número de dígitos a ser impressos sejam especificados exatamente, mesmo que alguns deles tenham que ser zeros. Trata-se da forma I<w>.<m>, onde <m> indica o número mínimo de dígitos a ser impressos. Neste caso, o valor 99 impresso sob o controle de I5.3 aparecerá como ␣␣099. O valor <m> pode ser zero, em cuja situação o campo será preenchido por brancos se o número impresso for 0. Na entrada, I<w>.<m> é interpretado exatamente da mesma forma que I<w>.

Inteiros também podem ser convertidos pelos descritores B<w>[.<m>], O<w>[.<m>] e Z<w>[.<m>]. Estes são similares ao descritor I, porém destinados a representar o número inteiro nos sistemas numéricos binário, octal e hexadecimal, respectivamente.

Números reais (F<w>.<d>, E<w>.<d>[E<e>], EN<w>.<d>[E<e>], ES<w>.<d>[E<e>], D<w>.<d>)

A forma básica, F<w>.<d>, gera um número real de ponto flutuante. O ponto decimal conta como uma posição no campo. Na entrada de dados, se a string possui um ponto decimal, o valor de <d> é ignorado. Por exemplo, a leitura da string ␣9.3729␣ com o descritor F8.3 causa a transferência do valor 9.3729. Todos os dígitos são usados, mas arredondamento pode ocorrer devido à representação finita de um número pelo computador.

Há outras duas formas de entrada de números aceitáveis pelo descritor F<w>.<d>:

1. Número sem ponto decimal. Neste caso, os <d> dígitos mais à direita serão tomados como a parte fracionário do número. Por exemplo, a string ␣-14629será lida pelo descritor F7.2 como -146.29.

2. Número real na forma padrão; a qual envolve uma parte exponencial (seção 3.3), como o número `-14.629E-2` ou sua forma variante, onde o expoente sempre tem sinal e o indicador de expoente "`E`" é omitido: `-14.629-2`. Neste caso, o campo `<d>` será novamente ignorado e o número na forma exponencial será escrita na forma de ponto flutuante. Sob o controle do descritor `F9.1`, a string anterior será convertida ao número `0.14629`.

Estas propriedades aplicam-se à entrada de dados.

Na saída de dados, os valores são arredondados seguindo as regras normais de aritmética. Assim, o valor `10.9336`, sob o controle do descritor `F8.3` irá aparecer escrito como ␣␣`10.934` e sob o controle de `F4.0` irá aparecer como ␣`11.` (note o ponto).

O descritor de edição `E` possui duas formas, `E<w>.<d>` e `E<w>.<d>E<e>` e é uma maneira mais apropriada de transferir números abaixo de `0.01` ou acima de `1000`.

As regras para estes descritores na entrada de dados são idênticas às do descritor `F<w>.<d>`. Na saída com o descritor `E<w>.<d>`, uma string de caractere contendo a parte inteira com valor absoluto menor que um e quatro caracteres que consistem ou no `E` seguido por um sinal e dois dígitos ou de um sinal seguido por três dígitos. Assim, o número 1.234×10^{23}, convertido pelo descritor `E10.4` vai gerar a string ␣`.1234E+24` ou ␣`.1234+024`. A forma contendo a letra `E` não é usada se a magnitude do expoente exceder 99. Por exemplo, `E10.4` irá imprimir o número 1.234×10^{-150} como ␣`.1234-149`. Alguns processadores põe um zero antes do ponto decimal. Note que agora o valor de `<w>` deve ser grande o suficiente para acomodar, além dos dígitos, o sinal (se for negativo), o ponto decimal, a letra `E` (caso possível), o sinal do expoente e o expoente. Assim, caso se tentasse imprimir o número anterior com o descritor `F9.4`, apareceria `*********`.

Na segunda forma do descritor, `E<w>.<d>E<e>`, o campo `<e>` determina o número de dígitos no expoente. Esta forma é obrigatória para números cujos expoentes têm valor absoluto maior que 999. Assim, o número 1.234×10^{1234}, com o descritor `E12.4E4` é transferido como a string ␣`.1234E+1235`.

O descritor de edição `EN` implementa a notação de engenharia. Ele atua de forma semelhante ao descritor `E`, exceto que na saída o expoente decimal é múltiplo de 3, a parte inteira é maior ou igual a 1 e menor que 1000 e o fator de escala não tem efeito. Assim, o número `0.0217` transferido sob `EN9.2` será convertido a `21.70E-03` ou `21.70-003`.

O descritor de edição `ES` implementa a notação científica. Ele atua de forma semelhante ao descritor `E`, exceto que na saída o valor absoluto da parte inteira é maior ou igual a 1 e menor que 10 e o fator de escala não tem efeito. Assim, o número `0.0217` transferido sob `ES9.2` será convertido a `2.17E-02` ou `2.17E-002`.

Números complexos (`F<w>.<d>`, `E<w>.<d>[E<e>]`, `EN<w>.<d>[E<e>]`, `ES<w>.<d>[E<e>]`, `D<w>.<d>`)

Números complexos podem ser editados sob o controle de pares dos mesmos descritores de números reais. Os dois descritores não necessitam ser idênticos. O valor complexo `(0.1,100.)`, convertido sob o controle de `F6.1,8.1` seria convertido a ␣␣␣`0.1`␣␣`.1E+03`. Os dois descritores podem ser separados por uma constante de caracteres e descritores de controle de edição.

Valores lógicos (`L<w>`)

Valores lógicos podem ser editados usando o descritor `L<w>`. este define um campo de tamanho `<w>` o qual, na entrada consiste de brancos opcionais, opcionalmente seguidos por um ponto decimal, seguido por `T` ou `F`. Opcionalmente, este caractere pode ser seguido por caracteres adicionais. Assim, o descritor `L7` permite que as strings `.TRUE.` e `.FALSE.` sejam lidas com o seu significado correto.

Na saída, um dos caracteres `T` ou `F` irá aparecer na posição mais à direita do campo.

Variáveis de caracteres `A[<w>]`

Valores de caracteres podem ser editados usando o descritor `A` com ou sem o campo `<w>`. Sem o campo, a largura do campo de entrada ou saída é determinado pelo tamanho real do ítem na lista de E/S, medido em termos do número de caracteres de qualquer espécie.

Por outro lado, usando-se o campo `<w>` pode-se determinar o tamanho desejado da string lida ou escrita. Por exemplo, dada a palavra `TEMPORARIO`, temos os seguintes resultados no processo de escrita:

Descritor	Registro escrito
`A`	`TEMPORARIO`
`A11`	␣`TEMPORARIO`
`A8`	`TEMPORAR`

Ou seja, quando o campo do descritor é menor que o tamanho real do registro, este é escrito com os caracteres mais à esquerda. Por outro lado, em caso de leitura da mesma string, as seguintes formas serão geradas:

Descritor	Registro lido
A	TEMPORARIO
A11	♭TEMPORARIO
A8	TEMPORAR♭♭

Todos os caracteres transferidos sob o controle de um descritor `A` ou `A<w>` têm a espécie do ítem na lista de E/S. Além disso, este descritor é o único que pode ser usado para transmitir outros tipos de caracteres, distintos dos caracteres padrões.

Descritor geral `G<w>.<d>[E<e>]`

O descritor de edição geral pode ser usado para qualquer tipo intrínseco de dados. Quando usado para os tipos real ou complexo, a sua ação é idêntica à do descritor `E<w>.<d>[E<e>]`, exceto que na saída de dados, quando a magnitude (n) do valor está no intervalo

$$0.1 - 0.5 \times 10^{-\texttt{<d>}-1} \leq n < 10^{\texttt{<d>}} - 0.5$$

ou zero quando `<d>= 0`, são convertidas como com o descritor `F`, seguidos pelo mesmo número de brancos que o descritor `E` teria reservado à parte exponencial. Esta forma é útil para escrever valores cujas magnitudes não são bem conhecidas previamente e quando a conversão do descritor `F` é preferível à do descritor `E`.

Quando o descritor `G` é usado para os tipos inteiro, lógico e de caracteres, ele segue as regras dos respectivos descritores de edição .

Tipos derivados

Valores de tipos derivados são editados pela sequência apropriada de descritores de edição correspondentes aos tipos intrínsecos dos componentes do tipo derivado. Como exemplo,

```
TYPE :: STRING
   INTEGER             :: COMP
   CHARACTER(LEN= 20) :: PALAVRA
END TYPE STRING
TYPE(STRING) :: TEXTO
READ(*, '(I2,A)')TEXTO
```

Edição de tamanho mínimo de campo

Para possibilitar que os registros de saída contenham o mínimo possível de espaço não usado, os descritores `I`, `F`, `B`, `O` e `Z` podem especificar um tamanho de campo igual a zero, como em `I0` ou `F0.3`. Isto não denota um campo de tamanho nulo, mas um campo com o tamanho mínimo necessário para conter o valor de saída em questão. Este recurso existe para que o programador não precise se preocupar se o tamanho de um campo é suficiente para conter o dado, em cuja situação o uso padrão destes descritores iria gerar uma saída composta somente por asteriscos.

Descritor de edição de string de caracteres

Uma constante de caracteres da espécie padrão, sem um parâmetro de espécie especificado, pode ser transferida a um arquivo de saída inserindo-a na especificação de formato, como no exemplo:

```
PRINT"('Este é um exemplo!')"
```

O qual vai gerar o registro de saída:

```
Este é um exemplo
```

cada vez que o comando seja executado. Descritores de edição de string de caracteres não podem ser usados em entrada de dados.

9.9.3 Descritores de controle de edição

Um descritor de controle de edição age de duas formas: determinando como texto é organizado ou afetando as conversões realizadas por descritores de edição de dados subsequentes.

Formato dos descritores de controle de edição

Um descritor de controle de edição pode assumir uma das seguintes formas em uma lista de formatação de E/S:

```
<c>
<c><n>
<n><c>
```

onde:

`<c>` é um dos seguintes códigos de formato: `T`, `TL`, `TR`, `X`, `S`, `SP`, `SS`, `BN`, `BZ`, `P`, dois pontos `(:)` e a barra `(/)`.

`<n>` é um número de posições de caracteres. Este número não pode ser uma variável; deve ser uma constante inteira positiva sem especificação de parâmetro de espécie de tipo. O intervalo de `<n>` inicia em `1` até um valor que depende do sistema. Em processadores Intel de 32 bits, por exemplo, o maior valor é `<n>= 2**15 - 1`.

Em geral, descritores de controle de edição não são repetíveis. A única exceção é a barra `(/)`, a qual pode também ser precedida por um contador de repetição.

Os descritores de controle possuem a seguintes formas específicas quanto às suas funções:

Tipo de controle	Descritores de controle
Posicional	`T<n>`, `TL<n>`, `TR<n>`, `<n>X`
Sinal	`S`, `SP`, `SS`
Interpretação de brancos	`BN`, `BZ`
Fator de escala	`<k>P`
Miscelâneo	`:`, `/`

O descritor `P` é uma exceção à sintaxe geral dos descritores. Ele é precedido por um fator de escala `(<k>)`, em vez de um especificador de posição.

Descritores de controle de edição também podem ser agrupados em parênteses e precedidos por um contador de repetição do grupo.

Edição posicional

Os descritores `T`, `TL`, `TR` e `X` especificam a posição onde o próximo caractere é transferido de ou para um registro.

Na saída, estes descritores não executam a conversão e transferência de caracteres propriamente dita, além de não afetarem o tamanho do registro. Se caracteres são transferidos a posições na posição especificada por um destes descritores, ou após esta, posições saltadas e que não eram previamente preenchidas por brancos passam a ser preenchidas. O resultado é como se o registro por completo estava inicialmente preenchido por brancos.

Edição `T`

O descritor `T` especifica uma posição de caractere em um registro de E/S. Ele toma a seguinte forma:

```
T<n>
```

onde `<n>` é uma constante positiva que indica a posição de caractere do registro, relativa ao limite à esquerda do tabulador.

Na entrada, o descritor posiciona o registro externo a ser lido na posição de caractere especificada por `<n>`. Na saída, o descritor indica que a transferência de dados inicia na `<n>`-ésima posição de caractere do registro externo.

Exemplos. Suponha que um arquivo possua um registro contendo o valor `ABC␣␣␣XYZ`e o seguinte comando de leitura seja executado:

```
CHARACTER(LEN= 3) :: VALOR1, VALOR2
...
READ(11,'(T7,A3,T1,A3)')VALOR1, VALOR2
```

Os valores lidos serão `VALOR1= 'XYZ'` e `VALOR2= 'ABC'`.

Suponha agora que o seguinte comando seja executado:

```
PRINT 25
25 FORMAT(T51,'COLUNA 2',T21,'COLUNA 1')
```

a seguinte linha é impressa na tela do monitor (ou na saída padrão):

```
00000000011111111112222222222333333333344444444445555555556666666666777777777 8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
                    |                               |
                    COLUNA 1                        COLUNA 2
```

Deve-se notar que as constantes de caractere foram impressas iniciando nas colunas 20 e 50 e não nas colunas 21 e 51. Isto ocorreu porque o primeiro caractere do registro impresso foi reservado como um caractere de controle. Este é o comportamento padrão na saída de dados.

Edição `TL`

O descritor `TL` especifica uma posição de caractere *à esquerda* da posição corrente no registro de E/S. O descritor toma a seguinte forma:

```
TL<n>
```

onde `<n>` é uma constante inteira positiva indicando a `<n>`-ésima posição à esquerda do caractere corrente. Se `<n>` é maior ou igual à posição corrente, o próximo caractere acessado é o primeiro caractere do registro.

Exemplo. No exemplo anterior, temos:

```
PRINT 25
25 FORMAT(T51,'COLUNA 2',T21,'COLUNA 1',TL20,'COLUNA 3')
```

o que gera na tela:

```
00000000011111111112222222222333333333344444444445555555556666666666777777777 8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
       |           |                               |
       COLUNA 3    COLUNA 1                        COLUNA 2
```

Edição `TR`

O descritor `TR` especifica uma posição de caractere *à direita* da posição corrente no registro de E/S. O descritor toma a seguinte forma:

```
TR<n>
```

onde `<n>` é uma constante inteira positiva indicando a `<n>`-ésima posição à direita do caractere corrente.

Edição `X`

O descritor `X` especifica uma posição de caractere à direita da posição corrente no registro de E/S. Este descritor é equivalente ao descritor `TR`. O descritor `X` toma a seguinte forma:

```
<n>X
```

onde `<n>` é uma constante inteira positiva indicando a `<n>`-ésima posição à direita do caractere corrente. Na saída, este descritor não gera a gravação de nenhum caractere quando ele se encontra no final de uma especificação de formato.

Edição de sinal

Os descritores `S`, `SP` e `SS` controlam a saída do sinal de mais `(+)` opcional dentro de campos de saída numéricos. Estes descritores não têm efeito durante a execução de comandos de entrada de dados.

Dentro de uma especificação de formato, um descritor de edição de sinal afeta todos os descritores de dados subsequentes `I`, `F`, `E`, `EN`, `ES`, `D` e `G` até que outro descritor de edição de sinal seja incluído na formatação.

Edição `SP`

O descritor `SP` força o processador a *produzir* um sinal de mais em qualquer posição subsequente onde ele seria opcional. O descritor toma a forma simples

```
SP
```

Edição `SS`

O descritor `SS` força o processador a *suprimir* um sinal de mais em qualquer posição subsequente onde ele seria opcional. O descritor toma a forma simples

```
SS
```

Edição `S`

O descritor `S` retorna o status do sinal de mais como opcional para todos os campos numéricos subsequentes. O descritor toma a forma simples

```
S
```

Interpretação de brancos

Os descritores `BN` e `BZ` controlam a interpretação de brancos posteriores embebidos dentro de campos de entrada numéricos. Estes descritores não têm efeito durante a execução de comandos de saída de dados.

Dentro de uma especificação de formato, um descritor de edição de brancos afeta todos os descritores de dados subsequentes `I`, `B`, `O`, `Z`, `F`, `E`, `EN`, `ES`, `D` e `G` até que outro descritor de edição de brancos seja incluído na formatação.

Os descritores de edição de brancos sobrepõe-se ao efeito do especificador `BLANK` durante a execução de um comando de entrada de dados.

Edição `BN`

O descritor `BN` força o processador a *ignorar* todos os brancos posteriores embebidos em campos de entrada numéricos. O descritor toma a forma simples:

```
BN
```

O campo de entrada é tratado como se todos os brancos tivessem sido removidos e o restante do campo é escrito na posição mais à direita. Um campo todo composto de brancos é tratado como zero.

Edição `BZ`

O descritor `BZ` força o processador a *interpretar* todos os brancos posteriores embebidos em campos de entrada numéricos como zeros. O descritor toma a forma simples:

```
BN
```

Edição de fator de escala `P`

O descritor `P` especifica um fator de escala, o qual move a posição do ponto decimal em valores reais e das duas partes de valores complexos. O descritor toma a forma

```
<k>P
```

onde <k> é uma constante inteira com sinal (o sinal é opcional, se positiva) especificando o número de posições, para a esquerda ou para a direita, que o ponto decimal deve se mover (o fator de escala). O intervalo de valores de <k> é de -128 a 127.

No início de um comando de E/S formatado, o valor do fator de escala é zero. Se um descritor de escala é especificado, o fator de escala é fixado, o qual afeta todos os descritores de edição de dados reais subsequentes até que outro descritor de escala ocorra. Para redefinir a escala a zero, deve-se obrigatoriamente especificar 0P.

Na entrada, um fator de escala positivo move o ponto decimal para a esquerda e um fator negativo move o ponto decimal para direita. Quando um campo de entrada, usando os descritores F, E, D, EN, ES ou G, contém um expoente explícito, o fator de escala não tem efeito. Nos outros casos, o valor interno do dado lido na lista de E/S é igual ao campo externo multiplicado por $10^{-\langle k \rangle}$. Por exemplo, um fator de escala 2P multiplica um valor de entrada por 0.01, movendo o ponto decimal duas posições *à esquerda*. Um fator de escala -2P multiplica o valor de entrada por 100, movendo o ponto decimal duas posições *à direita*.

A seguinte tabela apresenta alguns exemplos do uso do especificador <k>P na entrada de dados:

Formato	Campo de entrada	Valor interno
3PE10.5	␣␣37.614␣␣	.037614
3PE10.5	␣␣37.614E2	3761.4
-3PE10.5	␣␣␣37.614␣	37614.0

O fator de escala deve preceder o primeiro descritor de edição real associado com ele, mas não necessariamente deve preceder o descritor propriamente dito. Por exemplo, todos os seguintes formatos têm o mesmo efeito:

```
(3P,I6,F6.3,E8.1)
(I6,3P,F6.3,E8.1)
(I6,3PF6.3,E8.1)
```

Note que se o fator de escala precede imediatamente o descritor real associado, a vírgula é opcional.

Na saída, um fator de escala positivo move o ponto decimal para a direita e um fator de escala negativo move o ponto decimal para a esquerda. Neste caso, o efeito do fator de escala depende em que tipo de edição real está associada com o mesmo, como segue:

- Para edição F, o valor externo iguala o valor interno da lista de E/S multiplicado por $10^{\langle k \rangle}$, alterando a magnitude do dado.
- Para edições E e D, o campo decimal externo da lista de E/S é multiplicado por $10^{\langle k \rangle}$ e <k> é subtraído do expoente, alterando a magnitude do dado. Um fator de escala positivo diminui o expoente; um fator de escala negativo aumenta o expoente. Para fator de escala positivo, <k> deve ser menor que <d>+2 senão ocorrerá um erro no processo de conversão de saída.
- Para edição G, o fator de escala não tem efeito se a magnitude do dado a ser impresso está dentro do intervalo efetivo do descritor. Se a magnitude estiver fora do intervalo efetivo do descritor, edição E é usada e o fator de escala tem o mesmo efeito como neste caso.
- Para edições EN e ES, o fator de escala não tem efeito.

A seguir, alguns exemplos de saída de dados usando o descritor P:

Formato	Campo de entrada	Valor interno
1PE12.3	-270.139	␣␣-2.701E+02
1PE12.2	-270.139	␣␣␣-2.70E+02
-1PE12.2	-270.139	␣␣␣-0.03E+04

O exemplo a seguir também usa edição P:

```
REAL, DIMENSION(6) :: A= 25.0
WRITE(6,10) A
10 FORMAT(' ', F8.2,2PF8.2,F8.2)
```

resultando os seguintes valores gravados na unidade 6:

```
  25.00  2500.00  2500.00
2500.00  2500.00  2500.00
```

Edição com barra (/)

O descritor / termina a transferência de dados para o registro corrente e inicia a transferência de dados para um registro novo. Em um arquivo ou na tela, o descritor tem a ação de iniciar uma nova linha.

A forma que o descritor toma é:

```
[<r>]/
```

onde <r> é um contador de repetição. O intervalo de <r> inicia em 1 até um valor que depende do sistema. Em processadores intel de 32 bits, o valor máximo é <r>= 2**15-1.

Múltiplas barras forçam o sistema a pular registros de entrada ou a gerar registros brancos na saída, como segue:

- Quando n barras consecutivas aparecem entre dois descritores de edição, n - 1 registros são pulados na entrada ou n - 1 registros brancos são gerados na saída. A primeira barra termina o registro corrente. A segunda barra termina o primeiro pulo ou registro em branco e assim por diante.
- Quando n barras consecutivas aparecem no início ou final de uma especificação de formato, n registros são pulados ou n registros em branco são gerados na saída, porque o parênteses inicial e final da especificação de formato são, por si mesmos, iniciadores ou finalizadores de registros, respectivamente

Por exemplo, dado a seguinte formatação de saída:

```
WRITE(6,99)
99 FORMAT('1',T51,'Linha Cabeçalho'//T51,'Sublinha Cabeçalho'//)
```

irá gerar os seguintes registros no arquivo:

```
00000000011111111112222222222333333333344444444445555555555666666666677777777778
12345678901234567890123456789012345678901234567890123456789012345678901234567890
                                                  |
                                                  Linha Cabeçalho
<linha em branco>
                                                  Sublinha Cabeçalho
<linha em branco>
<linha em branco>
```

Edição com dois pontos (:)

O descritor : termina o controle de formato se não houver mais ítens na lista de E/S. Por exemplo, supondo os seguintes comandos:

```
PRINT 1, 3
PRINT 2, 13
1 FORMAT(' I=',I2,' J=',I2)
2 FORMAT(' K=',I2,:,' L=',I2)
```

as seguintes linhas aparecem na saída:

```
I=␣3␣J=
K=13
```

Se houver ítens de E/S restantes, o descritor : não tem efeito.

9.9.4 Descritores de edição de strings

Descritores de edição de strings controlam a saída das constantes de string. Estes descritores são:

- Constantes de caracteres
- Descritor de edição H (eliminado no Fortran 95).

Um descritor de edição de strings não pode ser precedido por um contador de repetição; contudo, eles podem fazer parte de um grupo contido entre parênteses, o qual, por sua vez, é precedido por um contador de repetição.

Edição de constantes de caracteres

O descritor de constante de caracteres provoca a saída de uma constante de string em um registro externo. Ele vem em duas formas:

- '<string>'
- "<string>"

onde <string> é uma constante de caracteres, sem especificação de espécie. O seu comprimento é o número de caracteres entre os delimitadores; dois delimitadores consecutivos são contados como um caractere.

Para incluir um apóstrofe (') em uma constante de caractere que é delimitada por apóstrofes, deve-se colocar duas apóstrofes consecutivas ('') na formatação. Por exemplo,

```
50 FORMAT('Today''s date is:  ',i2,'/',i2,'/',i2)
```

Da mesma forma, para incluir aspas (") em uma constante de caractere que é delimitada por aspas, deve-se colocar duas aspas consecutivas na formatação.

Como exemplo do uso dos comandos de E/S apresentados nas seções 9.5 – 9.9, o programa a seguir abre o arquivo `pessoal.dat`, o qual contém registros dos nomes de pessoas (até 15 caracteres), idade (inteiro de 3 dígitos), altura (em metros e centímetros) e número de telefone (inteiro de 8 dígitos). O programa lê estes dados contidos em `pessoal.dat` e os imprime no monitor no formato:

```
                      Altura
    Nome      Idade   (metros)  Tel.  No.
    ----      -----    ------    --------
P. A. Silva    45       1,80     33233454
J. C. Pedra    47       1,75     34931458
etc.
```

```
PROGRAM Pessoais_dados
    IMPLICIT NONE
    CHARACTER(LEN=15) :: name
    INTEGER :: age, tel_no, stat
    REAL :: height
     OPEN(UNIT=8, FILE='pessoal.dat')
     WRITE(*,100)
100 FORMAT(T24,'Altura')
     WRITE(*,200)
200 FORMAT(T4,'Nome',T17,'Idade',T23,'(metros)',T32,'Tel. No.')
     WRITE(*,300)
300 FORMAT(T4,'----',T17,'-----',T23,' ------ ',T32,'--------')
     DO
      READ(8, FMT='(a15,1x,i3,1x,f4.2,1x,i8)', IOSTAT= stat) name, &
                                                  age, height, tel_no
      IF (stat<0) EXIT                    ! Testa final de arquivo
      WRITE(*,400) name, age, height, tel_no
400  FORMAT(A,T18,I3,T25,F4.2,T32,I8)
     END DO
  END PROGRAM Pessoais_dados
```

sendo que o arquivo `pessoal.dat` possui a seguinte formatação:

```
P. A. Silva     045 1.80 33233454
J. C. Pedra     047 1.75 34931458
```

Cabe ressaltar também que os comandos `FORMAT` utilizados nos rótulos `100`, `200`, `300` e `400` poderiam ser igualmente substituídos pelo especificador `FMT=`. Por exemplo, as linhas

```
      WRITE(*,200)
200 FORMAT(T4,'Nome',T17,'Idade',T23,'(metros)',T32,'Tel. No.')
```

são equivalentes a

```
WRITE(*, FMT='(T4,"Nome",T17,"Idade",T23,"(metros)",T32,"Tel.  No.")')
```

9.10 Comando CLOSE

O propósito do comando CLOSE é desconectar um arquivo de uma unidade. Sua forma é:

```
CLOSE([UNIT=]<u>[, IOSTAT=<ios>][, ERR= <err-label>][, STATUS=<status>])
```

onde <u>, <ios> e <err-label> têm os mesmos significados descritos para o comando OPEN (seção 9.5). Novamente, especificadores em palavras-chaves podem aparecer em qualquer ordem, mas o especificador de unidade deve ser o primeiro ser for usada a forma posicional.

A função do especificador STATUS= é de determinar o que acontecerá com o arquivo uma vez desconectado. O valor de <status>, a qual é uma expressão escalar de caracteres da espécie padrão, pode ser um dos valores 'KEEP' ou 'DELETE', ignorando qualquer branco posterior. Se o valor é 'KEEP', um arquivo que existe continuará existindo após a execução do comando CLOSE, e pode ser posteriormente conectado novamente a uma unidade. Se o valor é 'DELETE', o arquivo não mais existirá após a execução do comando. Se o especificador for omitido, o valor padrão é 'KEEP', exceto se o arquivo tem o status 'SCRATCH', em cujo caso o valor padrão é 'DELETE'.

Em qualquer caso, a unidade fica livre para ser conectada novamente a um arquivo. O comando CLOSE pode aparecer em qualquer ponto no programa e se é executado para uma unidade não existente ou desconectada, nada acontece.

No final da execução de um programa, todas as unidades conectadas são fechadas, como se um comando CLOSE sem o especificador STATUS= fosse aplicado a cada unidade conectada.

Como exemplo:

```
CLOSE(2, IOSTAT=IOS, ERR=99, STATUS='DELETE')
```

9.11 Comando INQUIRE

O status de um arquivo pode ser definido pelo sistema operacional antes da execução do programa ou pelo programa durante a sua execução, seja por um comando OPEN ou seja por alguma ação sobre um arquivo pré-conectado que o faz existir. Em qualquer momento, durante a execução do programa, é possível inquirir a respeito do status e atributos de um arquivo usando o comando INQUIRE.

Usando uma variante deste comando, é possível determinar-se o status de uma unidade; por exemplo, se o número de unidade existe para o sistema em questão, se o número está conectado a um arquivo e, em caso afirmativo, quais os atributos do arquivo. Outra variante do comando permite inquirir-se a respeito do tamanho de uma lista de saída quando usada para escrever um registro não formatado.

Alguns atributos que podem ser determinados pelo uso do comando INQUIRE são dependentes de outros. Por exemplo, se um arquivo não está conectado a uma unidade, não faz sentido inquirir-se a respeito da forma de acesso que está sendo usada para este arquivo. Se esta inquirição é feita, de qualquer forma, o especificador relevante fica indefinido.

As três variantes do comando são conhecidas como INQUIRE por arquivo, INQUIRE por unidade e INQUIRE por lista de saída. Na descrição feita a seguir, as primeiras duas variantes são descritas juntas. Suas formas são:

```
INQUIRE({[UNIT=]<u>|FILE=<arq>}[, IOSTAT=<ios>][, ERR=<err-label>]           &
        [, EXIST=<ex>][, OPENED=<open>][, NUMBER=<num>][, NAMED=<nmd>]       &
        [, NAME=<nam>][, ACCESS=<acc>][, SEQUENTIAL=<seq>][, DIRECT=<dir>]   &
        [, FORM=<frm>][, FORMATTED=<fmt>][, UNFORMATTED=<unf>][, RECL=<rec>] &
        [, NEXTREC=<nr>][, BLANK=<bl>][, POSITION=<pos>][, ACTION=<act>]     &
        [, READ=<rd>][, WRITE=<wr>][, READWRITE=<rw>][, DE-
LIM=<del>][, PAD=<pad>])
```

onde os especificadores entre chaves ({}) são excludentes; o primeiro deve ser usado no caso de INQUIRE por unidade e o segundo no caso de INQUIRE por arquivo. Neste último caso, <arq> é uma expressão escalar de caracteres cujo valor, ignorando quaisquer brancos posteriores, fornece o nome do arquivo envolvido, incluindo o caminho. A interpretação de <arq> depende do sistema. Em um sistema Unix/Linux, o nome depende do caso.

Um especificador não pode aparecer mais de uma vez na lista de especificadores opcionais. Todas as atribuições de valores aos especificadores seguem as regras usuais e todos os valores do tipo de caracteres, exceto no caso de NAME= são maiúsculos. Os especificadores, cujos valores são todos escalares e da espécie padrão são:

`IOSTAT= <ios>,` tem o mesmo significado descrito no comando `OPEN` (seção 9.5). A variável `<ios>` é a única definida se uma condição de erro ocorre durante a execução do `INQUIRE`.

`ERR= <err-label>,` tem o mesmo significado descrito no comando `OPEN` (seção 9.5).

`EXIST= <ex>,` onde `<ex>` é uma variável lógica. O valor `.TRUE.` é atribuído se o arquivo (ou unidade) existir e `.FALSE.` em caso contrário.

`OPENED= <open>,` onde `<open>` é uma variável lógica. O valor `.TRUE.` é atribuído se o arquivo (ou unidade) estiver conectado a uma unidade (ou arquivo) e `.FALSE.` em caso contrário.

`NUMBER= <num>,` onde `<num>` é uma variável inteira à qual é atribuído o número da unidade conectada ao arquivo, ou `-1` se nenhuma unidade estiver conectada ao arquivo.

`NAMED= <nmd>,`

`NAME= <nam>,` onde `<nmd>` é uma variável lógica à qual o valor `.TRUE.` é atribuído se o arquivo tem um nome ou `.FALSE.` em caso contrário. Se o arquivo tem um nome, este será atribuído à variável de caracteres `<nam>`. Este valor não é necessariamente o mesmo que é dado no especificador `FILE=`, se usado, mas pode ser qualificado de alguma forma. Contudo, em qualquer situação é um nome válido para uso em um comando `OPEN` subsequente. Assim, o `INQUIRE` pode ser usado para determinar o valor real de um arquivo antes deste ser conectado. Dependendo do sistema, o nome depende do caso.

`ACCESS= <acc>,` onde `<acc>` é uma variável de caracteres à qual são atribuídos um dos valores `'SEQUENTIAL'` ou `'DIRECT'` dependendo do método de acesso para um arquivo que está conectado e `'UNDEFINED'` se não houver conexão.

`SEQUENTIAL= <seq>,`

`DIRECT= <dir>,` onde `<seq>` e `<dir>` são variáveis de caracteres às quais são atribuídos os valores `'YES'`, `'NO'` ou `'UNKNOWN'`, dependendo se o arquivo *puder* ser aberto para acesso direto ou sequencial, respectivamente, ou se isto não pode ser determinado.

`FORM= <frm>,` onde `<frm>` é uma variável de caracteres à qual são atribuídos um dos valores `'FORMATTED'` ou `'UNFORMATTED'`, dependendo na forma para a qual o arquivo é realmente conectado, ou `'UNDEFINED'` se não houver conexão.

`FORMATTED= <fmt>,`

`UNFORMATTED= <unf>,` onde `<fmt>` e `<unf>` são variáveis de caracteres às quais são atribuídos os valores `'YES'`, `'NO'` ou `'UNKNOWN'`, dependendo se o arquivo *puder* ser aberto para acesso formatado ou não formatado, respectivamente, ou se isto não pode ser determinado.

`RECL= <rec>,` onde `<rec>` é uma variável inteira à qual é atribuído o valor do tamanho do registro de um arquivo conectado para acesso direto, ou o tamanho máximo do registro permitido para um arquivo conectado para acesso sequencial. O comprimento é o número de caracteres para registros formatados contendo somente caracteres do tipo padrão e dependente do sistema em caso contrário. Se não houver conexão, `<rec>` resulta indefinido.

`NEXTREC= <nr>,` onde `<nr>` é uma variável inteira à qual é atribuído o valor do número do último registro lido ou escrito, mais um. Se nenhum registro foi ainda lido ou escrito, `<nr>= 1`. Se o arquivo não estiver conectado para acesso direto ou se a posição é indeterminada devido a um erro anterior, `<nr>` resulta indefinido.

`BLANK= <bl>,` onde `<bl>` é uma variável de caracteres à qual são atribuídos os valores `'NULL'` ou `'ZERO'`, dependendo se os brancos em campos numéricos devem ser por padrão interpretados como campos de nulos ou de zeros, respectivamente, ou `'UNDEFINED'` se ou não houver conexão ou se a conexão não for para E/S formatada.

`POSITION= <pos>,` onde `<pos>` é uma variável de caracteres à qual são atribuídos os valores `'REWIND'`, `'APPEND'` ou `'ASIS'`, conforme especificado no correspondente comando `OPEN`, se o arquivo não foi reposicionado desde que foi aberto. Se não houver conexão ou se o arquivo está conectado para acesso direto, o valor é `'UNDEFINED'`. Se o arquivo foi reposicionado desde que a conexão foi estabelecida, o valor depende do processador (mas não deve ser `'REWIND'` ou `'APPEND'` exceto se estes corresponderem à verdadeira posição).

ACTION= <act>, onde <act> é uma variável de caracteres à qual são atribuídos os valores 'READ', 'WRITE' ou 'READWRITE', conforme a conexão. Se não houver conexão, o valor é 'UNDEFINED'.

READ= <rd>, onde <rd> é uma variável de caracteres à qual são atribuídos os valores 'YES', 'NO' ou 'UNKNOWN', dependendo se leitura é permitida, não permitida ou indeterminada para o arquivo.

WRITE= <wr>, onde <wr> é uma variável de caracteres à qual são atribuídos os valores 'YES', 'NO' ou 'UNKNOWN', dependendo se escrita é permitida, não permitida ou indeterminada para o arquivo.

READWRITE= <rw>, onde <rw> é uma variável de caracteres à qual são atribuídos os valores 'YES', 'NO' ou 'UNKNOWN', dependendo se leitura e escrita são permitidas, não permitidas ou indeterminadas para o arquivo.

DELIM= <del>, onde <del> é uma variável de caracteres à qual são atribuídos os valores 'QUOTE', 'APOSTROPHE' ou 'NONE', conforme especificado pelo correspondente comando OPEN (ou pelo valor padrão). Se não houver conexão, ou se o arquivo não estiver conectado para E/S formatada, o valor é 'UNDEFINED'.

PAD= <pad>, onde <pad> é uma variável de caracteres à qual são atribuídos os valores 'YES', caso assim especificado pelo correspondente comando OPEN (ou pelo valor padrão); em caso contrário, o valor é 'NO'.

Uma variável que é a especificação em um comando INQUIRE, ou está associada com um, não deve aparecer em outra especificador no mesmo comando.

A terceira variante do comando INQUIRE, INQUIRE por lista de E/S, tem a forma:

```
INQUIRE(IOLENGTH=<comp>) <lista-saída>
```

onde <comp> é uma variável inteira escalar padrão que é usada para determinar o comprimento de uma <lista-saída> não formatada em unidades que dependem do processador. Esta variável pode ser usada para estabelecer se, por exemplo, uma lista de saída é muito grande para o tamanho do registro dado pelo especificador RECL= de um comando OPEN, ou ser usado como o valor do comprimento a ser determinado ao especificador RECL=.

Um exemplo de uso do comando INQUIRE é dado abaixo:

```
LOGICAL             :: EX, OP
CHARACTER(LEN= 11) :: NAM, ACC, SEQ, FRM
INTEGER             :: IREC, NR
...
OPEN(2, IOSTAT= IOS, ERR= 99, FILE= 'Cidades', STATUS= 'NEW', &
     ACCESS= 'DIRECT', RECL= 100)
INQUIRE(2, ERR= 99, EXIST= EX, OPENED= OP, NAME= NAM, ACCESS= ACC, &
        SEQUENTIAL= SEQ, FORM= FRM, RECL= IREC, NEXTREC= NR)
```

Após execução bem sucedida dos comandos OPEN e INQUIRE, as variáveis terão os seguintes valores:

```
EX= .TRUE.
OP= .TRUE.
NAM= 'Cidades␣␣␣'
ACC= 'DIRECT␣␣␣␣'
SEQ= 'NO␣␣␣␣␣␣␣'
FRM= 'UNFORMATTED'
IREC= 100
NR= 1
```

9.12 Outros comandos de posicionamento

Outros comandos que exercem funções de controle no arquivo, em adição à entrada e saída de dados, são fornecidos abaixo.

9.12.1 Comando BACKSPACE

Pode acontecer em um programa que uma série de registros sejam escritos e que, por alguma razão, o último registro escrito deve ser substituído por um novo; isto é, o último registro deve ser sobrescrito. De forma similar, durante a leitura dos registros, pode ser necessário reler o último registro, ou verificar por leitura o último registro escrito. Para estes propósitos, Fortran fornece o comando `BACKSPACE`, o qual tem a sintaxe

```
BACKSPACE([UNIT=]<u>[, IOSTAT=<ios>][, ERR=<err-label>])
```

onde `<u>` é uma expressão inteira escalar padrão que designa o número da unidade e os outros especificadores opcionais têm o mesmo significado que no comando `READ` (seção 9.6).

A ação deste comando é posicionar o arquivo antes do registro corrente, se houver. Se o comando for tentado quando o registro estiver no início do arquivo, nada ocorre. Se o arquivo estiver posicionado após um registro de final de arquivo, este resulta posicionado antes deste registro.

Não é possível solicitar `BACKSPACE` em um arquivo que não exista, nem sobre um registro escrito por um comando `NAMELIST`. Uma serie de comandos `BACKSPACE` resultará no retrocesso no número correspondente de registros.

9.12.2 Comando REWIND

De forma semelhante a uma releitura, re-escritura ou verificação por leitura de um registro, uma operação similar pode ser realizada sobre um arquivo completo. Com este propósito, o comando `REWIND`:

```
REWIND([UNIT=]<u>[, IOSTAT=<ios>][ERR=<err-label>])
```

pode ser usado para reposicionar um arquivo, cujo número de unidade é especificado pela expressão escalar inteira `<u>`. Novamente, os demais especificadores opcionais têm o mesmo significado que no comando `READ`.

Se o arquivo já estiver no seu início, nada ocorre. O mesmo ocorre caso o arquivo não exista.

9.12.3 Comando ENDFILE

O final de um arquivo conectado para acesso sequencial é normalmente marcado por um registro especial, denominado *registro de final de arquivo*, o qual é assim identificado pelo computador.

Quando necessário, é possível escrever-se um registro de final de arquivo explicitamente, usando o comando `ENDFILE`:

```
ENDFILE([UNIT=]<u>[, IOSTAT=<ios>][, ERR=<err-label>])
```

onde todos os argumentos têm o mesmo significado que nos comandos anteriores.

O arquivo é então posicionado após o registro de final de arquivo. Este registro, se lido subsequentemente por um programa, deve ser manipulado usado a especificação `END=<end-label>` do comando `READ` senão a execução do programa irá normalmente terminar.

Antes da transferência de dados, um arquivo não pode estar posicionado após o registro de final de arquivo, mas é possível retroceder com os comandos `BACKSPACE` ou `REWIND`, o que permite a ocorrência de outras transferências de dados.

Um registro de final de arquivo é automaticamente escrito sempre quando ou uma operação de retrocesso parcial ou completo segue um operação de escrita como a operação seguinte na unidade; quando o arquivo é fechado por um comando `CLOSE`, por um novo comando `OPEN` na mesma unidade ou por encerramento normal do programa.

Se o arquivo também pode ser conectado para acesso direto, somente os registros além do registro de final de arquivo são considerados como escritos e somente estes podem ser lidos em uma conexão de acesso direto subsequente.

Índice Remissivo