UM RESUMO DA LINGUAGEM DE PROGRAMAÇÃO FORTRAN 77

Oyanarte Portilho
Instituto de Física, UnB – 2007

Introdução

Apesar da linguagem de programação Fortran (acrônimo de *FOR*mula *TRAN*slation) ter surgido na década de 50, ela ainda é a mais utilizada em programação científica (cálculo numérico), principalmente na Física. Estima-se que mais de 90% do software utilizado nesse campo da Ciência está escrito em Fortran. Outras linguagens surgiram ao longo do tempo, prometendo "desbancar" o Fortran, como Algol, Basic, Pascal, C e C++. Com a exceção de C e C++, que se tornaram linguagens extremamente poderosas, as demais foram perdendo o seu prestígio. Mas, mesmo em comparação a C, o Fortran é mais simples de programar, pelo menos na minha opinião (a rigor, a linguagem mais "fácil" é aquela que o programador melhor domina). E, naquilo que o Fortran levava desvantagem em relação a essa linguagem, do ponto de vista de cálculo numérico, ele acabou por incorporar muitos desses recursos, como o uso de ponteiros e alocação dinâmica de memória, nas suas versões mais recentes: Fortran 90 e Fortran 95. Porém nos limitaremos nestas notas ao Fortran 77 que não dispõe desses recursos mas que satisfaz plenamente a maioria das necessidades.

Cada linguagem está associada a um *compilador*. Este é um programa instalado no sistema que tem por finalidade gerar, a partir de um arquivo do tipo texto (*programa-fonte*) criado com um editor apropriado e que contém os comandos daquele programa em particular, um arquivo executável. Se o programa-fonte não estiver escrito de modo a satisfazer rigorosamente a sintaxe da linguagem, o compilador apontará erros na etapa de compilação. Só após esses erros tiverem sido eliminados é que o compilador gerará o arquivo executável correspondente. Por outro lado, o programa-fonte poderá conter erros na sua estruturação lógica, o que certamente produzirá resultados inexatos mas isso não pode ser detectado pelo compilador. Depende da atenção e experiência do programador testá-lo e detectar eventuais erros de lógica, de modo a torná-lo confiável para o fim a que se propõe.

Certamente há inumeráveis formas de se escrever um programa que funcione produzindo resultados corretos. Entretanto, em cálculos mais complexos, o fator *tempo de execução* pode se tornar determinante quanto à possibilidade de se alcançar os resultados com a qualidade e precisão desejadas. Aí, a programação se torna uma arte que busca usar os recursos da linguagem e os algoritmos da forma mais eficiente: é a *otimização* do programa.

Estrutura do programa

Um programa em Fortran visa essencialmente resolver um problema de cálculo numérico, embora seja possível fazer mais algumas coisas como gerar gráficos ou lidar com textos. A *estrutura* é basicamente a seguinte:

```
program principal
   (cabeçalho)
  . . .
   (série de comandos, um em cada linha)
  . . .
end
function funcao(a,b,c)
   (cabeçalho)
  . . .
```

```
   (série de comandos, um em cada linha)
  . . .
end
subroutine tarefa(d,e,f)
   (cabeçalho)
  . . .
   (série de comandos, um em cada linha)
  . . .
end
```

Aqui aparecem alguns termos em negrito, que são palavras reservadas da linguagem e que não devem ser utilizadas arbitrariamente pois têm significado especial. No exemplo, temos um *programa principal*, definido por **`program`**, seguido de um nome à sua escolha (chamei de `principal`), um cabeçalho e uma série de comandos cujo último é **`end`**; uma *função*, definida por **`function`**, seguida de um nome à sua escolha (chamei de `funcao`) e parâmetros (considerei três, que chamei de `a,b,c`), um cabeçalho e uma série de comandos cujo último é também **`end`**; e, finalmente, uma *sub-rotina* **`subroutine`**, chamada `tarefa`, e uma estrutura semelhante a da **`function`**. Só pode haver um programa principal, mas pode haver uma quantidade qualquer de funções e sub-rotinas. As funções e sub-rotinas geralmente contêm uma série de comandos que, de outra forma, apareceriam repetidamente no programa principal ou em outras funções e sub-rotinas. Então a sua finalidade básica é evitar essas repetições. Pode-se, por outro lado, optar por escrever essas funções e sub-rotinas como forma de estruturar mais claramente o programa em diversos blocos, cada qual com uma ou diversas finalidades, tornando-o mais fácil de ser compreendido quando examinado por outra pessoa que não o seu autor. A diferença entre função e sub-rotina ficará mais clara posteriormente.

Estrutura de uma linha de comando

Todas as linhas de comando devem satisfazer a uma certa convenção de estrutura, advinda do tempo em eles eram "escritos" em cartões perfurados. Cada cartão tinha 80 colunas e isso impôs limitação na quantidade de caracteres por linha que se mantém até hoje no Fortran 77, por tradição, embora versões mais modernas tenham superado essas limitações. A convenção é a seguinte:

- Coluna 1: quando contém `c`, `C` ou `*` o compilador Fortran considera o conteúdo da linha como um comentário, os quais são muito úteis para relembrar ao programador a finalidade do(s) próximo(s) comando(s)
- Colunas 1-5: podem ser usadas para dar um número para o comando, o qual serve de referência necessária para outros comandos, como veremos depois
- Coluna 6: se estiver preenchida com algum caractere (exceto no caso de ser linha de comentário), significa que é continuação da linha anterior; pode haver uma quantidade qualquer de linhas de continuação na maioria dos compiladores
- Colunas 7-72: contêm o comando propriamente dito; se não couber no espaço, dar continuidade usando o recurso da coluna 6
- Colunas 73-80: considera-se também como uma forma de comentário (recurso pouco utilizado; antigamente, no tempo dos cartões, servia para numerá-los seqüencialmente de forma a tornar mais simples a tarefa de colocá-los na ordem certa quando caíam e se misturavam)

Nomes e tipos de variáveis

Os *nomes* das variáveis necessariamente começam por uma letra e os demais caracteres podem ser letras, números ou os caracteres especiais `$` e `_`. Não há distinção entre letras maiúsculas e minúsculas. Por exemplo, `NOME`, `nome`, `Nome`, `nOME`, `NoMe`, etc., descrevem a mesma variável, função ou sub-rotina. Nas primeiras versões do F77 havia limite de 6 caracteres nos nomes mas hoje esse valor aumentou, o que permite ao programador escolher aquele que achar mais adequado. Internamente, o nome de uma variável apenas indica ao computador uma posição de memória.

No cabeçalho do programa deve-se definir, opcionalmente, o *tipo* de cada variável, Os tipos não **`REAL`**, **`INTEGER`**, **`COMPLEX`**, **`CHARACTER`** e **`LOGICAL`**. Há uma importante distinção entre as variáveis inteiras (ou de ponto fixo) e reais (ou de ponto flutuante) pois a aritmética envolvida com os números inteiros geralmente conduz a resultados exatos – exceto em divisões cujo resultado não é inteiro, como 1/3 – e ocupam menos memória. As variáveis reais passam por uma deficiência na própria representação – nem todos os números reais são representáveis de forma exata na base binária usada internamente nos computadores e isso pode resultar em erros que, ao se acumularem após uma quantidade muito grande de operações, conduzem a resultados imprecisos. Portanto, deve-se, sendo possível e até certo ponto, priorizar as variáveis inteiras.

Em variáveis do tipo **`COMPLEX`** são guardados números complexos, que têm uma parte real e uma imaginária. Nas variáveis do tipo **`LOGICAL`** podemos guardar apenas dois valores: **`.TRUE.`** e **`.FALSE.`**. Nas variáveis do tipo **`CHARACTER`** podemos guardar conjuntos de caracteres (conhecidos igualmente como *strings* ou variáveis alfanuméricas), como nomes de pessoas ou seus telefones (embora os telefones possam também ser armazenados em variáveis inteiras). Não abordaremos aqui as variáveis do tipo **`COMPLEX`** e **`LOGICAL`** por serem menos usadas.

A faixa de valores das variáveis e a quantidade máxima de dígitos considerada nas operações aritméticas dependem da quantidade de bytes de memória (1 byte = 8 bits) usados para as armazenar. Pode-se ocupar 1, 2, 4, 8 ou até mesmo 16 bytes, dependendo do compilador e do tipo da variável. Alguns exemplos ilustram a sintaxe de declarações de tipo, que vão no cabeçalho:

```
REAL*8 a,b,c2(5,10)
INTEGER x,qq,w3l2norte
CHARACTER*15 nome(1000),status(1000)
```

A primeira linha define como reais de 8 bytes as variáveis `a`, `b`, `c2`; a segunda, como inteiras de 4 bytes (embora não apareça o `*4`, esse é o valor *default* para as inteiras) as variáveis `x`, `qq`, `w3l2norte`; a terceira define como alfanuméricas, de no máximo 15 caracteres, as variáveis `nome` e `status`. Se não aparecesse o `*8` na primeira linha, o compilador tomaria como sendo de 4 bytes (valor *default* para as reais). As faixas de variações das variáveis inteiras e reais e opções de precisão são as seguintes (às vezes depende do compilador):

`INTEGER``*2`	de –32768 a +32767
`INTEGER``*4`	de –2147483648 a +2147483647
`REAL``*4`	de $\pm 2{,}9\times10^{-39}$ a $\pm 1{,}7\times10^{+38}$, com 7 dígitos (precisão simples)
`REAL``*8`	de $\pm 5{,}6\times10^{-309}$ a $\pm 9{,}0\times10^{+307}$, com 15 dígitos (precisão dupla)
`REAL``*16`	de $\pm 8{,}4\times10^{-4933}$ a $\pm 5{,}9\times10^{+4931}$, com 33 dígitos (precisão quádrupla; poucos compiladores têm esta opção)

A declaração de tipo **`DOUBLE PRECISION`** é equivalente a **`REAL`**`*8`.

Quando as declarações de tipo não são feitas, o compilador considera como **`REAL`**`*4` todas as variáveis cujos nomes começam nas faixas A-H ou O-Z e **`INTEGER`**`*4` as que começam

na faixa I-N. É possível alterar essas faixas por meio do comando **IMPLICIT**, como num dos exemplos:

```
IMPLICIT REAL*8 (A-H,O-Z)
IMPLICIT REAL*4 (M-N)
IMPLICIT INTEGER*2 (A-C)
```

Um outro comando, que seguir um **IMPLICIT**, definindo tipo diverso para uma lista de variáveis, abre exceções e tem prioridade sobre este.

Uma outra forma de classificar as variáveis é distingui-las entre *variáveis simples* e *variáveis indexadas*. As últimas, que podem ser de qualquer um dos tipos já definidos, distinguem-se das primeiras por terem índices associados a elas. Geralmente representam vetores (1 índice) ou matrizes bi e tridimensionais (2 e 3 índices). A quantidade de índices é limitada a 7. Os índices devem ser números inteiros ou variáveis inteiras. Se for definido por uma expressão envolvendo variáveis reais, o resultado é convertido para inteiro. O compilador deve receber informação sobre a faixa de variação de cada índice, a fim de que uma área de memória adequada seja reservada. Quando não especificado, o menor índice vale 1. Essa especificação pode ser feita de várias maneiras, sempre no cabeçalho do programa:

a) no momento da especificação do tipo, como apareceu em exemplo anterior

```
REAL*8 a,b,c2(5,10)
CHARACTER*15 nome(1000),status(1000)
```

Aqui a variável de dupla precisão `c2` é uma matriz bidimensional, em que o primeiro índice começa em 1 (não especificado) e termina em 5 e o segundo começa em 1 e vai até 10. Essa variável contém 5×10 = 50 elementos. As variáveis alfanuméricas `nome` e `status` são vetores (matrizes unidimensionais), cada qual com 1000 elementos.

b) por meio da declaração **DIMENSION**. Poderíamos ter, por exemplo, de forma equivalente à acima, a seqüência

```
REAL*8 a,b,c2
CHARACTER*15 nome,status
DIMENSION c2(5,10),nome(1000),status(1000)
```

c) por meio dos comandos **COMMON** e **DATA**, como veremos depois.

Quando conveniente, o menor valor de um índice pode ser especificado, de modo a torná-lo diferente de 1. Os exemplos ilustram a sintaxe:

```
DIMENSION x(-5:10,0:100)
INTEGER quantidade(10:30)
```

Aqui a matriz real `x` é bidimensional, tal que o seu primeiro índice vai de –5 a 10 e o segundo de 0 a 100. Tem, assim, 16×101 = 1616 elementos. O vetor de números inteiros `quantidade` tem 21 elementos.

Constantes

As constantes, assim como as variáveis, podem ser *inteiras* (ponto fixo) ou *reais* (ponto flutuante). As constantes inteiras não têm parte decimal, ou seja, são números inteiros sem ponto decimal:

```
123456
-501
0
5
```

As constantes reais, por outro lado, possuem o ponto decimal:

```
123.456
-0.0799
0.
5.
```

Podem ser escritas, ainda, usando-se a notação científica, separando-se a primeira parte (mantissa) do expoente de 10:

```
1.23456E+2
-7.99E-2
0.E0
0E0
5E0
```

Nesse caso as constantes reais acima são de precisão simples. Para precisão dupla e quádrupla use `D` e `Q`, respectivamente, no lugar de `E`.

Operações e expressões aritméticas

As variáveis e constantes podem sofrer operações dentro de uma expressão aritmética. Os operadores aritméticos são `+`, `-`, `*`, `/`, `**`, que representam respectivamente soma, subtração, produto, divisão e potenciação. Por exemplo

```
a+7.5**3-9./x
```

Existe uma ordem de prioridade que define quais operações o compilador deve efetuar primeiramente, a qual segue as regras comuns de operações aritméticas em Matemática: 1.º potenciações, 2.º divisões e multiplicações e 3.º somas e subtrações. Assim, a expressão acima é equivalente a

$$a + 7{,}5^3 - \frac{9}{x}$$

Numa expressão aritmética também podemos introduzir parênteses, que fazem o papel de parênteses, colchetes, chaves, etc., comuns em expressões aritméticas da Matemática. O compilador passa, nesse caso, a dar prioridade ao cálculo do que há dentro dos parênteses, saindo do mais interno para os mais externos. Por exemplo

```
((((x+0.5)*x-7.5)*x-2.3)*x-1.1)*x+4.33
```

equivale a

$$x^5 + 0{,}5x^4 - 7{,}5x^3 - 2{,}3x^2 - 1{,}1x + 4{,}33$$

Note que a quantidade de parênteses que abrem deve ser a mesma quantidade dos que fecham. Uma outra forma, aparentemente mais simples, de se escrever essa expressão seria

```
x**5+0.5*x**4-7.5*x**3-2.3*x**2-1.1*x+4.33
```

Porém quando se leva em conta a otimização do programa tem-se que considerar o tempo realizado em cada operação. As mais demoradas são: 1.º potenciação, 2.º divisão, 3.º multiplicação, 4.º somas e subtrações. Daí se nota que a primeira forma é preferível. Um pouco mais de esforço por parte do programador pode resultar em programas sensivelmente mais eficientes.

Nesse aspecto, vale chamar a atenção que a potenciação acontece de forma diferente, dependendo se o expoente é um número inteiro ou real. No primeiro caso, a operação é realizada por multiplicação otimizada da base por ela mesma. Isso significa que `2**7` é calculado como $\left(2^3\right)^2 \times 2$, usando aritmética de inteiros, que conduz ao resultado exato 128. Por outro lado, para `2**(7.)` o cálculo é feito usando logaritmo natural e a função exponencial $e^{7\ln 2}$, o qual, além de ser muito mais demorado já que essas funções são calculadas por séries, apresenta imprecisão na última casa decimal. Portanto, sempre que o expoente for inteiro, use esse tipo de variável ou constante para o expoente ao definir as potenciações.

Como já foi mencionado anteriormente, as operações aritméticas envolvendo constantes e variáveis inteiras em geral conduzem a resultados exatos, com exceção de alguns casos de divisão. Por exemplo, `1/3` dá como resultado 0 enquanto que `1./3.` resulta em 0,3333333 (se considerarmos possíveis erros de truncamento ou de representação inadequada de uma parte dos números reais na base binária, o último dígito pode apresentar imprecisões, dando, por exemplo 0,3333332 ou 0,3333334). Apesar do resultado da divisão `1/3` não ter dado correto, pois sofreu truncamento para 0, isso pode ser explorado favoravelmente em certas situações. Por outro lado, as expressões `1/3.` e `1./3` são equivalentes e conduzem ao resultado idêntico ao de `1./3.`, isso porque estamos mesclando os dois tipos de constantes. Nesse caso, a constante inteira é convertida automaticamente para real antes que a divisão seja realizada.

Exercício: Que resultados espera para as seguintes expressões aritméticas

a) `4**(1/2)` b) `4**0.5` c) `2**(-1)` d) `2.**(-1)` e) `(1./9.)-(1D0/9D0)`
f) `5./9.-0.55555555` g) `(7.**0.5)**2-7.`

Comandos de atribuição

O resultado de uma expressão aritmética é guardado numa posição de memória, definida por uma variável, por meio de um comando de atribuição como no exemplo

```
discriminante=b**2-4.*a*c
```

Note que no lado esquerdo do sinal = vai o nome da variável que está recebendo o valor do cálculo da expressão que está no lado direito. Ainda mais, no lado esquerdo não pode aparecer nenhuma operação aritmética, exceto em índices, como em

```
y(i+1,j-1,k-n)=z+2.5
```

O sinal = tem aqui um significado diferente do matemático pois um comando como

```
i=i+1
```

é perfeitamente admissível e significa "acrescente 1 ao valor presente na variável `i` e atribua a ela mesma o novo valor".

A atribuição de valor para uma variável alfanumérica segue a sintaxe

```
CHARACTER*10 a
. . .
a='abacate'
```

ou seja, o *string* devem vir entre apóstrofos (`'`). Pode-se também usar aspas (`"`). A única operação possível para *strings* é a concatenação, que é representada por duas barras `//`:

```
CHARACTER*7 a,c
CHARACTER*8 d
CHARACTER*1 b
. . .
a='abacate'
b='s'
c='a'//'b'
d=a//b
```

e teremos em `c` o *string* `ab` e em `d` o *string* `abacates`. Por outro lado, é possível acessar uma parte do *string*. Por exemplo, em `a(5:7)` temos `ate`, em `a(:3)` temos `aba` (a omissão do primeiro parâmetro pressupõe que vale 1) e em `a(6:)` temos `te` (a omissão do último parâmetro pressupõe que vale o comprimento do *string*).

Para as variáveis lógicas, pode-se atribuir somente dois valores:

```
LOGICAL a,b
. . .
a=.true.
b=.false.
```

O corpo de um programa é normalmente composto, em sua maior parte, por comandos de atribuição, além de comandos que permitem a entrada (por meio de leitura de dados digitados *on line*, na etapa de execução, ou em arquivos previamente gerados) e saída de dados (na tela ou em arquivos), produção de laços, e desvios condicionais ou incondicionais.

Uma forma de atribuir *valores iniciais* para certas variáveis é por meio do comando **`DATA`**, o qual só pode ser posicionado no cabeçalho do programa principal, após a declaração de tipo já ter sido feita. Por exemplo,

```
DATA a,b,c /1.,2.,3./
```

Aqui as variáveis reais `a`, `b`, `c` tomam os valores iniciais 1, 2, e 3. Note que são apenas valores iniciais e que no decorrer da execução esses valores podem ser alterados. Uma maneira equivalente de se dar o comando é

```
DATA a/1./,b/2./,c/3./
```

Pode-se usar o **DATA** para dar valores também para variáveis indexadas e mesmo definir as suas dimensões:

```
DATA a(5)/2.3,4.2,7.9,4.5,0./
```

em que está-se definindo `a(1)` como 2,3 , `a(2)` como 4,2 , etc. Quando se trata de uma variável indexada de mais de uma dimensão, a listagem de valores deve acompanhar primeiro os índices da direita. Por exemplo, numa matriz bidimensional, a seqüência (usando o **DIMENSION** para dimensionar)

```
DIMENSION a(2,3)
DATA a/1.,2.,3.,4.,5.,6./
```

guarda inicialmente a matriz a com os valores 1, 2 e 3 na primeira linha, e 4, 5 e 6 na segunda. Note que poderíamos igualmente definir

```
a(1,1)=1.
a(1,2)=2.
a(1,3)=3.
a(2,1)=4.
a(2,2)=5.
a(2,3)=6.
```

Mas em certas situações o comando **DATA** é muito útil, como no exemplo

```
DATA a(1000)/1000*1./
```

que atribui o valor 1 a todos os 1000 elementos da matriz a.

Uma outra forma de atribuir valores, agora não só iniciais mas também imutáveis, também no cabeçalho, é por meio do comando **PARAMETER** cuja sintaxe é

```
PARAMETER(nome1=..., nome2=..., ...)
```

onde está se atribuindo à variável `nome1` o valor representado pelas reticências, o mesmo para `nome2`, etc. É importante que **PARAMETER** apareça depois das declarações de tipo. Caso contrário o compilador toma os tipos implícitos. Por exemplo,

```
REAL*8 pi,q_eletron
PARAMETER(pi=3.14159265358979d0,q_eletron=1.6021773349d0)
```

A vantagem de se fazer atribuições via **PARAMETER** é que, se acidentalmente o programa tentar alterar esses valores em algum ponto, isso vai provocar um erro. Nesse exemplo, estamos atribuindo os valores de pi e da carga do elétron, que são constantes as quais certamente deverão permanecer inalteradas durante a execução do programa. Porém se o programador não for desatento esse cuidado é desnecessário.

Comandos de desvio

Os comandos de desvio são de dois tipos: *incondicionais* e *condicionais*.

Desvio incondicional

O comando de desvio incondicional é o **goto** , que pode ser escrito também como **go to** pois os espaços em branco são ignorados. Como o nome diz, ele transfere a execução do programa para o comando cujo número aparece à frente do comando. É claro que o comando referenciado deve existir; caso, contrário, haverá erro na fase de compilação. Lembre que se deve usar as colunas 1-5 para numerar comandos. Como exemplo veja o trecho

```
    . . .
    a=a+2.
    go to 10
30  a=b+c
    go to 20
10  x=y+c-a
    c=0.5*(a+c)
    go to 30
20  a=a+1.
    . . .
```

A transferência incondicional deve, dentro do possível, ser evitada. Para alguns, é indício de que o programa está mal-escrito. Mas para um principiante não se trata um erro tão grave, embora indique que, com algum esforço, esses desvios talvez possam ser reformulados e eliminados.

Desvio condicional, expressões lógicas e operadores

O desvio condicional é produzido pelo comando **if**. A estrutura, na forma mais simples, de um comando **if** é a seguinte

if(*expressão lógica***) then**
(seqüência de comandos)
endif

onde *expressão lógica* é uma expressão que pode ser falsa ou verdadeira. Se for verdadeira, a seqüência de comandos entre o **if** e o **endif** são executados. Caso contrário, esses comandos são ignorados e o próximo comando a ser executado é o que vier logo após o **endif**. As expressões lógicas envolvem *operadores lógicos* e *operadores relacionais*. Entre os operadores lógicos mais usados temos **.and.** ("e" – conjunção lógica) e **.or.** ("ou" – disjunção lógica). Entre os relacionais temos **.lt.** (menor que), **.le.** (menor ou igual a), **.eq.** (igual a), **.ne.** (diferente de), **.gt.** (maior que), **.ge.** (maior ou igual a). Por exemplo,

```
a=b**2-y
x=z/(j+1.)
u6=z+2.
rw3=(e-3.7)*(f+f-2.)
if((a.gt.x).and.(u6.lt.rw3*1e-10/a)) then
  a=5.*a
  x=0.5*x
endif
```

Note que se costuma usar tabulação deslocada dentro do bloco **if - endif** para facilitar a sua visualização. Pode-se ter também definir alternativa usando-se o **else** dentro do bloco **if - endif**:

if(*expressão lógica***) then**
(seqüência de comandos)

```
else
    (seqüência de comandos)
endif
```

que significa que a primeira seqüência será executada se a expressão lógica for verdadeira; caso contrário, a segunda seqüência será executada. Em seguida, os comandos que seguirem o **endif** serão executados. O **else** também pode ser condicionado por meio um ou vários **elseif**:

```
if(expressão lógica) then
    (seqüência de comandos)
elseif(expressão lógica) then
    (seqüência de comandos)
elseif(expressão lógica) then
    (seqüência de comandos)
endif
```

Por exemplo,

```
if(j.gt.k) then
  j=j-1
elseif(j.eq.k) then
  j=j+2
elseif(j.lt.-5*k) then
  j=j+1
else
  j=0
  go to 10
endif
```

Note que o último comando `j=0` só será executado se as alternativas anteriores forem falsas. Porém a presença do **else** não é obrigatória dentro do bloco **if - endif**. Ele foi incluído aqui apenas para enriquecer o exemplo. Uma seqüência de blocos **if - endif** aninhados também é admissível:

```
if(expressão lógica) then
    (seqüência de comandos)
    if(expressão lógica) then
        (seqüência de comandos)
        if(expressão lógica) then
            (seqüência de comandos)
        endif
        (seqüência de comandos)
    endif
    (seqüência de comandos)
endif
```

O que falamos até agora do comando **if** foi a sua forma atual, também chamada de *if lógico*. Há uma outra forma, já em desuso, conhecida como *if aritmético*, que apresentaremos apenas por razões históricas. Por ser menos eficiente na sua execução, não se recomenda que seja empregado. A sintaxe é

```
     if(expressão aritmética) n1,n2,n3
n1   (comando)
     go to n4
n2   (comando)
     go to n5
```

```
n3   (comando)
```

Aqui n1, n2, n3 são três rótulos de comandos, os quais aparecem no **if** e necessariamente um deles na linha que segue o **if**. Os outros dois rótulos podem aparecer em qualquer parte do programa. Pode-se ter dois dos rótulos iguais, mas não os três. O significado dessa sintaxe é a seguinte: se *expressão aritmética* for negativa, a execução é desviada para o comando de rótulo n1; se for nula é desviada para o comando de rótulo n2; se for positiva é desviada para o comando de rótulo n3. Por exemplo,

```
10  j=j+1
    if(j-k) 10,20,30
20  j=j-1
    go to 40
30  j=0
40  j1=j
```

Comandos de repetição

Os comandos de repetição controlam um conjunto de linhas de comando (*laços* ou *loops*) em que geralmente uma variável inteira sofre variação dentro de uma faixa de valores ou enquanto uma determinada condição for satisfeita. Um desses comandos é o **do** que tem a seguinte estrutura

```
do n=n1,n2,n3
 (seqüência de comandos)
enddo
```

sendo que n é a variável inteira de controle, n1 e n2 são de preferência constantes , variáveis inteiras ou expressões que as envolvam (se forem expressões envolvendo constantes ou variáveis reais o compilador truncará o resultado para um valor inteiro) e que definem respectivamente o início e o fim da seqüência de valores que n sofrerá variação, e n3 define o passo de cada variação. Note que n3 pode ser negativo. Se n3 for omitido o compilador supõe que vale 1. Por exemplo, a seqüência

```
isoma=0
do i=1,1000
  isoma=isoma+i
enddo
```

permite calcular a soma dos números inteiros entre 1 e 1000. Note que a primeira linha garante que o valor inicial de isoma é 0. Esse procedimento é chamado de *inicialização de variável*. Alguns compiladores fazem isso automaticamente, mas nem todos. Assim, nunca deixe de inicializar as variáveis em seu programa pois, caso contrário, poderá obter resultados bizarros.

Observe que um bloco **do - enddo** pode ser substituído, de forma menos eficiente, por comandos **if** e **go to**, como exemplificado abaixo para o mesmo cálculo da soma e 1 a 1000:

```
   i=0
   isoma=0
1  i=i+1
   if(i.le.1000) then
     isoma=isoma+i
     go to 1
   endif
```

É possível "aninhar" conjuntos de laços. Por exemplo, a seqüência

```
REAL*8 a(10,10),b(10,10),x(10,10),soma
. . .
do i=1,10
  do j=1,10
    soma=0d0
    do k=1,10
      soma=soma+a(i,k)*b(k,j)
    enddo
    x(i,j)=soma
  endo
enddo
```

calcula o produto das matrizes `a` e `b`, supostamente 10×10, em precisão dupla, e guarda o resultado na matriz `x` por meio da expressão

$$x_{ij} = \sum_{k=1}^{10} a_{ik}\, b_{kj}$$

Deve-se observar que não é permitido que um comando **`go to`** ou um **`if`** transfira a execução de fora para dentro de um bloco **`do - enddo`**, porém o contrário pode ocorrer.

O comando **`do`** aceita uma outra sintaxe, já em desuso, na qual emprega um rótulo para designar a última linha do seu bloco de ação, ao invés da presença do **`enddo`**:

```
   do m n=n1,n2,n3
     (seqüência de comandos)
m  (comando)
```

sendo que `m` é o rótulo mencionado. O comando na última linha pode ser um comando de atribuição ou simplesmente um indicativo de fim de bloco, que nesse caso é o **`continue`**:

```
   isoma=0
   do 7 i=1,1000
     isoma=isoma+i
7  continue
```

O segundo comando de repetição é o **`do while`** cuja sintaxe é

```
do while(expressão lógica)
      (seqüência de comandos)
enddo
```

A seqüência de comandos é executada enquanto *`expressão lógica`* for verdadeira. Por exemplo,

```
i=1
isoma=0
do while(i.le.1000)
  isoma=isoma+i
  i=i+1
enddo
```

produz o mesmo resultado para a soma dos números inteiros de 1 a 1000.

Funções e sub-rotinas

As *funções* e *sub-rotinas* são usadas para se evitar repetir várias vezes uma mesma seqüência de comandos. Por exemplo, podemos transformar a seqüência mostrada nas págs. 11-12 numa função ou numa sub-rotina que faz multiplicação de matrizes quando isso for feito mais de uma vez. A diferença entre função e sub-rotina é que, a grosso modo, as funções, as quais podem ter vários parâmetros de entrada, só admitem uma só variável de saída, enquanto que as sub-rotinas admitem uma quantidade indeterminada de variáveis de saída. Entretanto, essa regra pode ser quebrada facilmente, de modo que o uso de uma ou outra forma fica mais ao gosto do programador. Mas as maneiras com que uma função e uma sub-rotina são acionadas são diferentes. Note que uma função ou uma sub-rotina pode chamar outras funções / sub-rotinas. A única limitação do Fortran 77 é que não pode haver chamadas recursivas (a função / sub-rotina chamar a si mesma). Ilustremos inicialmente o caso do programa-principal chamar a multiplicação de matrizes quadradas, escrita na forma de uma sub-rotina (a saída é uma matriz e, portanto, é mais conveniente definir uma sub-rotina neste caso pois o resultado não é um único valor) que denominamos `multmat`:

```
      program produto
c     este programa calcula o produto de matrizes
c     programador:            data:
      real*8 m1(100,100),m2(100,100),m3(100,100)
      . . .
      (definição das matrizes m1 e m2, cada qual 10×10)
      . . .
      call multmat(10,m1,m2,m3)
      . . .
      (impressão dos resultados e/ou continuidade com outras operações)
      . . .
      stop
      end

      subroutine multmat(n,a,b,x)
      real*8 a(n,n),b(n,n),x(n,n),soma
      do i=1,n
        do j=1,n
          soma=0d0
          do k=1,n
            soma=soma+a(i,k)*b(k,j)
          enddo
          x(i,j)=soma
        endo
      enddo
      return
      end
```

Temos as seguintes observações:

1. Após a palavra **`subroutine`** vem o nome da mesma (`multmat`) e uma lista (opcional, como veremos depois) de variáveis que são os parâmetros de entrada (`n,a,b`) e de saída (`x`). Esses parâmetros podem ser variáveis simples ou indexadas mas estas últimas não mostram os seus índices. Isso é estabelecido na definição de tipo logo abaixo (no caso, o **`real`**`*8` está definindo as matrizes como de dimensão `n` e com precisão dupla). O nome `multmat` não pode ter o tipo definido pois é apenas um nome de chamada para a sub-rotina e não uma variável em si. Os programadores costumam convencionar listar primeiro os parâmetros de

entrada e em seguida os de saída, mas isso é arbitrário. O aviso para o computador retornar a execução para o programa que chamou a sub-rotina é dado pelo comando **`return`**, que pode aparecer em qualquer lugar do corpo da mesma, não necessariamente logo antes do **`end`**; é possível haver mais de um comando **`return`**. O comando **`return`** logo antes do **`end`** pode ser dispensado. Note que não definimos o tipo das variáveis `i`, `j`, `k`, as quais estão sendo consideradas como **`INTEGER`**`*4` por *default*.

2. Não é apropriado trabalhar com as variáveis de entrada alterando os seus valores. Isso significa que essas variáveis só devem aparecer no lado direito dos comandos de atribuição. Além disso, não é apropriado trabalhar com as variáveis de saída recursivamente. Isso significa que elas só devem aparecer no lado esquerdo dos comandos de atribuição.
3. Neste exemplo chamamos a sub-rotina a partir do programa principal. Essa chamada acontece pelo uso do comando **`call`** que é seguido do nome da sub-rotina e eventual lista de parâmetros. É essencial que a quantidade e tipos de variáveis nessa lista seja exatamente igual ao definido na sub-rotina. Mas os nomes podem ser diversos, de tal forma que a associação é feita pela ordem em que aparecem nas duas listas.
4. As matrizes `m1`, `m2`, `m3` foram dimensionadas como sendo 100×100 no programa principal mas na verdade elas são efetivamente 10×10 pois a `multmat` está sendo chamada com `n` igual a 10. Isso foi feito apenas para mostrar que o programa funciona corretamente assim mesmo, embora essa prática de "desperdício" de memória não seja recomendável. Porém, ao se escrever um programa mais genérico, a ser usado por outras pessoas, é melhor dimensionar as matrizes para um valor mais alto quando não se sabe o quanto o usuário vai precisar.
5. A execução do programa principal está sendo interrompida com o comando **`stop`**, o qual pode aparecer em qualquer parte do mesmo, não necessariamente antes do **`end`**; pode haver mais de um comando **`stop`**. Um comando **`stop`** imediatamente antes de um **`end`** é dispensável.

Vejamos agora o exemplo de uma função. Programaremos uma função que calcula o fatorial de um número inteiro, a qual chamaremos de `fatorial`. Os valores são guardados numa tabela, chamada `tab`:

```
      program comb
c     calculo dos fatoriais de 0 a 100
c     programador:            data:
      real*8 fatorial,tab(0:100)
      do i=0,100
        tab(i)=fatorial(i)
      enddo
      (outros comandos)
      end

      function fatorial(n)
      real*8 fatorial,aux
      if(n.eq.0.or.n.eq.1) then
        fatorial=1
        return
      endif
      aux=1d0
      i=2
      dowhile(i.le.n)
        aux=aux*i
        i=i+1
      enddo
      fatorial=aux
```

```
      return
      end
```

Notemos que

1. Como no caso da sub-rotinas, após a palavra **function** vem o seu nome (`fatorial`) e uma lista (opcional, como veremos depois) de variáveis que são os parâmetros de entrada (`n`, neste caso). A variável de saída é o próprio nome da função, razão pela qual o seu tipo deve ser definido, ainda que seja por *default*. Uma outra opção válida seria definir o seu tipo logo na primeira linha:

   ```
   real*8 function fatorial(n)
   ```

   A saída da função também acontece usando-se o **return**.
2. Como no caso das sub-rotinas, não é apropriado alterar os dados de entrada dentro da função.
3. A forma de se chamar uma função é a envolvendo numa expressão aritmética. Neste exemplo, temos no programa principal a chamada

   ```
   tab(i)=fatorial(i)
   ```

Mencionamos que há uma outra forma de transferir valores entre o programa principal, as funções e as sub-rotinas que não por meio da lista que aparece logo após o nome de uma função ou sub-rotina, fazendo até que essa lista seja dispensável. Esse recurso é o comando **COMMON**. Ele deve ser colocado no cabeçalho das funções e sub-rotinas e informa que a lista de variáveis a seguir têm as mesmas posições de memória (e, portanto, os mesmos valores) em todas essas funções e sub-rotinas. E se aparecer também no programa-principal, esses valores continuam disponíveis mesmo após a execução de uma determinada função ou sub-rotina ter sido concluída. Por exemplo, o programa de produto de matrizes pode ser escrito também da seguinte forma:

```
      program produto
c     este programa calcula o produto de matrizes
c     programador:            data:
      real*8 m1,m2,m3
      common m1(100,100),m2(100,100),m3(100,100),k
      . . .
(definição das matrizes m1 e m2, cada qual 100×100)
      . . .
      k=10
      call multmat
      . . .
(impressão dos resultados e/ou continuidade com outras operações)
      . . .
      stop
      end

      subroutine multmat
      real*8 a,b,x,soma
      common a(100,100),b(100,100),x(100,100),n
      do i=1,n
        do j=1,n
          soma=0d0
          do k=1,n
            soma=soma+a(i,k)*b(k,j)
          enddo
```

```
        x(i,j)=soma
      endo
    enddo
    return
    end
```

Observe que:

1. Os nomes das variáveis que aparecem nas listas dos dois comandos **COMMON** não são os mesmos. Porém isso não é problema pois a associação é feita pela ordem com que aparecem nas listas. Essa ordem não é completamente arbitrária: devem vir primeiro as variáveis de precisão quádrupla, quando houverem, seguidas pelas de precisão dupla, simples e variáveis inteiras.
2. O dimensionamento das matrizes está sendo feito usando-se o próprio **COMMON**.
3. A dimensão das matrizes `a`, `b` e `x` agora não é mais genérico como anteriormente pois os dois comandos **COMMON** devem ter listas que ocupam a mesma quantidade total de memória.

Pode acontecer que haja a necessidade de se ter vários comandos **COMMON** diferentes pois as listas de variáveis comuns entre, por exemplo, o programa-principal e uma sub-rotina e entre o programa-principal e uma outra sub-rotina sejam diferentes. Para distinguir os diversos comandos **COMMON** usam-se rótulos, que são nomes que começam por uma letra e que aparecem entre barras. Por exemplo,

```
program principal
real*8 m1,m2,m3,q1,q2,q2,x1,x2,x3,mt,cm,cc,ct
common /massas/m1,m2,m3,mt,cm
common /posicoes/x1,x2,x3
common /cargas/q1,q2,q3,ct,cc
 (definições das massas, cargas e posições)
call massa
call carga
 (outros comandos)
end

subroutine massa
real*8 massa_total,m1,m2,m3,p1,p2,p3,centro_de_massa
common /massas/m1,m2,m3,massa_total,centro_de_massa
common /posicoes/p1,p2,p3
massa_total=m1+m2+m3
centro_de_massa=(m1*p1+m2*p2+m3*p3)/massa_total
return
end

subroutine carga
real*8 carga_total,c1,c2,c3,x1,x2,x3,centro_de_carga
common /posicoes/x1,x2,x3
common /cargas/c1,c2,c3,carga_total,centro_de_carga
carga_total=c1+c2+c3
centro_de_carga=(c1*x1+c2*x2+c3*x3)/carga_total
return
end
```

Aqui temos três comandos **common** rotulados: o `/posicoes/` repassa informações do programa-principal para as duas sub-rotinas; o `/massas/` troca informações (entrada e saída) com a sub-rotina `massa` e o `/cargas/` troca informações com a sub-rotina `carga`. Note que os comandos **common**, escritos nessa forma, acabaram por tornar desnecessários parâmetros explícitos nas sub-rotinas.

Comandos de entrada / saída e formato

Opcionalmente, em vez de definir os valores iniciais de determinadas variáveis por meio do comando **DATA** ou via comandos de atribuição, deseja-se usar dados constantes de um arquivo do tipo texto (caracteres ASCII) previamente construído. Por outro lado, precisamos também de uma forma de tornar acessível as resultados gerados pelo programa, quer por torná-los visíveis na tela do terminal, quer por armazená-los num arquivo, o qual poderá, em seguida, ser examinado por meio de um recurso qualquer como um editor de textos. Esses objetivos são alcançados por meio dos comandos de entrada e saída.

Comecemos pelo *comando de entrada*, que é o comando **read**. Há duas formas de sintaxe. A mais simples se refere somente a leituras de dados que serão digitados, aparecendo na tela enquanto isso acontece. Inicialmente tomemos o *formato livre*, que significa que, não importando em quais colunas esses dados serão digitados, bastando que pelo menos haja um espaço em branco ou uma vírgula entre eles, o F77 vai "lê-los" corretamente. É simplesmente da forma

```
read *, lista de variáveis
```

onde nessa lista as variáveis estão separadas por vírgulas. Se houver um comando como esse, o F77 suspende a execução do programa, aguardando que o programador ou usuário digite os dados da lista no formato livre, representado pelo * nessa sintaxe. Porém na lista de variáveis pode haver variação de um ou mais índices entre os limites inferior e superior, para leitura de variáveis indexadas (**do** implícito). Por exemplo,

```
read *,a,n,(xmat(i),i=1,n)
```

Poderíamos então digitar, após a execução ter se interrompido nesse ponto:

```
7.9,4,-10,200,23,55
```

que significa que estaríamos atribuindo os valores 7,9 para a, 4 para n e -10, 200, 23, 55 respectivamente para xmat(1), xmat(2), xmat(3), xmat(4).

Consideremos agora o **read** com formato, que tem a seguinte sintaxe:

```
read(n1,n2) lista de variáveis
```

onde *n1* é um número inteiro, até 99, que define a unidade de entrada de dados e *n2* é o número do comando onde o formato de leitura (via comando **format**) está definido. Se no lugar de *n2* houver um asterisco, isso significa formato livre. Quando *n1* é um asterisco ou o número 5 isso significa que a entrada é a tela. Então **read**(*,*) equivale a **read** *,. Se a entrada for via arquivo de dados, o valor de *n1* é associado ao nome do arquivo por meio do comando **open** que deve aparecer de preferência como o primeiro comando após o cabeçalho. Por exemplo,

```
      real a,b
      integer jj
      . . .
      open(unit=10,file='dados.dat',status='old')
      . . .
      read(10,100) a,b,jj
100   format(f5.1,1x,e15.8,1x,i5)
```

Analisemos primeiramente o comando **open**. Ele pode ter vários parâmetros mas os principais são os três aqui usados: `unit`, que dá o número de referência da unidade de entrada, ou seja, o valor de *n1* no comando **read** associado; `file`, que define o nome do arquivo (entre apóstrofos) onde será feita a leitura dos dados e `status`, que tem as opções `'old'` (o arquivo referenciado já existe), `'new'` (o arquivo referenciado não existe e deve ser criado – opção só usada para a escrita – v. abaixo), `'unknown'` (verifica se o arquivo existe: em caso afirmativo, o usa; em caso contrário, o cria); `'scratch'` (cria o arquivo e o remove no término do programa). Então, no exemplo acima, temos que as variáveis `a`, `b`, `jj` serão lidas no arquivo pré-existente `dados.dat` obedecendo o formato prescrito na linha de comando `100`. Analisemos agora a sintaxe do comando **format**. Nesse exemplo, o `f5.1` especifica como será a leitura da variável `a`, sendo que o `f` significa que o valor está escrito na notação não científica, o `5` significa que usam-se as 5 primeiras colunas e o `1` significa que, se não aparecer o ponto decimal explicitamente, o valor será considerado como tendo uma casa decimal, considerando-se da direita para a esquerda, à partir da quinta coluna. O `1x` indica para desconsiderar a coluna seguinte (a sexta). O `e15.8` especifica como será a leitura da variável `b`, sendo que `e` significa que o valor está escrito na notação científica, o `15` significa que as serão usadas as próximas 15 colunas (da 7.ª à 21.ª) para esse valor e o `8` significa que na parte da mantissa há 8 dígitos após o ponto decimal, se este não aparecer explicitamente. Novamente temos o `1x` indicando para desconsiderar a coluna seguinte (a 22.ª). Em seguida, temos o `i` que indica que o próximo dado é um número inteiro (sem ponto decimal) que ocupa as próximas 5 colunas (da 23.ª até a 27.ª coluna). Assim, o arquivo `dados.dat` poderia conter uma linhas com os dados dispostos da seguinte forma:

```
  221      2.57903e-2 89847
12345678901234567890123456789O...
```

que significa que `a` terá o valor 22,1 pois foi especificado que o valor tem uma casa decimal; `b` terá o valor $2{,}57903\times10^{-2}$; `jj` terá o valor 89847. Note que os dados são ajustados da direita para a esquerda no campo reservado. Se tivéssemos ajustado pela esquerda

```
221   2.57903e-2      89847
12345678901234567890123456789O...
```

o valor de `a` seria 2210,0 pois os espaços em branco dentro do campo são interpretados como zeros. Os outros valores, de `b` e `jj`, se manteriam. Ao se explicitar o ponto decimal no dado, a regra definida no formato é superada. Por exemplo,

```
22.1     2.57903e-2   89847
12345678901234567890123456789O...
```

fornece também os dados 22,1, $2{,}57903\times10^{-2}$ e 89847. Por isso, não se deve preocupar muito com a definição da posição do ponto decimal no formato, no caso de variáveis reais. Basta estar atento para a quantidade total de colunas no campo.

Enquanto as letras `f` e `e` são usadas para definir campo no caso de variáveis reais e a letra `i` é usada para variáveis inteiras, o parâmetro que define campo, no caso de variáveis do tipo *string*, é a letra `a`. Por exemplo,

```
      character*5 nome1,nome2,nome3
      . . .
      read(*,300) nome1,nome2,nome3
300   format(3a5)
```

com dados (`3a5` é equivalente a `a5,a5,a5`)

```
 pato sapopavao
123456789012345...
```

guarda os *strings* `pato`, `sapo`, `pavao` nas variáveis `nome1`, `nome2`, `nome3`.

O comando **`read`** também admite transferência no fluxo de execução para determinados comandos quando houver fim de arquivo ou erro (de formato) no processo de leitura:

```
read(n1,n2,err=r1,end=r2) lista de variáveis
```

onde *`r1`* é o rótulo do comando para onde a execução deverá ser transferida se surgir erro de leitura da *`lista de variáveis`* e *`r2`* para onde será transferida no caso de se encontrar fim de arquivo. Por exemplo,

```
      integer telefones(10000)
      character*30 nomes(10000)
      open(unit=8,file='dados.dat',status='old')
      . . .
1     format(a30,1x,i8)
      do i=1,10000
        read(8,1,end=5) nomes(i),telefones(i)
      enddo
5     imax=i-1
      write(*,*) 'quantidade de dados no arquivo:',imax
      . . .
```

que imprime na tela a quantidade de dados no arquivo `dados.dat`.

Um outro comando que pode ser útil é o **`close`**. Suponhamos que um determinado arquivo de dados foi aberto para leitura por meio do **`open`** e a sua execução levará horas ou dias e desejamos disparar um segundo ou um terceiro programa que usará o mesmo arquivo de dados. Enquanto esse arquivo estiver aberto pelo primeiro programa, o seu acesso por outros programas não será possível. Entretanto, o programador poderá usar o **`close`** para fechá-lo, após a leitura de todos os dados ter se completado no primeiro programa (portanto, o **`close`** deve suceder o comando **`read`**). O acesso por outro programa passa então a ser liberado. A sintaxe é

```
close(unit=n1)
```

onde *`n1`* é o parâmetro da unidade, conforme aparece no **`open`** e no **`read`**.

Consideremos agora o *comando de saída* ou *de impressão*, o **`write`**, e seu equivalente, o **`print`**. Para saída de dados na tela em formato livre usa-se

```
print *, lista de variáveis
```

ou, equivalentemente,

```
write(*,*) lista de variáveis
```

Para saída em arquivo com um formato específico usa-se

```
write(n1,n2) lista de variáveis
```

onde *n1* e *n2* têm o mesmo significado que no comando **read**. Por exemplo, o comando **write** abaixo

```
      open(unit=8,file='saida.out',status='new')
      . . .
      a=1947.2879
      b=92.1256
      n=531
      . . .
      write(8,500) a,b,n
500   format('Resultados',//,'a=',e17.10,' b=',f6.2,' com',i4,
     1' iteracoes')
```

imprime no arquivo `saida.out` o seguinte

```
Resultados

a= 0.1947287900e+04 b= 92.13 com 531 iteracoes
12345678901234567890123456789012345678901234567890
```

Cada barra / no **format** causa uma mudança forçada de linha.

Funções de biblioteca

A linguagem Fortran possui uma variedade de funções úteis, que constam de sua biblioteca, e que se encontram disponíveis ao usuário. Geralmente há versões com nomes diferentes, no caso de funções reais, para precisão simples, dupla e quádrupla. Mas existe sempre um nome genérico que pode ser empregado, de tal forma que o compilador substitui pela versão apropriada após identificar a precisão mais elevada das variáveis e constantes que aparecem no argumento. Em seguida temos a relação das principais funções:

função	nome genérico	nomes específicos	observações
raiz quadrada	`sqrt`	`sqrt, dsqrt, qsqrt`	
logaritmo natural	`log`	`alog, dlog, qlog`	
logaritmo decimal	`log10`	`alog10, dlog10, qlog10`	
exponencial	`exp`	`exp, dexp, qexp`	
seno	`sin`	`sin, dsin, qsin`	o argumento deve ser em radianos
co-seno	`cos`	`cos, dcos, qcos`	idem
tangente	`tan`	`tan, dtan, qtan`	idem
arco seno	`asin`	`asin, dasin, qasin`	o resultado é dado em radianos
arco co-seno	`acos`	`acos, dacos, qacos`	idem
arco tangente	`atan`	`atan, datan, qatan`	idem
seno hiperbólico	`sinh`	`sinh, dsinh, qsinh`	
co-seno hiperbólico	`cosh`	`cosh, dcosh, qcosh`	
tangente hiperbólica	`tanh`	`tanh, dtanh, qtanh`	
valor absoluto	`abs`	`abs, dabs, qabs`	versão para argumento real
	`iabs`		versão para argumento

			inteiro
parte inteira (truncamento)	`int`		dá a parte inteira de um argumento real
inteiro mais próximo (arredondamento)	`nint`		dá o inteiro mais próximo de um argumento real
conversão para real*4	`real`		converte o argumento (inteiro ou real) para real com precisão simples
conversão para real*8	`dble`		converte o argumento (inteiro ou real) para real com precisão dupla
conversão para real*16	`qext`		converte o argumento (inteiro ou real) para real com precisão quádrupla
conversão para inteiro	`ifix`		converte o argumento real*4 para inteiro
maior valor de uma lista	`max`	`amax1`, `dmax1`, `qmax1`	deve haver pelo menos dois argumentos reais, separados por vírgula; o resultado é da mesma precisão dos argumentos
	`max0`		os argumentos são inteiros, assim como o resultado
	`max1`		os argumentos são reais e o resultado é inteiro
	`amax0`		os argumentos são inteiros e o resultado é real
menor valor de uma lista			trocar `max` por `min` na listagem acima, obtendo funções análogas
resto da divisão	`mod`	`mod`, `amod`, `dmod`	Dá o resto da divisão do primeiro argumento pelo segundo. A função `mod` é inteira e tem argumentos inteiros, enquanto `amod` e `dmod` são reais e têm argumentos reais
transferência de sinal	`sign`	`sign`, `dsign`, `qsign`	dá ao primeiro argumento o sinal do segundo
	`isign`		versão inteira, com argumentos inteiros
comprimento de *string*	`len`		dá a quantidade de caracteres do argumento alfanumérico
caractere ASCII	`char`		fornece o caractere a que se refere o argumento inteiro, segundo a tabela de caracteres ASCII

valor numérico	`ichar`		fornece o valor numérico referente ao argumento alfanumérico, segundo a tabela de caracteres ASCII (função inversa a `char`)

Compilação e execução

Após editar o programa-fonte, salve-o com sufixo `.f` ou `.for` . A próxima etapa é compilá-lo, para gerar o arquivo executável. Uma forma simples de evocar o compilador **`f77`** e proceder à compilação no sistema operacional Unix/Linux é a seguinte

```
f77 –o programa.exe programa.for
```

onde estamos supondo que o programa-fonte chama-se `programa.for`. Aqui, a opção `–o` permite dar um nome ao arquivo executável, que estamos chamando de `programa.exe`. Geralmente usa-se `.exe` ou `.x` como sufixo para os arquivos executáveis. A geração do executável só será bem-sucedida se não houver erros de sintaxe no programa-fonte. Fique atento a possíveis mensagens de erro nessa etapa e faça as devidas correções no programa-fonte quando necessário. A execução do arquivo `programa.exe` pode se dar de duas maneiras: no modo interativo (*foreground*) e no modo não-interativo (*background*). No primeiro caso basta digitar

```
programa.exe
```

e aguardar a sua finalização. Eventuais mensagens de erro de execução, geralmente ligado a falhas na estrutura lógica do programa, são apresentadas nessa etapa. Enquanto o programa está sendo executado o terminal fica "travado", não mostrando o *prompt* do sistema. Se se desejar suspender a execução do programa deve-se usar a seqüência control–c. A forma interativa é interessante para programas de curto tempo de execução. Para programas que levam um tempo longo, às vezes dias, é melhor usar o modo não-interativo, que usa a sintaxe

```
programa.exe &
```

onde o `&` caracteriza essa situação. A vantagem do modo não-interativo é que o usuário pode fechar a sessão na máquina e voltar depois para verificar o andamento da execução do programa ou examinar os resultados. Para saber se o programa ainda está sendo executado usa-se a combinação dos comandos **`ps`** e **`grep`** com uma das sintaxes (dependendo do sistema Unix)

```
ps axu | grep programa.exe
```

ou

```
ps -ef | grep programa.exe
```

Na XXX coluna encontra-se o tempo de execução até o momento e na primeira coluna aparece o pid (program identity). Esse número é essencial para a suspensão da execução do programa se o usuário suspeitar que há algo errado pois isso é alcançado via comando **`kill`** que tem a sintaxe

```
kill –9 pid
```

Às vezes o programa já executou e não há resultados ou eles são claramente inesperados ou bizarros. Possivelmente uma mensagem de erro teria sido mostrada na tela se o usuário ainda estivesse com a sessão aberta mas,como não estava, ela se perdeu. Para evitar essa possibilidade, pode disparar o programa no modo não interativo da seguinte forma:

```
programa.exe >> programa.erros &
```

que faz com que se houver alguma mensagem de erro de execução, ela seja guardada no arquivo `programa.erros` o qual poderá ser examinado posteriormente.

Vejamos um programa bem simples que mostra uma mensagem na tela:

```
program mensagem
write(*,*) 'olah, estou aqui.'
stop
end
```

Depois de compilar e executar aparecerá a mensagem

```
olah, estou aqui.
```

Um outro programa bem simples é aquele que calcula as raízes de uma equação do segundo grau, sejam reais ou imaginárias, em precisão dupla, interagindo com o usuário pela tela:

```
      program raizes
      implicit real*8(a-h,o-z)
      character*1 resp
100   format(//'Solucao da equacao do 2.o grau:',1pd12.5,' x**2+',
     11pd13.5,' x+',1pd13.5,' = 0')
200   format(/,5x,'existem duas raizes reais:',2(1pd18.10))
300   format(/,5x,'existe uma raiz real:',1pd18.10)
400   format(/,5x,'existem duas raizes imaginarias:',1pd17.10,
     1'+i(',1pd17.10,')',/,32x,1pd17.10,'+i(',1pd17.10,')')
500   format(a1)
1     write(*,*)'entre com o coeficiente do termo quadrático:'
      read(*,*) a
      write(*,*)'entre com o coeficiente do termo linear:'
      read(*,*) b
      write(*,*)'entre com o termo independente:'
      read(*,*) c
      write(*,100) a,b,c
      discr=b*b-4d0*a*c
      if(discr.gt.0d0) then
        raiz1=(-b+sqrt(discr))/(a+a)
        raiz2=(-b-sqrt(discr))/(a+a)
        write(*,200) raiz1,raiz2
      elseif(discr.eq.0d0) then
        raiz1=-b/(a+a)
        write(*,300) raiz1
      else
        preal=-b/(a+a)
        pimag=sqrt(-discr)/(a+a)
        write(*,400) preal,pimag,preal,-pimag
      endif
      write(*,*)
      write(*,*) 'deseja resolver outra equacao? (s ou n)'
      read(*,500) resp
```

```
      if(resp.eq.'s') go to 1
      stop
      end
```

Exercícios

1. Escreva um programa que produza uma tabela que relacione temperaturas Celsius e Fahrenheit. Na primeira coluna deve vir a temperatura em graus Celsius, começando em –200˚C e terminando em +200˚C, com passo de 1˚C, e na segunda coluna a correspondente temperatura em ˚F.
2. Escreva um programa que determine que dia do ano corresponde uma determinada data entre 1900 e 2100. Lembre-se que nos anos bissextos o mês de fevereiro tem 29 dias. Um ano é bissexto se for divisível por 400; caso contrário, se for divisível por 4 mas não por 100.
3. Um dos métodos para avaliar uma integral definida é por meio da quadratura do retângulo, pela qual a integral num sub-intervalo de largura h é aproximada pela área do retângulo com essa largura e comprimento igual ao valor da função avaliada no centro do intervalo, de tal forma que

$$\int_a^b f(x)\,dx \approx h\sum_{i=1}^{N} f(\bar{x}_i)$$

onde N é a quantidade de sub-intervalos em que $[a, b]$ é dividido, a largura do sub-intervalo vale

$$h = \frac{b-a}{N}$$

os pontos que definem os sub-intervalos estão localizados em

$$x_i = x_1 + (i-1)h \qquad\qquad i = 1,\ldots,N$$

e o ponto médio do sub-intervalo está em

$$\bar{x}_i = x_i + \frac{h}{2}$$

Escreva um programa para calcular a integral

$$I = \int_0^{\pi} \operatorname{sen}\theta\, d\theta$$

cujo resultado exato é 2. Experimente variar N para verificar o comportamento do resultado da integral, comparado com o valor exato.

4. Escreva um programa que construa todos os quadrados mágicos 3×3 possíveis, compostos pelos números 1, 2, ... , 9 tal que eles não aparecem repetidos e que a soma dos elementos de

qualquer linha ou de qualquer coluna ou daqueles na diagonal principal ou na secundária seja sempre igual a 15.

5. Suponha que dispõe de uma lista de números inteiros em ordem crescente, armazenados num vetor, e deseja verificar se um dado número está presente na lista. Escreva um programa que faça busca binária, isto é, que inicie comparando o número dado com o elemento central da lista. Se o número dado for maior, compare com o elemento central da metade superior da lista, etc. A resposta deve ser o índice que dá a posição do número na lista ou a informação de que não está presente na mesma.
6. Escreva um programa que ordene uma lista de números diferentes quaisquer em ordem crescente.
7. O número de cadastro de pessoas físicas do Ministério da Fazenda (CPF) tem 9 dígitos seguidos de dois dígitos verificadores, os quais servem como teste para erros de digitação na seqüência. Dada a seqüência dos 9 dígitos (n1, ..., n9) o primeiro dígito verificador (dv1) é gerado seguindo-se a regra: a) calcula-se a soma s1 = 10*n1+9*n2+... +3*n8+2*n9; b) calcula-se o resto da divisão de s1 por 11, r1 = s1 mod 11; c) subtrai-se r1 de 11: dv1 = 11–r1; d) se dv1 resultar 10 ou 11, transforme para 0. O segundo dígito verificador (dv2) é gerado usando-se o dv1: calcula-se a soma s2 = 11*n1+10*n2+...+4*n8+3*n9+2*dv1 e seguem-se os demais passos de forma semelhante. Escreva um programa para verificar se um CPF, dado numa seqüência de 11 dígitos, sem pontos ou hífen, é valido, ou seja, não contém erros de digitação.
8. Escreva um programa que conte a freqüência com que cada letra, maiúscula ou minúscula, ou dígito aparece num determinado texto. Desconsidere os demais caracteres. Suponha, para simplificar, que o texto não contém letras acentuadas e nem cedilhas. O texto a ser analisado deve estar guardado num arquivo ASCII, cujas linhas têm uma quantidade padrão de caracteres, exceto a última. A lista dos caracteres ASCII pode ser acessada, por exemplo, no anexo C da página http://www.geocities.com/helder_pc/fortran/ .
9. Escreva um programa que jogue o "jogo da velha" com o usuário, o qual deve ter a seguinte estrutura: inicializar a matriz 3×3 com zeros; pedir para o jogador escolher o seu símbolo: "X" ou "O"); pedir a jogada do usuário; gerar a jogada do computador (que simplesmente deve preencher com o seu símbolo o primeiro espaço vazio que encontrar – ele não é muito inteligente!); mostrar a matriz alterada na tela; verificar se há vencedor (linhas, colunas ou diagonais com um mesmo símbolo) e anunciá-lo; caso contrário, pedir nova jogada ao usuário, etc. Depois que um jogador vencer, o programa deve perguntar se o usuário quer jogar novamente. Se a resposta for negativa, terminar o programa.