Instituto de Física Armando Dias Tavares

Departamento de Física Aplicada e Termodinâmica

Fortran 95

(– VERSÃO PROVISÓRIA –)
março 2009

Prof. **Anibal Leonardo Pereira**

Rio de Janeiro
primeiro semestre - 2009

Sumário

Apresentação....3
Características do Fortran 95....4
1 – Processador....5
2 – Compatibilidade....5
3 – Características Eliminadas e Obsoletas....6
4 – Módulos....7
5 – Conjunto Básico de Caracteres do Fortran....7
6 – Totem....8
7 – Formato dos Programas Fontes....9
8 – Formato Fixo....10
9 – Formato Livre....11
10 – Comentários....12
11 – Linhas de Continuação....13
12 – Tipos de Declarações....14
13 – Estrutura de um Programa em Fortran 95....15
14 – Ordem das Declarações....16
15 – Declarações Obsoletas....17
16 – Tipos de Dados....19
17 – Dados do Tipo Numérico....19
18 – Dados do Tipo Não-numérico....23
19 – Substring....24
20 – Nomes....25
21 – Declaração de Variável....26
22 – Variáveis e Constantes....27
23 – Constante com Nome....29
24 – Inicializando uma Variável....30
25 – Declaração de Atribuição....32
26 – Operadores Aritméticos....33
27 – Operadores Relacionais....34
28 – Operadores Lógicos....38
29 – Operador Caractere....40
30 – Expressões Aritméticas – Modo Simples....41
31 – Expressões Aritméticas – Modo Misto....43
32 – Funções Intrínsecas....44
33 – Declaração Read....46
34 – Declaração Write....50
35 – Declaração Format....51
36 – Descritores de Formato....53
37 – Declaração if-then-else....59
38 – Declaração if-then-elseif....60
39 – If Encadeados....61
40 – Declaração GOTO....62
41 – Declaração Select case....63
42 – Declaração DO....65
43 – Do Encadeados....66
44 – Do Implícito....67
45 – Vetores e Matrizes....68
46 – Entrada e Saída de dados usando Matrizes....70
47 – Matriz Inteira....73
48 – Reshape....76
49 – Matriz local....77
50 – Forma Assumida de Matriz....78
51 – Funções Intrínsecas de Matriz....80
52 – Unidades de Programas....81
53 – Programa Principal....84
54 – Declaração Stop....85
55 – Declaração Return....86
56 – Subprogramas Externos....87
57 – Subprograma Internos....89

58 – Módulos 89
59 – Associação de Argumentos....91
60 – Atributo Intent....94
61 – Sub-rotina....96
62 – Chamando uma Sub-rotina....97
63 – Argumentos Opcionais....98
64 – Função Intrínseca Present....101
65 – Atributo Save....102
66 – Subprogramas como Argumento 103
67 – Funções....105
68 – Usando Funções....107
69 – Escopo das Variáveis....108
70 – Estrutura Básica de um Módulo....111
71 – Atributos Public e Private....113
72 – Compilando Módulos....114
73 – Bloco Interface....115
74 – Arquivos....116
75 – Fim de Arquivo....119
76 – Funções Intrínsecas....120

Apresentação

Esta nota de aulas está baseada no texto escrito no segundo semestre de 2007 e é uma versão provisória do texto que apresenta a linguagem de programação Fortran 95.

Entenda provisória como sendo a indicação de uma versão de trabalho (que está sendo escrito) o que indica que o texto sofrerá correções e modificações antes de ser considerado acabado.

A motivação inicial para a escrita deste texto foi a de conter o material apresentado em sala de aula sobre o Fortran 95. A intenção foi de disponibilizar um material que permitisse ao aluno anotar durante as aulas somente o que julgasse mais relevante (a seu critério) e mesmo assim ter acesso a todas as regras e informações apresentadas e discutidas em sala de aula e ainda a outras informações que tenham ou não sido citadas em sala de aula.

Inicialmente foi assim, mas logo tornou-se evidente que "algo" mais era necessário. A experiência mostrou claramente que para se possuir um melhor entendimento e conhecimento do Fortran 95 é necessário a leitura do padrão Fortran 95, porque o padrão especifica todas as características que devem ser conhecidas, entendidas e seguidas para se escrever um programa Fortran 95. Portanto a leitura do padrão é altamente recomendada, mas apesar disto o padrão raramente é lido porque (1) está escrito em inglês e (2) não é uma leitura muito agradável.

Quanto ao fato do padrão estar escrito em inglês, a solução é aprender inglês. Para um físico (engenheiro ou quem trabalha na área técnico-científica) aprender inglês é uma necessidade. A aprendizagem do inglês deve começar o mais cedo possível. Claramente esta solução (ler e entender inglês) é a melhor a longo prazo, mas não resolve o problema imediato. Por este motivo iniciei a escrita de um texto com a intenção de tratar da necessidade imediata, não pela simples tradução do padrão, mas sim pela apresentação comentada do mesmo material existente no padrão.

Entretanto como o texto iniciado ainda não está em condições de uso pelo alunos, fiz uma pequena revisão do texto anterior (este que está sendo apresentado na sequência) e o marquei como texto provisório.

Enquanto não termino a novo texto, espero que este possa ser útil no entendimento da linguagem Fortran.

Boa leitura e BOA SORTE!

Características do Fortran 95

Fortran (FORmula TRANslation) é uma linguagem de programação de alto nível projetada para ser utilizada em engenharia, matemática e aplicações científicas computacionalmente intensivas.

O FORTRAN (escrito em letras maiúsculas) foi a primeira linguagem de programação de alto nível criada em 1954, que evoluiu assim:

- FORTRAN
  ou FORTRAN I, foi projetado, desenvolvido e escrito entre 1954-57 e teve como líder do grupo John W. Backus

- FORTRAN II
  (1958) adicionou melhoramentos que possibilitaram a compilação do programa em módulos

- FORTRAN III
  (1958) não foi lançado para uso

- FORTRAN IV
  (1961) foi uma melhoria do FORTRAN II. Do FORTRAN II até o FORTRAN IV existia compatibilidade, os programa escritos em versões anteriores podiam ser executados pela versão mais nova

- FORTRAN 66
  (1966) não apresentava compatibilidade completa com as versões anteriores e foi considerada o primeiro padrão de linguagem da alto nível

- FORTRAN 77
  (1977) apresentava compatibilidade completa com a versão anterior. Acrescentou inúmeras melhorias, ainda hoje é possível encontrar muitos programas escritos em FORTRAN 77

- Fortran 90
  (1990) o grande salto na linguagem Fortran. A partir dela o FORTRAN passou a ser escrito Fortran. Esta versão transformou o Fortran numa linguagem moderna com todas as características importantes

- Fortran 95
  (1995) acrescentou algumas melhorias ao Fortran 90

- Fortran 2003
  (2003) o último padrão, está em processo de implementação nos compiladores modernos. Nem todos ainda implementaram completamente as regras do padrão 2003

Com o Fortran 95, a linguagem está preparada para a computação paralela, principalmente quando se usa a extensão HPF (High Performance Fortran).

Porque o padrão Fortran 90, a menos das **características** (features) marcadas como **removidas** (deleted) e **obsoletas** (obsolete), está completamente contido no padrão 95 (Fortran 95) todo programa escrito em Fortran 90 que não faça uso das características removidas e obsoletas é também um programa Fortran 95.

O Fortran 95 é uma evolução do padrão Fortran 90, uma pequena revisão onde foram resolvidas questões interpretativas, tanto da semântica quanto da sintaxe do Fortran 90. Além de resolver as questões interpretativas o Fortran 95 ampliou o Fortran 90 com três extensões:

- construtor e declaração FORALL
- procedimentos (procedures) PURE e ELEMENTAL
- inicialização de ponteiros e estruturas

FORALL

A declaração FORALL especifica os elementos de matriz, seções de matriz, subcadeia (*substring*) de caracteres ou alvos de ponteiros como função do índice do elemento. A forma do FORALL é muito parecida com a estrutura da declaração DO encadeada. O grande benefício do FORALL aparece quando é utilizado na computação paralela.

PURE

Funções podem ter efeitos colaterais (*side effects*). Efeitos colaterais são uma permanente dor de cabeça para o programador, mas na programação em paralelo são efeitos catastróficos. O Fortran 95 permite especificar uma função livre de efeitos colaterais, as chamadas funções e sub-rotinas "puras" que são declaradas com o uso da palavra-chave PURE na sua declaração. Uma forma restrita de uma função pura pode ser chamada elementalmente. Estas funções ELEMENTAIS são especialmente importantes para a computação paralela.

INICIALIZAÇÃO

O Fortran 95 resolveu o problema da inicialização de um ponteiro com a função intrínseca NULL e o problema da inicialização (fornecer valores iniciais) para os componentes dos tipos derivados e objetos declarados do tipo automático.

1 – Processador

Um termo muito importante que precisa ser bem entendido é **processador**.

Um processador é a combinação do sistema de computação (o hardware do sistema) e do mecanismo (software) pelo qual os programas são transformados para uso no sistema de computador (neste sistema em particular).

Portanto, processador é a combinação hardware e software que está sendo utilizado.

2 – Compatibilidade

Um programa é considerado compatível com o Fortran 95 (*standard-conforming program*) se ele utiliza somente as formas e relações descritas no padrão Fortran 95 (padrão 95).

> **Neste texto muito frequentemente será apresentado a expressão em inglês. A intenção de disponibilizar o termo em inglês é o de facilitar o processo de procura no texto original.**

Uma unidade de programa é compatível como o padrão Fortran 95 se ela pode ser incluída em um programa Fortran 95 e mesmo assim o programa continua compatível com o padrão Fortran 95.

Um processador é compatível (está de acordo com o padrão Fortran 95) se, mesmo obedecendo as limitações impostas pelo processador (conjunto hardware/software), ele é capaz de:

1. executar qualquer programa Fortran 95 compatível obedecendo as regras e interpretações do padrão Fortran 95
2. detectar e reportar o uso de características obsoletas em uma unidade de programa
3. detectar e reportar o uso de características não listadas no padrão Fortran 95
4. detectar e reportar valores de parâmetro de tipo não suportados pelo processador
5. detectar e reportar formato do código fonte e caracteres não suportados pelo padrão
6. detectar e reportar o uso não consistente das regras de escopo, de nomes, de rótulos, de operadores e símbolos de atribuição
7. detectar e reportar uso de procedimentos intrínsecos não listados no padrão
8. ser capaz de reportar as razões de rejeição de uma unidade de programa submetida à compilação

Entretanto, em uma especificação de formato que não é parte de uma declaração FORMAT, o processador não precisa detectar ou reportar o uso de uma característica obsoleta ou removida ou o uso adicional de formas ou relacionamentos.

Escopo:

Escopo é um conceito (tal como processador) muito importante que na maioria das vezes é visto superficialmente e depois esquecido ou mesmo simplesmente deixado de lado. Erro grave, gravíssimo, pois não ter noção clara do que é escopo dificulta muito o perfeito entendimento da linguagem.

No momento, entenda escopo como sendo o "local" onde recursos são disponíveis para uso.

Num exemplo bastante simplório, imagine que você, seus pais e uma irmã (ou irmão, conforme o caso) vivam na mesma casa. Em principio todos os recursos existentes na residência estão acessíveis (disponibilizados para uso), então o seu escopo seria a casa (todos os recursos existentes na casa estão disponíveis para uso). Bem, na maioria das vezes isto não é verdade, pois existem recursos dentro da casa que você não pode usar. Por exemplo, alguns pertences da sua mãe (seu pai), alguns (ou até mesmo todos) os pertences de sua irmã (irmão). Os recursos indisponíveis para seu uso estão fora do seu escopo. Os escopos do seu pai, da sua mãe, de sua irmã (irmão) são diferentes,mesmo todos vivendo na mesma casa, onde todos os recursos estão fisicamente colocados, não?

Escopo identifica tudo aquilo que está disponível para uso. **O que não está disponível para uso está fora do escopo.**

Um processador compatível com o padrão Fortran 95, pode permitir formas adicionais e relacionamentos também adicionais, desde que garanta que estas adições não conflitem com as formas e relacionamentos definidos no padrão.

Um processador compatível com o padrão Fortran 95 pode acrescentar procedimentos intrínsecos adicionais, mesmo que esta adição possa causar conflito com o nome de um procedimento em uma unidade de programa compatível com o padrão. Se este tipo de conflito vier a ocorrer e a chamada envolver o nome de um procedimento externo, o processador deve usar o procedimento intrínseco a menos que o nome esteja colocado no corpo de uma interface ou que o atributo EXTERNAL seja utilizado no mesmo escopo de unidade.

Um programa compatível com o padrão Fortran 95 não pode utilizar procedimentos intrínsecos não-padrão que foram adicionados pelo processador.

Porque um programa conforme com o padrão Fortran 95 solicita tarefas demasiadas à um processador que não é compatível com o padrão ou que possua itens adicionais que não são portáveis, tais como procedimentos externos definidos por outros meios que não o Fortran 95, a compatibilidade com o padrão não garante que o programa será executado consistentemente em todos os processadores compatíveis com o padrão Fortran 95.

Em alguns casos, o padrão permite a utilização de facilidade que não são completamente compatíveis com o padrão. Estes acréscimos são identificados como dependente do processador (**processador dependente**) e têm que ser utilizados com métodos e semântica determinados pelo processador.

Estas características permitem aos fabricantes de compiladores incrementar seus programas e assim disponibilizar recursos além do padrão de forma que os usuários venham a se interessar pelo seus compiladores e não pelo compilador do concorrente.

Lembre-se ao utilizar os recursos adicionais, que são dependente do processador, você está abrindo mão da portabilidade do programa.

A portabilidade só é garantida pelo padrão, desde que você siga estritamente o padrão

3 – Características Eliminadas e Obsoletas

O padrão Fortran 95 trata com especial atenção o investimento feito (tempo e esforço) ao se escrever programas em Fortran nas versões anteriores pela inclusão de todos os elementos (menos cinco deles) da linguagem Fortran 90 que não são dependente do processador.

O padrão Fortran 95 identifica duas categorias de características (*features*): características removidas (*deleted features*) e características obsoletas (*obsolescent features*).

Existem cinco características marcadas como removidas. São as características consideradas redundantes no FORTRAN 77 e pouco usadas no Fortran 90.

As características obsoletas são consideradas redundantes no Fortran 90, mas ainda são muito utilizadas.

Natureza das característica marcadas como removidas

- existe um método melhor do que aquele existente no FORTRAN 77
- as características removidas não são encontradas no padrão Fortran 95

Natureza das característica marcadas como obsoleta

- existe um método melhor do que aquele existente no Fortran 90
- é recomendado que os programadores não utilizem estes métodos marcados como obsoletos em novos programas e convertam os códigos já escritos para não utilizarem este método
- a característica em questão se tornar irrelevante nos programas escritos em Fortran. Revisão futura do padrão, feito pelo comitê revisor, deve remover esta característica do novo padrão
- o comitê só considerará para remoção no novo padrão as características que estão na lista obsoleta
- processadores devem continuar suportando estas características enquanto elas continuarem sendo largamente utilizadas nos programas Fortran

4 – Módulos

Existem várias facilidades no padrão Fortran 95 que encorajam o uso modular e o reuso de software. Definições de dados (*data*) e de procedimentos (*procedures*) podem ser organizadas em unidades de programas chamadas módulos (subprograma módulo – *modules*) e tornadas acessíveis a outras unidades de programas.

Além disto, um módulo provê facilidade para criar dados globais (*global data*) e biblioteca de procedimentos (*procedures library*), além também de prover um mecanismo para abstração de dados (*data abstraction*).

Consulte o Padrão Fortran 95, texto original:
Fortran 95 standard – Final draft – J3/97-007R2

encontrado em: http://j3-fortran.org/doc/standing/archive/007/97-007r2/pdf/97-007r2.pdf [em 07/03/2009]

5 – Conjunto Básico de Caracteres do Fortran

O Fortran 95 utiliza o seguinte conjunto de caracteres:

- **Letras**

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

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

- **Dígitos**

  0 1 2 3 4 5 6 7 8 9

- **Caracteres Especiais**

	Espaço	**-**	sinal de menos	**$**	sinal de cifrão
'	Apóstrofo	**/**	barra	**;**	ponto e vírgula
"	Aspas	**:**	dois pontos	**<**	sinal de menor
(	Abre parênteses	**=**	sinal de igual	**>**	sinal de maior
)	Fecha parênteses	**_**	sublinha	**%**	sinal de percentagem
*****	Asterisco	**!**	sinal exclamação	**?**	sinal de interrogação
+	Sinal de mais	**&**	e-comercial	**,**	vírgula
.	Ponto				

O Fortran 95 é capaz de imprimir os caracteres da tabela ASCII, ou seja:

- caractere tab (número decimal 9 na tabela ascii)
- e mais os caracteres 32 ate 126, da tabela

Todos os caracteres imprimíveis que não estão no conjunto de caracteres do Fortran 95 só podem aparecer em comentários e em constante do tipo caractere. Além disto, o compilador do Fortran 95 trata as letras maiúsculas e minúsculas equivalentemente, exceto quando estão dentro de uma cadeia de constantes caractere (dentro de um *string*).

6 – Totem

Para um compilador converter o código fonte de um programa escrito em linguagem de alto nível em instruções de máquina (em linguagem de máquina) ele primeiro identifica quais símbolos (os totens ; tokens em inglês) estão presentes no código fonte para então passar à analise de cada um dos totens encontrados. No Fortran 95 os totens são criados com o conjunto de caracteres alfanuméricos (letras e números) e com os caracteres especiais.

Depois do programa fonte ter sido convertido em totens pelo compilador o programa é analisado pelo compilador (o programa escrito em alto nível, programa fonte, passa a ter significado para o compilador). No Fortran 95 os totens são de 5 tipos: **nomes** (ou identificadores), **literais**, **operadores**, **separadores** e **palavras-chaves**.

- **Nomes**
  nomes (ou identificadores) são usados para identificar unidades de programas (programas, funções, sub-rotinas e módulos), variáveis e constantes. Nomes, sempre são construídos com caracteres alfanuméricos, sendo o primeiro caractere, necessariamente não numérico

  exemplos: program teste (o nome teste é usado para identificar a unidade programa)
  massa (pode identificar uma variável de nome massa)
  pi (pode identificar uma constante com nome pi)

- **Literais**
  usados para inserir valores específicos em variáveis e constantes. Podem ser numéricos e não numéricos

  exemplos: 154.32 (identifica o número 154.32)
  .true. (valor lógico verdadeiro)
  "Raul" (constante caractere com o valor Raul)

- **Operadores**
  caracteres especiais (ou combinação de caracteres) que identificam uma operação a ser realizada

  exemplos: + - == > **

- **Separadores**
  símbolos usados para indicar onde os grupos de códigos são divididos, separados ou agrupados

  exemplos: ( ) (/ ; ; %

- **Palavras-chaves**
  o Fortran 95 (Fortran 90), diferentemente das versões anteriores (FORTRAN 77 e anteriores) **não utiliza palavras-chaves reservadas**

  Palavras-chaves geram totens que tem um significado especial para a linguagem. Palavras-chaves reservadas são aquelas que não podendo ser utilizadas com outra finalidade daquela definida pelo padrão da linguagem (por isto são reservadas). Exemplos de palavras chaves: **public else end read print**

  O Fortran 95 possuí palavras-chaves (sim elas existem), entretanto não reserva nenhuma delas. Por isto, caso seja necessário definir uma variável, uma constante ou outra entidade usando o nome **public** isto pode ser feito, ***porque o Fortran 95 irá saber distinguir a variável public da declaração (ou do atributo public) baseado no contexto do programa***

  Entretanto, acredito que deve ser óbvio (principalmente) para um programador que ele deve evitar ao máximo utilizar nomes que sejam palavras-chaves da linguagem

  Por exemplo, considere o segmento de programa:

```
.............
read = 3
if = 2.5
end = if ** read
print*, end
.............
```

  tudo isto está correto, mas não é aconselhável (perda de clareza)!

- *Espaço em branco*
  No Fortran 95 espaço em **branco é** um totem de um caractere que, diferentemente das versões anteriores, é **significativo** conforme o contexto em que é usado

  O uso de **branco(s) dentro de outro totem é proibido**. Para simular o uso de branco dentro de um totem, o recurso mais usado é a utilização da sublinha (_) para substituir o espaço em branco

  exemplo: **massa_da_terra** deve ser usado no lugar de ***massa da terra***

  Um ou mais brancos podem ser usados entre totens, para melhorar o aspecto (mudar a diagramação) do código fonte que é lido por um humano, portanto: **entre totens brancos não são significativos**

- *Comentário*
  o totem de comentário também é um totem de um caractere. O Fortran 95 utiliza a exclamação (!) como símbolo de comentário. A partir da exclamação, tudo que existir, nesta linha, é um comentário (só dentro desta linha, só até o final da linha em questão)

7 – Formato dos Programas Fontes

Um código fonte escrito em **Fortran 90** (Fortran 90, não Fortran 95) pode estar tanto no formato livre quanto no formato fixo. O formato fixo é o mesmo utilizado nas versões anteriores do Fortran 90. O formato livre foi uma novidade incorporada à linguagem Fortran pela versão 90.

É importante que os dois formatos não sejam usados simultaneamente num mesmo programa fonte. É possível (mas não recomendado) que se utilize diferente formatos em diferentes programas fontes.

Como o Fortran (todas as versões) não distingue letras maiúsculas de minúsculas, pode-se escrever um programa fonte todo em maiúscula (não recomendado), todo em minúscula (aceitável) ou misturando-se letras maiúsculas e minúsculas (recomendado), desde que se adote um estilo coerente, bem definido e claro.

Tanto no formato livre quanto no fixo, alguns totens são importantes e merecem ser destacados. São eles:

- **!** indicador de comentário (símbolo ou caractere de comentário)
- **;** separador de declarações
- **&** indicador de continuação (símbolo de continuação)
- **170** rótulo (label) é caracterizado por um número inteiro

8 – Formato Fixo

No Fortran 95, o formato fixo é identificado como um formato obsoleto, por isto mesmo não é recomendável seu uso em programas novos. Ele existe apenas para manter compatibilidade com as versões anteriores.

Quando se utiliza o formato fixo têm-se que escrever os códigos fontes do programa da coluna 1 até a coluna 72 de uma linha. Tudo que estiver escrito depois da coluna 72 até a coluna 80 é ignorado e nenhuma mensagem de aviso será emitida pelo compilador por ter desconsiderado o que está escrito entre as colunas 73 e 80.

No **formato fixo**, exceto no contexto de caractere (string) **espaço em branco não é significativo** e por isto pode ser usado livremente pelo programa todo. É recomendável, entretanto, usar os espaços em branco desta forma somente quando seu uso aumentar a legibilidade do programa.

A posição de coluna para cada campo segue a seguinte regra:

Campo	Coluna
Rótulo da declaração	1 a 5
Indicador de continuação	6
Declarações do Fortran	7 a 72
Uso livre	73 a 80

A figura 1, obtida na wikipedia (Título: Punched card site: http://en.wikipedia.org/wiki/Punch_card em 12/11/2006) mostra uma foto de um cartão perfurado.

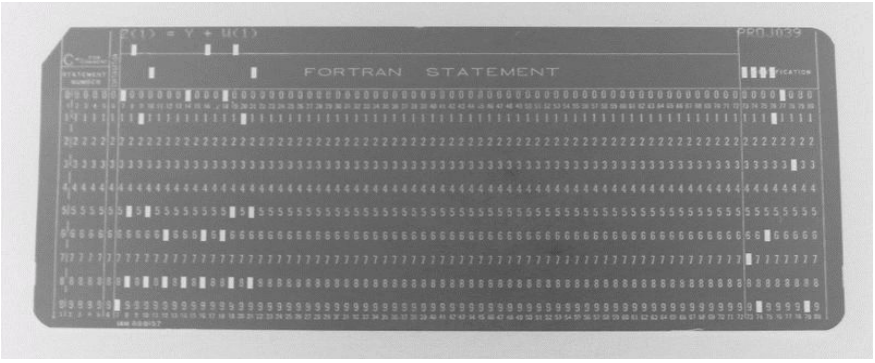

Figura 1 – Cartão perfurado de 80 colunas com o código: Z(1) = Y + W(1)

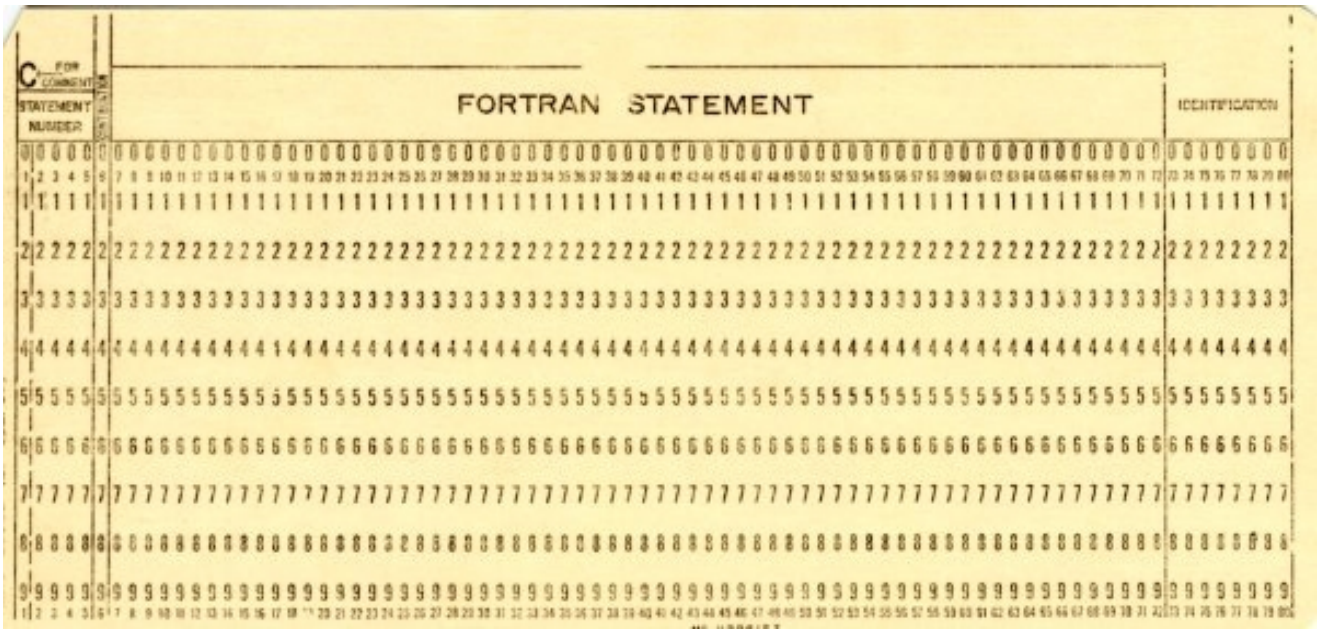

Figura 2 - Cartão perfurado. FONTE: http://pt.wikipedia.org/wiki/Cartão_perfurado [em 04/03/2009]

Para exemplificar, um seguimento de programa bastante simples, escrito no formato fixo, é mostrado a seguir:

```
         1         2         3        |estes números imitando uma régua foram
123456789012345678901234567890    <== |colocados aqui apenas para facilitar a
|    ||  |         |         |        |visualização das posições ocupadas
      program soma
      real x, y, c

      read*, x, y
      c = x
     * +  y
c impressao do resultado

      print*, c
   20 stop
      end
```

Apresentação esquemática de um cartão

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	..	..	..	80
						p	r	o	g	r	a	m		s	o	m	a													
						r	e	a	l		x	,	y	,	c															
						r	e	a	d	*	,		x	,		y														
						c		=		x																				
					*		+	y																						
c		i	m	p	r	e	s	s	a	o		d	o		r	e	s	u	l	t	a	d	o							
						p	r	i	n	t	*	,		c																
			2	0		s	t	o	p																					
						e	n	d																						

• **Indicador de comentário** caractere **c** ou * na primeira coluna ou então uma linha inteiramente branca	• **Indicador de continuação** qualquer caractere, exceto 0 (zero) ou branco, escrito na coluna 6 que não esteja numa linha de comentário	• **Rótulo** (label) 1 a 5 decimais (1 até 99999) escrito nas coluna 1 a 5. O número não pode ser repetido. Não necessitam estar ordenados

9 – Formato Livre

O **Fortran 95 só dispõe do formato livre**, enquanto o *Fortran 90 permite o uso tanto do formato livre quanto do formato fixo*.

As principais características do formato livre são:

- Dentro de uma linha válida, **escreva aonde quiser**
- Comprimento de uma linha pode ser de até **132 caracteres**
- Comentários são escritos após o símbolo de exclamação (!)
- O caractere e-comercial (&) é usado para caracterizar que existe uma linha de continuação, isto é, ele é o indicador de linha de continuação
- Letras maiúsculas e minúsculas podem ser usadas sem distinção
- Ponto e vírgula (;) pode ser usado para separar mais de um comando por linha (separador de declarações). **Não utilize esta opção, porque ela reduz muito a legibilidade do programa**

- Espaços em branco são significativos

  Conforme o contexto, espaços em branco serão significativos. Espaço em branco não pode aparecer dentro de um totem.. Por exemplo: o totem "**print**" não pode ser escrito como:
  "**p r i n t**", "**p rint**", "**pri nt**" ou qualquer outra combinação com branco

  Brancos podem (e devem) **ser usado entre totens** para aumentar a legibilidade do programa

  A expressão " **`k=x*y`** " pode ser escrito " **`k = x * y`** " sem problema algum

Algumas palavras-chaves (goto, elseif, inout, selectcase e aquelas que iniciam com a palavra end) podem ser vistas como sendo formada por um totem único ou então composta de dois totens. Elas podem ser escritas com ou sem branco entre elas.

Lista das palavras-chaves que não admitem branco e as que podem utilizar

Branco opcional	**Branco requerido**
`Block data`	`case default`
`double precison`	`do while`
`else if`	`implicit` *`type-specifer`*
`end block data`	`implicit none`
`end do`	`interface assignment`
`end file`	`interface operator`
`end function`	`module procedure`
`end if`	`recursive function`
`end interface`	`recursive subroutine`
`end module`	`recursive` *`type-specifier`* `function`
`end program`	*`type-specifier`* `function`
`end select`	*`type-specifier`* `recursive function`
`end subroutine`	
`end type`	
`end where`	
`go to`	
`in out`	
`select case`	

10 – Comentários

Comentários podem e devem ser usado com liberdade para aumentar a legibilidade do programa e gerar a documentação interna.

As regras que se aplicam aos comentários são:

- Todos os caracteres (tudo) escritos depois de um ponto de exclamação (!) é um comentário
- Quando o caractere de exclamação aparece **dentro de uma cadeia de caracteres** (string) ele **não é** interpretado como sendo **um comentário**
- Comentário pode ser usado dentro de uma linha
- O caractere de exclamação (!) aparecendo no início da linha indica uma linha inteira de comentário
- Uma linha em branco é também entendida como sendo uma linha de comentário.

Exemplo:

```
program tempo_de_vida
  !-----------------------------------------------------
  ! Propósito: Programa exemplo contendo comentários
  !            Calcula quantos anos uma pessoa tem em 2006
  !-----------------------------------------------------
  ! Autor: Anibal L. Pereira   12/11/2006
  !-----------------------------------------------------
  implicit none
  integer:: ano_nascimento, anos_vida
! esta é uma linha de comentário

  ! as linhas em branco acima e abaixo desta também são linhas de comentário

  read*, ano_nascimento                ! solicita o ano de nascimento
  anos_vida = 2006 - ano_nascimento    ! calcula quantos anos tem em 2006

  print*, "Esta exclamação ! não é interpretada como um comentário"

  print*, "Em 2006 você tem ", anos_vida,  "anos de vida"

end program tempo_de_vida
```

11 – Linhas de Continuação

Uma declaração no Fortran 95 é escrita em uma linha única, uma declaração por linha. Mais de uma declaração [separadas por ponto e vírgula (;)], na mesma linha, é possível mas não recomendado.

Se uma declaração é muito longa para caber em uma linha (mais de 132 caracteres) ela pode ser continuada. Se necessário, pode-se utilizar até **39 linhas de continuação no formato livre**.

As regras básicas da utilização do indicador de continuação são:

- linha que termina com o símbolo **&** (e-comercial ou ampersand em inglês) indica que ela será continuada, a menos que ele apareça dentro de um string ou dentro de um comentário
- a continuação é feita pela primeira linha não comentada que segue o indicador de continuação

Observe a declaração destacada no programa linhas.

```
program linhas
.....................
.....................
carros_produzidos = 200.5 * (horas_trabalhadas &
+ horas_extras)/ 15.2
.....................
.....................
end program linhas
```

A declaração é equivalente a:

```
carros_produzidos = 200.5 * (horas_trabalhadas + horas_extras) / 15.2
```

ou seja, as duas linhas destacadas no programa fonte, para o compilador, na realidade formam uma linha única.

Com relação ao uso do caractere de continuação pode-se destacar o seguinte:

- O caractere de continuação, **&**, não faz parte da declaração
  Ele é usado apenas para indicar que a linha em que está colocado continua. O compilador não escreve este caractere na declaração quando faz a tradução para a linguagem de máquina.

- Pode-se escrever linhas de comentários entre o indicador de continuação (&) e a linha de continuação
  Isto é possível porque o compilador ignora as linhas de comentários.

O programa linhas_2 faz o mesmo que o programa linhas:

```
program linhas_2
.....................
.....................
carros_produzidos = 200.5 * (horas_trabalhadas &
! esta é uma linha de comentário colocada entre as linhas de continuação
+ horas_extras))/ 15.2
.....................
.....................
end program linhas_2
```

Também é possível escrever assim:

```
program linhas_3
.....................
.....................
carros_produzidos = 200.5 * (horas_trabal&
     &hadas + horas_extras))/ 15.2
.....................
.....................
end program linhas_3
```

Observe que no programa linhas_3 existe um caractere de continuação, &, escrito no início da linha de continuação. A primeira linha terminou com o símbolo & e a linha de continuação iniciou com o símbolo &. Neste caso não pode haver espaços antes e depois dos &. O resultado será:

```
carros_produzidos = 200.5 * (horas_trabalhadas + horas_extras))/ 15.2
```

Caso fosse escrito como mostrado a seguir, o código está errado.

```
carros_produzidos = 200.5 * (horas_trabal   &
       &hadas + horas_extras))/ 15.2
```

Errado porque será escrito da forma mostrada abaixo (o erro foi destacado em negrito):

```
carros_produzidos = 200.5 * (horas_trabal    hadas + horas_extras))/ 15.2
```

Importante : A divisão de variáveis, constantes e números conforme visto pode ser feita mas, é recomendável não se fazer uso deste recurso, para realizar esta tarefa, porque isto reduz muito a legibilidade do programa. Utilize este recurso somente quando for impossível evitá-lo ou então quando isto permitir aumentar a legibilidade do programa

12 – Tipos de Declarações

As declarações do Fortran podem ser classificadas em dois tipos: executáveis e não-executáveis.

- ### Declarações executáveis

também chamadas de comandos ou instruções, são as declarações que instruem o computador qual ação deve ser realizada. Elas permitem a execução de cálculos, entrada e saída (I/O) de dados e controlam o fluxo do programa.

- ### Declarações não-executáveis

aquelas que descrevem características das unidades do programa, dos dados, contém informações para edição, etc. Estas declarações, obviamente, não produzem nenhuma ação ou execução.

Para uma definição mais explicita, pode-se também utilizar a seguinte classificação:

- **Declaração de controle**
  ou declaração de fluxo, permite alterar a seqüência de execução das declarações usadas num programa

- **Declaração de entrada e saída (I/O)**
  permite realizar a transferência de informação entre dispositivos ou entre o computador e um usuário

- **Declaração de subprograma**
  atribui nomes e passa argumentos (também chamados de parâmetros) para os subprogramas (funções, sub-rotinas e módulos)

- **Declaração de especificação**
  Usada para identificar uma ou mais características: são comandos não-executáveis

- **Declaração de atribuição**
  entra com um valor na variável ou constante (escreve ou atribui um valor)

O programa area_circulo identifica cada uma das declarações utilizadas.

```
program area_circulo                                    ! declaração de subprograma       (instrução)
 ! programa para calcular a área do circulo             ! comentário
 ! escrito por  ALP em 06/04/2002                       ! comentário
 real    :: raio, abc                                   ! declaração de especificação
 integer :: abc                                         ! declaração de especificação
                                                        ! linha em branco – comentário
 abc = 2                                                ! declaração de atribuição
 read (unit =*, fmt =*)  raio                           ! declaração de I/O               (instrução)
 print*,"area do circulo =", 3.14159 * ((raio)**abc)    ! declaração de I/O               (instrução)
end program area_circulo                                ! declaração de subprograma       (instrução)
```

13 – Estrutura de um Programa em Fortran 95

Um programa Fortran 95 tem uma das seguinte estruturas:

```
program nome_do_programa

  [parte contendo especificações do programa]

  [parte contendo os comandos do programa]

  [parte contendo os subprogramas]

end program nome_do_programa
```

```
program nome_do_programa
implicit none

  [parte contendo especificações do programa]

  [parte contendo os comandos do programa]

contains

  [parte contendo os subprogramas]

end program nome_do_programa
```

- o texto que aparece entre colchetes "[ ]" é opcional

- o programa inicia com a palavra-chave **program**
  seguido do nome do programa (*nome_do_programa*). O padrão Fortran 95 não obriga o uso de um nome para o programa, entretanto é altamente recomendado o seu uso

- comando **implicit none**
  seu uso é recomendado pela boa técnica de programação, porque obriga o programador a declarar todas as variáveis (procedimento altamente recomendado).
  A boa técnica de programação considera a declaração das variáveis necessária, apesar do padrão Fortran 95 não obrigar seu uso. Muito provavelmente a futura versão do Fortran irá suprimir a declaração implícita de variáveis, o que eliminará a necessidade de se utilizar esta declaração de anulação

- parte contendo as **especificações** do programa
  colocam-se aqui as especificações que serão usadas no programa

- parte contendo os **comandos** do programa
  colocam-se aqui as declarações de instruções usadas no programa
  não opcional quando se deseja que o programa faça algo

- parte contendo os **subprogramas**
  aqui será colocada a parte que contem um conjunto de subprogramas internos (opcional).
  Se existir, os subprogramas, necessariamente, estarão colocados depois da palavra-chave contains.
  Caso não existam subprograma, a palavra-chave contains pode ser omitida

- terminando com a palavra-chave **end program**
  a declaração end program deve ser seguida pelo nome do programa.
  O uso do nome do programa é obrigatório se ele foi usado na declaração program, caso contrário é opcional

Além disto, todo programa deve ter (para manter sua legibilidade) linhas de comentários.

14 – Ordem das Declarações

A figura 3 identifica a ordem em que as declarações podem aparecer em todos as unidade de programas e subprogramas no Fortran 95. As colunas (linhas verticais) separam tipos de declarações que podem ser intercaladas. As linhas horizontais separam declarações que não podem ser intercaladas.

Subprogramas módulo e subprogramas internos tem que ser escritos depois da declaração CONTAINS.

Em um subprograma as declarações não executáveis, escritas entre a declaração USE e a declaração CONTAINS, geralmente precedem as declarações executáveis, entretanto as declarações ENTRY, FORMAT e DATA podem ser colocadas no meio de declarações executáveis (entretanto, de preferencia em colocá-las antes das declarações executáveis. Alguns programadores adotam o estilo de colocá-las no final do programa).

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, MODULE statement</td></tr>
<tr><td colspan="3">USE statement</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY
statements</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER statement</td><td>IMPLICIT statements</td></tr>
<tr><td>PARAMETER and DATA
statements</td><td>Derived-Type Definition, Interface blocks, Type declaration statements, Statement function statements and specification statements</td></tr>
<tr><td>DATA statements</td><td>Executable constructs</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">Internal or module procedures</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

Figura 3 - Ordem das Declarações

As declarações executáveis de um programa devem ser colocadas na ordem lógica do programa.

A declaração IMPLICIT NONE tem que ser colocada logo depois da declaração de associação USE.

As declarações PUBLIC e PRIVATE só são permitidas dentro de módulos.

O quadro que segue identifica quais declarações as unidades de escopo podem conter.

Declarações	**Unidade de escopo**						
	Programa Principal	Módulo	Bloco dados	Subprograma Externo	Subprograma Módulo	Subprograma Interno	Corpo Interface
Declaração USE	sim	sim	sim	sim	sim	sim	sim
Declaração ENTRY	não	não	não	sim	sim	não	não
Declaração FORMAT	sim	não	não	sim	sim	sim	não
Declaração DATA	sim	sim	sim	sim	sim	sim	não
Definição tipo-derivado	sim	sim	sim	sim	sim	sim	sim
Bloco Interface	sim	sim	não	sim	sim	sim	sim
Declaração executável	sim	não	não	sim	sim	sim	não
Declaração CONTAINS	sim	sim	não	sim	sim	não	não
As declarações : PARAMETER IMPLICIT de especificação de declaração de tipo	sim	sim	sim	sim	sim	sim	sim

15 – Declarações Obsoletas

A adoção de um novo padrão para a linguagem requer que haja um certo tempo de espera antes de se remover declarações da linguagem. Este intervalo de tempo é usado para divulgação das declarações que estão sendo removidas.

Todas as declarações que serão removidas na versão seguinte são conservadas no padrão atual mas, marcadas como declarações obsoletas.

Fortran 90

As declarações do FORTRAN 77 que foram conservadas no Fortran 90 apenas para manter a compatibilidade entre as versões, porque o Fortran 90 tem formas mais adequadas e poderosas para realizar a mesma tarefa, são as declarações obsoletas.

Declarações Obsoletas:

- **IF aritmético**
  recomenda-se o uso da **declaração bloco if-then-else** para substituir esta declaração

- **Assigned FORMAT specifiers**
  recomenda-se o uso de expressões caractere para definir especificações de formato

- **ASSIGN** e **assigned GOTO**
  usualmente são usadas para simular subprogramas internos, o que agora pode ser feito facilmente

- **alternate RETURN**
  recomenda-se retornar um inteiro e testar este inteiro para definir ações subseqüentes

- **PAUSE statement**
  recomenda-se usar uma declaração read que espera por uma dado de entrada

- **H edit descriptor**
  recomenda-se o uso de aspas (") ou apóstrofo (')

Características removidas:

- variáveis reais e de dupla precisão na declaração **do** e nas expressões de controle
- compartilhar mesma terminação de vários laços de iteração (declaração **do**)
- término da declaração **do** em outra declaração que não seja as declarações **continue** ou **end do**
- possibilidade de entrar dentro de uma declaração **if** vindo de fora

Fortran 95

O Fortran 95 removeu algumas das características e declarações marcadas como obsoletas no Fortran 90 e identificou outras declarações como obsoletas.

Com a remoção das declarações obsoletas do Fortran 90, as declarações consideradas redundantes no FORTRAN 77 não mais estão disponíveis no Fortran 95.

São elas:

- **Assigned FORMAT specifiers**
  recomenda-se o uso de expressões caractere para definir especificações de formato
- **ASSIGN e assigned GOTO**
  usualmente são usadas para simular subprogramas internos, o que agora pode ser feito facilmente
- **PAUSE statement**
  recomenda-se usar uma declaração read que espera por uma dado de entrada
- **H edit descriptor**
  recomenda-se o uso de aspas (") ou apóstrofo (')

Características removidas:

- variáveis reais e de dupla precisão em declaração **do** e em expressão de controle
- possibilidade de entrar dentro de uma declaração **if** vindo de fora
- formato fixo
- compartilhar mesma terminação de várias declarações **do**
- statement functions (função de comando = definida pelo programador numa única declaração)

Características obsoletas:

- IF aritmético
- computed GOTO
- alternate RETURN
- DATA statement
- CHARACTER*(*) forma da declaração character

16 – Tipos de Dados

O Fortran 95 tem a capacidade de manipular três tipos de dados:

1) **números** muito freqüentes nas aplicações científicas

2) **caracteres** usado para guardar informações literais

3) **bits** usado para guardar informação lógica e gráficos no formato bitmap

Todo os três tipos de dados (**tipo**, por simplicidade) sempre têm associado à eles:

- um nome (identificador)
- um conjunto de valores
- uma forma de atribuir valores e
- um conjunto de operações

Uma classificação mais informativa é obtida dividindo-se os tipos em: numéricos e não-numéricos.

- **Tipo Numérico**

  integer
  real
  complex

- **Tipo Não-numérico**

  character
  logical

É importante lembrar que o **tipo** do dado, **sempre**:

> **define a forma como o dado é guardado internamente no computador e quais são as regras e operações que se aplica à ele**.

Esta lembrança e entendimento é muito importante porque permitirá resolver questões aparentemente sem sentido quando não se leva em consideração este conhecimento.

Lembre-se o computador tem formas e características próprias para lidar e armazenar dados e informações que são diferentes daquelas com as quais estamos habituados.

17 – Dados do Tipo Numérico

Quando se faz a identificação do tipo da variável ou de uma constante, está-se, simultaneamente, especificando o nome da variável ou da constante. Isto é chamado de definição da variável ou da constante.

A identificação do tipo (declaração de tipo) define como o conteúdo (o dado) será armazenada internamente no computador e quais são a regras e as operações que se pode aplicar à ele.

Por exemplo, uma variável inteira só pode armazenar valores do tipo inteiro, uma variável do tipo lógica só pode armazenar valores lógicos, e assim por diante. Além disto, a operação de divisão de dois números inteiros é realizada internamente ao computador de forma diferente da mesma operação (divisão) de dois números reais. Isto podem gerar resultados diferentes, conforme o contexto utilizado.

- ## Números Inteiros

Números inteiros são bastante usados, principalmente nos processos de contagem e indexações. A grande vantagem da utilização do número inteiro vem do fato da sua representação interna nos computadores ser exata.

Por exemplo, os números `1, -1, 65, 0, 450 e -450` são números inteiros típicos e por isto devem ser guardados numa variável do tipo inteiro. O número `133147` é também um inteiro.

A figura 4 exemplifica como o computador representa internamente o número inteiro `133147`:

número 133147 (decimal)			
1º Byte	2º Byte	3º Byte	4º Byte
0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	0 0 0 0 1 0 0 0	0 0 0 1 1 0 1 1
4 Bytes = 32 bits			

Figura 4 - Representação interna do inteiro `133147` no computador

A figura 4 mostra a representação binária do número inteiro **`+133147`**, isto é o computador guarda o número 133147 como **00000000000000100000100000011011**. Para possibilitar uma melhor visualização os 4 bytes apareceram separados por espaços (isto não ocorre no computador).

Usualmente a técnica chamada de complemento de dois (*two's complement*) é usada para representar números inteiros internamente nos computadores.

A técnica segue a seguinte regra:

- Número inteiro positivos são escritos diretamente
- Número inteiro negativos são obtidos escrevendo-se o inteiro positivo, invertendo-se todos os bits e depois adicionando-se um ao número

Para exemplificar, considere a representação binária (usando 4 bits = 1 nibble) do número inteiro 3.

Binário		**Decimal**
0011	$= -\ 0x2^3 + (\ 0x2^2 + 1x2^1 + 1x2^0\ ) = 0 + (0 + 2 + 1)$	**+3**
1101	$= -\ 1x2^3 + (\ 1x2^2 + 0x2^1 + 1x2^0\ ) = -\ 8 + (4 + 0 + 1)$	**-3**

A expressão que permite avaliar um número binário inteiro, usando-se a regra do complemento de 2 é:

$$n = -b_0 \cdot 2^{n-1} + \sum_{i=1}^{n-2} b_i \cdot 2^{n-1-i}$$

Então o número inteiro representado na figura 4 **`00000000000000100000100000011011`** é:

$$n = -0\ x2^{31} + 0\ x2^{30} + 0\ x2^{29} + 0\ x2^{28} + 0\ x2^{27} + 0\ x2^{26} + 0\ x2^{25} + 0\ x2^{24} + \quad (1^0 \text{ Byte})$$
$$0\ x2^{23} + 0\ x2^{22} + 0\ x2^{21} + 0\ x2^{20} + 0\ x2^{19} + 0\ x2^{18} + 1\ x2^{17} + 0\ x2^{16} + \quad (2^0 \text{ Byte})$$
$$0\ x2^{15} + 0\ x2^{14} + 0\ x2^{13} + 0\ x2^{12} + 1\ x2^{11} + 0\ x2^{10} + 0\ x2^{09} + 0\ x2^{08} + \quad (3^0 \text{ Byte})$$
$$0\ x2^{07} + 0\ x2^{06} + 0\ x2^{05} + 1\ x2^{04} + 1\ x2^{03} + 0\ x2^{02} + 1\ x2^{01} + 1\ x2^{0} = \quad (4^0 \text{ Byte})$$

$$n = 1\ x2^{17} + 1\ x2^{11} + 1\ x2^{4} + 1\ x2^{3} + 1\ x2^{1} + 1\ x2^{0} =$$

$$n = 1 \times 131072 + 1 \times 2048 + 1 \times 16 + 1 \times 8 + 1 \times 2 + 1 \times 1 = 133147$$

Computadores que utilizam palavras de 4 Bytes (32 bits) são bastantes comuns atualmente. Observe que usando 32 bits o computador pode representar números inteiros, positivos e negativos, entre:

- -2^{31} - 2 147 483 648 (limite inferior)
- $+2^{31}-1$ + 2 147 483 647 (limite superior)

Um computador que utilize uma palavra de 2 bytes (16 bits) pode representar números inteiros na faixa:

-32666 até **+32767** **[** (-2^{15}) até $(+2^{15}-1)$ **]**

Uma característica importante de uma variável ou constante inteira é que ela não têm parte fracionária e por isto não podem conter o ponto decimal.

Para os inteiro, admite-se que o ponto decimal está fixo a sua direita e por isto ele também é chamado de número de ponto fixo.

- ## Números Reais

Números inteiros podem ser representados de forma exata, dentro de sua faixa de utilização. Entretanto eles não podem representar números com parte fracionária, que são a maioria dos números com que trabalhamos.

Números reais, aqueles que possuem parte decimal, são também chamados de número de ponto flutuante, pois podem ter mais de uma representação.

Por exemplo, o número `0.1` pode ser representado por uma das formas:

$0.1 = 1.0 \times 10^{-1} = 10.0 \times 10^{-2}$

Diferentemente do números inteiros (ponto fixo) os número reais devem ser escritos contendo o ponto decimal.

Então: os números **`4.`** (número quatro seguido do ponto decimal), **`7.`** e **`39.`** são números reais, mesmo que não tenham a sua parte decimal representada. A utilização do ponto decimal no número caracteriza um dado do tipo real. Entretanto, a melhor forma de escrever um número real é aquela em que se é bastante explícito, por isto deve-se usar o ponto decimal e o zero. Deve-se escrever: **`4.0`** ; **`7.0`** e **`39.0`**. O zero depois do ponto não deve ser opcional.

Observação: Para nós brasileiros deveria ser " vírgula decimal" e não "ponto decimal", mas como o Fortran utiliza a convenção americana, os programas fontes deverão ser escritos com ponto e não com vírgula. Por isto utiliza-se a expressão ponto decimal e não vírgula.

Representação interna de um número real

Os computadores representam internamente os números reais e inteiro de uma forma bastante diferente. A figura 5 destaca (entre outras informações) como o computador armazenar um número do tipo real usando uma palavra de 4 bytes (32 bits).

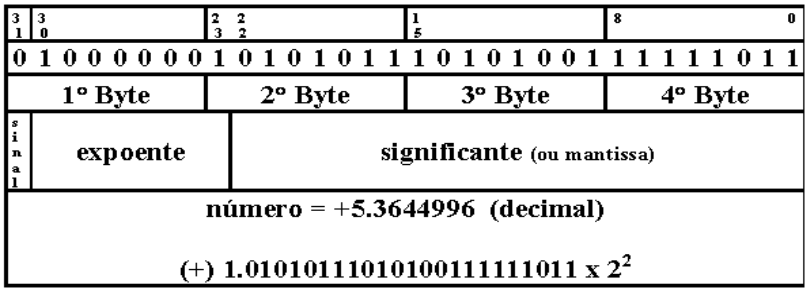

Figura 5 - Representação de um número real +5.3644996

Os números reais são escritos usando-se essencialmente duas partes: o significante (também chamado de principal ou mantissa) e o expoente.

O significante contem a parte decimal do número (bits 0 a 22) e é escrito com o ponto decimal que pode aparecer no início ou no meio do mesmo.

O expoente (bits 23 a 30) representa a potência de 2 (lembre-se o computador é binário, base dois) que deve multiplicar o significante para a obtenção o número real.

O primeiro bit (o bit 31) é usado para caracterizar o sinal do número.

Observe que os números reais que um computador pode representar estão limitados pelos números de bits que eles usam tanto para guardar o significante (mantissa) quanto para guardar o expoente.

Um número real é então expresso por: $\pm d.dd\ldots d \times B^e$

Observe que utilizam-se **p** dígitos para guardar o valor do significante (principal ou mantissa) e o expoente foi identificado pela letra e.

$$\pm \underbrace{d.dd\ldots d}_{\substack{significante\\ com\\ p\ dígitos}} \times \underbrace{B^e}_{\substack{B=base\\ e=expoente}}$$

A expressão que permite calcular o número é: $\pm\left(d_0 + d_1 B^{-1} + \cdots + d_{p-1} B^{-(p-1)}\right) \cdot B^e$

Para exemplificar, considere o número real binário que está representado na figura 5.

0 1 0 0 0 0 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 0 1 1 1 1 1 1 1 0 1 1

- **Sinal** (1 bit)
  0 = (bit número 31) representa o sinal [zero é positivo; um é negativo]

- **Expoente** (8 bits)
  1 0 0 0 0 0 0 1 (bits 23 a 30)
  é o número: $1x2^7 + 0x2^6 + 0x2^5 + 0x2^4 + 0x2^3 + 0x2^2 + 0x2^1 + 1x2^0 = 128 + 1 = 129$
  como o expoente do número pode ser positivo ou negativo, ele deve ser calculado pela expressão **[e = x – 127]**, então: 129 – 127 = +2. Finalmente o expoente fica sendo: **e = 2**

  observe que subtrair 127 do número permite representar expoentes positivos e negativos sem que se reserve um bit (dos 8 bits utilizados) para se guardar o sinal. Esta técnica de deslocamento (bias) é o padrão adotado

- **Significativo** (23 bits)
  0 1 0 1 0 1 1 1 0 1 0 1 0 0 1 1 1 1 1 1 0 1 1 (bits 0 a 22)
  temos:

$$0 \times 2^{-01} + 1 \times 2^{-02} + 0 \times 2^{-03} + 1 \times 2^{-04} + 0 \times 2^{-05} + 1 \times 2^{-06} + 1 \times 2^{-07} + 1 \times 2^{-08} +$$
$$0 \times 2^{-09} + 1 \times 2^{-10} + 0 \times 2^{-11} + 1 \times 2^{-12} + 0 \times 2^{-13} + 0 \times 2^{-14} + 1 \times 2^{-15} + 1 \times 2^{-16} +$$
$$1 \times 2^{-17} + 1 \times 2^{-18} + 1 \times 2^{-19} + 1 \times 2^{-20} + 0 \times 2^{-21} + 1 \times 2^{-22} + 1 \times 2^{-23} =$$

$$0.25 + 0.0625 + 0.015625 + 0.0078125 + 0.0039062 +$$
$$0.0009765 + 0.0002441 + 0.0000305 + 0.0000152 +$$
$$0.0000076 + 0.0000038 + 0.0000019 + 0.0000009 + 0.0000002 + 0.0000001 = \mathbf{0.3411245}$$

  Como o resultado é normalizado, temos que fazer $\mathbf{d_0 = 1}$, então, o resultado final será: **1.3411245**

  o resultado obtido foi próximo do ***valor real*** **(1.3411249)**. A diferença de 0.0000004 é conseqüência dos cálculos terem sido feitos em uma calculadora de mão cuja arredondamento prejudicou a exatidão dos resultados

Finalmente tem-se o número real: + 1.3411249 x 2^2 = 5.3644996

Para dar ênfase e ficar ainda mais evidente que números inteiros e números reais são tratados internamente de forma completamente diferente pelo computador, veja as representações interna do **número 10** quando ele é definido como sendo do tipo inteiro e do tipo real:

inteiro: `00000000000000000000000000001010` = **10** (2+8=10)

real: `01000001001000000000000000000000` = **10.000000** ($+1.2500000 \times 2^3$)

- ## Números Complexos

Fortran é uma linguagem que pode trabalhar diretamente com números complexos. Número complexos são bastante úteis para cálculos em varias área da ciência (por exemplo: electromagnetismo e ótica).

Um número complexo é representado por um par de números reais:

x + j y ou então **x + i y**

onde **j** (ou **i**) é o número imaginário $\sqrt{-1}$.

O primeiro componente do número representa a parte real do número complexo enquanto o segundo componente representa a parte imaginária do número complexo.

Em Fortran um número complexo é escrito da seguinte forma:

z = **(7.5, -3.1)** ! z é o número complexo: z = 7.5 - **i** 3.1 ou se preferir z = 7.5 - **j** 3.1

18 – Dados do Tipo Não-numérico

Dados do tipo não-numéricos são dados do tipo caractere ou lógico.

- ## Caractere

Um objeto do tipo caractere (**character**) é formado por uma cadeia de caracteres (também chamado de string). Por exemplo na variável caractere bairro está sendo colocado o valor *tijuca*:

bairro = "tijuca"

O conjunto de caracteres que forma a palavra ***tijuca*** [t i j u c a] é uma cadeia de caracteres que está sendo guardada na variável bairro, que, obviamente, é uma variável do tipo caractere.

Toda variável ou constante do tipo caractere tem que ter um comprimento. Este comprimento (tamanho) é definido quando a variável ou constante é criada. Veja a definição da variável caractere chamada **nome**:

character (len=10) :: nome ! define a variável caractere nome como comprimento de 10 caracteres

Variáveis e constantes do tipo caractere podem ser justapostas (concatenadas) pelo operador concatenação "//" (barra-barra ou duas barras) que , no Fortran, é o único operador que existe para trabalhar com strings. Então:

nome // sobrenome ! **concatena** a variável caractere **nome** com a variável caractere **sobrenome**

Caso estas duas variáveis caractere contenham os valores:

nome = "Paulo"
sobrenome = "Cesar"

a concatenação **nome // sobrenome** terá como resultado **PauloCesar**

Observe que é possível fazer a seguinte operação:

nome // " " // sobrenome e obter: **Paulo Cesar**

- ## Lógico

Uma entidade (variável ou constante) do tipo lógico só pode assumir dois valores: verdadeiro (**.true.**) ou falso (**.false.**).

Observe uma característica muito importante: os pontos antes e depois de true e false: **`.true.  .false.`**

A representação interna de uma variável ou constante lógica é feita usando-se somente um bit, porque são necessários somente dois valores (0 ou 1) para representar o seu conteúdo.

Variáveis e constantes lógicas têm seu valor definido por uma declaração de atribuição. Por exemplo, seja a variável "ligado" uma variável lógica, então:

```
ligado = .true.     ! variável lógica ligado passa a conter o valor verdadeiro
ligado = .false.
```

Usando-se as declarações **write** e **read** pode-se mostra e ler os valores das variável e das constantes (de qualquer tipo).

```
...................
...................
logical:: ligado
...................
...................
ligado = .true.
write(unit=*,fmt=*) ligado
...................
...................
```

a **saída** que será mostrada na tela do micro **será** a letra **T**

saída **T** ou **F** será mostrado para os valores .true. e .false.

entrada **T** ou **F** podem ser fornecidos para entrar com o valor .true. e .false.

Como em todo tipo de dado existe um conjunto de operadores lógicos com os quais se pode avaliar e criar expressões lógicas.

19 – Substring

Substrings significa subcadeia de caracteres. Uma subcadeia é uma seqüência de uma ou mais unidades de armazenamento (posições) dentro de uma cadeia de caracteres (string).

Na cadeia de caracteres (string) as posições são numeradas da esquerda para a direita, a partir do número um.

Usando-se o conceito de subcadeia (substring) pode-se manipular uma parte do string.

- **FORMA GERAL**

<nome da variável>(exp1 : exp2)

onde: exp1 e exp2 são expressões aritméticas inteiras

exp1 --> especifica a posição do primeiro caractere do substring

exp2 --> especifica a posição do último caractere do substring

Se exp1 estiver ausente então o seu valor é assumido ser um. Então (**:exp2**) é o mesmo que (**1:exp2**).

alfa="abcdefgh"	conteúdo do string	a	b	c	d	e	f	g	h
	posições do string	1	2	3	4	5	6	7	8

então:

alfa(:5) será igual a "**abcde**" observe que é o mesmo que: alfa(1:5), pelo omissão de exp1

Se exp2 estiver ausente (exp1:) o valor é assumido ser igual ao do último caractere do string

alfa = "abcdefgh"

alfa(3:) será igual a "**defgh**" neste caso tem-se o mesmo que alfa(3:8), pois exp2=8

Observe que:

alfa(2:5)	será igual a "bcde"
alfa(3:3)	será igual a "c"
alfa(:)	será o string inteiro "abcdefgh"
alfa(1:3)	será "abc"

Comando de atribuição com substrings

Um substring pode ser usado do lado esquerdo de um comando de atribuição. Neste caso o valor fornecido será escrito nas posições especificadas do string.

Considere a variável caractere alfa:

alfa="abcdefghi"	a	b	c	d	e	f	g	h	i	conteúdo do string
	1	2	3	4	5	6	7	8	9	posições do string

As declarações a seguir produzem novos valores:

alfa(3:5)="123"
teremos um novo valor para alfa ==> alfa = "ab123fghi"

alfa(1:1)="1"
novo valor ==> alfa = "1bcdefghi"

alfa(6:)="zxwy"
novo valor ==> alfa = "abcdezxwy"

20 – Nomes

Toda entidade no Fortran 95 deve possuir um nome (identificador). Um nome válido em Fortran 95 deve seguir as seguintes regras:

- pode conter até 31 caracteres
- só podem conter letras, números e o símbolo sublinha (em inglês underscore)
- o primeiro caractere tem que ser uma letra
- pode ser escrito com letra maiúscula, letra minúscula ou com os dois tipos

  o uso de letras maiúsculas e minúsculas numa mesma entidade só deve ser usada quando se intenciona usar a técnica do "camelo" para escrever os nomes das variáveis. Por exemplo: ConstanteSolar (no lugar de) constante_solar

- brancos dentro do nome de variáveis não é permitido

Como os nomes podem ter até 31 caracteres, ele devem ser significativos e descritivos, isto é, devem facilitar a identificação da variável ou constante e ao mesmo tempo facilitar o entendimento do seu uso e mesmo do programa.

Por exemplo: **massa_atomica** no lugar de **ma** para expressar a massa atômica

Exemplos de nomes:

nomes corretos:	massa	
	media_anual	
	tempo_decaimento2	
	angulo	
nomes incorretos:	1_massa	porque inicia com número
	_comprimento	inicia com uma sublinha
	tempo_decaimento-total	usa hífen no nome

21 – Declaração de Variável

A declaração de tipo é usada para especificar o tipo e, simultaneamente, declarar uma variável. Não esquecer que ao se declarar o tipo de uma variável está-se definindo o nome da variável, e o mais importante, definindo-se como ela será representada internamente no computador e quais regras e operações são possíveis de serem realizadas com elas.

As regras, operações e os atributos são imediatamente associados às variáveis e constantes por default, sem nenhuma interferência do programador. O Fortran disponibiliza meios de alterar algumas das características e/ou atributos das variáveis.

A declaração de tipo (e também declaração de variável) tem a seguinte forma:

type-specifier :: lista

onde type-specifier tem que ser um dos tipos abaixo

- **integer** : variável ou constante pode conter números inteiros
- **real** : variável ou constante com números real
- **complex** : variável ou constante com número complexo
- **logical** : variável ou constante com valor lógico (true ou false)
- **character** : variável ou constante com caracteres

Na declaração de tipo, **lista** especifica uma variável ou várias variáveis separadas por vírgula.

exemplos:

```
integer :: media                    ! variável inteira media
integer :: media, media_anual, erro ! 3 variáveis inteiras: media, media_anual e erro
real    :: massa_atomica, raio      ! 2 reais: massa_atomica e raio
complex :: z1, impedancia           ! 2 variáveis complexas, z1 e impedancia
logical :: estado                   ! variável lógica estado
character(len=2) :: sigla_estado    ! variável caractere sigla_estado com comprimento 2
```

- **comprimento de uma variável CARACTERE**

No Fortran, uma variável ou constante **caractere** tem um comprimento fixo. Este comprimento deve ser especificado quando se declara a variável. A palavra-chave que especifica o comprimento da variável ou constante caractere é **len**. Ela deve ser usada seguida do sinal de igual e do comprimento desejado, tudo entre parênteses. O Fortran 95 especifica que se o comprimento (len) for omitido ele será feito unitário (len=1, por default).

Exemplos:

```
character(len=15):: nome, nome_familia  ! 2 variáveis tendo cada uma
                                        ! um comprimento de 15 caracteres
character(len=1):: nome_meio     ! variável nome_meio com comprimento unitário
```

- Pode-se declarar variáveis de vários comprimentos como se segue.

```
character(len=15):: cidade, distrito
character(len=20):: estado
character(len=2) :: sigla_estado
```

- A declaração `character(len=*)` refere-se a um comprimento que será determinado posteriormente. Assume que o comprimento será obtido quando se entra com o valor na variável.

```
character(len=*):: profissao  !terá comprimento igual a 6 se o valor "medico" for usado
                              !ou comprimento igual a 10 se contiver "enfermeiro"
```

22 – Variáveis e Constantes

Variáveis e constantes são usadas para guardar os dados necessários aos nossos programas que serão usados e manipulados durante a execução do programa.

Uma **variável** contem um **valor** (dado) que **pode ser alterado** durante a execução do programa, enquanto uma **constante** contém valores que **não serão alterado** durante a execução do programa.

A declaração de atribuição é usada para entrar (colocar) um valor ou dado tanto nas variáveis quanto nas constantes.

Variável

Conforme dito, a declaração de uma variável é feita quando se especifica o seu tipo.

O Fortran é uma linguagem tipificada, isto é, ele sempre verifica o tipo da variável ou da constante antes de trabalhar com ela. Faz isto para poder ler corretamente o dado e para usar as operações adequadas àquela variável ou constante.

Constante

Uma constante que não possua um nome é chamada de constante literal, que usualmente é tratada apenas pelo termo constante. Uma constante literal depois de definida passa a ser um totem usado para representar um certo dado (valor) em particular.

As constante literal só podem ser de algum tipo intrínseco, isto é, de um dos tipos que o Fortran aceita.

O Fortran utiliza 5 (cinco) tipos intrínsecos de constantes:

- Inteira
- Real
- Complexa
- Lógica
- Caractere

- **Constante Inteira**

É formada por um conjunto de dígitos. Sem ponto decimal. O sinal é opcional.

exemplos corretos: 3
-567
+1
0

exemplos incorretos: -3.2 ! não pode conter ponto decimal
+123,6 ! não pode conter vírgula
--12 ! muitos sinais

- ## Constante Real

A constante real, possui duas formas de ser representada: decimal e exponencial.

- **Decimal**
  é obrigatório o uso do ponto decimal (use o zero se necessário). Não pode conter vírgula e o sinal é opcional

exemplos corretos:
```
73.8
-567.0   ! bom estilo de programação usa zero
+1.0     ! depois do ponto decimal
0.0
```

exemplos incorretos:
```
-32      ! deve conter ponto decimal
+123,6   ! não pode conter vírgula
--12.0   ! muitos sinais
```

- **Exponencial**
  Especifica o valor da mantissa, seguido do símbolo "E" ou "e" (de expoente), seguido do valor numérico do expoente (número inteiro). Sinal é opcional

exemplos corretos:

29.8E3		29.8×10^{3}
-5.0e-2	! bom estilo de programação usa zero	-5.0×10^{-2}
+1.0E0	! depois do ponto decimal	1×10^{0}
0.3e2		0.3×10^{2}

exemplos incorretos:
```
-32E5       ! escreva -32.0E5
+123,6 E8   ! errado: não pode conter vírgula
-12.0-3     ! errado: falta o símbolo E
```

- ## Constante Complexa

Uma constante complexa é formada por um par de números reais ou inteiros, separados por vírgula, colocado entre parêntese. Segue as mesmas regras para as constantes inteiras e reais

exemplos corretos:

(3.0, -2.8)	3.0 + **i** 2.8
(1.0, 34E-3)	1.0 + **i** 0.034
(2.0, 6.0)	2.0 + **i** 6.0

exemplos incorretos: (-3,2, 5.7) ! número real não pode conter vírgula

- ## Constante Lógica

A constante lógica no Fortran pode assumir somente dois valores: " **.true.** " e " **.false.** ". Observe que os valores (true e false) têm que estar entre pontos

exemplos corretos:
```
.TRUE.
.FALSE.
.true.
.false.
```

exemplos incorretos:
```
TRUE     ! faltam os pontos
.true    ! falta um ponto
.false,  ! uso de vírgula
```

- ## Constante Caractere

O conteúdo de uma constante caractere deve estar colocada entre aspas (") ou apóstrofo (').

Os caracteres que pode ser usado numa constante caractere são aqueles que fazem parte do conjunto de caracteres do Fortran.

Dentro de uma constante caractere pode-se usar espaço em branco a vontade. Eles serão mantido onde colocados, da forma que foram colocados.

A constante pode também conter o caractere aspas ou o caractere apóstrofo, mas neste caso a constante caractere deve ser envolvido pelo outro caractere.

Exemplo:

' raio do "circulo" = ' ou então **" raio do 'circulo' = "**

" 12'3/4 " ou então **' 12''3/4 '** o uso de dois apóstrofos seguidos conta como um e pode ser usado, entretanto é recomendado escrever **" 12'3/4 "**

O comprimento da constante caractere é determinado pelo número de caracteres que ela contém. Zero para uma constante caractere vazia. Observe que **''** (dois apóstrofos seguidos) ou **""** (duas aspas seguidas) representa uma constante caractere vazia, portanto possuí comprimento zero

Exemplos de constantes caractere:

exemplos corretos:

```
"paulo"                 ! comprimento 5
"paulo  "               ! comprimento 7
" oh! sera possivel? "  ! uso de ! dentro string; comprimento 20
" nao continua &"       ! uso de símbolo de continuação; comprimento 15
```

exemplos incorretos:

```
'entre com o raio       ! falta apóstrofo
entre com o angulo"     ! falta aspas
'comprimento"           ! uso de apóstrofo e aspa ao mesmo tempo
```

23 – Constante com Nome

Uma constante com nome (named constant) é um valor constante que pode ser identificado por um nome, daí a sua denominação: constante com nome. A grande vantagem do uso da constante com nome é a possibilidade de se definir o "conteúdo" da constante em um único lugar do programa e usá-lo em qualquer lugar do programa.

Caso seja necessário atribuir outro valor à constante será necessário alterar o valor (na sua definição) e realizar novamente a compilação do programa. O novo valor será usado pelo programa todo.

Uma questão que sempre aparece é:

Por que usar uma constante com nome no lugar de uma variável contendo o valor constante?

Porque uma variável utilizada como constante poderia ser inadvertidamente alterada durante a execução do programa enquanto a constante com nome não pode. Esta é uma grande vantagem. Uma vantagem muito significativa!

A declaração de uma constante com nome faz uso do atributo **parameter** usado junto com a declaração de definição de tipo da constante seguido do nome da constante com seu valor ou conteúdo.

TYPE NAME , parameter :: List of names

Exemplos:

```
real, parameter:: planck = 6.626075E-34   !define a constante de Planck 6.626075x10^-34
real, parameter :: pi=3,14159, avogadro = 6.023E23
```

Depois que um valor é guardado numa constante com nome, ele pode ser usada no programa escrevendo-se o nome utilizado na definição da constante.

Exemplo:

```
real:: massa, n_moles
real,parameter:: massa_molecular_gas = 32.0       ! linha 2

    print*, " Entre com a massa do gas "
    read*, massa
    n_moles = massa / massa_molecular_gas
    ................
    ................
```

observe que se o gás for trocado (32 é o oxigênio) basta altera o valor utilizado na linha 2 pelo novo valor.

Muito melhor que escrever:

```
n_moles = massa / 32.0
```

porque isto implicaria em ter que trocar o valor 32 em todas as linhas do programa onde aparece a constante literal 32.0.

Outra vantagem: o valor da constante não poderá ser trocada mesmo durante a execução do programa.

Então:

```
real:: massa_molecular
massa_molecular = 32.0
................
................  não é tão seguro quanto a constante com nome.
```

Pode-se definir constantes com nome com todos os tipos usado pelo Fortran.

```
Inteira:     integer, parameter :: limite = 30, n_maximo = 100
Real:        real, parameter    :: pi = 3.14159
Complexa:    complex, parameter :: fase_inicial = (0.0, 3.4)
Lógica:      logical, parameter :: chave1 = .TRUE. , off = .FALSE.
Caractere:   character(len=3), parameter :: chefe = "Rui"
```

Nota Importante: Constante com nome ***não pode aparecer no lado esquerdo de uma expressão***

```
..............
real:: m = 39.7
real, parameter:: gravidade = 9.8065
..............
..............
gravidade = m * 37.0 /3.7     ! ERRADO: constante com nome não pode
..............                ! ter seu valor alterado
..............
m = 190.90                    ! observe que m é uma constante literal
..............                ! e portanto será trocada, sem problema
..............
```

24 – Inicializando uma Variável

Uma variável pode ser vista como sendo uma caixa onde se pode guardar um certo conteúdo (dado). É muito importante ter sempre em mente que a caixa (a variável) não pode guardar qualquer tipo de dado (qualquer tipo de conteúdo) e mais: que toda caixa terá um nome que permitirá sua localização e manuseio pelo programa.

Quando se cria uma variável o computador na realidade pega (reserva) uma memória disponível (que aqui está sendo chamado de caixa). A memória reservada não é "limpa", isto é, seu conteúdo não é apagado. Portanto a posição de memória que for reservada para ser usada pode ou não estar vazia. Usualmente não estará vazia.

Lembre-se, quando uma variável é criada, o que se faz é separar e reservar um segmento (pedaço) da memória, nada é colocado (escrito) nela. Portanto antes de usar uma variável é recomendável que ela receba um valor conhecido, para substituir o valor aleatório contém. Não assuma que o compilador fará isto por você. Ele não fará!

Usar uma variável não inicializada pode causar resultados inesperados.

Para atribuir um valor à uma variável pode-se utilizar um dos seguinte métodos:

- inicializar a variável quando definir a variável (quando o programa for executado)
- ler o valor de um arquivo ou fornecer o valor pelo teclado
- usar a declaração de atribuição

- **Inicializando uma variável**

A inicialização de uma variável é feita quando o computador carrega o programa para a memória principal, no processo de preparação para a sua execução. A inicialização é portanto feita antes do programa iniciar.

A inicialização é feita quando se define a variável

Por exemplo:

```
real:: pi = 3.14159       ! coloque o sinal de igual seguido do valor desejado ou
                          ! uma expressão contendo constante ou constante com nome
integer:: bolas=10
character(len=3)::estado="off"
logical:: troca=.true.
complex:: fase=(0.0,3.4)
```

- **usando uma expressão**

```
real, parameter:: g=9.81, m=75
real           :: força_gravitacional=g * m
```

- **lendo um valor de um arquivo**

```
program calcula_integral
..............
..............
real    :: valor_x1, valor_x2, valor_x3
integer ::n                                  ! número de dados lidos
..............
..............
..............
read(unit=10,fmt=*) n, valor_x1, &
                       valor_x2, &
                       valor_x3     !aqui é atribuído o valor das variáveis que
                                    !até então estavam indefinidas
..............
..............
end program calcula_integral
```

- **valor fornecido pelo teclado**

```
..............
..............
real   :: valor_x1, valor_x2, valor_x3
integer:: n                               ! número de dados lidos
..............
..............
read(unit=*,fmt=*) "escreva: n, valor_x1, valor_x2, valor_x3", &
                             n, valor_x1, valor_x2, valor_x3
..............
..............
```

- **declaração de atribuição**

```
..............
..............
real   :: valor_x1, valor_x2, valor_x3
integer:: n
```

```
..............
..............
valor_x1 = 234.5
valor_x2 = 195.0
valor_x3 = 0.32
n = 3
..............
..............
```

25 – Declaração de Atribuição

A declaração de atribuição tem a seguinte forma:

variável = expressão

A declaração de atribuição utiliza o " sinal de igual " (operador de atribuição no Fortran). O sinal de igual deve ser entendido de maneira especial. A melhor forma de entender a declaração de atribuição é:

```
variável ← expressão
```

O que o operador de atribuição faz é escrever na variável ou constante o dado escrito a sua direita, por isto o uso da seta.

Lembrete: a variável é a caixa, então o operador de atribuição coloca dentro da caixa algo. Coloca-se (escreve-se ou atribui-se) na memória (variável ou caixa) o dado (conteúdo ou valor)

```
esquerda ←  direita
variável ← dado
variável ← valor
variável ←  expressão
constante ← valor
```

Quando se usa uma expressão com o operador de atribuição (variável ← expressão), a **expressão é avaliada e o resultado escrito na variável.**

As expressões utilizadas podem ser de tipo: não mistas e mistas.

- **Expressão não mistas (ou de um tipo único)**

  Neste caso a expressão primeiro é avaliada e depois escrita na variável a esquerda que deve ser do mesmo tipo usado na expressão.

  Exemplos:

  Tipo inteiro:

```
integer:: total, soma=30, n=2
..............
..............
total = (soma/5)**n   ! todos os valores são inteiros
..............
```

  Tipo real, tipo complexo e tipo caractere:

```
real            :: area = 0.0, x=10.67, y=5.9e2
complex         :: impedancia=(0.0,0.0), z1=(2.4,5.0), z2=(65.1,4.2)
character(len=1):: ch1="D", ch2="D"
```

```
character(len=2):: estado=" LI"
.............
.............
area = (y – x)*(156.8 + x*2 - (y / x))
.............
.............
impedancia = z1 - z2
.............
.............
estado = ch1//ch2      ! variável estado será "DD"
.............
.............
```

- **Expressão mistas (variáveis com tipo diferentes)**

```
variável = expressão mista
```

Qualquer que seja o caso a expressão primeiro é avaliada e depois escrita na variável a esquerda mas, neste caso o resultado é convertido para o tipo à esquerda do sinal de atribuição (regra de coerção).

Exemplos:

- **Variável do tipo Inteira**

no caso da expressão ser do tipo real e a variável do tipo inteira

a expressão é avaliada e quando for escrita perderá a parte decimal porque um inteiro não pode guarda a parte decimal (fração) do número.

```
integer:: temperatura = 0
real   :: sonda, carga = 27.0
.............
.............
sonda = 10.0
temperatura = carga / sonda   ! temperatura conterá o número 2 e não 2.7
.............
.............
```

- **Variável do tipo Real**

```
.............
.............
real   :: area
integer:: l = 5, d = 2
.............
.............
area = l * d       ! resposta 10.000000, não o inteiro 10
.............
.............
```

26 – Operadores Aritméticos

O Fortran tem quatro tipos de operadores: aritméticos, relacionais, lógicos e caractere.

Uma expressão é uma combinação de operadores, operandos e parênteses arranjados de forma conveniente.

Operadores Aritméticos

Operador	Nome	Prioridade	Associação
**	exponenciação	1	direita para esquerda
* /	multiplicação divisão	2	esquerda para direita
+ -	adição subtração	3	esquerda para direita

Os operadores de mesma prioridade são avaliados da esquerda para a direita.

Observe a regra de associação da exponenciação !

Os operandos de uma expressão são os fatores a serem usados no cálculo.

Regras básicas:

- Os parênteses são utilizados para agrupar termos. Seu uso no Fortran é semelhante ao da matemática com a qual estamos acostumados

  x = a + (b*x) + (c*x**2) + (d*x**3)

- Os parêntese serão utilizados para alterar a prioridade e a forma de associação definida na tabela acima.

  x = a + b*x + c*x**2 a expressão é avaliada:
  1 => a exponenciação
  2 => as multiplicações da esquerda para a direita
  3 => as adições da esquerda para a direita

  x = (a + b*x) + c*x**2 a expressão é avaliada:
  1 => a multiplicação dentro do parênteses
  2 => a adição dentro do parênteses
  3 => a exponenciação
  4 => a multiplicação
  5 => a adição

- Dois operadores aritméticos não podem aparecer juntos

  x = 4*-2 **deve ser escrito** x = 4 * (-2)

  use os parênteses para separa os operadores
- Os operadores são absolutamente necessários

  x = a + bx + cx**2 + dx**3 **errado**

  x = a + b*x + c*x**2 + d*x**3 **expressão correta**

27 – Operadores Relacionais

A forma geral de uma expressão envolvendo operadores relacionais é:

operando **operador** ***operando***

onde os operandos podem ser expressões aritméticas ou strings.

Existem 6 (seis) operadores relacionais no Fortran.

Operadores Relacionais

	Fortran 95	FORTRAN 77
<	**menor que**	.lt.
<=	**menor ou igual a**	.le.
>	**maior que**	.gt.
>=	**maior ou igual a**	.ge.
= =	**igual a**	.eq.
/=	**diferente de**	.ne.

A direita foi escrito a forma deste operadores usada pelo FORTRAN 77. Veja como o Fortran 95 melhorou muito a maneira de representar estes operadores.

Entre os operadores relacionais não há prioridades. Todos tem a mesma prioridade.

Entretanto os operadores aritméticos tem prioridade maior que os relacionais e o operador concatenação (//) tem prioridade maior que todos.

- ## Comparando expressões aritméticas

  - ***A resposta da comparação é sempre um valor lógico***

    4 + 6 >= 11 terá como resposta o valor **.false.**

  - ***Expressões aritméticas de modo misto, os operandos inteiros são convertidos para tipo real***

    7.2 + 3.0 < 7 +1 será avaliada como

    7.2 + 3.0 < 7.0 + 1.0 10.2 < 10.0 Resposta: **.false.**

  - ***Todos os operadores relacionais tem prioridade menor que os operadores aritméticos***

    isto significa que as expressões aritméticas serão sempre avaliadas primeiro, antes de serem usados os operadores relacionais

    sendo x = 3
    y = 30
    m = 10
    k = 20
    k * x - x**2 >= k * m + y / x

    --> 20*3 - 3**2 >= 20*10 + 30/3
    --> 20*3 - (3**2) >= 20*10 + 30/3
    --> (20*3) - 9 >= 20*10 + 30/3
    --> 60 - 9 >= (20*10) + 30/3
    --> 60 - 9 >= 200 + (30/3)
    --> (60 - 9) >= 200 + 10
    --> 51 >= (200 + 10)
    --> 51 >= 210
    --> **.false.**

    Lembre-se: a melhor forma de se escrever as expressões é usando parênteses, pois assim fica bastante claro a ordem das operações

    (k * x - (x**2)) >= (k * m + y / x)

- ## Comparando cadeias de caracteres (strings)

Computadores usam número binários para representar internamente tanto texto quanto números. Entre os fabricantes de computadores dois conjuntos de códigos (tabelas) são usados para especificar a relação entre os números binários (interno ao computador) e o conjunto de caracteres alfanuméricos que usualmente utilizamos.

As tabelas mais usadas são:

- **EBCDIC**
  EBCDIC - **E**xtended **B**inary **C**oded **D**ecimal **I**nterchange **C**ode
  Criada e usada pela IBM em todos os seus computadores, exceto nos PC

- **ASCII**
  ASCII – **A**merican National **S**tandard **C**ode for **I**nformation **I**nterchange
  Usada pelos outros fabricantes e posteriormente pela IBM também

O Fortran faz uso da tabela ascii como tabela padrão.

Mas qualquer que seja a tabela usada, os caracteres e símbolos estarão sempre arranjados em uma ordem numérica na tabela.

Observe os números decimais que identificam a ordem (posição) em que o caractere ou símbolo está na tabela ascii.

Tabela ASCII

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	032	(space)	064	@	096	
001	☺	033	!	065	A	097	a
002	●	034	"	066	B	098	b
003	♥	035	#	067	C	099	c
004	♦	036	$	068	D	100	d
005	♣	037	%	069	E	101	e
006	♠	038	&	070	F	102	f
007	(Beep)	039	'	071	G	103	g
008	■	040	(	072	H	104	h
009	(Tab)	041	)	073	I	105	i
010	(Line feed)	042	*	074	J	106	j
011	(Home)	043	+	075	K	107	k
012	(Form feed)	044	,	076	L	108	l
013	(Carriage return)	045	-	077	M	109	m
014	♫	046	.	078	N	110	n
015	☼	047	/	079	O	111	o
016	►	048	0	080	P	112	p
017	◄	049	1	081	Q	113	q
018	↕	050	2	082	R	114	r
019	!!	051	3	083	S	115	s
020	¶	052	4	084	T	116	t
021	§	053	5	085	U	117	u
022	▬	054	6	086	V	118	v
023	↨	055	7	087	W	119	w
024	↑	056	8	088	X	120	x
025	↓	057	9	089	Y	121	y
026	→	058	:	090	Z	122	z
027	←	059	;	091	[	123	{
028	(Cursor right)	060	<	092	\	124	;
029	(Cursor left)	061	=	093	]	125	}
030	(Cursor up)	062	>	094	^	126	~
031	(Cursor down)	063	?	095	_	127	⌂

Mesmo que o computador com o qual esteja trabalhando utilize outra tabela que não a ascii (caso pouco provável), ele obedecerá o seguinte critério:

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

A < B < C < D < E < F < G < H < I < J < K < L < M < N < O < P < Q < R < S < T < U < V < W < X < Y < Z
a < b < c < d < e < f < g < h < i < j < k < l < m < n < o < p < q < r < s < t < u < v < w < x < y < z

A comparação de caracteres segue a seqüência definida na tabela ascii. A posição (número) é que será usado para estabelecer o critério de comparação.

A expressão ("A" < "B") pode ser entendida como (65 < 66) o que obviamente terá como resposta **.true.**

Escrevendo-se ("A" = = "a") terá como resposta **.false.**, por quê?

Porque teremos (65 = = 97), que obviamente é falso. Observe a tabela ascii.

Comparando strings

A comparação de uma cadeia de caracteres (string) fica bastante simplificada quando se usa a tabela ascii.

Apesar de pouco provável, não admita que o seu computador utiliza a tabela ascii como padrão, sem testar.

As regras básicas para a comparação de strings são:

- a comparação sempre inicia no primeiro caractere (primeiro a esquerda) do string e segue até o último
- se os dois caracteres são **iguais** segue para comparar o seguinte
- se **diferentes**, o string contendo o menor caractere é considerado o menor
- encontrado um diferente a caracterização é feita e o processo para
- se for percorrido todo o string, eles serão considerados iguais

Exemplos

- Compare "abcdef" e "abcefg"

```
a   B   c   d   e   f
=   =   =   <
a   B   c   e   f   g
```

Os 3 primeiros são iguais nos dois strings. O quarto caractere de cada string é diferente. Neste caso "**d** " (do primeiro) é menor que "**e**" (do segundo), por isto temos: ("abcdef" < "abcefg") resposta: .true.

- Compare "01357" e "013579"

```
0   1   3   5   7
=   =   =   =   =   <
0   1   3   5   7   9
```

Observe que o primeiro string por ser menor é completado com "espaço em branco" até ter o mesmo tamanho que o segundo e então comparado. Como "branco" é menor que "9" então: ("01357" < "013579") resposta: .true.

- Compare "Ana" com "Rui"

```
A  n  a
<
R  u  i
```

O primeiro caractere já determina a resposta: ("Ana" < "Rui") terá como resposta: .true.

Prioridade com o operador concatenação

O operador concatenação (//) tem prioridade maior que os operadores relacionais.

Então todos as concatenações serão feitas antes da avaliação.

"Luiz" // " Silva" < "Luiz" // (" da" // " Silva")

--> ["Luiz" // " Silva"] < ("Luiz" // ((" da" // " Silva")))
--> "Luiz Silva" < ("Luiz" // [" da" // " Silva"])
--> "Luiz Silva" < ["Luiz" // " da Silva"]
--> "Luiz Silva" < "Luiz da Silva"
--> **.false.**

28 – Operadores Lógicos

O Fortran possui 5 operadores lógicos:

- **Operadores Lógicos**

Operador	Nome	Prioridade	Associação
.not.	**não**	1	direita para esquerda
.and.	**e**	2	esquerda para direita
.or.	**ou**	3	esquerda para direita
.eqv.	**equivalente**	4	esquerda para direita
.neqv.	**não equivalente**	4	esquerda para direita

Numa expressão os operadores de mesma prioridade são avaliados da esquerda para a direita.

Prioridade entre operadores

Prioridades:
1 operadores aritméticos
2 operadores relacionais
3 operadores lógicos

- **Tabelas verdades**:

A avaliação de uma expressão lógica típica: ***operando operador operando***

NOT

O operador lógico NOT é um operador unário , i.e, um operador que é usado assim: **operador operando**

operador	operando	resultado
.not.	.true.	.false.
.not.	.false.	.true.

AND

Observe que o operador lógico AND é usado assim: **operando operador operando resultado**

operando	operador	operando	resultado
.true.	.and.	.true.	.true.
.true.	.and.	.false.	.false.
.false.	.and.	.true.	.false.
.false.	.and.	.false.	.false.

Então a resposta é verdade quando ambos os operadores forem verdadeiros.

OR

Observe que o operador lógico OR é usado assim: **operando operador operando resultado**

operando	operador	operando	resultado
.true.	.or.	.true.	.true.
.true.	.or.	.false.	.true.
.false.	.or.	.true.	.true.
.false.	.or.	.false.	.false.

Então se um dos operandos é verdadeiro a resposta é verdadeira.

EQV

Observe que o operador lógico EQV é usado assim: **operando operador operando resultado**

operando	operador	operando	resultado
.true.	.eqv.	.true.	.true.
.true.	.eqv.	.false.	.false.
.false.	.eqv.	.true.	.false.
.false.	.eqv.	.false.	.true.

Então a resposta é verdadeira somente se ambos os operadores forem iguais.

NEQV

Observe que o operador lógico NEQV é usado assim: **operando operador operando resultado**

operando	operador	operando	resultado
.true.	**.neqv.**	**.true.**	**.false.**
.true.	**.neqv.**	**.false.**	**.true.**
.false.	**.neqv.**	**.true.**	**.true.**
.false.	**.neqv.**	**.false.**	**.false.**

Então a resposta é verdadeira somente se ambos os operadores forem diferentes.

- **As regras básicas para os operadores lógicos**
  - Operadores relacionais não podem comparar valores lógicos
  - Use o operador **.eqv.** (**não** o = =) para **verificar a igualdade** entre operandos lógicos
  - Use operador **.neqv.** (**não** o /=) para **verificar a desigualdade** entre operandos lógicos

- **Atribuição de valores lógicos**

  O resultado de uma expressão lógica pode ser colocada em uma variável lógica.

  Por exemplo:

```
.............
.............
logical:: Resultado1, Resultado2
logical:: ligado=.true., desligado=.false.
.............
.............
Resultado1 = .not. ligado .and. desligado           ! resultado1 será .false.
Resultado2 = (.not. desligado) .or. (.not. ligado)  ! resultado2 será .true.
.............
.............
```

29 – Operador Caractere

Strings podem ser unidos (concatenados) em um único string. Esta operação é conhecida como concatenação.

A concatenação de strings é realizada com o operador //. Fortran só possui este operador para trabalhar com variáveis e constantes do tipo caractere.

Por exemplo:

```
character(len=3):: dia = "10 "
character(len=5):: mes="abril"
character(len=8):: dm
.....................
.....................
dm = dia // mês
.....................
.....................
```

A variável dm conterá o dado "10 abril"

O operador caractere não pode ser usado com operadores aritméticos.

Dado dois strings (a e b) de comprimentos m e n a concatenação destes dois strings (a // b) gera um string com um tamanho de m+n.

Exemplo:

observação: para clareza do texto foi usado o símbolo " ¯ ". Ele está sendo usado somente para dar destaque, isto é, tornar bastante evidente os espaços em branco.

```
character(len=5) :: nome = "Paulo", nome_2= "Jose¯"
character(len=8) :: nome_meio="Buscate¯", sobrenome="Filho¯ ¯ ¯"
character(len=13):: c_nome1
character(len=21):: c_nome2
.......................
.......................
c_nome1 = nome // nome_meio
c_nome2 = nome_2  // nome_meio //sobrenome
.......................
.......................
```

A variável c_nome1 conterá o seguinte:

```
c_nome1 → "PauloBuscate¯"              ( 5 + 8 = 13 )
```

A variável c_nome2, mostrada com mais destaque:

```
c_nome2   →  " J o s e ¯ B u s c a t e ¯ F i l h o ¯ ¯ ¯ "   (5+8+8 = 21)
             1 2 3 4 5 6 7 8 9 1 1 1 1 1 1 1 1 1 1 2 2
                               0 1 2 3 4 5 6 7 8 9 0 1
```

30 – Expressões Aritméticas – Modo Simples

Uma expressão aritmética [aquelas que só usam operações de: adição (+), subtração (-), multiplicação (*), divisão (/) e exponenciação (**)] pode ser escrita em 4 modos: modo inteiro, modo real, modo complexo e modo misto.

No modo inteiro todas as variáveis, funções e constantes referenciadas são do tipo inteiro. No modo real todos os operandos das expressões são do tipo real, enquanto o modo complexo só tem variáveis e constantes do tipo complexo.

O modo misto é obtido quando se usa operandos de mais de um tipo.

No modo simples (real, inteiro ou complexo), o resultado da operação é idêntico ao dos operandos, então:

operando	operador	operando	resultado
inteiro	**op**	**inteiro**	**inteiro**
real	**op**	**real**	**real**
complexo	**op**	**complexo**	**complexo**

Exemplos:

Expressão	modo	valor calculado	
18 + 6	inteiro	24	inteiro
2.0 + 3.2	real	5.2	real
3 / 2	inteiro	1	inteiro
0.3 + 1.4	complexo	1.7	complexo
-5**2	inteiro	25	inteiro

- **Regras para avaliação de expressões**
  - As expressões são sempre avaliadas da esquerda para a direita
  - As operações são efetuadas iniciando-se com a de maior prioridade seguindo-se os outros na ordem decrescente

tipo	**prioridade**
Cálculo de funções	1
exponenciação	2
multiplicação e divisão	3
adição e subtração	4

- Dois operadores com mesma prioridade serão avaliados segundo a regra de associatividade

operadores	**associação**
exponenciação	direita para esquerda
multiplicação e divisão	esquerda para direita
adição e subtração	esquerda para direita
multiplicação e divisão	esquerda para direita

- Use parênteses para alterar ou tornar explicita as operações desejada

Exemplos:

avalie a expressão 5 * 4 * 3 / 2**2

```
5 * 4 * 3 / 2 ** 2
→ 5 * 4 * 3 / (2 ** 2)
→ (5 * 4) * 3 /4
→ (20 *3) / 4
→ 60 / 4
→ 15                    ! de preferência a (5*4*3)/(2**2)
```

A ordem pode ser alterada usando-se parêntese

```
5.0 * 4.0 * (3.0 / 2.0) ** 2.0
→ 5.0 * 4.0 * (.5 ** 2.0)
→ (.0 * 4.0)* 2.25
→ 20.0 * 2.25
→ 45.0
```

Observe ainda:

a**b**c → a**(b**c) mesma prioridade
regra associatividade é da direita para a esquerda

(a**b)**c → alterada a ordem

a*b/c → (a*b)/c mesma prioridade
regra associatividade é esquerda para direita

a*(b/c) → alterada a ordem

ATENÇÃO: Observe as regras com respeito a **Exponenciação**

- a**b **com a e b inteiro será uma variável inteira**

  2**(1/2) → 1 porque 1 / 2 = 0,5 será convertido para 0

  2**(-3) → 0 porque é 1 / (2**3) = 1 / 9 = 0.111 = 0

  2**3 → 8

- a**b **com a e b real será uma variável real**

  3.0**(1.0 / 2.0) → 1.732

  2.0**(- 3.0) → 0.111

  2.0**3.0 → 8.0

  (-3)**(1.0 / 3.0) → **um número negativo elevado a um número real não pode ser calculado**

31 – Expressões Aritméticas – Modo Misto

As expressões que contém operandos numéricos de diferentes tipos são ditas expressões modo misto.

Nas expressões mistas, exceto quando eleva um valor real ou complexo a uma potência inteira, o objeto mais simples (ou fraco) dos dois tipos de dados será convertido para o tipo mais forte.

Tabela com o tipo obtido na avaliação da expressão **a .op. b** onde **.op.** é um dos operadores**: +, -, *** ou **/**

tipo de **a**	tipo de **b**	**tipo do resultado**	**exemplo**
inteiro	**inteiro**	**inteiro**	**2 * 4 = 8**
inteiro	**real**	**real**	**2 * 4.0 = 8.0**
inteiro	**complexo**	**complexo**	
real	**inteiro**	**real**	**2.0 * 4 = 8.0**
real	**real**	**real**	**2.0 * 4.0 = 8.0**
real	**complexo**	**complexo**	
complexo	**inteiro**	**complexo**	
complexo	**real**	**complexo**	
complexo	**complexo**	**complexo**	

Tabela com o tipo obtido na avaliação da expressão **a**b**

tipo de **a**	tipo de **b**	**tipo do resultado**	**exemplo**
inteiro	**inteiro**	**inteiro**	**2**3 = 8**
inteiro	**real**	**real**	**2**3.0 = 8.0**
inteiro	**complexo**	**complexo**	
real	**inteiro**	**real**	**2.0**3 = 8.0**
real	**real**	**real**	**2.0**3.0 = 8.0**
real	**complexo**	**complexo**	
complexo	**inteiro**	**complexo**	
complexo	**real**	**complexo**	
complexo	**complexo**	**complexo**	

Regras

- se os operandos são do mesmo tipo, calcule o resultado do operador
- se diferentes converta o mais fraco no tipo do mais forte e calcule o resultado do operador
- **COERÇÃO**: uma expressão de qualquer tipo pode ser escrita numa declaração de atribuição. Fortran automaticamente (lei da coerção) converte a resposta para o tipo da variável que vai receber o resultado

variável real = expressão inteira → transforma valor inteiro em real
variável inteira = expressão real → transforma valor real em inteiro

```
real    :: resultado
integer:: centroide
..............
..............
resultado = (5*4)/3             ! resultado = 6.0  (cálculo = 6.666)
centroide = 31.0/8.0            ! centróide = 3    (cálculo = 3.875)
```

Exceção: a**b sendo **a** do tipo real e **b** do tipo inteiro. Neste caso o cálculo é feito multiplicando-se cópias de a: 1.5**3 = 1.5 * 1.5 * 1.5 (resposta = 3.375)

Exemplos:

4.7 + 3 = 7.5
1 / 8 = 0
2.0 / 8 = 0.25
-3**2.0 = -9.0
4.0**(1 / 2) = 1.0 → 4.0**0 = 1.0 (divisão inteira 1 / 2 = 0.5 → 0)
((31.0 - 20) / 3) * 9) + 85 = 117.994
→ ((11.0 / 3) * 9) + 85
→ (3.666 * 9) + 85
→ 32.994 + 85
→ 117.994

32 – Funções Intrínsecas

O Fortran possui um conjunto de operações mais elaboradas que as operações aritméticas fundamentais (+, -, *, /). São as funções intrínseca.

Por exemplo: a = sqrt(b) a função raiz quadrada apropriada será utilizada conforme o tipo da variável b seja real ou complexo.

As funções intrínsecas são fornecidas com o compilador pelo fabricante. Elas são pré-compiladas. Elas serão automaticamente incorporadas no programa objeto durante a etapa chamada de link edição. Aonde o programa **ld** (loader) encontrar uma referência a uma função intrínseca ele insere os códigos da função no programa executável.

As regras de utilização da funções são similares as da utilização de nomes das variáveis. A grande diferença é que os nomes das funções são seguidos pelos argumentos, colocados entre parênteses.

Então, para se trabalhar com funções é necessário conhecer:

- os nomes das funções
- tipo de valor retornado pela função
- número de argumentos utilizados
- tipos dos argumentos
- faixa em que os argumentos são válidos

Por exemplo: x = sqrt(16) retorna o valor inteiro 4;

x = sqrt(16.0) retorna o valor real 4.0 e

x = sqrt(-9) retorna erro, porque o valor do argumento é negativo e não é válido para esta função.

Todos estes fatores estão especificados na definição das funções.

Alguns exemplos de funções:

Funções	Significado	Faixa	Tipo Arg.	Tipo da função
abs(x)	valor absoluto de x		inteiro	inteiro
sqrt(x)	raiz quadrada de x	x >= 0.0	real	real
sin(x)	seno de x	0.0< x <1.0	real	real
exp(x)	exponencial de x	x > 0.0	real	real
log(x)	log (base e) de x	x > 0.0	real	real
len(x)	comprimento string		caractere	inteiro
int(x)	converte p/inteiro		real	inteiro
real(x)	converte p/real		inteiro	real
achar(i)	retorna o caractere	i entre 1 e 127	inteiro	caractere
fraction(x)	parte fracional de x		real	real

Prioridade

- As funções tem a maior prioridade (exceção: operadores unários)

tipo	prioridade
funções	1
operadores aritméticos	2
operadores relacionais	3
operadores lógicos	4

- Os argumentos de uma função podem ser expressões. Neste caso as expressões serão avaliadas primeiro e então passadas para a função

Exemplo:

```
.............
.............
real  ::  delta, a = 1.0, b = -3.0, c = -18.0
.............
.............
delta = (-b + sqrt(b**2 - 4.0*a*c))/(2.0*a)
.............
.............
```

então

```
(-b + sqrt(b**2 - 4.0*a*c))/(2.0*a)

→(3.0 + sqrt([b*b] - 4.0*a*c))/(2.0*a)
→(3.0 + sqrt(9.0 - [4.0*1.0*(-18.0)])/(2.0*a)
→(3.0 + sqrt([9.0 + 72.0]))/(2.0*a)
→(3.0 + [sqrt(81.0)])/(2.0*a)
→([3.0 + 9.0])/(2.0*a)
→12.0 /([2.0*1.0])
→12.0 / 2.0
→6.0
```

33 – Declaração Read

A declaração **read** do Fortran permite ler dados e associá-los a um conjunto de variáveis.

Os dados a serem lidos, podem ser fornecidos tanto pelo teclado quanto por um arquivo de dados.

- **Forma geral da declaração read**

read(unit=u, fmt="fs") lista entrada

onde:

u	número que especifica o dispositivo (unidade) que será usado o número **5** ou ***** especifica o teclado
fs	identifica o formato que será usada
lista entrada	lista de variáveis separadas por vírgula, na ordem em que serão lidas

Lista-dirigida:

Uma lista cuja formatação é especificada por um asterisco (*) é chamada de lista-dirigida (list-directed).

Uma lista-dirigida à entrada (list-directed input) especifica uma lista de variáveis enviada (dirigida) para a entrada (read) enquanto lista-dirigida à saída (list-directed output) é a lista envia para a saída

Normalmente, quando não é necessário um controle maior sobre a formatação que será usada, uma lista-dirigida é simples e adequada para a maioria das situações de entrada e saída de dados. Entretanto, quando se deseja controlar a formatação usada ela não pode ser usada.

Na saída, a aparência exata dos dados de uma lista-dirigida é inteiramente dependente do compilador usado, por isto é diferente de fabricante para fabricante.

exemplos:
```
read(unit=12, fmt=*) a                                  ! lista-dirigida
read(unit=5, fmt="(6I3)") v1, v2, v3, v4, v6, v7
read(unit=29, fmt="(I4 ,I2 , I2)") temperatura, dia, mes

read(unit=*, fmt=*) a, b, c          ! lê (pelo teclado) as variáveis da lista-dirigida
read(unit=*, fmt="(2i5)") e, f
read(unit=45, fmt=*) k, d, w         ! lista-dirigida lida no dispositivo 45
```

Observação: foi utilizado somente o formato I (inteiro). O primeiro número indica quantas repetições são feitas do formato e o segundo número indica quantos dígitos serão usados no número

- ## Forma simplificada do comando read

read*, lista entrada

Esta declaração só permite entrar dados pelo teclado. Ela faz a leitura dos dados fornecidos pelo teclado e atribui os valores lidos às variáveis da lista.

lista-dirigida

O * usado na declaração refere-se ao formato de leitura, que neste caso é livre (*).

```
read*, v1

read*, a, b, c

read*, temperatura, dia, mes
```

lista variáveis

Caso necessário pode-se usar um formato definido pelo programador. Exemplos:

```
read "(I5)", v1

read"(i2, 3i5)", a, b, c, d
```

- **Leitura de dados contidos em um arquivo**

Pode-se ler os dados contidos num arquivo

- Para ter acesso ao arquivo temos que garantir acesso à ele com o comando **open**

open(unit=u, file="fname", status="sta", action="act", position="pos")

u	expressão inteira (mais freqüentemente uma constante inteira) escolhida arbitrariamente pelo programador, que passará a identificar o arquivo
fname	nome do arquivo com o qual se vai trabalhar se necessário fornecer o caminho
sta	status new, old, replace, scratch ,unknown
act	ação a ser realizada sobre o arquivo read, write, readwrite
pos	posição onde inicia a leitura do arquivo, se o arquivo for seqüencial default é seqüencial rewind, backspace

exemplos:
```
open(unit=20, file="dados.data", status="old", action="read", position="rewind")
open(unit=32, file="/home/paulo/resultados/temperaturas.data ", status="old", &
action="read", position="rewind")
```

Ao término do trabalho com o arquivo deve ser fechado com a declaração **close**.

```
close(unit=25)
close(unit=32)
```

- Cada declaração read lê uma linha do arquivo por vez

```
.................
.................
character(len=10) :: nome
real              :: nota_mecanica, nota_termodinamica, cr
integer           :: ano_ingresso
.................
.................
open(unit=10, file="notas.data", status="old", action="read", position="rewind")
.................
.................
read(unit=10, fmt=*) nome, nota_mecanica, nota_termodinamica, cr, ano_ingresso
.................
.................
```

O arquivo notas.data pode ter a seguinte forma:

```
"paulo"     8.9     9.2     8.32    1999
"antonio"   7.2     10.0    7.9     2001
"pedro"     8.2     9.4     8.2     2002
.............................
```

observe que os dados são separados por espaço dentro do arquivo.

O seguimento de programa mostrado faz a leitura da primeira linha (só um read) então as variáveis ficam com os seguintes valores:

```
nome = paulo
nota_mecanica = 8.9
nota_termodinamica = 9.2
cr = 8.32
ano_ingresso = 1999
```

- Necessitando de novos valores é necessário fazer nova leitura do arquivo. Cada execução do comando read inicia a entrada de dados uma nova linha do arquivo.

```
.............
.............
read(unit=10, fmt=*) nome, nota_mecanica, nota_termodinamica, cr, ano_ingresso
read(unit=10, fmt=*) nome, nota_mecanica, nota_termodinamica, cr, ano_ingresso
read(unit=10, fmt=*) nome, nota_mecanica, nota_termodinamica, cr, ano_ingresso
.............
.............
```

após a execução do 3 (três) declarações read as variáveis conterão os dados do pedro, i. é,

nome= pedro
nota_mecanica = 8.2
nota_termodinamica = 9.4
CR = 8.2
ano_ingresso = 2002

a leitura de um arquivo inteiro, usualmente, é feita com o uso da declaração **do**

- Quando o número de variáveis for menor que o número de dados do arquivo, os dados extras existentes no arquivo não serão lidos.

```
real ::  a, b, c, d, e, f
.....................
.....................
open(unit=50, file="numeros.data", status="old", action="read", position="rewind")
.................
.................
read(unit=50, fmt=*) a, b, c
read(unit=50, fmt=*) d, e, f
.................
.................
```

se o arquivo de dados (numeros.data) contiver:

```
10.0    2.00    30.0    4.00
5.0     6.00    70.0    8.01
```

teremos:

a = 10.0
b = 2.00
c = 30.0
d = 5.0
e = 6.00
f = 70.0 4.00 e 8.01 não serão lidos

- Lendo dado inteiro para uma variável real

  neste caso será feita a conversão do dado de inteiro para real

- Lendo dado real para uma variável inteira

  neste caso ocorrerá um erro

- Lendo string em variável caractere

  comprimento dado igual que ao comprimento variável caractere → ok

  comprimento dado maior que o comprimento da variável neste caso o valor lido será cortado

  comprimento dado menor que o comprimento da variável o valor será completado com branco

- Uma declaração read sem lista de entrada, simplesmente, pula uma linha no arquivo de entrada

```
integer ::  a, b, c, d
.....................
read(unit=70, fmt=*) a, b
read(unit=70, fmt=*)
read(unit=50, fmt=*) c, d
```

com o arquivo de dados

```
100  400  700          implica em:    a=100
200  500  800                         b=400
300  600  900                         c=300
                                       d=600
```

34 – Declaração Write

Saída de dados, seja para o vídeo do computador seja para um arquivo pode ser feita com a declaração **write**.

- **Forma geral**

write(unit=u, fmt="fs") lista_saída

u	expressão inteira (mais freqüentemente uma constante inteira) escolhida arbitrariamente pelo programador, que passará a identificar o arquivo
fs	formato a ser usado se necessário fornecer o caminho
lista_saída	lista com os nomes das variáveis, separadas por vírgula, na ordem em que serão escritas

exemplos:

```
write(unit=*, fmt=*)  e1, e2                    ! lista dirigida à entrada
write(unit=*, fmt="(5I3)") a, b, c, d, f
write(unit=15, fmt="(I4)") h
write(unit=46, fmt=*) x**2, (a - (d/s)), u      ! lista dirigida à entrada
```

- **Formas simplificadas**

Escreve na tela do computador (vídeo).

WRITE

write*, lista_saída

exemplos:

```
write*, " entre com o valor da nota"                ! lista-dirigida à entrada
write*, "a=",a, "b=", b, "C=", c                    ! lista-dirigida à entrada
write"(3i8)", resistencia, impedancia, frequencia   ! lista de variáveis
```

PRINT

Somente para impressão na tela do monitor.

print*, lista_saída

exemplos:

```
print*, " entre com o valor da nota"                    ! lista-dirigida à entrada
print*, "a=",a, "b=", b, "C=", c                        ! lista-dirigida à entrada
print"(3i8)", resistencia, impedancia, frequencia        ! lista variáveis
```

- Cada declaração write inicia uma nova linha

```
integer                          ::  dia, ano
character(len=*), parameter ::  mes = "abril"
.....................
.....................
dia = 10
ano = 2002
write(unit=*, fmt=*)  "A data do exame é ", dia, "de", mes, "de", ano
write(unit=*, fmt=*)
write(unit=*, fmt=*) " A data do exame é ", dia
write(unit=*, fmt=*) "de", mes
write(unit=8, fmt=*) "de", ano
```

o programa produz a seguinte saída:

```
A data do exame é 10 de abril de 2002
                                                    ! linha em branco
A data do exame é 10
de abril
de 2002
```

a linha em branco é produzida pela declaração write(unit=*, fmt=*)

- Se uma linha for muito grande para caber na tela do computador ela será escrita em mais de uma linha.

- **Saída Formatada**

  O Fortran mostra todos os dados relevantes, sem nenhuma perda, de dado

  por exemplo uma saída pode ser do tipo 186.345456632
  caso queira escrever 186.34

  aí sim é necessário utilizar o formato para escrever o dado com o formato desejado.

35 – Declaração Format

Uma declaração format pode ser usada para definir uma formato diferente do formato default usado pelo compilador.

Declarações de entrara e saída escrita **com formato livre** (lista-dirigida).

```
write(unit=12, fmt=*) lista saída
print *, lista saída
read(unit=5, fmt=*) lista entrada
read*, lista entrada
o uso do " * " diz que se está usando o formato livre
```

Com o formato livre não se perde nenhuma informação mas, também, não se tem nenhum controle sobre o formato de apresentação do dado.

Exemplo:

print*, "resistencia igual a ", resistencia

gerar uma saída do tipo: resistencia igual a 12.345634

Uma saída do tipo resistencia igual a 12.34

só pode ser conseguida com o uso da declaração format

A declaração format pode ser usada para controlar a aparência dos dados tanto de entrada quanto de saída.

A forma geral da declaração Format é:

label **format** (lista descritores)

onde: label (rótulo) é um número entre 1 e 99999 usado para identificar a declaração format. Não pode ser repetido
lista de descritores indica os formatos que serão usados com os dados

Exemplo:

```
.............
.............
100 format (3I5, F12.4, E10.3)
200 format (a3, 2i10)
.............
.............
read(unit=*, fmt=100) a, a1, a2, temp, coeficiente
write(unit=23, fmt=200) massa, viscosidade, temperatura
.............
.............
```

As declarações Format podem ser colocadas em qualquer ponto do programa. Usualmente elas são agrupadas e colocadas ou no início ou na parte final do programa.

A lista de descritores é formada por vários descritores de formato, separados por virgulas.

Por questão de clareza (legibilidade) a **declaração Format** deve ser **substituída**, sempre que possível, **por uma constante caractere**.

Existem 3 formas de se especificar o formato dos dados.

- escrever o formato numa **constante com nome** tipo **caractere** e usá-la no lugar da declaração Format

esta forma é **MUITO RECOMENDADA !**

```
.............................
.............................
character(len=*), parameter :: fs1 = "(i2,f12.3)"
character(len=*), parameter :: fs2 = " (4i3,5e15.3,a5) "
.............................
.............................
read(unit=*, fmt=fs1) lista entrada
write(unit=*, fmt=fs2) lista saída
print fs1, lista saída
.............................
.............................
```

atenção: na especificação do tamanho da constante com nome caractere deve-se usar o * (asterisco)

- escrever o **formato numa variável caractere** e usá-la no lugar do formato

  PODE SER USADO ! *entretanto dê **preferência ao uso de uma constante com nome***

```
character(len=10) :: fs1

fs1 = "(I5,5F8.2)"
read(unit=*, fmt=fs1 )  lista entrada
write(unit=*, fmt=fs1) lista saída
print fs1, lista saída
```

  atenção: na especificação do tamanho da constante caractere o comprimento tem que ser igual ou maior que o comprimento do formato. Se isto não acontecer, um erro ocorrerá quando da execução do programa

- usando a declaração format

  DÊ PREFERENCIA à constante com nome caractere

```
real      :: a, b
integer :: cont
.............
.............
10 format (3I5, 7F12.4, 2E10.3)
20 format (a3, 3I10)
.............
.............
read(unit=*, fmt=10) lista entrada
Print 20, lista saída
write(unit=23, fmt=10) lista saída
```

  A declaração format deve ser evitada porque utiliza um label. Labels devem ser evitados nos programas Fortran 95.

36 – Descritores de Formato

Um descritor de formato (edit descriptor) fornece informações ao sistema de como ele deve trabalhar o dado para sua saída ou entrada.

Portanto os descritores devem, essencialmente, conter informações sobre a forma e a quantidade de dados. Por isto os seguintes detalhes são importantes:

número de repetições	a
número de dígitos usados	w
número mínimo de posições	m
número de decimais	d
número de dígitos no expoente	e

Importante: O número de posições não está indicando a precisão (número de algarismos significativos) do número. Esta informação é caracterizada pela precisão do número real.

Os descritores são usados somente para mudar a aparência com que o dado será mostrado. Nada mais do que isto.

- ## Inteiro

Descritor **I**.

aIw.m

a	repetições
I	inteiro
w	tamanho do campo contando com o sinal
m	número mínimo de dígitos

O inteiros são escritos justificados a direita nos seus campos. Os espaços não usados do campo são deixados em branco. **Na entrada de dados**, os números inteiros são lidos da esquerda para a direita.

- ## I, Iw, Iw.m

Descritor **I**

com I5 o valor **-99** será: bb-99 (**-99**; b representa branco. Temos 2 brancos antes do número: 2+3=5)

com I5.3 o valor **99** será mostrado como: bb099 (**099**, ou seja, tendo 2 brancos antes: 2+3=5)

Descritor I com repetição

aI, aIw, aIw.m

Se w for muito pequeno para conter o número:

saída: será w asteriscos

entrada: serão lidos só os números mais a esquerda

o número 1999 e o descritor I3 gera uma saída igual a: ***
o número 1234 com o descrito I3 na entrada gera um dado igual a: 123

O descritor **I6.4** especifica um total de 6 campos incluindo o sinal com um mínimo de 4 dígitos:

```
write(unit=*, fmt="(2I6.4) ") 56, -912
```

gera a saída: 0056 -0912

- ## Real

Números reais podem ser escritos basicamente em 2 formatos: com e sem expoentes

- ### Real sem Expoente

Descritor **F**.

aFw.d

a	repetições
F	real (ponto decimal)
w	tamanho do campo contando com o sinal e o ponto
d	número dígitos depois do ponto decimal

O reais são escritos justificados a direita nos seus campos. Os espaços anteriores não usados do campo são deixados em branco. Se necessário serão colocados tantos zeros quanto necessário para obedecer a especificação da quantidade de decimais.

- **F, Fw, Fw.d**

Descritor **F**

O número 18.7 com o descritor F7.3 será mostrado como: 18.700

Descritor F com repetição

aF, aFw, aFw.d

Se w for muito pequeno para conter o número:

saída: será w asteriscos

entrada: **é bastante complicada. Não usar entrada formatada, preferencialmente**.

As regras que se aplicam à entrada não são complicadas, mas não serão abordadas aqui. A complicação da entrada formatada vem da combinação: regras + usuário.

O descritor F7.4 especifica um total de 6 dados incluindo o sinal e o ponto com 4 decimais

write(unit=*, fmt="(F8.4)") -18.34

terá a saída: -18.3400

- **Expoente**

Descritor **E**.

aEw.d

a	repetições
F	real (ponto flutuante com expoente)
w	tamanho do campo contando com o sinal e o ponto
d	número dígitos depois do ponto decimal

O valor numérico representado está entre **0.1 e 1.0**

- **E, Ew, Ew.d**

Descritor **E**

o descritor E terá a forma: **s0.xxxEsxx**

onde s sinal
x dígitos
E representa a base 10 (10^{sxx})

Se w for muito pequeno para conter o número:

saída: será w asteriscos

entrada: **é bastante complicada. Não usar preferencialmente**.

O número 0.000012 com o descritor E9.3 o valor será: 0.120E-04

O número -0.000012 será: *********

O número 6128.3 será: 0.612E04

Descritor **E com repetição**

aE, aEw, aEw.d

Formas alternativas:

Temos possibilidade de escrever números seguindo a notação usada pela engenharia ou pela notação científica.

Notação de Engenharia	Notação Científica
EN	**ES**
o valor representado está entre **1.0 e 1000** número **0.0217** com descritor **EN9.2** = **21.70E-03**	o valor representado está entre **1.0 e 10** número **0.0217** com descritor **ES9.2** = **2.17E-02**

- **Caractere**

Variáveis caractere podem ser formatadas com o descrito A,

Descritor **A**.

aAw

a	repetições
A	caractere alfanumérico
w	tamanho do campo

- **A, Aw**

Descritor **A**

Lê e escreve um caractere simples ou um string (cadeia de caracteres)

Descritor **A com repetição**

aA, aAw

Se o campo w é maior que o número de caracteres então os caracteres são justificados a direita e o campo preenchido com branco

Considere uma saída formatada usando a palavra **computacional** (13 caracteres)

Com o descritor A13 teremos: computacional
Com o descritor A15 teremos: **bb**computacional (bb são 2 brancos colocados antes da palavra)
Com o descritor A11 teremos: computacion

Entrada: Na entrada do dado, o string não precisa estar entre apóstrofo.

- **Posições em Branco**

Este descritor é obsoleto para o Fortran 90 e foi eliminado no Fortran 95.

aX

a → repetições
X → posições

- aX - pula "a" posições
- Na entrada os caracteres serão ignorados
- Na saída "a" espaços em branco serão escritos

- **Tabulador**

Usado para pular diretamente para uma coluna especificada.

Descritor **T**.

aTc

a	repetições
T	caractere alfanumérico
c	número da coluna

Move o cursor para a coluna c.

Os tabuladores podem ser considerados absolutos e relativos:

Absoluto:

exemplo: write(unit=*,fmt="(tc15)") "DIAS"

Pula diretamente para a coluna 15 e escreve a palavra DIAS.

A contagem das posições é absoluta, isto é, ela inicia a esquerda usando identificando a primeira coluna como coluna 1.

Relativos:

Descritores **TR** e **TL**.

Estes descritores iniciam a contagem da posição em que se encontra o cursor:
TR (tab right) conta para a direita
TL (tab left) conta para a esquerda

exemplo:

write(unit=*, fmt="(t15,a,tl16,a)") "DIAS", "CONTROLE DE"

Primeiro escreve DIAS: (Colunas 15-18), com o cursor terminando na coluna 19.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 **15** **16** **17** **18** 19 20 21 22

D **I** **A** **S**

Depois escreve CONTROLE DE: (Colunas 3-13), com o cursor terminando na coluna 14

1 2 **3** 4 5 6 7 8 9 10 11 12 13 14 **15** **16** **17** **18** 19 20 21 22

C O N T R O L E D E D I A S

- **Lógico**

Usado para se trabalhar com valores lógicos.

Descritor **L**.

aLw

a	repetições
L	lógico
w	tamanho do campo

O tamanho do campo define quantos caracteres serão usados.

entrada:
pode ser: **t, f, .true. ou .false.** (ou então: T, F, .TRUE. ou .FALSE.)

saída:
será escrito: **T**, **F**
A saída será justificada a direita no campo reservado.

exemplo:

```
.....
logical:: f=.false.
.....
.....
write(unit=*, fmt="(l1,l2,l3,l4,l5,l6)") f,f,f,f,f,f
.....
.....
```

Produzirá a saída:

1 2 **3** 4 5 **6** 7 8 9 **10** 11 12 13 14 **15** 16 17 18 19 20 **21** 22

F F F F F F

- **Outros**

- barra / especifica uma nova linha

```
.....
integer:: a=5000, b=1000
.....
.....
write(unit=*, fmt="(i4,a,i4)") a, " ",b
write(unit=*, fmt="(i4,/,i4)") a, b
.....
.....
```

irá produzir a saída: 5000 1000
5000
1000

- parênteses permite agrupa vários descritores:

  4(I5.5,F12.6) é o mesmo que: (I5.5,F12.6, I5.5,F12.6, I5.5,F12.6, I5.5,F12.6)

37 – Declaração if-then-else

A forma geral da Declaração if-then-else é:

```
if (expressão-lógica) then
  bloco-1 declarações
else
  bloco-2 declarações
end if
```

A execução desta declaração segue a seguinte norma:

- expressão-lógica é avaliada resultando no valor .true. ou .false.
- se o resultado for **.true.**
  bloco-1 é executado
- se o resultado for **.false.**
  bloco-2 é executado
- depois disto a declaração que segue a declaração IF é executada.

Exemplo

```
.....
integer :: numero
.....
print*, "entre com um número"
read*, numero
  if (abs(numero)>= 50) then
     write(unit=*, fmt=*) "modulo numero igual ou maior que 50"
  else
     write(unit=*, fmt=*) "modulo numero menor que 50"
  endif
.....
.....
```

Todo número que for maior que 50 ou menor que –50 fará o programa escrever na tela do micro:

modulo numero igual ou maior que 50

Números entre –50 e +50 escreve a sentença:

modulo numero menor que 50

Forma mais simples

```
if (expressão-logica) then
  bloco declarações
end if
```

Esta forma mais simplificada só oferece uma opção para execução de declarações.

- a expressão lógica é avaliada
- se ela for **.true.**, o **bloco de declarações é executado**
- o resultado da avaliação for **.false.**, nada é feito. O programa passa para a declaração que segue o IF.

A mais simples delas

if (expressão-logica) uma-declaração-somente

onde uma-declaração-somente é exatamente isto, só uma declaração, que será executada ou não

- a expressão lógica é avaliada
- se o resultado for **.true.**, a declaração especificada será executada
- se o resultado for **.false.**, a declaração não é executada

38 – Declaração if-then-elseif

A declaração if-then-elseif é a forma mais geral de uma declaração IF.

```
if (expressão-lógica-1) then
  bloco-1 declarações
else if (expressão-lógica-2) then
  bloco-2 declarações
else if (expressão-lógica-3) then
  bloco-3 declarações
else if (.....) then
  ...........
else
  bloco-else de declarações
end if
```

Ela se processa assim:

- expressão-lógica-1 é avaliada
  se o resultado for .true. , o bloco-1 declarações é executado.
  Depois o programa passa para a declaração que segue a declaração if-then-elseif

- Se a expressão-lógica-1 é .false. , avalia-se a expressão-lógica-2
  se o resultado da avaliação da expressão-lógica-2 for verdadeira (.true.)
  executa-se o bloco-2 declarações
  Depois o programa passa para a declaração que segue a declaração if-then-elseif

- Se a expressão-lógica-2 é .false. , avalia-se a expressão-lógica-3
  se o resultado da avaliação da expressão-lógica-3 for verdadeira (.true.)
  executa-se o bloco-3 declarações
  Depois o programa passa para a declaração que segue a declaração if-then-elseif

- sempre que obtiver uma resposta .false. O Fortran continua a avaliar as expressões lógicas que se seguem nos **elseif**

- Se a última expressão for .false. e a declaração **else** estiver presente
  as declarações bloco-else serão executadas
  e depois passa-se a declaração que segue o if-then-elseif

Exemplo

```
.............
real:: a, b, c, resultado
integer :: idade
character (len=10) :: classe
.............
if( idade<= 11) then                    ! classifica a pessoa em: criança, adolescente ou adulto
  classe ="crianca"
else if (idade <= 18) then
  classe = "adolescente"
else
  classe = "adulto"
end if
if (a < b .and. a < c) then
  resultado = a
else if (b < a .and. b < c) then
  resultado = b
else
  resultado = c
end if
```

39 – If Encadeados

A declaração IF pode aparecer encadeada, isto é, uma dentro da outra.

Entretanto existem algumas regras importantes que não podem ser violadas para se usar a declaração encadeada.

- **A regra geral é**

Cada **if** que é utilizado encadeado (dentro de outro) deve estar completamente contido dentro de um dos blocos do **if** que o contem.

A forma geral de IF encadeado é:

```
if (exp1-logica) then
    bloco-declarações
    if (exp2-logica) then
      bloco-declarações
    else
      bloco-declarações
    end if
    bloco-declarações
else
    bloco-declarações
    if (exp3-logica) then
      bloco-declarações
    end if
    bloco-declarações
end if
```

Exemplo 1

```
integer  ::  a
.............
if (a > 0) then
   print*,"numero positivo"
else
   if (a < 0) then
      print*, "numero negativo"
   else
      print*, "zero"
   end if
end if
.............
```

Exemplo 2

```
integer :: b
.............
   if (b < 0) then
      print*, -b
   else
      if (b <= 1) then
         print*, b**2
      else
         print*, 2*b
      end if
   end if
.............
```

Exemplo 3

```
Integer :: x, y, z, menor
.............
if (x < y) then
   if (x < z) then
      menor = x
   else
      menor = z
   end if
else
   if (y < z) then
      menor = y
   else
      menor = z
   end if
end if
print*, "o menor valor é ", menor
.............
```

40 – Declaração GOTO

Todo programa que utiliza esta declaração de uma forma ou outra acaba sendo um programa difícil de se manter e pior, acaba perdendo a sua legibilidade. Portanto evite ao máximo utilizar esta declaração. Sempre existe uma forma alternativa de se escrever o programa sem o uso da declaração go to !

Ela foi marcado como obsoleta, isto é, será removido das futuras atualizações do Fortran.

Recomendação: **Não utilize a declaração GO TO**

Sempre haverá uma forma alternativa ao seu uso, portanto nunca faça uso dela.

- **Go to incondicional**

go to label

label identifica uma declaração, para onde o programa será direcionado de forma incondicional
O label (rótulo) é um número inteiro, com no máximo 5 dígitos, escrito a esquerda de uma declaração.
Label também são marcados como obsoletos e por isto também devem ser evitados nos programas.

exemplo:

```
.........
x = x**2
goto 20
y = 4.5
20 y = 123.1
.........
```

neste exemplo o valor y = 4.5 nunca será utilizado.
O número 20 é um label que identificam a declaração y=123.1

A declaração **goto computada** também é uma ***declaração obsoleta e deve ser evitada***.

Sua forma é: ***go to(label_1, label_2,.....) expressão-inteira***

ela causa a transferência do controle do programa um dos label identificados dentro do parênteses conforme o resultado da avaliação de expressão-inteira.

41 – Declaração Select case

A declaração select case é bastante útil para se selecionar várias opções.

A grande diferença entre a declaração **if-then-elseif** e a declaração **select case** é que a declaração select case tem somente uma expressão para ser avaliada. Esta expressão pode então assumir uma série de valores pré-definidos.

- **A declaração select case**

```
select case (expressão)
case (seletor-1)
        bloco-declarações-1
case (seletor-2)
        bloco-declarações-2
case (seletor-3)
        bloco-declarações-3
        .............
case (seletor-i)
        bloco-declarações-n
case default
        bloco-default
end select
```

A expressão **seletor-i** que aparece depois da expressão case é uma expressão cujo resultado pode ser do tipo: inteiro, caractere ou lógico.

- **Tipo real não pode ser usado como seletor!**

O seletor é um valor ou uma lista de valores, separadas por vírgulas, que pode ter uma uma das seguintes formas.

valor	**só um valor**
valor-1 : valor-2	**valor-1 tem que ser menor que valor-2**
valor-1 :	**valores maior que valor-1**
: valor-2	**valores menor que valor-2**

valor, valor-1 e valor-2 são constantes ou então constantes com nomes.

- **As regras que se aplicam ao select case são:**

- a expressão é avaliada
- se o resultado estiver no case (seletor-i) então o bloco-declarações-i será executado

  depois da execução do bloco o controle passa para a declaração depois do end select
- se o resultado não estiver em algum dos case (seletor-i)

  executa-se o case default

Exemplo 1

```
.....
integer :: numero
.....
select case (numero)
case (2000, -2000)
        print*,  "2000 ou -2000"
case (:1000)
        print*,  "positivo"
case (0)
        print*,  "zero"
case (-1000:)
        print*,  "negativo"
case default
        print*,  "numero fora da faixa -1000 a +1000"
end select
.....
.....
```

Exemplo 2

```
integer  :: i
.....
.....
select case (i)
case (3,5,7)
        print*, "i é primo"
case (10:)
        print*, "i > 10"
case default
        print*, "i não é primo e i<10"
end select
.....
.....
```

Exemplo 3

```
character (len=1) :: L
.....
.....
select case (L)
case ("a" : "j")
        write(unit=*, fmt=*)"letra entre a e j"
case ("l": "p", "u": "y")
        write(unit=*, fmt=*)"letra entre l e p e entre u e y"
case ("z", "q": "t")
        write(unit=*, fmt=*)"as letras z, q, r, s, t"
case default
        write(unit=*, fmt=*)"não é uma letra"
end select
.....
.....
```

42 – Declaração DO

A declaração **do** é uma declaração de iteração, isto é, capaz de repetir um conjunto de instruções.

- **A forma geral da declaração é:**

```
do var = expr1, expr2 [, expr3]
  bloco-declarações
end do
```

onde:

var	variável escalar inteira
expr1	expressão escalar inteira que define o valor inicial
expr2	define valor final
expr3	opcional. Define o passo, se utilizado não pode ser zero

O número de iterações (repetições) é definido pela expressão:

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

Exemplos

Exemplo 1

```
integer  :: i
.....
.....
do i = 1, 5
.....
     ! faz 5 repetições: 1, 2, 3, 4 e 5
.....
end do
.....
```

Exemplo 2

```
integer :: k
.....
.....
do k = -7, 4, 3
.....
     ! faz 4 repetições: -7, -4, -1, 2
.....
end do
.....
```

Exemplo 3

```
integer  ::cont
.....
.....
do cont = 10, 20, 2
.....
     ! faz 6 repetições: 10 ,12 , 14, 16, 18, 20
.....
end do
.....
```

Exemplo 4

```
integer :: j
.....
.....
do k = -7, 4, 3
.....
     ! faz 5 repetições: 20, 15, 10, 5, 0
.....
end do
.....
```

Quando trabalhar com a declaração **do**, tenha sempre em mente:

- os valores para: expr1, expr2 e expr3 são calculados só uma vez, no início da execução da declaração **do**
- quando a variável **var** for maior que o valor de **expr2** a declaração **do** encerra sua execução
- o valor da **expr2** pode ser menor que o valor de **expr1**
- quando se utilizar **expr3** ele nunca pode ser igual a zero
  - **passo negativo**
    significa decréscimo na contagem, neste caso, quando **var** for menor que o valor **expr2** o **do** termina
  - **passo positivo**
    contador vai sempre incrementando o seu valor

- **nunca** altere (por qualquer meio) o valor do contador **var** dentro da declaração **do**
- **nunca** altere os valores das variáveis expr1, expr2 e expr3 dentro da declaração **do**
- **nunca** use variáveis **reais** na declaração do **do**

Exemplo: FATORIAL

```
integer :: fatorial, n, i
.............
fatorial = 1
print*, " entre com o número "
read(*,*) n
do i = 1, n
   fatorial = fatorial * i
end do
.............
```

O que o programa faz é calcular a expressão: 1 * 2 * 3 *...* (n-1) * n

- ## Transferindo Controle

Quando existe necessidade de se pular uma iteração ou de se abandonar o processo de iteração antes dele estar acabado faça uso das declarações:

- **EXIT**
  use a declaração **exit** para abandonar a qualquer momento um **do**
- **CYCLE**
  use a declaração **cycle** para iniciar outra iteração do **do**

Exemplos:

```
integer :: i
.................
   do i=1,10, 2
     ! originalmente faz: 1, 3, 5, 7 e 9
.................
     if(i==3) cycle
     ! com este if só faz a iterações: 1, 5, 7 e 9
.................
   end do
.................
.................
```

```
integer :: k
.................
  do k=1,100
    ! originalmente faz 100 repetições
.................
    if(i= =34) exit
    ! na realidade só faz 34 iterações
.................
  end do
.................
.................
```

43 – Do Encadeados

A declaração **do** pode ser utilizada encadeado, isto é, uma dentro do outra.

- **A regra básica é:**

A declaração DO interna deve estar completamente contida em algum bloco da declaração **do** mais externa.

```
do var1 = e1, e2, e3
   bloco-1 declarações
   do var2 = e1, e2, e3
     bloco-2 declarações
   end do
   bloco-3 declarações
end do
```

ATENÇÃO: a declaração **exit** referente a uma declaração **do** interna faz a saída deste desta declaração **do** para uma declaração **do** mais externa.

Exemplos

```
integer :: i, j
.....
do i = 1, 5
  do j = 1, 10
    print*, (i+j)**2
  end do
end do
.....
.....
```

```
integer  :: x, y
integer  :: b, c
.....
.....
do x = 2, 15
  do y = 1, x-1
    b = 2x*x - 2y*y
    c = x**3 + y**3
    print*, b, c
  end do
end do
.....
.....
```

```
integer  :: i, j, soma= 0
.....
.....
do i = 1, 5
  do j = 1, i
    soma= soma + j
  end do
  print*, soma
end do
.....
.....
```

44 – Do Implícito

A declaração **do** implícito provê um meio rápido de listar vários itens. Ele é muito útil em listas de saída e de entrada

- **A forma geral da declaração do implícito é:**

(dlist, do-var = expr-i, expr-f[,expr-p])

onde:

dlist	lista de elementos
do-var	variável inteira
expr-i	valor inicial
expr-f	valor final
expr-p	passo

outra forma de escrever é:

(item-1, item-2, ..., item-n, do-var = inicial, final, passo)

(item-1, item-2, ..., item-n, do-var = inicial, final)

Quando o passo não é especificado ele é assumido ser unitário.
Os itens podem ser expressões ou elementos de uma matriz ou de um vetor.

Exemplos

print*, (i, i = -1, 2)

foi usado uma declaração **do** implícito com a declaração print.
Ela vai escrever na tela do mico:

-1, 0, 1, 2

print*, (a(i), i = 1, 3)

produz uma saída com os elementos do vetor a:

a(1), a(2), a(3)

print*, (i, i*i, i = 1, 7, 2)

esta declaração imprime o seguinte:

1, 1, 3, 9, 5, 25, 7, 49

- **Do implícito encadeados**

Declarações **do**s implícitos podem ser encadeados. Veja o exemplo:

(i+j, (j, j = 2,6,2), i = 1, 3)

que seria avaliada assim:

[1+j, (j,j=2,6,2)]

[2+j, (j,j=2,6,2)]

[3+j, (j,j=2,6,2)]

Gerando a seguinte saída:

3, 5, 7, 4, 6, 8, 5, 7, 9

Atente ao fato da declaração **do** mais **interna** estar **completamente contida** na declaração **do** externa.

45 – Vetores e Matrizes

Uma matriz é uma coleção de dados da mesma espécie identificados pelo mesmo nome. Cada elemento da matriz é identificada por um índice ou subscrito.

As matrizes em Fortran 95 podem ter até 7 dimensões.

Uma matriz de uma dimensão é chamada de vetor, enquanto que a matriz de dimensão zero é um escalar.

A terminologia empregada com matrizes

- **rank**
  número de dimensões
- **extent**
  número de elementos em uma dimensão
- **shape**
  vetor com valores das extensões
- **size**
  produto das extensões
- **conformance**
  mesmo vetor com valores de extensões (mesma forma)

- **Declarando uma Matriz**

A sintaxe para declara uma matriz é:

type, dimension(extensão) :: lista-entidades

onde:

type	real integer complex logical tipo derivado
dimension	atributo dimension, requerido para definir a dimensão da matriz (rank)
extensão	é o tamanho (extensão) de cada dimensão (extent)
lista-entidades	é a lista contendo os nomes das matrizes, que serão criadas, separadas por vírgula

Exemplos

VETOR	MATRIZ
real, dimension(1:10) :: dados Vetor (matriz) de dados real de uma dimensão com 10 elementos Neste caso os limites inferior e superior foram especificados(1:10) rank = 1 extent = 10 shape = (/10/) size = 10	**real, dimension(3,3) :: dados** Matriz de dados real de **duas dimensões** com 9 elementos rank = 2 extent = 3 e 3 shape = (/3, 3/) size = 9
integer, dimension(10) :: coeficientes Vetor denominado coeficientes do tipo inteiro de uma dimensão com 10 elementos rank = 1 extent = 10 shape = (/10/) size = 10	**integer, dimension(10,4,5) :: coeficientes** Matriz chamada coeficientes do tipo inteiro de 3 dimensões com 1000 elementos rank = 3 extent = 10, 4 e 5 shape = (/10, 4, 5/) size = 200
integer, dimension(10:20) :: lados Vetor chamado lados do tipo inteiroa de uma dimensão com 11 elementos rank = 1 extent = 11 shape = (/11/) size = 11	**real, dimension(10:12,2:4) :: lados** Matriz chamada lados,tipo real, de duas dimensões com 9 elementos rank = 2 extent = 3 e 3 shape = (/3, 3/) size = 9 As matrizes dados e lados tem **mesma forma** **conformable** ==> ***mesma "shape"***

Os inteiros usados para definir a extensão de uma matriz (ou vetor) pode ser uma constante com nome

Exemplo:

```
.....
.....
integer, parameter :: max = 10
integer, parameter :: min = 1
real, dimension(min:max) :: pesos
.....
.....
```

isto define a matriz de 10 elementos: pesos(1:10)

rank = 1
extent = 10
shape = (/10/)
size = 10

- **Elemento de uma matriz**

Um elemento de uma matriz pode ser referenciado (individualizado) pelo seu índice ou subscrito. A forma de refenciar um elemento de matriz é:

matriz-nome (expr)

onde:

matriz-nome	define a matriz que está sendo usada
expr	especifica o elemento expressão inteira que deve necessariamente resultar em um inteiro dentro da extensão da dimensão utilizada

Exemplo:

```
.....
.....
real, dimension(1:4)     :: soma
.....
.....
```

define o vetor chamado soma que possuí 4 elementos: soma(1) soma(2) soma(3) soma(4)

Então o seguimento de programa a seguir permite que se atribua (entre com) o valor 4.0*i (quatro vezes o indice do elemento da matriz) em cada elemento.

```
.....
.....
do i=1,4
    soma(i) = 4.0 * i
end do
.....
.....
```

46 – Entrada e Saída de dados usando Matrizes

Entrada de Dados

Construtor de Matriz

No Fortran 95 uma matriz de rank 1 (**um vetor**) pode ser construído com o uma lista de elementos envolvidos entre os tokens: " **(** / " e " **/)** ".

Esta forma é chamado de construtor de matriz.

Exemplos:

```
.....
integer, dimension(1:4) :: du
.....
.....
du = (/ 1, 3, 6, 7 /)
.....
.....
```

tem-se:

du(1)=1
du(2)=3
du(3)=6
du(4)=7

O construtor da matriz pode ser escrito de outras formas

	tem-se:
.....	
integer, dimension(6) :: f	f(1)=1
.....	f(2)=4
.....	f(3)=9
f = (/ (i**2,i=1,6) /)	f(4)=16
.....	f(5)=25
.....	f(6)=36
.....	produz o mesmo vetor
integer,dimension(6):: f=(/1,4,9,16,25,36/)	
.....	f(1)=1
.....	f(2)=4
	f(3)=9
	f(4)=16
	f(5)=25
	f(6)=36
.....	produz o seguinte vetor:
integer, dimension(5):: a	
.....	a(1)=1
.....	a(2)=2
a=(/ 1, (i, i=2,6,2), 5)	a(3)=4
.....	a(4)=6
.....	a(5)=5

Declaração Reshape

Uma matriz (rank > 1) pode ser preenchida usando-se o construtor de matrizes mas, neste caso tem-se que fazer uso da declaração **reshape**.

A declaração reshape reformula uma matriz, permitindo definir uma nova forma para a matriz.

Ela pode ser usada para criar conformidade entre matrizes.

Um construtor de matriz deve ter o tamanho (size) exato da matriz. Como o construtor de matriz é formado por uma única linha, temos que utilizar a declaração reshape para altera sua forma (shape) para a forma da matriz alvo.

A declaração reshape: **reshape(dados, shape)**

Exemplo:

```
.....
integer, dimension(4,3):: resultados
.....
resultados = reshape((/1, 4, 7, 10, 2, 5, 8, 11, 3, 6, 9, 12/), (/4,3/))
.....
! mesma declaração mas agora usando
! uma diagramação mais favorável à visualização
.....
resultados = reshape(  (/  1,  4,  7, 10,  &
                           2,  5,  8, 11,  &
                           3,  6,  9, 12    /),  (/4,3/)  )
.....
```

$$resultados = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

Observação: o construtor de matriz entra com os dados uma coluna depois da outra)

Uma forma bastante simples de se entrar com dados num matriz **usando o teclado** é:

```
.....
integer, parameter :: nf=10
integer, dimension(nf) :: a
integer                :: i
.....
.....
print*, " Entre com ", nf, "inteiros"
do i = 1, nf
    read*, a(i)
end do
.....
.....
```

Um do implícito também pode ser usado

```
.....
integer, parameter :: nf=10
integer, dimension(nf) :: a
integer                :: i
.....
.....
print*, " Entre com ", nf, "inteiros"
read*, (a(i), i=1,nf)
.....
.....
```

- **Arquivos**

Os dois exemplos anteriores têm uma característica importante a ser destacada:

A definição do valor da constante com nome nf, identifica quantos dados serão lidos. Admita que nf seja igual a 4, então:

```
.....
integer, parameter :: nf=10                cada declaração read*, a(i)
integer, dimension(nf) :: a
integer                :: i                lê um dado
.....
.....                                      a(1)
print*, " Entre com ", nf, "inteiros"      a(2)
do i = 1, nf                               a(3)
    read*, a(i)                            a(4)
end do
.....                                      porque cada declaração read lê uma linha
.....
```

Então se os dados forem colocados num arquivo para serem lidos o arquivo tem que ter o seguinte formato:

```
30
50
70
90          que irá gerar os: a(1)=30  a(2)=50  a(3)=70  a(4)=90
```

Saída de Dados

Os mesmos princípios básicos se aplicam à saída de dados.

Exemplos

```
.....                                          irá gerar uma saída com a seguinte diagramação
integer, parameter       :: nf=5
integer, dimension(nf) :: a, b
integer                  :: i                      a(1)  b(1)
.....                                              a(2)  b(2)
do i = 1, nf                                       a(3)  b(3)
  write(unit=*, fmt=*)  a(i), b(i)                 a(4)  b(4)
end do                                             a(5)  b(5)
.....
.....
```

por outro lado a declaração

```
.....
integer, parameter       :: nf=5
integer, dimension(nf) :: a, b
integer                  :: i
.....
do i = 1, nf
write(unit=*, fmt=*)  (a(i), b(i), i=1, nf)
end do
.....
.....
```

irá gerar a seguinte saída

```
a(1) b(1) a(2) b(2) a(3) b(3) a(4) b(4) a(5) b(5)
```

tudo em uma única linha

47 – Matriz Inteira

Uma característica importante do Fortran 90/95 é a referência e as operações com matrizes inteira, isto é, a matriz é referenciada simplesmente pelo seu nome.

Por exemplo, a operação de adição "**c=a+b**" onde todos os elementos da operação são matrizes é perfeitamente possível se as matrizes forem conforme (mesma forma).

Os escalares são sempre conforme com qualquer matriz. Então "**b=a+3**" é sempre uma operação possível pois gera uma matriz que é sempre conforme.

Funções intrínseca elementares podem ser usadas com matrizes. Por exemplo a expressão:
"**c = sin(a) + b**" com matrizes conforme é possível.

Estas características do Fortran 90/95 permitem escrever códigos fontes mais legíveis e fáceis de serem interpretados.

Referência a uma matriz

- **Matriz Inteira**

As expressões

```
.....
real, dimension(3,3) :: a
.....
a = 0.0
.....
.....
```

coloca todos os elementos da matriz igual a zero

```
.....
real, dimension(5,3) :: b, c, d
.....
b = c + d
.....
.....
```

soma a matriz c com a matriz d
coloca o resultado na matriz b
(se elas forem conforme)

```
.....
real, dimension(3,4) :: c, k
.....
k = sin(c)
.....
.....
```

escreve a matriz k
onde cada elemento é o seno dos elementos da matriz c

- **Elementos da Matriz**

As expressões

coloca o valor zero no elemento a(1)

```
.....
real, dimension(3,3) :: a
.....
a(1) = 0.0
.....
.....
```

```
.....                              faz o elemento (2,2) da matriz b
real, dimension(8) :: a,
real, dimension(5,3) :: b, c       ser igual soma dos elementos
.....
b(2,2) = a(5) + c(3,1)             a(5)   e   c(3,1)
.....
.....
```

- **Seção da Matriz**

As expressões

atribui o valor zero aos elementos

```
.....
real, dimension(10) :: a           a(2)
.....                              a(3)
a(2:4) = 0.0                       a(4)
.....
.....
```

```
                                   soma uma subseção de c
                                   com o escalar 1.0
.....
real, dimension(4,5) :: b,         coloca o resultado numa subseção de b
real, dimension(3,4) :: c
.....                              b(3,4) = c(1,2) + 1.0
b(3:4, 4:5) = c(1:2, 2:3) + 1.0    b(4,4) = c(2,2) + 1.0
.....                              b(3,5) = c(1,3) + 1.0
.....                              b(4,5) = c(2,3) + 1.0
```

- **Clareza e concisão:**

  - **colocando zero na matriz**

```
.....
integer, parameter       :: i=10, f=60
integer, dimension(i:f) :: a
integer                  :: k      atribui o valor zero a cada um dos elementos da matriz
.....
do k = i, f
   a(k) = 0
end do
.....
.....
```

Com o Fortran 90/95 podemos fazer:

```
.....
```

```
integer, parameter       :: i=10, f=60
integer, dimension(i:f) :: a                    matriz inteira
integer                  :: k
.....                                           veja a simplicidade e clareza
a=0.0
.....
.....
```

- **operando sobre os elementos da matriz**

```
.....
real, dimension(5,5) :: b, c, d
integer              :: i, j
.....
do j=1,5
   do i=1,5
      b(i,j) = c(i,j) * d(i,j) - b(i,j)**2
   end do
end do
.....
.....
```

Com o Fortran 90/95 podemos fazer:

```
.....
real, dimension(5,5) :: b, c, d                 matriz inteira
.....
b = c * d - b**2                                veja a simplicidade e clareza
.....
.....
```

- **Entrada e Saída**

Um matriz em Fortran é guardada como elementos contínuos de memória. Por isto quando se trabalha com matrizes a referência a ela obedece a ordem como ela é guardada na memória do computador.

O Fortran estabelece que a ordem é por coluna:

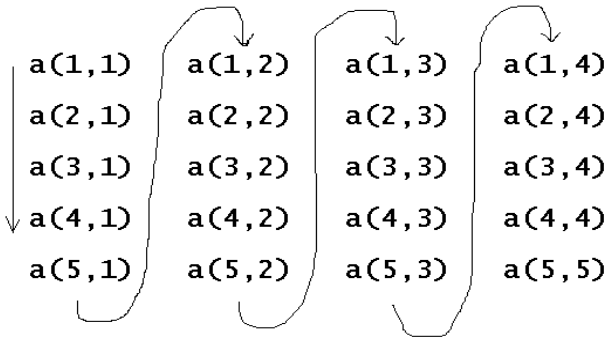

Figura 43.1 - Ordem de leitura de uma matriz

Então :

```
.....
real, dimension(5,5) :: a
.....
print*, a
.....
.....
```

irá produzir a seguinte saída

a(1,1) a(2,1) a(3,1) a(4,1) a(5,1) a(1,2) a(2,2) a(2,3) a(2,4) a(2,5) a(1,3) .....

ou seja a declaração lista imprime a primeira coluna seguida da segunda coluna, seguida da terceira coluna e assim por diante.

Uma declaração read (por exemplo read*, a) irá atribuir os valores de entrada aos elementos da matriz na mesma ordem, isto é, os valores serão colocados primeiro na primeira coluna, depois na segunda coluna, terceira coluna e assim por diante.

- ## Mudando a ordem

Esta ordem pode ser controlada (alterada) usando-se a funções intrínsecas: **transpose** ou **cshift**.

Exemplos:

Considere a matriz a, uma matrix 3x3, contendo os seguintes valores:

a(1,1)	a(1,2)	a(1,3)		1	4	7
a(2,1)	a(2,2)	a(2,3)	=	2	5	8
a(3,1)	a(3,2)	a(3,3)		3	6	9

então as declarações:

```
print*, "elemento                =", a(1,2)

print*, "seção                   =", a(:,1)

print*, "sub-matriz              =", a(:2,:2)

print*, "matriz inteira          =", a

print*, "transposta da matriz  =", transpose(a)
```

irão produzir as seguintes saídas:

```
elemento                = 4

seção                   = 1 2 3

sub-matriz              = 1 2 4 5

matriz inteira          = 1 2 3 4 5 6 7 8 9

transposta da matriz    = 1 4 7 2 5 8 3 6 9
```

48 – Reshape

Entrada de Dados numa Matriz

Um construtor de matriz é uma das formas utilizada pelo Fortran 90/95 para entrar com dados em um vetor (uma matriz de rank 1). Por exemplo:

```
.....
integer, dimension(4) :: du
.....
du = (/ 1, 3, 6, 7 /)
.....                          cria o vetor:  du(1)=1  du(2)=3  du(3)=6  du(4)=7
```

Quando for necessário entrar com dados em uma matriz (rank>1) pode-se utilizar um construtor de matriz, mas neste caso tem-se que usar a declaração reshape.

RESHAPE

A forma geral é: reshape (dados, shape)

onde:

dados	dados para a matriz Pode ser a própria atribuição de valores da matriz (o construtor de matriz)
shape	diz a forma com que a matriz deve ser construída

Exemplos

```
.....
integer, dimension(9)    :: v
integer, dimension(3,3)  :: m

!entra com os valores no vetor
v=(/1, 1, 1, 2, 2, 2, 3, 3, 3/)
.....
m = reshape( v, (/3,3/) )
```

pega o vetor
v(1)=1
v(2)=1
v(3)=1
v(4)=2
v(5)=2
v(6)=2
v(7)=3
v(8)=3
v(9)=3 ou seja: 1, 1, 1, 2, 2, 2, 3, 3, 3

e coloca na matriz m(3,3)

```
m(1,1)  m(1,2)  m(1,3)        1  2  3
m(2,1)  m(2,2)  m(2,3)   =    1  2  3
m(3,1)  m(3,2)  m(3,3)        1  2  3
```

Também poderíamos escrever

```
m=reshape( (/ 1,1,1,2,2,2,3,3,3 /), (/3,3/) )
```

outro exemplo:

```
.....
integer, dimension(2,3)  :: m
.....
.....
m=reshape( (/2, 3, 4, 5, 6, 7/) , (/2,3/) )
```

gera a matriz

```
2  4  6
3  5  7
```

49 – Matriz local

O Fortran identificam as variáveis usadas num programa como: locais ou globais, isto é, variáveis que são usadas somente dentro do programa e aquelas que são compartilhadas com outros subprogramas.

Uma variável local (**uma matriz local, também**) é aquela que só pode ser usada dentro do subprograma (função, sub-rotina, modulo) em que está definida.

Uma matriz declarada dentro de um subprograma (função, sub-rotina, modulo), que pretende ser usada localmente não pode ser identificada com a declaração **intent**.

Por exemplo, no **programa resp** é feita uma chamada à **sub-rotina** denominada **sub** que está colocada dentro do **módulo ele1**.

Observe que dentro da sub-rotina **sub** é feita a definição do **vetor x** e da **matriz k**. Observe também que na definição do vetor e da matriz, a extensão (tamanho de cada dimensão) é feita usando-se um valor que será passado à sub-rotina como argumento. Esta é uma forma antiga de trabalhar. O Fortran 95 possuí forma mais adequada para realizar a mesma tarefa.

Observe, também, que estas duas matrizes são locais. Não são compartilhadas, portanto não utilizam o atributo intent na sua definição.

Por sua vez, a **variável** real **d** (definida na sub-rotina) está sendo usada para representar a **variável m** (definida no programa principal) **não é uma variável local**. Observe a utilização do atributo intent na sua definição. Ela está sendo compartilhada, isto é, a sub-rotina pode alterar o valor da variável m, que foi definida no programa principal.

programa principal:

```
program resp
  use ele1
  implicit none
  integer :: tam = 10
  real    :: m=5.7
.....
.....
call  sub(tam,m)
.....
.....
end program resp
```

Módulo

```
module ele1
.....
.....
contains

    subroutine sub(len,d)
    implicit none
    integer, intent(in)                :: len
    real, intent(inout)                :: d
    integer, dimension(1:len)          :: x
    real, dimension(len-1:len+1)       :: k
    .....
    .....
    end subroutine sub
end module ele1
```

50 – Forma Assumida de Matriz

Uma matriz que é usada como um argumento mudo num subprograma (função, sub-rotina) é chamada de matriz de forma assumida. Por quê?

Porque ao se passar a matriz (como um argumento) para uma função ou sub-rotina, é necessário que as informações sobre a matriz (tipo, rank, extent, shape, size) também sejam passadas ao subprograma.

Uma interface explícita é o mecanismo utilizado para se fornecer (passar) todos os detalhes da matriz que é passada como argumento para um subprograma, tal como: tipo, rank, extent, shape, size. Estas informações são essenciais para que o subprograma possa trabalhar com a matriz.

Caso uma interface explícita não seja usada, as informações sobre a matriz tem que ser passados (fornecidos) ao subprograma por algum meio, por exemplo como argumentos. (este é o modo antigo de se tratar este problema, citado no item 45)

Uma forma alternativa à utilização de uma interface explícita é usar uma a **forma assumida de matriz**.

Toda matriz que é passada como argumento para um subprograma em que a dimensão (rank) da matriz é conhecida mas as extenções (extent) de cada dimensão não são, é o que chamamos de uma matriz de forma assumida.

- **Forma geral**

Para declarar uma matriz de forma assumida utilize: **:**

:	dois pontos Para cada dimensão expecifique somente o dois pontos Neste caso é assumido que limite-inferior = 1 Note que a forma (shape) é passada e não os limites inferior e superior da matriz
limite-inferior:	Neste caso o limite infrior é expecificado exemplo: **5:**

Ao passar matriz para um subprograma de preferência à utilização de uma matriz de forma assumida.

exemplos:

..... integer, dimension(-2:2) :: a	indica a matriz a(-2) a(-1) a(0) a(1) a(2)
se no subprograma a matriz for declarada como sendo subroutine ppp (y, ...) ..... integer, dimension (2:), intent(in) :: v .... end subroutine ppp	teremos uma matriz também com 5 elementos mas com a seguinte característica v(2) v(3) v(4) v(5) v(6)
Então poderia ter a seguinte maneira de referenciar a matriz No que passa a matriz e chama o subprograma ..... integer, dimension(-3:3) :: x .... call qualquer(x, -3)	O subprograma teria que conter as declarações referente a uma matriz de forma assumida, por exemplo: subroutine qualquer(y, linferior) integer, dimension (linferior:), intent(in) :: y .... end subroutine qualquer Neste caso a extensão da **matriz y** será a mesma que a da matriz usada como **argumento mudo x**

Observe:

A extensão da matriz não é passada.

Se o limite inferior for diferente, então isto deve ser considerado quando se for referenciar os elementos da matriz

Existem 3 funções intrínsecas que podem ser usadas para se obter informação sobre o tamanho (size) e a extensão (extent) de uma matriz

- **size(x)** retorna o número de elementos da matriz x
- **lbound(x)** retorna limite inferior da matriz
- **ubound(x)** retorna limite superiro da matriz

Exemplo

```
program matriz
implicit none
integer, parameter          :: li = 10
integer, parameter          :: ls = 50
integer, dimension(li:ls)   :: dd
real, dimension(1:li)       :: vv
.................
call sub(dd, vv, li, z)
.................
contains

  subroutine sub(x, y, linferior, k)
       integer, intent(in)                  :: linferior
       integer, intent(in)                  :: k
       integer,dimension(linferior:), intent(in):: x
       real, dimension(:), intent(out)         :: y
       ..............
       ..............
  end subroutine sub

end program matriz
```

51 – Funções Intrínsecas de Matriz

Algumas funções já existem diretamente no Fortran 90/95, que permitem trabalhar com a matriz de forma mais fácil.

- **A funções para se trabalhar com matrizes são:**

Funções	Significado
Aritmética	
dot_product(vector_a, vector_b)	produto escalar
matmul(array_a, array_b)	multiplicação de matrizes
transpose(array)	transposta da matriz
Redução	
all(mask)	verdadeiro se todos os elementos são verdadeiros
any(mask)	verdadeiro se qualquer elemento é verdadeiro
count(mask)	número de elementos verdadeiros
maxval(array)	valor do maior elemento da matriz
minval(array)	valor do menor elemento da matriz
product(array)	produto dos elementos da matriz
sum(array)	soma dos elementos
Indagação	
allocated(array)	verdade se a matriz foi alocada
lbound(array[,dim])	valor do limite inferior
shape(source)	retorna a forma da matriz
size(array[,dim])	tamanho da matriz
ubound(array[,dim])	valor do limite superior
Construção	
merge(tsource,fsource,mask)	seleciona uma de duas alternativas
pack(array, mask[,vector])	escreve vetor com elementos verdadeiro do teste
unpack(vector,mask, filed)	obtém valores do vetor que seja verdadeiro ao teste
reshape(source,shape[,pad][,order])	muda a forma de uma matriz
spread(source,dim,ncopies)	faz ncopias da matriz

Manipulação

cshift(array,shift[,dim])	permutação cíclica
eoshift(array,shift[,boundary][,dim])	permutação não cíclica

Localização

maxloc(array[,mask])	retorna posição do elemento com maior valor
minloc(array[,mask])	retorna posição do elemento com menos valor

Exemplos

Dada a matriz xm(3,3)

```
50  47  58
38  43  90
66  59  11
```

```
a = maxval(xm)        ! a = 90, maior valor
a = sum(xm)           ! a = 462, soma dos elementos
a = maxloc(xm)        ! a = (/2, 3/) posição: linha  2, coluna 3
a = transpose(xm)     ! retorna a transposta da matriz xm
a = ubound(xm,1)      ! a = 3
```

52 – Unidades de Programas

É possível escrever um programa Fortran como uma única unidade. Entretanto isto não é uma boa prática de programação e deve ser evitado.

A moderna técnica diz que se deve escrever um programa em módulos (unidades menores) que são mais fáceis de serem tratadas do que um programa monolítico grande.

Cada unidade de programa, que compõe o programa final, deve realizar uma tarefa (idealmente uma tarefa única) porque esta forma permite que a unidade de programa seja escrita, compilada e testada de forma independente.

Há (pelo menos) duas razões principais para se dividir e subdividir um programa:

- **Modularidade**
  dividir um programa grande em pedaços (segmentos) onde cada pedaço pode ser trabalhado separadamente

- **Esconder dados**
  proteger os dados e as informações de outras partes do programa de forma que eles não possam ser afetados indevidamente

O Fortran 95 utiliza três (3) tipos de unidades de programa

> Na realidade possuí 4 mas uma delas (declaração data block) está marcada como obsoleta, portanto não deve ser utilizada em novos programas.

- **programa principal**
  (main program) é o local onde a execução do programa inicia e deve acabar. Pode conter chamadas à subprogramas (funções e sub-rotinas).

- **módulo**
  a unidade de programa utilizada para conter declarações (principalmente as globais) e os subprogramas que ficarão disponíveis à outras unidades

- **subprogramas externo**
  são funções e sub-rotinas escritas em arquivos separados, como uma unidade a parte. O Fortran 95 não privilegia esta forma. Preferencialmente utilize os subprogramas dentro de módulos.

Um programa completo em Fortran 95 tem que ter um programa main (ao menos ele). Um programa completo usualmente é um programa principal (simplesmente chamado de programa) e um ou mais módulos que conterá(ão) um ou mais subprogramas (funções e sub-rotinas).

• Subprogramas

Subprogramas usualmente são as funções e sub-rotinas, que também são denominadas de procedimentos. Os subprogramas são utilizados por outras unidade.

- **função**
  uma função (necessariamente) retorna um valor (ou resultado) e não altera os valores dos seus argumentos

- **sub-rotina**
  uma sub-rotina, retorna ou não um valo. Ela pode ou não alterar os valores dos argumentos. Normalmente ela é usada para realizar tarefas complexas

  Funções e sub-rotinas são também chamadas de procedimentos (procedures).

  As funções e sub-rotinas podem ser: internas e externas.

subprograma externo

```
subroutine xxx(a,b)
.....
.....
end subroutine xxx
```

subprograma interno

```
program yyy
.....
.....
contains

subroutine xxx(a,b)
.....
.....
end subroutine xxx

end program yyy
```

Um subprograma externo é armazenado num arquivo independente e por isto pode ser compilado e testado separadamente das outras unidades do programa.

Os subprogramas internos, obviamente, não, pois estão armazenados (colocadas) dentro de outras unidades.

A figura 6 mostra um esquema gráfico das unidades.

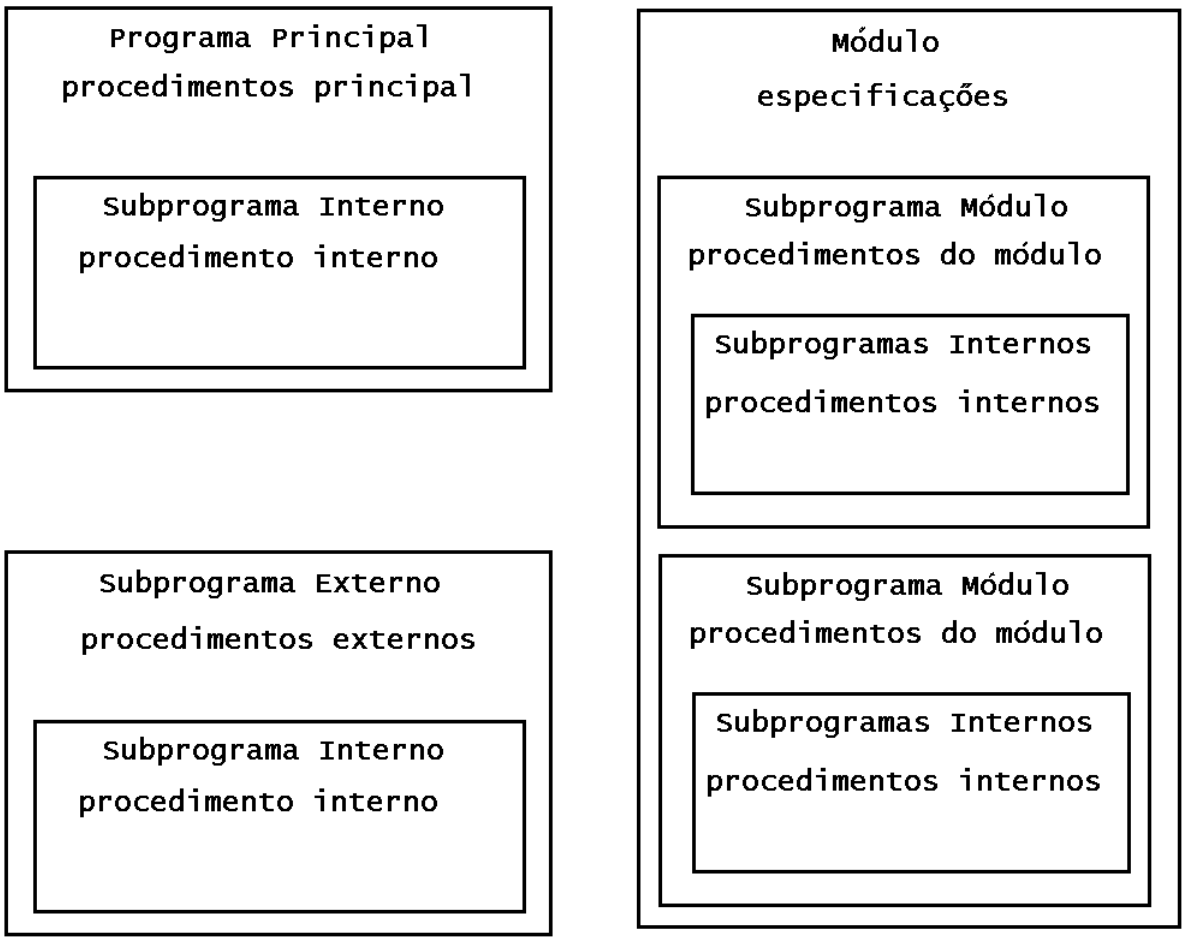

Figura 6 - Programa principal, subprograma externo e módulo

Observe que o programa principal (esquematicamente) está armazenado num arquivo. O módulo em outro arquivo e o subprograma externo em outro arquivo. Portanto temos 3 arquivos, cada um contendo uma unidade de programa.

Conforme dito, os programas completos de Fortran 95, deverão utilizar um arquivo para o programa principal, que preferencialmente não utilizará subprogramas internos e um (ou mais) arquivos para os módulos, este(s) sim conterão os subprogramas (funções e sub-rotinas) utilizados.

Um programa principal (ou mesmo um subprograma) pode utilizar os subprogramas disponibilizados nos módulos. Uma chamada de um subprograma, entretanto, deve ser feita conforme o tipo do subprograma.

- **sub-rotina**
  uma sub-rotina é utilizada (acessada) através do uso da declaração **call**

- **função**
  a função, por sua vez, é utilizada simplesmente fazendo-se uso do nome da função

HOST

Um conceito importante:

Chamamos de Host (hospedeiro) a unidade de programa que contém subprogramas.

Então um módulo que contenha sub-rotinas e funções é o host delas. Módulos são as unidades de programas especialmente adequada para serem usadas como hosts.

O conceito de host (parente, pai ou hospedeiro) será bastante útil quando for tratado a problema das variáveis globais e locais.

53 – Programa Principal

Todo programa completo em Fortran tem que ter um e somente um programa principal, que geralmente contem chamadas a subprogramas.

- **Um programa principal tem a seguinte forma:**

program prog_nome	program prog_nome
[parte especificação]	[parte especificação]
[parte execução]	[parte execução]
end program prog_nome	contains
	[subprogramas]
	end program prog_nome

- a declaração **program** é opcional, mas recomendada. Se usada é necessário utilizar um **prog_nome** para o programa
- **prog_nome** (o nome dado ao programa) tem que ser um nome válido em Fortran 95
- a declaração **end** é obrigatória e serve a 2 propósitos:
  - identificar para o compilador onde o programa principal acaba
  - durante a execução, ao encontrar esta declaração o programa termina a execução
- Deve-se dar preferência ao uso da declaração **end program prog_nome**
  se a declaração program foi usada a declaração end program prog_nome é obrigatória, no lugar de end

Um programa principal pode conter subprogramas (ou procedimentos = funções e sub-rotinas), como mostrado na figura 04 e na tabela acima. Neste caso é obrigatório o uso da declaração **contains**.

A declaração contains indica a presença de um ou mais procedimentos (subprograma interno) na unidade em questão.

Exemplos

end	***Isto é que é um programa mínimo!***
	Um programa principal contendo somente uma linha e uma declaração !
	O que ele faz? **Nada** ! **Nada mesmo** !!!!
	Mas é um programa perfeitamente válido em Fortran

Agora veja o programa nada

```
program nada
!============================
! nada.f95  programa que não faz nada
! autor: Anibal L. Pereira
! escrito: 21/04/2002
!============================
    ! não faz nada

end program nada
```

Este é um programa muito melhor !

Não acha !!!!

A diferença é que ele é muito mais explicativo (lembra do preceito clareza?)

Aqui seguiu-se as normas mínimas de uma boa programação.

```
program ola
!============================
! para dar destaque ao que interessa
! será suprimido o cabeçalho
!============================

   print*, "ola"

end program ola
```

O programa ola é o programa principal

```
program ola2
  character(len=10) :: nome
  print*, " qual e seu nome ? "
  read*, nome
  call escreve(nome)

contains

   subroutine escreve(w)
     character(len=10), intent(in) :: w

     write(unit=*, fmt="(a, a)"), "ola", w
   end subroutine escreve

end program ola2
```

O programa ola2 é um programa principal que contêm um subprograma

contêm um subprograma interno

54 – Declaração Stop

Para terminar a execução de um programa pode-se executar a declaração **stop**, ou seja, quando o programa encontra a declaração stop, ele termina a execução do programa.

A declaração **stop** para a execução do programa, não importando onde ela esteja. Se for usada dentro de um subprograma, a declaração **stop** não termina com a execução do subprograma em questão mas, acaba com a execução do programa principal.

Para o término somente do subprograma utilize a declaração **return**.

É possível usar mais de uma declaração stop espalhada pelo programa.

Entretanto a boa norma de programação recomenda o uso parcimonioso (limita ao máximo o uso) da declaração **stop**.

A declaração stop tem a forma

stop

Exemplos:

```
program para_1
  character(len=10) :: nome
.....                                  Programa só continua execução
.....                                  se a variável caractere
  print*, " qual e' seu nome ? "       for igual a Paulo
  read*, nome
  if (nome /= "Paulo") then
    print*, "Voce nao e' o Paulo"
    stop
  end if
.....
.....
end program para_2
```

```
program para_2
  character(len=10) :: nome
  integer           :: p
.....                                  O Programa para a execução
.....                                  quando encontra a declaração stop
  print*, " qual e' seu nome ? "
  read*, nome                          Mesmo que ela esteja dentro da sub-rotina
  if (nome /= "Paulo") then
    p = 1
  end if
.....
call escreve(nome, p)
.....

contains

   subroutine escreve(w, x)
     character(len=10), intent(in) :: w
     integer, intent(in) :: x

     if ( p = = 1) then
     write(unit=*, fmt="(a, a)"), "ola", w
     write(unit=*, fmt="(a)"),&
        "Onde esta' o Paulo ? "
     end if
      stop
   end subroutine escreve

end program para_2
```

---

55 – Declaração Return

A declaração return interrompe a execução do subprograma na qual ela está colocada. Ela não entrrompe o programa principal.

Ela promove o retorno do comando para a unidade de programa que fez a chamada do subprograma.

Sua sintaxe é:

return

Exemplos

```
Program retorna
  implicit none
  integer :: i=10, j

j = 20
  call acaba_antes(i, j)
  print*, j

contains

     subroutine acaba_antes(k, l)
     implicit none
     integer,intent(in)  :: k
     integer,intent(out) :: l
     integer             :: m

      do m = 1, k
         l = m
         if(l = = 5) return
      end do
     end subroutine acaba_antes

end program retorna
```

Mostra uso da declaração **return** para interromper um subprograma

O programa principal recebe de volta o valor 5 e o escreve na tela

---

```
programa aborta
implicit none
  integer :: i=10, j

j = 20
  call acaba(i, j)
  print*, j

contains

     subroutine acaba(k, l)
     implicit none
     integer,intent(in)  :: k
     integer,intent(out) :: l
     integer             :: m

      do m = 1, k
         l = m
         if(l = = 5) stop
      end do
     end subroutine acaba
```

Uso da declaração stop no lugar da declaração return

Nenhuma saída é mostrada
o programa principal para de ser executado

A declaração stop interrompe a execução da sub-rotina e do programa principal

56 – Subprogramas Externos

Subprogramas externos são chamados pelo programa principal (ou de outro lugar) para a realização de uma tarefa bem definida.

Os subprogramas externos são as funções e as sub-rotinas que estão escritas e guardadas em arquivos separados do programa principal e não estão contidos dentro de módulos.

A principal vantagem do uso de subprogramas externos decorre do fato deles poderem ser escritos, compilados e testados em separado. Fora isto, a única razão para usá-los é de natureza organizacional, isto é, para poder armazená-los em arquivos separados.

O Fortran 95 não recomenda o uso de subprogramas externos.

Uma maneira muito mais prática e vantajosa é o uso de módulos contendo os subprogramas que forem necessários.

- **Um subprograma externo tem a seguinte forma:**

```
subroutine nome(lista-arg)                    este programa fonte deve ser escrito
 [parte especificação]                        em um arquivo separado
 [parte execução]

end subroutine nome
```

```
function nome(lista-arg)                      este programa fonte deve ser escrito
 [parte especificação]                        em um arquivo separado
 [parte execução]

end function nome
```

Para o Fortran 95

Recomenda-se o uso de módulos que conterão os subprogramas necessários.

Os módulos são guardados em arquivos separados (externos) ao programa principal.

```
module nome                                   O módulo pode conter um ou mais subprogramas
 [parte especificação]
                                              Todos eles (módulo+subprogramas) estão contidos num
contains                                      arquivo
    [subprogramas]
end module nome
```

57 – Subprograma Internos

Subprogramas internos são aqueles colocados dentro do programa principal ou dentro de módulos.

- **Um subprograma interno tem a seguinte forma:**

```
program nome

  [parte especificação]                    tudo contido em um único arquivo
  [parte execução]

contains

   subprogramas internos

end program nome
```

```
module nome
                                           tudo contido em um único arquivo
  [parte especificação]
  [parte execução]

contains

   subprogramas internos

end module nome
```

O uso da declaração **contains** não é opcional.

ATENÇÃO: Os subprogramas internos, aqueles contidos num programa principal ou nos módulo, automaticamente possuí acesso a todas as "entidades" do host.

É possível a um subprograma chamar outro subprograma interno que esteja contido no host ou no módulo que o host tem acesso.

58 – Módulos

Módulos são usados para gerar dados globais e são também importantes para conter dados derivado, além claro de ser o local ideal para se colocar subprogramas.

As principais vantagens do uso de módulos são:

- **declarar objetos globais**
- **declarar procedimentos** (funções e sub-rotinas)
- capacidade de **controlar acesso a unidades de programas**
- capacidade de "empacotar", isto é, **colocar junto um conjunto de subprogramas**

Um módulo tem a seguinte forma:

```
module nome

  [parte especificação]
  [parte execução]

contains

   subprogramas internos

end module nome
```

a forma mais simples é:

```
module nome

  [parte especificação]

end module nome
```

O módulo é acessado usando-se a declaração USE. Isto é, no programa que necessita ter acesso as variáveis e subprogramas contidos num módulo, deve-se usar a declaração **use**.

A declaração use deve aparecer no início do programa ou unidade de programa que for usar o módulo.

Exemplo

programa guardado num arquivo próprio

```
program usa_soma
use fsoma
implicit none
real     :: a, b, c, theta

a = 1.0
b = 5.0
c = soma(a, b)
theta = pi * sin(a)
print*, "theta=", theta, "a=", a

end program usa_soma
```

Este programa principal é bastante simples

define 4 variáveis reais: a, b, c e theta

calcula a soma das variáveis a e b (**c=a+b**)

calcula o valor theta e então imprime os resultados

Observe:

1) o uso da declaração use no programa principal para ter acesso às variáveis e subprogramas que foram declarados públicos no módulo
2) a adição de a e b é feita pela função soma, que está no módulo
3) a constante com nome pi também está definida e acessível no módulo

---

módulo guardado num arquivo separado

```
module fsoma

real, public, parameter:: pi=3.14159

public :: soma


contains

  function soma(x,y) result(s)
    real, intent(in) :: x, y
    real             :: s

    s = x + y

  end function soma

end module fsoma
```

Este módulo é bem simples

contem a definição de uma **constante com nome** real chamada **pi**

e uma função chamada **soma**

Observe que ambas foram tornadas públicas para que o programa que faça uso do módulo tenha acesso a elas.

Neste módulo tudo foi tornado público (acessível)

59 – Associação de Argumentos

As principais vantagens do uso de subprograma (função e sub-rotina) são:

- possibilidade de se testar independente cada subprograma
  as funções e sub-rotinas podem ser escritas e testadas de forma independente

- reusabilidade dos códigos
  ou seja a possibilidade de se usar e reutilizar quantas vezes forem necessária os códigos fontes dos subprogramas

- isolamento dos dados
  as variáveis e constantes (os dados) definidos nos subprograma não são acessíveis fora deles, a menos que eles sejam explicitamente disponibilizados, portanto os dados são isolados do programa principal

Uma forma bastante segura e prática dos subprogramas se comunicarem como programa principal, isto é, receber e enviar informações é através dos argumentos do subprograma (que também são chamados de parâmetros).

Os itens colocados dentro de parênteses que seguem o nome da função ou sub-rotina são chamados de argumentos e são usados para transmitir informações relevantes entre o programa e o subprograma.

A principal vantagem do uso dos argumentos é o total controle do argumento pelo uso do atributo intent.

Outra forma de se compartilhar dados entre o programa principal e os subprogramas é através da declaração use, que um mecanismo que permite disponibilizar dados entre unidades de programa.

- **Argumentos Reais e Formais**

O uso de subprogramas implica necessariamente na utilização de 2 ambientes diferentes. São eles:

- o programa que chama ou utiliza o subprograma
- e o subprograma a função ou sub-rotina utilizada

As variáveis ou constantes definidas no programa, que são utilizadas na chamada do subprograma (função e/ou sub-rotina) como argumentos são os **argumentos reais** (actual arguments). As variáveis utilizadas como argumentos no subprograma são os **argumentos mudos (**ou **argumentos formais** ou **parâmetros)** [dummy arguments].

```
program principal
  .....
  .....
  call integral(a, b, c)
  .....
  .....
  .....
contains

   subroutine integral(x, y, z)
   .....
   end subroutine sub

end program principal
```

Neste exemplo utiliza-se uma sub-rotina interna chamada integral que utiliza 3 argumentos formais: x, y e z

os argumentos dentro do **programa** são os
argumentos reais: a, b, c

os argumentos dentro da **sub-rotina** são os
argumentos formais: x, y, z

observe a figura 06

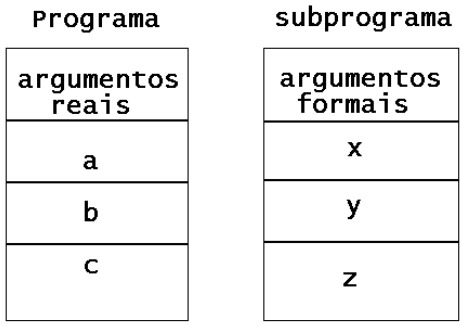

Figura 6 - Argumentos reais e argumentos formais

- **Intenção dos argumentos**

No Fortran 95, os subprogramas têm que declarar o tipo de argumento que utilizam, isto é, tem que identificar qual a intenção do argumento. Se o argumento intenciona ser somente de entrada, somente de saída ou de entrada e saída:

- **in**
  intent(in) indica que o argumento é recebido pelo subprograma e ao final da execução do subprograma o valor do argumento real associado não será (não pode ser) alterado. Dentro do subprograma alterações do valor do argumento formal não afetam o valor do argumento real associado.

- **out**
  intent(out) indica que o argumento não vem de fora (do programa que chama). O valor do argumento formal será calculado pelo subprograma e depois passado para fora, para o programa que chamou a função ou sub-rotina

- **in out**
  intent(inout) indica que o valor do argumento formal é recebido de fora (do programa que chama) e dentro do subprograma este valor pode sofrer alteração que será enviada para o programa que chamou a função (repassado de volta)

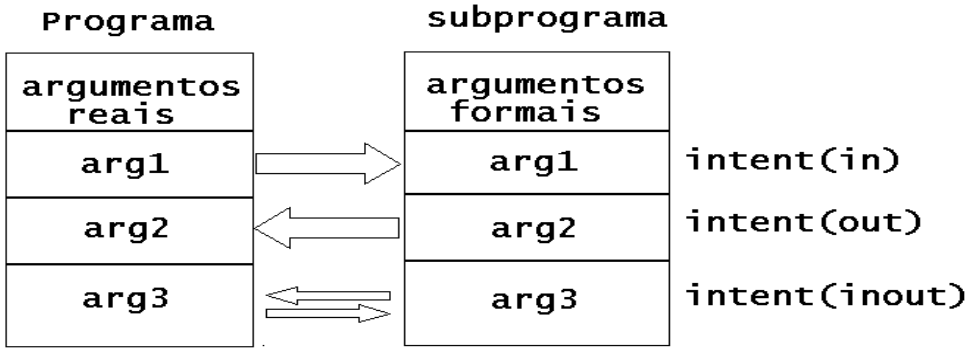

Figura 7 - Intenção dos Argumentos

- **Variáveis locais**

As variáveis e constantes que são locais, isto é, que existem somente dentro do subprograma não podem ser declaradas com o atributo intent

```
program principal
  integer :: a, b, c, d
  .............
  call sub(a, b, c)
  .............
contains

  subroutine sub(x, y, z)
    integer, intent(in)    :: x
    integer, intent(out)   :: y
    integer, intent(inout) :: z
    integer                :: d
  .............
  end subroutine sub

end program principal
```

Observe o isolamento das variáveis

a variável **d** foi definida no programa principal

uma variável chamada **d** foi declarada no subprograma

a variável **d** declarada **no subprograma é local**

por isto pode ter o mesmo nome da variável d no programa principal. Elas são diferentes, elas estão isoladas

Atente ao fato da variável **d no subprograma** não foi declarada quanto a sua intenção portanto **é uma variável local**

Associabilidade

- **Quanto ao Tipo**

Os argumentos reais têm que concordar quanto ao tipo, isto é:

argumento real	associação	argumento formal
inteiro	⟵ ⟶	inteiro
real	⟵ ⟶	real
complexo	⟵ ⟶	complexo
lógico	⟵ ⟶	lógico
caractere	⟵ ⟶	caractere
tipo derivado	⟵ ⟶	tipo derivado

- **Quanto a Posição**

A posição dos argumentos formais define a associação com os argumentos reais.

Definindo-se a sub-rotina poli com os seguintes argumentos formais: x, y, z	**argumentos reais: a, b, c**	**argumentos reais: a, b, c**
subroutine poli(**x, y, z**)	call poli(**a, b, c**)	call poli(**b, c, a**)
	a ⟶ x b ⟶ y c ⟶ z	b ⟶ x c ⟶ y a ⟶ z

60 – Atributo Intent

Todo subprograma (sub-rotina e função) em Fortran 95 têm que declara a intenção de seus argumentos. Os argumentos podem ser:

- somente de entrada no subprograma - **intent(in)**
  o valor será recebido pelo subprograma. Não poderá ser alterado dentro do subprograma

- somente de saída do subprograma - **intent(out)**
  o subprograma irá passar para a unidade de programa que o chamou um valor através deste argumento

- entrada e saída no subprograma - **intent(inout)**
  o argumento é recebido, trabalhado (alterado ou não) e depois passado de volta para o programa que chamou o subprograma

  Uma regra básica para os argumentos de um subprograma é: **declare todos os argumento quanto a sua intenção**.

  A figura 8, identifica de forma visual as intenções dos argumentos.

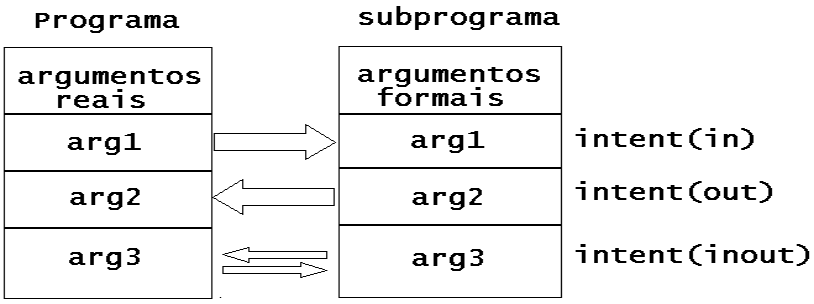

Figura 8 - Intenção dos Argumentos

Exemplo:

```
program teste
 use prog
 implicit none
 real   :: v = 2.5, theta = 30.5
 integer:: h_max = 1.0
 ..............

 call n_altura(v, theta, h_max)

 ..............
end program teste
```

Observe que o módulo prog disponibiliza 3 sub-rotinas, mas o programa principal só utiliza uma sub-rotina

Isto é perfeitamente legal

```
module prog
  implicit none
  public:: area, hipotenusa, n_altura

contains
  subroutine  area(a, b, c, an, bn, cn)
    real, intent(in) :: a, b, c
    real, intent(out):: an, bn, cn
    ..............
  end subroutine area

  subroutine hipotenusa(a, b, c)
    real, intent(in) :: a, b
    real, intent(out):: c
    ..............
  end subroutine hipotenusa

  subroutine n_altura(v1,theta1,h)
    real, intent(in)    :: v1
    real, intent(inout) :: theta1
    integer, intent(out):: h
      theta1 = theta1/2.0
      h =((v1**2)*sin(2*theta1))/10.0
  end subroutine n_altura
end module prog
```

A associação dos argumentos reais com os argumentos formais deve respeitar:

- **intent(in)**
  se o argumento real for uma **expressão**
  ela é primeiro avaliada, guardada numa variável provisória e então passada ao subprograma
  se o argumento real for uma **variável**
  a variável é passada diretamente para o subprograma
  se o argumento real for uma **constante**
  a constante é passada diretamente para o subprograma

- **intent(out)**
  este argumento tem que ser uma **variável**

- **intent(inout)**
  este argumento tem que ser uma **variável**

```
programa um
 use ppp
 implicit none
 integer :: a, b, c
 ..............
 ..............
 call dois(a,c*b,20,b,c)
 ..............
 ..............
end program um
```

```
module ppp
  implicit none
  public :: dois

contains

  subroutine dois(u, v, w, x, z)
    integer, intent(in):: u
    real, intent(inout):: v     ← errado: tem que ser variável
    integer, intent(out) :: w   ← errado: tem que ser variável
    integer, intent(inout):: x
    integer, intent(out)  :: z
    ..............
    ..............
  end subroutine dois

end module ppp
```

Os argumentos têm que concordar quanto ao tipo.

```
programa tp
 use ff
 implicit none
 integer:: a, c
 real   :: b
 characeter(len=1):: l
 logical:: etd
 ..................
 ..................
 call err(a, a*c, l, etd, c)
 ..................
 ..............
end program tp
```

```
module ff
 implicit none
 public :: err

contains

 subroutine err(u, v, w, x, z)
   integer, intent(out):: u
   integer, intent(in) :: v
   real, intent(inout) :: w              ← errado no tipo
   logical, intent(in) :: x
   character(len=2), intent(out):: z  ← errado no tipo
   ......................
   ......................
 end subroutine err

end module ff
```

A associação dos argumentos reais com os argumentos formais se dá pela posição que os argumentos reais ocupam na lista.

61 – Sub-rotina

A sub-rotina é uma das entidades mais importantes do Fortran devido a sua grande flexibilidade. Ela não obriga, como a função obriga, o retorno de um valor para o programa que utiliza a sub-rotina.

Uma sub-rotina pode ser usada para as mais diversas tarefas, por exemplo: cálculos, atividades de entrada e saída de dados, preparação de ambiente de trabalho, etc...

Hoje em dia, com o Fortran 95, muitas das tarefas feitas antes pela sub-rotina externas agora passaram a ser melhor realizadas pela sub-rotinas colocadas em um módulo.

Os conceitos básicos que se aplica, às sub-rotinas são:

Uma sub-rotina sempre inicia com a declaração subroutine e termina com a declaração end subroutine.

```
subroutine nome_da_sub-rotina (arg1, arg2, ..., argn)

  [parte especificação]

  [parte execução]

end subroutine nome_da_sub-rotina
```

- após o nome da sub-rotina, entre parênteses, é colocada a lista de argumentos formais, separados por vírgulas
- uma sub-rotina pode ser externa (pouco usada) ou interna (no programa principal ou em um módulo)

Uma sub-rotina não necessita, não tem a obrigatoriedade de, usar argumentos formais e portanto pode ser escrita como

```
subroutine nome(  )

  [parte especificação]
  [parte execução]

end subroutine
```

os parênteses podem ser escritos, para deixar claro que a lista de argumentos formais está mesmo vazia, não foi um possível esquecimento.

- **Regras básicas**
  - Uma sub-rotina é um conjunto completo de declarações. Usualmente ela recebe alguma entrada (input) de fora através dos argumentos formais, realiza algumas ações (cálculos, etc) e usualmente retorna (ou não) algum resultado via seus argumentos formais
  - a utilização da sub-rotina, ou seja, a chamada a sub-rotina é sempre feita via uma declaração **call**
  - quando o fluxo de execução das declarações na sub-rotina chega à declaração end subroutine os resultados armazenados nos seus argumentos formais de saída são passados de volta para quem chamou a sub-rotina. O mesmo ocorre quando a declaração **return** é encontrada

62 – Chamando uma Sub-rotina

A declaração call é usada para chamar uma sub-rotina. A sintaxe é:

call nome_da_sub-rotina (arg1, arg2, ..., argn)

Os argumentos não são obrigatórios para a sub-rotina. Se a sub-rotina não utilizar argumentos formais, quando ela for usada (chamada) os argumentos também não serão usados, entretanto é recomendável que se utilize os parênteses para deixar bem claro a inexistência dos argumentos.

call nome_da_sub-rotina ( )

Quando a sub-rotina utiliza argumentos, eles são identificados em reais e formais. Os argumentos reais são aqueles usados dentro do programa que utiliza a sub-rotina, isto é, que aparecem na chamada da sub-rotina. Os argumentos formais são os que são utilizados dentro da sub-rotina, isto é, aqueles usados quando se escreveu a sub-rotina.

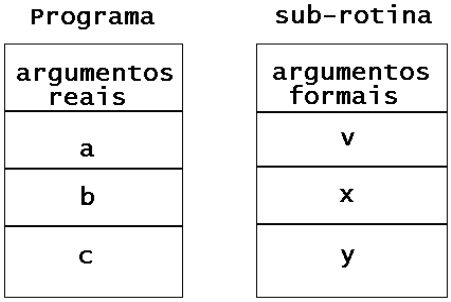

Figura 9 - Argumentos Reais e Formais

As sub-rotinas devem identificam claramente os seus argumentos, isto é, declarar explicitamente seus argumentos quanto a sua intenção. Esta declaração de intenção é feita usando o **atributos intent**.

Os argumentos podem ser:

- somente de entrada na sub-rotina - intent(in)
- somente de saída - intent(out)
- ou de entrada e saída - intent(inout)

Importante

- quantidade de argumentos
  devem ser iguais tanto na lista de argumentos utilizados no programa (argumentos reais) quanto na lista da sub-rotina (argumentos formais), a menos que se use argumentos opcionais

- tipos
  o tipo de cada argumentos, nas duas listas, devem ser iguais entre si

- característica dos argumento
  argumentos reais associados a argumentos formais para as intenções de entrada-e-saída (inout) e de saída (out) só podem ser variáveis

Exemplos

```
program exemplo1
  implicit none
  integer a, b, c
  ......................
  ......................
  call maior(a *5, b, c)
  print*, "maior valor=", c
  ......................

contains

 subroutine maior(v, x, y)
   integer, intent(in) :: v, x
   integer, intent(out):: y

   if (v > x) then
      y = v
   else
      y = x
   end if
 end subroutine maior

end program exemplo1
```

```
program exemplo2
  implicit none
  ..................
  ..................
  call ola( )
  ..................
  ..................

contains

  subroutine ola( )

      print*," ola´, tudo bem ? "

  end subroutine ola

end program exemplo2
```

63 – Argumentos Opcionais

Quando se chama um subprograma (função ou sub-rotina) os argumentos reais e formais sempre devem concordar em:

- número
- tipo
- ordem

Esta regra, entretanto, pode ser alterada, com o uso de uma interface explicita. Quando uma interface explicita é utilizada o compilador conhece as informações necessárias sobre as entidades usadas nos subprogramas e com isto ele pode realizar procedimentos que possibilitam encontrar inconsistências.

Módulos e subprogramas internos têm sempre uma interface explicita, por default. **Esta é uma das grandes vantagens do uso de módulo** (e de subprogramas internos). Por isto, os programas externos não são recomendados no Fortran 95.

Entretanto, caso seja utilizado um subprograma externo, é necessário que se escreva uma interface explicita. Por default, um programa externo não tem como disponibilizar as informações para o programa principal, a menos que se utilize um bloco interface. Sempre faça uso de um bloco interface quando utilizar subprogramas externos.

A forma geral de um bloco interface externa é:

Um bloco interface para um subprograma é especificado duplicando-se as informações do cabeçalho do subprograma, e portanto tem a seguinte forma:

interface
corpo-da-interface
end interface

onde

corpo-da-interface é uma cópia exata das especificações do subprograma
seus argumentos formais
suas especificações
e a declaração end

Cada bloco interface consiste da declaração inicial subroutine ou function do subprograma externo correspondente, as especificações correspondentes de seus argumentos formais e de suas variáveis locais. Terminando com a declaração final end subroutine ou end function do programa externo.

Exemplo:

```
program int
 implicit none

 interface
   real subroutine res(a, b, c)
     real, intent(out)  :: a
     integer, intent(in):: b
     real, intent(inout):: c
   end subroutine res
 end interface
 ..................
 ..................
 call res(x,i,y)
  ..................
end program int
```

Programa principal

usuário da **sub-rotina externa res**

Observe a utilização do bloco interface (interface block)

Sub-rotina externa ⇨

```
Subprograma escrito num arquivo separado do
programa principal


   subroutine res(a, b, c)
      real, intent(out)  :: a
      integer, intent(in):: b
      real, intent(inout):: c

      a = 2.0**b
      c = float(b)/a

   end subroutine res
```

- keyword argumet

Conforme vimos os argumentos reais devem concordar em número, tipo e posição com os argumentos formais. O Fortran 95 permite que se especifique apenas alguns dos argumentos reais (diferir em número) ou então colocá-los em ordem diferente (posição diferente), desde que se utilize uma interface explicita no programa que chama o subprograma.

Este é um dos motivos para o uso de um bloco interface para um subprograma externo. Outro seria a possibilidade de se utilizar argumentos do tipo matriz de forma assumida.

Observe que o uso de um bloco interface é desnecessário quando se colocar os subprogramas dentro de módulos e o tornamos disponível com a declaração use ou então quando se utiliza subprogramas internos.

Então, existindo uma interface explicita (bloco interface para subprogramas externos ou subprogramas colocados em módulos ou como subprogramas internos), é possível usar-se uma palavra-chave (keyword) junto com os argumentos para aumentar a flexibilidade de programação.

A vantagem do uso da palavra-chave (keyword) é :

- clareza
- possibilidade de alterar a ordem dos argumentos

Esta facilidade é executada usando-se a forma geral:

palavra-chave = argumento-real (keyword = actual_argument)

onde:

palavra-chave é o nome do argumento formal que será associado com o argumento real.

observou o uso de argumentos reais e formais?

Exemplo:

```
program ordem
  implicit none
  ..............
  interface
    function abc(x, y, z) result(k)
      real, intent(inout) :: x
      integer, intent(in) :: y
      integer, intent(out):: z
      real:: k
    end function abc
  end interface
  ..............
  velo = abc(10.0,20,30)
  ..............
end program ordem
```

Programa principal usando bloco interface

mesmo usando a interface explicita, o tipo, ordem e posição dos argumentos reais tem que obedecer as dos argumentos formais, porque não se utilizou nenhuma palavra-chave com os argumentos

então:

x recebe o valor 10.0
y recebe 20
z recebe 30

subprograma (função) externa guardada num arquivo separado

```
function abc(x, y, z) result(k)
  real, intent(inout) :: x
  integer, intent(in) :: y
  integer, intent(out):: z
  real:: k
  ..................
  ..................
end function abc
```

Alterando a ordem:

Agora observe como fica quando se utiliza as seguintes chamadas da função

`velo=abc(x=10.0,y=20,z=30)`	as palavras-chaves forma usadas todos os argumentos foram usados argumentos na mesma ordem os tipos das variáveis não podem divergir
`velo=abc(y=20,x=10.0,z=30)`	todos os argumentos foram usados argumentos fora de ordem os tipos das variáveis não podem divergir
`velo=abc(z=30,y=20,x=10)`	a ordem pode ser alterada quando se usa as palavras-chaves

Observe que em qualquer caso temos:

x = 10.0
y = 20
z = 30

observe como o uso das palavras-chaves torna muito mais claro (legível) o programa

```
.............                    .............
.............                    .............
velo=abc(10.0,20,30)             velo=abc(x=10.0,y=20,z=30)
.............                    .............
```

recomenda-se o uso das palavras-chaves em todos os casos

Tornando Opcional um argumento:

Uma interface explicita permite o uso de um atributo opcional. Argumento opcional que dizer a ausência deste argumento na chamada do subprograma

O argumento que pode ser omitido no programa tem que ser identificado com o atributo optional.

Exemplo:

```
program ordem
 .............
 interface
   function abc(x, y, z)result(local)
    real,intent(inout):: x
    integer,optional,intent(in):: y
    integer,intent(out):: z
    integer:: local
   end function abc
 end interface
 .............
 velo = abc(x=10.0,z=30)
 .............
end program ordem
```

um programa externo necessita do uso de um bloco interface

o argumento formal y foi marcada como opcional
por isto ele pode estar ou não na lista de argumentos reais.

omissão de y

velo = abc(x=10.0, z=30)

subprograma (função) externa guardada num arquivo separado

observe o uso da função intrínseca present para verificar se o argumento real correspondente está ou não presente na lista de argumentos passados ao subprograma

```
function abc(x, y, z)result(local)
   real,intent(inout) :: x
   integer,optinal, intent(in):: y
   integer,intent(out):: z
   integer:: local
   ...................
   if (present(y)) then
       local = y
   else
       local = 25
   end if
   ...................
 end function abc
```

64 – Função Intrínseca Present

A função intrínseca present testa se um argumento está ou não presente na lista de argumentos reais.

Sua sintaxe é:

present(A)

onde A é o nome do argumento que está sendo testado

Esta função testa a existência ou não do argumento na lista de argumentos reais. Se o resultado do teste for:

verdadeiro o argumento está presente (existe)

falso o argumento está ausente

65 – Atributo Save

O valor de todas as variáveis e matrizes locais se tornam indefinidos quando o subprograma é deixado.

A próxima vez que o subprograma for invocado as variáveis e matrizes locais podem ou não terem os mesmos valores que da última vez que foram chamados.

Por outro lado, quando os valores iniciais das variáveis e matrizes locais são atribuídos na sua declaração de definição, os valores serão preservados e estarão sempre disponíveis em qualquer chamada do subprograma. De outra forma estes valores não são preservados.

Exemplo:

```
program conta
 implicit none
 integer:: i

 do i =1, 5
    call imprime( )
 end do

contains

  subroutine imprime( )
    integer :: k
    print*, k
    k = k + 1
  end subroutine imprime

end program conta
```

este programa pretende fazer uma contagem até 5

como as variáveis locais não são preservadas entre chamadas, o resultado final pode ser:

8886484
8886485
8886486
8886487
8886488

```
program conta
 implicit none
 integer:: i

 do i =1, 10
    call imprime( )
 end do

contains

  subroutine imprime( )
    integer :: k
    print*, k = 1
    k = k + 1
  end subroutine imprime

end program conta
```

Observe que o valor inicial da constante inteira local k está sendo atribuído na sua própria definição

neste caso os valores são preservado e o programa funciona como desejado.

1
2
3
4
5

Um ponto importante a ser destacado é que a constante é inicializada na primeira chamada da subrotina mas, depois em cada chamada ela tem o mesmo valor que tinha quando deixou a subrotina na última chamada.

A variável não é reiniciada nas próximas chamadas mas, retém os valores.

Todas variável inicializada num subprograma é automaticamente salva, portanto não necessita do atributo save

Quando se deseja manter os valores das variáveis locais num subprograma e estas variáveis não são inicializadas na sua definição eles devem receber o atributo **save**.

- **atributo save**

deve ser utilizar o atributo save a todas as variáveis que necessitarem manter seus valores entre chamadas.

```
program conta
 implicit none
 integer:: i

 do i =1, 5
    call imprime( )
 end do

contains

   subroutine imprime( )
     integer, save :: k
     k = k + 1
     print*, k
   end subroutine imprime

end program conta
```

observe o mesmo programa conta agora utilizando o atributo save

como agora as variáveis locais possuem o atributo save elas serão são preservadas entre chamadas, o resultado final será:

1
2
3
4
5

- **declaração save**

a declaração save pode ser utilizada para manter os valores das variáveis entre chamadas.

```
program com
 ......................
 call mad(var1, var2)
 ......................

contains

   subroutine mad(arg1, arg2)
    real, intent(in):: arg1, arg2
    integer :: s, v, x, z, r
    save :: s, v, x
    ......................
   end subroutine mad

end program com
```

uso da declaração save

observe que somente as variáveis locais s, v e x irão preservar seus valores

as outras variáveis locais não serão preservadas entre chamadas

Esta forma de usar a declaração save não é recomendada

De preferência ao atributo save

```
program com2
 ......................
 call mad(var1, var2)
 ......................

contains

  subroutine mad(arg1, arg2)
    real, intent(in) :: arg1, arg2
    integer :: s, v, x, z, r
    save
    ......................
  end subroutine mad

end program com2
```

uso da declaração save

observe que todas as variáveis locais irão preservar seus valores entre chamadas

Esta forma de usar a declaração save não é recomendada

De preferência ao atributo save

66 – Subprogramas como Argumento

Usar um subprograma como um argumento numa chamada de uma função ou subrotina é perfeitamente possível e muitas vezes necessário.

Se o subprograma estiver dentro de um módulo ou é um subprograma interno, as informações necessárias são conhecida (interface implícita) e tudo funciona de forma adequada.

Quando um subprograma (função ou subrotina) definido pelo programador é utilizado como argumento em uma chamada do subprograma, um ponteiro deste subprograma é passado para o programa. e o subprograma será usado.

Exemplo:

```
program funcao_ext
 implicit none
 real :: a1=3.0, b1=5.0, f

 call soma(a1, b1, f)

 print*, " F= ", f

contains

  subroutine soma(a, b, d)
   real, intent(in) :: a, b
   real, intent(out):: d

   d = treis_x(a, b)+treis_x(a, b)

  end subroutine soma

  function treis_x(x, y) result(z)
    real, intent(in):: x, y
    real            :: z
    z = 3*x + 3*y
  end function treis_x

end program funcao_ext
```

Esta é uma forma mais complexa de se calcular

$$f = (3*a + 3*b) + (3*a + 3*b)$$

cuja resposta é 48

entretanto este exemplo permite ilustrar a utilização de um subprogramas sendo usado como argumento

Subprograma externo

Quando o subprograma não está num módulo e também não é um subprograma interno, é necessário que um bloco de interface ou o **atributo external** (ou a declaração external) seja utilizado. Estas são a forma que permitem que as informações necessárias sejam obtidas.

O atributo external deve ser usado junto com a declaração de tipo. O atributo external para uma ou mais subrotinas devem ser declarados numa declaração.

Exemplo:

```
program soma_f
  implicit none
  real  :: a, b
  external  quad, poli

  z1 = soma(quad, a, b)
  z2 = soma(poli, a, b)
  print*, z1, z2

conatins

 function soma(funcao,q,w)result(f)
  real, external  :: funcao
  real, intent(in):: q, w
  real:: f

   f = funcao(q) + funcao(w)

 end function soma
end program soma_f
```

O programa faz a seguinte soma: $f = a + b$
onde a e b podem ser definidos por funções. Então

$f = a^2 + b^2$

$f = (a^2 + 5\,a - 3) + (b^2 + 5\,b - 3)$

observe que a função soma utiliza a função quad ou poli, conforme a chamada

a função quad é guardada num arquivo separado, portanto é uma função externa

```
function quad(x) result(f)
  implicit none
  real, intent(in):: x
  real            :: f

  f = x * x

end function quad
```

---

igualmente, a função poli é guardada num arquivo separado, portanto é também uma função externa

```
function poli(x) result(f)
  implicit none
  real, intent(in):: x
  real            :: f

  f = x**2+ 5 * x - 3.0

end function poli
```

67 – Funções

Uma função em Fortran é um subprograma cujo resultado é um número, um valor lógico, um string ou uma matriz.

Existem dois tipos de funções:

- funções intrínsecas
  funções tais como sin(x), sqrt(x), log(x) que fazem parte da linguagem e são fornecida com ela

- funções definidas pelo usuário
  são subprogramas escrito pelo programador

Uma função (ou um subprograma função) tem a seguinte forma básica:

```
function  nome_função(arg1, arg2, ..., argn)

  [parte especificações]
  [parte execução]

end function nome_função
```

O Fortran 95 não impõe mas, recomenda separar o nome da função da variável local que retornará o resultado da função. Então **dê preferência a forma abaixo**, qualquer que seja o caso, isto é, com ou sem recursão:

```
function nome_função(arg1, arg2, ..., argn) result(var-resultado)

  [parte especificações]
  [parte execução]

end function nome_função
```

- A função inicia com a declaração function e termina com a declaração end function
- depois do nome da função segue um par de parênteses que contem uma lista de argumentos separados por vírgula. São os argumentos formais ou mudos
- segue-se a palavra chave result e entre parênteses a variável que retornará o valor da função
- é importante lembrar que os argumentos formais não podem ser expressões

Se uma função não necessita de nenhum argumentos formal então ela pode ser escrita assim:

```
function nome_função( ) result (var-resultado)

  [parte especificações]
  [parte execução]

end function nome_função
```

os argumentos não foram usados, mas o par de parêntese tem que ser utilizado.

Regras básicas

- Uma função é um conjunto completo de declarações que recebe alguma entrada (input) de fora através dos argumentos, realiza cálculos e retorna o resultado
- é necessário em algum momento associar o valor calculado à variável que então irá retornar o valor obtido

  var-resultado = valor final

Uma função deve receber valores pelos argumentos formais. Ela realiza os cálculos necessários com eles e coloca o resultado na variável var-resultado que é a responsável por passar o valor para quem chamou a função.

Todos os argumentos da função devem ter sua intenção declarada. A boa técnica de programação diz que se deve usar somente argumentos como sendo de entrada [intent(in)].

A alteração de valores do programa host por uma função pode ser feito mas deve ser evitado. Isto é conhecido como efeito colateral [side-effects]. Esta técnica deve ser evitada ao máximo.

Exemplos

```
function soma(a, b) result (z)
 integer, intent(in):: a, b
 integer            :: z

 z = a + b

end function soma
```

```
function positivo(x) result (positivo)
 real, intent(in):: x
 logical         :: positivo

 if (x > 0.0) then
    positivo = .true.
 else
    positivo = .false.
 end if

end function positivo
```

```
   program dois
     integer:: a = 1, b = 2, c

     c = soma(a, b)
     print*, c

   contains

     function soma(x,y) result(m)
      integer,intent(in):: x, y
      integer           :: m

      m = x + y
     end function soma

   end program dois
```

```
      function soma(a, b) result(ss)
         integer, intent(in):: a, b
         integer            :: ss

          ss = a + b

      end function soma
```

68 – Usando Funções

A utilização de uma função definida pelo usuário é bastante similar à forma como se utiliza uma função implícita. Basta escrever o nome da função com os argumentos necessário, onde desejado.

Por exemplo, tendo-se a função soma

```
program teste
 integer:: resultado
 integer:: a = 1, b = 2
 ..................
 resultado = soma(a, b)
 ..............
contains

  function soma(a, b) result(s)
   integer, intent(in):: a, b
   integer            :: s

   s = a + b
  end function soma
end program teste
```

veja a função soma definida pelo programador

ela foi escrita como uma função interna

ela pode ser usada no programa assim

resultado = soma(a, b)

onde a variável inteira resultado conterá o resultado obtido pelo uso da função soma

neste caso será o valor inteiro 3 [1+2=3]

Observe a similaridade da forma de utilização da função definida pelo programador com o uso da função intrínseca **seno**.

```
real:: alfa, angl
.............
alfa = 1.7

angl = sin(alfa)
.............
```

É importante não esquecer:

- o número de argumentos reais tem que ser igual ao número de argumentos formais
- os tipos dos argumentos reais tem que ser os mesmos que os dos argumentos formais
- os argumentos reais, eles podem ser:
  - expressões
  - variáveis
  - constantes
- todos os argumentos formais devem ser declarados quanto a intenção, que normalmente é intent(in)
- se o argumento formal for do tipo entrada (in) a função não pode altera o seu valor
- finalmente não se pode esquecer de entrar com o valor final (resultado) na variável de retorno da função

Nota importante

Uma função interna não pode conter outra função. Isto é, não pode ter função interna.

```
program errado
  implicit none
  ..............

contains

  real function interna1(....)
     ............

  contains

     integer function interna2(....)
       ..............
     end function interna2

  end function interna1

end program errado
```

este programa está errado

porque existe uma função interna (interna2)

dentro de uma função interna (interna1)

69 – Escopo das Variáveis

O escopo de uma variável refere-se a forma como diferentes partes de um programa têm acesso às variáveis. As questões básicas são:

- uma função pode usar uma variável definida num programa principal?
- pode um programa principal usar uma variável definida dentro de um subprograma?

As regras de escopo dão a resposta a estas perguntas. Regras de escopo diz respeito a visibilidade e acessibilidade das variáveis, parâmetros e funções.

O conceito de host (hospedeiro) é importante. O host é a unidade de programa que "contém" um subprograma. Pode-se afirmar que um programa principal que contenha subprogramas internos é o host (hospedeiro) destes subprogramas.

- **As regras de escopo são:**

REGRA 1 : o escopo (ambiente) de um dado é a unidade de programa onde ele é declarado. Onde ele é visível e acessível. Um dado definido localmente num subprograma não é visível fora do subprograma

REGRA 2 : subprogramas internos herdam as "entidades" por associação de hospedagem (HOST Association)

REGRA 3 : dados e subprogramas declarados em módulos são visíveis por associação via **declaração use** (USE Association). A declaração **use** permite tornar (seletivamente) acessível os dados e subprogramas que contém. Modulos são muito usados para se definir "dados" globais.

- **Entidades Locais**

Uma das principais vantagem do uso de um subprograma decorre do fato de suas variáveis locais serem completamente separadas (não visíveis e não acessíveis) por outras unidades de programa (**regra de escopo 1**).

Um hosts não tem acesso às variáveis locais de subprogramas que contem (subprogramas internos e os colocados nos módulos). Por isto as variáveis locais podem usar os mesmos nome das variáveis do host e mesmo assim continuam sendo variáveis diferentes.

```
program ambiente_1
  implicit none
  real:: a, b=5.0
  ...................
  d = xyz(b)
  ..................
contains

  function xyz(n) result(m)
  real, intent(in):: n
  real            :: m
  real            :: b=2.0
  ................
  m = n / b
  ................
  end function xyz

end program ambiente_1
```

Observe a existência de 2 ambientes:

1- ambiente do host (programa principal)
2- ambiente dentro do subprograma (função)

observe como a token **contains** separa bem os 2 ambientes

a variável **b** na função xyz (**ambiente 2**)

não é visível pelo programa principal (**host** = ambiente 1)

e por isto **b** (no ambiente 1)
é diferente de **b** (no ambiente 2)

e pode ter valor diferente

- **Entidades Globais**

  - **Associação por Hospedagem - (HOST Association)**

Também chamada de **herança**.

Quando os dados de um programa devem ser compartilhados com um subprograma deve-se dar preferência a compartilhar os dados via os argumentos. A vantagem do uso dos argumentos é o total controle que temos sobre os dados pelo uso do atributo **intent**.

Mas os dados podem ser compartilhados entre o host (ou hospeiro) e os subprogramas por herança. Esta forma de compartilhar dados é possível mas não é uma forma recomendável, devido ao menor controle que se tem sobre os dados. Além disto esta forma de trabalhar leva a uma perda de clareza e legibilidade do programa. O que é péssimo.

A regra 2 (regra de escopo 2) diz que os dados definidos no host são herdados (visíveis e acessíveis) pelos subprogramas. Então:

```
program calculo
  implicit none
  real:: a, b, c
  real:: num = 0
  ..................
  call ppp(a, b)
  call kkk(a)
  ..................
  ..................
contains

  subroutine ppp(d,e)
    real, intent(in):: d, e
    real            :: T

    T = d * e
    num = num * d
  end subroutine ppp

   function kkk(f) result(m)
     real, intent(in):: f
     real            :: m

     m = f + (f**2)
   end subroutine kkk

end program calculo
```

Observe que as variáveis num e c são variáveis globais

Elas são herdadas (por host association) pelos subprogramas

evite usar as variáveis globais dentro dos subprogramas.

Não se recomenda usar herança de dados !

por causa do que é chamado efeito colateral

mas principalmente pela perda de clareza e legibilidade do programa

- **Associação use - (USE Association)**

Forma **Altamente recomendada !**

A melhor forma de se definir variáveis globais é via a **associação use**. A vantagem de se usar a **associação use** é o total controle sobre os dados.

Dados contidos num módulo são acessíveis (tornam-se visíveis) com o uso da **declaração use** (use association).

Quando se define um dado como global, é necessário não se utilizar o mesmo nome quando for preciso definir variáveis locais.

Nomes de variáveis globais não podem ser usados para definir variáveis locais.

Caso um módulo tenha várias variáveis globais, é possível tornar acessível apenas algumas delas (não todas), usando-se o atributo **only**. O atributo only é usado para restringir as variáveis, isto é, quais variáveis do módulo serão visíveis.

Sua forma é: **use** *modulo-nome***, only:** *lista-de-nomes*

Exemplo: **use funcoes, only : pi, veloc_luz**

neste caso o módulo funcoes tem várias variáveis globais, mas somente as variáveis pi e veloc_luz serão visíveis.

Outra possibilidade é a de se utilizar uma clausula de renomeação, isto é, quando uma variável global (no módulo) tem um nome igual a uma variável definida no programa que vai importar as variáveis globais, pode-se renomear a variável global.

Sua forma é: **use** *modulo-nome***,** var1 => var2

Exemplo: **use funcoes, veloc_luz => velo**

neste caso o módulo funcoes tem todas as suas variáveis globais acessíveis e mais, a variável global veloc_luz é dentro do programa importador identificada pelo nome velo (renomeação).

Exemplo:

```
program integral
  use global
  implicit none
  ..................
  ..................
contains
   function xyz(n) resuslt(m)
     real, intent(in):: n
     real            :: b, c, m
     ................
   end function xyz
end program integral
```

A parte pública do módulo global (não mostrado aqui)

é tornada acessível para o programa principal, chamado integral por associação **use**

a função xyz tem acesso as variáveis globais por associação por hospedagem (host association)

70 – Estrutura Básica de um Módulo

Módulo é a forma mais adequada e fácil que se tem de compartilhar dados e subprogramas entre diferentes unidades de programas. Os módulos permitem que dados e subprogramas sejam usados e reutilizados quantas vezes for necessário, em muitos programas. Módulos são utilizados para:

- construção de uma biblioteca de rotinas
- criar e disponibilizar dados globais
- ou melhor ainda a combinação das duas facilidades listadas

Módulo é muito poderoso e útil para disponibilizar dados definidos pelo usuário.

- **A forma genérica de um módulo é:**

```
module  nome-módulo
  implicit none

  [parte com especificações]

contains

  [ subprogramas internos]

end modulo nome-módulo
```

Um programa pode usar, acessar, o conteúdo público de um módulo com a declaração **use**.

Dados Globais

Coloque todos seus dados globais em um módulo. Faça uso da declaração **use** para torná-lo acessíveis.

```
module ppp
  implicit none
  integer, public:: v_inic = 10
  real,public,parameter::pi=3.14159

end module ppp



program www
  use ppp
  implicit none
  ............
  y = v_inic * sin(2*pi*theta)
  ............
  ............
end program www
```

O módulo está colocado em um arquivo separado do programa principal.

Ele contem uma variável (v_inic) inicializada com o valor 10 e uma constante com nome chamada pi (pi=3.14159)

que estão sendo usadas no programa principal www

Observe que os dados (v_inic e pi) foram tornados acessíveis ao programa principal pelo uso da declaração **use** (use association)

Todos os dados no módulo ppp são acessíveis

```
module ppp
  implicit none
  integer, public:: v_inic = 10
  real,public    :: a, b, c
  real,public,parameter::pi=3.14159

end module ppp



program www
  use ppp, only a, b, c
  implicit none
  ............
  ............
  ............
end program www
```

Observe que agora a declaração use foi usada com o atributo **only**

Neste caso apenas as variáveis a, b e c são globais e acessíveis no programa principal

Um módulo não tem existência por si só, ele deve ser utilizado por outra unidade de programa, principalmente por um programa principal.

Exemplos:

```
modulo constantes
 implicit none
 real,public,parameter::PI=3.1415926
 real,public,parameter::g=9,8065
 integer,public::contador1

end module constantes
```

este módulo define constantes globais que deve ser usado assim:

```
program ttt
  use constantes
  implicit none
  ..............
end program ttt
```

```
módulo medias
!
public::media_aritmetica,function geometrica

contains

 function media_aritmetica(a, b, c) result(z)
   implicit none
   real, intent(in):: a, b, c
   real            :: z

   media_aritmetica = (a + b + c) / 3.0

 end function media_aritmetica
 !
 function geometrica(a, b, c) result(u)
  implicit none
  real, intent(in):: a, b, c
  real            :: u

  geometrica = (a + b + c)**(1 / 3.0)

 end function geometrica

end module medias
```

módulo medias só contem subprogramas

```
program ddd
 use medias
 implicit none
 ..............
 num1=media_aritmetica(x, f, h*x)
 num2 = geometrica(t, d, w)
 ..............
end program ddd
```

71 – Atributos Public e Private

Todas as entidades no módulo são públicas por default. Isto significa que usando-se a declaração **use** todas as entidades se tornarão globais.

Entretanto é possível restringir o acesso aos dados quando necessário. Para isto pode-se utilizar os atributos **public** e **private**, ou então as declarações **public** e **private**.

- **Atributo public ou private**

  type, public :: nome-1, nome-2, ..., nome-n
  type, private :: nome-1, nome-2, ..., nome-n

  ou seja especificando-se o atributo public na definição de um dado ele se torna público (ou privado)

- **Declaração public ou private**

  public :: nome-1, nome-2, ..., nome-n
  private :: nome-1, nome-2, ..., nome-n

Quando se deseja tornara público (ou privado) um subprograma utiliza-se a declaração public (ou private), não os atributos, que são aplicáveis às variáveis e constantes.

Lembre-se um dado não público (private) é um dado que só é acessível (só é visível) dentro do módulo.

Toda dado que não for explicitamente dito privado é por default público e portanto acessível fora do módulo. Entretanto é uma boa técnica de programação explicitar sempre quando um dado é publico ou não. Isto mantém a legibilidade do programa.

Exemplo:

```
module FFF
  implicit none
  integer, public:: a, b
  real, private  :: c
  logical, private:: d
  real,public,parameter::const=1,3945
  !
  public:: vol, dist

contains

  function vol( )
     ..............
  end function vol
!
  subroutine dist(x)
     ............
  end subroutine

end module FFF
```

observe que as variáveis a e b são públicas (globais)

as variáveis c e d são somente visíveis dentro do módulo FFF

a função vol e a sub-rotina dist são públicas

72 – Compilando Módulos

Normalmente os programas fontes são guardados em um arquivo que possuí a extensão ".f90" ou ".f95". Módulos também são guardados em arquivos com a mesma extensão.

Para se fazer a compilação de um programa fonte Fortran 95, usando-se o compilador **F95**, deve-se executar o seguinte comando:

F95 arquivo.f90 (ou F95 arquivo.f95)

Este comando irá compilar o programa chamado arquivo.f90 (ou arquivo.f95). Não havendo erros, no programa fonte, imediatamente após a compilação de forma inteiramente automática é feita o processo de link edição que então irá gerar o programa executável.

Devido a não se ter definido um nome para o programa executável, este será identificado pelo nome default: **a.out**.

Diferentes compiladores tem diferentes "personalidade", isto significa que usando um compilador diferente o processo e a seqüência pode diferir um pouco, mas essencialmente será similar ao indicado aqui.

Admita que tenhamos os seguintes arquivos:

- **módulos**
  modulo-1.f95 modulo-2.f95 modulo-3.f95

- **programa principal**
  programa-1.f95

O comando

F95 -o integral modulo-1.f95 modulo-2.f95 modulo-3.f95 programa-1.f95

utilizado com a opção **-o** (de output) o compilador **F95** irá compilar os módulos: **modulo-1**, **modulo-2** e **modulo-3** com o **programa-1** gerando o *programa executável* **integral**.

Neste processo, é necessário que se escreva os módulos que não usam outros módulos primeiro, seguidos por aqueles que usam os módulos já listados e finalmente seguido do programa principal.

É possível também se fazer a compilação dos módulos separadamente. Isto irá gerar dois outros arquivos: nome.o (arquivo objeto) e nome.mod (arquivo com informações sobre o módulo).

Então:

F95 -c modulo-1.f95

com a opção **-c** (c de compilar) significa: compile sem gerar o programa executável, mas gere o arquivo objeto chamado **modulo-1.o** (nome-do-módulo seguido de um ponto e da letra o).

Depois de compilar todos os módulos separadamente pode-se:

- ***compilar o programa principal com os objetos***

  F95 -o integral modulo-1.o modulo-2.o modulo-3.o programa-1.f95

- ***ou então compilar o programa principal para gerar o objeto e depois gerar o executável***

  F95 -c programa-1.f95
  F95 -o integral modulo-1.o modulo-2.o modulo-3.o programa-1.o

Note que a ordem dos programas pode ser relevante.

73 – Bloco Interface

Todo subprograma interno e os módulos têm uma interface explícita. Todo subprograma externo têm uma interface implícita e portanto necessita de um bloco de interface explícita.

Então quando se utiliza um subprograma externo deve-se utilizar um bloco interface para que o programa que for utilizar o subprograma tenha acesso a várias informações que de outra forma não disporia.

- **A sintaxe para um bloco interface é:**

```
interface
  corpo_1_da_interface
  corpo_2_da_interface
  corpo_3_da_interface
end interface
```

Cada interface é constituída de:

declaração inicial do subprograma (função ou sub-rotina)
declarações que dizem respeito aos argumentos usados
declarações de especificação do subprograma usado
declaração final do subprograma

De fato o que se faz é: copiar as informações de especificação do subprograma para dentro do bloco interface, então:

```
interface
   function hipo(a, b, c) result(z)
      real, intent(in) :: a, b, c
      real             :: z
   end function hipo
end interface
```

Esta declaração bloco interface contém as informações necessárias da função externa hipo que serão passadas ao programa que vai utilizar esta função.

Aonde colocar um bloco de interface? Coloque depois da declaração **implicit none**.

exemplo:

```
function hipo(a, b, c) result(z)
   real, intent(in):: a, b, c
   real            :: z
   ......................
   ......................
   ......................
end function hipo
```

hipo é uma subprograma externo

um subprograma guardado em um arquivo separado

```
program ddd
  implicit none
  interface
   function hipo(a, b, c) result(z)
     real, intent(in) :: a, b, c
     real             :: z
   end function hipo
  end interface
  ......................
  ......................
  y = hipo(x0, x1, y)
  ......................
  ......................
end program ddd
```

utilização do subprograma externo hipo no programa principal ddd

74 – Arquivos

No Fortran 95 um arquivo de informações é constituído por linhas. Cada linha do arquivo é considerada um registro (record). Portanto um arquivo consiste numa sucessão seqüencial de registros.

O acesso a estes registros podem ser feito de forma seqüencial ou direta, dependendo de como o arquivo foi gravado e de qual meio é utilizado (fita, disco rígido) para guardá-lo.

O arquivo seqüencial é o mais fácil de se usar e por isto é o mais usado, independentemente do meio no qual é guardado.

Quando é necessário realizar-se leituras e escritas no arquivo isto é feito usando-se as declarações: read e write.

Para o Fortran é necessário que o arquivo seja identificado por um número (usualmente entre 1 e 99) antes que seja possível realizar qualquer trabalho com ele. Esta identificação é feita com a declaração **open**.

OPEN

Uma declaração open típica tem a forma:

open(unit=u, file="fname", status="sta", action="act", position="pos")

u	expressão inteira (mais freqüentemente uma constante inteira) escolhida arbitrariamente pelo programador, que passará a identificar o arquivo
fname	nome do arquivo com o qual se vai trabalhar se necessário fornecer o caminho
sta	status new, old, replace, scratch ,unknown
act	ação a ser realizada sobre o arquivo read, write, readwrite
pos	posição onde inicia a leitura do arquivo, se o arquivo for seqüencial default é seqüencial rewind, backspace

exemplos:
```
open(unit=20, file="dados.data", status="old", action="read", position="rewind")
open(unit=32, file="/home/paulo/resultados/temperaturas.data ", status="old", &
action="read", position="rewind")
```

u - o número da unidade u deve estar entre 1-99

CLOSE

Ao término do trabalho com o arquivo deve ser fechado com a declaração **close**.

```
close(unit=25)
close(unit=32)
```

READ / WRITE

A leitura e a escrita de dados num arquivo usa os comandos read e write da forma usual.

Exemplo:

para escrever um arquivo do tipo:

```
-12.330     234.500
26.654      -124.667
0.234       451.213
.....
.....
```

pode-se usar a declaração:

```
write(unit=37, fmt="(2f9.3) ") nval, va_x
```

também:

```
read(unit=74, fmt=*) conts, num, sup
```

lê as 3 variáveis do arquivo conectado na unidade 74

Observe o seguinte:

Cada declaração read lê uma linha do arquivo por vez

```
.....
.....
character(len=10) :: nome
real              :: nota_mecanica, nota_termodinamica, cr
integer           :: ano_ingresso
.....
.....
open(unit=10, file="notas.data", status="old", action="read", position="rewind")
.....
.....
read(unit=10, fmt=*) nome, nota_mecanica, nota_termodinamica, cr, ano_ingresso
.....
.....
```

O arquivo notas.data pode ter a seguinte forma:

```
"paulo"     8.9    9.2    8.32    1999
"antonio"   7.2    10.0   7.9     2001
"pedro"     8.2    9.4    8.2     2002
.............................
```

observe que os dados são separados por espaço dentro do arquivo.

O seguimento de programa mostrado faz a leitura da primeira linha (só um read) então as variáveis ficam com os seguintes valores:

nome = paulo
nota_mecanica = 8.9
nota_termodinamica = 9.2
cr = 8.32
ano_ingresso = 1999

Necessitando de novos valores é necessário fazer nova leitura do arquivo. Cada execução do comando read inicia a entrada de dados uma nova linha do arquivo.

```
.....
.....
read(unit=10, fmt=*) nome, nota_mecanica, nota_termodinamica, cr, ano_ingresso
read(unit=10, fmt=*) nome, nota_mecanica, nota_termodinamica, cr, ano_ingresso
read(unit=10, fmt=*) nome, nota_mecanica, nota_termodinamica, cr, ano_ingresso
.....
.....
```

após a execução do 3 (três) declarações read as variáveis conterão os dados do pedro, i. é,

nome= pedro
nota_mecanica = 8.2
nota_termodinamica = 9.4
CR = 8.2
ano_ingresso = 2002

a leitura de um arquivo inteiro, usualmente, é feita com o uso da declaração **do**

Quando o número de variáveis for menor que o número de dados do arquivo, os dados extras existentes no arquivo não serão lidos.

```
real ::  a, b, c, d, e, f
.....................
.....................
open(unit=50, file="numeros.data", status="old", action="read", position="rewind")
.....................
.....................
read(unit=50, fmt=*) a, b, c
read(unit=50, fmt=*) d, e, f
.....................
.....................
```

se o arquivo de dados (numeros.data) contiver:

```
10.0    2.00    30.0    4.00
5.0     6.00    70.0    8.01
```

teremos:

a = 10.0
b = 2.00
c = 30.0
d = 5.0
e = 6.00
f = 70.0
4.00 e 8.01 não serão lidos

75 – Fim de Arquivo

Em muitas situações o número de itens a ser lido de um arquivo é conhecido. É uma prática comum colocar o número de itens como um registro no próprio arquivo. Lê-se este valor e assim estamos de posse do número de itens existente no arquivo. Mas nem sempre isto é feito.

Nestes caso temos que usar a facilidade "**iostat**" que fornece o status da operação.

Então:

```
integer :: IO_status
.............................
.............................
open(unit=28, file="notas.data", status="old", action="read", position="rewind")
.............................
.............................
read(unit=28, fmt=*, iostat=iostat_status) var1, var2, ..., varn
```

permite ler da unidade 28, com formato livre as variaveis: var1, var2, ..., varn .

Depois de executar uma leitura com a declaração read o compilador escreve na variável inteira IO_status a atual condição de leitura.

- se IO_staus é zero, a leitura foi executada corretamente e as variáveis lidas estão de posse de seus valores. Tudo normal
- se o valor da variável IO_status é positivo, a leitura que acabou de ser feita tem algum problema. Por exemplo um número real sendo fornecido a uma variável inteira, etc... Se IO_status é positivo, você terá lixo nas variáveis que acabou de ler.
- se o valor de IO_status é negativo . isto significa que o FIM-DO-ARQUIVO foi alcançado. Algumas ou todas as variáveis que acabaram de ser lidas não receberam valores do arquivo.

Exemplo

```
integer :: io_estado
integer :: a, b, c
.....................
.....................
! lendo os dados de um arquivo
do
  read(unit=20,fmt=*,iostat=io_estado)  a, b, c
  ! testando
  if (io_estado > 0) then
    print*, "algo errado com seus dados"
    stop
  elseif (io_estado < 0) then
    ! chegou ao fim do arquivo
    ! acaba com as leituras
    print*, "leu todos os dados"
    exit
  else
    ! nada de anormal aconteceu
    ! faz outra leitura
  end if
end do
.....................
.....................
```

76 – Funções Intrínsecas

Funções intrínsecas são funções (e algumas sub-rotinas) que são incluídas na biblioteca de funções do Fortran. O Fortran 95 possuí 4 (quatro) classes de funções:

- funções **elementar** (Elemental procedure)
  seus argumentos formais podem ser escalar ou matriz

- função **inquisitória** (Inquiry function)
  a resposta depende da propriedade do argumento principal

- função **transformacional** (Transformational function)
  transforma uma matriz em outra matriz

- **subprograma não elementar** (Nonelemental procedures)
  sub-rotina

Índice alfabético das funções e sub-rotinas intrínsecas do Fortran 95

1	abs(A)	58	lle(string_a, string_b)
2	achar(I)	59	llt(string_a, string_b)
3	acos(x)	60	log(x)
4	adjustl(string)	61	log10(x)
5	adjustr(string)	62	logical(l[,kind])
6	aimag(z)	63	matmul(matrix_a, matriz_b)
7	aint(A,[kind])	64	max(a1, a2[,a3...])
8	all(mask[,dim])	65	maxexponent(x)
9	allocated(array)	66	maxloc(array[,mask])
10	anint(A[,kind])	67	maxval(array[,dim][,mask])
11	any(mask[,dim])	68	merge(tsource, fsource, mask)
12	asin(x)	69	min(a1, a2[,a3...])
13	associated(point[,target])	70	minexponent(x)
14	atan(x)	71	minloc(array, mask)
15	atan2(y, x)	72	minval(array[,dim][,mask])
16	bit_size(I)	73	mod(a, p)
17	btest(I, pos)	74	module(a, p)
18	ceiling(A)	75	mvbits(from, frompos, len, to, topos)
19	char(I[,kind])	76	nearest(x, s)
20	cmplx(x[,y][,kind])	77	nint(a[,kind])
21	conjg(z)	78	not(i)
22	cos(x)	79	null(mod)
23	cosh(x)	80	pack(array, mask[,vector])
24	count(mask[,dim])	81	precision(x)
25	cpu_time(time)	82	present(a)
26	cshift(array,shift[,dim])	83	product(array[,dim][,mask])
27	date_and_time([date][,time][,zone][,values])	84	radix(x)
28	dble(A)	85	random_number(harvest)
29	digits(x)	86	randomseed([size][,put][,get])
30	dim(x, y)	87	range(x)
31	dot_product(vector_a, vector_b)	88	real(a[,kind])
32	dprod(x, y)	89	repeat(string, ncopies)
33	eoshift(array,shift[,boundadry][,dim])	90	reshape(source, shape[,pad][,order])
34	epsilon(x)	91	rrspacing(x)
35	exp(x)	92	scale(x)
36	exponent(x)	93	scan(string, set[,back])
37	floor(A)	94	selected_int_kind(r)
38	fraction(x)	95	selected_real_kind([p][,r])
39	huge(x)	96	set_exponent(x, i)
40	iachar(c)	97	shape(source)
41	iand(i, j)	98	sign(a, b)
42	ibclr(i, pos)	99	sin(x)
43	ibits(i, pos, len)	100	sinh(x)
44	ibset(i, pos)	101	size(array[,dim])
45	ichar(c)	102	spacing(x)
46	ieor(i, j)	103	spread(source, dim, ncopies)
47	index(string, substring[,back])	104	sqrt(x)
48	int(a[,kind])	105	sum(array[,dim][,mask])
49	ior(i, j)	116	system_clock([count][,count_rate][,count_max])
50	ishft(i, shift)	107	tan(x)
51	ishftc(i, shift[,size])	108	tanh(x)
52	kind(x)	109	tiny(x)
53	lbound(array[,dim])	110	transfer(source, mold[,size])
54	len(string)	111	transpose(matrix)
55	len_string(string)	112	trim(string)
56	lge(string_a, string_b)	113	ubound(array[,dim])
57	lgt(string_a, string_b)	114	unpack(vector, mask, field)
		115	verify(string, set[,back])

Descrição das Funções Intrínsecas

1	**abs(A)**
Descrição:	Retorna o valor absoluto do argumento
Classe:	Função elementar
Argumentos:	**A** deve ser do tipo inteiro, real ou complexo
Resultado:	se A é inteiro ou real o resultado é |A| se A é complexo [o complexo (x, y)] então o resultado é obtido por $\sqrt{x^2+y^2}$
Exemplos:	real:: num = -23.5, resultado resultado = abs(num) ! resultado = 23.5 resultado = abs(7.0,8.1) ! resultado = 10.630145

2	**achar(I)**
Descrição:	Retorna o caractere de uma posição específica na tabela ascii, mesmo que o conjunto de caracteres default do processador seja diferente. **Achar** é o inverso da função **Iachar**
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro
Resultado:	Caractere de comprimento 1. Se $0 \leqslant I \leqslant 127$, o resultado é o caractere na posição **I** da tabela ascii, desde que o processador possa representar o caractere. Para todos os outros casos o resultado é dependente do processador usado
Exemplos:	character(len=1):: resultado integer:: c=65 resultado = achar(c) ! resultado = A resultado = achar(35) ! resultado = # resultado = achar(63) ! resultado = ?

3	**acos(x)**
Descrição:	Retorna o arco-coseno do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser real e estar entre: $-1 \leq x \leq +1$
Resultado:	Mesmo tipo de x. O resultado é o valor $\cos^{-1}(x)$ expresso em radianos na faixa: $0 \leq a\cos(x) \leq \pi$
Exemplos:	30^0 = 0.523598 radianos cos(0.523598)=0.866025 real:: theta, resultado theta= 0.866025 resultado = acos(theta) ! resultado = 0.523599 valor que identifica 30°

4	**adjustl(string)**
Descrição:	Ajusta o string a esquerda, removendo brancos anteriores e anexando brancos no final
Classe:	Função elementar
Argumentos:	**string** tem que ser do tipo caractere
Resultado:	Tipo caractere com mesmo comprimento e mesmo parâmetro de tipo (kind) que o argumento. O resultado é o mesmo string exceto que qualquer branco anterior (a esquerda) é removido e o mesmo número de brancos são acrescidos ao final do string (a direita)
Exemplos:	aqui o sublinha(_) está sendo usado para representar um espaço em branco character(len=9)::texto, resultado texto="_ _ _ _ Paulo" resultado = adjustl(texto) !resulta em: "Paulo _ _ _ _" resultado = adjustl("_ _ _ Carlos") !resulta em: "Carlos _ _ _"

5	**adjustr(string)**
Descrição:	Ajusta a direita o string removendo brancos posteriores e anexa brancos no início
Classe:	Função elementar
Argumentos:	**string** tem que ser do tipo caractere
Resultado:	Tipo caractere, com mesmo comprimento e mesmo parâmetro de tipo (kind) de string. O resultado é o mesmo string exceto que qualquer branco posterior (a direita) é removido e o mesmo número de brancos são acrescidos no início do string (a esquerda)

Exemplos:	aqui o sublinha(_) está sendo usado para representar um espaço em branco
	
	`character(len=9)::texto, resultado`
	`texto="Mario _ _ _ _"`
	`resultado=adjustr(texto)  !resulta em: "_ _ _ _ Mario"`
	`resultado=adjustr("Rui _ _ _ _ _ _")  !resulta em: "_ _ _ _ _ _ Rui"`

6	**aimag(z)**
Descrição:	Retorna a parte imaginária de um número complexo
Classe:	Função elementar
Argumentos:	**z** tem que ser do tipo complexo
Resultado:	Tipo real. Se **z** tem o valor (x, y), o resultado tem o valor y

Exemplos:	o número complexo a=4+7j é representado por a=(4.0,7.0)
	
	`complex:: a`
	`a=(4.0,7.0)`
	`resultado = aimag(a)  ! resultado = 7.0`
	`resulatdo = aimag(5.2,9.4)  ! resultado = 9.4`

7	**aint(A [,kind])**
Descrição:	Trunca o valor para um número inteiro
Classe:	Função elementar
Argumentos:	**A** tem que ser do tipo real **kind** (opcional) expressão inteira escalar de inicialização
Resultado:	Se o parâmetro de tipo (**kind**) está presente o tipo é dado por kind, caso contrário o tipo é o mesmo de **A**. **A** é o maior inteiro que não excede **A** e o sinal é o mesmo do de **A**. Se |**A**| é menor que 1, **aint(A)** vale zero

Exemplos:	`real:: a = 5.89, resultado`
	`resultado=aint(a)  ! resultado = 5.0`
	`resultado=aint(-1.45)  ! resultado = -1.0`

8	**all(mask [,dim])**
Descrição:	Determina se todos os valores são verdadeiros em uma matriz ou em uma dimensão da matriz
Classe:	Função transformacional
Argumentos:	**mask** tem que ser uma matriz lógica **dim** (opcional) tem que ser um escalar inteiro com valores na faixa entre 1 e r, onde r é o número de dimensões (rank) da matriz **mask**
Resultado:	Matriz ou escalar do tipo lógico. O resultado é um escalar se **dim** for omitido ou **mask** for um vetor (dimensão = 1). O resultado escalar é **.TRUE.** (verdadeiro) somente se todos os elementos da matriz **mask** forem verdadeiros ou **mask** tem tamanho zero. O resultado é **.FALSE.** (falso) se algum elemento de **mask** for falso. A matriz resultado é do mesmo tipo porém com uma dimensão menor que a dimensão de **mask** em uma unidade
Exemplos:	`logical, dimension(4):: holder1,holder2` `logical:: resultado1,resultado2` `  holder1=(/.true.,.true.,.true.,.true./)` `  resultado1=all(holder1)  ! resultado1=T` `  holder2=(/.true.,.false.,.true.,.true./)` `  resultado2=all(holder2)  ! resultado2=F` resultado1 é igual a .TRUE. porque todos os elementos do argumento são TRUE resultado2 é igual a .FALSE. porque um dos elementos do argumento é FALSE `Seja A=| 1 7 7	e B=| 0 7 8 |` `	3 9 8		2 9 9 |`  `integer,dimension(2,3)::A,B` `logical,dimension(3)::resultado1` `logical,dimension(2)::resultado2` ` A=(/1,3,7,9,7,8/),(/2,3/)` ` B=(/0,2,7,9,8,9/),(/2,3/)` ` resultado1=all(A = = B,dim=1) !resultado1=(F,T,F)` ` resultado2=all(A = = B,dim=2) !resultado2=(F,F)` all(A = = B,dim=1) testa para verificar se cada coluna da matriz A é igual a cada elemento correspondente da matriz B. O resultado é um vetor (com 3 posições) que possui o valor (.false.,.true.,.false.) porque somente a segunda coluna possui valores iguais em A e B all(A = = B,dim=2) testa as linhas para verificar se todos os elementos são iguais em cada linha nas duas matrizes. O resultado é uma matriz de dimensão 1 (vetor) com 2 elementos (.false.,.false.) porque as duas linha são diferentes

9	**allocated(array)**
Descrição:	Indica quando uma matriz está alocada
Classe:	Função inquisitória
Argumentos:	**array** tem que ser uma matriz do tipo alocável
Resultado:	Escalar lógico. O resultado é verdadeiro (**.TRUE.**) se a matriz está alocada, falso (**.FALSE.**) se a matriz não está alocada ou indefinido se o status de alocação é indefinido

Exemplos:	`real,allocatable,dimension(:,:)::veloc`
	`logical::matriz_esta_alocada`
	`  matriz_esta_alocada=allocated(veloc)  !matriz_esta_alocada = F`
	
	Como a matriz veloc não está alocada a variável lógica matriz_esta_alocada contém o valor .FALSE.
	
	`real,allocatable,dimension(:,:)::veloc`
	`logical::matriz_esta_alocada`
	`  allocate(veloc(3,5))`
	`  matriz_esta_alocada=allocated(veloc)  !matriz_esta_alocada = T`
	
	Agora a matriz veloc está alocada, e a variável lógica matriz_esta_alocada contém o valor .TRUE.

10	**anint(A [,kind])**
Descrição:	Arredonda para o número inteiro mais próximo
Classe:	Função elementar
Argumentos:	**A** tem que ser do tipo real **kind** (opcional) expressão escalar inteira de inicialização
Resultado:	Tipo real. Se o parâmetro de tipo (**kind**) está presente o tipo é especificado por ele. Caso ausente o resultado tem mesmo tipo de **A**. Se **A** é maior que zero, **anint(A)** assume o valor **anint(A + 0.5);** se **A** é menor ou igual a zero, **anint(A)** tem o valor de **anint(A – 0.5)**
Exemplos:	real:: A, B A=4.34 B=anint(A) !B= 4.0 B=anint(4.5) !B= 5.0 B=anint(-2.639) !B=-3.0

11	**any(mask [,dim])**
Descrição:	Determina se algum valor é verdadeiro em uma matriz ou em uma dimensão da matriz
Classe:	Função transformacional
Argumentos:	**mask** tem que ser uma matriz lógica **dim** (opcional) expressão inteira escalar com valor entre 1 e r, onde r é a dimensão (rank) de **mask**
Resultado:	Matriz ou escalar do tipo lógico. O resultado é um escalar se **dim** é omitido ou **mask** tem dimensão 1. O escalar é verdadeiro (.TRUE.) se algum elemento da matriz **mask** é verdadeiro. O resultado é falso (.FALSE.) se nenhum elemento da matriz **mask** é verdadeiro ou **mask** tem tamanho zero. A matriz resultado tem o mesmo parâmetro de tipo (kind parameter) da matriz **mask** e uma dimensão (rank) que é 1 unidade menor que a matriz **mask**. Cada elemento na matriz resultado é verdadeiro se algum elemento da dimensão definida pela matriz **mask** é verdadeiro
Exemplos:	Considere as matrizes: $A = \begin{bmatrix} 1 & 6 & 9 \\ 3 & 4 & 8 \end{bmatrix}$ $B = \begin{bmatrix} 2 & 6 & 9 \\ 0 & 4 & 7 \end{bmatrix}$ integer,dimension(2,3)::A,B logical,dimension(3)::C logical,dimension(2)::D A=reshape((/1,3,6,4,9,8/),(/2,3/)) B=reshape((/2,0,6,4,9,7/),(/2,3/)) C=any(A==B,dim=1) !C=(F,T,T) D=any(A==B,dim=2) !D=(T,T) print*,"C= ",C print*,"D= ",D any(A = = B,dim=1) testa para ver se algum elemento de cada coluna da matriz A é igual ao elemento correspondente na matriz B. O resultado é (.FALSE.,.TRUE.,.TRUE.) porque: 1ª coluna não tem nenhum elemento igual 2ª e 3ª colunas tem pelo menos 1 elemento igual any(A = = B,dim=2) testa para ver se algum elemento de cada linha da matriz A é igual ao elemento correspondente na matriz B. O resultado é (.TRUE.,.TRUE.) porque: 1ª linha tem pelo menos um elemento igual 2ª linha tem pelo menos um elemento igual

12	**asin(x)**
Descrição:	Calcula o arco-seno do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real e estar entre: $-1 \le x \le +1$
Resultado:	Mesmo que **x**. O resultado é o valor $\sin^{-1}(x)$ expresso em radianos na faixa: $-\frac{\pi}{2} \leqslant asin(x) \leqslant +\frac{\pi}{2}$

Exemplos:	`real::theta_r, & !ângulo em radianos`
	`  theta_g, & !ângulo em graus`
	`  x !seno do ângulo (60°)`
	`x = 0.8660254`
	`theta_r = asin(x) !theta_r = 1,0471975`
	`theta_g = theta_r*(180.0/3.14159) !theta_g = 60.000042`

13	**associated(point [,target])**
Descrição:	Retorna o estado da associação do ponteiro
Classe:	Função inquisitória
Argumentos:	**point** tem que ser um ponteiro (de qualquer tipo) **target** (opcional) tem que ser um ponteiro ou alvo
Resultado:	Tipo escalar lógico. Se somente **point** é usado, o resultado é verdadeiro (.TRUE.) se ele está associado com um alvo, senão é falso. Se **target** também é usado e é um alvo, o resultado é verdadeiro se **point** está associado com **target**, caso contrário é falso. Se **target** é um ponteiro, o resultado é verdadeiro se ambos, **point** e **target** estão associados com o mesmo **target**, caso contrário é falso

Exemplos:	`real,pointer::p,q`
	`real,target::x,y`
	`logical::aponta`
	`x=5.0`
	`y=2.0`
	`aponta=associated(p) ! aponta = F`
	`p=>x ! agora p aponta para x`
	`aponta=associated(p,x) ! aponta = T`
	`aponta=associated(p,y) ! aponta = F`

14	**atan(x)**
Descrição:	Calcula o arco-tangente do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real
Resultado:	Mesmo que **x**. o valor está na faixa $-\frac{\pi}{2} \leqslant atan(x) \leqslant +\frac{\pi}{2}$

Exemplos:	`real:: theta_r, & !ângulo em radianos`
	`  theta_g, & !ângulo em graus`
	`  x !tangente do ângulo`
	`x=1.7320508 !tangente de 60°`
	`theta_r = atan(x) !theta_r = 1.4071976`
	`theta_g = theta_r*(180.0/3.14159) !theta_g = 60.0000496`

15	**atan2(x,y)**
Descrição:	Calcula o arco-tangente em radianos. O resultado é o valor principal do argumento de um número complexo não nulo (x, y)
Classe:	Função elementar
Argumentos:	**y** tem que ser real **x** tem que ser do mesmo tipo e do mesmo parâmetro de tipo (kind parameter) que **y**. Se **x** tem valor zero **y** não pode ser zero
Resultado:	resultado é do mesmo tipo que x e está expresso em radianos. O resultado está entre $-\pi$ e $+\pi$. Se x ≠ 0, o resultado é igual a $\arctan\left(\frac{y}{x}\right)$ se y > 0, o resultado é positivo se y < 0, o resultado é negativo se y = 0, o resultado é 0 (se x > 0) ou positivo $+\pi$ (se x < 0) se x = 0, o valor absoluto do resultado é $\frac{\pi}{2}$
Exemplos:	real::x, y, theta_r, theta_g x=1.73205 y=1.0 theta_r = atan2(x,y) ! theta_r = 1.0471973 theta_g = theta_r*(180.0/3.14159) ! theta_g = 60.0000381 ! theta_r = atan2(-1.73205,1.0) ! theta_r = -1.0471973 theta_g = theta_r*(180.0/3.14159) ! theta_g = -60.0000381 ! theta_r = atan2(18.9,3.3) ! theta_r = 1.397936 theta_g = theta_r*(180.0/3.14159) ! theta_g = 80.09589

16	**bit_size(I)**
Descrição:	Retorna o número de bits num inteiro
Classe:	Função inquisitória
Argumentos:	**I** tem que ser do tipo inteiro
Resultado:	Tipo escalar inteiro. O resultado é o número de bits para o inteiro
Exemplos:	integer:: I,N I = 10 N = bit_size(I) !N=32 precisão simples 32 bits integer,parameter::dp=selected_int_kind(15) integer(kind=dp):: I integer::N I = 10 N = bit_size(I) !N=64 precisão dupla 64 bits

17	**btest(I, pos)**
Descrição:	Testa o valor de um bit num inteiro
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro **pos** tipo inteiro, não negativo, menor que **bit_size(I)**
Resultado:	Resultado é lógico. O resultado é verdadeiro (.TRUE.) se o bit **pos** de **I** contém o valor 1. O resultado é falso (.FALSE.) se o bit **pos** contém o valor 0
Exemplos:	número inteiro de precisão simples utiliza 32 bits o inteiro 8 é representado em binário por: 00000000000000000000000000001000 **obs:** contagem dos bits inicia no zero, da direita para a esquerda) 33 2 1 0 10 0 0 | | | | 00000000000000000000000000001000 --> número 8 integer::I logical::res I=8 res=btest(I,3) ! res=T integer::I logical::res I=6842 ! 6842 = 00000000000000000011010101111010 res=btest(I,6) ! res=F res=btest(I,9) ! res=T integer::B, j B=6842 print*,"B=",(btest(B,j),j=31,0,-1) resulta em: B= F F F F F F F F F F F F F F F F T T F T F T F T T T F T F que representa o inteiro B= 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 1 0 1 1 1 0 1 0

18	**ceiling(A [,kind])**
Descrição:	Retorna o menor inteiro maior que ou igual ao do argumento
Classe:	Função elementar
Argumentos:	**A** tem que ser do tipo real
Resultado:	Tipo inteiro. O resultado é um valor igual ao menor inteiro maior que ou igual a **A**
Exemplos:	real:: x integer:: R x = -2.7 R = ceiling(x) ! -2 x = 3.2 R = ceiling(x) ! -4

19	**char(I [,kind])**
Descrição:	Retorna o caractere de uma dada posição na tabela ascii. É o inverso da função **ichar**
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro, com valor entre 0 e (n - 1), onde n é o número de caracteres da tabela ascii **kind** (opcional) expressão escalar inteira de inicialização
Resultado:	resultado é do tipo caractere com comprimento 1. O resultado é o caractere que está na posição I da tabela ascii
Exemplos:	character(len=1):: resultado integer:: C C = 76 resultado = char(C) ! resultado = L resultado = char(97) ! resulatdo = a

20	**cmplx(x [,y,kind])**
Descrição:	Converte o argumento para o tipo complexo
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo inteiro, real ou complexo **y** (opcional) real ou inteiro. Não pode ser usado se x for complexo **kind** (opcional) expressão escalar inteira de inicialização
Resultado:	Tipo complexo. Se **kind** está presente, o tipo é especificado por ele, caso contrário é do tipo real default. Se somente um argumento não complexo é usado, ele é convertido na parte real do número complexo, sendo atribuído o valor 0 à parte imaginária. Se **y** não existe e **x** é complexo, tudo se passa como se o **y** existisse e o seu valor fosse **aimag(x)**. Se dois números não complexos são usados, o valor complexo é produzido colocando-se o primeiro argumento na parte real e o segundo na parte imaginária
Exemplos:	integer::I real::x complex::C1,C2,C3 I=-3 x=84.7 C1=cmplx(-3) !C1=(-3.0,0.0) C2=cmplx(x) !C2=(84.7,0.0) C3=cmplx(I,x) !C3=(-3.0,84.7)

21	**conjg(z)**
Descrição:	Calcula o conjugado do número complexo
Classe:	Função elementar
Argumentos:	**z** tem que ser do tipo complexo
Resultado:	Tipo complexo. Se **z** tem o valor (x, y) o resultado tem o valor **(x, -y)**
Exemplos:	complex::A,CA A=(2.0,3.0) CA=conjg(A) ! CA=(2.0,-3.0)

22	**cos(x)**
Descrição:	Retorna o co-seno do argumento em radianos
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real ou complexo
Resultado:	Mesmo tipo de **x**. **x** tem que ser em radianos. Se **x** for complexo, sua parte real é vista como sendo o valor em radianos
Exemplos:	real:: theta1,theta2 real:: C1,C2 theta1=1.989670 !1.989670=1140 theta2=0.6981317 !0.6981317=400 C1=cos(theta1) !C1=?0.4067318 C2=cos(theta2) !C2= 0.7660444

23	**cosh(x)**
Descrição:	Retorna o co-seno hiperbólico do argumento em radianos
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real
Resultado:	Mesmo tipo de **x**. **x** é o co-seno hiperbólico do valor em radianos
Exemplos:	real:: theta1,theta2 real:: C1,C2 theta1=1.989670 !1.989670=114o theta2=0.6981317 !0.6981317=40o C1=cosh(theta1) !C1=3.724931 C2=cosh(theta2) !C2=1.253754

24	**count(mask [,dim])**
Descrição:	Conta o número de elementos verdadeiros numa matriz ou numa dimensão da matriz
Classe:	Função transformacional
Argumentos:	**mask** tem que ser uma matriz lógica **dim** (opcional) expressão escalar inteira com o valor entre 1 e r, onde r é a dimensão (rank) de **mask**
Resultado:	Matriz ou escalar do tipo inteiro. O resultado é escalar se **dim** é omitido ou **mask** é um vetor (rank=1). Um escalar tem valor igual ao número de elementos verdadeiros de **mask**. Se **mask** tem dimensão zero, o resultado é zero. A matriz resultado tem dimensão menor em 1 que **mask** e a sua forma (shape) é a mesma de **mask**. Cada elemento na matriz resultado é igual ao número de elementos que são verdadeiros (.TRUE.) em uma matriz de uma dimensão definida por **mask**

Exemplos:	Matrizes: $A = \begin{bmatrix} 1 & 6 & 9 \\ 3 & 4 & 8 \end{bmatrix}$ $B = \begin{bmatrix} 2 & 6 & 9 \\ 0 & 4 & 7 \end{bmatrix}$
	`logical,dimension(3)::C1, C2`
	`integer::resultado1, resultado2`
	`  C1=(/.true.,.false.,.true./)`
	`  C2=(/.true.,.true.,.true./)`
	`  resultado1=count(C1)  !resultado1=2`
	`  resultado2=count(C2)  !resultado2=3`
	
	`count(C1) tem valor 2 porque 2 elementos são verdadeiros`
	`count(C2) tem valor 3 porque 3 elementos são verdadeiros`
	
	`Considere as matrizes: A e B`
	
	`integer,dimension(2,3)::A, B`
	`integer,dimension(3)::resultado`
	`  A=reshape((/1,3,6,4,9,8/),(/2,3/))`
	`  B=reshape((/2,0,6,4,9,7/),(/2,3/))`
	`  resultado=count(A/=B,dim=1)  !resultado=(2,0,1)`
	
	`count(A/=B,dim=1) testa para ver quantos elementos em cada coluna de A não é igual ao elemento correspondente na matriz B. O resultado é (2,0,1) porque:`
	`  1ª coluna não tem nenhum elemento igual`
	`  2ª coluna são todos iguais`
	`  3ª coluna tem pelo menos 1 elemento diferente`
	
	`Observação: matriz lógica (A/=B)= | T F F |`
	`  | T F T |`

25	**cpu_time(time)**
Descrição:	Retorna o tempo do processador
Classe:	Sub-rotina não elementar
Argumentos:	**time** tem que ser escalar e do tipo real. É um argumento intent(out)
Resultado:	Valor do tempo no computador. Se o processador não puder retornar um tempo que tenha significado, um valor negativo será retornado

Exemplos:	`real:: t1, t2`
	`  ..................`
	`  call cpu_time(t1)`
	`  !`
	`  ! <<< códigos aqui >>>`
	`  !`
	`  call cpu_time(t2)`
	`  ..................  ! t2-t1 será igual ao tempo que o programa levou para executar os`
	`  ! códigos que estão colocados entre a chamada da sub-rotina com`
	`  ! t1 e a chamada da sub-rotina com t2`

26	**cshift(array, shift [,dim])**
Descrição:	Realiza um deslocamento circular em um vetor (rank=1) ou deslocamento cíclico (permutação cíclica) de todos os elementos de uma seção de ordem 1 de uma matriz
Classe:	Função transformacional
Argumentos:	**array** tem que ser uma matriz (qualquer tipo de dado) **shift** tem que ser um escalar inteiro ou uma matriz com um uma dimensão menor em 1 que **array** e forma (shape) igual a forma de **array** **dim** (opcional) tem que ser um escalar inteiro com um valor na faixa de 1 a r, onde r é a dimensão (rank) de **array**. Se **dim** for omitida ela é assumida ser igual a 1
Resultado:	resultado é uma matriz com o mesmo tipo e mesmo parâmetro de tipo (kind) e forma de **array**. Se **array** é um vetor (tem dimensão 1), o elemento i do resultado é array(1+modulo(i+shift-1, size(array))). O mesmo deslocamento é aplicado a cada elemento. Se **array** tem dimensão (rank) maior que 1, cada seção do resultado é deslocado • pelo valor de **shift**, se **shift** for escalar • de acordo com o valor em **shift**, se **shift** for uma matriz O valor de **shift** determina a quantidade e a direção da permutação circular. Quando **shift** é positivo isto produz um deslocamento para a esquerda (linhas) ou para cima(colunas). Um valor negativo de **shift** causa deslocamento para a direita (linhas) ou para baixo (colunas). Se **shift** é zero não há permutação
Exemplos:	integer,dimension(6)::V, permutacao1, permutacao2 V=(/1,2,3,4,5,6/) permutacao1 = cshift(V,shift=2) !permutacao1=(3,4,5,6,1,2) permutacao2 = cshift(V,shift=-2) !permutacao2=(5,6,1,2,3,4) cshift(V,shift=2) faz a permutação circular de v para a esquerda de 2 posições, produzindo o valor (3,4,5,6,1,2) cshift(V,shift=-2) faz a permutação circular de v para a direita de 2 posições, produzindo o valor (5,6,1,2,3,4) Considere a matriz: $A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$ integer,dimension(3,3)::A, desloc1, desloc2, desloc3 A=reshape((/1,2,3,4,5,6,7,8,9/),(/3,3/)) desloc1 = cshift(A,shift=1,dim=1) desloc2 = cshift(A,shift=1,dim=2) desloc3 = cshift(A,shift=(/1,-2,0/),dim=2) cshift(A,shift=1,dim=1) produz a matriz desloc1= | 2 5 8 | | 3 6 9 | | 1 4 7 | cada linha é deslocados para a cima de 1 posição. O deslocamento é circular cshift(A,shift=1,dim=2) produz a matriz desloc2= | 4 7 1 | | 5 8 2 | | 6 9 3 | cada coluna é deslocados para esquerda 1 posição. O deslocamento é circular cshift(A,shift=(/1,-1,0/),dim=2) produz a matriz desloc3= | 4 7 1 | | 8 2 5 | | 3 6 9 | 1°) cada coluna na linha 1 é deslocado para esquerda 1 posição 2°) cada coluna na linha 2 é deslocado para a direita 1 posição 3°) as colunas da linha 3 não são deslocadas Todos os deslocamentos são circulares

27	**date_and_time([date, time, zone, values])**
Descrição:	Retorna a data e o tempo no formato definido na ISO 8601:1988
Classe:	Sub-rotina não elementar
Argumentos:	4 argumentos opcionais: **date** tem que ser um escalar tipo caractere; comprimento de no mínimo 8 para conter o valor completo; seu formato é ccyymmdd, onde: cc = século yy = ano dentro do século mm = mês dentro do ano dd = dia dentro do mês **time** tem que ser um escalar tipo caractere; comprimento de pelo menos 10 caracteres para conter todo o valor; formato: hhmmss.sss onde: hh = hora do dia mm = minutos da hora ss.sss = segundos e milisegundos do minuto **zone** tem que ser um escalar tipo caractere; comprimento de pelo menos 5 caracteres; valores da forma: ±hhmm, onde: hh = horas mm = minutos que marcam a diferença com respeito à UTC (Coordinated Universal Time **values** matriz de rank1 tipo inteiro; comprimento de pelo menos 8; Valores retornados são: values(1) = ano em 4 dígitos values(2) = mês do ano values(3) = dia do ano values(4) = diferença do tempo com respeito a UTC em minutos values(5) = hora do dia (0 a 23) values(6) = minutos da hora (0 a 59) values(7) = segundos do minuto (0 a 59) values(8) = milisegundos do segundo (0 a 999)
Resultado:	

Exemplos:	`character(len=8)::date`
	`character(len=10)::time`
	`character(len=5)::zone`
	`integer,dimension(8)::values`
	
	`call DATE_AND_TIME(date, time, zone, values)`
	`print*,"data=",date`
	`print*,"tempo=",time`
	`print*,"zona=",zone`
	`print*,"values(1)=",values(1)`
	`print*,"values(2)=",values(2)`
	`print*,"values(3)=",values(3)`
	`print*,"values(4)=",values(4)`
	`print*,"values(5)=",values(5)`
	`print*,"values(6)=",values(6)`
	`print*,"values(7)=",values(7)`
	`print*,"values(8)=",values(8)`
	
	`!Executando o programas em 25 de março de 2006 as 14h 21min 15s 440ms tem-se:`
	
	`data=20060325`
	`tempo=142115.440`
	`zona=-0300`
	`values(1)= 2006`
	`values(2)= 3`
	`values(3)= 25`
	`values(4)= -180`
	`values(5)= 14`
	`values(6)= 21`
	`values(7)= 15`
	`values(8)= 440`

28	**dble(A)**
Descrição:	Converte um número para um número de precisão dupla
Classe:	Função elementar
Argumentos:	**A** tem que ser inteiro, real ou complexo
Resultado:	O resultado é o mesmo número em precisão dupla

Exemplos:	`real:: x`
	`  x=10.0`
	`  print*,"    x=", x`
	`  print*,"dupla=", dble(x)`
	
	`!saida do programa:`
	
	`    x = 10.00000`
	`dupla = 10.0000000000000`

29	**digits(x)**
Descrição:	Retorna o número de algarismos significativos na mantissa (ou significante) para o número de mesmo parâmetro de tipo (kind parameter) que o argumento
Classe:	Função inquisitória
Argumentos:	**x** tem que ser do tipo inteiro ou real
Resultado:	Escalar do tipo inteiro. O resultado tem o valor **q** se **x** é do tipo inteiro; tem valor **p** se **x** é real **q** representa o número de dígitos **p** representa o número de dígitos do significante

Exemplos:	`integer:: I, q, p`
	`real:: x`
	`  q = digits(I)          ! q = 31  inteiro de precisão simples`
	`  p = digits(x)          ! p = 24  real de precisão simples`
	
	`integer,parameter::idp = selected_int_kind(10)`
	`integer,parameter::dp = selected_real_kind(14)`
	`integer(kind=idp):: I`
	`integer::p, q`
	`real(kind=dp):: x`
	`  q=digits(I)          ! q = 63   inteiro de precisão dupla`
	`  p=digits(x)          ! p = 53   real de precisão dupla`

30	**dim(x, y)**
Descrição:	Retorna diferença entre dois números (se a diferença for positiva)
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo inteiro ou real **y** tem que ser de mesmo tipo e mesmo parâmetro de tipo (kind parameter) de **x**
Resultado:	Mesmo tipo de **x**. O valor do resultado é (x – y) se **x** é maior que **y**; se (y > x) o resultado é zero

Exemplos:	`real:: x, y, z1, z2`
	`integer:: a, b, r1, r2`
	`  x=10.0`
	`  y=2.0`
	`  z1=dim(x,y)      !z1=8.0`
	`  z2=dim(y,x)      !z2=0.0`
	`  a=-40`
	`  b=15`
	`  r1=dim(a,b)       !0`
	`  r2=dim(b,a)       !55`

31	**dot_product(vector_a, vector_b)**
Descrição:	Calcula o produto vetorial de vetores numéricos ou lógicos
Classe:	Função transformacional
Argumentos:	**vector_a** tem que ser um vetor numérico (inteiro, real ou complexo) ou vetor lógico **vector_b** tem que ser um vetor do mesmo tipo que **vector_a**. Tem que ter o mesmo tamanho do **vector_a**
Resultado:	Escalar: O tipo depende do tipo de **vector_a**. Se **vector_a** é inteiro ou real, o resultado é **sum(vector_a * vector_b)** Se **vector_a** é complexo, o resultado é: **sum(conj(vector_a * vector_b))** Se vector_a é lógico, o resultado tem o valor **any(vector_a .and. vector_b)** Se os dois vetores tem tamanho zero, o resultado é zero se o vetor é do tipo numérico e falso se o vetor é do tipo lógico
Exemplos:	integer,dimension(3)::A,B logical,dimension(2)::D,E integer::C logical::F A=(/1,2,3/) B=(/3,4,5/) C=dot_product(A,B) !C=26 (1x3)+(2x4)+(3x5)=26 D=(/.true.,.false./) E=(/.false.,.true./) F=dot_product(D,E) !F=F

32	**dprod(x, y)**
Descrição:	Produz um produto de precisão dupla
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real **y** tem que ser do tipo real
Resultado:	Real de dupla precisão. O resultado é igual a x*y
Exemplos:	real::x, y x = 10.0 y = 20.0 print*," x*y=", x*y print*,"dupla=", dprod(x,y) !saída do programa: x = 200.0000 dupla = 200.00000000000

33	**eoshift(array, shift [,boundary, dim])**
Descrição:	Realiza um deslocamento end-off (não cíclico) ou realiza um deslocamento não cíclico em uma seção de dimensão 1 em uma dimensão de uma matriz de dimensão 2 ou maior. Elementos são deslocados para fora da borda da seção e copias dos valores da fronteira são preenchidas na outra borda. Diferentes seções podem ter diferentes valores de bordas e podem ser deslocados de valores diferentes em direções diferentes
Classe:	Função transformacional
Argumentos:	**array** tem que ser uma matriz **shift** tem que ser um escalar inteiro ou uma matriz com dimensão (rank) menor em 1 que **array** **boundary** (opcional) tem que ter mesmo tipo e forma que **array**. É um escalar ou uma matriz com uma dimensão menor em 1 que **array** Se **boundary** não é especificado ,ele segue os seguintes valores defaults: tipo matriz — valor fronteira inteiro — 0 real — 0.0 complexo — (0.0, 0.0) lógico — .FALSE. caractere(len) — len espaços em brancos **dim** (opcional) tem que ser um escalar inteiro com o valor entre 1 e n, onde n é a dimensão (rank) de **array**. Se **dim** é omitido ele é assumido ser 1
Resultado:	Uma matriz com mesmo tipo e mesmo parâmetro de tipo (kind parameter) e forma de **array**. Se **array** é um vetor (rank=1) o mesmo deslocamento é aplicado a cada elemento. Se um elemento é deslocado fora da borda do vetor, o valor **boundary** é colocado no lugar vago. Se **array** é uma matriz com dimensão igual ou maior que 2, cada seção do resultado é deslocado: ▪ pelo valor de **shift**, se **shift** é escalar ▪ de acordo com o valor de **shift**, se **shift** é uma matriz Se um elemento é deslocado para fora da seção, o valor **boundary** é colocado na outra borda da seção. O valor **shift** determina a quantidade e a direção do deslocamento. Um **shift** positivo produz um deslocamento para a esquerda (nas linhas) e para cima (nas colunas). Um **shift** negativo causa deslocamento para a direita (linhas) e para baixo (colunas)

Exemplos:	`integer,dimension(6)::V,D`
	`  V=(/1,2,3,4,5,6/)`
	`  D=eoshift(V,shift=2)  !D=(3,4,5,6,0,0)`
	
	Considere a matriz: $A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$
	
	`integer,dimension(3,3)::A, D1, D2`
	`  A=reshape((/1,2,3,4,5,6,7,8,9/),(/3,3/))`
	`  D1=eoshift(A,shift=1,dim=1)`
	`  D2=eoshift(A,shift=2,dim=2)`
	
	`  eoshift(A,shift=1,dim=1)  D1= | 2 5 8 |`
	`  | 3 6 9 |`
	`  | 0 0 0 |`
	As linhas são deslocadas para cima 1 posição. Como o deslocamento não é circular, as linhas movidas para fora da matriz são perdidas e os espaços liberados são preenchidos com zeros (argumento boundary não foi utilizado)
	
	`  eoshift(A,shift=-2,dim=2)  D2= | 0 0 1 |`
	`  | 0 0 2 |`
	`  | 0 0 3 |`
	As colunas são deslocado para a direita 2 posição e os espaços preenchidos com zeros
	
	`integer,dimension(3,3)::A,D`
	`integer,dimension(2,2)::E`
	`  A=reshape((/1,2,3,4,5,6,7,8,9/),(/3,3/))`
	`  E=reshape((/20,20,20,20/),(/2,2/))`
	`  D=eoshift(A,shift=1,boundary=E,dim=2)`
	
	`eoshift(A,shift=1,boundary=E,dim=2)  D= | 4 7 20 |`
	`  | 5 8 20 |`
	`  | 6 9 20 |`
	As colunas são deslocado para a esquerda 1 posição e os espaços desocupados são preenchidos com os valores da matriz E

34	**epsilon(x)**
Descrição:	Retorna um valor positivo, que é quase sempre muito pequeno quando comparado com a unidade
Classe:	Função inquisitória
Argumentos:	**x** tem que ser do tipo real. Pode ser escalar ou matriz
Resultado:	Escalar do mesmo tipo e mesmo parâmetro de tipo (kind parameter)que **x**. O resultado é o valor b^{1-p} onde b é a base e p é o número de algarismos no significante (mantissa)
Exemplos:	real,parameter::dp=selected_real_kind=(15) real:: x, E1, E2 real(kind=dp)::Y E1 = epsilon(x) !E1 = 2^{-23} = 1.1920929E-07 E2 = epsilon(y) !E2 = 2^{-52} = 2.2204460E-16

35	**exp(x)**
Descrição:	Calcula a função exponencial do argumento
Classe:	Função elementar
Argumentos:	**x** pode ser do tipo real ou complexo
Resultado:	Mesmo tipo de **x**. O resultado é e^x . Se **x** é complexo, sua parte imaginária é em radianos
Exemplos:	real::x, y x=2.0 y=exp(x) ! y=7.389056

36	**exponent(x)**
Descrição:	Retorna o expoente do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real
Resultado:	Tipo inteiro. Se **x** não é zero o resultado é o expoente do número. O resultado é indefinido se estiver fora da faixa dos números inteiros em uso. Se x é zero o expoente de **x** é zero
Exemplos:	real::x,y x=2.0 y=exponent(x) !y=2 x=16.1 y=exponent(x) !y=5

37	**floor(A [,kind])**
Descrição:	Retorna o maior inteiro igual ou menor que seu argumento
Classe:	Função elementar
Argumentos:	**A** tem que ser do tipo real
Resultado:	Tipo inteiro default ou inteiro com tipo especificado por kind. O resultado é igual ao maior inteiro igual ou menor que **A**
Exemplos:	real:: x integer::R x=4.8 R=floor(x) !R=4 x=-5.6 R=floor(x) !R=-6

38	**fraction(x)**
Descrição:	Retorna a parte fracionária do modelo numérico utilizado na representação do número real **x**
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real
Resultado:	Tipo real. O resultado vale $X \times b^{-e}$ onde b é a base enquanto e é um valor inteiro entre e_{min} e e_{max} que depende do modelo numérico do argumento. Se x é zero o resultado é zero. valor de e e_{min} e_{max} precisão simples -125 128 precisão dupla -1021 1024 estendida -16381 16384
Exemplos:	real::x, R x=3.0 R=fraction(x) ! R=0.75

39	**huge(x)**
Descrição:	Retorna o maior número do modelo numérico usado no argumento
Classe:	Função inquisitória
Argumentos:	**x** tem que ser do tipo real. Pode ser escalar ou matriz
Resultado:	Mesmo tipo e mesmo parâmetro de tipo (kind) que **x**. Se **x** é inteiro o resultado é $r^q - 1$. Se **x** for real vale $\left(1 - b^{-p}\right) b^{e_{max}}$. Para os inteiros, **q** representa a base (números binários b = 2), **p** a quantidade de números utilizados no significante (mantissa) e a letra **e** é um inteiro entre e_{min} e e_{max} valor de e e_{min} e_{max} real precisão simples -125 128 real precisão dupla -1021 1024
Exemplos:	integer,parameter::dp=selected_real_kind(15) real::x_sp, h1 real(kind=dp)::x_dp, h2 h1=huge(x_sp) !h1 =(1- 2^{-24})x2^{128} = 3.4028235E+38 h2=huge(x_dp) !h2 =(1- 2^{-52})x2^{1024}= 1.797693134862316E+308

40	**iachar(C)**
Descrição:	Retorna o número que define a posição do caractere na tabela ascii
Classe:	Função elementar
Argumentos:	**C** tem que ser do tipo caractere de tamanho 1
Resultado:	Tipo inteiro. O resultado a posição do caractere na tabela ascii. seu valor está entre $0 \leq iachar(c) \leq 127$
Exemplos:	integer:: P character(len=1)::C C = "W" p=iachar(C) !p=87

41	**iand(I, J)**
Descrição:	Calcula um e lógico
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro **J** inteiro do mesmo parâmetro de tipo que **I**
Resultado:	Mesmo tipo de **I**. O resultado é obtido combinando I e J bit a bit conforme a tabela abaixo. I J iand(I, J) 1 1 1 1 0 0 0 1 0 0 0 0
Exemplos:	iand(2, 3) retorna 2 00000000000000000000000000000010 = número 2 00000000000000000000000000000011 = número 3 00000000000000000000000000000010 = combinação bit a bit que dá o número 2

42	**ibclr(I, pos)**
Descrição:	Escreve o valor zero no bit especificado do argumento (limpa o bit)
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro **pos** tem que ser do tipo inteiro. Não pode ser negativo e tem que ser menor que **bit_size(i)**
Resultado:	Mesmo tipo que **I**. O resultado tem o mesmo valor que a sequência de bits de **I**, exceto que o bit **pos** é recebe o valor zero
Exemplos:	ibclr(18, 1) vale 16 00000000000000000000000000010010 = inteiro 18 00000000000000000000000000010000 bit 1 foi zerado, então o inteiro vale 16

43	**ibits(I, pos, len)**
Descrição:	Extrai uma sequência de bits (um campo de bits) do argumento
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro **pos** tem que ser do tipo inteiro. Não pode ser negativo e **pos+len** tem que ser igual ou menor que **bit_size(i)** **len** tem que ser do tipo inteiro. Não pode ser negativo.
Resultado:	Resultado mesmo tipo de **I**. O resultado tem o valor da sequência de **len** bits em **I**, iniciando na posição **pos** ajustado a direita e com todos os outros bits zeros
Exemplos:	ibits(12, 1, 4) tem o valor 6 00000000000000000000000000001100 = inteiro 12 00000000000000000000000000000110 = inteiro 6 ibits(173, 2, 7) tem o valor 21 00000000000000000000000010101101 = inteiro 173 00000000000000000000000000101011 = inteiro 43

44	**ibset(I, pos)**
Descrição:	Escreve o valor 1 no bit especificado do argumento
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro **pos** tem que ser do tipo inteiro. Não pode ser negativo e tem que ser menor que **bit_size(i)**
Resultado:	Resultado mesmo tipo de **I.** O resultado tem o valor da sequência de bits exceto o bit **pos** de **I** assume o valor 1
Exemplos:	ibset(8,1) vale 10 00000000000000000000000000001000 = inteiro 8 00000000000000000000000000001010 = inteiro 10

45	**ichar(C)**
Descrição:	Retorna a posição do caractere especificado no conjunto de caracteres usado pelo processador
Classe:	Função elementar
Argumentos:	**C** tem que ser do tipo caractere de comprimento 1
Resultado:	Tipo inteiro. O resultado é a posição do caractere na tabela de caracteres usado pelo processador
Exemplos:	ichar("W") vale 87 ichar("#") vale 35

46	**ieor(I, J)**
Descrição:	Calcula um ou lógico
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro **J** inteiro do mesmo parâmetro de tipo que **I**
Resultado:	Mesmo tipo de **I.** O resultado é obtido combinando I e J bit a bit conforme a tabela abaixo. I J ieor(I, J) 1 1 0 1 0 1 0 1 1 0 0 0
Exemplos:	integer::A, B, C A = 1 B = 4 C = ieor(A, B) !C = 5 00000000000000000000000000000001 = número 1 00000000000000000000000000000100 = número 4 00000000000000000000000000000101 = combinação bit a bit que dá o número 5

47	**index(string, substring [,back])**
Descrição:	Retorna a posição inicial de um substring dentro de um string
Classe:	Função elementar
Argumentos:	**string** tem que ser do tipo caractere **substring** tem que ser do tipo caractere **back** (opcional) tem que ser do tipo lógico
Resultado:	Tipo inteiro. Se **back** não aparece (ou aparece com o valor falso) o valor obtido é o menor valor de **I** tal que **string(I : I+len(substring)-1) = substring** ou zero se este valor não existir. **len(string)<len(substring)**, o valor zero é retornado. **len(substring) =zero**, o valor 1 é obtido. Se **back** aparece com o valor verdadeiro, o valor retornado é o maior valor de **I** tal que **string(I : I+len(substring)-1) = substring** (ou zero se este valor não existe). **len(string)<len(substring)**, o valor zero é retornado. **len(substring) =zero**, len(string)+1 é o valor obtido

Exemplos:	`character(len=8)::C`
	`integer::R`
	`  C="converte"`
	`  R=index("converte","o",back=.true.)  !R = 2`
	`  R=index("converte","verte",back=.true.)  !R = 4`
	`  R=index("converte","",back=.true.)  !R = 9`

48	**int(A [,kind])**
Descrição:	Converte um valor para o tipo inteiro
Classe:	Função elementar
Argumentos:	**A** tem que ser do tipo inteiro, real ou complexo **kind** (opcional) tem que ser expressão escalar inteira de inicialização
Resultado:	Tipo inteiro. Se o parâmetro de tipo (kind parameter) estiver presente ele especifica o tipo de inteiro. Quando **kind** não estiver presente, se **A** é do tipo inteiro, **int(A) = A**. Se **A** é real e **|A|<1, int(A) = 0**. Se **A** é real e **|A|>=1, int(A)** é o inteiro cuja magnitude é o maior inteiro que não excede **A** e o sinal é o mesmo de **A**. Se **A** é complexo, **int(A) = A** é o valor aplicando as regras acima à parte real do número

Exemplos:	`complex::C=(5.4, 9.3)`
	`real:: x`
	`integer:: I`
	
	`  I=int(C)  !I = 5`
	`  x=7.0`
	`  I=int(x)  !I = 7`
	`  x=5.2`
	`  I=int(x)  !I = 5`
	`  x=0.8`
	`  I=int(x)  !I = 0`
	`  x=-9.7`
	`  I=int(x)  !I = -9`

49	**ior(I, J)**
Descrição:	Calcula um ou inclusivo nos bits correspondentes
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro **J** inteiro do mesmo tipo que I
Resultado:	Mesmo tipo de **I**. O resultado é obtido combinando I e J bit a bit conforme a tabela abaixo. I J ior(I, J) 1 1 1 1 0 1 0 1 1 0 0 0

Exemplos:	`integer:: A, B, C`
	`  A = 1`
	`  B = 4`
	`  C = ior(1, 4)  !C=5`
	
	`00000000000000000000000000000001 = número 1`
	`00000000000000000000000000000100 = número 4`
	`00000000000000000000000000000101 = combinação bit a bit que dá o número 5`

50	**ishft(I, shift)**
Descrição:	Realiza um deslocamento (para esquerda ou direita) dos bits do argumento
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro **shift** tem que ser inteiro. O valor absoluto de **shift** tem que ser menor ou igual a **bit_size(i)**
Resultado:	Mesmo tipo de **I**. O resultado tem o valor obtido por deslocamentos do bits de **I** de **shift** posições. Se **shift** é positivo, o deslocamento é para a esquerda, se negativo desloca para a direita. Se **shift** é zero nenhum deslocamento é feito. Bits deslocados para fora , pela esquerda ou pela direita, são perdidos. Zeros são deslocados para dentro, pelo lado oposto para preencher as posições vagas
Exemplos:	integer:: A, B, C A = 2 B = 1 C = ishft(A, B) !C = 4 00000000000000000000000000000010 = número 2 (desloca para esquerda) 00000000000000000000000000000100 = número 4 integer:: A, B, C A = 2 B = -1 C = ishft(A, B) !C = 1 00000000000000000000000000000010 = número 2 (desloca para a direita) 00000000000000000000000000000001 = número 1

51	**ishftc(I, shift [,size])**
Descrição:	Realiza uma permutação cíclica dos bits mais a direita do argumento
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro **shift** tem que se inteiro. O valor absoluto de **shift** tem que ser menor ou igual a **size** **size** (opcional) tem que se inteiro. O valor de **size** tem que ser positivo e não pode exceder **bit_size(i)**. Se **size** for omitido, ele é assumido ser tamanho igual a **bit_size(i)**
Resultado:	Mesmo tipo de **I**. Retorna o valor de I com os size bits mais a direita circularmente deslocados à direita **shift** bits. Se **shift** é negativo, o deslocamento é para a esquerda. Se **shift** é positivo, o deslocamento é para a direita. Se **shift** é zero, nenhum deslocamento é realizado
Exemplos:	integer::I, S, Z, R I = 4 S = 2 Z = 4 R = ishftc(I, S, Z) !R = 1 00000000000000000000000000000100 = número 4 00000000000000000000000000000001 = número 1

52	**kind(x)**
Descrição:	Retorna o valor do **parâmetro de tipo do tipo** especificado no argumento
Classe:	Função inquisitória
Argumentos:	**x** tem que ser de algum tipo intrínseco, pode ser escalar ou matriz
Resultado:	Resultado é um escalar do tipo inteiro. O resultado tem um valor igual ao valor do parâmetro de tipo do tipo **x**. Para um inteiro o parâmetro de tipo (kind parameter) usualmente é igual a 1, 2, 4, 8 (ou outros valores dependendo do compilador). Para um real o parâmetro de tipo (kind parameter) usualmente é igual a 4, 8, 10 (ou outros valores dependendo do processador em uso)

Exemplos:	`integer,parameter::i1=selected_int_kind(2)`
	`integer,parameter::i2=selected_int_kind(3)`
	`integer,parameter::i3=selected_int_kind(5)`
	`integer,parameter::i4=selected_int_kind(18)`
	`integer,parameter::r1=selected_real_kind(1,10)`
	`integer,parameter::r2=selected_real_kind(14,10)`
	`integer,parameter::r3=selected_real_kind(18,10)`
	`integer,parameter::r4=selected_real_kind(18,2000)`
	`integer::k1, k2, k3, k4`
	`integer(kind=i1)::a1`
	`integer(kind=i2)::a2`
	`integer(kind=i3)::a3`
	`integer(kind=i4)::a4`
	`real(kind=r1)::b1`
	`real(kind=r2)::b2`
	`real(kind=r3)::b3`
	`real(kind=r4)::b4`
	`  k1=kind(a1)  !k1=1`
	`  k2=kind(a2)  !k2=2`
	`  k3=kind(a3)  !k3=4`
	`  k4=kind(a4)  !k4=8`
	`!`
	`  k1=kind(b1)  !k1=4`
	`  k2=kind(b2)  !k2=8`
	`  k3=kind(b3)  !k3=10`
	`  k4=kind(b4)  !k4=10`

53	**lbound(array [,dim])**
Descrição:	Retorna o limite inferior de todas as dimensões de uma matriz, ou o limite inferior de uma dimensão especificada
Classe:	Função inquisitória
Argumentos:	**array** tem que ser uma matriz (de qualquer tipo). Não pode ser uma matriz alocável que não esteja alocada, ou um ponteiro não alocado **dim** (opcional) tem que ser um escalar inteiro com o valor na faixa 1 a n, onde n é a dimensão (rank) da matriz
Resultado:	Tipo inteiro. Se **dim** está presente, o resultado é um escalar. Se não presente o resultado é um vetor com um elemento para cada dimensão de **array**. Cada elemento no resultado corresponde a uma dimensão de **array**. Se **array** é uma seção de um matriz ou uma expressão matricial que não é a matriz inteira, cada elemento do resultado tem o valor 1. Se **array** é uma matriz inteira **lbound(array, dim)** tem um valor igual ao valor inferior do subscrito **dim** de **array** (se **array(dim)** não é nulo). Se **array(dim)** é zero, o elemento correspondente do resultado é 1

Exemplos:	`real,dimension(3,6:9)::a`
	`real,dimension(2:8,-5:15)::b`
	`integer,dimension(2)::c,d,e`
	`integer::F`
	`  c=lbound(a)  !c=(1,6)`
	`  F=lbound(a,dim=2)  !F=6`
	`  d=lbound(b)  !d=(2,-5)`
	`  e=lbound(b(5:8,:))  !e=(1,1)  argumento é uma seção de matriz`

54	**len(string)**
Descrição:	Retorna o comprimento de uma expressão caractere
Classe:	Função inquisitória
Argumentos:	**string** tem que ser do tipo caractere. Pode ser um escalar ou matriz
Resultado:	Tipo inteiro. O resultado é igual ao número de caracteres em **string** (se for escalar) ou em um elemento de **string** (se for uma matriz)

Exemplos:	`character(len=6)::A`
	`integer::B`
	`  A="OLA"`
	`  B=len(A)  !B=12`
	`  B=len("computador")  !B=10 matriz`

55	**len_trim(string)**
Descrição:	Retorna o comprimento de um argumento caractere sem contar os brancos posteriores
Classe:	Função elementar
Argumentos:	**string** tem que ser do tipo caractere
Resultado:	Tipo inteiro. O resultado é igual ao número de caracteres restantes depois dos brancos posteriores de **string** terem sido removidos. Se o argumento contém somente um espaço em branco o resultado é zero

Exemplos:	o símbolo _ foi utilizado para marcar espaço em branco
	
	`character(len=10)::texto`
	`integer::N`
	`  texto = "A_CASA_ _ _ _"`
	`  N=len_trim(texto)  !N = 6`

56	**lge(string_a, string_b)**
Descrição:	Determina se um string é lexicalmente maior que ou igual que outro string, baseado na tabela ascii, mesmo que o conjunto de caracteres do processador seja diferente
Classe:	Função elementar
Argumentos:	**string_a** tem que ser do tipo caractere **string_b** tem que ser do tipo caractere
Resultado:	Tipo lógico. Se os strings tem tamanhos diferentes, a comparação é feita como se o menor tivesse sido aumentado a direita com brancos até o tamanho do maior. O resultado é verdadeiro (.TRUE.) se os strings são iguais, ambos têm comprimento zero ou se o **string_a** segue o **string_b** na tabela ascii. Falso se for o contrário

Exemplos:	`character(len=3):: A, B, C`
	`character(len=5)::D`
	`logical::R1`
	`  A="one"`
	`  B="six"`
	`  R1=lge(A, B)  !R1 = F`
	`  C="two"`
	`  D="three"`
	`  R1=lge(C, D)  !R1 = T`

57	**lgt(string_a, string_b)**
Descrição:	Determina se um string é lexicalmente maior que ou igual que outro string, baseado na tabela ascii, mesmo que o conjunto de caracteres do processador seja diferente
Classe:	Função elementar
Argumentos:	**string_a** tem que ser do tipo caractere **string_b** tem que ser do tipo caractere
Resultado:	Tipo lógico. Se os strings tem tamanhos diferentes, a comparação é feita como se o menor tivesse sido aumentado a direita com brancos até o tamanho do maior. O resultado é verdadeiro (.TRUE.) se os strings são iguais, ambos têm comprimento zero ou se o **string_a** segue o **string_b** na tabela ascii. Falso se for o contrário

Exemplos:	`character(len=3):: A, B, C`
	`character(len=5)::D`
	`logical::R1`
	`  A="one"`
	`  B="six"`
	`  R1=lge(A, B)  !R1 = F`
	`  C="two"`
	`  D="three"`
	`  R1=lge(C, D)  !R1 = T`

58	**lle(string_a, string_b)**
Descrição:	Determina se um string é lexicalmente menor que ou igual que outro string, baseado na tabela ascii.
Classe:	Função elementar
Argumentos:	**string_a** tem que ser do tipo caractere **string_b** tem que ser do tipo caractere
Resultado:	Tipo lógico. Se os strings tem tamanhos diferentes, a comparação é feita como se o menor tivesse sido aumentado a direita com brancos até o tamanho do maior. O resultado é verdadeiro (.TRUE.) se os strings são iguais, ambos têm comprimento zero ou se o **string_a** antecede o **string_b** na tabela ascii. Falso se for o contrário.

Exemplos:	`character(len=3):: A,C`
	`character(len=4)::D`
	`character(len=5)::B`
	`logical::R1`
	`  A="two"`
	`  B="three"`
	`  R1=lle(A, B)  !R1 = F`
	`  C="one"`
	`  D="four"`
	`  R1=lle(C, D)  !R1 = F`

59	**llt(string_a, string_b)**
Descrição:	Determina se um string é lexicalmente menor que outro string, baseado na tabela ascii.
Classe:	Função elementar
Argumentos:	**string_a** tem que ser do tipo caractere **string_b** tem que ser do tipo caractere
Resultado:	Tipo lógico. Se os strings tem tamanhos diferentes, a comparação é feita como se o menor tivesse sido aumentado a direita com brancos até o tamanho do maior. O resultado é verdadeiro (.TRUE.) se o **string_a** antecede **string_b** na tabela ascii, falso caso contrário. Se os dois strings tiverem comprimento zero o resultado é falso

Exemplos:	`character(len=3):: A,B`
	`character(len=4)::C`
	`logical::R1`
	`  A="one"`
	`  B="six"`
	`  R1=llt(A, B)  !R1 = F`
	`  C="four"`
	`  R1=llt(A, C)  !R1 = T`

60	**log(x)**
Descrição:	Retorna o logaritmo natural do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser real ou complexo. Se **x** é real, seu valor tem que ser maior que zero. Se **x** é complexo seu valor não pode ser zero
Resultado:	Resultado de mesmo tipo que **x**. O resultado é igual a $\ln(x)$. Se o argumento é complexo o resultado é o valor principal da parte imaginária w na faixa $-\pi < w \leqslant \pi$. A parte imaginária do resultado é π se a parte real do argumento é menor que zero e a parte imaginária do argumento é zero
Exemplos:	real:: x, L x=8.0 L=log(x) !2.079442 x=25.0 L=log(x) !3.218876

61	**log10(x)**
Descrição:	Retorna o logaritmo decimal do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser real. O valor de **x** tem que ser maior que zero
Resultado:	Resultado Mesmo tipo de **x**. O resultado é $\log_{10}(x)$
Exemplos:	real:: x, L x=8.0 L=log(x) !0.903089 x=25.0 L=log(x) !1.397940

62	**logical(L [,kind])**
Descrição:	Converte o valor lógico do argumento em um valor lógico com tipo definido pelo parâmetro de tipo (kind)
Classe:	Função elementar
Argumentos:	**L** tem que ser do tipo lógico **kind** (opcional) tem que ser uma expressão escalar inteira de inicialização
Resultado:	Tipo lógico. Se **kind** está presente ele será usado para definir o tipo, caso contrário o tipo default lógico será usado
Exemplos:	logical::LT, LF, R LT=.true. LF=.false. R=logical(LT .and. LF) !R=F R=logical(LT .or. .not.LF) !R=T

63	**matmul(matrix_a, matrix_b)**
Descrição:	Realiza multiplicação de matrizes numéricas e lógicas
Classe:	Função transformacional
Argumentos:	**matriz_a** tem que ser do uma matriz de dimensão 1 ou 2. Tem que ser numérico (inteiro, real ou complexo) ou do tipo lógico **matriz_b** tem que ser do mesmo tipo da **matriz_a** se ela for numérica, ou do tipo lógico se a **matriz_a** for lógica. Ao menos um dos argumentos tem que ser de dimensão 2. O tamanho da dimensão 1 da **matriz_b** tem que ser igual ao tamanho da última dimensão da **matriz_a**
Resultado:	Se os argumentos são numéricos, o tipo e parâmetro de tipo (kind) dos resultados são os mesmos da expressão **matriz_a * matriz_b**. Se os argumentos são lógicos o resultado é lógico com o mesmo subtipo da expressão **matriz_a .and. matriz_b**. A forma dos resultados dependem da forma dos argumentos. O resultado vale: Se a **matriz_a** tem a forma (n, m) e a **matriz_b** tem a forma (m, k) o resultado é uma matriz de rank 2 e shape (n, k). O elemento (i, j) do resultado é obtido por: **dot_procuct(matriz_a(i, :), matriz_b(:, j)**. Se a **matriz_a** tem forma (m) e a matriz_b é do tipo (m, k) o resultado é de rank 1 e shape (k); O elemento j do resultado é: **dot_product(matriz_a(:), matriz_b(: , j))**. Se a matriz_a tem rank (n, m) e matriz_b tem shape (m), o resultado tem rank 1 e shape (n); o elemento i do resultado é: **dot_product(matriz_a(i, :), matriz_b(:))**

Exemplos:	Seja $A = \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$ $B = \begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix}$ e os vetores: X (1, 2) e Y (1, 2, 3)
	`integer,dimension(2,3)::A`
	`integer,dimension(3,2)::B`
	`integer,dimension(2,2)::mat1`
	`integer,dimension(2)::x,mat3`
	`integer,dimension(3)::y,mat2`
	`A=reshape((/2,3,3,4,4,5/),(/2,3/))`
	`B=reshape((/2,3,4,3,4,5/),(/3,2/))`
	`x=(/1,2/)`
	`y=(/1,2,3/)`
	`mat1=matmul(A,B)`
	`mat2=matmul(x,A)`
	`mat3=matmul(A,y)`
	
	`matmul(A,B) produz o resultado | 29 38 |`
	`| 38 50 |`
	`matmul(x,A) produz o resultado (8,11,14)`
	`matmul(A,y) produz o resultado (20,26)`

64	**max(a1, a2 [,a3,a4,.....])**
Descrição:	Retorna o valor máximo dos argumentos
Classe:	Função elementar
Argumentos:	**a1** (obrigatório) **a2** (obrigatório) **a3,a4....** (opcionais) têm todos que ter o mesmo tipo (inteiro ou real) e mesmo parâmetro de tipo (kind)
Resultado:	O resultado é o argumento de maior valor

Exemplos:	`integer:i1,i2,i3,ir1`
	`real::a1,a2,a3,r1`
	`  i1=4`
	`  i2=7`
	`  ir1=max(i1,i2)  !ir1=7`
	`  i1=14`
	
	`i2=32`
	`i3=-50`
	`ir1=max(i1,i2,i3)  !ir1=32`
	`a1=2.0`
	`a2=-8.0`
	`a3=6.0`
	`r1=max(a1,a2,a3)  !r1=6.0`

65	**maxexponent(x)**
Descrição:	Retorna o maior expoente do modelo numérico representado o mesmo tipo e subtipo que o argumento
Classe:	Função inquisitória
Argumentos:	**x** tem que ser do tipo real. Tem que ser escalar ou matriz
Resultado:	Escalar tipo inteiro. O resultado é e_{max} que pode ser: real precisão simples e_{max} = 128 precisão dupla e_{max} = 1024 maior ainda e_{max} = 16384
Exemplos:	integer:: me real:: x me=maxexponent(x) !me=128 precisão simples integer,parameter::dp=selected_real_kind(14) integer,parameter::dpx=selected_real_kind(16) integer::me1,me2 real(kind=dp):: x real(kind=dpx):: xx me1=maxexponent(x) !me1=1024 dupla precisão me2=maxexponent(xx) !me2=16384 precisão estendida

66	**maxloc(array, dim [,mask])** ou **maxloc(array [,mask])**
Descrição:	Retorna a localização do primeiro elemento da matriz **array** que tem o maior valor sob controle da matriz **mask**
Classe:	Função transformacional
Argumentos:	**array** tem que ser uma matriz do tipo inteiro ou real **dim** tem ????**?????? completar ???????????**?????? **mask** tem que ser uma matriz lógica conforme com **array**
Resultado:	Matriz do tipo inteiro. O resultado segue as regras: A matriz resultado tem dimensão (rank) 1 e tamanho igual ao tamanho de **array** maxloc(array), os elementos da matriz resultado são os subscrito das localizações do elemento que tem valores máximos em **array** maxloc(array, mask=mask), os elementos da matriz resultado são os subscrito das localizações do elementos com maior valor, correspondente à condição **mask** Se mais de um elemento tem valor máximo, o elemento cujo subscrito é retornado é o primeiro elemento da matriz resultado. Se **array** tem tamanho zero, o valor do resultado é indefinido
Exemplos:	Seja $C = \begin{bmatrix} 4 & 0 & -3 & 2 \\ 3 & 1 & -2 & 6 \\ -1 & -4 & 5 & -5 \end{bmatrix}$ integer,dimension(4)::A integer,dimension(1)::B integer,dimension(3,4)::C integer,dimension(2)::D A=(/3,7,4,7/) B=maxloc(A) !B=2 matriz A tem dimensão=1 ! C=reshape((/4,3,-1,0,1,-4,-3,-2,5,2,6,-5/),(/3,4/)) ! D=maxloc(C) !D=(2,4) matriz C tem dimensão=2 D=maxloc(C,mask=C<5) !D=(1,1) B=maxloc(A) vale 2 porque esta é a posição do primeiro máximo do vetor A D=maxloc(C) vale (2,4) porque o valor máximo está em C(2,4)=6 D=maxloc(C,mask=C<5) vale (1,1) porque o valor máximo inferior a 5 na matriz C está em C(1,1)=4

67	**maxval(array , dim [,mask])** ou **maxval(array [,mask])**
Descrição:	Retorna o valor máximo de todos os elementos de uma matriz, de um conjunto de elementos de uma matriz, ou os elementos de uma dimensão específica de uma matriz
Classe:	Função transformacional
Argumentos:	**array** tem que ser uma matriz do tipo inteiro ou real **dim** tem que ser uma expressão inteira escalar na faixa 1 a n, onde n é a dimensão (rank) da matriz **mask** (opcional) tem que ser uma matriz lógica conformável com **array**
Resultado:	Matriz ou escalar com os mesmos tipo de dados de **array**. Resultado é um escalar se **dim** for omitido, ou se **array** tem dimensão (rank) igual a 1. As seguintes regras são aplicadas se **dim** é omitido: **maxval(array),** o resultado é o valor máximo dos elementos de **array** **maxval(array, mask=mask),**o resultado é igual ao valor máximo dos elementos da matriz correspondendo as condições especificadas por **mask** As seguintes regras são aplicadas quando **dim** é usado a matriz resultado tem uma dimensão (rank) menor em 1 que a dimensão de **array** e forma (shape) de **array** se **array** é um vetor (rank=1), **maxval(array, dim[,mask])** é igual a **maxval(array[,mask=mask])** Se o tamanho de **array** é zero, ou se **mask** está presente e não tem elementos verdadeiro, o resultado é o maior número negativo que o compilador por representar
Exemplos:	Seja $A = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix}$

```
integer,dimension(3)::VET
integer::B
  VET=(/2,3,4/)
  B=maxval(VET)    !B=4

maxval((/2,3,4/)) vale 4, porque este é o valor máximo no vetor VET

integer,dimension(2,3)::A
integer::B
  A=reshape((/2,5,3,6,4,7/),(/2,3/))
  B=maxval(A)    !B=7

maxval(A) vale 7 porque este é o maior valor na matriz A

integer,dimension(2,3)::A
integer,dimension(3)::B
integer,dimension(2)::C
  A=reshape((/2,5,3,6,4,7/),(/2,3/))

  B=maxval(A,dim=1)    !B=(5,6,7)
  C=maxval(A,dim=2)    !C=(4,7)

maxval(A,dim=1) vale (5,6,7) porque
          5 é o valor máximo da coluna 1
          6 é o máximo da coluna 2
          7 é o máximo da coluna 3
maxval(A,dim=2) vale (4,7)
          4 é o valor máximo da linha 1
          7 é o valor máximo da linha 2

 Seja  B= | -2  3 -4 |
          | -5 -6  7 |

integer,dimension(2,3)::B
integer::C
  B=reshape((/-2,-5,3,-6,-4,7/),(/2,3/))
  C=maxval(B,mask=B<=0)    !C=-2
maxval(B,mask=B<=0) vale -2 porque este é o maior valor menor que zero na matriz
```

68	**merge(Tsource, Fsource, mask)**
Descrição:	Seleciona entre dois valores ou entre os elementos correspondentes de duas matrizes, de acordo com as condições especificadas por **mask**
Classe:	Função elementar
Argumentos:	**Tsource** tem que ser escalar ou uma matriz (de qualquer tipo de dados) **Fsource** tem que ser escalar ou uma matriz do mesmo tipo de **Tsource** **mask** tem que ser uma matriz lógica
Resultado:	Mesmo tipo de **Tsource.** O valor **mask** determina se o resultado é obtido de **Tsource (mask = .TRUE.)** ou de **Fsource (mask = .FALSE.)**
Exemplos:	$A=\begin{bmatrix}1 & 3 & 5\\ 2 & 4 & 6\end{bmatrix}, B=\begin{bmatrix}8 & 9 & 0\\ 1 & 2 & 3\end{bmatrix}, C=\begin{bmatrix}F & T & T\\ T & T & F\end{bmatrix}$

```
integer,dimension(2,3)::A, B, D
logical,dimension(2,3)::C

A=reshape((/1,2,3,4,5,6/),(/2,3/))
B=reshape((/8,1,9,2,0,3/),(/2,3/))
C=reshape((/.false.,.true., .true. ,.true., .true. ,.false./),(/2,3/))
!
  D=merge(A,B,C)     ! gera D= | 8 3 5 |
                              | 2 4 3 |
```

69	**min(a1, a2 [,a3,a4,...])**
Descrição:	Retorna o valor mínimo do argumento
Classe:	Função elementar
Argumentos:	**a1** (obrigatório) **a2** (obrigatório) **a3,a4....** (opcionais) têm todos que ter o mesmo tipo (inteiro ou real) e mesmo parâmetro de tipo (kind)
Resultado:	O resultado é o argumento de menor valor
Exemplos:	

```
 integer::i1,i2,i3,i4,R
  i1=2
  i2=-8
  i3=6
  i4=-5
  R=min(i1,i2,i3,i4)    !R=-8
```

70	**minexponent(x)**
Descrição:	Retorna o expoente mínimo no modelo numérico de mesmo tipo e mesmo parâmetro de tipo (kind) do argumento
Classe:	Função inquisitória
Argumentos:	**x** tem que ser do tipo real. Pode se escalar ou matriz
Resultado:	Escalar do tipo inteiro. O resultado vale e_{min} onde e_{min} que pode ser:

	e_{min}
real precisão simples	-125
precisão dupla	-1021
maior ainda	-16381

Exemplos:

```
 integer:: me
real:: x
  me=minexponent(x)     !me=-125 precisão simples

integer,parameter::dp=selected_real_kind(14)
integer,parameter::dpx=selected_real_kind(16)
integer::me1,me2
real(kind=dp):: x
real(kind=dpx):: xx
  me1=minexponent(x)      !me1=-1021   dupla precisão
  me2=minexponent(xx)     !me2=-16381  precisão estendida
```

71	**minloc(array, dim, [mask])** ou **minloc(array [,mask])**
Descrição:	Retorna a localização do elemento de menor valor de uma matriz, ou de um conjunto de elementos de uma matriz
Classe:	Função transformacional
Argumentos:	**array** tem que ser uma matriz do tipo inteiro ou real **dim** ??**????????? completar ??????????????????????????????**??? **mask** (opcional) tem que ser uma matriz lógica conforme com **array**
Resultado:	Matriz do tipo inteiro. O resultado segue as regras: A matriz resultado tem dimensão (rank) 1 e tamanho igual ao tamanho de **array** minloc(array), os elementos da matriz resultado são os subscrito das localizações do elemento que tem valores mínimos em **array** minloc(array, mask=mask), os elementos da matriz resultado são os subscrito das localizações do elementos com menor valor, correspondente à condição **mask** Se mais de um elemento tem valor mínimo, o elemento cujo subscrito é retornado é o primeiro elemento da matriz resultado. Se **array** tem tamanho zero, o valor do resultado é indefinido
Exemplos:	$C = \begin{bmatrix} 4 & 0 & -3 & 2 \\ 3 & 1 & -2 & 6 \\ -1 & -4 & 5 & -5 \end{bmatrix}$

```
integer,dimension(4)::A
integer,dimension(1)::B
!
integer,dimension(3,4)::C
integer,dimension(2)::D
!
integer,dimension(4)::E
integer,dimension(3)::F
!
  A=(/3, 1, 1, 7/)
  B=minloc(A)       ! B=2

B=minloc(A) vale 2 porque esta é a posição do primeiro mínimo do vetor A

  C=reshape((/4,3,-1,0,1,-4,-3,-2,5,2,6,-5/),(/3,4/))
!
  D=minloc(C)              !D=(3,4)  matriz C tem dimensão=2
  D=minloc(C,mask=C>-5)    !D=(3,2)
  E=minloc(C,dim=1)        !D=(3,3,1,3)
  F=minloc(C,dim=2)        !D=(3,3,4)

minloc(C,mask=C>-5) vale (3,2)
porque estes são os subscritos do valor mínimo (-5) que aparace na matriz C

minloc(C,dim=1) vale (3,3,1,3)
    3 é o subscrito do valor mínimo (-1) na coluna 1
    3 é o subscrito do valor mínimo (-4) na coluna 2
    1 é o subscrito do valor mínimo (-3) na coluna 3
    3 é o subscrito do valor mínimo (-5) na coluna 4

minloc(A,dim=2) vale (3,3,4)
    3 é o subscrito do valor mínimo (-3) na linha 1
    3 é o subscrito do valor mínimo (-2) na linha 2
    4 é o subscrito do valor mínimo (-5) na linha 3
```

71	**minval(array, dim, [mask])** ou **minval(array [,mask])**
Descrição:	Retorna o valor mínimo de todos os elementos de uma matriz, de um conjunto de elementos da matriz ou dos elementos de uma dimensão específica da matriz
Classe:	Função transformacional
Argumentos:	**array** tem que ser uma matriz do tipo inteiro ou real **dim** ??**?????? completar ??????????????????????????????????**??? **mask** (opcional) tem que ser uma matriz lógica conforme com **array**
Resultado:	Matriz ou escalar com os mesmos tipo de dados de **array**. Resultado é um escalar se **dim** for omitido, ou se **array** tem dimensão igual a 1. As seguintes regras são aplicadas se **dim** é omitido: ▪ minval(array), o resultado é o valor mínimo dos elementos de **array** ▪ minval(array, mask=mask), o resultado é igual ao valor mínimo dos elementos da matriz correspondendo as condições especificadas por **mask** As seguintes regras são aplicadas quando **dim** é usado ▪ a matriz resultado tem uma dimensão (rank) menor em 1 que a dimensão de **array** e forma (shape) de **array** ▪ se array é um vetor (rank=1), minval(array, dim[,mask]) é igual a minval(array[,mask=mask]) Se o tamanho de **array** é zero, ou se não há elementos true em mask, o resultado (se **dim** é omitido) ou cada elemento na matriz resultado (se **dim** é especificada) tem um valor igual ao maior número positivo que o compilador por representar

Exemplos:	$A = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix}$
	
	`integer,dimension(3)::D`
	`integer::E`
	`  D=(/2,3,4/)`
	`  E=minval(A)  !E=2`
	
	`  minval((/2,3,4/)) vale 2 porque este é o valor mínimo no vetor`
	
	`integer,dimension(2,3)::A`
	`integer,dimension(3)::B`
	`integer,dimension(2)::C`
	`  A=reshape((/2,5,3,6,4,7/),(/2,3/))`
	`!`
	`  B=minval(A,dim=1)  !B=(2,3,4)`
	`  C=minval(A,dim=2)  !B=(2,5)`
	
	`minval(A,dim=1) vale (2,3,4)`
	`  2 é o valor mínimo da coluna 1`
	`  3 é o mínimo da coluna 2`
	`  4 é o mínimo da coluna 3`
	`minval(A,dim=2) vale (2,5)`
	`  2 é o valor mínimo da linha 1`
	`  5 é o valor mínimo da linha 2`

73	**mod(a, p)**
Descrição:	Retorna o resto de uma divisão com os argumentos. Tem o sinal do primeiro argumento
Classe:	Função elementar
Argumentos:	**a** tem que ser do tipo inteiro ou real **p** tem que ter o mesmo tipo e mesmo parâmetro de tipo (kind) que a
Resultado:	Mesmo tipo de **a**. Se **p** não é zero, o valor do resultado é: $a - \text{int}\left(\frac{a}{p}\right) * p$ Se **p** é igual a zero, o resultado é indefinido
Exemplos:	integer::A=7, B=3, C, D real::X=9.0, Y=-6.0, Z ! C=mod(A, B) !C = 1 Z=mod(X, Y) !Z = 3.0 ! C=mod(-9, 6) !C= - 3 D=modulo(5, 3) !D= 2 D=modulo(-5, 3) !D= -2 D=modulo(5, -3) !D= 2 D=modulo(-5, -3) !D= -2

74	**modulo(a, p)**
Descrição:	Função módulo
Classe:	Função elementar
Argumentos:	**a** tem que ser do tipo inteiro ou real **p** tem que ter o mesmo tipo e mesmo parâmetro de tipo (kind) que a
Resultado:	Mesmo tipo de **a**. O resultado depende de **a**, como especificado: se **a** é inteiro e **p** é diferente de zero, o resultado é obtido por: $a - floor\left(\frac{real(a)}{real(p)}\right) * p$ se **a** é real e **p** é diferente de zero, o resultado é obtido por: $a - floor\left(\frac{a}{p}\right) * p$ Se **p** é igual a zero (independente do tipo de **a**) o resultado é indefinido **ATENÇÃO**: as funções mod e modulo retornam valores absolutos iguais em todos os casos exceto quando um dos argumentos é negativo. Isto é devido ao cálculo ser efetuado com a função int (mod) e a função floor (modulo)
Exemplos:	integer::A=7, B=3, C, D real::X=9.0, Y=-6.0, Z ! C=modulo(A,B) !C=1 Z=modulo(X,Y) !Z=-3.0 ! C=modulo(-9, 6) !C=3 D=modulo(5, 3) !D=2 5 são 2 unidades maior 3 D=modulo(-5, 3) !D=1 -5 é 1 unidade maior que -6 D=modulo(5, -3) !D=-1 5 é 1 unidade menor que 6 D=modulo(-5, -3) !D=-2 5 são 2 unidades menor que -3

75	**mvbits(From, Frompos, Len, To, Topos)**
Descrição:	Copia uma sequência de bits (um campo de bits) de uma localização para outra
Classe:	Subprograma não elementar
Argumentos:	**from** pode ser inteiro de qualquer tipo. Representa a localização de onde um campo de bit será transferida **frompos** pode ser inteiro de qualquer tipo. Não pode ser negativo. Identifica a posição do primeiro bit do campo transferido de **from**. frompos+len tem que ser menor ou igual a bit_size(from) **len** pode ser inteiro de qualquer tipo. Não pode ser negativo. Identifica o comprimento do campo transferido de **from**. **to** pode ser inteiro de qualquer tipo, mas tem que ter o mesmo parâmetro de tipo (kind) de **from**. Representa a localização para qual o campo de bits é transferido preenchido com uma cópia da sequência de bits de comprimento **len** iniciando na posição **frompos** de **from** e posição topos de to. Nenhum outro bit de to é alterado. Ao retornar, len bits de to (iniciando em topos) são iguais aos valores que **len** bits de **from** (iniciando em **frompos**). **topos** pode ser inteiro de qualquer tipo. Não pode ser negativo. Identifica a posição inicial (dentro de to) para os bits que estão sendo transferidos. topos+len tem que ser menor ou igual a bit_size(to)
Resultado:	

Exemplos:	`integer::A, B, C, to, E`
	`  A=7`
	`  B=2`
	`  C=2`
	`  to=6`
	`  E=0`
	`  call mvbits(A,B,C,to,E)  !to = 5`
	
	A varável to cotém inicialmente o inteiro 6, seu valor após a chamada da sub-rotina mvbits com os argumentos mostrados é 5
	
	`  call mvbits(7,2,2,to,0) vale 5`
	
	to= 00000000000000000000000000000110 = número 6
	to= 00000000000000000000000000000101 = número 5

76	**nearrest(x, s)**
Descrição:	Retorna o menor número diferente (representável pelo processador) numa dada direção
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real **s** tem que ser do tipo real e diferente de zero
Resultado:	Mesmo tipo de **x**. O resultado é igual ao número distinto mais próximo de **x**, possível de ser representado no modelo numérico usado, na direção do infinito com mesmo sinal de **s**
Exemplos:	real::X=3.0, Y=2.0, Z Z=nearest(X,Y) !Z= 3.0000002 Z=nearest(3.0,2.0) !Z= 3.0000002 Z=nearest(2.0,3.0) !Z= 2.0000002 Z=nearest(-2.0,3.0) !Z=-1.99999999 Z=nearest(-2.0,-3.0) !Z=-2.0000002 Z=nearest(2.0,-3.0) !Z= 1.9999999

77	**nint(A [,kind])**
Descrição:	Retorna o inteiro mais próximo do argumento
Classe:	Função elementar
Argumentos:	**a** tem que ser do tipo real **kind** (opcional) tem que ser uma expressão escalar inteira de inicialização
Resultado:	Tipo inteiro. Se parâmetro de tipo estiver presente o tipo é especificado por ele. Se a é maior que zero, nint(a) é calculado por nint(a+0.5). se a é menor ou igual a zero nint(a) é calculado por nint(a-0.5)
Exemplos:	real::x, y integer::a, b x= 3.879 y=-2.789 a=nint(x) !a=4 b=nint(y) !b=-3

78	**not(I)**
Descrição:	Retorna o complemento lógico do argumento
Classe:	Função elementar
Argumentos:	**I** tem que ser do tipo inteiro
Resultado:	Mesmo tipo de **I**. O resultado é calculado pelo complemento do número, bit a bit, de acordo com a tabela verdade abaixo. I not(I) 1 0 0 1
Exemplos:	integer::I I=170 ! 00000000000000000000010101010 = +170 J=not(I) !J=-171 11111111111111111111101010101 = -170 I=19410 ! 00000000000000001001011110010 = +19410 not(I) ! 11111111111111110110100001101 = -19411

79	**null(mold)**
Descrição:	Retorna um ponteiro desassociado
Classe:	Função transformacional
Argumentos:	**mold** tem que ser um ponteiro e pode ser de qualquer tipo. ??**?? completar ?????????**??
Resultado:	Mesmo tipo de **mold** se **mold** estiver presente ou definido pelo contexto. ??**???? completar ??????????????????**??
Exemplos:	integer::I??**?? completar ??????????????????**??

80	**pack(array, mask [,vector])**
Descrição:	Pega elementos de uma matriz e os coloca dentro de um vetor (matriz de rank 1) sob controle de **mask**
Classe:	Função transformacional
Argumentos:	**array** tem que ser uma matriz de qualquer tipo **mask** tem que uma matriz do tipo lógico e conforme. Determina quais elementos são pegos do **array** **vector** (opcional) tem que ser um vetor (rank 1), com o mesmo tipo de **array**. O tamanho tem que ser de pelo menos t, onde t é o número de elementos verdadeiros em **mask**. Se **mask** é um escalar com o valor true, vector tem que ter tantos elementos quantos existem em **array** Os elementos de vector são usados para preencher a matriz resultado se não existir nenhum elemento selecionado por **mask**
Resultado:	O resultado é uma matriz de rank 1 com o mesmo tipo de **array**. Se vector está presente o size do resultado é o mesmo size de vector. Caso contrário o size é definido pelo número de elementos verdadeiros em mask ou o número de elementos da matriz (se mask for um escalar com valor true. O processamento dos elementos de **array** é feito ordenadamente do 1 para o último. O elemento i é obtido da matriz **array** que corresponde ao i-éssimo elemento verdadeiro de **mask**. Se vector está presente e tem mais elementos que valores verdadeiros em **mask** qualquer elemento do resultado que estiver vazio, é preenchido pelo valor correspondente de vector

Exemplos:	$A = \begin{bmatrix} 0 & 8 & 0 \\ 0 & 0 & 0 \\ 7 & 0 & 0 \end{bmatrix}$
	`integer,dimension(3,3)::A`
	`integer,dimension(2)::AP`
	`integer,dimension(6)::V, AP1`
	`  A=reshape((/0,0,7,8,0,0,0,0,0/),(/3,3/))`
	`  V=(/1,2,3,4,5,6/)`
	`  AP=pack(A,mask= A/=0 )  !AP=(7,8)`
	`  AP1=pack(A,mask=A/=0,vector=V)  !AP1=(7,8,3,4,5,6)`

81	**precision(x)**
Descrição:	Retorna a precisão decimal do modelo usado para representar números reais com a mesmo parâmetro de tipo (kind) do argumento
Classe:	Função inquisitória
Argumentos:	**x** tem que ser do tipo real ou complexo. Pode ser um escalar ou uma matriz
Resultado:	Escalar do tipo inteiro. O resultado tem o valor **int((digits(x)-1)*log10(radix(x)))**. Se radix(x) é uma potência de 10 , 1 é adicionado ao resultado
Exemplos:	integer, parameter:: dp=selected_real_kind(14) real::x real(kind=dp)::X_dp integer::P1, P2 p1=precision(x) !p1=6 p2=precision(X_dp) !p2=15 O valor 6 é obtido de int(int(24-1)*log10(2.0))=int(6.923690)= 6 O valor 15 é obtido de int(int(53-1)*log10(2.0))=int(15.65356)= 15

82	**present(A)**
Descrição:	Retorna a informação sobre a presença ou não de um argumento formal ou mudo (informa se um argumento real foi usado na chamada do subprograma)
Classe:	Função inquisitória
Argumentos:	**A** tem que ser um argumento opcional do subprograma
Resultado:	Resultado é escalar do tipo lógico. O resultado é .TRUE. se **A** estiver presente, caso contrário o resultado é .FALSE.

Exemplos:	Programa principal pode conter as chamadas:
	.....
	call verifica(2,25,k) !passa 3 argumentos reais
	.....
	call verifica(y=25,z=k) !passa só 2 argumentos reais
	call verifica(z=k, y=25) !também passa somente 2 argumentos reais
	!
	subroutine verifica(x, y, z)
	integer, intent(in), optional:: x
	integer, intent(in):: y
	integer, intent(out):: z
	
	if(present(x)) then
	z = (x**2+y**2)+10
	else
	z = (2*y)+10
	endif
	end subroutine verifica

83	**poduct(array, dim [,mask])** ou **product(array [,mask])**
Descrição:	Retorna o produto de todos os elementos de uma matriz ou de uma dimensão especificada da matriz
Classe:	Função transformacional
Argumentos:	**array** tem que ser uma matriz do tipo inteiro ou real **dim** tem que ser um escalar inteiro com valor entre 1 e n, onde n é ao rank de array **mask** (opcional) tem que ser do tipo lógico e conforme com **array**
Resultado:	Uma matriz ou um escalar do mesmo tipo que **array**. O resultado é um escalar se **dim** é omitido ou **array** é um vetor (rank=1). as seguintes regras se aplicam: Se **dim** é omitido: ▪ **product(array)** produz um resultado que é o produto de todos os elementos de **array**. Se **array** tem size zero, o resultado é 1 ▪ **product(array, mask=mask)** produz um resultado que é o produto de todos os elementos de **array** que tem um correspondente valor true em **mask**. Se não houver elemento verdadeiro, o resultado é 1. Se **dim** é utilizado: ▪ se **array** é um vetor (rank=1) o resultado é o mesmo que **product(array[,mask])** ▪ a matriz resultado tem uma dimensão (rank) menor em 1 que a dimensão de array e forma (shape) definida pela forma da **array**
Exemplos:	$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 3 & 5 \end{bmatrix}$

```
integer,dimension(3)::p
integer::R
  p=(/2,3,4/)
  R=product(p)       !R=24   (2*3*4=24)

integer,dimension(2,3)::A
integer,dimension(3)::B
integer,dimension(2)::C
  A=reshape((/1,2,4,3,7,5/),(/2,3/))
  B=product(A,dim=1)                    !B=(2,12,35)
  C=product(A,dim=2)                    !C=(28,30)

 product(A,dim=1)  vale (2,12,35)
                            2  é obtido de 1*2
                            12 é obtido de 4*3
                            35 é obtido de 7*5
 product(A,dim=2)  vale (28,30)     Obtido de 1*4*7=28 e 2*3*5=30
```

84	**radix(x)**
Descrição:	Retorna a base do modelo representativo dos números de mesmo tipo (type) e mesmo parâmetro de tipo (kind) que o argumento
Classe:	Função inquisitória
Argumentos:	**x** tem que ser do tipo inteiro ou real. Pode ser escalar ou matriz
Resultado:	Escalar do tipo inteiro. Para um argumento inteiro, o resultado tem o valor **r**, onde r é o **radix** (base) do sistema numérico usado. Para um argumento real o resultado tem o valor b (base) do modelo numérico usado no computador
Exemplos:	real::x integer::I, R1, R2 R1=radix(x) !R1 = 2 R2=radix(I) !R2 = 2

85	**random_number(harvest)**
Descrição:	Retorna um número pseudo-aleatório ou uma matriz de números pseudo-aleatórios
Classe:	Subprograma não elementar
Argumentos:	**harvest** tem que ser do tipo real. Ele é um argumento do tipo intent(out) e pode ser um escalar ou uma matriz. Conterá números pseudo-aleatórios obtidos de uma distribuição uniforme na faixa $0 \leqslant x < 1$
Resultado:	
Exemplos:	real:: x real, dimension(3,3):: y ! call random_number(harvest=x) ! x conterá o número aleatório call random_number(y) ! y matriz de números aleatórios

86	**random_seed([size, put, get])**
Descrição:	Reinicia ou inicia a semente do gerador de números aleatórios
Classe:	Subprograma não elementar
Argumentos:	**harvest** tem que ser do tipo real. Ele é um argumento do tipo intent(out) e pode ser um escalar ou uma matriz. Conterá números pseudo-aleatórios obtidos de uma distribuição uniforme na faixa $0 \leqslant x < 1$
Resultado:	
Exemplos:	call random_seed ! processador inicia a semente aleatoriamente ! usando a data e o tempo do relógio do computador call random_seed(SIZE=M) ! M contém o número de inteiros utilizado call random_seed(PUT=SEED(1:M)) ! utiliza semente do usuário call random_seed(GET=OLD(1:M)) ! lê a semente atual

87	**range(x)**
Descrição:	Retorna a faixa decimal usada no expoente do modelo numérico de mesmo parâmetro de tipo (kind) do argumento
Classe:	Função inquisitória
Argumentos:	**x** tem que ser do tipo inteiro, real ou complexo. Pode ser um escalar ou uma matriz
Resultado:	Resultado é um escalar do tipo inteiro. Para um argumento inteiro, o resultado vale **int(log10(huge(x))).** Para um argumento real ou complexo o resultado é avaliado por: **int(min(log10(huge(x), -log10(tiny(x))))**
Exemplos:	real::x integer::R R=range(x) !R = 37 range(x) vale 37 é obtido por: huge(x) = (1-2-24)x 2128 tiny(x) = 2-126 int(min(log10(huge(x),-log10(tiny(x)))) = 37 ! integer,parameter::dp=selected_real_kind(14) integer,parameter::dpx=selected_real_kind(16) real(kind=dp)::x real(kind=dpx)::y integer::R1,R2 R1=range(x) !R1 = 307 R2=range(y) !R2 = 4931

88	**real(A [,kind])**
Descrição:	Converte um valor para o tipo real
Classe:	Função elementar
Argumentos:	**A** tem que ser do tipo inteiro, real ou complexo **kind** (opcional) tem que ser uma expressão escalar inteira de inicialização
Resultado:	Tipo real. Se kind estiver presente, o parâmetro de tipo é definido por ele. Se **A** é inteiro ou real, o resultado é iguala uma aproximação de **A**. Se **A** é complexo, o resultado é igual a uma aproximação da parte real de **A**

Exemplos:	`integer::ix`
	`complex::y`
	`real::z`
	`  ix=-4`
	`  y=(7.0,5.3)`
	`  z=real(ix)  !z =-4.0`
	`  z=real(y)  !z = 7.0`

89	**repeat(string, ncopies)**
Descrição:	Concatena várias cópias de um string
Classe:	Função transformacional
Argumentos:	**string** tem que ser um escalar tipo caractere **ncopies** tem que ser escalar inteiro. Não pode ser negativo
Resultado:	Escalar do tipo caractere com um comprimento de **ncopies *len(string)**. O parâmetro de tipo (kind) é o mesmo de **string**. O resultado é obtido pela concatenação de **ncopies** de **string**

Exemplos:	`character(len=3)::texto1="ola"`
	`character(len=12)::Rep`
	`  rep=repeat(texto1,4)  !rep=olaolaolaola`
	`  print*,repeat("W",3)  !WWW`
	`  print*,repeat("W",0)  !não imprime nada pois é um string de comprimento zero`

90	**reshape(source, shape [,pad, order])**
Descrição:	Constrói uma matriz com uma forma (shape) diferente da forma do matriz usada no argumento
Classe:	Função transformacional
Argumentos:	**source** tem que ser uma matriz (com dados de qualquer tipo). Ela fornece os elementos para a matriz resultado. Seu tamanho (size) tem que ser maior ou igual a product(shape) se **pad** é omitido ou tem tamanho (size) zero **shape** tem que ser uma matriz do tipo inteiro com até 7 elementos, com dimensão (rank) 1 e tamanho (size) constante. Ela define a forma (shape) da matriz resultado. Seu tamanho (size) tem que ser positivo e seus elementos não podem ter números negativos **pad** (opcional) tem que ser uma matriz com o mesmo tipo e mesmo parâmetro de tipo (kind) que **source**. É usado para preencher valores extras na matriz resultado se ela for maior que **source** **order** (opcional) tem que ser uma matriz do tipo inteira com a mesma forma (shape) que a matriz **shape**. Seus elementos tem que ser uma permutação de (1, 2, 3, ...,n), onde n é o tamanho (size) da matriz **shape**. Se o argumento **order** é omitida, ele é assumida ser (1, 2, ..., n)
Resultado:	O resultado é uma matriz de forma (shape) **shape** com o mesmo tipo e mesmo parâmetro de tipo (kind) que **source**. O resultado tem um tamanho (size) igual ao produto dos valores de shape. Na matriz resultado, os elementos de **source** são colocados na ordem das dimensões especificadas pelo parâmetro **order**. Se o parâmetro **order** é omitido, os elementos da matriz são colocados na ordem normal da matriz. Os elementos de **source** são seguidos (se necessário) pelos elementos da matriz pad na ordem usual. Se necessário copias adicionais de **pad** são usadas até que todos os elementos da matriz resultado tenham valores

Exemplos:	`integer,dimension(2,3)::A`
	`integer,dimension(2,4)::B`
	`A=reshape((/3,4,5,6,7,8/),(/2,3/))` $A = \begin{bmatrix} 3 & 5 & 7 \\ 4 & 6 & 8 \end{bmatrix}$
	`B=reshape((/3,4,5,6,7,8/),(/2,4/),(/1,1/),(/2,1/))` $B = \begin{bmatrix} 3 & 4 & 5 & 6 \\ 7 & 8 & 1 & 1 \end{bmatrix}$

91	**rrspacing(x)**
Descrição:	Retorna o reciproco dos espaços relativos do modelo numérico perto dos valores usados no argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real
Resultado:	Mesmo tipo de **x**. O resultado é calculado por: $\lvert x \times b^{-e} \rvert \times b^{p}$ onde x = número real b = base e = um inteiro entre e_{min} e e_{max} p = números de dígitos no principal e_{min} e_{max} precisão simples -125 128 dupla precisão -1021 1024 estendido -16381 16384
Exemplos:	real::x,y x=-3.0 y=rrspacing(x) ! $0.75*2^{24}$ = 1.2582912E+07

92	**scale(x, i)**
Descrição:	Retorna o valor usado no expoente (do modelo usado no argumento) que foi alterado por um valor especificado
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real **i** tem que ser inteiro
Resultado:	Mesmo tipo que **x**. O resultado é calculado por: $x \times b^{-I}$
Exemplos:	real::x,r x=3.0 r=scale(3.0,2) !r=12.0 r=scale(3.0,3) !r=24.0

93	**scan(string, set [,back])**
Descrição:	Procura por um string especificado num conjunto de caracteres
Classe:	Função elementar
Argumentos:	**string** tem que ser do tipo caractere **set** tem que ser tipo caractere de mesma subtipo que **string** **back** (opcional) tem que ser do tipo lógico
Resultado:	Tipo inteiro. Se **back** é omitido (ou é falso) e **string** tem pelo menos 1 caractere que existe no **set**, o resultado é a posição do caractere mais a esquerda no **string**. Se **back** é utilizado e possui o valor true, e o **string** tem, pelo menos 1 caractere especificado no **set**, o resultado é a posição mais a direita do **string**
Exemplos:	integer::N character(len=7)::texto texto="passeio" N=scan(texto,"as") !N = 2 N=scan(texto,"ai",back=.true.) !N = 6 N=scan(texto,"as",back=.true.) !N = 4

94	**selected_int_kind(r)**
Descrição:	Retorna o valor numérico que identifica o parâmetro de tipo do dado de tipo inteiro
Classe:	Função transformacional
Argumentos:	**r** tem que ser inteiro escalar
Resultado:	Resultado é um escalar do tipo inteiro. O resultado é igual ao valor do parâmetro de tipo do tipo inteiro que representa todos os valores n na faixa $-10^r < n < 10^r$. Se o parâmetro de tipo não está disponível no processador, o resultado será −1. Se mais de um valor existe para o parâmetro de tipo, o valor retornado aquele que fornece a menor faixa para o expoente decimal
Exemplos:	integer,parameter::i1b=selected_int_kind(2) integer,parameter::i2b=selected_int_kind(4) integer,parameter::i3b=selected_int_kind(9) integer,parameter::i4b=selected_int_kind(10) integer,parameter::i5b=selected_int_kind(18) integer,parameter::i6b=selected_int_kind(19) ! print*,"i1b=",i1b !i1b = 1 byte obs: o valor retornado é dependente do print*,"i2b=",i2b !i2b = 2 processador utilizado no computador print*,"i3b=",i3b !i3b = 4 print*,"i4b=",i4b !i4b = 8 print*,"i5b=",i5b !i5b = 8 print*,"i6b=",i6b !i6b = -1 não pode ser representado neste computador

95	**selected_real_kind([p, r])**
Descrição:	Retorna o valor numérico que identifica o parâmetro de tipo (kind) do real
Classe:	Função transformacional
Argumentos:	Pelo menos um argumento tem que estar presente **p** (opcional) tem que ser um escalar do tipo inteiro **r** (opcional) tem que ser um escalar do tipo inteiro
Resultado:	Resultado é um escalar do tipo inteiro default. O resultado é igual ao valor do parâmetro de tipo do dado real que possui uma precisão decimal (como a precisão retornada pela função precision) de pelo menos p dígitos e uma faixa de expoentes decimais (retornada pela função range) de pelo menos r. Se o parâmetro de tipo não está disponível no processador, o resultado será: −1 se a precisão não estiver disponível −2 se a faixa do expoente não está disponível −3 se nenhuma delas é disponível Se mais de 1 parâmetro de tipo satisfizer o critério, o valor retornado é aquele que fornece a menor precisão decimal
Exemplos:	integer,parameter::sp=selected_real_kind(6) integer,parameter::dp=selected_real_kind(14) integer,parameter::dpx=selected_real_kind(16) integer,parameter::dpn=selected_real_kind(20) ! print*,"sp=",sp !sp = 4 bytes obs: o valor retornado é dependente do print*,"dp=",dp !dp = 8 processador utilizado print*,"dpx=",dpx !dpx = 10 print*,"dpn=",dpn !dpn = -1 não pode ser representado neste computador

96	**set_exponent(x, i)**
Descrição:	Retorna o número cuja parte fracionária é a parte fracionária (significante) de x e cuja parte expoente é igual a i, no modelo numérico do Fortran para o tipo real de mesmo parâmetro de tipo (kind) que x
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real **i** tem que ser inteiro
Resultado:	Mesmo tipo que **x**. O resultado tem o valor **x*radix(x)**(i-exponent(x))**
Exemplos:	real::x,r x=3.0 r=set_exponent(x,1) !r=1.5

97	**shape(source)**
Descrição:	Retorna a forma (shape) de uma matriz ou um escalar
Classe:	Função inquisitória
Argumentos:	**source** é um escalar ou matriz (de qualquer tipo de dado). Não pode ser uma matriz de forma assumida, um ponteiro não associado ou uma matriz alocável que não esteja alocada
Resultado:	Vetor (matriz rank=1) tipo inteiro default cujo tamanho é igual a dimensão (rank) de **source**. O resultado é igual a forma (shape) de **source**
Exemplos:	shape(2) é igual ao valor obtido de uma matriz de dimensão 1 (rank=1) e tamanho zero Definida a matriz B(2:4, -3:1) então shape(B) vale (3, 5) integer,dimension(3)::V1 integer,dimension(1)::Rv1 V1=(/2,4,6/) Rv1=shape(V1) !Rv1=3 porque o vetor tem 3 elementos integer,dimension(3,5)::A integer,dimension(2)::R A=reshape((/1,2,3,1,2,3,1,2,3,1,2,3,1,2,3/),(/3,5/)) R=shape(A) shape(A) é igual (3,5) matriz A= | 1 1 1 1 1 | | 2 2 2 2 2 | | 3 3 3 3 3 |

98	**sign(a, b)**
Descrição:	Retorna o valor absoluto de **a** vezes o sinal de **b**
Classe:	Função elementar
Argumentos:	**a** tem que ser do tipo inteiro ou real **b** tem que ter o mesmo tipo e mesmo parâmetro de tipo (kind parameter) de **a**
Resultado:	O resultado é do mesmo tipo de **a**. O resultado é $\lvert a\rvert$ se $b \geqslant 0$ e $-\lvert a\rvert$ $b<0$
Exemplos:	real::x, y, Z x=4.0 y=-6.0 Z=sign(x,y) !Z=-4.0 x=-5.0 y=2.0 Z=sign(x,y) !Z=5.0

99	**sin(x)**
Descrição:	Calcula a função seno do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real ou complexo. Tem que ser fornecido em radianos Se **x** é complexo, sua parte real é tratada como sendo em radianos
Resultado:	Mesmo tipo de **x**
Exemplos:	real::theta,x theta=2.094395 !theta= 120^0 x=sin(theta) !x=0.8660255

100	**sinh(x)**
Descrição:	Calcula a função seno hiperbólica do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real
Resultado:	Mesmo tipo de **x**
Exemplos:	real::theta,x theta=2.094395 !theta= 120^0 x=sinh(theta) !x=3.998691

101	**size(array [,dim])**
Descrição:	Retorna o número total de elementos de uma matriz, ou a extensão de uma dimensão especificada da matriz
Classe:	Função inquisitória
Argumentos:	**array** tem que ser uma matriz (de qualquer tipo de dados). Não pode ser um ponteiro dissociado ou uma matriz alocável não alocada. Pode ser uma matriz de forma assumida se **dim** estiver presente com um valor menor que a dimensão (rank) de **array** **dim** (opcional) tem que ser escalar do tipo inteiro com o valor entre 1 e n, onde n é a dimensão (rank) de **array**
Resultado:	Escalar tipo inteiro. Se **dim** está presente, o resultado é a extensão da dimensão **dim** de **array**, caso contrário é o número total de elementos em **array**
Exemplos:	$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$

```
integer,dimension(4,6)::A
integer::S,SL,SC
  A=reshape((/1,1,1,1,2,2,2,2,3,3,3,3, &
              4,4,4,4,5,5,5,5,6,6,6,6/),(/4,6/))
!
  S=size(A)          !S=24     24 elementos
  SL=size(A,dim=1)   !SL=4      4 linhas
  SC=size(A,dim=2)   !SC=6      6 colunas
```

102	**spacing(x)**
Descrição:	Retorna o espaçamento absoluto do modelo numérico usado perto do valor do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real
Resultado:	Mesmo tipo de **x**. O resultado tem o valor $b^e - p$. onde b = base p = quantidade de dígitos usado no principal e = inteiro entre e_{min} e e_{max} Se o resultado está fora da faixa de representação o resultado será obtido por TINY(x) p / e_{min} / e_{max} precisão simples 24 precisão simples -125 128 precisão dupla 53 precisão dupla -1021 1024 estendida 113 estendida -16381 16384
Exemplos:	real::x, y, esp1, esp2 x=3.0 y=3000.0 esp1=spacing(x) !esp1=2^{-22}= 2.3841858E-07 esp2=spacing(y) !esp2=2^{-12}= 2.4414063E-04

103	**spread(source, dim, ncopies)**
Descrição:	Cria uma réplica de uma matriz com uma dimensão a mais, copiando os elementos existentes ao longo de uma dimensão especificada
Classe:	Função transformacional
Argumentos:	**source** tem que ser um escalar ou matriz (de qualquer tipo de dados). A dimensão (rank) tem que ser menor que 7 **dim** tem que ser escalar do tipo inteiro. Tem que ter um valor na faixa 1 a r+1 (inclusive), onde r é a dimensão (rank) de **source** **ncopies** tem que ser um escalar do tipo inteiro. Será usado para definir a extensão da dimensão extra na matriz resultado
Resultado:	Uma matriz de mesmo tipo que **source** com dimensão (rank) 1 unidade maior que **source.** Se **source** é uma matriz, cada elemento da matriz resultado na dimensão **dim** é igual ao correspondente elemento de **source** Se **source** é um escalar, o resultado é um vetor (matriz rank=1) com **ncopies** elementos, cada um com o valor de **ncopies** Se **ncopies** <= zero, o resultado é uma matriz de comprimento zero
Exemplos:	character(len=1)::tex character,dimension(4)::texto tex="b" texto=spread(tex,dim=1,ncopies=4) !texto="bbbb" integer,dimension(3):: num integer,dimension(4,3)::M43 integer,dimension(3,4)::M_34 num=(/1,2,3/) !M43= | 1 2 3 | M43=spread(num,dim=1,ncopies=4) ! | 1 2 3 | M_34=spread(num,dim=2,ncopies=4) ! | 1 2 3 | ! | 1 2 3 | spread(num,dim=2,ncopies=4) é a matriz M_34=| 1 1 1 1 | | 2 2 2 2 | | 3 3 3 3 |

104	**sqrt(x)**
Descrição:	Calcula a raiz quadrada do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real ou complexo. Se **x** é real, seu valor tem que ser maior ou igual a zero
Resultado:	Mesmo tipo que **x**. O resultado é a raiz quadrada de **x**. Se **x** é complexo o resultado é o valor principal com a parte real não negativa. quando a parte real do resultado é zero, a parte imaginária é maior ou igual a zero

Exemplos:	`real::x=r`
	`  x=25.0`
	`  r=sqrt(x)  !r=5.0`

105	**sum(array, dim [,mask])** ou **sum(array [,mask])**
Descrição:	Retorna a soma de todos os elementos de uma matriz ou os elementos de uma dimensão especificada da matriz
Classe:	Função transformacional
Argumentos:	**array** tem que ser uma matriz do tipo inteira, real ou complexa **dim** tem que ser um escalar inteiro com valor na faixa 1 a n, onde n é a dimensão (rank) de **array** **mask** (opcional) tem que ser do tipo lógico e conforme com **array**
Resultado:	O resultado é um escalar se **dim** é omitido ou **array** tem dimensão (rank) 1. As seguintes regras se aplicam: ▪ sum(array) retorna a soma de todos os elementos de **array**. Se **array** tem dimensão zero, o resultado é zero. ▪ sum(array, mask=mask) retorna a soma de todos os elementos de **array** correspondentes aos elementos true de **mask**. Se não existir elemento verdadeiro o resultado é zero. Se **dim** é utilizado: ▪ se **array** tem dimensão (rank) 1 o valor é igual ao obtido com sum(array, [mask=mask]) a matriz resultado tem uma dimensão (rank) 1 unidade menor que **array** e forma (shape) de **array**
Exemplos:	$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

```
integer,dimension(3)::V
integer::S
  V=(/2,3,4/)
  S=sum(V)         !S=9    (2+3+4=9)
  S=sum(V,dim=1)   !S=9

integer,dimension(2,3)::A
integer,dimension(3)::S1
integer,dimension(2)::S2
  A=reshape((/1,4,2,5,3,6/),(/2,3/))
  S1=sum(A,dim=1)
  S2=sum(A,dim=2)
                      sum(A,dim=1) vale (5,7,9)
                                                 5 é a soma de 1+4
                                                 7=2+5
                                                 9=3+6
sum(A,dim=2) vale (6,15)   6 =1+2+3 ; 15=4+5+6
```

106	**system_clock[count, count_rate, count_max])**
Descrição:	Retorna dado do tipo inteiro do relógio real do computador
Classe:	Subprograma não elementar
Argumentos:	**count** (opcional) tem que ser um escalar e do tipo inteiro default. Recebe o valor atual do relógio do processador. O valor é incrementado de 1 para cada **count** do relógio até o valor count_max ser alcançado e é zerado na próxima contagem (**count** está na faixa 0 a count_max) (opcional) **count_rate** (opcional) tem que ser escalar e do tipo inteiro default. Recebe o número de contagens por segundo (freqüência) do processador **count_max** (opcional) tem que ser escalar e do tipo inteiro default. Recebe o valor máximo que **count** pode conter (huge(0))
Resultado:	
Exemplos:	integer:: ic, cr, cm call system_clock(count=ic, count_rate=cr, count_max=cm) print *,"ic= ", ic, "cr= ",cr, "cm= ",cm produz uma saída do tipo: ic=963517674 cr=1000 cm=2147483647

107	**tan(x)**
Descrição:	Calcula a função tangente do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real. Tem que estar em radianos
Resultado:	Mesmo tipo de **x**
Exemplos:	real::theta,x theta=1.221730 !theta= 70^0 x=tan(theta) !x=2.747473

108	**tanh(x)**
Descrição:	Calcula a função tangente hiperbólica do argumento
Classe:	Função elementar
Argumentos:	**x** tem que ser do tipo real
Resultado:	Mesmo tipo de **x**
Exemplos:	real::theta,x theta=1.221730 !theta= 70^0 x=tanh(theta) !x=0.8401638

109	**tiny(x)**
Descrição:	Retorna o menor número do modelo numérico usado para representar um número de mesmo tipo e mesmo parâmetro de tipo (kind parameter) que o argumento
Classe:	Função inquisitória
Argumentos:	**x** tem que ser do tipo real. Pode ser escalar ou matriz
Resultado:	Escalar de mesmo tipo e mesmo parâmetro de tipo (kind parameter) de **x**. O resultado é obtido por $b^{e_{min}-1}$ onde: b = base e e_{min} = inteiro e_{min} precisão simples -125 precisão dupla -1021 estendida -16381
Exemplos:	real::X,y y=tiny(x) !$y=e^{-126}$ = 1.1754944E-38

110	**transfer(source, mold [,size])**
Descrição:	Converte um conjunto de bits de source de acordo com o tipo e parâmetro de tipo de mold
Classe:	Função transformacional
Argumentos:	**source** tem que ser do escalar ou matriz (de qualquer tipo de dado) **mold** tem que ser um escalar ou uma matriz (de qualquer tipo de dado). Provê as características do tipo (não o valor) para o resultado **size** (opcional) tem que ser escalar do tipo inteiro. Especifica o número de elementos usados no resultado
Resultado:	Tem o mesmo tipo e mesmo parâmetro de tipo (kind parameter) que **mold**. Se **mold** é um escalar e **size** é omitido, o resultado é um escalar Se **mold** é uma matriz e **size** é omitido, o resultado é um vetor (matriz de rank 1). Seu tamanho é o menor possível, adequado para conter todos os elementos de **source** Se **size** estiver presente o resultado é um vetor (matriz de rank 1) de tamanho **size** Se o tamanho físico usado no resultado for maior que **source**, o resultado conterá os bits mais a direita de **source**, ficando os bits mais a esquerda indefinidos Se o tamanho físico usado no resultado for menor que **source**, o resultado conterá os bits mais a direita de **source**
Exemplos:	real,dimension(3)::A real,dimension(2)::T A=(/2.2, 3.3, 4.4/) T=transfer(A,((0.0, 0.0))) !T=(2.2, 2.2)

111	**transpose(matrix)**
Descrição:	Transposta de uma matriz de dimensão (rank) 2
Classe:	Função transformacional
Argumentos:	**matrix** tem que ser uma matriz de dimensão (rank) 2 (de qualquer tipo de dado)
Resultado:	Matriz de dimensão 2 com o mesmo tipo e mesmo parâmetro de tipo que **matrix.** Sua forma (shape) é (n, m) onde (m, n) é a forma de **matrix.** O elemento (i, j) do resultado é o elemento matrix(j, i) onde i varia na faixa 1 a n e j varia na faixa 1 a m
Exemplos:	Seja a matriz $A = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 1 \end{bmatrix}$ integer,dimension(3,3)::A,T A=reshape((/2,5,8,3,6,9,4,7,1/),(/3,3/)) T=transpose(A) $T = \begin{bmatrix} 2 & 5 & 8 \\ 3 & 6 & 9 \\ 4 & 7 & 1 \end{bmatrix}$

112	**trim(string)**
Descrição:	Retorna o argumento com brancos a direita removidos
Classe:	Função transformacional
Argumentos:	**string** tem que ser escalar do tipo caractere
Resultado:	Caractere de mesmo tipo e subtipo que **string**. Seu comprimento é o mesmo de **string** menos os branco existentes a direita. O resultado é igual a string exceto que os branco existentes à direita são removidos. Se **string** é um único espaço em branco, o resultado terá tamanho zero

Exemplos:	o símbolo _ foi utilizado para marcar espaço em branco
	
	`character(len=12)::texto`
	`integer::N`
	` texto="baixinho  "  ! "baixinho _ _ _ _"  tamanho=12`
	`!`
	` N=trim(texto)  ! "baixinho"  tamanho=8`

113	**ubound(array [,dim])**
Descrição:	Retorna o limite superior de todas as dimensões de uma matriz ou o limite superior de uma dimensão especificada de uma matriz
Classe:	Função inquisitória
Argumentos:	**array** tem que ser uma matriz (de qualquer tipo de dado). Não pode ser uma matriz alocável que não esteja alocada ou um ponteiro não associado. Pode ser uma matriz de forma assumida se **dim** estiver presente com um valor menor que a dimensão (rank) de **array** **dim** (opcional) tem que ser um escalar do tipo inteiro com valor entre 1 e n, onde n é a dimensão (rank) de **array**
Resultado:	Tipo inteiro default. Se **dim** estiver presente o resultado é um escalar, caso contrário o resulatdo é um vetor (matriz de rank=1) com um elemento para cada dimensão de **array**. Cada elemento no vetor corresponde a uma dimensão de **array**. Se **array** é uma seção de matriz ou uma expressão matricial que não é a matriz inteira, **ubound(array, dim)** tem um valor igual ao número de elementos em uma dada dimensão. Se **array** é a matriz inteira, **ubound(array, dim)** tem um valor igual ao limite superior para o subscrito **dim** de **array** (se **dim** é diferente de zero). Se **dim** tem tamanho zero, o elemento correspondente no resultado também tem tamanho zero

Exemplos:	`real::A(1:3,5:8)`
	`real :: B(2:8,-3:20)`
	`integer:: R1`
	`integer,dimension(2)::R2,R3,R4`
	`  R2=ubound(A)  !R2=(3, 8)`
	`  R1=ubound(A,dim=2)  !R1=8`
	`  R3=ubound(B)  !R3=(8, 20)`
	`  R4=ubound(B(5:8,:))  !R4=(4, 24)`
	
	$A = \begin{vmatrix} a_{15} & a_{16} & a_{17} & a_{18} \\ a_{25} & a_{26} & a_{27} & a_{28} \\ a_{35} & a_{36} & a_{37} & a_{38} \end{vmatrix}$ $B = \begin{vmatrix} b_{-23} & b_{-22} & b_{-21} & b_{20} & \dots & b_{220} \\ b_{-33} & b_{-32} & b_{-31} & b_{30} & \dots & b_{320} \\ b_{-43} & b_{-42} & b_{-41} & b_{40} & \dots & b_{420} \\ b_{-53} & b_{-52} & b_{-51} & b_{50} & \dots & b_{520} \\ b_{-63} & b_{-62} & b_{-61} & b_{60} & \dots & b_{620} \\ b_{-73} & b_{-72} & b_{-71} & b_{70} & \dots & b_{720} \\ b_{-83} & b_{-82} & b_{-81} & b_{80} & \dots & b_{820} \end{vmatrix}$

114	**unpack(vector, mask, field)**
Descrição:	Pega elementos de um vetor e coloca-os dentro de uma matriz sob o controle de mask
Classe:	Função transformacional
Argumentos:	**vector** tem que ser um vetor (matriz rank=1) de qualquer tipo. Seu tamanho tem que ser de pelo menos t, onde t é o número de elementos verdadeiros em **mask** **mask** tem que ser uma matriz lógica. Ela determina aonde os elementos de vector serão colocados quando forem transferidos **field** tem que ser do mesmo tipo e mesmo parâmetro de tipo (kind parameter) que vector e conforme com **mask**. Os elementos em **field** são inseridos na matriz resultado quando o elemento correspondente em **mask** temo valor falso
Resultado:	Matriz de mesma forma (shape) que **mask** e mesmo tipo e parâmetro de tipo (kind parameter) que **vector**. Os elementos no resultado são preenchidos na ordem da matriz. Se o elemento i de **mask** é verdadeiro, o elemento correspondente no resultado é preenchido com o próximo elemento de **vector** caso contrário, ele é preenchido por **field** (**field** é um escalar) ou o i-éssimo elemento de **field** (se **field** for uma matriz)

Exemplos:	$A=\begin{bmatrix}0&0&1\\1&0&1\\1&0&0\end{bmatrix}$ B=(2, 3, 4, 5) $C=\begin{bmatrix}T&F&F\\F&T&F\\T&T&F\end{bmatrix}$
	`integer,dimension(4)::B`
	`integer,dimension(3,3)::A,R1,R2`
	`logical,dimension(3,3)::C`
	`  B=(/2,3,4,5/)`
	`  A=reshape((/0,1,1,0,0,0,1,1,0/),(/3,3/))`
	`  C=reshape((/.true. ,.false.,.true.,  &`
	`              .false.,.true. ,.true.,  &`
	`              .false.,.false.,.false./),(/3,3/))`
	`!`
	`  R1=unpack(B,mask=C,field=A)`
	`  R2=unpack(B,mask=C,field=9)`

115	**verify(string, set [, back])**
Descrição:	Verifica se um conjunto de caracteres contém todos os caracteres em um string pela identificação do primeiro caractere no string que não está no conjunto
Classe:	Função elementar
Argumentos:	**string** tem que ser do tipo caractere **set** tem que ser do tipo caractere da mesmo parâmetro de tipo (kind parameter) que string **back** tem que ser do tipo lógico
Resultado:	Inteiro default. Se **back** é omitido (ou está presente com o valor falso) e **string** tem pelo menos 1 caractere que não está em **set** o resultado é a posição mais a direita de **string** que não está em **set** Se **back** está presente com o valor verdadeiro (.TRUE.) e **string** tem pelo menos 1 caractere que não está em **set**, o resultado é a posição do caractere mais a direita de **string** que não está no **set** Se cada caractere de **string** existe em **set** ou se o comprimento de **string** é zero, o resultado é zero

Exemplos:	`character(len=6)::texto`
	`integer::N1,N2`
	`  texto="ahhhho"`
	`  N1=verify(texto,"ao")  !N1=2`
	`  N2=verify(texto,"ao",back=.true.)  !N2=5`