



Centro Universitário Adventista de São Paulo

Campus São Paulo

Curso de Ciência da Computação

Disciplina: Paradigmas de Linguagens de Programação

Fortran 90

Por

Weyden Daniel Castro Bravo

Professor da disciplina
Dr. Ausberto S. Castro Vera

São Paulo, 10 de abril de 2014

1	História da linguagem de programação FORTRAN	6
1.1	Características do desenvolvimento do FORTRAN.....	7
1.2	FORTRAN 0	7
1.3	FORTRAN I	7
1.4	FORTRAN II.....	9
1.5	FORTRAN III	9
1.6	FORTRAN IV	10
1.7	FORTRAN 77	10
1.8	FORTRAN 90	10
2	O Paradigma Imperativo.....	11
3	Introdução à programação em FORTRAN.....	12
3.1	Escrevendo um programa em FORTRAN.....	12
3.2	Tipos de dados.....	13
3.2.1	Inteiro	13
3.2.2	Real	13
3.2.3	Complexo	14
3.2.4	Lógico.....	14
3.2.5	String de caracteres.....	14
3.3	Identificadores	15
3.4	Declaração de variáveis	16
3.5	Atribuindo uma constante a um nome.....	18
3.6	A instrução de atribuição.....	19
4	Operadores e expressões.....	19
4.1	Aritméticos	20
4.1.1	Expressões aritméticas de modo simples	21
4.1.2	Expressões aritméticas de modo misto.....	21
4.2	Relacionais	22
4.3	Lógicos	24
4.3.1	Tabelas verdade	25
4.4	Caracteres	27
4.4.1	Operador de concatenação: //	27
4.4.2	Substrings	27
5	Entrada e saída em lista	30
5.1	READ	31
5.2	WRITE	32
6	Seleção.....	33
6.1	IF lógico.....	34
6.2	IF-THEN-END IF	35
6.3	IF-THEN-ELSE-END IF	38
6.4	Estruturas de seleção aninhadas	40
6.5	SELECT CASE	45
7	Repetição	48
7.1	Loop-DO.....	48
7.2	Loop-DO com variável de controle	51
7.3	Estruturas de repetição aninhadas.....	55

8	Entrada e saída formatada.....	57
8.1	Os formatos da linguagem FORTRAN	57
8.2	Saída de dados	60
8.2.1	INTEGER: descritor I.....	60
8.2.2	REAL: descritor F	61
8.2.3	REAL: descritor E	63
8.2.4	LOGICAL: descritor L	67
8.2.5	CHARACTER: descritor A	68
8.3	Entrada de dados.....	69
8.3.1	INTEGER: descritor I.....	69
8.3.2	REAL: descritores F e E.....	69
8.3.3	LOGICAL: descritor L	69
8.3.4	CHARACTER: descritor A	70
8.4	Controle de posição horizontal	71
8.4.1	Espaçamento horizontal: nX.....	71
8.4.2	Tabulação: Tc, TLc e TRc	72
8.5	Controle de posição vertical	72
8.5.1	O descritor de edição barra: /.....	72
8.6	Agrupamento de descritores	73
9	Funções	73
9.1	Chamada de funções	77
9.2	Regras de escopo	78
10	Módulos	78
11	Subrotinas	79
11.1	INTENT.....	80
11.2	O comando CALL	80
12	Arrays	80
12.1	Arrays de uma dimensão	81
12.1.1	Declarando um array	81
12.1.2	Loop-DO implícito	81
12.1.3	Entrada e saída de dados em um array	82
12.2	Arrays multi-dimensionais	83
12.2.1	Declarando um array	84
12.2.2	Entrada e saída de dados em um array	84
13	Compiladores.....	85
13.1	Salford FTM95.....	85
13.2	Force	87
14	Os 5 exemplos	88
14.1	Quatro operações aritméticas.....	88
14.2	Programa Gráfico	90
14.3	QUICKSORT	94
14.4	Vetores e Matrizes	100
14.5	Problema prático.....	102
14.5.1	Equação quadrática.....	102
14.5.2	Busca binária com quicksort.....	104
	Referências	111

1 História da linguagem de programação FORTRAN

No final da década de 40 e início da década de 50, reinavam os sistemas interpretativos. Uma das razões para isso, foi que a manipulação de números reais era simulada em software, uma vez que não havia suporte de hardware para isso. Grande parte do tempo de processamento era gasto no processamento de números reais em software e na simulação de indexação, tornando a interpretação uma despesa aceitável. Havia ainda os que preferiam a eficiência da linguagem de máquina codificada à mão.

Em maio de 1954, o anúncio do sistema IBM 704 marcou o fim da interpretação (pelo menos na computação científica), pois trazia como principais novidades: instruções de indexação e implementação de números reais em hardware (ver Figura 1.1).

As novas capacidades introduzidas pelo IBM 704 estimularam o desenvolvimento da linguagem FORTRAN (FORMula TRANslating), o que representou um grande avanço na computação.



Figura 1.1 – IBM 704

Foto extraída do site:

http://www.bitsavers.org/pdf/brl/compSurvey_Mar1961/BRL_jpgs/0405_IBM_704.jpg

1.1 Características do desenvolvimento do FORTRAN

Embora haja controvérsias, o FORTRAN é considerado por muitos como a primeira linguagem de alto nível compilada.

Deve-se considerar o ambiente peculiar em que se deu o seu desenvolvimento:

- Os computadores eram lentos e pouco confiáveis.
- Usado principalmente na computação científica.
- A programação de computadores não era eficiente.
- O objetivo principal dos primeiros compiladores FORTRAN era a velocidade da geração do código-objeto, devido ao seu custo elevado em relação ao custo dos programadores.

Esse ambiente influenciou diretamente as primeiras versões da linguagem.

1.2 FORTRAN 0

Em novembro de 1954, na IBM, o grupo liderado por John Backus produzem o relatório: “The IBM Mathematical FORMula TRANslating System: FORTRAN”. Era a descrição da primeira versão do FORTRAN (conhecida como FORTRAN 0), antes de ser implementada.

A primeira versão do FORTRAN prometia a eficiência dos programas codificados à mão e a facilidade dos programas escritos em pseudocódigo. Também prometia eliminar os erros de codificação e o processo de depuração.

1.3 FORTRAN I

O Fortran 0 modificado e implementado, foi chamado de FORTRAN I, e seu compilador foi lançado em abril de 1957.

Principais características do FORTRAN I:

- Formatação de entrada/saída.
- Nomes de variáveis de até seis caracteres (eram permitidos apenas dois caracteres no FORTRAN 0). Variáveis com nomes que iniciavam com I, J, K, L, M e N eram do tipo inteiro, e todas as outras eram do tipo real.
- Sub-rotinas definidas pelo usuário (embora elas ainda não pudessem ser compiladas separadamente).
- Instrução de seleção IF (expressão aritmética) N1, N2, N3
- Instrução DO loop.
- Limite virtual do tamanho dos programas: entre 300 e 400 linhas (esse limite era uma combinação da falta de capacidade de compilação independente do FORTRAN com a má confiabilidade do 704).

O FORTRAN 0 usava operadores relacionais como “<” e “>”, mas, como a tabela de caracteres do 704 não possuía esses caracteres, eles não puderam ser implementados no FORTRAN 1. Além disso, o IF lógico original foi substituído pela seleção aritmética (ver Figura 1.2), uma vez que a máquina possuía uma instrução de seleção de três caminhos baseado em registradores.

Apesar do ceticismo de muitos usuários em relação à eficiência do FORTRAN, uma pesquisa de 1958 mostrou que metade do código escrito para os 704 era feito em FORTRAN, isso dá uma idéia do sucesso da linguagem.

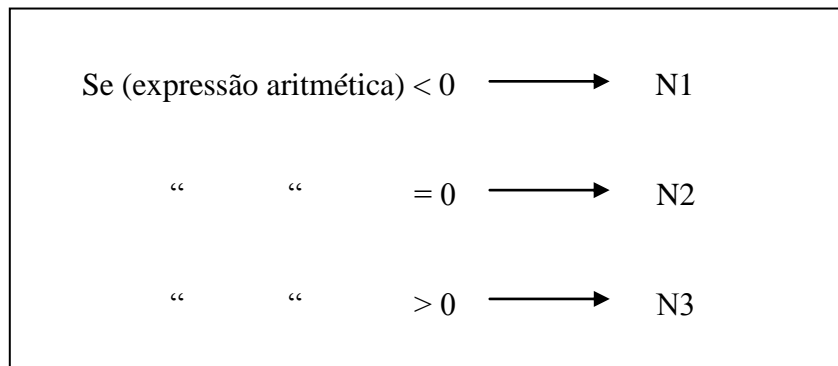


Figura 1.2
Seleção aritmética

1.4 FORTRAN II

O compilador FORTRAN II foi distribuído em 1958. Dentre as principais melhorias que ele trouxe, destaca-se a compilação independente de sub-rotinas. A compilação independente tornou possível escrever programas mais longos, além disso, podiam ser incluídas versões em linguagem de máquina, o que reduzia consideravelmente o tempo de compilação.

1.5 FORTRAN III

A terceira versão da linguagem de programação FORTRAN, foi desenvolvida mas jamais distribuída para uso público. Sua grande dependência de recursos de máquina, acabou inviabilizando o seu uso.

1.6 FORTRAN IV

Em 1962 é lançado o FORTRAN IV, que se tornou uma das linguagens mais usadas do seu tempo. Em relação ao FORTRAN II, o FORTRAN IV evoluiu em vários aspectos, como:

- Declarações de tipo explícitas para variáveis.
- Uma construção lógica IF.
- Capacidade de passar subprogramas como parâmetros a outros subprogramas.

1.7 FORTRAN 77

Foi lançada em 1978, trazendo novos recursos, tais como:

- Manipulação de cadeias de caracteres.
- Instruções lógicas de controle de laço.
- IF com ELSE opcional.

1.8 FORTRAN 90

Padronizada em 1992 (ANSI, 1992), essa versão evoluiu muito em relação a sua Antecessora. A seguir serão descritas as mudanças mais significativas.

- Um conjunto de funções para operações com matrizes.
- Dados dinâmicos: as matrizes podem ser alocadas e desalocadas por comando, se tiverem sido declaradas como ALLOCATABLE.
- Tipos derivados: uma forma de registros.
- Ponteiros são incluídos na linguagem.
- CASE: instrução de seleção múltipla.

- EXIT: instrução que permite sair de um laço sem completá-lo.
- CYCLE: retorna para o início do laço sem completar o ciclo.
- Os subprogramas podem ser recursivos.
- Módulos PRIVATE ou PUBLIC.

2 O Paradigma Imperativo

O paradigma imperativo é o mais usado atualmente. A maioria dos programas existentes no mercado foram feitos em uma linguagem imperativa.

Mas, o que caracteriza esse paradigma? A programação imperativa recebeu esse nome, devido à palavra latina “imperare” que significa comandar. Nesse paradigma, os comandos são responsáveis por atualizar (comandar) as variáveis armazenadas na memória. Processo esse realizado na forma seqüencial, na seqüência em que os comandos foram dispostos pelo programador.

Há vários fatores que contribuem para o amplo sucesso desse paradigma de programação. A compatibilidade entre as linguagens imperativas e as arquiteturas de máquina existentes, possibilita uma implementação mais eficiente e uma execução mais rápida de programas escritos em uma linguagem imperativa. A grande maioria dos programadores profissionais são especialistas, principalmente ou exclusivamente em linguagens imperativas. A semelhança entre o paradigma imperativo e o mundo natural, onde ações (comandos) modificam objetos (variáveis).

Podemos citar como exemplos de linguagens imperativas: Basic, Cobol, Fortran, C, Pascal e Ada.

3 Introdução à programação em FORTRAN

3.1 Escrevendo um programa em FORTRAN.

Os programas em FORTRAN têm a seguinte forma:

```
PROGRAM nome_do_programa
IMPLICIT NONE
[declaracao_variaveis_e_constantes]
[execução_programa]
[subprogramas]
END PROGRAM nome_do_programa
```

O programa começa com a palavra-chave “PROGRAM”, seguida da declaração “IMPLICIT NONE”. Após isso, temos o programa em si, e o que estiver dentro de [] é opcional.. Um programa em FORTRAN termina com “END PROGRAM” seguido do nome do programa.

Devem ser usados **comentários** para dar legibilidade aos programas em FORTRAN. Como eles são ignorados pelo compilador, o programador pode escrever o que desejar neles. Tudo o que vêm depois do sinal de exclamação “!”, é um comentário. Um comentário pode estar tanto no meio de uma linha, como ocupar uma linha inteira. Ex:

```
Read(*, *) A ! lê o valor de A (comentário no meio da linha)
! Comentário ocupando uma linha inteira.
```

Se uma declaração for muito extensa e não couber em uma única linha, ela pode **continuar em outra linha**, usando o caractere “&”. Ex:

```
RESU = (A * B) + 153 + K /    &
      (B + 45)
```

A expressão acima é igual a:

```
RESU = (A * B) + 153 + K / (B + 45)
```

Se for necessária uma **continuação de linha sem espaços**, usamos “&” nas duas linhas, sem deixar espaços entre o “e-comercial” e os demais caracteres da linha. Ex:

```
X = 4 * AreaDoTri&  
    &ângulo
```

A expressão acima é igual a:

```
X = 4 * AreaDoTriangulo
```

3.2 Tipos de dados.

3.2.1 Inteiro

Dados do tipo inteiro, são números positivos ou negativos, sem parte fracionária. Ou seja, não são permitidos vírgula nem ponto decimal. Ex: 12, -87, -3, +761.

3.2.2 Real

Dados do tipo real, podem ser representados na forma decimal ou exponencial.

- *Representação Decimal*: O número deve necessariamente possuir ponto decimal (lembrando que vírgulas não são permitidas), mas o sinal é opcional.

Certo: 15.43, 158., -25.67, -.21.

Errado: 48,21; 27.

- *Representação Exponencial*: É um número inteiro ou real em representação decimal, seguido de “E” ou “e”, seguido do expoente (um número inteiro).

Exemplos:

64.3158E2 ou 64.3158e2 é igual a 6431.58

8.92E-2 ou 8.92e-2 é igual a 0.0892

-5.5E-1 ou -5.5e-1 é igual a -0.55

3.2.3 Complexo

Dados do tipo complexo não serão abordados neste curso.

3.2.4 Lógico

Dados do tipo lógico são “true” e “false”. Em FORTRAN eles devem ser escritos como .TRUE. e .FALSE.

3.2.5 String de caracteres

Dados do tipo caractere são representados entre aspas duplas ou apóstrofes. O comprimento de uma string é o número de caracteres entre as aspas duplas ou apóstrofes,

incluindo os espaços. Se o comprimento de uma string for zero, dizemos que é uma string vazia.

Exemplos corretos:

‘Brasil’ e “Brasil”: comprimento = 6

‘ ’ e “ ”: comprimento = 1

‘’ e “””: comprimento = 0 (string vazia)

Exemplos incorretos:

“cachorro

‘televisão’

Se um apóstrofo é usado em uma string, então aspas duplas devem ser usadas para envolver a string. Exemplo:

“I don’t believe!”

3.3 Identificadores

Os identificadores de FORTRAN podem ter até 31 caracteres, sendo que o primeiro caractere deve ser uma letra. Os caracteres restantes, se houverem podem ser letras, dígitos ou underlines. Os identificadores de FORTRAN são *case insensitive*, ou seja, BRASIL, Brasil, BrAsIl, brasil, brasiL são identificadores iguais.

Embora FORTRAN permita usar palavras-chave da linguagem como identificadores (é possível usar como identificadores, palavras como: END, PROGRAM, DO, ...), essa não é uma boa prática de programação, pois afeta a legibilidade do programa.

Usar nomes significativos para os identificadores é uma boa prática de programação, pois evita quantidade excessiva de comentários.

CERTO: N, num, resu, M25

ERRADO: 6J, num#

3.4 Declaração de variáveis

A declaração de variáveis em FORTRAN é feita da seguinte forma:

```
tipo :: lista_de_variáveis
```

Tipo pode ser INTEGER, REAL, COMPLEX, LOGICAL ou CHARACTER. A lista de variáveis é o nome das variáveis separadas com vírgulas. Exemplo:

- Inteiros e Reais

```
INTEGER :: num, soma, Total
```

```
REAL :: media, resu, num1
```

- Caracteres

A declaração de caracteres deve levar em conta o comprimento. Isso pode ser feito de duas maneiras:

- CHARACTER(LEN=i) declara variáveis do tipo CHARACTER de tamanho i. Por exemplo, Curso e Universidade são variáveis do tipo CHARACTER que podem armazenar uma string de no máximo 40 caracteres.

```
CHARACTER(LEN=40) :: Curso, Universidade
```

- CHARACTER(i) também declara variáveis do tipo CHARACTER de

tamanho i. Por exemplo, Nome e Telefone são variáveis do tipo CHARACTER que podem armazenar uma string de no máximo 15 caracteres.

```
CHARACTER(15) :: Nome, Telefone
```

Para variáveis do tipo CHARACTER que armazenam apenas um caractere (comprimento 1), a parte do comprimento pode ser omitida na declaração. Por exemplo, classe e tipo são variáveis do tipo CHARACTER que podem armazenar no máximo 1 caractere.

```
CHARACTER :: classe, tipo
```

Existe ainda a possibilidade de declarar variáveis de tamanhos diferentes em uma única instrução. Para isso, devemos colocar “*i” na direita da variável cujo comprimento queremos especificar. Desse modo apenas essa variável terá o seu comprimento especificado, as outras variáveis da lista não serão afetadas. No exemplo abaixo, as variáveis podem armazenar no máximo 20 caracteres, com exceção da variável endereço que armazena até 30 caracteres e da variável telefone que armazena até 12 caracteres.

```
CHARACTER(LEN=20) :: nome, endereço*30, empresa, cargo, telefone*12
```

Podemos ainda declarar variáveis do tipo caractere, usando o “especificador de tamanho assumido”. Ou seja, colocar um asterisco *, no lugar do comprimento, quer dizer que o comprimento das variáveis declaradas será determinado em outro lugar. Esse tipo de declaração geralmente é usada em argumentos de subprogramas e em PARAMETER.

Exemplo:

```
CHARACTER(LEN=*) :: cargo, empresa
```

3.5 Atribuindo uma constante a um nome

Em FORTRAN podemos atribuir uma constante a um nome, usando o atributo `PARAMETER`.

Às vezes, digitar várias vezes uma constante pode ser incômodo e nada prático, como por exemplo, a constante matemática 3.1415926. Nesses casos, é conveniente usar um nome (por exemplo `PI`) para se referir à constante.

Usar um nome para referenciar uma constante pode também aumentar a legibilidade do programa, tornando-o auto-explicativo. Assim, podemos usar um nome como “Limite” ou “Max” para nos referirmos ao tamanho de um vetor.

Devemos lembrar que esse nome não é uma variável, ele é apenas um alias, um nome alternativo para a constante. Depois de atribuir

Para atribuir uma constante a um nome, devemos colocar o atributo `PARAMETER` depois do <tipo> (separando as palavras com uma vírgula) e antes dos dois pontos duplos (`::`). Depois de cada nome, vem o sinal igual (`=`), seguido da expressão cujo valor será atribuído ao nome.

Exemplos:

```
INTEGER, PARAMETER :: Limite = 100, habitantes = 30 000
```

```
REAL, PARAMETER :: PI = 3.1415926, juros = 6.8
```

```
CHARACTER(LEN=5), PARAMETER :: Nome = 'Lucas', Idade = '28'
```

Um nome pode ser definido como o resultado de uma operação aritmética.
Exemplo:

```
INTEGER, PARAMETER :: Horas = 160, Valor_Hora: 15, Salário = Horas *  
                        Valor_Hora
```

Para evitar falta ou desperdício de espaço para armazenar uma string, podemos usar o “especificador de tamanho assumido”, através dele o comprimento da string apontada pelo nome é determinada pela própria string. No exemplo abaixo, como o comprimento de ”Lucas” e “28” é 4 e 2, então, o comprimento dos nomes: Nome e Idade é 4 e 2 respectivamente.

```
CHARACTER(LEN=*), PARAMETER :: Nome = 'Lucas', Idade = '28'
```

3.6 A instrução de atribuição

A instrução de atribuição têm a seguinte forma:

variavel = expressão

Essa expressão armazena o resultado da expressão que está à direita do operador de atribuição, na variável que se encontra na esquerda. Se os tipos da variável e do resultado da expressão forem diferentes, o resultado será convertido para o tipo da variável.

Exemplo:

```
INTEGER :: Base, Altura, AreaTr

Base = 10
Altura = 6
AreaTr = (Base * Altura) / 2
```

4 Operadores e expressões

FORTTRAN têm quatro tipos de operadores: aritméticos, relacionais, lógicos e de

caracteres. A figura 3.1 mostra três desses tipos, além de seu nível de prioridade e a ordem de associatividade.



Maior prioridade		TIPO	OPERADORES					ASSOCIATIVIDADE
		Aritméticos	**					dir. p/ esq.
			*	/				esq. p/ dir.
			+	-				esq. p/ dir.
		Relacionais	<=	>	>=	==	/=	nenhuma
Menor prioridade		Lógicos	.NOT.					dir. p/ esq.
			.AND.					esq. p/ dir.
			.OR.					esq. p/ dir.
			.EQV.		.NEQV.			esq. p/ dir.

Figura 3.1
Tabela de Operadores

4.1 Aritméticos

Como vimos na tabela, os operadores aritméticos são: adição (+), subtração (-), multiplicação (*), divisão (/) e exponenciação (**). Uma expressão aritmética pode ser construída usando esses operadores e operandos dos tipos INTEGER, REAL ou COMPLEX.

Para avaliar uma expressão aritmética devemos lembrar de algumas regras importantes. As expressões sempre são avaliadas da esquerda para a direita. Quando um operador é encontrado, sua prioridade é comparada com a prioridade do próximo operador. Se o nível de prioridade do próximo operador é menor, o primeiro operador é executado. Se o nível de prioridade do próximo operador é igual, as regras de associatividade determinam

qual operador deve ser executado primeiro. Se o nível de prioridade do próximo operador é maior, a avaliação da expressão deve continuar com o próximo operador.

As expressões aritméticas podem ser classificadas em simples ou mistas.

4.1.1 Expressões aritméticas de modo simples

Quando todos os valores ou variáveis são do mesmo tipo.

Exemplo:

```
resu = 2 * 4 * 5 / 3 ** 2
resu = (2 * 4) * 5 / 3 ** 2
resu = 8 * 5 / 3 ** 2
resu = (8 * 5) / 3 ** 2
resu = 40 / 3 ** 2
resu = 40 / (3 ** 2)
resu = 40 / 9
resu = 4
```

4.1.2 Expressões aritméticas de modo misto

Quando temos valores ou variáveis de tipos diferentes (INTEGER, REAL ou COMPLEX). Considerando uma expressão que contenha valores ou variáveis dos tipos INTEGER e REAL, os valores inteiros serão sempre convertidos para valores reais. As possíveis combinações de INTEGER e REAL são mostradas na tabela abaixo:

	INTEGER	REAL
INTEGER	INTEGER	REAL
REAL	REAL	REAL

Para resolver expressões aritméticas de modo misto, devemos primeiramente, seguir as regras de avaliação para expressões aritméticas de modo simples. Para cada operador localizado, devemos ver se os seus operandos são do mesmo tipo ou não. Se os operandos são do mesmo tipo, devemos simplesmente calcular o resultado desse operador. Mas se um operando for um número inteiro e o outro um número real, devemos converter o operando inteiro para um número real (adicionando *.0* no final do número inteiro), e então calcular o resultado do operador.

Exemplo:

```
resu = 5 * (11.0 - 5) ** 2 / 4 + 9
resu = 5 * (11.0 - 5.0) ** 2 / 4 + 9
resu = 5 * 6.0 ** 2 / 4 + 9
resu = 5 * (6.0 ** 2) / 4 + 9
resu = 5 * 36.0 / 4 + 9
resu = (5 * 36.0) / 4 + 9
resu = 180.0 / 4 + 9
resu = 180.0 / 4.0 + 9
resu = (180.0 / 4.0) + 9
resu = 45.0 + 9
resu = 45.0 + 9.0
resu = 54.0
```

Exceção: O operador de exponenciação é uma exceção à regra. Na expressão $a ** b$ por exemplo, a é um número real, enquanto b é um inteiro positivo.

4.2 Relacionais

Em FORTRAN temos seis operadores relacionais, são eles:

<	menor que
---	-----------

<=	menor que ou igual a
>	maior que
>=	maior que ou igual a
==	igual a
/=	diferente de

Os operadores relacionais em FORTRAN têm dois operandos que podem ser aritméticos ou strings, mas os dois devem ser do mesmo tipo, ou seja, dois operandos aritméticos ou dois operandos do tipo string.

Com os operadores relacionais, o programador pode fazer comparações, as quais produzem como resultado, um valor lógico (true ou false).

Exemplo:

$10 + 2 > 17 - 3$ FALSE.

$10 * 2 == 12 + 8$TRUE.

Também podemos utilizar os operadores relacionais para comparar strings de caracteres. A comparação é feita caractere por caractere, e da esquerda para à direita. Se dois caracteres iguais forem encontrados, a comparação continua com o próximo par de caracteres. Mas se os caracteres forem diferentes, a string com o menor caractere (ordem alfabética, já que a tabela ASCII está organizada em ordem alfabética) é considerada a menor string.

Exemplo:

c	A	r	r	o	ç	a		
=	=	=	=					
c	A	r	r	a	p	a	t	o

A comparação “carroça < carrapato” é .TRUE.

4.3 Lógicos

Em FORTRAN temos seis operadores lógicos, são eles:

.NOT.	NÃO lógico
.AND.	E lógico
.OR.	OU lógico
.EQV.	EQUIVALÊNCIA lógica
.NEQV.	NÃO EQUIVALÊNCIA lógica

Os operadores lógicos podem ser usados apenas em expressões que produzam como resultado valores lógicos (.TRUE. ou .FALSE.). Devemos lembrar que os operadores lógicos têm menor prioridade que os operadores aritméticos e relacionais, e que os operadores relacionais não podem ser usados para comparar expressões que produzam valores lógicos como resultado.

A seguir temos um exemplo, de comparações realizadas com operadores lógicos (se você tiver alguma dúvida sobre o resultado produzido pelos operadores lógicos, veja a próxima seção, sobre tabelas verdade):

```

      expr1      expr2      expr3
      ┌───┐      ┌───┐      ┌───┐
resu = .NOT. (4 + 3 > 2 * 3) .OR. .NOT. (7 + 8 == 6 * 3 - 3) .AND. (15 + 9 <= 48 / 4)
resu = .NOT. (4 + 3 > (2 * 3)) .OR. .NOT. (7 + 8 == (6 * 3) - 3) .AND. (15 + 9 <= (48 / 4))
resu = .NOT. (4 + 3 > 6) .OR. .NOT. (7 + 8 == 18 - 3) .AND. (15 + 9 <= 12)
resu = .NOT. ((4 + 3) > 6) .OR. .NOT. ((7 + 8) == (18 - 3)) .AND. ((15 + 9) <= 12)
resu = .NOT. (7 > 6) .OR. .NOT. (15 == 15) .AND. (24 <= 12)
resu = .NOT. .TRUE. .OR. .NOT. .TRUE. .AND. .FALSE.
resu = (.NOT. .TRUE.) .OR. .NOT. .TRUE. .AND. .FALSE.
resu = .FALSE. .OR. .NOT. .TRUE. .AND. .FALSE.
resu = .FALSE. .OR. (.NOT. .TRUE.) .AND. .FALSE.
resu = .FALSE. .OR. .FALSE. .AND. .FALSE.
resu = .FALSE. .OR. (.FALSE. .AND. .FALSE.)
```



```
resu = .FALSE. .OR. .FALSE.
```

```
resu = .FALSE.
```

O resultado da expressão lógica acima, pode ser atribuído a uma variável do tipo LOGICAL:

```
LOGICAL :: resu
```

```
resu = .NOT. expr1 .OR. .NOT. expr2 .AND. expr3
```

4.3.1 Tabelas verdade

Os valores lógicos produzidos pelas expressões lógicas são determinados pelas tabelas verdade dos operadores correspondentes.

- Tabela verdade do operador .NOT.

expressão	!(expressão)
.TRUE.	.FALSE.
.FALSE.	.TRUE.

O operador .NOT. é um operador unário, ou seja, tem um único operando.

- Tabela verdade do operador .AND.

expr1	expr2	expr1 .AND. expr2
.TRUE.	.TRUE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.

.FALSE.	.TRUE.	.FALSE.
.FALSE.	.FALSE.	.FALSE.

- Tabela verdade do operador .OR.

expr1	expr2	expr1 .OR. expr2
.TRUE.	.TRUE.	.TRUE.
.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.TRUE.
.FALSE.	.FALSE.	.FALSE.

- Tabela verdade do operador .EQV.

expr1	expr2	expr1 .EQV. expr2
.TRUE.	.TRUE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.
.FALSE.	.FALSE.	.TRUE.

É um operador muito usado para comparar se dois valores lógicos são iguais.

- Tabela verdade do operador .NEQV.

expr1	expr2	Expr1 .NEQV. expr2
.TRUE.	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.TRUE.
.FALSE.	.FALSE.	.FALSE.

.NEQV. é o oposto de .EQV., e é usado para comparar se dois valores lógicos são diferentes.

4.4 Caracteres

4.4.1 Operador de concatenação: //

Fortran têm apenas um operador de caracteres, o operador de concatenação: “//”. Se temos duas strings, str1 e str2 de comprimento “i” e “j”, respectivamente, a concatenação de str1 e str2 é escrita como “str1 // str2” e têm comprimento “i + j”.

Exemplo:

```
CHARACTER (LEN == 6) :: livro = "livro"
CHARACTER (LEN == 6) :: antigo = "antigo"
CHARACTER (LEN == 12) :: resu

resu = livro // antigo
```

* A variável “resu” contém a string “livro*antigo”, onde “*” denota um espaço. Esse espaço veio da variável “livro” que tem “um espaço sobrando”.

4.4.2 Substrings

Uma substring é qualquer conjunto de caracteres consecutivos, pertencentes a uma string. Para indicar uma substring, nós devemos anexar o “especificador de comprimento” no fim da variável CHARACTER.

O “especificador de comprimento” tem a seguinte forma: “(num1 : num2)”, onde “num1” é um número inteiro que indica a primeira posição da substring, e “num2” é um

número inteiro que indica a última posição da substring. Assim se o conteúdo da variável “nome” é “Mariana”, então “nome(2:4)” é a string “ari”.

* “num1” e “num2” também podem ser uma expressão como “2 + 3” ou “4 * 3”.

Se “num1” não for colocado, o valor assumido será 1. Se “num2” não for colocado, o valor assumido será o último caractere da string. Assim “nome(5:)” é a string “ana”.

Devemos lembrar que “num1” deve ser maior ou igual a 1, e “num2” deve ser menor ou igual que o comprimento da string.

Uma variável CHARACTER com um “especificador de comprimento” pode ser usada no lado esquerdo de uma atribuição. Isso significa que a string do lado direito será atribuída à substring especificada da variável do lado esquerdo. Para exemplificar, usaremos a variável “curso”, que tem comprimento 10 e o seguinte conteúdo: “computacao”.

```
curso(2:4) = "abcd"      ! O conteúdo de "curso" é "cabcdtacao".
```

```
curso(8:) = "abc"       ! O conteúdo de "curso" é "computaabc".
```

```
curso(4:7) = "ab"       ! O conteúdo de "curso" é "comab**cao"
```

O programa a seguir, usa substrings e o operador de concatenação para mostrar a data e a hora (veja o capítulo 4 para saber mais sobre o comando WRITE).

```
! Autor: Weyden Daniel Castro
! Data: 05/2005
! Universidade: UNASP
! Curso: Ciencia da Computacao
! Disciplina: Paradigmas de Linguagens de Programacao

! -----
! Este programa usa DATE_AND_TIME() para extrair a data e a hora
```

```

! do sistema. Em seguida, converte essa informacao para um formato
! mais legivel. Isso e realizado usando substrings e o operador
! de concatenacao: "//".
! -----

PROGRAM DataHora
    IMPLICIT NONE

    CHARACTER(LEN = 8)  :: DataINFO, DataLegivel*12 ! aaaammdd
    CHARACTER(LEN = 4)  :: Ano, Mes*2, Dia*2

    CHARACTER(LEN = 10) :: HoraINFO, HoraLegivel*12 ! hhmmss.sss
    CHARACTER(LEN = 2)  :: Hora, Minuto, Segundo*6

    CALL DATE_AND_TIME(DataINFO, HoraINFO)

! decomposicao de DataINFO em ano, mes e dia, usando substrings.
! DataINFO recebe a informacao na forma de aaaammdd, onde aaaa = ano,
! mm = mes e dd = dia

    Ano = DataINFO(1:4)
    Mes = DataINFO(5:6)
    Dia = DataINFO(7:8)

    DataLegivel = Dia // '/' // Mes // '/' // Ano

    WRITE(*,*) ' Data do Sistema -> ', DataINFO
    WRITE(*,*) '           Ano -> ', Ano
    WRITE(*,*) '           Mes -> ', Mes
    WRITE(*,*) '           Dia -> ', Dia
    WRITE(*,*) ' Data Legivel -> ', DataLegivel

! decomposicao de HoraINFO em hora, minuto e segundo, usando substrings.
! HoraINFO recebe a informacao na forma de hhmmss.sss, onde h = hora, m
= minuto
! e s = segundo

    Hora = HoraINFO(1:2)

```

```

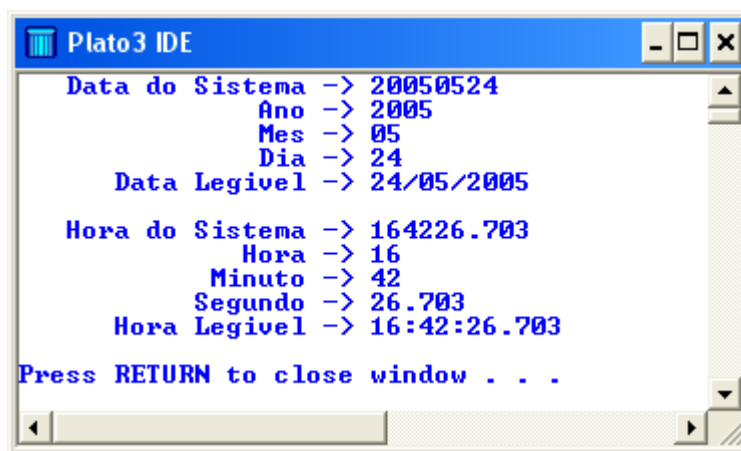
Minuto  = HoraINFO(3:4)
Segundo = HoraINFO(5:10)

HoraLegivel = Hora // ':' // Minuto // ':' // Segundo

WRITE(*,*)
WRITE(*,*) '  Hora do Sistema -> ', HoraINFO
WRITE(*,*) '           Hora -> ', Hora
WRITE(*,*) '           Minuto -> ', Minuto
WRITE(*,*) '           Segundo -> ', Segundo
WRITE(*,*) '  Hora Legivel -> ', HoraLegivel

END PROGRAM  DataHora

```



5 Entrada e saída em lista

5.1 READ

A entrada de dados em lista é conseguida com o comando READ. Com esse comando, é possível ler, do teclado, valores para um conjunto de variáveis. O comando READ tem as seguintes formas:

```
READ(*, *) var1, var2, ..., varn  
READ(*, *)
```

A primeira forma consiste de READ(*, *), seguido de uma lista de variáveis. Essa forma irá ler valores do teclado, e colocar esses valores nas variáveis, na respectiva ordem em que foram listadas.

A segunda forma, sem nenhuma lista de variáveis, simplesmente pula uma linha de entradas.

O seguinte exemplo lê strings para as variáveis: nome, sobrenome e empresa, e em seguida, lê valores inteiros para as variáveis: idade e código.

```
CHARACTER (LEN = 15) :: nome, sobrenome, empresa  
INTEGER :: idade, codigo  
  
READ(*, *) nome, sobrenome, empresa, idade, codigo
```

A seguir, algumas regras que devem ser observadas no uso de READ:

- READ começa a receber entrada de valores, com uma nova linha, assim, se o número de valores inseridos for maior que o número de variáveis listadas, os valores excedentes serão ignorados.

Exemplo:

Considerando o seguinte:

```
INTEGER :: A, B, C, D, E, F
```

```
READ(*,*) A, B, C
```

```
READ(*,*) D, E, F
```

Se o usuário digitar os valores:

```
100 200 300 400
```

```
500 600 700 800
```

As variáveis A, B, C, D, E, F, e G receberão 100, 200, 300, 500, 600 e 700, respectivamente. Já os valores 400 e 800 serão perdidos, pois excedem o número de variáveis listas nas expressões READ correspondentes.

- Uma limitada conversão de tipos é possível em um comando READ. Se um valor inteiro for inserido para uma variável do tipo REAL, o inteiro inserido será convertido para um número real. Mas, se um valor real for inserido para uma variável do tipo INTEGER, um erro ocorrerá.
- Se o comprimento de uma string inserida for menor que o comprimento da variável CHARACTER, os caracteres restantes serão preenchidos com espaços. Mas, se o comprimento de uma string inserida for maior que o comprimento da variável CHARACTER, a string perderá os caracteres necessários para que o seu comprimento seja igual ao da variável onde será armazenada.
- Para inserir um valor lógico usando READ, o usuário deve digitar T para .TRUE. e F para .FALSE.

5.2 WRITE

A saída de dados em lista é conseguida com o comando WRITE. Esse comando permite mostrar os resultados de um conjunto de expressões e strings de caracteres. Em geral, WRITE é utilizado para imprimir na tela do computador. O comando WRITE tem as seguintes formas:

```
WRITE(*, *) exp1, exp2, ..., expn  
WRITE(*, *)
```

A primeira forma consiste de WRITE(*, *), seguido de uma lista de expressões aritméticas ou strings de caracteres. O computador avalia as expressões aritméticas e mostra os resultados, mas se nenhum valor for atribuído à variável, o resultado exibido será imprevisível.

A segunda forma, que é apenas WRITE(*, *), mostra uma linha em branco.

O seguinte exemplo mostra a string contida na variável Figura, o resultado de “Lado1 + Lado2”, e o número REAL contido na variável Área.

```
CHARACTER(LEN=10) :: Figura  
REAL :: Lado1, Lado2, Altura, Area  
  
WRITE(*, *) Figura, (Lado1 + Lado2), Área
```

Para mostrar valores lógicos usando WRITE, FORTRAN usa T para .TRUE. e F para .FALSE.

6 Seleção

6.1 IF lógico

O IF lógico é a forma mais simples de seleção que a linguagem FORTRAN possui. Essa estrutura tem a seguinte forma:

```
IF (expressao-logica) unica-instrucao
```

Essa estrutura permite uma única instrução, que não pode ser um outro IF. A execução do IF lógico começa com a avaliação da expressão lógica, que produz um valor lógico. Se o resultado da avaliação é `.TRUE.`, a “unica-instrucao” é executada, e o programa continua com a próxima instrução após a estrutura IF. Mas, se o resultado da avaliação é `.FALSE.`, o programa não executa a “unica-instrucao”, pulando direto para a próxima instrução após a estrutura IF. Veja o seguinte exemplo:

```
! Autor: Weyden Daniel Castro
! Data: 05/2005
! Universidade: UNASP
! Curso: Ciencia da Computacao
! Disciplina: Paradigmas de Linguagens de Programacao

! -----
! Este programa le dois valores inteiros, e encontra o menor deles.
! -----

PROGRAM IFmenor
  IMPLICIT NONE

  INTEGER :: a, b, menor

  WRITE(*, *) '===== '
  WRITE(*, *) 'MENOR DE DOIS INTEIROS'
  WRITE(*, *) '===== '
  WRITE(*, *)
  WRITE(*, *) 'Insira dois numeros inteiros'
  READ(*,*) a, b
```

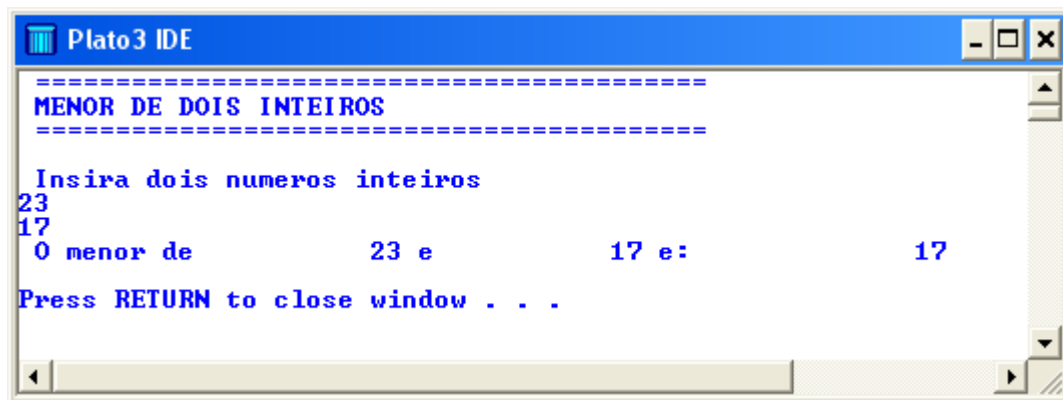
```
menor = a
```

! se a expressao logica for verdadeira, a instrucao "Menor = b" sera executada

```
IF (a > b)  menor = b
```

```
WRITE(*,*)  'O menor de ', a, ' e ', b, ' e: ', menor
```

```
END PROGRAM IFmenor
```



6.2 IF–THEN–END IF

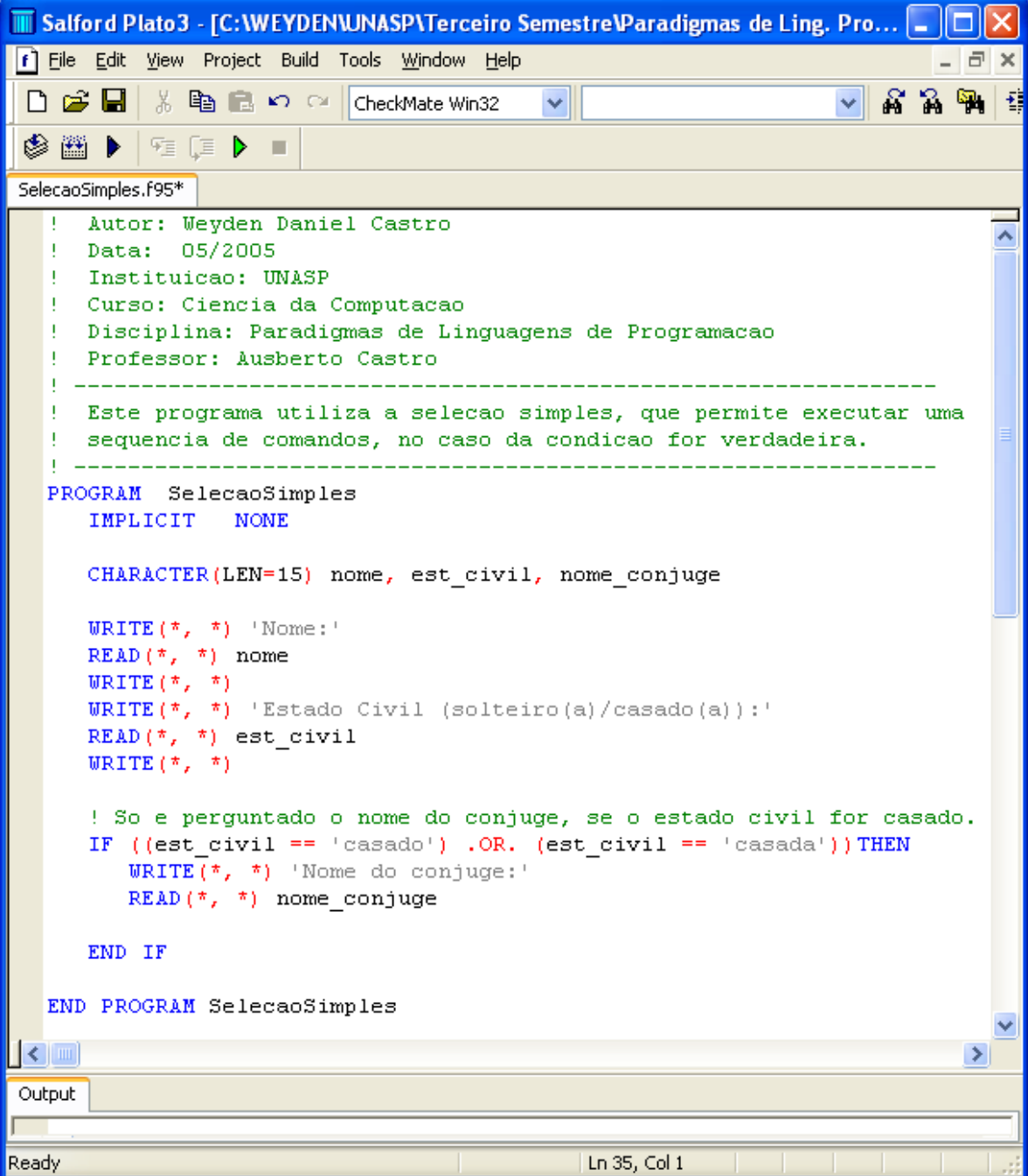
A estrutura “IF–THEN–END IF”, executa ações apenas se a condição (expressão lógica) for verdadeira. O que diferencia a estrutura “IF logico”, da estrutura “IF–THEN–END IF”, é que esta pode executar um conjunto de instruções, enquanto aquela executa uma única instrução. Veja abaixo a forma da estrutura “IF–THEN–END IF”:

```
IF (expressao-logica) THEN
    instruções
END IF
```

A execução da estrutura “IF–THEN–END IF” começa com a avaliação da expressão lógica, que produz um valor lógico. Se o resultado da avaliação é .TRUE., as “instruções”

no corpo da estrutura são executadas. Mas, se o resultado da avaliação é `.FALSE.`, o programa não executa as “instruções” do corpo da estrutura, pulando direto para a próxima instrução após a estrutura `IF–THEN–END IF`.

Veja no seguinte exemplo de programa, que será executado duas vezes para mostrar que as instruções do corpo da estrutura `IF–THEN–END IF`, só são executadas se a expressão lógica for `.TRUE.`:



Salford Plato3 - [C:\WEYDEN\UNASP\Terceiro Semestre\Paradigmas de Ling. Pro...

File Edit View Project Build Tools Window Help

CheckMate Win32

SelecaoSimples.f95*

```
! Autor: Weyden Daniel Castro
! Data: 05/2005
! Instituicao: UNASP
! Curso: Ciencia da Computacao
! Disciplina: Paradigmas de Linguagens de Programacao
! Professor: Ausberto Castro
! -----
! Este programa utiliza a selecao simples, que permite executar uma
! sequencia de comandos, no caso da condicao for verdadeira.
! -----
PROGRAM SelecaoSimples
  IMPLICIT NONE

  CHARACTER(LEN=15) nome, est_civil, nome_conjuge

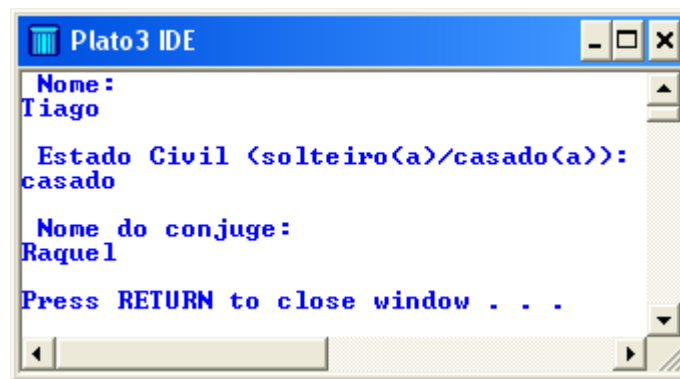
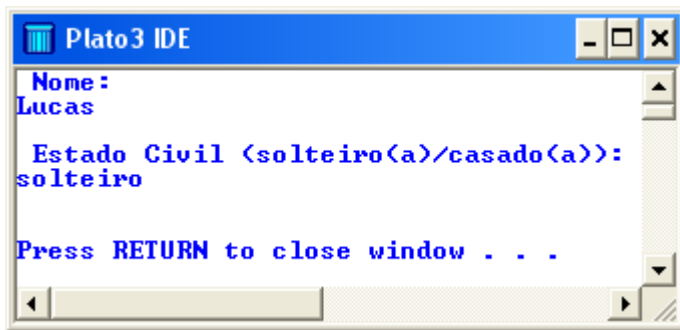
  WRITE(*, *) 'Nome:'
  READ(*, *) nome
  WRITE(*, *)
  WRITE(*, *) 'Estado Civil (solteiro(a)/casado(a)):'
  READ(*, *) est_civil
  WRITE(*, *)

  ! So e perguntado o nome do conjuge, se o estado civil for casado.
  IF ((est_civil == 'casado') .OR. (est_civil == 'casada')) THEN
    WRITE(*, *) 'Nome do conjuge:'
    READ(*, *) nome_conjuge
  END IF

END PROGRAM SelecaoSimples
```

Output

Ready Ln 35, Col 1



6.3 IF-THEN-ELSE-END IF

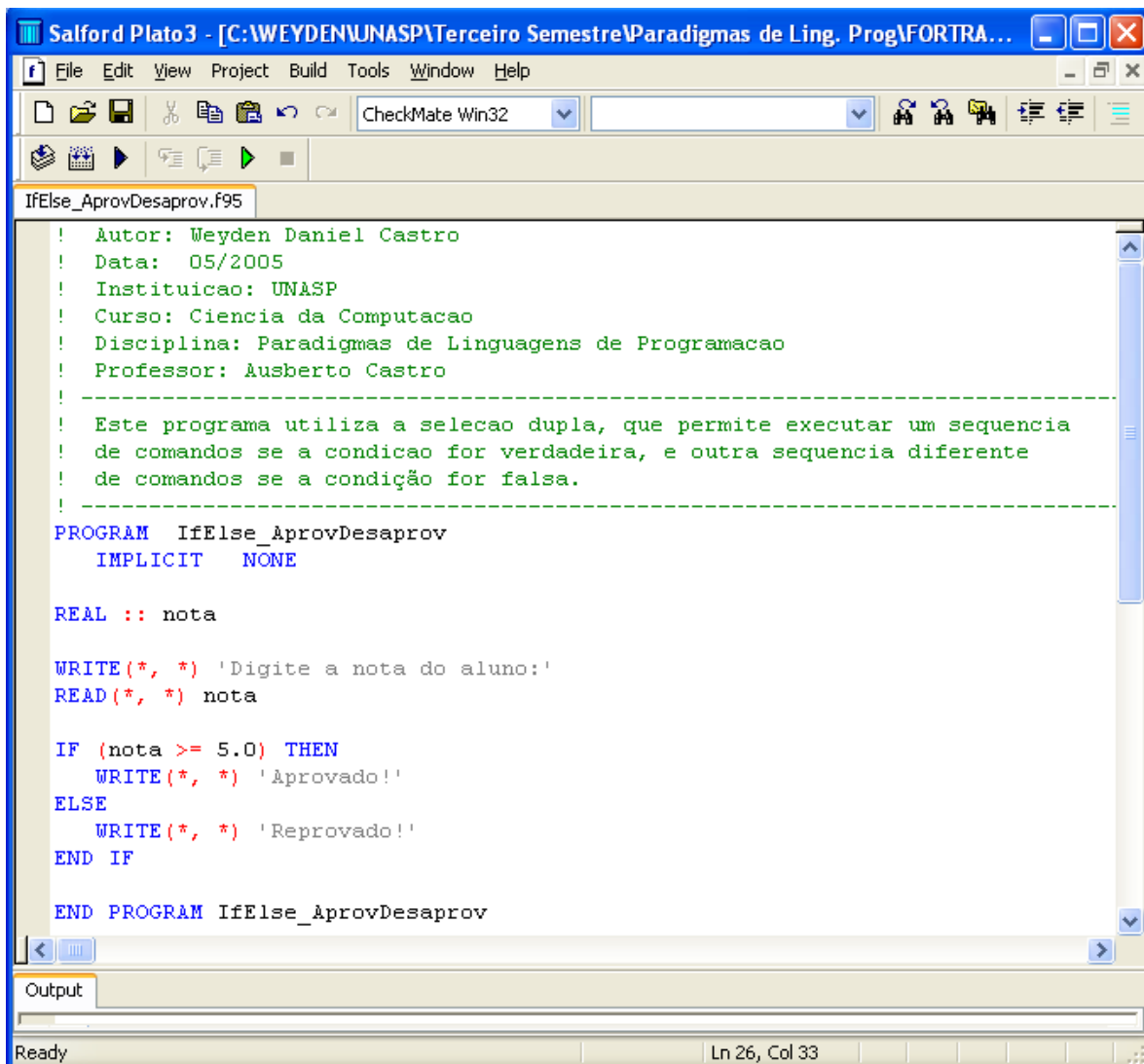
Essa é a **seleção dupla** da linguagem FORTRAN. Ou seja, um determinado grupo de instruções é executado se a expressão lógica for verdadeira (.TRUE.), e outro grupo de instruções será executado se a expressão lógica for falsa (.FALSE.). Veja abaixo a forma da estrutura IF-THEN-ELSE-END IF:

```
IF (expressao-logica) THEN
    instruções-1
ELSE
    instruções-2
END IF
```

A execução da estrutura “IF-THEN-ELSE-END IF” começa com a avaliação da expressão lógica, que produz um valor lógico. Se o resultado da avaliação é .TRUE., o

primeiro grupo de instruções (instruções-1) é executado. Mas, se o resultado da avaliação é .FALSE., o segundo grupo de instruções (instruções-2) é executado.

Veja o seguinte programa que lê a nota do aluno, e avalia se ele merece ser reprovado ou aprovado. Ele ilustra a utilidade da estrutura “IF–THEN–ELSE–END IF”, que permite que uma ação diferente seja tomada se a condição for verdadeira ou falsa.



The screenshot shows the Salford Plato3 IDE window titled "Salford Plato3 - [C:\WEYDEN\UNASP\Terceiro Semestre\Paradigmas de Ling. Prog\FORTRA...". The menu bar includes File, Edit, View, Project, Build, Tools, Window, and Help. The toolbar contains icons for file operations and execution. The active file is "IfElse_AprovDesaprov.f95". The code editor displays the following Fortran 90 program:

```
! Autor: Weyden Daniel Castro
! Data: 05/2005
! Instituicao: UNASP
! Curso: Ciencia da Computacao
! Disciplina: Paradigmas de Linguagens de Programacao
! Professor: Ausberto Castro
! -----
! Este programa utiliza a selecao dupla, que permite executar um sequencia
! de comandos se a condicao for verdadeira, e outra sequencia diferente
! de comandos se a condicao for falsa.
! -----
PROGRAM IfElse_AprovDesaprov
  IMPLICIT NONE

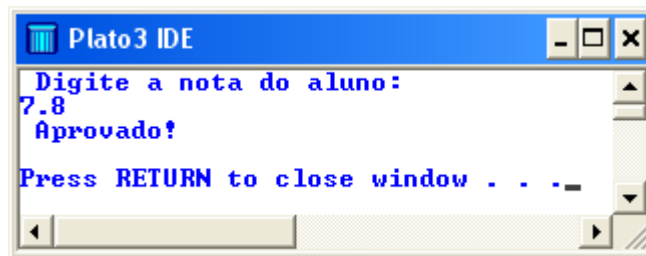
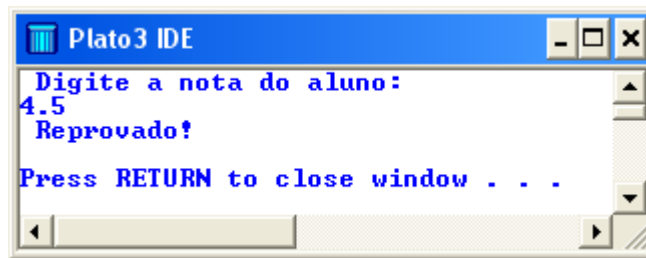
  REAL :: nota

  WRITE(*, *) 'Digite a nota do aluno:'
  READ(*, *) nota

  IF (nota >= 5.0) THEN
    WRITE(*, *) 'Aprovado!'
  ELSE
    WRITE(*, *) 'Reprovado!'
  END IF

END PROGRAM IfElse_AprovDesaprov
```

At the bottom, there is an "Output" window and a status bar showing "Ready" and "Ln 26, Col 33".



6.4 Estruturas de seleção aninhadas

As partes THEN e ELSE das estruturas de seleção, podem conter outras estruturas de seleção em qualquer uma de suas três formas (IF lógico, IF-THEN-END IF, IF-THEN-ELSE-END IF). Esse tipo de construção pode ser muito útil para o programador, que pode aninhar a quantidade de estruturas que for necessária. No entanto, muitas estruturas aninhadas prejudicam a legibilidade do programa.

Veja o seguinte programa para descobrir o menor de três números:

```
! Autor: Weyden Daniel Castro
! Data: 05/2005
! Instituicao: UNASP
! Curso: Ciencia da Computacao
! Disciplina: Paradigmas de Linguagens de Programacao
! Professor: Ausberto Castro
```

```
! -----
```



```

! Este programa calcula o menor de tres numeros inteiros
! usando estruturas de seleção aninhadas.
! -----

PROGRAM  aninhamento_selec
  IMPLICIT  NONE

  INTEGER :: a, b, c
  INTEGER :: menor

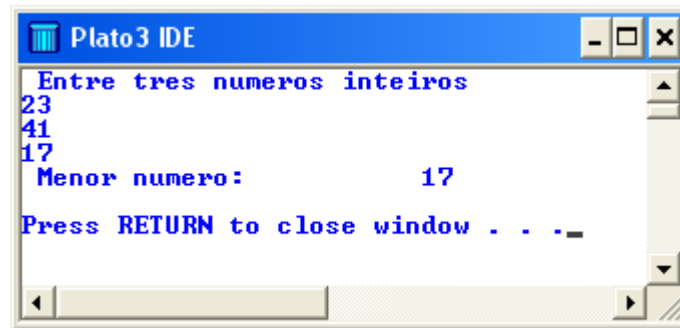
  WRITE(*, *) 'Entre tres numeros inteiros'
  READ(*, *) a, b, c

  IF (a < b) THEN
    IF (a < c) THEN
      menor = a
    ELSE
      menor = c
    END IF
  ELSE
    IF (b < c) THEN
      menor = b
    ELSE
      menor = c
    END IF
  END IF

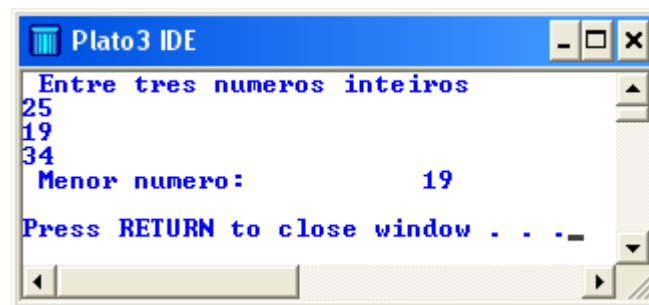
  WRITE(*, *) 'Menor numero: ', menor

END PROGRAM aninhamento_selec

```



```
Plato3 IDE
Entre tres numeros inteiros
23
41
17
Menor numero:          17
Press RETURN to close window . . . -
```



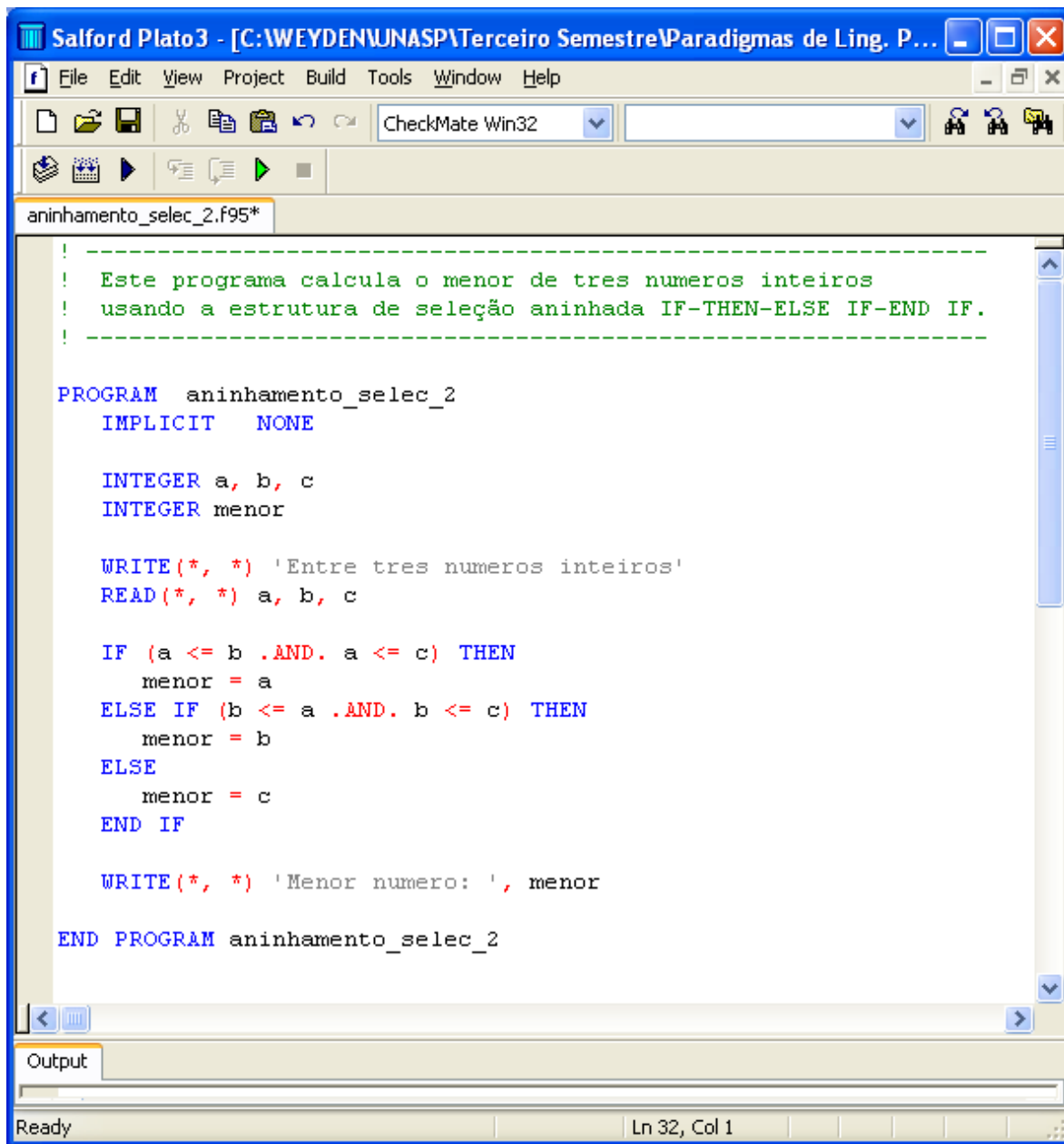
```
Plato3 IDE
Entre tres numeros inteiros
25
19
34
Menor numero:          19
Press RETURN to close window . . . -
```

Como dissemos anteriormente, muitos níveis de aninhamento podem tornar o programa menos legível. A linguagem FORTRAN fornece um meio mais curto de aninhar estruturas IF-THEN-ELSE-END IF. É a versão IF-THEN-ELSE IF-END IF, cuja sintaxe é mostrada abaixo:

```
IF (expressao-logica-1) THEN
    instrucoes-1
ELSE IF (expressao-logica-2) THEN
    instrucoes-2
ELSE IF (expressao-logica-3) THEN
    instrucoes-3
ELSE IF (.....) THEN
    .....
ELSE
    instrucoes
END IF
```

Primeiro a expressão-lógica é avaliada. Se o resultado for `.FALSE.` e houver um `ELSE` em seguida, as instruções do `ELSE` serão executadas. Mas se o resultado da avaliação for `.TRUE.`, ou for `.FALSE.` e não houver um `ELSE` em seguida, FORTRAN executa a instrução depois de `END IF`.

`IF-THEN-ELSE IF-END IF` pode economizar espaço e tornar os programas mais legíveis, mas nem todos os `IF` aninhados podem ser convertidos para `IF-THEN-ELSE IF-END IF`. Veja, o exemplo do programa em que devemos encontrar o menor de três números, ele teve que passar por algumas modificações para se adaptar à estrutura `IF-THEN-ELSE IF-END IF`.



Salford Plato3 - [C:\WEYDEN\UNASP\Terceiro Semestre\Paradigmas de Ling. P...

File Edit View Project Build Tools Window Help

CheckMate Win32

aninhamento_selec_2.f95*

```
! Este programa calcula o menor de tres numeros inteiros
! usando a estrutura de seleção aninhada IF-THEN-ELSE IF-END IF.
!

PROGRAM aninhamento_selec_2
  IMPLICIT NONE

  INTEGER a, b, c
  INTEGER menor

  WRITE (*, *) 'Entre tres numeros inteiros'
  READ (*, *) a, b, c

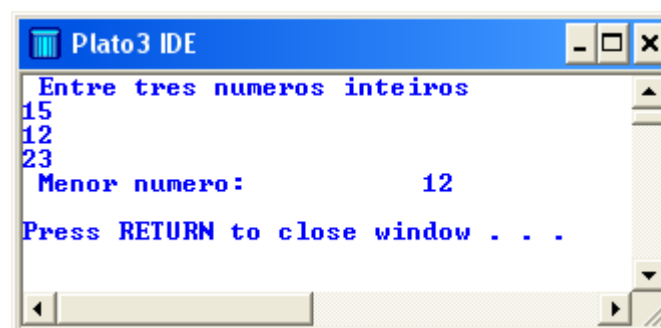
  IF (a <= b .AND. a <= c) THEN
    menor = a
  ELSE IF (b <= a .AND. b <= c) THEN
    menor = b
  ELSE
    menor = c
  END IF

  WRITE (*, *) 'Menor numero: ', menor

END PROGRAM aninhamento_selec_2
```

Output

Ready Ln 32, Col 1



Plato3 IDE

```
Entre tres numeros inteiros
15
12
23
Menor numero:          12
Press RETURN to close window . . .
```

6.5 *SELECT CASE*

Muitas vezes um algoritmo precisa testar uma variável ou expressão para cada um dos valores que ela pode assumir, e executar seqüências de instruções diferentes para cada caso. FORTRAN fornece uma estrutura de **seleção múltipla** para atender essa necessidade, e sua sintaxe é mostrada em seguida.

```
SELECT CASE (seletor)
  CASE (lista de rotulos-1)
    instrucoes-1
  CASE (lista de rotulos-2)
    instrucoes-2
  CASE (lista de rotulos-3)
    instrucoes-3
  .....
  CASE (lista de rotulos-n)
    instrucoes-n
  CASE DEFAULT
    instrucoes-DEFAULT
END SELECT
```

A estrutura `SELECT CASE` consiste de uma série de listas de rótulos `CASE`, um `CASE DEFAULT` opcional, um seletor e os conjuntos de instruções.

O seletor é uma expressão (incluindo variáveis), cujo resultado é do tipo `INTEGER`, `CHARACTER` ou `LOGICAL` (o tipo `REAL` não pode ser usado).

A execução da estrutura `SELECT CASE` começa com a avaliação da expressão do seletor. Se em alguma lista de rótulos `CASE` houver o mesmo valor da expressão, o seu grupo de instruções é executado e a estrutura `SELECT CASE` termina sua execução, indo para `END SELECT`. Mas se não houver o mesmo valor em nenhuma lista de rótulos `CASE`, o programa verifica se há um `CASE DEFAULT`. Se houver, as instruções do `CASE DEFAULT` são executadas e a estrutura `SELECT CASE` termina sua execução, indo para

END SELECT. Mas se não houver um CASE DEFAULT, a estrutura SELECT CASE termina sua execução, indo para END SELECT.

O uso de CASE DEFAULT é opcional, mas o seu uso é recomendado pois assegura que um dos rótulos será executado, independente do valor do seletor.

A lista de rótulos consiste de rótulos (constantes ou alias definidas por PARAMETER) separados por vírgulas (se houver mais de um rótulo na lista), e cada rótulo pode estar em uma das quatro formas aceitas pela linguagem FORTRAN:

```
valor                ! um unico valor

valor1 : valor2      ! todos os valores entre valor1 e valor2, inclusive
                  ! valor1 e valor2

valor1 :              ! todos os valores maiores que ou iguais a valor1

: valor2             ! todos os valores menores que ou iguais a valor2
```

valor1, valor2 e valor3 são constantes ou alias definidas por PARAMETER, e devem ter o mesmo tipo do seletor.

EXEMPLO:

```
! Autor:.....Weyden Daniel Castro
! Data:.....05/2005
! Instituicao:...UNASP
! Curso:.....Ciencia da Computacao
! Disciplina:....Paradigmas de Linguagens de Programacao
! Professor:.....Ausberto Castro

! -----
! Este programa le um caractere e identifica se e uma vogal, uma
! consoante, um digito, um operador aritmetico (+, -, * e /), um
! espaco, ou outro caractere.
! A entrada de caracteres pode ser feita com aspas ou sem aspas.
! -----
```

```

PROGRAM TesteCaracteres

  IMPLICIT NONE

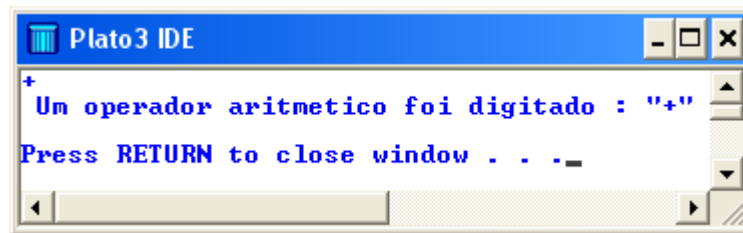
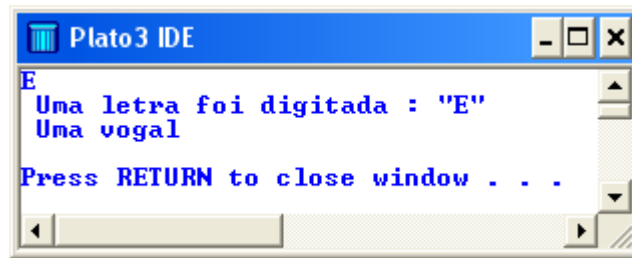
  CHARACTER(LEN=1) :: caractere

  READ(*,*) caractere

  SELECT CASE (caractere)
    CASE ('A' : 'Z', 'a' : 'z')
! todas as letras
      WRITE(*,*) 'Uma letra foi digitada : "', caractere, '"'
      SELECT CASE (caractere)
! as vogais
        CASE ('A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u')
          WRITE(*,*) 'Uma vogal'
        CASE DEFAULT
! as consoantes
          WRITE(*,*) 'Uma consoante'
        END SELECT
      CASE ('0' : '9')
! um digito
        WRITE(*,*) 'Um digito foi digitado : "', caractere, '"'
        CASE ('+', '-', '*', '/')
! um operador
          WRITE(*,*) 'Um operador aritmetico foi digitado : "',
caractere, '"'
          CASE (' ')
! um espaco
            WRITE(*,*) 'Um espaco foi digitado : "', caractere, '"'
            CASE DEFAULT
! outros caracteres
              WRITE(*,*) 'Outro caractere foi digitado : "', caractere, '"'
            END SELECT
          END CASE
        END CASE
      END SELECT

END PROGRAM TesteCaracteres

```



7 Repetição

7.1 Loop-DO

O loop DO é uma estrutura de repetição muito simples. Com ele podemos executar repetidamente um conjunto de instruções, inserindo o código no programa apenas uma vez. A estrutura DO tem a seguinte sintaxe:

```
DO
    instrucoes
END DO
```

É necessário tomar cuidado para não gerar um loop infinito, se nenhum mecanismo de saída for utilizado, as instruções do loop serão executadas continuamente sem parar.

Por isso, FORTRAN fornece o comando EXIT. Esse comando é utilizado para sair de um loop, quando EXIT é executado o programa sai do loop-DO mais interno (no caso de

estruturas aninhadas), em que EXIT se encontra inserido. Geralmente, EXIT é utilizado dentro de uma estrutura de seleção para ser executado quando alguma condição for satisfeita.

```
DO
    instrucoes
    IF(expressão-logica) EXIT
END DO
```

Veja o seguinte exemplo de um programa que verifica se um número é primo ou não, usando o comando EXIT para sair do loop-DO quando a condição for satisfeita.

```
! -----
! Este programa verifica se um numero inteiro, dado pelo usuario,
! e um numero primo.

! 1) Primeiro, o programa verifica se o numero e menor que 2, se for,
!     o numero e considerado invalido.

! 2) Depois, o programasma verifica se o numero inserido e 2. Se for,
!     o numero e primo.

! 3) Em seguida, o programa verifica se o numero e par, se for, o numero
!     e descartado, pois, nenhum par e primo.

! 4) Finalmente, como restam apenas numeros impares, o programa utiliza
!     um divisor impar, incrementado de 2 a cada iteracao, para verificar
!     se o numero e primo ou nao. Dois resultados podem acontecer:
!     a) Se um dos divisores, conseguir dividir o numero, o numero
!         nao e primo.
!     b) Se o divisor e maior que a raiz quadrada do numero, o numero
!         e primo.
! -----

PROGRAM numero_primo
    IMPLICIT NONE

    INTEGER :: Numero
    INTEGER :: Divisor
```

```

WRITE(*, *)
WRITE(*, *) ' Checa se um numero inteiro positivo e um numero PRIMO'
WRITE(*, *) '===== '

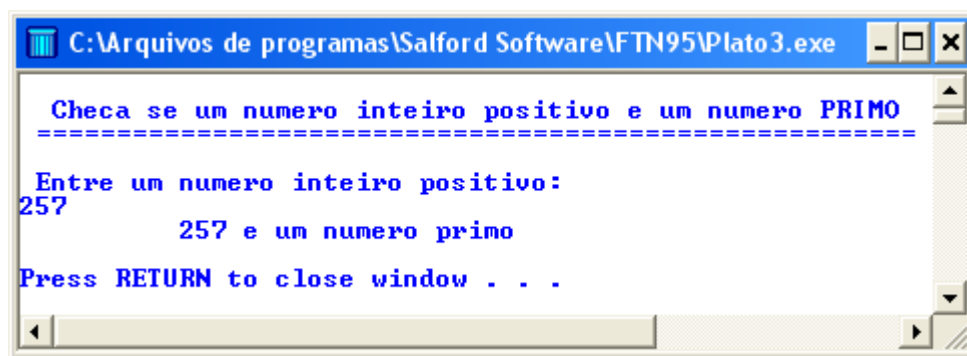
WRITE(*, *)
WRITE(*, *) 'Entre um numero inteiro positivo:'

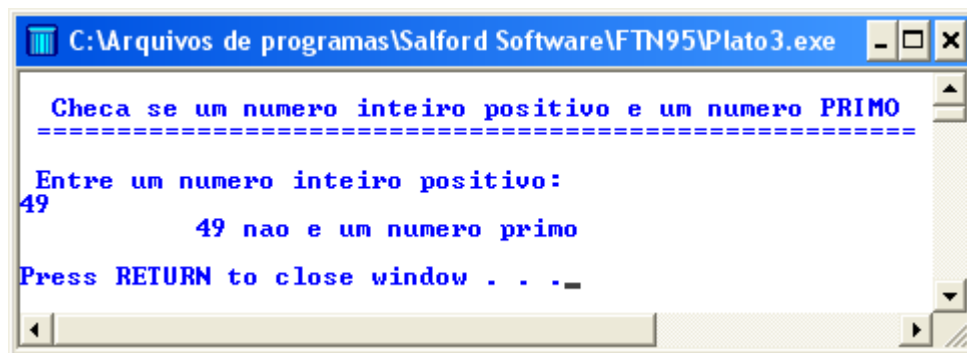
READ(*,*)  Numero                ! le um numero do usuario
IF (Numero < 2) THEN              ! se for < 2, o numero nao e primo
    WRITE(*,*) 'Entrada Invalida'
ELSE IF (Numero == 2) THEN       ! se for = 2, o numero e primo
    WRITE(*,*)  Numero, ' e um numero primo'
ELSE IF (MOD(Numero,2) == 0) THEN ! se for par, o numero nao e primo
    WRITE(*,*)  Numero, ' nao e um numero primo'
ELSE                            ! se o programa chegou ate aqui, restam apenas numero impares >= 3
    Divisor = 3                  ! divisor comeca com 3

DO
    IF (Divisor*Divisor > Numero .OR. MOD(Numero, Divisor) == 0) EXIT
    Divisor = Divisor + 2        ! incrementa o divisor para o proximo impar
END DO

! verifica qual das duas condicoes acima, foi satisfeita.
IF (Divisor*Divisor > Numero) THEN
    WRITE(*,*)  Numero, ' e um numero primo'
ELSE
    WRITE(*,*)  Numero, ' nao e um numero primo'
END IF
END IF
END PROGRAM  numero_primo

```





7.2 Loop-DO com variável de controle

FORTTRAN fornece duas estruturas de repetição: “loop-DO” e “loop-DO com variável de controle”. Veja abaixo a sintaxe de loop-DO com variável de controle:

```
DO variavel_controle = valor_inicial, valor_final, [tamanho_passo]
    instruções
END DO
```

Onde temos que, `variavel_controle` é uma variável do tipo `INTEGER`, `valor_inicial` e `valor_final` são duas expressões do tipo `INTEGER`, e `tamanho_passo` também é uma expressão do tipo `INTEGER` cujo valor não pode ser zero. Note que o uso de `tamanho_passo` é opcional (ele está entre chaves), se ele for omitido, FORTRAN assumirá 1 como padrão para `tamanho_passo`. Quanto às instruções, são um conjunto de comandos, geralmente chamado de corpo do loop-DO. No corpo do loop-DO, podem ser colocadas qualquer instrução executável, incluindo estruturas de seleção e repetição.

Antes da execução começar, os valores de `valor_inicial`, `valor_final` e `tamanho_passo` (deve ser diferente de zero) são computados. A variável `controle` recebe o valor de `valor_inicial`.

Se o valor de `tamanho_passo` é positivo, `variavel_controle` será incrementada a cada iteração, ou seja o seu valor vai aumentar. O programa então verifica se o valor da

variavel_controle é menor que ou igual ao valor de valor_final, se for, as instruções são executadas. Em seguida, o valor de tamanho_passo é adicionado ao valor de variavel_controle, e a iteração é executada de novo. Quando o valor de variavel_controle se tornar maior que o valor de valor_final, o loop-DO com variável de controle termina, e o programa vai para END DO.

Se o valor de tamanho_passo é negativo, variavel_controle é decrementada a cada iteração, ou seja o seu valor vai diminuir. O programa então verifica se o valor da variavel_controle é maior que ou igual ao valor de valor_final, se for, as instruções são executadas. Em seguida, o valor de tamanho_passo é adicionado ao valor de variavel_controle, e a iteração é executada de novo. Quando o valor da variavel_controle se tornar menor que o valor de valor_final, o loop-DO com variável de controle termina, e o programa vai para END DO.

Embora seja possível usar o tipo REAL para variavel_controle, valor_inicial, valor_final e tamanho_passo, isso não é recomendado pois é provável que essa possibilidade seja eliminada em uma futura padronização da linguagem FORTRAN. Para usar valores do tipo REAL, utilize a forma geral do loop-DO.

Veja a seguir um programa que calcula o fatorial de um número inteiro e positivo utilizando a estrutura de repetição loop-DO com variável de controle.

```
! -----
! Este programa calcula o fatorial de um numero inteiro
! nao negativo dado pelo usuario.
! O fatorial de N e igual a:

! N! = 1 * 2 * 3 * ... * ( N - 1 ) * N

! e por definicao 0! = 1.
! -----

PROGRAM  fatorial
      IMPLICIT  NONE
```

```

INTEGER :: N, i, fat

WRITE(*,*) 'Este programa calcula o fatorial de'
WRITE(*,*) 'um inteiro nao negativo'
WRITE(*,*)
WRITE(*,*) 'Digite o valor de N --> '
READ(*,*) N
WRITE(*,*)

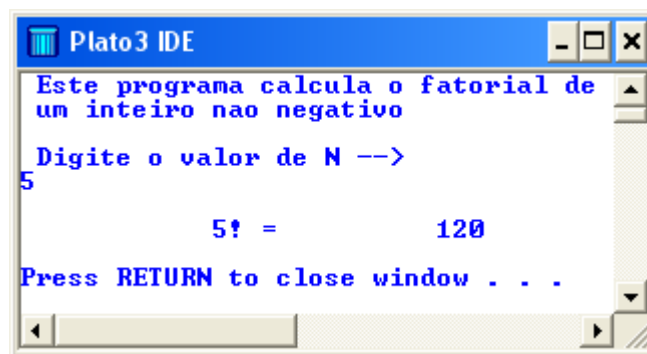
IF (N < 0) THEN                                ! o programa nao aceita N < 0
    WRITE(*,*) 'ERRO: N nao pode ser um numero negativo'
    WRITE(*,*) 'Sua Entrada N = ', N
ELSE IF (N == 0) THEN                          ! 0! = 1
    WRITE(*,*) '0! = 1'
ELSE                                           ! N > 0
    fat = 1

    DO i = 1, N
        fat = fat * i
    END DO

    WRITE(*,*) N, '! = ', fat
END IF

END PROGRAM fatorial

```



Além do comando EXIT, FORTRAN também tem o comando CYCLE, usado para pular para a próxima iteração do loop-DO. Veja abaixo a sua sintaxe:

```

DO
    instrucoes-1
    CYCLE
    instrucoes-2
END DO

```

CYCLE pode ser usado com os dois tipos de loops_DO (“loop-DO” e “loop-DO com variável de controle”). É importante lembrar que se CYCLE for usado com um “loop-DO com variável de controle”, devemos acrescentar as informações de controle à sintaxe acima.

A seguir há um exemplo de um programa que lê todas as letras do alfabeto a partir da tabela ASCII e mostra apenas as vogais.

```

! -----
! Este programa mostra apenas as letras do alfabeto que sao vogais,
! desprezando as consoantes.
! -----

PROGRAM vogais
    IMPLICIT NONE

    INTEGER :: i

    DO i = 65, 90      ! o alfabeto inteiro em maiusculas, pela tabela ASCII

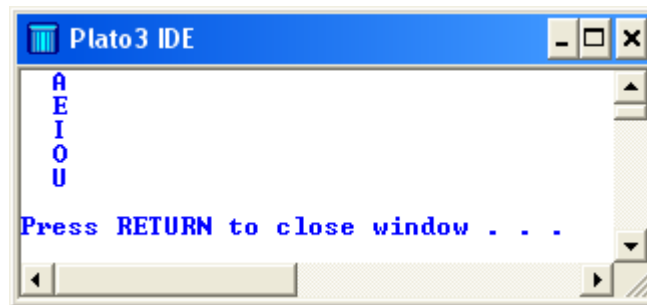
        IF (CHAR(i) /= 'A' .AND. CHAR(i) /= 'E' .AND. CHAR(i) /= 'I' .AND.    &
            CHAR(i) /= 'O' .AND. CHAR(i) /= 'U') CYCLE

        WRITE(*, *) ' ', CHAR(i)

    END DO

END PROGRAM vogais

```



7.3 Estruturas de repetição aninhadas

Um loop-DO pode conter outros loops-DO em seu corpo (conjunto de instruções). Nesse caso, o comando EXIT faz com que o programa saia apenas do loop_DO no qual o comando está inserido, ou seja, se o loop-DO que contém o comando EXIT estiver dentro de outro loop-DO, o controle do programa passa para o loop-DO mais externo.

Veja um exemplo da sintaxe de dois loops-DO aninhados (mais loops podem ser aninhados):

```
DO
    instrucoes-1

    DO
        instrucoes-2
    END DO

    instrucoes-3
END DO
```

A seguir, um exemplo de um programa que mostra todos os números primos entre 2 e N, usando estruturas de repetição aninhadas (para mais detalhes sobre o algoritmo, ver o exemplo de números primos na seção 6.1):

```
! Autor:.....Weyden Daniel Castro
! Data:.....05/2005
```

```

!  Instituicao:...UNASP
!  Curso:.....Ciencia da Computacao
!  Disciplina:....Paradigmas de Linguagens de Programacao
!  Professor:.....Ausberto Castro

! -----
! Este programa mostra todos os numeros primos na faixa de 2 a N,
! sendo N um numero inteiro fornecido pelo usuario.
! -----

PROGRAM  Primos_2_N
  IMPLICIT  NONE

  INTEGER  :: faixa, numero, divisor, cont

  WRITE(*,*)  'Qual e a faixa desejada [ >= 2 ]? '
  DO
    ! executa ate uma entrada correta ser inserida
    READ(*,*)  faixa
    IF (faixa >= 2) EXIT
    ! sai do loop se a entrada for correta
    WRITE(*,*)  'A faixa deve ser >= 2.  Voce colocou = ', faixa
    WRITE(*,*)  'Por favor, tente de novo:'
  END DO

  cont = 1
    ! 2 e o primeiro numero primo da lista
  WRITE(*,*)
  WRITE(*,*)  'Numero primo ', cont, ' ) ', 2
  DO numero = 3, faixa, 2
    ! testa todos os numeros impares entre 2 e numero

    divisor = 3
      ! divisor comeca com 3
    DO
      IF (divisor*divisor > numero .OR. MOD(numero,divisor) == 0) EXIT
      divisor = divisor + 2
      ! proximo impar
    END DO

    IF (divisor*divisor > numero) THEN
      ! nenhum divisor conseguiu dividir o
      ! numero
      cont = cont + 1
      ! O numero e primo
      WRITE(*,*)  'Numero primo ', cont, ' ) ', numero
    END IF
  END DO

  WRITE(*,*)

```



```

WRITE(*,*) 'Existem ', cont, ' primos entre 2 e ', faixa

END PROGRAM Primos_2_N

```

```

Qual e a faixa desejada [ >= 2 ]?
100

Numero primo      1>          2
Numero primo      2>          3
Numero primo      3>          5
Numero primo      4>          7
Numero primo      5>         11
Numero primo      6>         13
Numero primo      7>         17
Numero primo      8>         19
Numero primo      9>         23
Numero primo     10>         29
Numero primo     11>         31
Numero primo     12>         37
Numero primo     13>         41
Numero primo     14>         43
Numero primo     15>         47
Numero primo     16>         53
Numero primo     17>         59
Numero primo     18>         61
Numero primo     19>         67
Numero primo     20>         71
Numero primo     21>         73
Numero primo     22>         79
Numero primo     23>         83
Numero primo     24>         89
Numero primo     25>         97

Existem          25 primos entre 2 e          100

Press RETURN to close window . . .

```

8 Entrada e saída formatada

8.1 Os formatos da linguagem FORTRAN

Nós temos utilizado até agora entrada e saída em lista, o qual é um tipo de entrada e

saída de dados muito fácil de usar. No entanto, não temos controle sobre a aparência da entrada e da saída de dados. Para resolver esse problema nós devemos usar os **formatos** da linguagem FORTRAN.

Um formato consiste de um par de parênteses que contém **descritores de edição de formatos** separados por vírgulas:

```
(... descritor_N, descritor_N+1, descritor_N+2, ...)
```

Os descritores de edição de formatos dizem ao sistema como manipular a entrada e saída de variáveis e expressões. Passam informações importantes, como o número de posições a ser usado.

Daqui em diante, serão utilizados os seguintes símbolos para os descritores:

- **w**: o número de posições a ser usado.
- **m**: o número mínimo de posições a ser usado.
- **d**: o número de dígitos à direita do ponto decimal.
- **e**: o número de dígitos na parte do expoente.

Embora nós possamos imprimir um número com quantas posições desejarmos, isso não afeta em nada a precisão do número (a quantidade de dígitos que o computador é capaz de armazenar), é apenas para questões de entrada e saída.

A seguinte tabela mostra os descritores de edição de formatos da linguagem FORTRAN. Eles serão estudados com mais detalhes nas seções seguintes.

Propósito		Descritores	
Ler/escrever INTEGERs		Iw	Iw.m
Ler/escrever REALs	Forma decimal	Fw.d	
	Forma exponencial	Ew.d	Ew.dEe
	Forma científica	ESw.d	ESw.dEe
	Forma de engenharia	ENw.d	ENw.dEe
Ler/escrever LOGICALs		Lw	

Ler/escrever CHARACTERs		A	Aw
Posicionamento	Horizontal	nX	
	Tabulação	Tc	TLc e TRc
	Vertical	/	
Outros	Agrupamento	r(...)	
	Controle de avaliação de formatos	:	
	Controle de sinal	S, SP e SS	
	Controle de espaços em branco	BN e BZ	

FORTTRAN aceita três maneiras diferentes de utilizar um formato. Um deles é o comando **FORMAT**, mas ele não será usado aqui, pois os outros dois métodos são bastante eficientes e atendem as necessidades do programador.

- Escrever o formato como uma string de caracteres e usá-lo no lugar do segundo asterisco dos comandos **READ(*, *)** e **WRITE(*, *)**.

```

READ(*, '(A, 3I10)')      ... variaveis ...
READ(*, "(F12.3)")        ... variaveis ...

WRITE(*, '(F10.3, I4)')   ... variaveis e expressoes ...
WRITE(*, "(3F7.2)")       ... variaveis e expressoes ...

```

- Como um formato é também uma string de caracteres, nós podemos declarar um **PARAMETER** do tipo **CHARACTER** para armazenar um formato.

```

CHARACTER(LEN=30), PARAMETER :: FMT1 = "(3I4, F8.2, A5)"
CHARACTER(LEN=*) , PARAMETER  :: FMT2 = "(4F12.3 E10.4)"

READ(*, FMT1)  ... variaveis ...
READ(*, FMT1)  ... variaveis ...

WRITE(*, FMT2) ... variaveis e expressoes ...
WRITE(*, FMT2) ... variaveis e expressoes ...

```

Nós também podemos usar uma variável do tipo **CHARACTER** para armazenar um formato.

```

CHARACTER(LEN=80) :: fmt

fmt = "(A6, 8I4)"

READ(*, fmt)  ... variaveis ...

WRITE(*, fmt) ... variaveis e expressoes ...

```

8.2 Saída de dados

8.2.1 INTEGER: descritor I

Para a saída de dados do tipo INTEGER, FORTRAN possui os descritores **Iw** e **Iw.m**. A seguir, a forma geral desses descritores:

rIw e rIw.m

Significados dos símbolos:

- **I** = INTEGER.
- **w** = largura do campo. Ou seja, o número de posições em que um número inteiro pode ser imprimido.
- **m** = número mínimo de posições (de um total de w posições) que devem conter dígitos. Se o número a ser imprimido tiver menos que m dígitos, as posições restantes são preenchidas com 0s. Se o número a ser imprimido tiver mais que m dígitos, m é ignorado. Observe que w deve ser positivo e maior que ou igual a m (w pode ser zero).
- **r** = indicador de repetição. Indica o número de vezes que o descritor deve ser repetido. Por exemplo, 3I6 é equivalente a I6, I6, I6.

OBSERVAÇÕES:

- O sinal de um número também necessita de uma posição, mas apenas sinais negativos são imprimidos.
- Se o número de posições é menor que o número de dígitos mais o sinal, todas as posições w são preenchidas com asteriscos.

A seguinte tabela mostra os resultados de diferentes formatos aplicados ao comando WRITE. Considere a = 12, b = -12 e c = 12345.

Comando WRITE		Resultado				
WRITE(*, "(I4)")	a				1	2
WRITE(*, "(I4.2)")	a				1	2
WRITE(*, "(I4.3)")	a			0	1	2
WRITE(*, "(I4.4)")	a		0	0	1	2
WRITE(*, "(I4)")	b			-	1	2
WRITE(*, "(I4.2)")	b			-	1	2
WRITE(*, "(I4.3)")	b		-	0	1	2
WRITE(*, "(I4.4)")	b	-	0	0	1	2
WRITE(*, "(I4)")	c	*	*	*	*	*

Podemos ter mais de uma variável INTEGER por comando WRITE:

```
INTEGER :: a = 4, b = 56  
WRITE(*, "(I4, I5.2)") a, b
```

Podemos ainda, ter mais de uma variável INTEGER para o mesmo descritor de edição de formato, nesse caso é útil o indicador de repetição, pois dispensa a repetição do descritor, basta indicar o número de vezes que queremos repeti-lo. Considere o seguinte exemplo:

```
INTEGER :: a = 4, b = 56, c = 176  
WRITE(*, "(2I4, I5.2)") a, b, c
```

Os descritores são 2I4 e I5.2, e o formato é equivalente a (I4, I4, I5.2). O comando WRITE acima, produz o seguinte resultado:

			4			5	6			1	7	6
--	--	--	---	--	--	---	---	--	--	---	---	---

8.2.2 REAL: descritor F

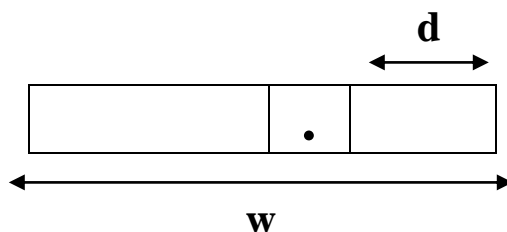
Para a saída de dados do tipo REAL, FORTRAN possui o descritor **Fw.d**. A seguir, a forma geral desse descritor:

rFw.d

Significados dos símbolos:

- **F** = REAL.
- **w** = largura do campo. Ou seja, o número de posições em que um número real pode ser imprimido.
- **d** = número de dígitos depois do ponto decimal. Ou seja, de um total de w posições, as d posições mais à direita são para a parte fracionária de um número real, e a

posição $d + 1$ a partir da direita é um ponto decimal. As $w-(d+1)$ posições restantes são para a parte inteira. Isso é mostrado na figura abaixo:



- **r** = indicador de repetição. Indica o número de vezes que o descritor deve ser repetido. Por exemplo, 3F5.2 é equivalente a F5.2, F5.2, F5.2.

OBSERVAÇÕES:

- Quando um número real é imprimido, sua parte inteira pode usar apenas $w - (d+1)$ posições. Se o número de posições não for suficiente para imprimir a parte inteira totalmente, todas as posições w serão preenchidas com asteriscos.
- Se a parte inteira contém um sinal, w deve ser maior que ou igual a $d + 2$.
- Geralmente, a parte fracionária tem mais do que d dígitos, nesse caso, o dígito $d + 1$ será arredondado para o dígito d . Por exemplo, se imprimirmos 4.57 com F3.1, teremos como resultado, 4.6, pois 7 é arredondado.
- Se a parte fracionária for menor que d dígitos, 0s serão adicionados no final.
- d pode ser zero, nesse caso a posição mais à direita será o ponto decimal.

A seguinte tabela mostra os resultados de diferentes formatos aplicados ao comando WRITE. Considere $a = 123.456$ e $b = -12.34$.

Comando WRITE	Resultado						
WRITE(*, "(F8.0)") a				1	2	3	.
WRITE(*, "(F8.1)") a				1	2	3	. 5
WRITE(*, "(F8.2)") a			1	2	3	.	4 6
WRITE(*, "(F8.3)") a		1	2	3	.	4	5 6
WRITE(*, "(F8.4)") a	1	2	3	.	4	5	6 0
WRITE(*, "(F8.5)") a	*	*	*	*	*	*	*
WRITE(*, "(F8.0)") b				-	1	2	.
WRITE(*, "(F8.1)") b				-	1	2	. 3
WRITE(*, "(F8.2)") b			-	1	2	.	3 4
WRITE(*, "(F8.3)") b		-	1	2	.	3	4 0

WRITE(*, "(F8.4)") b	-	1	2	.	3	4	0	0
WRITE(*, "(F8.5)") b	*	*	*	*	*	*	*	*

Podemos ter mais de uma variável REAL por comando WRITE:

```
REAL :: a = 36.0, b = -12.45
WRITE(*, "(F8.3, F7.2)") a, b
```

Podemos ainda, ter mais de uma variável REAL para o mesmo descritor de edição de formato, nesse caso é útil o indicador de repetição, pois dispensa a repetição do descritor, basta indicar o número de vezes que queremos repeti-lo. Considere o seguinte exemplo:

```
REAL :: a = 36.0, b = -12.45, c = 4.26
WRITE(*, "(2F8.3, F7.2)") a, b, c
```

Os descritores são 2F8.3 e F7.2, e o formato é equivalente a (F8.3, F8.3, F7.2). O comando WRITE acima, produz o seguinte resultado:

		3	6	.	0	0	0		-	1	2	.	4	5	0			4	.	2	6
--	--	---	---	---	---	---	---	--	---	---	---	---	---	---	---	--	--	---	---	---	---

8.2.3 REAL: descritor E

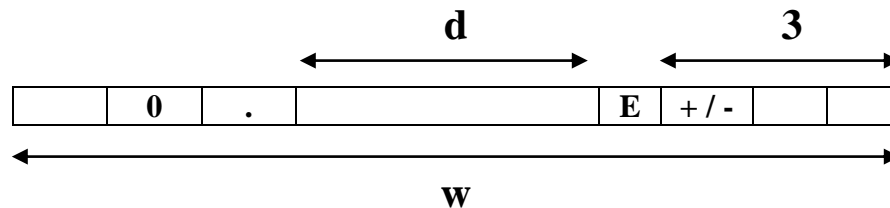
Para saída de dados do tipo REAL na forma exponencial, FORTRAN possui os descritores **Ew.d** e **Ew.dEe**. A seguir, a forma geral desses descritores:

rEw.d e **rEw.dEe**

Significados dos símbolos:

- **E** = números do tipo REAL na forma exponencial.
- **w** = largura do campo. Ou seja, o número de posições em que um número real, na forma exponencial, pode ser imprimido.
- **d** = número em forma normalizada, ou seja, com exceção do zero antes do ponto, todos os dígitos do número se encontram depois do ponto decimal.

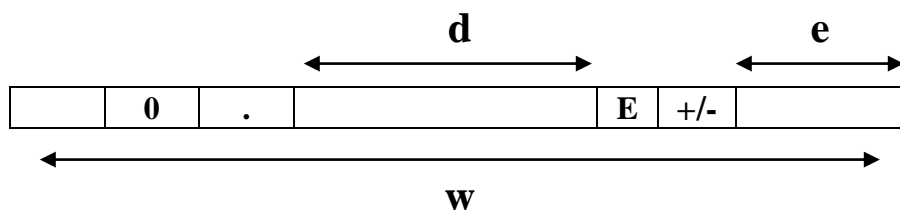
- **r** = indicador de repetição. Indica o número de vezes que o descritor deve ser repetido. Por exemplo 4E15.7E4 é equivalente a E15.7E4, E15.7E4, E15.7E4, E15.7E4.
- O descritor **Ew.d** gera números da seguinte forma:



Antes de um número REAL ser imprimido em forma exponencial, ele é convertido para a forma normalizada: $s0.XXX...XXX \cdot 10^{xxxx}$, onde *s* é o sinal (+ ou -) do número e do expoente e *x* representa um dígito. Por exemplo 134.5231 é convertido para $0.1245231 \cdot 10^3$.

O descritor **Ew.d**, como podemos ver na figura acima, reserva as três últimas posições para o expoente e seu sinal, além de uma posição para o descritor **E**, e posições para o dígito 0, para o ponto decimal e para o sinal do número (se houver). Sendo assim é fortemente recomendado ter um número *w* com no mínimo *d* + 7 posições, lembrando que se não houverem posições suficientes para o número ser impresso, todas as posições *w* serão preenchidas com asteriscos.

- O descritor **Ew.dEe** gera números da seguinte forma:



A diferença deste descritor para o descritor anterior é que neste descritor, o número de posições do expoente é o número de *e*. Com o descritor **Ew.d**, não era possível ter expoentes maiores que 99 ou menores que -99 devido ao espaço limitado de duas posições mais o sinal.

Como esta forma reserva além das posições de *d* e de *e*, uma posição para o caractere **E**, uma posição para o sinal do expoente, um ponto decimal, um zero e o sinal do número (se houver), é fortemente recomendado que o *w* seja maior ou igual a *d* + *e* + 5.

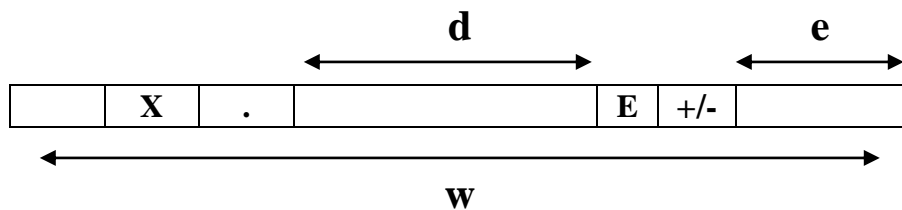
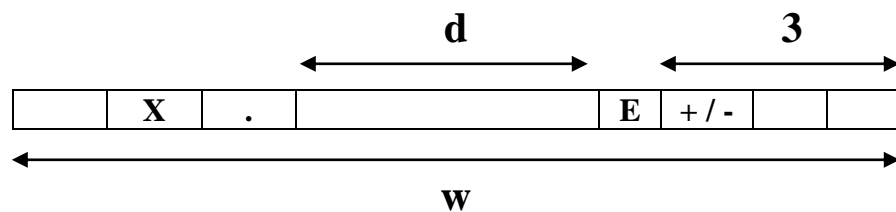
A seguinte tabela mostra os resultados de diferentes formatos aplicados ao comando **WRITE**. Considere $\pi = 3.1415926$.

Comando WRITE		Resultado										
WRITE(*,“(E12.4)”) PI			0	.	3	1	4	2	E	+	0	1
WRITE(*,“(E12.3E4)”) PI		0	.	3	1	4	E	+	0	0	0	1
WRITE(*,“(E12.5E1)”) PI			0	.	3	1	4	1	6	E	+	1
WRITE(*,“(E12.5E2)”) PI		0	.	3	1	4	1	6	E	+	0	1
WRITE(*,“(E12.2E1)”) PI						0	.	3	1	E	+	1

Cientistas e engenheiros têm uma maneira um pouco diferente de usar números na forma exponencial, para cada um desses dois grupos, FORTRAN têm um descritor de edição de formato, os quais serão apresentados a seguir:

Para a saída de dados na forma exponencial, de acordo com a forma usada pelos cientistas, FORTRAN possui o descritor **ES** que imprime número reais na **forma científica**. Existem duas variâncias do descritor ES, os descritores **ESw.d** e **ESw.dEe**.

A forma científica consiste de um dígito diferente de zero na parte inteira do número na forma exponencial. Se esse número for zero, todos os dígitos imprimidos nas posições w serão zeros. As figuras abaixo, mostram a forma científica utilizada com os seus dois descritores:

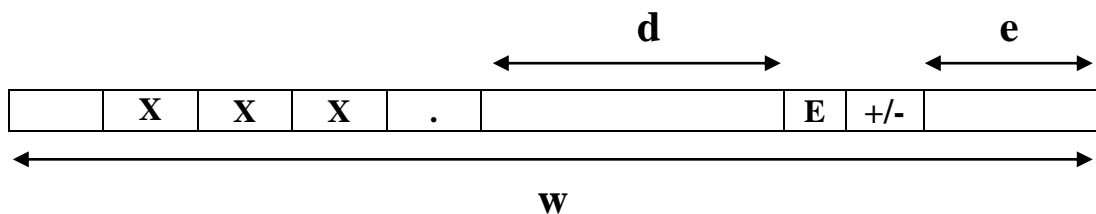
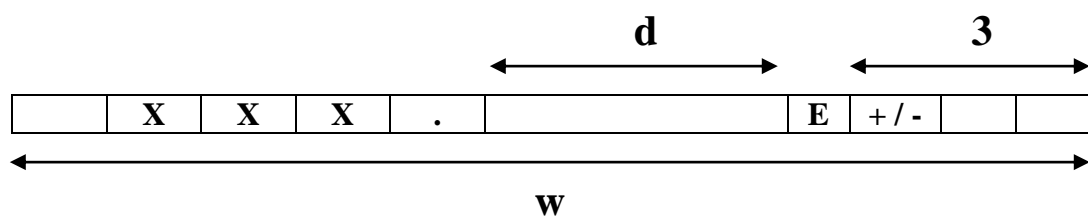


Por exemplo, a conversão do número 127.3156 para a forma científica, utilizando o formato ES10.3 produz a seguinte saída:

	1	.	2	7	3	E	+	0	2
--	---	---	---	---	---	---	---	---	---

Para a saída de dados na forma exponencial, de acordo com a forma usada pelos engenheiros, FORTRAN possui o descritor **EN**. Existem duas variâncias do descritor EN, os descritores **ENw.d** e **ENw.dEe**.

A forma de engenharia consiste de um a três dígitos diferentes de zero na parte inteira do número, e um expoente múltiplo de três. As figuras abaixo, mostram a forma de engenharia utilizada com os seus dois descritores:



Por exemplo, a conversão do número 0.0000528597 para a forma de engenharia, requer um expoente múltiplo de 3, e uma parte inteira com um a três dígitos diferentes de zero. Começamos deslocando o ponto decimal 3 (para obter um expoente múltiplo de 3) casas para a direita, obtendo assim $000.0528597 \cdot 10^{-3}$. Ainda temos 0s na parte inteira, por isso vamos deslocar o ponto decimal mais 3 casas para a direita, obtendo assim: $052.8597 \cdot 10^{-6}$ ou $52.8597 \cdot 10^{-6}$. Agora o número está em forma de engenharia, e pode ser impresso com um dos descritores EN.

Vejamos como seria impresso o número 0.0000528597, usando o formato EN14.5:

		5	2	.	8	5	9	7	E	-	0	6
--	--	---	---	---	---	---	---	---	---	---	---	---

8.2.4 LOGICAL: descritor L

Para a saída de dados do tipo LOGICAL, FORTRAN possui o descritor **Lw**. A seguir, a forma geral desse descritor:

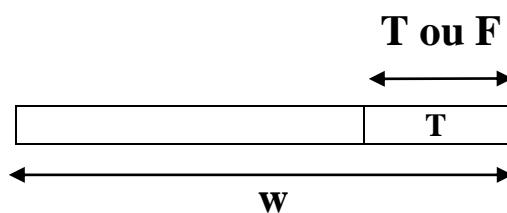
rLw

FORTRAN usa `.TRUE.` para indicar o valor lógico verdadeiro, e `.FALSE.` para indicar o valor lógico falso. No entanto, para imprimir um valor lógico, FORTRAN usa T e F.

Significados dos símbolos:

- **L** = LOGICAL.
- **w** = largura do campo. Ou seja, o número de posições em que um valor lógico pode ser imprimido.
- **r** = indicador de repetição. Indica o número de vezes que o descritor deve ser repetido. Por exemplo, 2L4 é equivalente a L4, L4.

A saída de um valor lógico consiste de apenas um caractere (T ou F) localizado na posição mais à direita de w. As posições restantes são preenchidas com espaços, como mostrado na figura abaixo.



A seguinte tabela mostra os resultados de diferentes formatos aplicados ao comando WRITE. Considere a = `.TRUE.`, b = `.FALSE.` e c = `.TRUE.`.

Comando WRITE	Resultado						
WRITE(*, "(L3, L4)") a, b			T				F
WRITE(*, "(3L2)") a, b, c		T		F		T	

8.2.5 CHARACTER: descritor A

Para a saída de dados do tipo CHARACTER, FORTRAN possui os descritores **A** e **Aw**. A seguir, a forma geral desses descritores:

rA e rAw

Significados dos símbolos:

- **A** = CHARACTER
- **w** = largura do campo. Ou seja, o número de posições em que uma string de caracteres pode ser imprimida.
- **r** = indicador de repetição. Indica o número de vezes que o descritor deve ser repetido. Por exemplo, 4A8 é equivalente a A8, A8, A8, A8.

Para imprimir dados do tipo CHARACTER corretamente, é preciso observar algumas regras:

- Se **w** é maior que o comprimento da string de caracteres, todos os caracteres da string podem ser imprimidos, alinhados a direita. Os espaços que sobram são preenchidos com espaços.
- Se **w** é menor que o comprimento da string de caracteres, apenas os **w** caracteres mais a esquerda da string são imprimidos.
- Se **w** é igual ao comprimento da string de caracteres, todos os caracteres podem ser imprimidos sem nenhum problema.
- Se **w** estiver faltando, ou seja, for utilizado o descritor **A** sozinho, assume-se que o valor de **w** é o comprimento da string.

OBSERVAÇÃO:

O comprimento da string pode conter espaços, os quais serão imprimidos junto com a string se ela for impressa.

A seguinte tabela mostra os resultados de diferentes formatos aplicados ao comando WRITE.

CHARACTER(LEN = 5) :: a = "UNASP"

Comando WRITE	Resultado
WRITE(*, "(A1)") a	U
WRITE(*, "(A3)") a	U N A
WRITE(*, "(A5)") a	U N A S P
WRITE(*, "(A7)") a	U N A S P
WRITE(*, "(A)") a	U N A S P

8.3 Entrada de dados

Como os descritores já foram apresentados na seção 7.2, não tornaremos a explicá-los.

8.3.1 INTEGER: descritor I

Embora possam ser usados os dois descritores da saída de números inteiros: **Iw** e **Iw.m**, eles produzem resultados idênticos, por isso utilizaremos apenas o descritor **Iw**, cuja forma geral é:

rIw

Exemplo:

```
INTEGER :: a, b, c, d  
READ(*, "(I3, I5, I7, I10)") a, b, c, d
```

8.3.2 REAL: descritores F e E

Embora possam ser usados os descritores **Fw.d**, **Ew.d**, **ESw.d**, **ENw.d**, **Ew.dEe**, **ESw.dEe** e **ENw.dEe** para a entrada de números do tipo **REAL**, todos eles produzem resultados idênticos. Assim, nós usaremos apenas o descritor **Fw.d**, que tem a seguinte forma geral:

rFw.d

Exemplo:

```
REAL :: a, b, c  
READ(*, "(F7.0, F3.1, F7.4)") a, b, c
```

8.3.3 LOGICAL: descritor L

A entrada de dados do tipo LOGICAL é feita com o descritor **Lw**, o qual tem a seguinte forma geral:

rLw

Dentro das posições w, a entrada pode ser feita com certa liberdade, desde que seguindo as seguintes regras:

- A entrada pode começar com qualquer número de espaços (dentro de w).
- Depois, podemos ter um ponto opcional, fica a critério do usuário (que pode desejar inserir `.TRUE.` por exemplo).
- A seguir, deve vir **T** ou **F**, representando o valor lógico a ser inserido (verdadeiro ou falso).
- Depois disso, pode vir “qualquer coisa” (respeitando o limite de posições w, o usuário pode colocar o que desejar).

Mas por que o usuário colocaria outras coisas além de T e F?

Para ter uma resposta mais significativa. O programa pode, por exemplo, sugerir que o usuário digite `TRUE` ou `.TRUE.` para verdadeiro, e `FALSE` ou `.FALSE.` para falso.

Mas, não são apenas palavras significativas que são aceitas, FORTRAN considera palavras como: `Tampa`, `TUDO`, `Ta bom`, `Tampinha`,...., como entradas válidas para o valor lógico `.TRUE.`, e palavras como: `FALTA`, `.Feio`, `Fato`, `FIRMEZA`, como entradas válidas para o valor lógico `.FALSE.`.

Como exemplo de entradas incorretas, podemos ter: `Não FALSO`, `..FALSO` (c/ 2 pontos), e qualquer outra que não siga as regras apresentadas acima.

8.3.4 CHARACTER: descritor A

Para a entrada de dados do tipo CHARACTER, são usados os descritores **Aw** e **A**, que têm a seguinte forma geral:

rAw e rA

Devemos estar atentos aos valores de *w* e do comprimento da variável do tipo CHARACTER que receberá a entrada do usuário.

- *w* < comprimento da variável CHARACTER: Há espaço suficiente na variável para armazenar os caracteres lidos, e espaços são adicionados na variável para preencher os demais espaços.
- *w* = comprimento da variável CHARACTER. Temos exatamente o espaço necessário na variável para armazenar os caracteres lidos, todos os caracteres são armazenados sem problemas.
- *w* > comprimento da variável CHARACTER. Não há espaço suficiente na variável para armazenar todos os caracteres lidos. São armazenados os caracteres mais a direita nas posições *w* lidas, de acordo com o comprimento da variável.

8.4 Controle de posição horizontal

8.4.1 Espaçamento horizontal: nX

O descritor X é usado para pular *n* posições horizontais. A sua forma geral é a seguinte:

nX

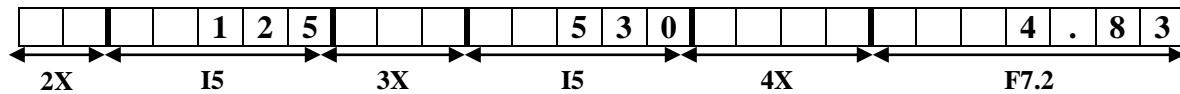
Quando esse descritor é usado, o programa pula as próximas *n* posições. Esse descritor pode ser usado tanto para entrada quanto para saída de dados.

Não confunda o *n*, com o indicador de repetição. O descritor X, diferente dos outros descritores (I, F, L e A), tem o seu número de posições antes do descritor.

Veja o seguinte exemplo:

```
CHARACTER (LEN=30)  :: FMT = " (2X, I5, 3X, I5, 4X, F7.2) "  
INTEGER              :: a = 125  
INTEGER              :: b = 530  
REAL                 :: c = 4.8267
```

```
READ (*, FMT)  a, b, c
```



8.4.2 Tabulação: Tc, TLc e TRc

Os descritores de edição **Tc**, **TLc** e **TRc**, permitem mover o cursor para um ponto específico na linha de entrada ou saída atual. A seguir, a forma geral desses descritores:

Tc, TLc e TRc

Significados dos símbolos:

- **Tc** = move para a posição c.
- **TLc** = move c posições para trás.
- **TRc** = move c posições para frente.

8.5 Controle de posição vertical

8.5.1 O descritor de edição barra: /

O descritor de edição **barra (/)** permite pular para a linha seguinte em um processo de entrada ou saída de dados. A seguir a forma geral desse descritor:

/ e r/

Significados dos símbolos:

- **/** = descritor barra. Permite ao programa pular para a linha seguinte.
- **r** = indicador de repetição. Indica o número de vezes que o descritor barra será executado. Assim 3/ é equivalente a ///.

Se o descritor barra for usado na entrada de dados, como em um comando **READ** por exemplo, o programa pula para a seguinte linha. Qualquer dado inserido que sobrar na linha (estiver depois do descritor barra) será ignorado. Se houverem mais dados a serem inseridos depois do descritor barra, a entrada de dados continua na próxima linha.

Se o descritor barra for usado na saída de dados, como em um comando **WRITE** por exemplo, o programa pula para a seguinte linha. Se houverem mais dados a ser imprimidos depois do descritor barra, a saída de dados continua na próxima linha.

OBSERVAÇÃO:

Diferente dos outros descritores, um descritor barra não precisa ser separado de outros descritores por vírgulas. Um descritor vizinho pode ser outro descritor barra ou qualquer outro descritor.

```
READ(*,"(2X, I5, 2X, F12.3//I5,2X, F10.2)") a, b, c, d
```

8.6 Agrupamento de descritores

Às vezes é necessário agrupar vários descritores para serem executados em sequência. Outras vezes, além de agrupá-los, pode também ser necessário repetir o agrupamento algumas vezes. Para atender essa necessidade FORTRAN possui o descritor de agrupamento: (). A seguir, as formas gerais do descritor de agrupamento.

() e r()

- **()**: Os descritores contidos no descritor de agrupamento () são executados uma vez somente.
- **r()**: Os descritores contidos no descritor de agrupamento () são executados r vezes. Exemplo:

```
(I6, 3(I5, F7.3), I4)
```

é equivalente a:

```
(I6, I5, F7.3, I5, F7.3, I5, F7.3, I4)
```

- Podemos ter descritores de agrupamento **aninhados**. Útil quando queremos ter uma repetição de descritores dentro de outra repetição de descritores, como podemos ver no exemplo abaixo:

```
(I6, 2(3(I5, F7.3), I4))
```

é equivalente a:

```
(I6, I5, F7.3, I5, F7.3, I5, F7.3, I4, I5, F7.3, I5, F7.3, I5, F7.3, I4)
```

9 Funções

Até agora, os exemplos que nós vimos são de programas pequenos, feitos inteiramente

na função principal (“PROGRAM”). No entanto, os programas que resolvem problemas do mundo real, costumam ser bem maiores, o que acaba tornando-os difíceis de administrar pelo programador.

Uma técnica muito conhecida e de eficiência comprovada é a técnica “**dividir para conquistar**”. Essa técnica consiste em dividir o programa em partes menores, cada uma com uma função específica a realizar, e mais fácil de administrar do que se estivessem todas as partes juntas em PROGRAM.

Essas partes menores são chamadas de funções ou subprogramas. A seguir, a sintaxe de uma função:

```
tipo FUNCTION  nome_funcao (arg1, arg2, ..., argn)
    IMPLICIT  NONE

[declaracao_variaveis_e_constantes]
[instrucoes_executaveis]
[subprogramas]

END FUNCTION  nome_funcao
```

A definição da sintaxe da função começa com o **tipo** da função, o qual indica o tipo de dado que a função vai receber durante a sua execução e o tipo de dado que vai retornar quando a função terminar a sua execução.

Em seguida temos a palavra-chave **FUNCTION**, que caracteriza a nossa definição como uma função.

Depois vem o **nome da função**, que é o nome pelo qual essa função vai ser chamada para executar uma tarefa.

A **lista de argumentos** são os dados que a função necessita para executar a sua tarefa e que devem ser fornecidos na chamada da função. Os argumentos listados na definição são chamados de argumentos formais, enquanto que os argumentos passados na chamada da função são chamados de argumentos reais. Na definição da função, os argumentos formais ficam dentro de parênteses e separados por vírgulas. Apenas nomes de variáveis podem ser utilizados como argumentos formais, expressões não podem ser utilizadas. Se a função não necessitar de nenhum argumento formal, os parênteses devem ficar vazios: ().

A instrução **IMPLICIT NONE** é útil para evitar que FORTRAN atribua um tipo às variáveis automaticamente, de acordo com algumas regras consideradas obsoletas e não recomendáveis atualmente.

O **corpo da função** é idêntico ao corpo de PROGRAM com declaração, comandos executáveis e subprogramas (funções). Na parte de declarações, os argumentos formais devem ser declarados além do seu tipo de dado, como **INTENT(IN)**, indicando que o valor dessa variável será obtido apenas do argumento correspondente, e que a função não mudará o seu conteúdo.

A função termina com **END PROGRAM** seguido do nome da função.

Uma função é chamada dentro de PROGRAM, que especifica o nome da função a ser chamada e fornece as informações (argumentos reais) que a função necessita para fazer o seu trabalho.

A função recebe os argumentos passados como argumentos formais, faz algumas computações (ações realizadas por um computador) e retorna um resultado que é atribuído ao próprio nome da função, como podemos ver no exemplo abaixo:

```
nome_funcao = expressão
```

O nome da função não pode ser usado no lado direito de uma instrução de atribuição, a única exceção para essa regra são as funções recursivas.

A seguir um exemplo de função, que recebe um valor REAL indicando a nota de um aluno e retorna um valor lógico indicando a condição do aluno (.TRUE. para aprovado e .FALSE. para desaprovado).

```
LOGICAL FUNCTION  Aprov_Desaprov(nota)
  IMPLICIT  NONE

      REAL, INTENT(IN) :: nota

      IF (nota >= 5.0) THEN
        Aprov_Desaprov = .TRUE.
      ELSE
        Aprov_Desaprov = .FALSE.
      END IF
END FUNCTION  Aprov_Desaprov
```

O exemplo acima pode se tornar mais curto se usarmos a atribuição lógica. Nesse tipo de atribuição, o programa avalia se a expressão é .TRUE. ou .FALSE., e atribui o resultado à variável do lado esquerdo do operador de atribuição: =. Veja como ficaria:

```
Aprov_Desaprov = nota >= 5.0
```

Você deve estar se perguntando, onde eu coloco as minhas funções? Elas podem ser internas ou externas ao programa principal. Aqui, nós veremos apenas funções internas.

As funções internas ficam no final do programa principal, no lugar destinado aos subprogramas. Veja a sintaxe:

```
PROGRAM  nome_programa
  IMPLICIT  NONE
  [declaracao_variaveis_e_constantas]
  [instrucoes_executaveis]
CONTAINS
  [suas_funcoes]
END PROGRAM  nome_programa
```

Agora veja o seguinte exemplo que tem duas funções, para achar o maior e o menor numero digitado pelo usuário:

```
! Autor:.....Weyden Daniel Castro
! Data:.....05/2005
! Instituicao:...UNASP
! Curso:.....Ciencia da Computacao
! Disciplina:....Paradigmas de Linguagens de Programacao
! Professor:.....Ausberto Castro

! -----
! Este programa tem duas funcoes para calcular o maior e o menor
! numero de tres numeros inseridos pelo usuario.
! -----

PROGRAM maior_menor

    IMPLICIT NONE

    INTEGER :: num1, num2, num3
    INTEGER :: maior, menor

    WRITE(*, "(10X, A//)") "Programa maior e menor"
    WRITE(*, "(A//)") "Entre 3 numeros: "
    READ(*, *) num1, num2, num3
    maior = The_biggest(num1, num2, num3)
    menor = The_smallest(num1, num2, num3)
    WRITE(*, "(/A, I6)") "Maior: ", maior
    WRITE(*, "(/A, I6)") "Menor: ", menor

CONTAINS

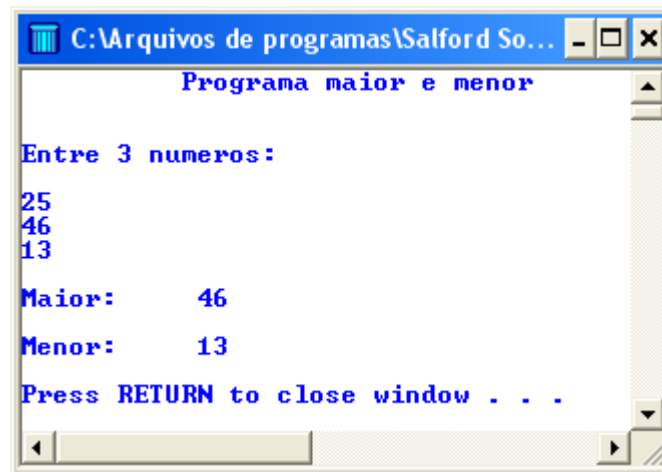
    INTEGER FUNCTION The_biggest(n1, n2, n3)
        IMPLICIT NONE
        INTEGER, INTENT(IN) :: n1, n2, n3

        The_biggest = n1
        IF (n2 > The_biggest) The_biggest = n2
        IF (n3 > The_biggest) The_biggest = n3
    END FUNCTION The_biggest

    INTEGER FUNCTION The_smallest(n1, n2, n3)
        IMPLICIT NONE
        INTEGER, INTENT(IN) :: n1, n2, n3

        The_smallest = n1
        IF (n2 < The_smallest) The_smallest = n2
        IF (n3 < The_smallest) The_smallest = n3
    END FUNCTION The_smallest

END PROGRAM maior_menor
```



OBSERVAÇÃO:

AS funções internas não podem conter outras funções dentro delas, isso é possível apenas para funções externas.

9.1 Chamada de funções

As funções podem ser chamadas em uma expressão ou na saída de dados, como em um comando WRITE por exemplo.

Essa chamada deve ser feita passando os argumentos reais que serão utilizados pela função, para executar a sua tarefa.

Supondo que tenhamos definido uma função media, que receba três números reais e devolva a média dos três números. Podemos chamar a função média em uma instrução como a seguinte:

```
media_numeros = media(5.0, 12.0, 8.0)
```

Nesse caso, a função media retorna 25.0, que é atribuído à variável media_numeros.

Algumas regras devem ser observadas para que os argumentos sejam passados corretamente para a função.

- O número de argumentos reais e de argumentos formais deve ser igual.
- Os tipos do argumento real e do argumento formal correspondente deve ser igual.
- Os argumentos reais podem ser constantes, expressões ou variáveis. Vejamos novamente a função média, sendo usada agora com argumentos diferentes.

```
REAL :: a = 5.0, b = 12.0
```

```

..... = ... + Average(5.0, 12.0) + ..... ! constantes
..... = ... + Average(a, b ) + ..... ! variaveis
..... = ... + Average(a+b, b*a, (b+c)/b) + ..... ! expressoes

```

9.2 Regras de escopo

O escopo de uma entidade (variável, parâmetro ou função) são os lugares em que essa entidade é visível ou acessível.

A seguir algumas regras que nos ajudarão a definir o escopo de uma entidade:

- O escopo de uma entidade é o programa ou função em que ela está declarada.
- Uma entidade global é visível para todas as funções internas da função em que ela está declarada, e para a própria função na qual ela está declarada.
- Uma entidade declarada no escopo de uma outra entidade é sempre uma entidade diferente ainda que tenham o mesmo nome.

10 Módulos

Um módulo é uma característica introduzida no FORTRAN 90 que permite agrupar funções. Assim fica mais fácil incluir grupos de funções amplamente usadas, em outros programas.

Veja a sintaxe de um módulo:

```

MODULE  nome_modulo
  IMPLICIT NONE
  [declaracao_variaveis_e_constantes]
CONTAINS
  [funcoes_internas]
END MODULE  nome_modulo

```

Um modulo não contém instruções executáveis como em um programa, ele contém apenas declarações e funções.

Para tornar o conteúdo de um módulo disponível em um programa ou em outro módulo, devemos utilizar o comando **USE** em uma de suas formas no início do programa ou módulo.

- `USE nome_modulo`

Indica que o programa vai usar o módulo de nome: nome_modulo. Torna todo o conteúdo (variáveis, parâmetros e funções internas) do módulo disponível para o programa ou módulo que contém o comando USE.

- `USE nome_modulo, ONLY: nome_1, nome_2, ..., nome_n`

Essa forma torna disponível apenas as entidades (variáveis, parâmetros ou funções internas) desejadas, ignorando as entidades que não são necessárias. Assim apenas as entidades listadas serão adicionadas.

- `novo_nome => nome_no_modulo`

Essa forma permite renomear o nome de uma entidade pertencente ao módulo que desejamos adicionar. Essa forma pode ser usada com qualquer uma das duas formas anteriores. Exemplos:

```
USE Constantes_matematicas, ONLY: log10, raiz_quadrada => sqrt
```

```
USE Constantes_matematicas, raiz_quadrada => sqrt
```

Renomear o nome de uma entidade de um módulo pode ser necessário se tivermos outro nome igual no programa ou módulo que chama o módulo.

Geralmente, programas e módulos são armazenados em arquivos diferentes, com o sufixo f.90. Mas, quando você compila o seu programa todos os módulos utilizados devem, ser compilados também.

11 Subrotinas

Uma subrotina em FORTRAN é muito parecida com uma função, com a única diferença que uma subrotina não retorna um valor no nome da função. Subrotinas retornam valores, mais do que um valor até, apenas não fazem isso no nome da função. Veja a baixo a sintaxe de uma subrotina:

```
SUBROUTINE nome_subrotina (arg1, arg2, ..., argn)
  IMPLICIT NONE
  [declaracao_variaveis_e_constantes]
  [instrucoes_executaveis]
  [subprogramas]
END SUBROUTINE nome_subrotina
```

A lista de argumentos de uma subrotina são chamados argumentos_formais, e têm as mesmas regras dos argumentos formais de uma função.

Uma subrotina pode não receber nenhum valor, ficando dois parênteses vazios () ou nenhum parêntese na sua especificação.

Diferente das funções, o nome de uma subrotina não pode receber nenhum valor, ele é usado apenas para referenciar a subrotina, como em um comando CALL por exemplo.

E uma característica muito importante, uma subrotina pode modificar o valor de um argumento formal, o que não era possível usando funções.

11.1 *INTENT*

Nos já vimos `INTENT(IN)` em funções, indicando que um argumento formal recebe valores apenas de fora da função, não podendo ser mudado dentro da função. Como uma subrotina não pode retornar valores através do seu nome, ela retorna o resultado de computações, se houverem, através de alguns de seus argumentos formais. Sendo assim, podemos ter três casos de argumentos formais a considerar:

- Se um argumento apenas recebe valores de fora da subrotina, ele é ainda declarado como `INTENT(IN)`.
- Se um argumento não recebe nenhum valor de fora da subrotina, sendo usado apenas para retornar um resultado, ele é declarado como `INTENT(OUT)`.
- Se um argumento recebe valores de fora da subrotina e retorna resultados, ele é declarado como `INTENT(INOUT)`.

11.2 *O comando CALL*

Diferente das funções, uma subrotina não pode ser chamada em uma expressão. Ela deve ser chamada com o comando `CALL`. A seguir, as sintaxes possíveis do comando `CALL`:

```
CALL nome_subrotina (arg1, arg2, ..., argn)
```

```
CALL nome_subrotina ()
```

```
CALL nome_subrotina
```

Se argumentos reais tiverem de ser passados, o comando `CALL` terá a primeira forma das sintaxes acima.

Se nenhum argumento real for passado, o comando `CALL` poderá ter a segunda ou a terceira forma das sintaxes acima.

Tenha cuidado para que o número de argumentos reais no comando `CALL` seja igual ao número de argumentos formais da subrotina.

12 Arrays

12.1 Arrays de uma dimensão

Um array é uma coleção de dados do mesmo tipo, alocados em posições de memória consecutivas. Para referenciar um elemento do array, devemos usar o nome do array seguido pelo índice do elemento desejado entre parênteses. Veja a sintaxe do elemento de um array:

```
nome_array (expressão_inteira)
```

Onde expressão_inteira é uma expressão que produz um resultado inteiro. O resultado inteiro da expressão_inteira, deve estar dentro da amplitude do array.

Um array de uma dimensão têm a seguintes propriedades:

- Um nome.
- Um tipo, que é o mesmo para todos os elementos do array.
- Uma amplitude, que é a faixa na qual os índices dos elementos do array podem variar. Um índice de um array deve ser um número inteiro. Se a amplitude de um array for de 1 a 10, 1 é o menor índice ou limite mínimo, e 10 é o maior índice ou limite máximo. A seguir a forma da amplitude um array:

```
limite_minimo : limite_maximo
```

Limite mínimo e limite máximo podem ser parâmetros que sejam valores inteiros. Se o limite mínimo de uma amplitude for 1, o limite mínimo e os dois pontos (:) podem ser omitidos.

12.1.1 Declarando um array

É o programador que deve especificar o tamanho da amplitude que ele deseja para o seu array, o que deve ser feita na declaração. Veja abaixo, a sintaxe da declaração de um array:

```
tipo, DIMENSION( amplitude ) :: nome-1, nome-2, ..., nome-n
```

Onde tipo é o tipo dos arrays declarados nesta declaração (nome-1, nome-2, ..., nome-n), DIMENSION é uma palavra-chave que deve ser colocada na declaração de um array, e “(nome-1, nome-2, ..., nome-n)” é uma lista de nomes de arrays que serão declarados nesta declaração.

12.1.2 Loop-DO implícito

Um loop-DO implícito provê uma maneira de listar muitos itens rapidamente. Os itens listados podem ser variáveis, expressões ou até loops-DO implícitos.

Veja abaixo a sintaxe de um loop-DO implícito:

```
(item-1, item-2, ..., item-n, variável_controle = initial, final, tamanho_passo)
```

Assim como em um loop-DO, se tamanho_passo for 1, tamanho_passo pode ser omitido.

Veja o seguinte exemplo, em que i vai assumir os valores 1, 2, 4, 6, 8 e 10. Os itens são: i e i*i.

```
( i, i*i, i = 1, 10, 2 )
```

Para i = 1, os itens listados são: 1 e 1 = 1*1. Para i = 2, os itens listados são 2 e 4. Para i = 4, os itens listados são 4 e 16. Para i = 6, os itens listados são 6 e 36. Para i = 8, os itens listados são 8 e 64. Para i = 10, os itens listados são 10 e 100.

Combinando os seis casos (1, 2, 4, 6, 8 e 10), o total de itens listados pelo loop-DO implícito é:

1, 1, 2, 4, 4, 16, 6, 36, 8, 64, 10, 100

12.1.3 Entrada e saída de dados em um array

Há duas maneiras de ler dados para um array, usando loop-DO ou usando loop-DO implícito. Analise os dois:

1) Loop-DO	2) Loop-DO implícito
<pre>INTEGER, DIMENSION(1:10) :: x INTEGER :: n, i READ(*,*) n DO i = 1, n READ(*,*) x(i) END DO</pre>	<pre>INTEGER, DIMENSION(1:10) :: x INTEGER :: n, i READ(*,*) n READ(*,*) (x(i), i=1, n)</pre>

O loop-DO, na verdade vai usar n comandos READ (um a cada iteração) para ler os elementos do array. Assim, a forma 1 vai poder ler uma entrada semelhante a essa:

```
5
10
20
30
40
50
```

Já o loop-DO implícito vai ler uma entrada utilizando apenas um comando READ. Se n for igual a 5, o READ acima seria equivalente ao seguinte READ:

```
READ(*,*) x(1), x(2), x(3), x(4), x(5)
```

Como esse formato utiliza apenas um comando READ para ler os elementos do array, vai poder ler uma entrada como a anterior e um a entrada como a entrada mostrada abaixo:

5
10 20 30 40 50

Também temos duas maneiras de imprimir dados de um array, usando loop-DO ou usando loop-DO implícito. Analise os dois:

1) Loop-DO	2) Loop-DO implícito
<pre> INTEGER, DIMENSION(1:10) :: x, y INTEGER :: n = 5, i DO i = 1, 5 WRITE(*,*) x(i), y(i) END DO </pre>	<pre> INTEGER, DIMENSION(1:10) :: x, y INTEGER :: n = 5, i WRITE(*,*) (x(i), y(i), i=1, n) </pre>

A primeira forma vai produzir uma saída em 5 linhas diferentes, enquanto a segunda forma vai produzir uma saída em apenas uma linha.

12.2 Arrays multi-dimensionais

Podemos dizer que um array multi-dimensional é uma tabela ou um grupo de tabelas. Por exemplo, se quisermos encontrar um aluno, podemos procurar em uma tabela bi-dimensional pela sua sala, e pelo seu número de chamada.

	1	2	3	4	5	6	7	8	9	10
11A	Carlos	Maria	Pedro	Joyce	Lucas	Marília	Fernanda	Tadeu	Armando	Priscila
21A	Camila	João	Amanda	Kátia	Gabriel	Daniel	Bianca	Paulo	Marcos	Bruna
31A	Henrique	Tiago	Mateus	Luisa	Sílvia	Denise	Juliana	Ana	Vitor	Roberto

Mas podemos procurar um aluno também pelo seu horário de estudos, pela sua sala e pela seu número de chamada. Isso pode ser feito pesquisando em um grupo de tabelas:

DIURNO										
	1	2	3	4	5	6	7	8	9	10
11A	Carlos	Maria	Pedro	Joyce	Lucas	Marília	Fernanda	Tadeu	Armando	Priscila
21A	Camila	João	Amanda	Kátia	Gabriel	Daniel	Bianca	Paulo	Marcos	Bruna
31A	Henrique	Tiago	Mateus	Luisa	Sílvia	Denise	Juliana	Ana	Vitor	Roberto

NOTURNO										
	1	2	3	4	5	6	7	8	9	10
11B	Carolina	Martha	Natália	Diego	Verônica	Gabriela	Renato	Sara	Tiago	José
21B	Manoel	Cláudia	Simone	Ricardo	Ludmila	Fernando	Antônio	Julia	Rosa	Paulo
31B	Alexandre	Sílvia	Raquel	Júlio	Henrique	Anderson	Andressa	Cíntia	Bruno	Taís

Os elementos de um array podem ser referenciados com um índice para cada dimensão. Veja os seguintes exemplos de elementos de arrays de duas dimensões:

`soma(4, 5)` ! O elemento está na 4^o linha e na 5^o coluna.

`alunos(3, 1)` ! O elemento está na 3^o linha e na 1^o coluna.

12.2.1 Declarando um array

Sua declaração é semelhante à declaração de arrays de uma dimensão, apenas declaramos mais amplitudes.

A declaração abaixo declara três arrays de duas dimensões:

```
REAL, DIMENSION(1:10, 1:10) :: a, b, c
```

As amplitudes de cada dimensão não precisam ser iguais. Veja o seguinte exemplo:

```
REAL, DIMENSION(1:15, 5:30) :: Funcionarios
```

12.2.2 Entrada e saída de dados em um array

Nós podemos ler ou imprimir elementos de um array, linha por linha ou coluna por coluna. Veja o seguinte exemplo em que um array é lido linha por linha:

```
INTEGER, PARAMETER :: MAXLIN = 2, MAXCOL = 4
INTEGER, DIMENSION(1 : MAXLIN, 1 : MAXCOL) :: M
INTEGER :: I, j
.
.
.
DO i = 1, MAXLIN
    DO j = 1, MAXCOL
        READ(*, *) M(i, j)
    END DO
END DO
.
.
.
```

13 Compiladores

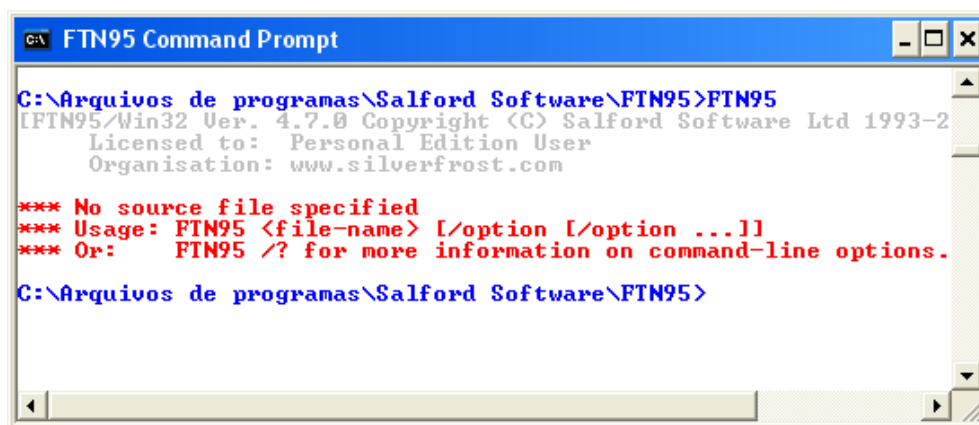
13.1 Salford FTN95

Empresa.....: Salford Software Ltd.

Compilador.....: Salford FTN95 Fortran for Windows,
Personal Edition, v. 4.80

IDE.....: Salford Plato v. 3.17

Website.....: <http://www.silverfrost.com/>

A screenshot of a Windows Command Prompt window titled 'C:\ FTN95 Command Prompt'. The window shows the execution of the 'FTN95' command. The output text is as follows:

```
C:\Arquivos de programas\Salford Software\FTN95>FTN95
[FTN95/Win32 Ver. 4.7.0 Copyright (C) Salford Software Ltd 1993-2
Licensed to: Personal Edition User
Organisation: www.silverfrost.com

*** No source file specified
*** Usage: FTN95 <file-name> [/option [/option ...]]
*** Or:   FTN95 /? for more information on command-line options.

C:\Arquivos de programas\Salford Software\FTN95>
```



Com o compilador Salford FTN95: Fortran for Windows você pode criar aplicações Convencionais Windows e .NET para console. Enquanto a maioria dos compiladores FORTRAN não têm suporte para menus ou ícones, Salford FTN95 possibilita a criação desse tipo de programas. Para auxiliar na construção de

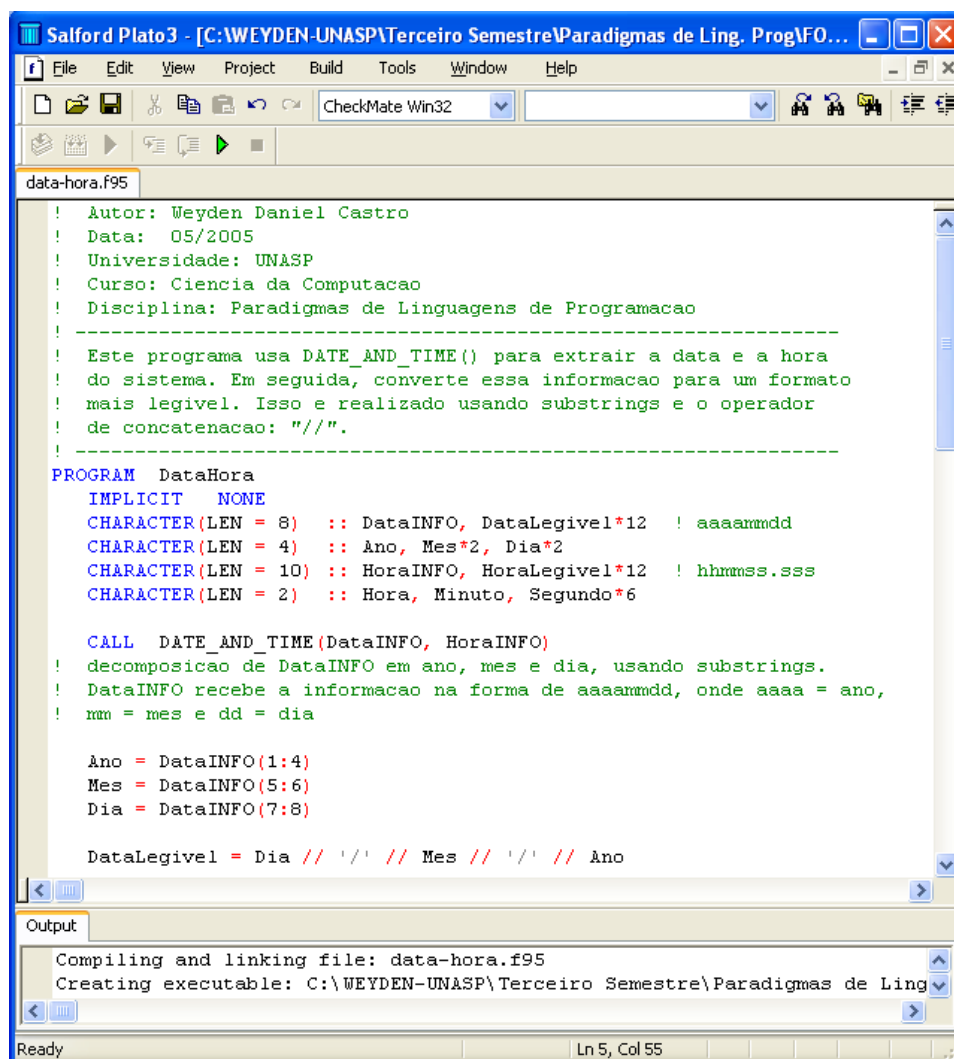
aplicações Windows, Salford FTN95 possui o **ClearWin+** um poderoso conjunto de rotinas para construir aplicações Windows.

É possível compilar e executar em Salford FTN95, programas em FORTRAN 77, FORTRAN 90, FORTRAN 95, e ainda, algumas características de FORTRAN 2003.

Com Salford, os programadores podem migrar seus códigos fonte para Microsoft .NET sem nenhuma restrição de versões da linguagem. Todas as características das versões FORTRAN 77, FORTRAN90 e FORTRAN95 podem ser convertidas instantaneamente para Microsoft.NET.

Salford FTN95 está disponível em três versões:

- **Personal:** Esta versão é gratuita, porém é de uso estritamente pessoal.
- **Commercial:** Essa licença é necessária para usar Salford em um ambiente comercial.
- **Academic:** O custo desta versão é menor em relação à versão comercial. Destina-se a uso em estabelecimentos educacionais, para pesquisas ou ensino.



```
Salford Plato3 - [C:\WEYDEN-UNASP\Terceiro Semestre\Paradigmas de Ling. Prog\FO...
File Edit View Project Build Tools Window Help
CheckMate Win32
data-hora.f95
! Autor: Weyden Daniel Castro
! Data: 05/2005
! Universidade: UNASP
! Curso: Ciencia da Computacao
! Disciplina: Paradigmas de Linguagens de Programacao
! -----
! Este programa usa DATE_AND_TIME() para extrair a data e a hora
! do sistema. Em seguida, converte essa informacao para um formato
! mais legivel. Isso e realizado usando substrings e o operador
! de concatenacao: "//".
! -----
PROGRAM DataHora
IMPLICIT NONE
CHARACTER(LEN = 8) :: DataINFO, DataLegivel*12 ! aaaammdd
CHARACTER(LEN = 4) :: Ano, Mes*2, Dia*2
CHARACTER(LEN = 10) :: HoraINFO, HoraLegivel*12 ! hhmmss.sss
CHARACTER(LEN = 2) :: Hora, Minuto, Segundo*6

CALL DATE_AND_TIME(DataINFO, HoraINFO)
! decomposicao de DataINFO em ano, mes e dia, usando substrings.
! DataINFO recebe a informacao na forma de aaaammdd, onde aaaa = ano,
! mm = mes e dd = dia

Ano = DataINFO(1:4)
Mes = DataINFO(5:6)
Dia = DataINFO(7:8)

DataLegivel = Dia // '/' // Mes // '/' // Ano

Output
Compiling and linking file: data-hora.f95
Creating executable: C:\WEYDEN-UNASP\Terceiro Semestre\Paradigmas de Ling
Ready Ln 5, Col 55
```

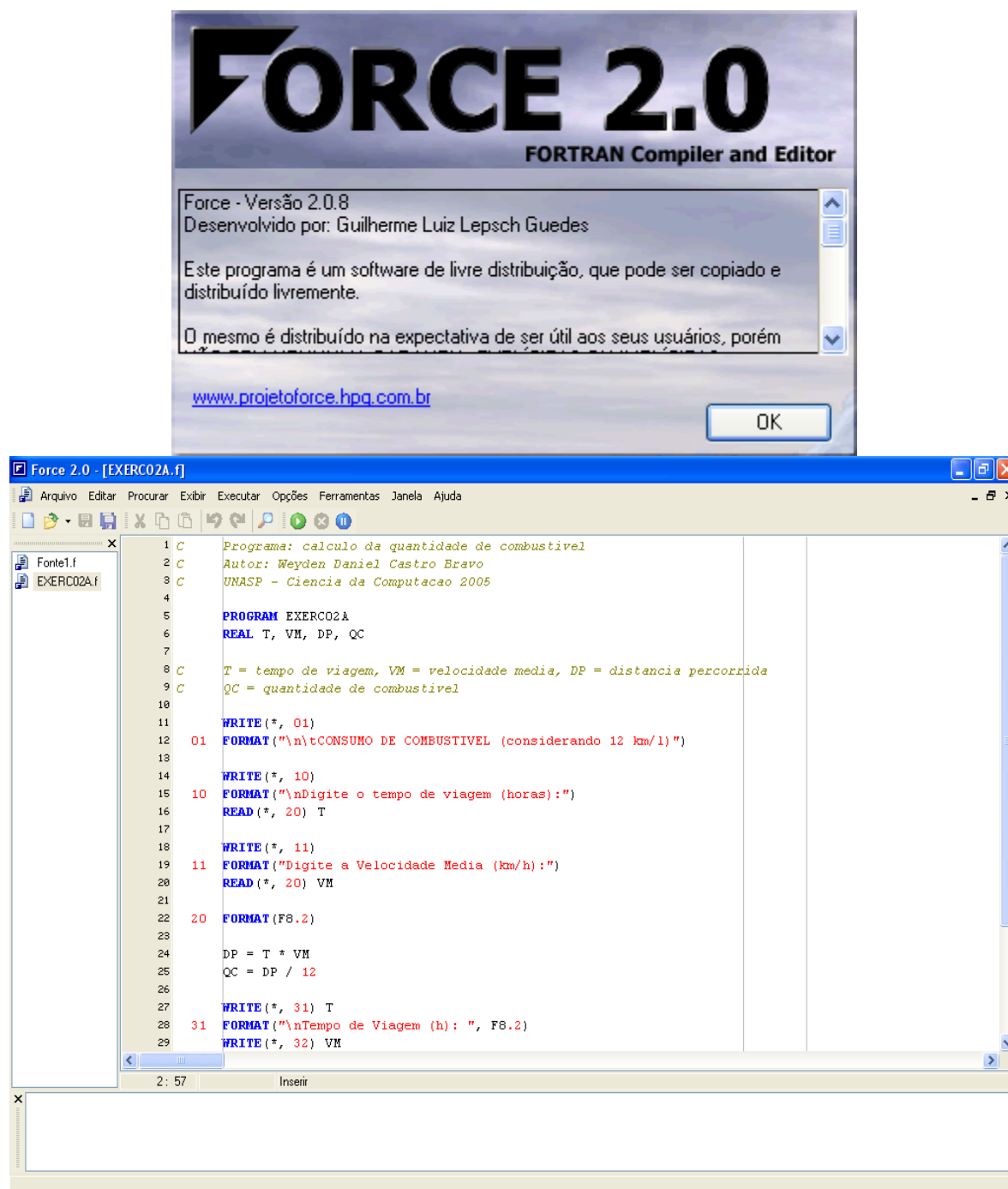
13.2 Force

Force é um ambiente de desenvolvimento gratuito para a linguagem FORTRAN 77.

Compilador.....: G77

IDE.....: Force 2.0 FORTRAN Compiler and Editor

Website.....: <http://www.guilherme.tk/>



14 Os 5 exemplos

14.1 Quatro operações aritméticas

```
! Autor:.....Weyden Daniel Castro
! Data:.....05/2005
! Instituicao:...UNASP
! Curso:.....Ciencia da Computacao
! Disciplina:....Paradigmas de Linguagens de Programacao
! Professor:.....Ausberto Castro

! -----
! Este programa tem um menu que permite ao usuário escolher
! uma das quatro operações aritméticas.
! -----

PROGRAM quatro_operacoes
  IMPLICIT NONE

  INTEGER :: opcao
  INTEGER :: num1, num2
  REAL    :: fnum1, fnum2

  WRITE(*, "(//15X, A)") "Operacoes Matematicas"
  WRITE(*, "(5X, A//)") "===== "

  WRITE(*, "(A)") "Escolha a opcao desejada:"
  WRITE(*, "(/A)") "1) Adicao"
  WRITE(*, "(/A)") "2) Subtracao"
  WRITE(*, "(/A)") "3) Multiplicacao"
  WRITE(*, "(/A)") "4) Divisao"
  WRITE(*, "(//A/)") "Opcao:"

  READ(*, *) opcao

  SELECT CASE (opcao)
    CASE (1)
      WRITE(*, "(//A/)") "Digite dois numeros inteiros: "
      READ(*, *) num1, num2
      WRITE(*, "(/A, I6)") "Soma: ", soma(num1, num2)

    CASE (2)
      WRITE(*, "(//A/)") "Digite dois numeros inteiros: "
      READ(*, *) num1, num2
      WRITE(*, "(/A, I6)") "Subtracao: ", subtracao(num1, num2)

    CASE (3)
      WRITE(*, "(//A/)") "Digite dois numeros inteiros: "
      READ(*, *) num1, num2
      WRITE(*, "(/A, I6)") "Multiplicacao: ", multiplicacao(num1, num2)

    CASE (4)
      WRITE(*, "(//A/)") "Digite dois numeros reais: "
      READ(*, *) fnum1, fnum2
      WRITE(*, "(/A, F10.3)") "Divisao: ", divisao(fnum1, fnum2)
```



```

        CASE DEFAULT
            WRITE(*, *) "Opcao Invalida"
        END SELECT

CONTAINS

!*****
!          FUNCAO SOMA
!*****

        INTEGER FUNCTION soma(n1, n2)
            IMPLICIT NONE

            INTEGER, INTENT(IN) :: n1, n2

            soma = n1 + n2

        END FUNCTION soma

!*****
!          FUNCAO SUBTRACAO
!*****

        INTEGER FUNCTION subtracao(n1, n2)
            IMPLICIT NONE

            INTEGER, INTENT(IN) :: n1, n2

            subtracao = n1 - n2

        END FUNCTION subtracao

!*****
!          FUNCAO MULTIPLICACAO
!*****

        INTEGER FUNCTION multiplicacao(n1, n2)
            IMPLICIT NONE

            INTEGER, INTENT(IN) :: n1, n2

            multiplicacao = n1 * n2

        END FUNCTION multiplicacao

!*****
!          FUNCAO DIVISAO
!*****

        REAL FUNCTION divisao(fn1, fn2)
            IMPLICIT NONE

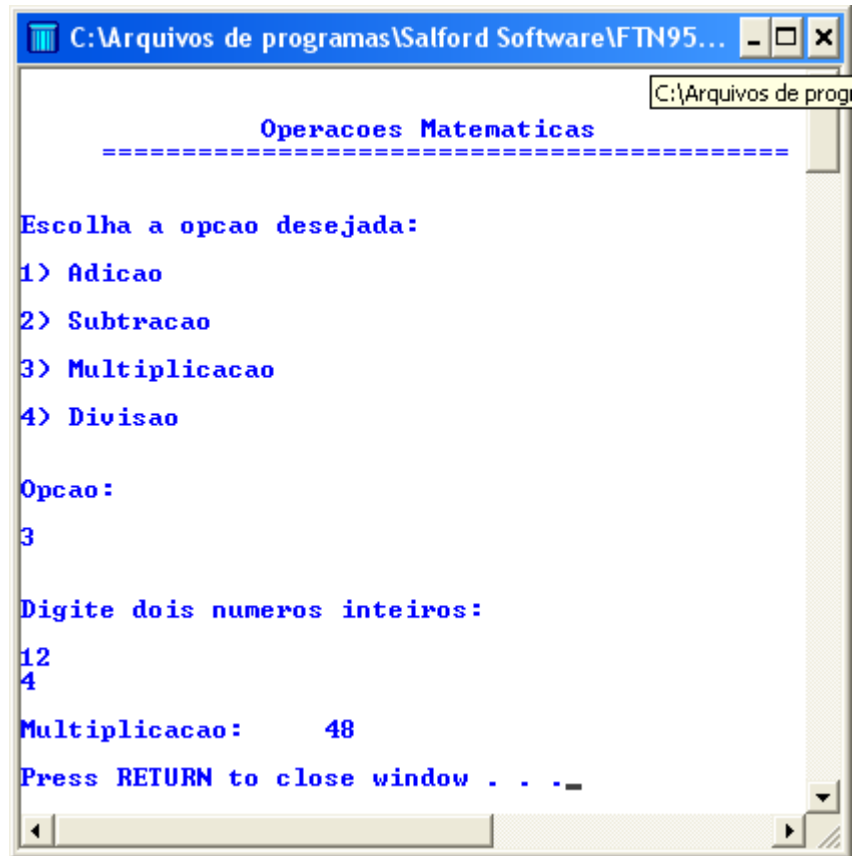
            REAL, INTENT(IN) :: fn1, fn2

            divisao = fn1 / fn2

        END FUNCTION divisao

END PROGRAM quatro_operacoes

```



14.2 Programa Gráfico

```
! Autor:.....Weyden Daniel Castro
! Data:.....05/2005
! Instituicao:...UNASP
! Curso:.....Ciencia da Computacao
! Disciplina:...Paradigmas de Linguagens de Programacao
! Professor:.....Ausberto Castro

!*****
****
!* Salford ClearWin+
*
!*
*
!*
*
!* compilar apoenas usando Fortran 95 1.6 ou superior
*
!*
*
```

```

!* Exemplo de grafico usando o formato %pl
*
!*
*
!*****
****

winapp
!
!
!      *****
!      *
!      *      Module of shared data for plot      *
!      *
!      *****
!

module data_to_plot
  use mswin
  parameter(narr=1000)
  integer, parameter :: rkind=selected_real_kind(15,307)
  integer :: param_window
  real (kind=rkind):: p1,p2,p3
  real (kind=rkind):: yarr1(narr)

  contains

!
!
!      *****
!      *
!      *      Routine to prepare plot data      *
!      *
!      *****
!

  subroutine prepare_data
    integer :: i,xx
    xx=0
    do i=1,narr
      yarr1(i)=p1*sin(xx/p3)*exp(-xx/p2)
      xx=xx+1
    enddo
  end subroutine prepare_data

!
!
!      *****
!      *
!      *      Function to respond to PLOT button      *
!      *
!      *****
!

  integer function re_plot()
    call prepare_data
    call simpleplot_redraw@
    re_plot=2
  end function re_plot

!
!
!      *****
!      *
!

```

```

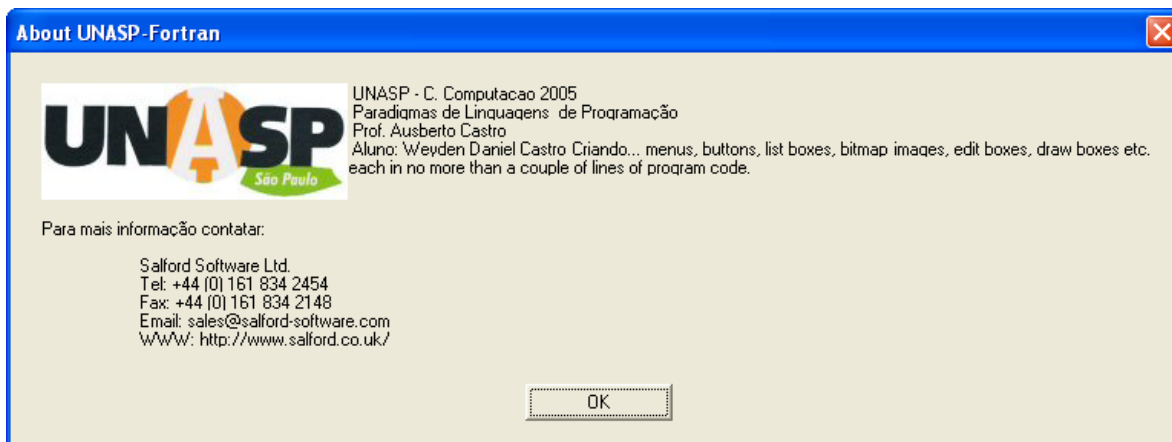
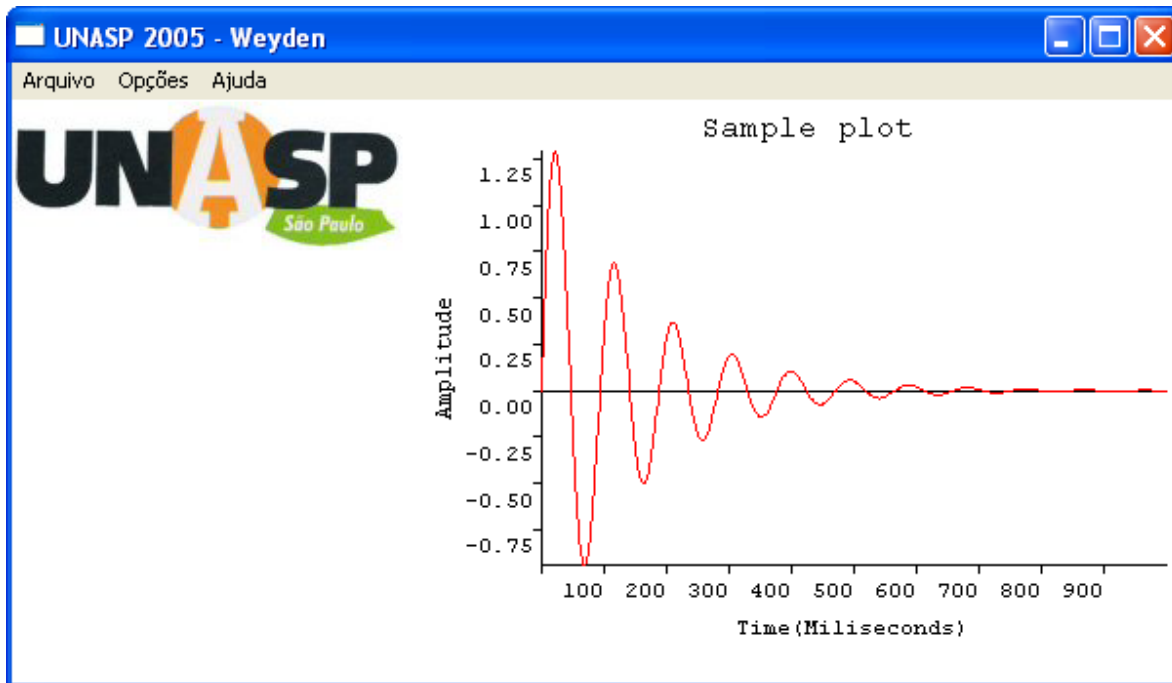
!      *      Function to display parameter box      *
!      *
!      *****
!
integer function set_params()
implicit none
integer :: i
!      Make sure we do not display the parameters window more than once
if(param_window < 0) return;
i=winio@('%ca[Plot parameters]&')
i=winio@('%ww[topmost]&')
i=winio@('%sy[3d]%sf&')
i=winio@('%eq[Y=P{sub 1}Sin(X/P{sub 3}) e{sup -X/P{sub
2}}]%ff%nl&',300,50)
i=winio@('%ob[scored]&')
i=winio@('P%sd1%`sd%fl%10rf%2nl&',0.01d0,1.0d10,p1)
i=winio@('P%sd2%`sd%fl%10rf%2nl&',0.01d0,1.0d10,p2)
i=winio@('P%sd3%`sd%fl%10rf&',0.01d0,1.0d10,p3)
i=winio@('%cb&')
i=winio@('%rj%^5bt[Plot]%2nl%rj%5bt[Close]&',re_plot)
i=winio@('%lw',param_window)
set_params=2
end function set_params
end module data_to_plot

!
!      *****
!      *
!      *      Main program - display window containing plot      *
!      *
!      *****
!
program simpleplot_example
use UNASP_about
use data_to_plot
use mswin
implicit none
integer :: i
p1=1.5;p2=150.0;p3=15
param_window=0
call prepare_data
i=winio@('%ca[UNASP 2005 - Weyden]&')
i=winio@('%bm[icon_1]&')
i=winio@('%ww[no_border]&')
i=winio@('%mn[&Arquivo[S&air],&Opções[Dar
&Parâmetros]]&', 'EXIT', set_params)
i=winio@('%mn[&Ajuda[Sobre]]&', about_box_cb)
i=winio@('%pl[x_axis="Time (Miliseconds)", y_axis=Amplitude, '&
//'title="Sample plot", colour=red]&', &
400,300,narr,0.0d0,1.0d0,yarr1)
i=winio@('%pv')
end

resources

icon_1 BITMAP "unaspsp2005.bmp"

```



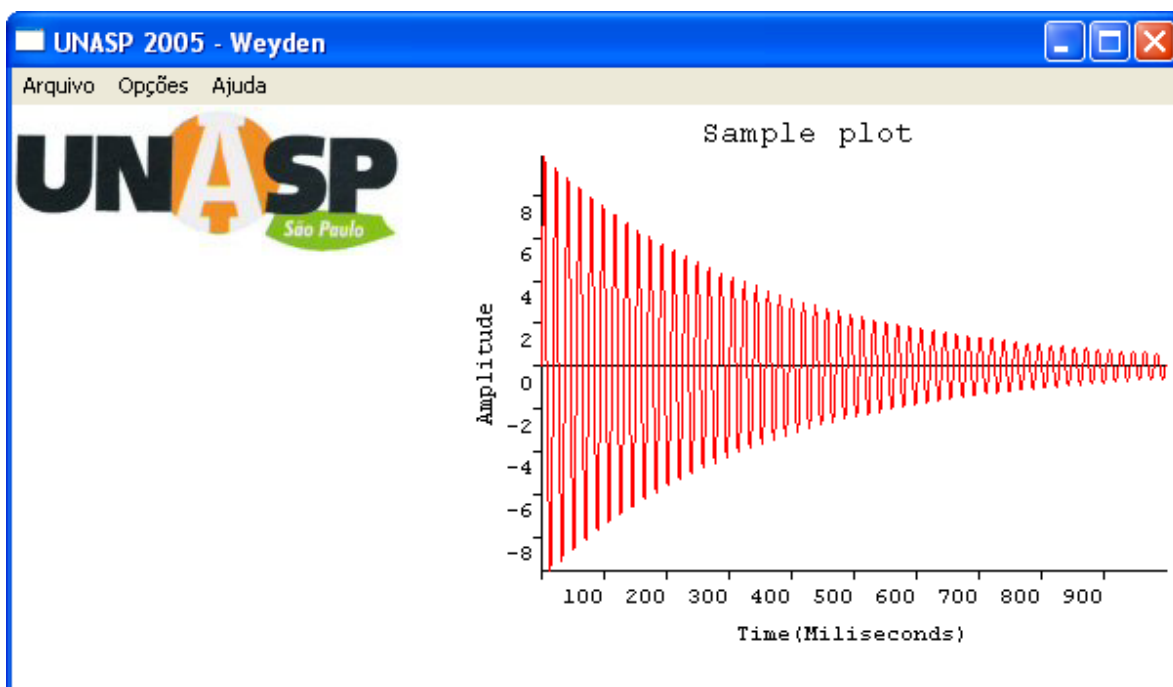
Plot parameters

$$Y = P_1 \sin(X/P_3) e^{-XP_2}$$

P₁
 Plot

P₂
 Close

P₃



14.3 QUICKSORT

```
! Autor:.....Weyden Daniel Castro
! Data:.....05/2005
! Instituicao:...UNASP
! Curso:.....Ciencia da Computacao
! Disciplina:....Paradigmas de Linguagens de Programacao
! Professor:.....Ausberto Castro
```

```

! -----
! PROGRAM Quicksort:
!   Este programa ordena uma lista de numeros inteiros, utilizando o
!   algoritmo
!   Quicksort.
! -----
PROGRAM Quicksort
  IMPLICIT NONE
  INTEGER, PARAMETER :: MAX_SIZE = 100
  INTEGER, DIMENSION(1:MAX_SIZE) :: InputData
  INTEGER, DIMENSION(1:MAX_SIZE) :: OutputData
  INTEGER                :: Tamanho
  INTEGER                :: i
  CHARACTER              :: c

  WRITE(*,*)
  WRITE(*,*) "Algoritmo Quicksort"
  WRITE(*,*) "====="
  WRITE(*,*)
  WRITE(*,*) "Quantos numeros deseja ordenar? (max 100)? : "
  READ(*,*) Tamanho
  WRITE(*,*) "Escreva os numeros a ordenar:"
  READ(*,*) (InputData(i), i = 1, Tamanho)

  WRITE(*,*) "Vetor de entrada:"
  WRITE(*,*) (InputData(i), i = 1, Tamanho)
  !chamada ao algoritmo quicksort
  CALL qsortd(InputData, OutputData, Tamanho)
  WRITE(*,*)
  WRITE(*,*) "Vetor Ordenado pelo algoritmo Quicksort:"
  WRITE(*,*) (InputData(OutputData(i)), i = 1, Tamanho)
  WRITE(*,*)

CONTAINS
  !=====
  SUBROUTINE qsortd(x,ind,n)

    ! Codigo convertido para FORTRAN 90 por Alan Miller
    ! Data: 2002-12-18

    IMPLICIT NONE
    INTEGER, PARAMETER :: dp = SELECTED_REAL_KIND(12, 60)

    !REAL (dp), INTENT(IN)  :: x(:)
    INTEGER, INTENT(IN)   :: x(:)
    INTEGER, INTENT(OUT)  :: ind(:)
    INTEGER, INTENT(IN)   :: n

!*****
***
!
RENKA
!
LAB.

ROBERT
OAK RIDGE NATL.

```

```

! THIS SUBROUTINE USES AN ORDER N*LOG(N) QUICK SORT TO SORT A
REAL (dp)
! ARRAY X INTO INCREASING ORDER. THE ALGORITHM IS AS FOLLOWS. IND
IS
! INITIALIZED TO THE ORDERED SEQUENCE OF INDICES 1,...,N, AND ALL
INTERCHANGES
! ARE APPLIED TO IND. X IS DIVIDED INTO TWO PORTIONS BY PICKING A
CENTRAL
! ELEMENT T. THE FIRST AND LAST ELEMENTS ARE COMPARED WITH T, AND
! INTERCHANGES ARE APPLIED AS NECESSARY SO THAT THE THREE VALUES
ARE IN
! ASCENDING ORDER. INTERCHANGES ARE THEN APPLIED SO THAT ALL
ELEMENTS
! GREATER THAN T ARE IN THE UPPER PORTION OF THE ARRAY AND ALL
ELEMENTS
! LESS THAN T ARE IN THE LOWER PORTION. THE UPPER AND LOWER
INDICES OF ONE
! OF THE PORTIONS ARE SAVED IN LOCAL ARRAYS, AND THE PROCESS IS
REPEATED
! ITERATIVELY ON THE OTHER PORTION. WHEN A PORTION IS COMPLETELY
SORTED,
! THE PROCESS BEGINS AGAIN BY RETRIEVING THE INDICES BOUNDING
ANOTHER
! UNSORTED PORTION.

! INPUT PARAMETERS - N - LENGTH OF THE ARRAY X.

!
! X - VECTOR OF LENGTH N TO BE SORTED.

!
! IND - VECTOR OF LENGTH >= N.

! N AND X ARE NOT ALTERED BY THIS ROUTINE.

! OUTPUT PARAMETER - IND - SEQUENCE OF INDICES 1,...,N PERMUTED IN
THE SAME
!
! FASHION AS X WOULD BE. THUS, THE
ORDERING ON
!
! X IS DEFINED BY Y(I) = X(IND(I)).

!*****
! NOTE -- IU AND IL MUST BE DIMENSIONED >= LOG(N) WHERE LOG HAS
BASE 2.

!*****

INTEGER :: iu(21), il(21)
INTEGER :: m, i, j, k, l, ij, it, itt, indx
REAL :: r
REAL (dp) :: t

! LOCAL PARAMETERS -

! IU,IL = TEMPORARY STORAGE FOR THE UPPER AND LOWER
! INDICES OF PORTIONS OF THE ARRAY X
! M = INDEX FOR IU AND IL
! I,J = LOWER AND UPPER INDICES OF A PORTION OF X
! K,L = INDICES IN THE RANGE I,...,J

```



```

! IJ =      RANDOMLY CHOSEN INDEX BETWEEN I AND J
! IT,ITT = TEMPORARY STORAGE FOR INTERCHANGES IN IND
! INDX =    TEMPORARY INDEX FOR X
! R =       PSEUDO RANDOM NUMBER FOR GENERATING IJ
! T =       CENTRAL ELEMENT OF X

IF (n <= 0) RETURN

! INITIALIZE IND, M, I, J, AND R

DO i = 1, n
    ind(i) = i
END DO
m = 1
i = 1
j = n
r = .375

! TOP OF LOOP

20 IF (i >= j) GO TO 70
IF (r <= .5898437) THEN
    r = r + .0390625
ELSE
    r = r - .21875
END IF

! INITIALIZE K

30 k = i

! SELECT A CENTRAL ELEMENT OF X AND SAVE IT IN T

ij = i + r*(j-i)
it = ind(ij)
t = x(it)

! IF THE FIRST ELEMENT OF THE ARRAY IS GREATER THAN T,
!   INTERCHANGE IT WITH T

indx = ind(i)
IF (x(indx) > t) THEN
    ind(ij) = indx
    ind(i) = it
    it = indx
    t = x(it)
END IF

! INITIALIZE L

l = j

! IF THE LAST ELEMENT OF THE ARRAY IS LESS THAN T,
!   INTERCHANGE IT WITH T

indx = ind(j)
IF (x(indx) >= t) GO TO 50

```

```

ind(ij) = indx
ind(j) = it
it = indx
t = x(it)

! IF THE FIRST ELEMENT OF THE ARRAY IS GREATER THAN T,
!   INTERCHANGE IT WITH T

indx = ind(i)
IF (x(indx) <= t) GO TO 50
ind(ij) = indx
ind(i) = it
it = indx
t = x(it)
GO TO 50

! INTERCHANGE ELEMENTS K AND L

40 itt = ind(l)
ind(l) = ind(k)
ind(k) = itt

! FIND AN ELEMENT IN THE UPPER PART OF THE ARRAY WHICH IS
!   NOT LARGER THAN T

50 l = l - 1
indx = ind(l)
IF (x(indx) > t) GO TO 50

! FIND AN ELEMENT IN THE LOWER PART OF THE ARRAY WHICH IS NOT
SMALLER THAN T

60 k = k + 1
indx = ind(k)
IF (x(indx) < t) GO TO 60

! IF K <= L, INTERCHANGE ELEMENTS K AND L

IF (k <= l) GO TO 40

! SAVE THE UPPER AND LOWER SUBSCRIPTS OF THE PORTION OF THE
!   ARRAY YET TO BE SORTED

IF (l-i > j-k) THEN
  il(m) = i
  iu(m) = l
  i = k
  m = m + 1
  GO TO 80
END IF

il(m) = k
iu(m) = j
j = l
m = m + 1
GO TO 80

```

```

! BEGIN AGAIN ON ANOTHER UNSORTED PORTION OF THE ARRAY

70 m = m - 1
IF (m == 0) RETURN
i = il(m)
j = iu(m)

80 IF (j-i >= 11) GO TO 30
IF (i == 1) GO TO 20
i = i - 1

! SORT ELEMENTS I+1,...,J. NOTE THAT 1 <= I < J AND J-I < 11.

90 i = i + 1
IF (i == j) GO TO 70
indx = ind(i+1)
t = x(indx)
it = indx
indx = ind(i)
IF (x(indx) <= t) GO TO 90
k = i

100 ind(k+1) = ind(k)
k = k - 1
indx = ind(k)
IF (t < x(indx)) GO TO 100

ind(k+1) = it
GO TO 90
END SUBROUTINE qsortd

END PROGRAM Quicksort

```

```

C:\Arquivos de programas\Salford Software\FTN95\Plato3.exe

Algoritmo Quicksort
=====

Quantos numeros deseja ordenar? <max 100>? :
5
Escreva os numeros a ordenar:
14 89 76 94 -7
Vetor de entrada:
      14      89      76      94      -7

Vetor Ordenado pelo algoritmo Quicksort:
      -7      14      76      89      94

Press RETURN to close window . . .

```

14.4 Vetores e Matrizes

```
! Autor:.....Weyden Daniel Castro
! Data:.....05/2005
! Instituicao:...UNASP
! Curso:.....Ciencia da Computacao
! Disciplina:....Paradigmas de Linguagens de Programacao
! Professor:.....Ausberto Castro

! -----
! Este programa permite multiplicar matrizes, desde que seja
! uma multiplicacao valida (o número de colunas da primeira matriz
! deve ser igual ao numero de linhas da segunda matriz).
! -----
PROGRAM multiplicacao_matrizes
  IMPLICIT NONE

  INTEGER, PARAMETER :: MAX = 20
  INTEGER, DIMENSION(1:MAX, 1:MAX) :: X, Y, Z
  INTEGER :: linhaX, colunaX
  INTEGER :: linhaY, colunaY
  INTEGER :: linhaZ, colunaZ

  WRITE(*, "(/15X, A)") "Multiplicacao de Matrizes (MatrizA X MatrizB =
Matriz A X B)"

  WRITE(*, "(//A)") "Matriz A"
  CALL LeMatriz(X, linhaX, colunaX)
  WRITE(*, "(/A/)") "Dados inseridos na matriz A:"
  CALL MostraMatriz(X, linhaX, colunaX)

  WRITE(*, "(//A)") "Matriz B"
  CALL LeMatriz(Y, linhaY, colunaY)
  WRITE(*, "(/A/)") "Dados inseridos na matriz B:"
  CALL MostraMatriz(Y, linhaY, colunaY)
  IF(colunaX /= linhaY) THEN
    WRITE(*, *) "As matrizes nao podem ser multiplicadas."
  ELSE
    linhaZ = linhaX
    colunaZ = colunaY
    CALL Multiplicar(X, linhaX, colunaX, Y, colunaY, Z)
    WRITE(*, "(//A/)") "Matriz A*B:"
    CALL MostraMatriz(Z, linhaZ, colunaZ)
  END IF

CONTAINS
  SUBROUTINE LeMatriz(A, linha, coluna)
    IMPLICIT NONE
    INTEGER, DIMENSION(1: , 1: ), INTENT(OUT) :: A
    INTEGER, INTENT(OUT) :: linha, coluna
    INTEGER :: i, j

    WRITE(*, "(/A)") "Quantas linhas?"
    READ(*, *) linha
    WRITE(*, "(/A)") "Quantas colunas?"
```

```

        READ(*, *) coluna
        WRITE(*, "(/A/)") "Insira a matriz:"
        DO i = 1, linha
            READ(*, *) (A(i, j), j = 1, coluna)
        END DO
    END SUBROUTINE LeMatriz

    SUBROUTINE MostraMatriz(A, linha, coluna)
        IMPLICIT NONE
        INTEGER, DIMENSION(1: , 1: ), INTENT(IN) :: A
        INTEGER, INTENT(IN) :: linha, coluna
        INTEGER :: i, j

        DO i = 1, linha
            WRITE(*, *) (A(i, j), j = 1, coluna)
        END DO
    END SUBROUTINE MostraMatriz

    SUBROUTINE Multiplicar(A, linhaA, colunaA, B, colunaB, C)
        IMPLICIT NONE
        INTEGER, DIMENSION(1: , 1: ), INTENT(IN) :: A, B
        INTEGER, DIMENSION(1: , 1: ), INTENT(OUT) :: C
        INTEGER, INTENT(IN) :: linhaA, colunaA
        INTEGER, INTENT(IN) :: colunaB
        INTEGER :: soma
        INTEGER :: i, j, k

        DO i = 1, linhaA
            DO j = 1, colunaB
                soma = 0
                DO k = 1, colunaA
                    soma = soma + A(i, k) * B(k, j)
                END DO
                C(i, j) = soma
            END DO
        END DO
    END SUBROUTINE Multiplicar

END PROGRAM multiplicacao_matrizes

```

```
C:\Arquivos de programas\Salford Software\FTN95\Plato3.exe

Multiplicacao de Matrices (MatrizA X MatrizB = Matriz A X B)

Matriz A
Quantas linhas?
3
Quantas colunas?
2
Insira a matriz:
5 8
9 4
5 1
Dados inseridos na matriz A:
      5      8
      9      4
      5      1

Matriz B
Quantas linhas?
2
Quantas colunas?
4
Insira a matriz:
2 7 9 4
8 5 2 6
Dados inseridos na matriz B:
      2      7      9      4
      8      5      2      6

Matriz A*B:
      74      75      61      68
      50      83      89      60
      18      40      47      26

Press RETURN to close window . . . _
```

14.5 Problema prático

14.5.1 Equação quadrática

A seguir, apresentamos um programa que calcula as raízes da equação quadrática.

$$ax^2 + bx + c = 0$$

```

! Autor:.....Weyden Daniel Castro
! Data:.....05/2005
! Instituicao:...UNASP
! Curso:.....Ciencia da Computacao
! Disciplina:....Paradigmas de Linguagens de Programacao
! Professor:.....Ausberto Castro

! -----
! PROGRAMA Equacao Quadratica:
!
! Este programam solicita do usuario os valores de a, b e c,
! em seguida chama a rotina Resolver para obter as raizes de uma
! equacao quadratica.
! -----

PROGRAM EquacaoQuadratica
  IMPLICIT NONE

  INTEGER, PARAMETER :: NENHUMA_RAIZ = 0 ! possiveis tipos de retorno
  INTEGER, PARAMETER :: RAIZES_REPETIDAS = 1
  INTEGER, PARAMETER :: RAIZES_DIFERENTES = 2

  INTEGER :: TipoRaizes
  REAL :: a, b, c ! coeficientes
  REAL :: r1, r2 ! raizes

  WRITE(*, "(/20X, A)") "EQUACAO QUADRATICA"
  WRITE(*, "(/, A)") "De os valores de a, b e c: "
  READ(*,*) a, b, c ! le os coeficientes do usuario
  CALL Resolver(a, b, c, r1, r2, TipoRaizes) ! resolve a raiz quadratica
  SELECT CASE (TipoRaizes) ! verifica qual tipo de raizes foi
retornado
    CASE (NENHUMA_RAIZ)
      WRITE(*,*) "A equacao nao tem raizes reais."
    CASE (RAIZES_REPETIDAS)
      WRITE(*,*) "A equacao tem raizes repetidas.", r1
    CASE (RAIZES_DIFERENTES)
      WRITE(*,*) "A equacao tem duas raizes: ", r1, " e ", r2
  END SELECT

CONTAINS

! -----
! SUBROUTINE Resolver():
! Esta subrotina recebe os coeficientes de uma equacao quadratica
! e resolve a equacao. A subrotina retorna tres valores:
! -> tipo: Se a equacao nao tem raizes reais, seus argumentos formais
! retornam NENHUMA_RAIZ.
! Se a equacao tem raizes repetidas, seus argumentos formais
! retornam RAIZES_REPETIDAS.
! Se a equacao tem duas raizes diferentes, seus argumentos
! formais retornam RAIZES_DIFERENTES.
! Perceba que esses sao parametros declarados no programa
principal.

! -> raiz1 e raiz2: Se nao ha nenhuma raiz real, esses dois argumentos
formais
! retornam 0.0.
! Se as raizes sao repetidas, raiz1 retorna a raiz e raiz2
e zero.
! Se as raizes sao diferentes, raiz1 e raiz2 retornam as
raizes.
! -----

```

```

SUBROUTINE Resolver(a, b, c, raiz1, raiz2, tipo)
  IMPLICIT NONE

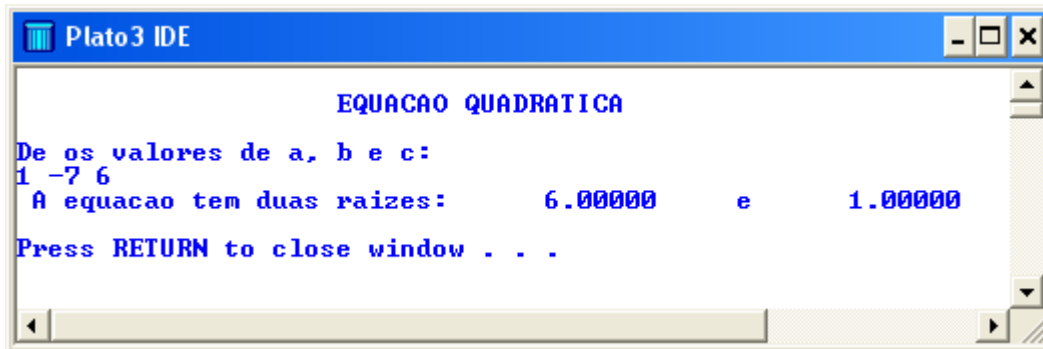
  REAL, INTENT(IN)      :: a, b, c
  REAL, INTENT(OUT)     :: raiz1, raiz2
  INTEGER, INTENT(OUT)  :: tipo

  REAL                  :: d          ! discriminante

  raiz1 = 0.0            ! inicializa as raizes para zero
  raiz2 = 0.0
  d      = b*b - 4.0*a*c    ! calcula o discriminante
  IF (d < 0.0) THEN        ! se discriminante < 0
    tipo = NENHUMA_RAIZ    ! nenhuma raiz
  ELSE IF (d == 0.0) THEN  ! se discriminante = 0
    tipo = RAIZES_REPETIDAS ! raizes repetidas
    raiz1 = -b/(2.0*a)
  ELSE                    ! senao
    tipo = RAIZES_DIFERENTES ! raizes diferentes
    d      = SQRT(d)
    raiz1 = (-b + d)/(2.0*a)
    raiz2 = (-b - d)/(2.0*a)
  END IF
END SUBROUTINE Resolver

END PROGRAM EquacaoQuadratica

```



14.5.2 Busca binária com quicksort

```

! Autor:.....Weyden Daniel Castro
! Data:.....05/2005
! Instituicao:...UNASP
! Curso:.....Ciencia da Computacao
! Disciplina:....Paradigmas de Linguagens de Programacao
! Professor:.....Ausberto Castro
! -----

! PROGRAMA Pesquisa Binaria

! Este programa procura um elemento em um array. Se ele for
! encontrado, a posicao do array em que o elemento se encontra

```



```

! e retornado.

!     O usuario da os elementos do array.
!     Para realizar a busca de um elemento, primeiro utilizamos o
! algoritmo quicksort para criar um vetor de indices, que indicam
! a posição dos elementos ordenados em forma crescente.
!     Com o vetor de indices, podemos fazer um vetor com os
! elementos ordenados em forma crescente, e usa-lo para procurar
! um elemento qualquer utilizando o algoritmo de busca binaria (ja
! que a busca binaria so e possivel com um vetor de elementos ordenados).
!     Finalmente, podemos mostrar o indice em que o elemento foi
! encontrado, utilizando o vetor de indices.
! -----

```

```

PROGRAM pesquisa_binaria
  IMPLICIT NONE
  IMPLICIT NONE
  INTEGER, PARAMETER :: TamanhoTabela = 100
  INTEGER, DIMENSION(1:TamanhoTabela) :: Tabela
  INTEGER, DIMENSION(1:TamanhoTabela) :: TabelaIndicesOrdenados
  INTEGER, DIMENSION(1:TamanhoTabela) :: TabelaOrdenada
  INTEGER :: TamanhoAtual
  INTEGER :: numero
  INTEGER :: posicao
  INTEGER :: i

  WRITE(*, "(//20X, A//)") "PESQUISA BINARIA"
  WRITE(*, "(/A//)") "Quantos dados voce quer inserir(maximo 100)?"
  READ(*,*) TamanhoAtual
  WRITE(*, "(/A//)") "Insira os dados:"
  READ(*,*) (Tabela(i), i = 1, TamanhoAtual)
  WRITE(*,*) "Tabela inserida:"
  WRITE(*,*) (Tabela(i), i = 1, TamanhoAtual)
  CALL qsortd(Tabela, TabelaIndicesOrdenados, TamanhoAtual)
  DO i = 1, TamanhoAtual
    TabelaOrdenada(i) = Tabela(TabelaIndicesOrdenados(i))
  END DO
  WRITE(*,*)
  DO
    ! procura o numero pelas iteracoes
    necessarias
    WRITE(*,*) "Que numero deseja pesquisar? "
    READ(*,*) numero
    posicao = BuscaBinaria(TabelaOrdenada, TamanhoAtual, numero)
    IF (posicao > 0) THEN
      ! mostra o resultado da pesquisa
      binaria
      posicao = TabelaIndicesOrdenados(posicao)
      WRITE(*, "(//2(A, I5))") "O numero ", numero, " foi encontrado
na posicao: ", posicao
    ELSE
      WRITE(*, "(//A, I5, A)") "O numero ", numero, " nao foi
encontrado."
    END IF
    EXIT
  END DO

  WRITE(*, "(/A)") "Operacao de busca na tabela completada."

```

CONTAINS

```
!=====
=====
!                                     SUBROUTINA QUICKSORT
!=====
=====
```

SUBROUTINE qsortd(x,ind,n)

```
IMPLICIT NONE
INTEGER, PARAMETER :: dp = SELECTED_REAL_KIND(12, 60)

!REAL (dp), INTENT(IN) :: x(:)
INTEGER, INTENT(IN) :: x(:)
INTEGER, INTENT(OUT) :: ind(:)
INTEGER, INTENT(IN) :: n

INTEGER :: iu(21), il(21)
INTEGER :: m, i, j, k, l, ij, it, itt, indx
REAL :: r
REAL (dp) :: t

IF (n <= 0) RETURN

! INITIALIZE IND, M, I, J, AND R

DO i = 1, n
    ind(i) = i
END DO
m = 1
i = 1
j = n
r = .375

! TOP OF LOOP

20 IF (i >= j) GO TO 70
IF (r <= .5898437) THEN
    r = r + .0390625
ELSE
    r = r - .21875
END IF

! INITIALIZE K

30 k = i

! SELECT A CENTRAL ELEMENT OF X AND SAVE IT IN T

ij = i + r*(j-i)
it = ind(ij)
t = x(it)

! IF THE FIRST ELEMENT OF THE ARRAY IS GREATER THAN T,
```

```

!   INTERCHANGE IT WITH T

indx = ind(i)
IF (x(indx) > t) THEN
    ind(ij) = indx
    ind(i) = it
    it = indx
    t = x(it)
END IF

! INITIALIZE L

l = j

! IF THE LAST ELEMENT OF THE ARRAY IS LESS THAN T,
!   INTERCHANGE IT WITH T

indx = ind(j)
IF (x(indx) >= t) GO TO 50
ind(ij) = indx
ind(j) = it
it = indx
t = x(it)

! IF THE FIRST ELEMENT OF THE ARRAY IS GREATER THAN T,
!   INTERCHANGE IT WITH T

indx = ind(i)
IF (x(indx) <= t) GO TO 50
ind(ij) = indx
ind(i) = it
it = indx
t = x(it)
GO TO 50

! INTERCHANGE ELEMENTS K AND L

40 itt = ind(l)
ind(l) = ind(k)
ind(k) = itt

! FIND AN ELEMENT IN THE UPPER PART OF THE ARRAY WHICH IS
!   NOT LARGER THAN T

50 l = l - 1
indx = ind(l)
IF (x(indx) > t) GO TO 50

! FIND AN ELEMENT IN THE LOWER PART OF THE ARRAY WHICH IS NOT
SMALLER THAN T

60 k = k + 1
indx = ind(k)
IF (x(indx) < t) GO TO 60

! IF K <= L, INTERCHANGE ELEMENTS K AND L

```

```

IF (k <= 1) GO TO 40

! SAVE THE UPPER AND LOWER SUBSCRIPTS OF THE PORTION OF THE
!   ARRAY YET TO BE SORTED

IF (l-i > j-k) THEN
  il(m) = i
  iu(m) = l
  i = k
  m = m + 1
  GO TO 80
END IF

il(m) = k
iu(m) = j
j = l
m = m + 1
GO TO 80

! BEGIN AGAIN ON ANOTHER UNSORTED PORTION OF THE ARRAY

70 m = m - 1
IF (m == 0) RETURN
i = il(m)
j = iu(m)

80 IF (j-i >= 11) GO TO 30
IF (i == 1) GO TO 20
i = i - 1

! SORT ELEMENTS I+1,...,J.  NOTE THAT 1 <= I < J AND J-I < 11.

90 i = i + 1
IF (i == j) GO TO 70
indx = ind(i+1)
t = x(indx)
it = indx
indx = ind(i)
IF (x(indx) <= t) GO TO 90
k = i

100 ind(k+1) = ind(k)
k = k - 1
indx = ind(k)
IF (t < x(indx)) GO TO 100

ind(k+1) = it
GO TO 90
END SUBROUTINE qsortd

! -----
! INTEGER FUNCTION  Busca Binaria():
!   Dado um array x() e um numero a ser pesquisado, esta funcao
!   determina
!   ! se o numero procurado esta no array x(). Se esta, o indice do numero
!   ! encontrado e retornado; se nao esta, 0 e retornado.

```

```

! -----

INTEGER FUNCTION BuscaBinaria(y, tamanho, num)
  IMPLICIT NONE
  INTEGER, DIMENSION(1:), INTENT(IN) :: y
  INTEGER, INTENT(IN) :: tamanho
  INTEGER, INTENT(IN) :: num
  INTEGER :: esquerda, direita, meio

  BuscaBinaria = 0 ! assumimos que o numero nao sera
  encontrado.
  esquerda = 1 ! limite minino
  direita = tamanho ! limite maximo
  DO ! loop-DO tem iteracoes ate todo o array
    inteiro for pesquisado
      IF (esquerda > direita) EXIT
      meio = (esquerda + direita) / 2
      IF (num == y(meio)) THEN ! o numero foi achado na metade do
array
        BuscaBinaria = meio ! Retorna a posicao
        EXIT ! and exit
      ELSE IF (num < y(meio)) THEN ! o numero e num < meio?
        direita = meio - 1 ! Ignore a metade da direita do
array..
      ELSE
        esquerda = meio + 1 ! ignore a metade da esquerda.
      END IF
    END DO ! volte e tente de novo
  END FUNCTION BuscaBinaria

END PROGRAM pesquisa_binaria

```

```
Plato3 IDE

PESQUISA BINARIA

Quantos dados voce quer inserir(maximo 100)?
6
Insira os dados:
24
35
89
31
74
13
Tabela inserida:
      24      35      89      31      74      13

Que numero deseja pesquisar?
31

O numero    31 foi encontrado na posicao:    4

Operacao de busca na tabela completada.
Press RETURN to close window . . .
```

Referências

[INTEL] **Intel Fortran Language Reference.** Document Number: 253261-002. World Wide Web: <<http://developer.intel.com>>. Intel Corporation, 2003-2004.

[MAN03] MANZANO, J. A. N. G. **Estudo dirigido de FORTRAN.** 1. ed. São Paulo: Érica, 2003.

[SEB03] SEBESTA, R. W. **Conceitos de Linguagens de Programação.** 5. ed. Porto Alegre: Bookman, 2003.

[WAT90] WATT, D. A. **Programming Language Concepts and Paradigms.** Edinburgh: Prentice Hall, 1990.

Páginas da internet:

SHENE C. K. **Fortran 90 Tutorial.** Disponível em:

<<http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/fortran.html>>.

The University of Liverpool. **F90 Course Development.** Disponível em:

<<http://www.liv.ac.uk/HPC/F90page.html>>.