```
Program Taylor;

var
   k, n:
   s, x:

function
   begin
      if

      els

   end;

function
   begin
      pot
   end;

begin
   write(                                  x: ');
   read(n
   s := 0
   for k:
      s :                                  );
   writeln('S= ', s);
   readkey;
end.
```

Blaise Pascal (1623-1662)

SUMÁRIO

FIGURAS

TABELAS

1. OBJETIVO DO DOCUMENTO

Apresentar um ambiente para aprendizado de programação baseado na linguagem Pascal, bem como um resumo desta linguagem. Trata-se de um material de apoio para à disciplina de Algoritmos: o leitor já deverá estar iniciado nas técnicas de construção de algoritmos e ter noções de português estruturado (ou portugol).

NÃO é objetivo apresentar a última palavra em ambiente de programação[1], e sim utilizar um software tipo freeware (acessível a todos!) com recursos suficientes e amigáveis para o objetivo principal, que é propiciar ao iniciante dar seus primeiros passos em programação.

2. INTRODUÇÃO

2.1. A Linguagem Pascal

A linguagem Pascal foi desenvolvida no início dos anos 70 por Nicklaus Wirth na Universidade Técnica de Zurique, Suíça, com o objetivo de oferecer uma linguagem para o ensino de programação que fosse simples, coerente e capaz de incentivar a confecção de programas claros e facilmente legíveis, favorecendo a utilização de boas técnicas de programação. Ela foi batizada com o nome de Pascal em homenagem a Blaise Pascal, filósofo e matemático francês que viveu entre 1623 e 1662, inventor da primeira calculadora mecânica.

O mais famoso compilador desta linguagem foi desenvolvido pela Borland International em 1983, e foi denominado Turbo Pascal. Foram lançadas várias versões deste compilador para ambiente DOS, culminando com o lançamento do Delphi[2] para programação em ambiente Windows. No Museum da Borland (http://community.borland.com/museum/) é possível obter versões antigas do Turbo Pascal gratuitamente.

2.2. Pascal ZIM!

O compilador Pascal ZIM! foi desenvolvido no Departamento de Ciências da Computação da Universidade de Brasília como resultado de pesquisas e trabalhos na área de algoritmos, tradutores e linguagens de programação. É do tipo freeware[3].

[1] Se você fosse aprender a pilotar, você acha que deveria iniciar num Boeing 777, ou num planador ou mono-motor? É por aí...

[2] Ambiente de desenvolvimento visual baseado no *Object Pascal* (Pascal orientado a objetos). No Museum da Borland há o porquê deste nome.

[3] Distribuição gratuita, não podendo ser comercializado.

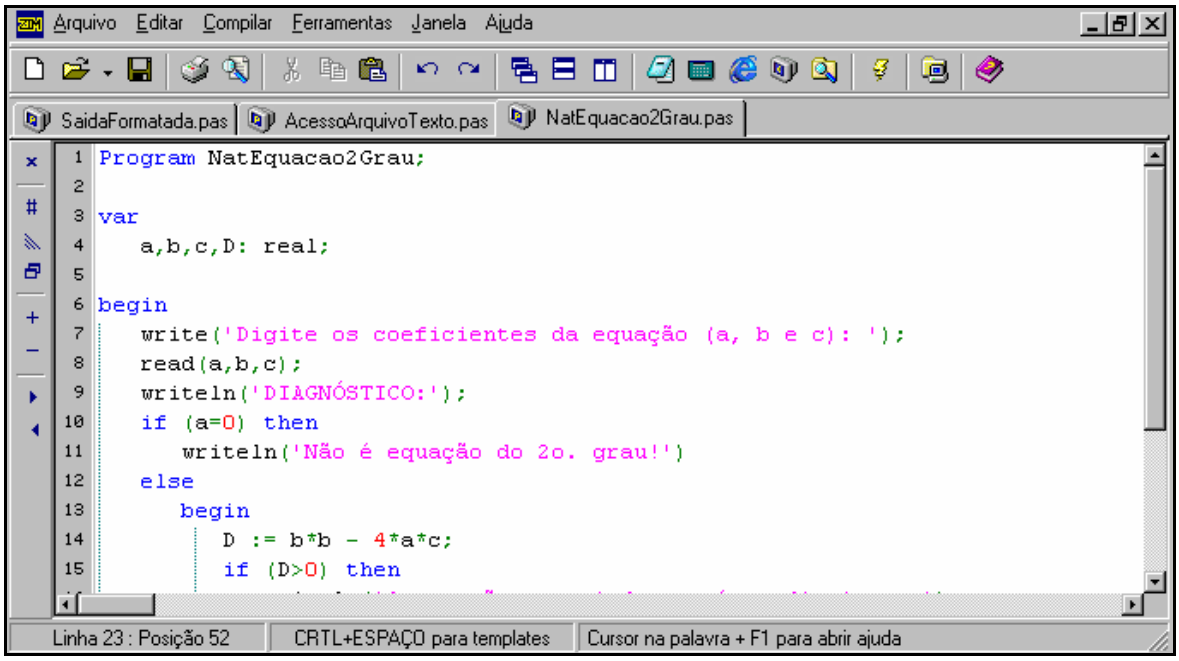


Figura 1. Visão da interface do Pascal ZIM!.

O Pascal ZIM! é adotado naquela Universidade como ferramenta de apoio ao ensino e aprendizagem de programação através da linguagem Pascal. Este compilador implementa um subconjunto desta linguagem e contém as estruturas de dados, funções e comandos mais utilizados por iniciantes. O arquivo de ajuda que acompanha o produto especifica as instruções suportadas. Este compilador pode ser obtido no site do PascalZIM!:

http://pascalzim.tripod.com/index.html

O aspecto da interface deste compilador pode ser visto na Figura 1. Notar a facilidade de se poder manter vários programas abertos simultaneamente (nesta figura são três, um por ficha, com o nome no padrão *nome_arquivo*.pas)

3. Resumo da Linguagem Pascal

3.1. Estrutura Básica de um programa Pascal

Na Figura 2 é mostrada a estrutura básica de um programa Pascal, exemplificado na Figura 3.

```
program Nomeprograma;
   Declarações
begin
   comandos
end.
```

Figura 2. Estrutura básica de um programa Pascal.

```
program Constante;

const a = 5;

begin
   writeln('Valor de a: ', a);
end.
```

Figura 3. Exemplo de um programa Pascal.

No Pascal padrão, a área de declarações é subdividida em sete sub-áreas: *uses*, *label*, *const*, *type*, *var*, *procedure* e *function*. Algumas destas sub-áreas (as suportadas pelo Pascal ZIM!) serão abordadas mais adiante.

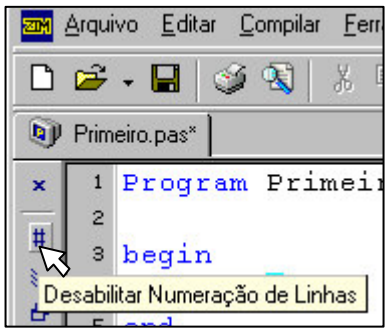
```
program Qualquer;
   Declarações
begin
   comandos
   begin
      comandos
   end;
   comandos
end.
```

Figura 4. Blocos em um programa Pascal.

O programa propriamente dito é escrito na área denominada corpo do programa, que é delimitada pelas palavras reservadas *begin* (início) e *end* (fim), seguida do ponto (.). O uso destas palavras caracteriza o que é chamado de ***bloco de comando***, sendo os blocos internos ao programa (usados nas estruturas de controle, como as de seleção e de repetição) finalizados com ponto-e-vírgula (;) - ver Figura 4.

3.2. Executando um Programa no Pascal.ZIM!

O procedimento de execução de um programa no Pascal ZIM! é bastante simples. Basta digitar o programa e clicar o botão "Executar", conforme mostrado na Figura 5. Outros botões relevantes - Salvar e Ajuda - são indicados. Para saber sobre os outros botões, basta passar o cursor do mouse sobre eles que uma pequena janela amarela será aberta indicando seu uso (figura ao lado).



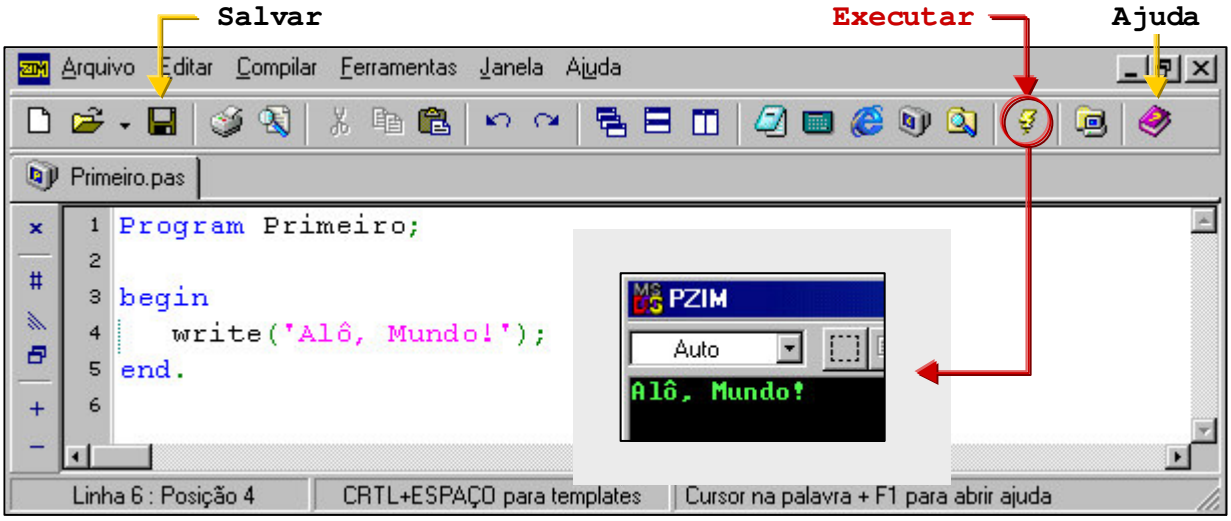


Figura 5. Primeiro programa no Pascal ZIM!: o tradicional "Alô, Mundo!".

Para o programa poder ser executado, ele deve estar sintaticamente correto (tudo escrito conforme a sintaxe da linguagem Pascal). Caso isto não ocorra, o compilador irá emitir uma mensagem avisando sobre os eventuais erros, indicando em que linha ocorreram e qual a natureza do erro. A Figura 6 mostra um exemplo disso para o primeiro programa (Figura 5), onde foi esquecido de se colocar um ponto-e-vírgula ao final do comando *write*: a sintaxe do Pascal exige um ponto-e-vírgula como terminador de comandos

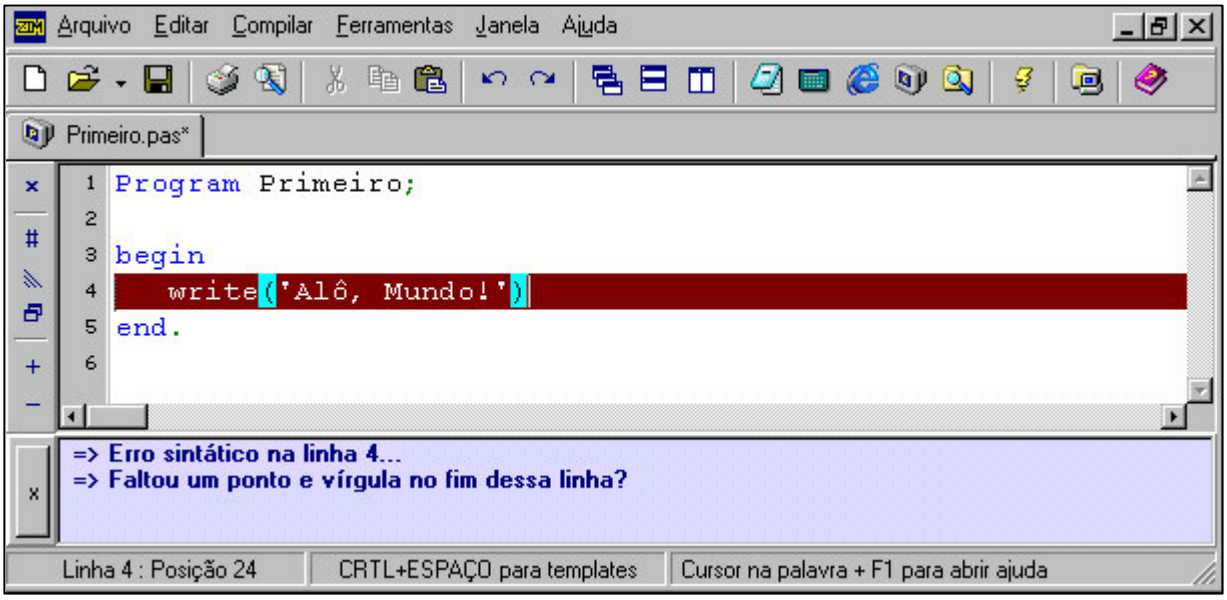


Figura 6. Mensagem de erro sintático do compilador.

3.3. Tipos de Dados Pré-definidos

Os tipos de dados predefinidos na linguagem Pascal, e implementados no compilador Pascal ZIM!, são mostrados na Tabela 1. A linguagem Pascal também permite que o usuário defina seus próprios tipos. Isto será visto mais adiante (item *3.16*).

Tipo	***Descrição***
Boolean	Define dois valores lógicos: FALSE e TRUE. Um dado do tipo booleano ocupa um byte de espaço na memória.
Char	Define os elementos do conjunto de caracteres que compõem o alfabeto ASCII, adicionados dos caracteres representados pelos códigos de 128 a 255. Um dado do tipo char ocupa um byte de espaço na memória.
Integer	Define os valores inteiros compreendidos no intervalo de -2.147.483.647 até 2.147.483.647. Um dado do tipo integer ocupa quatro bytes de espaço na memória.
Real	Define os valores reais definidos no intervalo de $3.4 \cdot 10^{-38}$ até $3.4 \cdot 10^{+38}$. Um dado do tipo real ocupa quatro bytes de espaço na memória.
String	Define uma cadeia de caracteres. Se nenhuma restrição de tamanho for especificada, um dado do tipo string é capaz de armazenar uma seqüência contendo até 255 caracteres, onde cada caracter ocupa um byte de espaço na memória. Uma cadeia de caracteres pode ter seu tamanho definido (contendo menos de 255 caracteres), onde o tamanho especifica o número máximo de caracteres contidos na cadeia. Essa especificação deve ser indicada entre colchetes, logo após a palavra reservada string,

Tabela 1. Tipos de dados pré-definidos do Pascal ZIM!

No item *3.5 Constantes e Variáveis* são apresentados alguns exemplos de declarações e usos que ilustram os tipos acima abordados.

3.4. Identificadores

Um identificador válido na linguagem Pascal é qualquer seqüência de caracteres que obedeça às seguintes regras:

- seja iniciada por um caractere alfabético (letras a,...,z, minúsculas ou maiúsculas) ;

- possuir, após o primeiro caractere, uma seqüência de caracteres que podem ser alfabéticos, numéricos (1, 2,..., 9, 0) ou o caractere _ ;
- não seja uma palavra reservada da linguagem Pascal.

Identificadores nomeiam variáveis, constantes, funções e procedimentos. Alguns exemplos podem ser vistos na Tabela 2.

Identificadores válidos	***Identificadores NÃO válidos***
A	1A
Nota	E(13)
P1	A:B
Meu_Identificador	Meu Identificador

Tabela 2. Exemplos de identificadores na linguagem Pascal.

3.5. Constantes e Variáveis

Constantes e variáveis são definidas na área de declarações, logo após as palavras *const* e *var*, respectivamente. A constante assume o tipo do valor a ela atribuído (imutável: não pode ser alterado ao longo do programa), enquanto a variável tem o seu tipo definido na declaração e pode ter seu valor alterado quando for conveniente. As respectivas declarações são[4]:

```
const identificador = valor;[...;identificador = valor;]
var identificador[,...,identificador]: tipo;
```

Na declaração acima, *tipo* é um tipo pré-definido (Tabela 1) ou definido pelo usuário (item *3.16*). Exemplos:

```
const
   a = 2;          { define uma constante inteira }
   w = 1.25;       { define uma constante real }
   teste = TRUE;   { define uma constante lógica }
   sim = 'S';      { define uma constante caractere }
   nome = 'José'; { define uma constante string }
var
   i, j: integer; { declara 2 variáveis inteiras }
   x, y, z: real; { declara 3 variáveis reais }
   flag: boolean; { declara uma variável lógica }
   letra: char;   { declara uma variável caractere }
   nome_comleto: string[80]; { declara uma string com 80 caracteres }
```

A Figura 7 mostra um exemplo de programa utilizando variáveis e constantes.

[4] Colchetes ([e]) sempre indicarão que aquilo por eles delimitado é opcional. Exceção se faz na definição de vetores/matrizes, onde os colchetes delimitam o "tamanho" destas estruturas.

```
program AreaCirculo;

const {Área de declaração de constantes}
   PI = 3.14159;

var {Área de declaração de variáveis}
   R: real;

begin
   write('Raio do círculo: ');
   read(R);
   writeln('Área do círculo: ',PI*R*R);
end.
```

RESULTADO (TELA CONSOLE)

```
Raio do círculo: 5 [enter]
Área do círculo: 78.539750
```

Figura 7. Programa com áreas de declaração para constantes e variáveis.

3.6. Comentários

Comentários são textos escritos dentro do programa (código-fonte) para explicar ou descrever alguns aspectos relativos ao mesmo. Os comentários podem ser colocados em qualquer lugar do programa, e não fazem parte dele. São colocados entre os símbolos “{“ e “} “ (chaves) ou “(*” e “*)”. Exemplos:

```
...
d := b*b - 4*a*c; { cálculo do discriminante }
...
(* Cálculo das raízes da equação *)
x1 := (-b + sqrt(d))/(2*a);
x2 := (-b - sqrt(d))/(2*a);
...
```

A Figura 7 também mostra um programa com comentários explicando onde são as sub-áreas de declaração de constantes e de variáveis.

3.7. Comando de Atribuição

O comando de atribuição simplesmente define o valor de uma variável. O símbolo de atribuição no Pascal é ":=", e a sintaxe é a seguinte:

```
identificador := expressão;
```

onde o identificador e o resultado da expressão devem ser de tipos compatíveis. Exemplos podem ser vistos na Figura 8.

```
program Atribuicao;

var
   A, B: integer;
   C, D: real;
   T, F: boolean;
   Cadeia: string;

begin
   A := 1;
   B := A + 3;
   C := 2.3;
   D := C/3 + A + B;
   T := (A > 2);
   F := false;
   cadeia := 'Um texto' ;
end.
```

Figura 8. Programa com múltiplas atribuições.

3.8. Operadores

A Tabela 3 resume os operadores do PascalZIM! utilizados nos diversos tipos de expressões.

Tipo	*Precedência*	*Operador*	*Símbolo*
Geral	0	parênteses	( )
	0	funções	f(x)
Aritmético	1	Multiplicação	*
	1	Divisão Real	/
	1	Divisão Inteira (truncada)	div
	1	Resto da Divisão Inteira	mod
	2	Subtração	-
	2	Adição	+
Relacional	3	Maior que	>
	3	Menor que	<
	3	Maior ou igual	>=
	3	Menor ou igual	<=
	3	Igual	=
	3	Diferente	<>
Lógico (boolean)	4	Não	not
	5	e	and
	6	ou	or

Tabela 3. Operadores do Pascal ZIM!.

Notar que não há o operador de potenciação. Esta operação deve ser implementada através de uma função utilizando a seguinte relação: $x^y = e^{y \cdot \ln(x)}$. Na linguagem Pascal, as funções exp(x) (isto é, e^x) e ln(x) são definidas. Para calcular x^2, pode usar sqr(x). Para $x^{1/2}$ existe sqrt(x).

Existe um operador utilizado com o tipo string, denominado **operador de concatenação**, (símbolo +) que efetua a junção de duas variáveis ou constantes do tipo string. A Figura 9 mostra um programa onde houve a concatenação de strings.

```
program Concatena;

var
   PreNome, SobreNome, NomeCompleto: string[30];

begin
   { Seja o nome Alberto Roberto Silva }
   PreNome := 'Alberto Roberto ';
   SobreNome := 'Silva';
   NomeCompleto := PreNome + SobreNome;
   writeln(NomeCompleto);
end.
```

RESULTADO

```
Alberto Roberto Silva
```

Figura 9. Programa com concatenação de strings.

3.9. Funções Pré-definidas

O compilador Pascal ZIM! implementa as seguintes funções: Abs, ArcTan, Chr, Cos, Eof, Exp, Ln, Length, Odd, Ord, Pred, Random, Readkey, Round, Sin, Sqr, Sqrt, Succ, Trunc, Upcase. A função Help (tópicos de ajuda) do compilador fornece a descrição destas funções.

3.10. Saída de Dados

Utilizam-se os comandos *write* e *writeln*, conforme a sintaxe abaixo:

```
write(expressão[,...,expressão]);
writeln(expressão[,...,expressão]);
```

A diferença entre um e outro é que o segundo comando faz o cursor passar para a próxima linha. A Figura 10 ilustra o uso destes comandos.

```
program Escreva;

const a = 5; b = 4; c = 7;

begin
   write(a,' e ');
   writeln(b);
   write('b - c = ', b - c);
   writeln(' e b + c = ', b + c);
end.
```

RESULTADO

```
5 e 4
b - c = -3 e b + c = 11
```

Figura 10. Programa com comandos de saída.

Em se tratando de números, há a possibilidade da **saída formatada**, que define o tamanho total do campo numérico (incluindo o ponto), bem como a quantidade de casas decimais. O programa da Figura 11 ilustra como apode ser feita formatação da saída para números. Neste programa, a função dos colchetes é unicamente para indicar o tamanho do campo. A sintaxe da formatação é:

```
indentificador:tamanho_do_campo:casas_decimais
```

```
Program SaidaFormatada;

const
   a = 12.431;
   b = 3;

begin
   writeln('a=[',a:8:1,']');
   writeln('a=[',a:8:2,']');
   writeln('a=[',a:8:3,']');
   writeln('a=[',a:10:3,']');
   writeln('b=[', b:3,']');
   writeln('b=[', b:4,']');
   writeln('b=[', b:5,']');
   readkey; {"Pausa"}
end.
```

RESULTADO

```
a=[    12.4]
a=[   12.43]
a=[   12.431]
a=[     12.431]
b=[  3]
b=[   3]
b=[    3]
```

Figura 11. Programa com comandos de saída formatada .

3.11. Entrada de Dados

Utilizam-se os comandos *read* e *readln*, conforme a sintaxe abaixo:

```
read(variável[,...,variável]);
readln(variável[,...,variável]);
```

A diferença entre um e outro é que o segundo comando obriga que a próxima leitura de dados seja feita na linha seguinte[5]. O programa mostrado na Figura 12 só funcionará se, após fornecido o primeiro valor (4), for pressionada a tecla enter, que obriga o cursor a passar para a próxima linha, onde o segundo valor (5) deve ser digitado.

```
program Leia;

var
   a,b,x,y: integer;

begin
   writeln('Digite x e y:');
   read(x);
   read(y);
   writeln('Valores: x=',x,' e y=',y);
   writeln; { Pula 1 linha }
   writeln('Digite a e b:');
   read(a,b);
   writeln('Valores: a=',a,' e b=',b);
end.
```

RESULTADO

```
Digite x e y:
4 [enter]
5 [enter]
Valores: x=4 e y=5

Digite a e b:
7 8 [enter]
Valores: a=7 e b=8
```

Figura 12. Programa com comandos de entrada e saída.

3.12. Estruturas de Seleção

3.12.1. Alternativa Simples: *if...then*

A sintaxe é

[5] Esta característica é melhor visível na leitura a partir de arquivos. Se for utilizado o comando *read*, os dados lidos podem ser digitados na mesma linha; com o *readln*, cada dado deve estar em uma linha.

```
if expressão then
  bloco_verdade;
```

onde expressão resulta em um valor lógico (true ou false - .V. ou .F.). Exemplo:

```
...
if (x > 10) then
 writeln('O valor da variável X é maior que 10');
...
```

3.12.2. Alternativa Composta: *if...then...else*

A sintaxe é

```
if expressão then
  bloco_verdade
else
  bloco_falsidade;
```

onde expressão resulta em um valor lógico (true ou false - .V. ou .F.).

Na Figura 13 é possível ver um programa onde este tipo de comando é utilizado. Trata-se do programa que analisa a natureza da solução de uma equação de 2º grau na forma $ax^2 + bx + c = 0$.

```
Program NatEquacao2Grau;

var
   a,b,c,D: real;

begin
   write('Digite os coeficientes da equação (a, b e c): ');
   read(a,b,c);
   writeln('DIAGNÓSTICO:');
   if (a=0) then
      writeln('Não é equação do 2o. grau!')
   else
      begin
         D := b*b - 4*a*c;
         if (D>0) then
            writeln('A equação possui duas raízes distintas.')
         else if (D=0) then
                 writeln('A equação possui duas raízes iguais.')
              else
                  writeln('A equação não possui raízes reais.');
      end;
   readkey; {espera o usuário pressionar uma tecla - "Pausa"}
end.
```

```
RESULTADO
Digite os coeficientes da equação (a, b e c): 1 3 2 [enter]
DIAGNÓSTICO:
A equação possui duas raízes distintas.
```

Figura 13. Programa com alternativa composta *if...then...else*.

3.12.3. Seleção Múltipla: *case...else...*

A sintaxe é

```
case seletor of
  lista_de_constantes: bloco;
  ...
  lista_de_constantes: bloco;
  else bloco;
end;
```

onde seletor é uma expressão do tipo inteiro ou caractere. A Figura 14 mostra um exemplo de programa utilizando este tipo de seleção.

```
program SelecaoMultipla;

var
   opcao: integer ;

begin
   writeln('[Opcao 1] [Opcao 2] [Opcao 3]');
   write('Escolha uma opcao: ');
   read(opcao);
   { escolha da opcao }
   case opcao of
      1: writeln('Você escolheu a opção 1...');
      2: writeln('Você escolheu a opção 2...');
      3: writeln('Você escolheu a opção 3...');
      else writeln('Você escolheu uma opção diferente de 1, 2, 3...');
   end;
end.
```

RESULTADO

```
[Opcao 1] [Opcao 2] [Opcao 3]
Escolha uma opção: 3 [enter]
Você escolheu a opção 3...
```

Figura 14. Programa com comando de seleção múltipla *case...of*.

3.13. Estruturas de Repetição

3.13.1. Controlada no Início: *while...do*

A sintaxe é

```
while condição do
  bloco;
```

O programa da Figura 15 é um exemplo de um programa utilizando esta estrutura de repetição. Este programa calcula o somatório de todos os números não nulos fornecidos; o zero é utilizado para encerrar o conjunto.

3.13.2. Controlada no Final: *repeat...until*

A sintaxe é

```
repeat
  comandos
until condição;
```

O programa da Figura 16 é um exemplo de um programa utilizando esta estrutura de repetição. Este programa calcula o quadrado de cinco números quaisquer fornecidos pelo usuário.

```
program Enquanto;

var
   x, s: real;

begin
   s:=0;
   writeln('Digite os valores (valor+[enter]):');
   read(x);
   while (x<>0) do
      begin
         s := s + x;
         read(x);
      end;
   writeln;
   writeln('A soma dos números é ',s);
   readkey; {espera o usuário pressionar uma tecla - "Pausa"}
end.
```

```
RESULTADO
Digite os valores (valor+[enter]):
3 [enter]
4 [enter]
6 [enter]
0 [enter]

A soma dos números é 13.000000
```

Figura 15. Programa utilizando comando *while...do*.

```
program Repita;

var
   i: integer;
   x: real;

begin
   i := 1;
   repeat
      write('Entre com x: ');
      read(x);
      writeln('x^2 = ', x*x);
      i := i + 1;
      writeln;
   until(i>5);
end.
```

```
RESULTADO
Entre com x: 1 [enter]
x^2 = 1.000000

Entre com x: 2 [enter]
x^2 = 4.000000

Entre com x: 3 [enter]
x^2 = 9.000000

Entre com x: 4 [enter]
x^2 = 16.000000

Entre com x: 5 [enter]
x^2 = 25.000000
```

Figura 16. Programa utilizando comando *repeat...until*.

3.13.3. Com Variável de Controle: *for...do*

A sintaxe é

```
for contador := ValorInicial to ValorFinal do
  bloco;
```

se *ValorInicial* < *ValorFinal*, *contador* aumentando de 1 em 1, ou

```
for contador := ValorInicial downto ValorFinal do
  bloco;
```

se *ValorInicial* > *ValorFinal*, *contador* diminuindo de 1 em 1.

O programa da Figura 17 realiza o somatório S = 1 + 2 + 3 + .. + N, dado N, utilizando este comando.

```
program Para;

var
   k, n, s: integer;

begin
   write('Digite n: ');
   readln(n);
   s := 0;
   for k := 1 to n do {ou for k := n downto 1 do}
      s := s + k;
   writeln;
   writeln('Somatório: ', S);
end.
```

RESULTADO

```
Digite n: 5 [enter]

Somatório: 15
```

Figura 17. Programa utilizando comando *for...do*.

3.14. Estruturas de Dados Homogêneas

Estruturas de dados homogêneas são tipos de dados que permitem agrupar em uma mesma estrutura variáveis do mesmo tipo básico.

3.14.1. Vetores

No caso de *vetores*, cada elemento possui um índice (unidimensional) que o diferencia dos outros elementos. A sintaxe da declaração de vetor é da forma:

```
var identificador[,..,identificador]: array[1..tamanho] of tipo_var;
```

O programa da Figura 18 permite a entrada de nomes e idades, e logo após imprime todos os nomes e idades contidos na memória através do uso de vetores.

```
program ExemploVetor;

const
   MAX = 3; { Quantidade máxima de nomes/idades a serem lidos }

var
   Nome: array[1..MAX] of string[30];
   Idade: array[1.. MAX] of integer;
   i: integer;

begin
   i := 1;
   repeat {repete até terminar a lista de nomes/idades}
      write('Nome: ');
      read(Nome[i]);  { entrada de um nome }
      write('Idade: ');
      read(Idade[i]);  { entrada de uma idade }
      i := i + 1;
   until (i>MAX);
   writeln('------------------------------');
   for i := 1 to MAX do
      begin
         write('Nome: ',Nome[i]);
         writeln(' - Idade: ',Idade[i]);
         readkey; {"Pausa": faz o programa imprimir um nome por vez}
      end;
end.
```

```
RESULTADO
Nome: Zeca  [enter]
Idade: 33   [enter]
Nome: Paulo [enter]
Idade: 43   [enter]
Nome: Maria [enter]
Idade: 19   [enter]
------------------------------
Nome: Zeca - Idade: 33  [enter]
Nome: Paulo - Idade: 43 [enter]
Nome: Maria - Idade: 19
```

Figura 18. Programa utilizando vetores.

3.14.2. Matrizes

Também são tipos de dados que permitem agrupar em uma mesma estrutura variáveis do mesmo tipo básico, sendo que cada elemento possui agora dois índices (bidimensional) que o diferencia dos outros elementos. A sintaxe da declaração da matriz é da forma:

```
var ident[,..,ident]: array[1..tam1,1..tam2] of tipo_var;
```

onde *ident* é o identificador da matriz, e *tam1* e *tam2* correspondem ao número de linhas e de colunas dela, respectivamente.

O programa da Figura 19 apresenta um exemplo do uso de matrizes. Este programa gera uma matriz quadrada onde a diagonal principal vale um, e os valores das diagonais adjacentes crescem em valores proporcionais à distância da diagonal principal.

```
program ExemploMatriz;

const
   MAX = 9; {Máxima ordem da matriz quadrada}

var
   M: array[1..MAX,1..MAX] of integer;
   i,j,n: integer;

begin
   write('Digite a ordem da matriz: ');
   read(n);
   for i:=1 to n do
      for j:=1 to n do
         M[i,j]:= abs(i-j)+1;
   writeln('Matriz: ');
   for i := 1 to n do
      begin
         for j:=1 to n do
           write(M[i,j]:2); {define tamanho do campo em 2}
         writeln;
      end;
end.
```

```
RESULTADO
Digite a ordem da matriz: 4 [enter]
Matriz:
 1 2 3 4
 2 1 2 3
 3 2 1 2
 4 3 2 1
```

Figura 19. Programa utilizando matrizes.

3.14.3. Vetores Multidimensionais

Um vetor é um vetor unidimensional; uma matriz é um vetor bidimensional. Vetores de mais dimensões são possíveis (como “tensores”), e a sintaxe da declaração é:

```
var ident[,..,ident]: array[1..tam1,..,tamn] of tipo_var;
```

Quanto a sua utilização, é semelhante ao de vetores e matrizes, alterando apenas a quantidade de dimensões/índices para ter acesso a cada elemento.

3.15. Estruturas de Dados Heterogêneas

Quando é necessário trabalhar com mais de um tipo de dado em uma mesma estrutura, utilizamos o *registro*. Por esta razão, este tipo de dado é considerado heterogêneo. As variáveis constituintes de um registro são denominadas campos. Em Pascal, os tipos registro devem ser declarados antes das definições das variáveis através da instrução *type* (visto no item *3.16 Tipos Definidos pelo Usuário*) em conjunto com a palavra reservada *record* (registro). A sintaxe é:

```
type
  identificador_registro = record
      <declaração dos campos>
  end;
var
  identificador_variavel: identificador_registro;
```

```
program ExemploRegistro;

type
   cadastro_aluno = record
      Nome: string;
      Nota1: real;
      Nota2: real;
   end;

var
   aluno: cadastro_aluno;

begin
   writeln('Cadastro de Aluno');
   writeln;
   write('Informe o nome...........: '); read(aluno.nome);
   write('Informe a primeira nota..: '); read(aluno.nota1);
   write('Informe a segunda nota...: '); read(aluno.nota2);
   writeln;
   writeln('Nome...: ',aluno.nome);
   writeln('Nota 1.: ',aluno.nota1:3:1);
   writeln('Nota 2.: ',aluno.nota2:3:1);
   writeln('Tecle [enter] para encerrar...'); readln; {"Pausa"}
end.
```

RESULTADO

```
Cadastro de Aluno

Informe o nome...........: Pedro [enter]
Informe a primeira nota..: 6.4   [enter]
Informe a segunda nota...: 8.3   [enter]

Nome...: Pedro
Nota 1.: 6.4
Nota 2.: 8.3
Tecle [enter] para encerrar...
```

Figura 20. Programa utilizando registro.

3.16. Tipos Definidos pelo Usuário

No item *3.15 Estruturas de Dados Heterogêneas* utilizamos a palavra reservada *type* para definirmos o tipo registro, para depois então definir uma variável registro.

Na verdade, a palavra reservada *type* define uma sub-área na área de declarações (rever Figura 2) onde novos tipos de dados podem ser criados pelo usuário. A palavra reservada *type* deve aparecer uma única vez dentro da área da declaração de dados, e a sub-área por ela definida deve estar antes da sub-área de declaração de variáveis (*var*). A sintaxe é:

```
type
  identificador_tipo = definição_novo_tipo;
```

onde *definição_novo_tipo* é um dos tipos estruturados vetor (multidemensional), registro, ponteiro ou outro tipo de dados simples. Exemplos

```
type
  intList = array[1..100] of integer ;
  matrix = array[0..9, 0..9] of real ;
  pInt = ^integer; { ponteiro para inteiro }
```

3.17. Módulos

3.17.1. Funções

Uma função é um bloco de programa no qual são válidas todas as regras de programação vistas. Uma função sempre retorna um valor (ao contrário da procedure, vista a seguir), que é utilizado exatamente no ponto onde a função é chamada no bloco de origem. O valor de retorno pode ser numérico, lógico ou literal. Sua sintaxe é

```
function id_funcao [(par[..,par]:tipo[;..;par[..,par]:tipo])];
var
   Declarações
begin
   Comandos
end;
```

A Figura 21 mostra um exemplo do uso de uma função.

```
Program ExemploFuncao;

var
   a, b: real;

function Hipotenusa(x:real; y:real):real;
   Begin
      Hipotenusa := sqrt(x*x + y*y);
   End;

begin
   write('Digite os valores dos catetos: ');
   read(a,b);
   writeln('A hipotenusa é ',Hipotenusa(a,b):5:2);
   readkey; {espera o usuário pressionar uma tecla - "Pausa"}
end.
```

```
RESULTADO
Digite os valores dos catetos: 3 4 [enter]
A hipotenusa é  5.00
```

Figura 21. Programa utilizando função.

3.17.2. Procedimentos

A diferença básica entre um procedimento e uma função é o fato do procedimento não possuir valor de retorno. Sua sintaxe é:

```
procedure id_procedimento [(par[..,par]:tipo[;..;par[..,par]:tipo])];
var
   Declarações
begin
   Comandos
end;
```

A Figura 22 mostra um exemplo do uso de um procedimento função.

```
Program ExemploProcedimento;

var
   a, b: real;

procedure CalculaHipotenusa(x:real; x:real);
   Begin
      writeln('A hipotenusa é ',sqrt(x*x + y*y):5:2);
   End;

begin
   write('Digite os valores dos catetos: ');
   read(a,b);
   CalculaHipotenusa(a,b);
   readkey; {espera o usuário pressionar uma tecla - "Pausa"}
end.
```

```
RESULTADO
Digite os valores dos catetos: 3 4 [enter]
A hipotenusa é  5.00
```

Figura 22. Programa utilizando procedimento.

Tanto as funções como os procedimentos devem ser colocados após todas as outras sub-áreas (como *type*, *var* e *const*) e antes do bloco do programa.

3.17.3. Passagem de Parâmetros

3.17.3.1. Por Valor

Nos programas da Figura 21 e Figura 22 a passagem dos parâmetros a e b foi realizada por valor: apenas seus valores foram copiados paras os argumentos x e y, respectivamente. Neste caso, não há possibilidade de se alterar o valor deste parâmetros

3.17.3.2. Por Referência

Quando o objetivo é alterar o valor de algum parâmetro, a passagem por referência é indicado. Para isto, deve ser colocado na frente dos argumentos da função ou procedimento a palavra reservada *var*. O programa da Figura 23 exemplifica a passagem por referência. Nele, a função *incrementa* altera o valor do argumento n (k passado por referência) por um valor d (parâmetro passado por valor, já que não será alterado).

```
Program PassaParametros;

var
   k: integer;

procedure incrementa(var n: integer; d: integer);
   begin
      n := n + d;
   end;

begin
   k := 1;
   writeln('k = ',k);
   incrementa(k,4);
   writeln('k = ',k);
   incrementa(k,-3);
   writeln('k = ',k);
   readkey; {"Pausa"}
end.
```

RESULTADO

```
k = 1
k = 5
k = 2
```

Figura 23. Programa utilizando procedimento com passagem de parâmetro por referência.

3.18. Escopo de Variáveis

A linguagem Pascal oferece as facilidades necessárias no tocante à modularização de programas, por meio de procedimentos e funções, como visto no item 3.17. Esses módulos podem ter variáveis próprias, ou utilizar as variáveis declaradas no programa principal.

É possível a declaração de variáveis com o mesmo identificador em módulos diferentes de mesmo nível (dois módulo distintos no mesmo programa), ou em módulos aninhados (um módulo dentro do outro). Conforme o contexto em que está inserida, uma variável pode ser considerada uma variável local ou um variável global. Por exemplo, as variáveis declaradas em um módulo A são consideradas locais à A, porém são consideradas variáveis globais aos sub-módulos contidos em A. Dessa forma, todas as variáveis declaradas no programa principal são consideradas globais às suas funções e procedimentos.

Variáveis locais com o mesmo identificador declaradas em módulos diferentes e no mesmo nível são invisíveis umas para o outras, ou seja, não causam conflito. Quando os módulos estão aninhados, as variáveis declaradas em cada módulo podem ser vistas e/ou utilizadas pelos respectivos sub-módulos. Porém, não serão utilizadas se forem declaradas variáveis com o mesmo identificador em seus sub-módulos, onde valerá somente as variáveis locais. Ou seja, se forem declaradas variáveis locais em um módulo A com o mesmo identificador que as variáveis globais à A, valerão as variáveis locais. Diz-se que a variável global de mesmo nome torna-se invisível dentro do módulo A.

O discutido acima define o que se denominam regras de escopo (onde "existe" a variável) e visibilidade das variáveis, ou seja, até onde as variáveis podem ser utilizadas e/ou visíveis a outros módulos. No caso de existirem variáveis locais e globais com o mesmo nome, alterações feitas nas variáveis locais não afetam as globais. Já no caso onde uma variável global é modificada, a próxima instrução que a acessar irá encontrar o valor dessa última atualização.

```
program A;
{ declaração de variáveis }
var
   m, n : real;

procedure B;
   { declaração de variáveis }
   var
      i, j : integer;
   begin
      { corpo do procedimento B }
   end;

procedure C;
   { declaração de variáveis }
   var
      i, j : integer;
      k, l : real;

   procedure D;
      { declaração de variáveis }
      var
         k, l : integer;
      begin
         { corpo do procedimento D }
      end;

   begin
      { corpo do procedimento C }
   end;

begin
   { corpo do programa principal }
end.
```

- As variáveis i e j, declaradas nos procedimentos B e C, são invisíveis entre si, e portanto não causam conflito.
- As variáveis k e l são válidas como reais em C. Apesar de serem globais, são invisíveis a D, pois foram redeclaradas como tipo inteiro. É como se k e l fossem declaradas com outros identificadores (internamente ao computador, é isto que acontece; o compilador trata disso).
- As variáveis i e j declaradas em C são globais a D, assim como as variáveis m e n são globais a todos os procedimentos.

Figura 24. Exemplo de escopo e visibilidade de variáveis.

3.19. Arquivos

O Pascal ZIM! só trabalha com arquivos texto e acesso seqüencial. O uso de arquivos texto na linguagem Pascal é feito através da definição de um tipo especial, TEXT, que identifica uma variável do tipo arquivo, definida na sub-área para declaração de variáveis.

Os comandos para tratamento de arquivos implementados no compilador são descritos a seguir.

3.19.1. Procedimento *Assign*

Um arquivo do tipo texto é referenciado dentro de um programa por uma variável do tipo TEXT. As operações de leitura e escrita em arquivo tomam como argumento essa variável. Assim, para

trabalhar com um arquivo texto deve-se criar uma associação entre a variável TEXT (variável arquivo) e o arquivo armazenado. Essa associação é feita através do comando *assign*, cuja sintaxe é

```
assign(variavel_arquivo, nome_arquivo);
```

onde *variavel_arquivo* é uma variável do tipo TEXT e *nome_arquivo* é uma cadeia de caracteres contendo o nome do arquivo associado, ou ainda uma variável do tipo string. Exemplo:

```
assign(arq,'c:\dados.txt');
```

O nome do arquivo externo pode ser definido por uma variável do tipo string, cujo valor pode ser determinado durante a execução do programa.

3.19.2. Procedimento *Reset*

O comando *reset* abre um arquivo já existente, posicionando o cursor de leitura no seu início. A sintaxe é:

```
reset(variavel_arquivo);
```

3.19.3. Procedimento *Rewrite*

O comando *rewrite* é usado para criar um arquivo do tipo texto ou, se ele já existir, para apagá-lo e criar um novo arquivo, posicionando o cursor de leitura no seu início. A sintaxe é

```
rewrite(variavel_arquivo);
```

onde variavelArquivo é uma variável do tipo TEXT.

3.19.4. Procedimento *Append*

Abre um arquivo já existente para escrita no final. A sintaxe é

```
append(variavel_arquivo);
```

onde *variavel_arquivo* é uma variável definida com o tipo TEXT.

3.19.5. Procedimento *Close*

Fecha um arquivo, salvando as alterações. A sintaxe é

```
close(variavel_arquivo);
```

onde *variavel_arquivo* é uma variável definida com o tipo TEXT.

3.19.6. Procedimentos *writeln* e *readln*

Os procedimentos *writeln* (*write*) e *readln* (*read*) podem ser utilizados para escrita/leitura em arquivos texto. Na Figura 25 pode-se ver um exemplo do uso de todos os procedimentos acima citados, mais os procedimentos de leitura e escrita.

```
Program AcessoArquivo;

const
   path = 'C:\temp\TEXTO.TXT';

var
   arq: Text;
   texto: string;

begin
   assign(arq,path) ;
   rewrite(arq); {Cria o arquivo}
   reset(arq);   {Abre o arquivo}
   writeln(arq,'Texto inicial');
   close(arq);   {Fecha o arquivo, salvando as alterações efetuadas}
   append(arq);  {Abre o arquivo para adicionar mais texto no seu final}
   writeln(arq,'Mais texto!') ;
   close(arq);   {Fecha o arquivo, salvando as alterações efetuadas}
   reset(arq);   {Abre o arquivo}
   while not eof(arq) do  { Lê todas as linhas até o final do arquivo}
      begin
         readln(arq,texto); {Lê linha do arquivo}
         writeln(texto);    {Escreve a linha lida na tela}
      end;
   close(arq);
end.
```

RESULTADO

Na tela:

No arquivo:

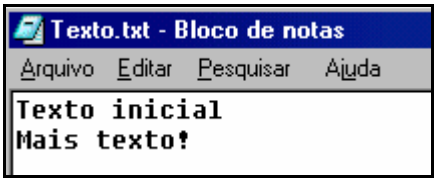


Figura 25. Programa utilizando procedimentos de acesso a arquivos.

4. Conclusão

4.1. Comentário Final

Como apresentado no início deste documento, o objetivo do presente material é apresentar um ambiente onde o iniciante em programação possa dar seus primeiros passos na construção e implementação de algoritmos. Neste aspecto, o uso da linguagem Pascal é bastante adequado, uma vez que foi criada originalmente com este objetivo. O compilador sugerido - Pascal ZIM! - implementa apenas parte dos recursos do Pascal padrão, mas suficientes para os objetivos deste trabalho. Aquele que quiser ir mais longe e utilizar um compilador Pascal completo pode encontrá-lo na Internet (item a seguir).

4.2. Compiladores Gratuitos

Existem vários compiladores da linguagem Pascal disponíveis gratuitamente na Internet. Alguns deles são:

- Free Pascal (Windows/Linux): http://www.freepascal.org/download.html.

- Dev-Pascal (Windows): http://www.bloodshed.net/devpascal.html.
- Borland Turbo-Pascal 7.01-FR (Francesa - DOS): http://www.inprise.fr/download/compilateurs/.
- Virtual Pascal 2.1: http://www.vpascal.com/. Este compilador é uma versão com "interface DOS para Windows" bastante semelhante ao Turbo-Pascal 7.0. É bem documentado. Vale à pena.
- Projeto Lazarus (Windows/Linux): http://www.lazarus.freepascal.org. É a tentativa de fazer um "Delphi" usando o Free Pascal. Apesar de ainda estar num estágio beta, é no mínimo interessante.

A descrição resumida dos softwares acima (e outros) podem ser encontrada no seguinte site:
http://www.thefreecountry.com/compilers/pascal.shtml

5. REFERÊNCIAS

1. Gottfried, B. S.: *Programação em Pascal*. Editora McGraw-Hill de Portugal Ltda, Lisboa, 1988.
2. Função Ajuda do Compilador Pascal ZIM!.