Guia rápido de referência da linguagem *Pascal*
Versão *Free Pascal*

Marcos Castilho Everaldo Gomes José Ivan Gonçalves Júnior
Loirto Alves dos Santos Rene Kultz
Eleandro Maschio Krynski Marcos Castilho

Versão 0.2
Dezembro de 2009

Este texto está em construção.
A versão atual pode ser encontrada em:
`http://www.inf.ufpr.br/cursos/ci055`.

Sumário

1 Introdução

Este texto é um material complementar à disciplina *CI055 - Algoritmos e Estruturas de Dados I*, ministrada no curso de Bacharelado em Ciência da Computação da Universidade Federal do Paraná.

O presente documento foi concebido com a finalidade de servir como um guia rápido de referência da linguagem *Pascal*, versão *Free Pascal*, destinado ao estudante no início do curso. Portanto, não dispensa o acompanhamento de material didático apropriado para o aprendizado de algoritmos. Definitivamente, este material não constitui por si só um curso de programação em *Pascal*.

Desta forma, recomenda-se consulta à farta literatura que apresenta em profundidade aspectos semânticos dos elementos da linguagem aqui abordados. Os estudantes são encorajados a ter por costume buscar outras fontes e conquistar autonomia para enfrentar situações das mais diversas em programação de computadores.

O material didático primário do curso é indicado na página oficial da disciplina[1] e inclui notas de aula[2], bem como demais referências bibliográficas pertinentes. Algumas das principais referências são: [Wir78], [VC07], [Car82],[Gui].

A documentação completa da linguagem pode ser encontrada juntamente com o compilador escolhido para este curso[3] (*Free Pascal*). Supõe-se que os estudantes tenham contato e estudem minimamente:

Guia de referência da linguagem: é uma referência para a linguagem *Pascal* tal como implementada pelo citado compilador. Descreve as construções permitidas e lista os tipos de dados suportados. Não é um tutorial.

Guia do programador: descreve as peculiaridades do compilador *Free Pascal* e provê uma visão de como o compilador gera o código e de como o programador pode modificar este código.

Guia do usuário: descreve o processo de instalação e uso do compilador *Free Pascal* nas diferentes plataformas suportadas.

Guia das bibliotecas: descreve as bibliotecas auxiliares que não estão disponíveis na versão básica do compilador. Serve para problemas que envolvem, por exemplo, bibliotecas gráficas, funções matemáticas complicadas, e uma série de outras possibilidades.

Aprenda *Pascal*: tutorial bastante didático, com exemplos e escrito em inglês de fácil compreensão[4]. Altamente recomendado.

O manual presente foi idealizado como um guia rápido, em português, para o material citado. A página oficial da disciplina, anteriormente mencionada, detém contatos atualizados para a notificação de possíveis erros, assim como para o recebimento de críticas e sugestões acerca deste guia.

[1] `http://www.inf.ufpr.br/cursos/ci055`
[2] `http://www.inf.ufpr.br/cursos/ci055/apostila.pdf`
[3] `http://www.freepascal.org`
[4] `http://www.taoyue.com/tutorials/pascal/contents.html`

2 Breve histórico da linguagem

A linguagem *Pascal* foi desenvolvida em 1968 pelo professor Niklaus Wirth, do Instituto de informática da ETH (*Eidgenössische Technische Hochschule*), em Zurique, Suíça. Sua denominação é uma homenagem a Blaise Pascal (1623-1662), matemático e filósofo francês que inventou a primeira calculadora mecânica. Foi baseada em algumas linguagens estruturadas existentes até então, como o ALGOL e PL/I.

O desejo de Wirth era dispor, para o ensino de programação, de uma nova linguagem que fosse simples, coerente e capaz de incentivar a confecção de programas claros e facilmente legíveis, favorecendo a utilização de boas técnicas de programação. Além disso, a linguagem deveria ser de fácil implementação, ou seja, os compiladores precisariam ser compactos, eficientes e econômicos [FBF+86].

O *Pascal* somente ganhou popularidade quando foi adotado pela Universidade da Califórnia (San Diego) em 1973. No mesmo período, em seus cursos, também foram feitas implementações de compiladores para mini e microcomputadores [Rin92].

Na década de 80, começou a ser utilizada para propósitos comerciais com o lançamento do *Turbo Pascal* pela empresa americana *Borland International*. Posteriormente, a linguagem foi sucedida pela linguagem *Object Pascal*, utilizadas nas IDEs *Borland Delphi*, *Kylix* e *Lazarus*. Atualmente, a linguagem *Pascal* foi padronizada pela ISO nos padrões *Standard Pascal* e *Advanced Pascal*.

Embora não tenha sido a primeira linguagem a incorporar os conceitos de Programação Estruturada, o *Pascal* é considerado um marco na evolução das linguagens de programação, devido a várias inovações introduzidas na época, tais como a criação de novos tipos, a estruturação de dados, a alocação dinâmica de variáveis com o auxílio de ponteiros, a declaração de identificadores para constantes, a utilização de procedimentos que lêem e escrevem campos individuais em vez de registros completos, o uso de funções e procedimentos recursivos, entre outras.

3 O compilador *Pascal*

Código-fonte é um conjunto de palavras ou símbolos escritos de maneira ordenada, contendo instruções em uma linguagem de programação conhecida. Normalmente, o código-fonte é escrito em uma linguagem de alto-nível, ou seja, com grande capacidade de abstração. O compilador é um programa que traduz um código-fonte escrito em uma linguagem de alto-nível em um programa de baixo-nível, ou seja, um programa que seja formado por um conjunto de instruções a serem realizadas pelo processador. O processo de compilação ocorre em duas etapas:

- Análise sintática de um código fonte, segundo as regras gramaticais da linguagem escolhida;
- Geração do programa executável semanticamente equivalente.

Este guia abrange a primeira parte, isto é, contém material que explica como escrever programas sintaticamente corretos. A geração do código-executável, por sua vez, é dependente de plataforma e de sistema operacional e foge do propósito deste documento.

Existem vários compiladores disponíveis para a linguagem *Pascal*. Alguns deles são sensíveis à plataforma. No contexto deste manual e da disciplina correlata, foi adotado o compilador *Free Pascal* por, basicamente, por dois motivos:

- É software livre;
- É fornecido para diversas plataformas, incluindo o *Linux*.

É absolutamente fundamental que os leitores deste material consultem o guia de referência da linguagem *Pascal* para o compilador *Free Pascal*, a fim de obter uma completa explicação de todas as regras para esta versão da linguagem. Este guia foi criado com o intuito de ser uma primeira referência a esta linguagem.

Quando forem apresentados exemplos de programas sendo compilados, serão feitos para a versão em *Linux*.

Conforme antedito, há muito material sobre outros compiladores para a linguagem. O leitor é encorajado a não se limitar ao corrente guia.

3.1 Obtendo o compilador

O compilador pode ser instalado a partir dos pacotes disponíveis na página do projeto *Free Pascal*[5], bastando que seja escolhido o pacote correspondente ao seu hardware e sistema operacional.

Nos laboratórios do Departamento de Informática da Universidade Federal do Paraná, usa-se o *Debian GNU/Linux*. Para instalar o compilador neste ambiente, basta executar, como *root*, o seguinte comando:

```
$ apt-get install fp-compiler
```

[5] `http://www.freepascal.org`

Caso queira, o usuário pode instalar todo o ambiente de desenvolvimento disponível em uma família de pacotes, incluindo a documentação, através do seguinte comando:

```
$ apt-get install fp-docs fpc lazarus
```

O *Lazarus* é um ambiente de desenvolvimento e pode ser usado alternativamente ao editor de texto. Todavia, este material é escrito para aqueles que usam editores de texto e *shell*.

3.2 Editando um programa-fonte

Para se gerar um programa-fonte deve-se usar um editor de texto simples. No *Linux*, existem vários editores, tais como o *vi*, o *emacs*, o *gedit* e o *joe*. Ainda há a opção de usar outro editor ASCII de sua preferência, incluindo o disponível no ambiente *Lazarus*.

No *Windows*, escolha um editor como o *Notepad* (Bloco de Notas), ou algum fornecido por um ambiente de desenvolvimento em *Pascal*, como o editor do *Turbo Pascal*.

Em nenhuma hipótese utilize editores das suites de escritório, pois estes inserem caracteres não-ASCII que o compilador não conhece. Além disso, os compiladores geralmente não sabem lidar com acentuação, negrito, itálico, entre outras formatações. Então, aceite o fato, escreva em ASCII puro.

3.3 Compilando

Para se compilar um programa-fonte em *Pascal* usando o *Free Pascal* no *Linux*, basta abrir um *shell* e executar o programa `fpc`. Suponha que o seu arquivo-fonte tenha o nome *alomamae* e contenha o seguinte texto:

```
program alomamae;

begin
   writeln ('Alo mamae!');
end.
```

Este código-fonte, após compilado e executado, produzirá na tela a mensagem "Alo mamae!".

Para compilar, abra um *shell* e digite:

```
$ fpc alomamae.pas
```

Este comando produziu em seu diretório de trabalho um arquivo binário de nome `alomamae`. Para executá-lo, faça o seguinte:

```
$ ./alomamae
```

e aparecerá na tela a mensagem de saída do programa.

Na seqüência deste material, será explicado como construir programas-fontes sintática e semanticamente corretos, que podem ser compilados sem gerarem erros de compilação.

3.4 Exemplo de interação com a máquina para compilar

A seguir, apresenta-se uma sessão de trabalho em que se compila o programa `alomamae.pas`.

```
$> ls -l
-rw-r--r-- 1 marcos c3sl 60 Ago  4 11:18 alomamae.pas
```

Este comando mostra que no diretório de trabalho existe um arquivo de nome `alomamae.pas` que foi criado às 11h18min do dia 4 de agosto de 2009. Este arquivo é de propriedade do usuário `marcos`, que pertence ao grupo `c3sl`. O arquivo pode ser lido por qualquer usuário, mas pode ser alterado apenas pelo dono. Como última informação, ele ocupa 60 bytes de disco.

```
$> fpc alomamae.pas
Free Pascal Compiler version 2.2.2-8 [2008/12/20] for i386
Copyright (c) 1993-2008 by Florian Klaempfl
Target OS: Linux for i386
Compiling alomamae.pas
Linking alomamae
6 lines compiled, 0.2 sec
```

O comando acima invocou o compilador, que imprimiu algumas mensagens de versão e autoria, fundamentalmente informando que compilou as 6 linhas do programa em 0.2 segundos.

```
$> ls -l
-rwxr-xr-x 1 marcos c3sl 114400 Ago  4 11:28 alomamae
-rw-r--r-- 1 marcos c3sl   1832 Ago  4 11:28 alomamae.o
-rw-r--r-- 1 marcos c3sl     60 Ago  4 11:18 alomamae.pas
```

O comando acima mostra que, às 11h28 minutos do dia 4 de agosto de 2009, foram criados, pelo compilador, dois arquivos novos: `alomamae` e `alomamae.o`. O primeiro é um arquivo executável ocupando 114.400 bytes em disco, o segundo é um arquivo temporário utilizado pelo compilador durante o processo de geração de código. O leitor deste material não deve se preocupar com este arquivo no momento.

```
$> ./alomamae
Alo mamae!
$>
```

Finalmente, foi executado o programa e vemos a saída, conforme esperado. Tentar executar o arquivo-fonte produziria o seguinte erro:

```
$> ./alomamae.pas
bash: ./alomamae.pas: Permissão negada
$>
```

Isto ocorre porque o código-fonte não é, como explicado, um arquivo executável, ele apenas contém as instruções que servem para que o compilador gere o programa de fato.

3.5 Erros de compilação

Caso o programa-fonte não respeite as regras gramaticais da linguagem (aqui, *Pascal*), o compilador produz mensagens de erro correspondentes. Observe o que ocorre se, por exemplo, fosse omitido o ";" ao final da primeira linha:

```
$> cat alomamae.pas
program alomamae

begin
    writeln ('Alo mamae!');
end.
$> fpc alomamae.pas
Free Pascal Compiler version 2.2.2-8 [2008/12/20] for i386
Copyright (c) 1993-2008 by Florian Klaempfl
Target OS: Linux for i386
Compiling alomamae.pas
alomamae.pas(3,1) Fatal: Syntax error, ";" expected but "BEGIN" found
Fatal: Compilation aborted
Error: /usr/bin/ppc386 returned an error exitcode (normal if you did not specify a sou
```

Durante o processo de compilação, após ler a palavra reservada "`program`" e o identificador "`alomamae`", o compilador esperava encontrar o símbolo ";". Ao invés disso, encontrou na linha 3, coluna 1, o símbolo "`begin`". Como resultado deste erro, não foi gerado o programa executável.

Na verdade, o compilador é incapaz de dizer, com absoluto acerto, o local exato onde ocorreu o erro. Neste caso, sabe-se que faltou um ponto-e-vírgula no final da linha 1.

3.5.1 Exercícios

Sugere-se que o leitor produza erros de sintaxe na sua cópia deste programa-fonte e que explore o compilador para começar a compreender as mensagens de erro e a procurar corrigi-los.

Experimente, por exemplo, as seguintes alterações:

- escreva `program` errado, por exemplo, `pogram`;
- escreva `writeln` errado, por exemplo, `writenl`;
- apague o ponto final após o `end` na última linha;
- esqueça de abrir ou de fechar as aspas da mensagem;
- esqueça de abrir ou de fechar os parênteses;
- escreva alomamãe com til na linha 1.

Faça uma alteração por vez e observe quais mensagens de erro serão retornadas pelo compilador.

4 Estrutura de um programa-fonte em *Pascal*

O código-fonte de um programa em *Pascal* deve ser estruturado de maneira que respeite a gramática formalizada para esta linguagem. Com isto, o compilador pode transformar o programa-fonte em um código-executável.

Um programa em *Pascal* consiste de dois blocos: o cabeçalho e o programa principal. A figura abaixo ilustra a estrutura básica de um programa. Os textos que aparecem entre os símbolos "<" e ">" não são parte do programa, mas estruturas que serão detalhadas mais à frente neste documento. Igualmente, os textos que aparecem entre os símbolos "(*" e "*)" constituem comentários no código e são explicados na seção 5.3.

```
(* inicio do cabecalho *)
program <nome_do_programa>;

      (* declaracao dos rotulos *)

      (* declaracao das constantes *)

      (* declaracao dos tipos *)

      (* declaracao das variaveis *)

      (* declaracao dos procedimentos e funcoes *)

(* termino do cabecalho *)

(* inicio do programa principal *)
begin

   <comandos>

end.
(* termino do programa principal *)
```

O programa propriamente dito, conhecido como *programa principal* é o texto que aparece entre as palavras reservadas (ver seção 5.2) `begin` e `end`. O programa termina com um ponto final[6].

O cabeçalho é tudo o que aparece a partir da palavra reservada `program` até o que vai imediatamente antes do `begin` do programa principal. O cabeçalho não constitui código executável, mas contém informações necessárias para o processo de compilação, bem como códigos para subprogramas (ver seção 4.1.5) que podem ser ativados por algum comando no programa principal.

Cada tópico será explicado no decorrer desta seção.

4.1 Cabeçalho (ou preâmbulo)

Todo programa em *Pascal* começa com a palavra reservada `program` (ver seção 5.2) seguida de um identificador (ver seção 5.4) que é o nome do programa e um símbolo

[6]Nas ilustrações deste texto, as palavras reservadas aparecem em negrito

de ponto-e-vírgula (;).

No cabeçalho também deve-se especificar, se necessário:

1. Os rótulos para comandos de desvio incondicional, se algum tiver sido usado;
2. As constantes, se alguma foi usada no programa;
3. Os tipos definidos pelo usuário, se algum foi necessário;
4. As variáveis globais usadas no programa;
5. Os códigos dos subprogramas eventualmente usados pelo programa.

As declarações devem ser inseridas nesta ordem, embora não seja obrigatório que se usem todas estas declarações. A seguir, será explicada como é a sintaxe de cada uma.

4.1.1 Seção de declaração de rótulos

Um rótulo (*label*) é um nome para uma localização no código-fonte para o qual o fluxo de execução do programa pode ser direcionado de outra localização pelo uso de um comando `goto`.

Os rótulos podem ser seqüência de dígitos ou identificadores.

Logo, os rótulos são apenas necessários quando se usa um ou mais comandos de desvio incondicional (`goto`, ver seção 7.4).

Exemplo:

```
label alo, 100, FIM;
```

4.1.2 Seção de declaração de constantes

Constantes são úteis quando o programador precisa se referir a informações que não se alteram durante o programa, ou seja, são estáticas. Contrastam, portanto, com as variáveis, que permitem reatribuição de valores.

A declaração é feita através da palavra reservada `const`. Nesta seção do programa, cada constante nova é declarada por um nome de identificador, ainda não utilizado, seguido do símbolo igual ("=") e sucedida pela constante numérica ou alfanumérica desejada, sempre terminando com um ponto-e-vírgula (";").

Usualmente, os nomes de constantes são escritos em letras maiúsculas para contrastarem com os nomes de variáveis.

Exemplo:

```
const
   PI = 3.14159;
   MEU_CPF = '000.123.456-78';
   ERRO_MSG = 'Erro na execucao do programa.';
   QUATRO = 4;
```

```
    MENOS_QUATRO = -quatro;
    MAX = 10000000;
```

4.1.3 Seção de declaração de tipos

Em *Pascal*, toda variável (ver seção 4.1.4) possui um tipo. Isto é relacionado com a maneira como o computador vai interpretar a seqüência de bits no endereço de memória acessado.

Na linguagem *Pascal* existem alguns tipos pré-definidos (ver seção 5.5). Aqui, contudo, o programador pode declarar seus próprios tipos de dados. Assim, um identificador pode ser usado para denotar este novo tipo ao se declarar variáveis no código-fonte.

A declaração é feita tendo como início a palavra reservada `type`. Segue então uma lista contendo um novo identificador, um símbolo de igual ("=") e a definição do novo tipo baseada em tipos pré-existentes. Finalmente um símbolo de ponto-e-vírgula (";").

Abaixo há alguns exemplos de como se declarar novos tipos.

Exemplo:

```
type
    diaMes = integer;
    tipoSaldo = real;
    letra_minuscula = 'a'..'z';
    nomes = array [1..MAX] of letra_minuscula;
```

4.1.4 Seção de declaração de variáveis globais

Em *Pascal*, toda variável global (ver seção 4.1.4) deve ser declarada. Assim, o compilador passa a conhecer aquele nome e também reserva espaço de memória para o conteúdo da variável. Adicionalmente, a declaração propicia que o compilador possa interpretar corretamente o mapeamento entre o formato binário da máquina, correspondente ao espaço de memória alocado, e a maneira como estes bits serão manipulados em operações, comandos de leitura ou de impressão.

Variáveis são declaradas através da palavra reservada `var`. Nesta seção do programa, cada nova variável (ou lista de variáveis separadas por vírgula) é declarada por um novo nome de identificador seguido do símbolo de dois pontos (":") e sucedido pelo nome do tipo da variável, seja um tipo pré-definido pelo compilador ou um novo tipo definido pelo programador (conforme explicado na seção 4.1.3). Como sempre, todo comando em *Pascal* deve ser terminado por um ponto-e-vírgula (";").

A diferença entre variáveis globais ou locais será caracterizada na seção 4.1.5.

Exemplos:

```
var
    i, j, k, idade: integer;
    b: byte;
```

```
    raiz, x, y, z, media: real;
    letra: char;
    nome: string;
    saldo: tiposaldo;
```

4.1.5 Seção de declaração de procedimentos e funções

Procedimentos e funções são formas de escrever subprogramas na linguagem *Pascal*. Estes códigos só são executados quando ativados no programa principal.

Assim é feita a declaração de um procedimento:

```
procedure <identificador> ( <lista de parametros> );
(* cabecalho do procedimento *)
begin
    (* corpo do procedimento *)
end;
```

Uma função, por sua vez, é declarada desta forma:

```
function <identificador> ( <lista de parametros> ) : <tipo>;
(* cabecalho da funcao *)
begin
    (* corpo da funcao *)
end;
```

As palavras `procedure` e `function` são reservadas. O corpo de um procedimento ou de uma função é constituído por comandos da mesma maneira que o programa principal (ver seção 4.2), com a diferença de que, nos subprogramas, os comandos só são executados quando o procedimento ou a função tiverem sido invocados no programa principal.

O cabeçalho de um procedimento ou de uma função é constituído da mesma maneira que o cabeçalho do programa principal. Isto é, permite a declaração de constantes, de variáveis ou de tipos. A diferença reside no fato de que, tais elementos, possuem abrangência (escopo) local e não estão acessíveis a outros subprogramas ou ao programa principal.

Uma variável declarada em um subprograma existe somente durante a execução deste. Pode, inclusive, ter o mesmo nome de um identificador previamente usado, seja em outro subprograma, seja até no programa principal.

A lista de parâmetros é uma seqüência (que pode ser vazia) de nomes de identificadores, separados por vírgula, com seus respectivos tipos precedidos por dois-pontos (“:”). Para cada novo tipo, deve-se usar o separador ponto-e-vírgula. Os parâmetros são argumentos repassados ao procedimento ou à função em foco.

O retorno, característica específica de funções, tem o respectivo tipo definido após a lista de parâmetros, precedido por dois pontos (“:”). O valor de retorno é especificado através de uma atribuição ao identificador da função, como se este fosse uma variável do tipo definido como retorno.

O compilador não aceita tipos estruturados como parâmetro ou como retorno de subprogramas, a menos que estes tenham sido previamente definidos pelo programador na seção de declaração de tipos.

A linha que contém o cabeçalho do procedimento ou função define a assinatura ou protótipo do subprograma, isto é, estabelece o nome, os parâmetros com respectivos tipos e, no caso das funções, o tipo de retorno.

Os parâmetros podem ser passados de duas maneiras, tanto para procedimentos quanto para funções: por valor ou por referência. Na assinatura do subprograma, a diferença é que os parâmetros passados por referência levam a palavra `var` antes da lista de parâmetros.

Os parâmetros passados por valor recebem uma cópia dos valores das variáveis usadas como argumento na chamada do subprograma. Os parâmetros passados por referência remetem, na verdade, à própria variável; que sofrerá todas as alterações realizadas.

Os subprogramas são invocados ou no programa principal ou por outro subprograma da seguinte maneira:

- Os procedimentos são invocados pelo seu nome e lista de argumentos, da mesma maneira como se invocaria um comando qualquer da linguagem, garantindo-se o número de parâmetros da assinatura do subprograma e os respectivos tipos;
- As funções são invocadas também por seu nome e lista de argumentos, contudo a invocação deve ser feita onde se espere um valor do mesmo tipo que o retorno da função virá a fornecer. Assim, se determinada função retornar um valor real, pode-se invocá-la em qualquer lugar onde se admita um valor real.

É importante observar que o compilador faz a checagem do número de parâmetros e respectivos tipos da lista de argumentos do cabeçalho do subprograma com os mesmos itens da ativação deste. Caso o número de argumento, a ordem, ou os tipos não compatibilizem, o compilador acusa erro.

Exemplos:

Neste primeiro exemplo, o procedimento `aloMamae` é ativado pelo programa principal.

```
program umProcedimento;
 procedure aloMamae;
    begin
       writeln;
       writeln('Alo Mamae!!!');
       writeln;
    end;
 begin
    aloMamae;
 end.
```

Aqui, faz-se a chamada do procedimento `perfil`. São passados como parâmetro o nome, a idade e o peso do indivíduo. Embora, as variáveis globais sejam denominadas

`n`, `i` e `p`, elas poderiam perfeitamente se chamarem `nome`, `idade` e `peso`, visto que as variáveis do procedimento `perfil` têm escopo local.

```
program outroProcedimento;
 procedure perfil(nome : string; idade : integer; peso : real);
    begin
       writeln('Nome:  ', nome);
       writeln('Idade: ', idade);
       writeln('Peso:  ', peso);
    end;
 var
    n : string;
    i : integer;
    p : real;
 begin
    n := 'Blaise Pascal';
    i := 39;
    p := 70.5;

    perfil(n, i, p);
 end.
```

Este terceiro exemplo apresenta um procedimento em que alguns parâmetros são agrupados por tipo no cabeçalho.

```
 procedure variosTipos(a, b : integer; c : real; d : integer);
    begin
       writeln('Inteiros: ', a, ' e ', b);
       writeln('Real: ', c);
       writeln('Novamente inteiro: ', d);
    end;
```

Neste quarto exemplo é mostrado como se faz a passagem de parâmetro por referência em um procedimento. Para funções, a sintaxe também é válida.

```
program porReferencia;
 procedure duplica(var n : integer);
    begin
       n := 2 * n;
    end;
 var
    numero : integer;
 begin
    numero := 10;
    duplica(numero);
    writeln(numero);
 end.
```

Exemplo que expõe um procedimento recursivo que conta de um inteiro n até zero.

```
procedure contaAteZero(n : integer);
   begin
```

```
    if (n >= 0) then
        begin
            writeln(n);
            contaAteZero(n - 1);
        end;
end;
```

Aqui, apresenta-se um exemplo em que se utiliza um tipo estruturado, predefinido, como parâmetro.

```
program tipoEstruturado;
 type
    vetor10int = array [1..10] of integer;
 procedure exibeVetor(v : vetor10int);
    var
        i : integer
    begin
        writeln;
        writeln('Vetor digitado:');
        for i := 1 to 10 do
            writeln(v[i]);
    end;
 var
    i : integer;
    vetor : vetor10int;
 begin
    for i := 1 to 10 do
        begin
            writeln('Digite um valor para a posicao ', i, ':');
            readln(vetor[i]);
        end;
    exibeVetor(vetor);
 end.
```

Neste outro exemplo, evidencia-se a chamada, a passagem de parâmetro e o retorno de uma função que recebe duas notas inteiras e devolve a média anual (um valor real) de um aluno.

```
program calculoDaMedia;
 function media(nota1, nota2 : integer) : real;
    begin
        media := (nota1 + nota2) / 2;
    end;
 var
    a, b : integer;
 begin
    writeln('Digite a media do primeiro semestre:');
    readln(a);

    writeln('Digite a media do segundo semestre:');
    readln(b);
```

```
    writeln('Media anual: ', media(a, b));
end.
```

Por fim, este exemplo apresenta o uso uma função recursiva que retorna o n-ézimo elemento da série de Fibonacci.

```
function fibonacci(n : integer) : integer;
   begin
      if (n < 3) then
         fibonacci := 1
      else
         fibonaci := fibonacci (n-1) + fibonacci(n-2);
   end;
```

4.2 Programa principal

O trecho demarcado como programa principal é onde, efetivamente, é escrito o programa na linguagem *Pascal*. Qualquer subprograma (procedimento ou função, tratados na seção 4.1.5) só é executado quando chamado primariamente a partir do programa principal.

O programa principal é constituído de comandos seqüenciais (ver seção 7) que são executados iniciando da instrução `begin` até que seja encontrada a instrução `end` e o ponto final do programa.

A execução dos comandos contidos no programa principal é feita de cima para baixo e da esquerda para a direita, ignorando-se os espaços em branco e as linhas vazias. Este comportamento só se altera quando utilizados comandos que manipulem o fluxo natural de execução do programa, conforme explicado na seção 7.

O próxima capítulo contém mais instruções sobre os elementos básicos da linguagem *Pascal*.

5 Elementos básicos

Neste capítulo, discutem-se os elementos básicos que compõem a linguagem *Pascal*. Em primeiro lugar, são apresentados os símbolos e as palavras reservadas usados na construção de um código-fonte (seções 5.1 e 5.2, respectivamente). Na seção 5.3, são discutidas as alternativas para a inserção de comentários no código-fonte. A seção 5.4 trata da criação de identificadores válidos para a linguagem *Pascal*.

A partir da seção 5.5, são apresentados os tipos de dados utilizados em *Pascal*. São elencados desde os tipos básicos, como o inteiro e o caractere, até aqueles mais complexos, como os que são voltados à construção de estruturas de dados.

5.1 Símbolos

É possível inserir quaisquer símbolos num arquivo-fonte em *Pascal*. O compilador dará um significado adequado, a cada símbolo presente no código.

Os símbolos seguintes têm significados especiais, por se tratarem de operadores ou indicadores de que determinada parte do código não se trata de uma instrução:

e/ou símbolos que podem indicar ao compilador que aquela parte do código onde se encontram não é uma instrução:

```
+ * / = < > [ ] . , ( ) : ^ @ { } $ #
```

Os símbolos abaixo seguem a mesma regra, mas são sempre usados em par (note que '`<=`', sem espaço, é diferente de '`< =`');

```
<= >= := += -= *= /= (* *) (. .) //
```

Dentro de uma `string`,tais símbolos fazem parte da palavra e não serão interpretados pelo compilador de outra forma.

5.2 Palavras reservadas

Palavras reservadas são componentes da própria linguagem *Pascal* e não podem ser redefinidas, ou seja, denominar elementos criados pelo programador. Por exemplo, `begin`, palavra reservada que indica o início de um bloco de código, não pode ser nome rótulos, constantes, tipos ou quaisquer outros identificadores no programa.

O compilador *Pascal* não faz distinção entre maiúsculas e minúsculas. Assim, tanto identificadores quando palavras reservadas podem ser grafados com variações de maiúsculas e minúsculas sem alteração de significado: `begin`, `BEGIN`, `Begin` ou `BeGiN`.

Conforme o guia de referência da linguagem *Free Pascal*, versão de dezembro de 2008, as palavras reservadas são divididas em três grupos. O primeiro contém as palavras reservadas para a versão do *Turbo Pascal*[7]. O segundo compreende as palavras reservadas do *Delphi*, e o último encerra as do *Free Pascal*.

[7] `http://pt.wikipedia.org/wiki/Turbo_Pascal`

Esta diferença existe porque no *Turbo Pascal* as palavras do *Delphi* não são reservadas.

A tabela abaixo apresenta uma lista das palavras reservadas reconhecidas pelo compilador *Turbo Pascal*:

absolute	else	nil	set
and	end	not	shl
array	file	object	shr
asm	for	of	string
begin	function	on	then
break	goto	operator	to
case	if	or	type
const	implementation	packed	unit
constructor	in	procedure	until
continue	inherited	program	uses
destructor	inline	record	var
div	interface	reintroduce	while
do	label	repeat	with
downto	mod	self	xor

Tabela 1: Palavras reservadas do *Turbo Pascal*

São palavras reservadas do *Delphi*:

as	class	except
exports	finalization	finally
initialization	is	library
on	out	property
raise	threadvar	try

Tabela 2: Palavras reservadas do *Free Pascal*

O compilador *Free Pascal* suporta todas as palavras reservadas descritas acima e mais as seguintes:

dispose	false	true
exit	new	

Tabela 3: Palavras reservadas do *Delphi*

Atenção: as palavras reservadas variam conforme o compilador utilizado. Portanto, é indispensável consultar o guia de referência para a versão do compilador que tenha em mãos.

5.3 Comentários

Os comentários são colocados no arquivo-fonte para ajudar a entender o que determinado bloco de código faz. Comentários não são interpretados pelo compilador.

O *Free Pascal* suporta três tipos de comentários. Dois destes já eram utilizados por compiladores *Pascal* antigos; um terceiro vem da linguagem C).

A seguir, uma lista de comentários válidos:

```
(* Eis um comentario.
   Ele pode ter mais de uma linha *)

{ Um outro comentario.
  Tambem pode ter mais de uma linha}

// Um comentario que compreende apenas ate o final da linha
```

Você pode colocar comentários dentro de comentários, com a condição de que os tipos de comentários sejam diferentes. Exemplos:

```
(* Isto eh um comentario.
   Ele pode ter mais de uma linha
   {
      Aqui esta outro comentario dentro do primeiro
      Isso eh totalmente valido
      // Este eh um terceiro comentario. Tambem valido.
   }
)*

// Isto eh um comentario. (* Este eh outro comentario
   Porem eh um comentario invalido, pelo excesso de linhas
*)

{ Primeiro comentario
  {
     Segundo comentario
     O primeiro comentario sera fechado a seguir
  }
  E a chave a seguir esta sobrando
}
```

5.4 Identificadores

Identificadores são utilizados para denominar programas, rótulos, constantes, variáveis, tipos, procedimentos, funções e `units` criados pelo programador.

São regras, em *Pascal* para se constituir corretamente um identificador:

1. Um identificador não pode ser igual a uma palavra reservada;
2. Um identificador deve ter no máximo 255 caracteres;
3. Os símbolos que constituem um identificador podem ser:

- letras: ‘A’ até ‘Z’ ou ‘a’ até ‘z’;
- dígitos: 0 até 9;
- um sublinhado: _.

4. O primeiro símbolo deve ser uma letra ou o sublinhado.

Assim, não são aceitos espaços ou caracteres especiais em qualquer parte de um identificador. Até mesmo o cedilha (‘ç’) é considerado um caractere especial.

São exemplos de identificadores válidos:

- `a`, `a1`, `b568`, `codigo`, `raizquadrada`, `preco_especial`;
- `_veiculo`, `elemento_do_conjunto`, `raiz_quadrada`, `b54_`;
- `CorVeiculo`, `Ano`, `MES`, `SaLdO`, `Parcela_Quitada`;
- `Este_nome_eh_muito_longo_e_raramente_deve_ser_usado_mas_eh_valido`.

Exemplos de identificadores **inválidos**:

- `12porcento`, `4por4`: iniciam por número;
- `cor veiculo`: contém espaço em branco;
- `preço`: possui um caracter inválido (cedilha);
- `pássaro`: apresenta acentuação;
- `%b54_`: não inicia por letra ou por sublinhado.

Observações:

- Não se pode declarar uma variável com o mesmo nome que foi atribuído ao programa ou a qualquer outro identificador previamente declarado no mesmo escopo[8];
- Como o compilador *Pascal* ignora o estado de maiúsculas e minúsculas, os identificadores podem ser grafados de maneiras diferentes e ter o mesmo significado, como por exemplo: `saldo`, `SALDO`, `Saldo` ou `SaLdO`.

[8]Para informações sobre escopo, veja a seção 4.1.5)

5.5 Tipos de dados em *Pascal*

A linguagem *Pascal* é, por definição, uma linguagem fortemente tipada. Ser tipada significa que as variáveis e constantes possuem um tipo precisamente definido, isto é, o compilador deve conhecer detalhes de quantidade de bytes para seu armazenamento e o formato de conversão decimal/binário.

Já, uma linguagem dita fortemente tipada, com raras exceções, não permite que tipos sejam misturados. Tal ocorrência provoca erro de compilação ou até mesmo durante a execução do programa.

Na seção 4.1.3 mostrou-se que é possível declarar novos tipos sob demanda do programador. Na presente seção, serão enumerados todos os tipos considerados *básicos* para o compilador *Free Pascal*. Convém lembrar que isto pode mudar de um compilador para outro. Logo, procure sempre informações atualizadas para a versão do compilador que tem em uso.

Os tipos que interessam para a disciplina introdutória de programação são os seguintes:

- a família de tipos ordinal;
- a família de tipos real;
- `boolean` (booleano);
- `char` (caractere);
- `string` (cadeia de caracteres);
- `array` (matriz);
- `record` (registro);
- `file` (arquivo).

Contudo, é importante observar que para níveis avançados de programação existem outros tipos pré-definidos na linguagem. Consultar o guia de referência para maiores informações.

5.5.1 A família de tipos ordinal

Os tipos ordinais representam números conhecidos em linguagem informal como sendo os *tipos inteiros*.

Segundo o guia de referência da linguagem, em sua versão de dezembro de 2008, tem-se que, com exceção do `int64`, `qword` e dos tipos reais, todos os tipos básicos são do tipo ordinal.

Conservam as seguintes características:

1. São enumeráveis e ordenados, ou seja, é possível iniciar uma contagem de um por um destes números em uma ordem específica. Esta propriedade implica que as operações de incremento, decremento e ordem funcionam;

2. Existem menores e maiores valores possíveis.

A seguir é relacionada uma lista com os tipos inteiros e respectivas faixas de valores. O programador deve escolher o tipo mais conveniente à representação da variável, pois a diferença básica entre eles é a quantidade de bytes usada em memória e o fato de necessitar que números negativos sejam representados. Para um estudante de Ciência da Computação, a diferença está na interpretação do número em binário, se está em complemento de 2 ou em "binário puro".

Tipo	Faixa dos limites	Tamanho em bytes
`byte`	0 ..255	1
`shortint`	-128 .. 127	1
`smallint`	-32768 .. 32767	2
`word`	0 .. 65535	2
`integer`	`smallint` ou `longint`	2 ou 4
`cardinal`	`longword`	4
`longint`	-2147483648 .. 2147483647	4
`longword`	0 .. 2147483647	4
`int64`	-9223372036854775808 .. 9223372036854775807	8
`qword`	0 .. 18446744073709551615	8

Tabela 4: Tipos ordinais em *Pascal*

Funções predefinidas relacionadas: `abs`, `chr`, `dec`, `inc`, `int`, `odd`, `ord`, `power`, `pred`, `random`, `randomize`, `round`, `succ`, `trunc` e `val`. Vide anexo A.

5.5.2 Tipo enumerável

Os tipos enumeráveis são definidos pelo programador. Possuem uma certa seqüência de possíveis valores ordinais que as variáveis daquele tipo poderão assumir.

Exemplo:

```
program tipo_enum;
 type
    TipoEsportes = (Futebol, Basquete, Volei);
 var
    Esporte1, Esporte2: TipoEsportes;
 begin
    Esporte1 := Futebol;
    Esporte2 := Basquete;
    { Perceba que Esporte1 tem o valor Futebol, e nao 'Futebol', pois
        nao se trata de um tipo string. }
    if Esporte1 = Futebol then
       writeln('O primeiro esporte eh Futebol');
 end.
```

5.5.3 Tipo sub-faixa

Os tipos sub-faixas possuem valores numa determinada escala (que pode ser constituída de números ou letras.

Exemplo:

```
program tipo_sub_faixa;
 type
    Tipo_um_a_cinco = 1 .. 5; // Vai de 1 ate 5
    Tipo_a_a_f = 'a' .. 'f';  // Vai de a ate f
 var
    Numero: Tipo_um_a_cinco;
    Letra: Tipo_a_a_f;
 begin
    Numero := 3; // OK. Esta na faixa
    writeln(Numero);
    Letra := 'p'; // OK. Esta na faixa
    writeln(Letra);
    Letra := 'z'; // Esta fora da faixa estabelecida. O compilador
       mostrara um warning
    writeln(Letra);
 end.
```

5.5.4 A família de tipos real

Os tipos reais compreendem aqueles representados internamente pelo computador e interpretados como sendo de ponto flutuante[9].

Assim como no caso dos ordinais, variam basicamente na quantidade de bytes usada para cada endereço de memória e na representação de sinais, seja no expoente ou na mantissa.

Abaixo apresenta-se uma lista com os tipos reais e suas faixas de valores, conforme o guia de referência em sua versão de dezembro de 2008.

Tipo	Faixa dos limites	Tamanho em bytes
`real`	depende da plataforma	4 ou 8
`single`	1.5E-45 .. 3.4E38	4
`double`	5.0E-324 .. 1.7E308	8
`extended`	1.9E-4932 .. 1.1E4932	10
`comp`	-2E64+1 .. 2E63-1	8
`currency`	-922337203685477.5808 .. 922337203685477.5807	19-20

Tabela 5: Tipos reais em *Pascal*

Aqui reside uma das raras exceções da checagem de tipo da linguagem, pois uma variável do tipo real pode receber a soma de um real com um inteiro. Neste caso,

[9]Apesar de estarmos no Brasil, em *Pascal* os números em ponto flutuante são escritos com um ponto, não com vírgula.

antes da operação, o computador faz a conversão do inteiro para um formato em ponto flutuante.

Exemplo:

```
program type_real;
 var
    a, b, c: real;
    d: integer;
 begin
    a := 14.5;
    b := sqrt(10); // b eh igual a raiz de 10
    d := 10; / d eh inteiro
    c := a + d; // c eh a soma de um real com inteiro
    writeln('a=', a, ' b=', b, ' c=', c, ' d=', d);
    writeln;
    writeln('a=', a:0:2); // ajusta para duas casas decimais apos o
       ponto
  end.
```

Funções predefinidas relacionadas: `abs`, `arctan`, `cos`, `exp`, `frac`, `int`, `log`, `pi`, `power`, `random`, `randomize`, `round`, `sin`, `sqr`, `sqrt`, `trunc` e `val`. Vide anexo A.

5.5.5 Tipo `boolean` (booleano)

O `boolean` (booleano) é um tipo lógico que assume apenas um entre dois possíveis valores: `false` ou `true`, que equivalem respectivamente a falso ou verdadeiro (0 ou 1).

Sempre que um valor diferente de 0 for associado a uma variável booleana, esta será verdadeira.

Exemplo:

```
program type_bool;
var
 a, b, c, d: boolean;
begin
 a := false;
 b := true;
 c := boolean(0);     // valor igual a zero: false
 d := boolean(-10);   // valor  diferente de zero: true
 writeln('Variaveis: a=', a, ' b=', b, ' c=', c, ' d=', d);
end.
```

As variáveis ou constantes do tipo `boolean` são elementos fundamentais na definição e entendimento de *expressões booleanas*, que serão apresentadas na seção 6.2.

5.5.6 Tipo `char` (caractere)

O tipo `char` (ou caractere) armazena apenas um caracter da tabela ASCII[10] e ocupa exatamente um byte de memória. Na realidade, não é o caractere que é armazenado na memória, mas o código ASCII correspondente.

O código ASCII do caractere 'A', por exemplo, é 65. Então, se for atribuído 'A' a uma variável do tipo caractere, na verdade será atribuído o valor 65 para esta variável. Pode-se atribuir diretamente o valor ASCII a uma variável caractere inserindo-se o sustenido na frente do número correspondente (por exemplo, `a := #65`).

Exemplo:

```
program type_char;
 var
    a, c, d: char;
  b: integer;
 begin
  a := 'A';           // armazena 65 na variavel a, que equivale a 'A'
  b := ord('A') + 2; // obtem o codigo ASCII de 'A' (65) e acrescenta 2
                      // observe que b, que recebe isto, eh um inteiro
  c := chr(b);        // armazena em c o caractere correspondente ao
                      // valor 67 em ASCII
  d :=  #68;
  writeln('Variaveis: a=', a, ' b=', b, ' c=', c, ' d=', d);
  // retorna a=A b=67 c=C
end.
```

Por se tratar de um tipo ordinal, o tipo char pode ser utilizado em comandos `case` (ver seção 7.5.3).

Funções predefinidas relacionadas: `chr`, `lowercase`, `ord` e `upcase`. Vide anexo A.

5.5.7 Tipo `string` (cadeia de caracteres)

O tipo `string` é utilizado para armazenar palavras, isto é, seqüências de símbolos ASCII.

Em *Pascal*, as palavras, assim como o tipo `char`, devem estar entre apóstrofos (aspas simples). Pode-se concatenar *strings* e caracteres usando o operador de adição ('+').

Exemplo:

```
program type_str;
var
  a, b, d: string;
  c: char;
begin
  a := 'hello ';
  b := 'world';
  c := '!';
  d := a + b + c;
```

[10] `http://pt.wikipedia.org/wiki/ASCII`

```
  writeln(d);      // hello world!
end.
```

Funções predefinidas relacionadas: `concat`, `copy`, `delete`, `insert`, `length`, `lowercase`, `pos`, `upcase` e `val`. Vide anexo A.

5.6 Tipo `array` (matriz)

Em sua forma elementar, o tipo `array` é utilizado para armazenar, sob um mesmo nome de variável, uma quantidade fixa de elementos do mesmo tipo. Por isto, o tipo `array` também é conhecido como um *arranjo homogêneo*. Para formas de uso mais complexas recomendamos a leitura do guia de referência da linguagem *Pascal*.

A declaração de uma variável do tipo `array` é feita assim:

```
array [<lista de tipos enumeraveis>] of <tipo>
```

onde `<tipo>` é qualquer tipo previamente definido pelo programador ou pré-definido pela linguagem, e `<lista de tipos enumeraveis>` é uma lista de faixas de números separadas por vírgula.

Por restrição dos tipos enumeráveis, os valores devem ser do tipo ordinal e conhecidos em tempo de compilação.

Os casos mais comuns de matrizes são de uma e de duas dimensões, embora possam ser declaradas com dimensões maiores. As de uma única dimensão são costumeiramente denominadas *vetores*, as de duas dimensões são chamadas de *matrizes* propriamente ditas.

Exemplos:

```
program tipo_array_1;
 const
    MAX = 10; MIN=-5;
 var
    // exemplos de vetores
    alunos: array[MIN..MAX] of string;
    notas: array[1..3] of integer;
    dados: array[0..MAX-1] of real;

    // exemplos de matrizes
    tabela: array[MIN..MAX, 1..10] of real;
    tabuleiro: array[1..3,1..MAX+1] of integer;
    dados: array[0..MAX-1,MIN..MAX] of real;

    // exemplos de array de dimensao 3
    cubo: array[MIN..MAX, 1..10, 0..50] of real;
 end.
```

Para se acessar o conteúdo de um vetor deve-se usar o nome da variável e um

indexador do tipo ordinal entre colchetes.

Exemplos:

```
alunos[2]:= 'Fulano de Tal';
notas[1]:= 95;
read(dados[9]);

tabela[1,5]:= 0.405;
write(tabuleiro[2,1]);
read(dados[0,0]);

cubo[1,2,3]:= PI-1;
```

Tentar acessar um índice fora da faixa definida no tipo enumerável gera erro de execução no programa.

O compilador não aceita que se passe o tipo `array` como parâmetro em funções e procedimentos, por isto normalmente o programador deve declarar um novo tipo para o `array`.

Exemplo:

```
type
   vetor_de_reais = array [1..10] of real;
var
   v: vetor_de_reais;
```

Vale ressaltar que o tipo `string` é uma forma especial de `array` de caracteres (`char`). Suponha uma palavra com 10 letras. Significa que esta `string` contém 10 caracteres. Para acessar cada caractere separadamente, utilize:
`nome_variavel[indice_do_caractere_desejado]`.

Exemplo:

```
var
   s: string;
begin
   s := 'hello world!';
   writeln (s[4]);      // resultado eh a segunda letra L de hello.
end.
```

5.7 Tipo `record` (registro)

O tipo `record` é usado para aglomerar sob um mesmo nome de variável uma coleção de outras variáveis de tipos potencialmente diferentes. Por isto é uma estrutura heterogênea, contrariamente ao tipo `array` que é homogêneo.

A sintaxe para se usar uma variável do tipo `record` é a seguinte:

```
record
   <Id_1> : <tipo_1>;
   <Id_2> : <tipo_2>;
    ..................
   <Id_n> : <tipo_n>;
end;
```

onde `<Id_i>` é um identificador e `<tipo_i>` é um tipo qualquer da linguagem ou previamente definido pelo programador.

Exemplo:

```
program tipo_record_1;
 var
    cliente = record
       nome: string;
       idade: integer;
       cpf: longint;
       sexo: char;
       endereco: string;
       salario: real;
    end;
```

Normalmente é boa política declarar um tipo para o `record`, como é mostrado a seguir:

Exemplo:

```
program tipo_record_1;
 type
    tipo_cliente = record
       nome: string;
       idade: integer;
       cpf: longint;
       sexo: char;
       endereco: string;
       salario: real;
    end;
 var
    cliente: tipo_cliente;
```

5.8 Tipo `file` (arquivo)

O tipo `file` é usado basicamente para armazenamento de dados em memória secundária. Contudo, é possível também se utilizar deste tipo para gravação na memória principal.

Desta forma, o tamanho que os arquivos podem ocupar dependem basicamente do espaço livre em disco ou em RAM.

Um arquivo é declarado da seguinte maneira:

```
var F: file of <tipo>;
```

observando-se que `<tipo>` pode ser uma estrutura de qualquer tipo, exceto o próprio tipo `file`.

É possível não se informar o tipo do arquivo, o que resulta em um arquivo sem tipo. Segundo o guia de referência, isto é equivalente a um arquivo de bytes.

Para se usar uma variável do tipo arquivo, alguns comandos são necessários:

assign
: associa o nome da variável (interno ao programa) a um nome de arquivo (em disco, por exemplo). O nome do arquivo é dependente do sistema operacional;

reset
: abre o arquivo para somente leitura;

rewrite
: abre o arquivo para leitura e escrita;

close
: fecha o arquivo;

Segue um exemplo em que um arquivo de registros (`record`) é usado para armazenar uma agenda em disco.

```
type
     agenda = record
          nome: string;
          rg, fone: longint;
     end;
var
     F: file of agenda;

begin
     assign (F,'~/dados/agenda.db');
     reset (F);
     // comandos que manipulam a agenda
     close (F);
end.
```

Um arquivo especial é o do tipo `text`, que armazena dados que podem ser manipulados pelas bibliotecas padrão de entrada e saída, inclusive as entradas default `input` (teclado) e `output` (tela).

O código a seguir mostra o uso de um arquivo do tipo `text`.

```
var
     F: text;
     i: integer;
begin
     assign (F,'~/dados/entrada.txt');
```

```
      reset (F);
      read (F,i);
      // comandos que manipulam a variavel i
      write (F,i); // imprime i no arquivo texto (ASCII);
      write (i);   // imprime i na tela;
      close (F);
end.
```

6 Expressões lógicas, aritméticas e operadores

Existem dois tipos de expressões em *Pascal*: as lógicas e as aritméticas. São basicamente usadas por comandos quando é necessário produzir um cálculo complexo ou na tomada de decisões em comandos que exigem certos testes.

Expressões são formadas por dois componentes: os operadores e os operandos. Como a linguagem *Pascal* é tipada, ocorre que as expressões também produzem resultados de tipo definido. Este tipo depende essencialmente dos operandos e dos operadores que compõem a expressão.

O restante desta seção detalha os aspectos mais importantes das construções que envolvem expressões.

6.1 Expressões aritméticas

São expressões aritméticas:

- Constantes;
- Variáveis dos tipos ordinal e real;
- Funções cujo tipo de retorno sejam ordinais ou reais;
- Se E_1 e E_2 são duas expressões aritiméticas, então as seguintes também são:
  - $E_1 + E_2$
  - $E_1 - E_2$
  - $E_1 * E_2$
  - E_1/E_2
  - E_1 div E_2
  - E_1 mod E_2
  - (E_1)
- Somente são expressões aritméticas as que podem ser construídas com as regras acima.

Os símbolos `+`, `-`, `*`, `/`, `div`, `mod` são operadores aritméticos binários, isto é, operam sobre dois operandos para produzir um resultado. A tabela abaixo explica cada um destes símbolos.

operador	significado
+	adição
-	subtração
*	multiplicação
/	divisão real
div	divisão inteira
mod	resto de divisão inteira

O tipo do retorno depende do tipo dos operandos e dos operadores. Em geral, as operações de adição e subtração e multiplicação retornam o mesmo tipo dos operandos. Se estiverem sendo operados um real e um inteiro, o resultado é do tipo real.

Para as operações de divisão inteira (`div`) e resto de divisão inteira (`mod`) o resultado é inteiro, mas os operandos também devem ser inteiros. As operações com a divisão real (`/`) resultam sempre em tipo real, mesmo que um ou mais operandos sejam do tipo inteiro.

A tabela abaixo resume esta explicação.

operador	tipo dos operandos	tipo do resultado
+	inteiros	inteiro
	reais	real
	real e inteiro	real
-	inteiros	inteiro
	reais	real
	real e inteiro	real
*	inteiros	inteiro
	reais	real
	real e inteiro	real
/	inteiros	real
	real e inteiro	real
	reais	real
div	inteiros	inteiro
	real e inteiro	ERRO
	reais	ERRO
mod	inteiros	inteiro
	real e inteiro	ERRO
	reais	ERRO

Observamos que, nas operações de divisão real ou inteira, o segundo operando não pode ser nulo, o que acarretará erro de execução.

6.1.1 Exemplos

São exemplos de expressões aritméticas bem formadas:

- `2 * 9 - 8 / 4`
- `2 * (9 - 8 / 4)`
- `cos (x/(y+1)) + ln (exp(y)) / sqrt (2.376 * x - PI)`
- `9 mod 5 + 2 * 7`

- `(9 mod (5 + 2)) * 7`
- `6.45 + 5.0 * 2.01`
- `(6 + 5) * 2`

As funções `cos`, `ln`, `exp` e `sqrt` são pré-definidas pelo compilador e retornam, respectivamente, o cosseno, logaritmo, exponencial e raiz quadrada do argumento.

6.1.2 Ordem de precedência

Em uma expressão complexa, o compilador executa cada operação separadamente segundo uma ordem pré-estabelecida. Isto é conhecido como *ordem de precedência* dos operadores.

Em outras palavras, suponha a expressão:

$$5 + 4 * 2$$

Sem maiores informações, é uma expressão ambígua, pois $5 + 4 = 9$ e $9 * 2 = 18$, mas por outro lado, $4 * 2 = 8$ e $5 + 8 = 13$. Isto reflete a diferença entre as expressões:

$$(5 + 4) * 2$$

e

$$5 + (4 * 2)$$

Em *Pascal* as operações são realizadas segundo a ordem de precedência seguinte:

- as operações entre parênteses ocorrem primeiro;
- em seguida, as de multiplicação ou divisão, seja real ou inteira;
- em seguida, as de adição ou subtração;
- finalmente, realiza operações da esquerda para a direita.

Assim, no exemplo acima $5 + 4 * 2$, significa que a multiplicação será feita em primeiro lugar, e o resultado será 13.

As expressões podem ser utilizadas nos seguintes casos:

- em comandos de atribuição (ver seção 7.1);
- em expressões lógicas (ver seção 6.2);
- como argumento de funções ou procedimentos (ver seção 4.1.5).

Expressão	Resultado
`2 * 9 - 8 / 4`	16
`2 * (9 - 8 / 4)`	14
`(2 * 9) - 8 / 4`	16
`9 mod 5 + 2 * 7`	18
`(9 mod (5 + 2)) * 7`	14
`6 + 5 * 2`	16
`(6 + 5) * 2`	22

Tabela 6: Exemplo de uso de expressões

6.2 Expressões lógicas

São expressões lógicas:

- constantes do tipo `boolean`;
- variáveis do tipo `boolean`;
- funções cujo tipo de retorno seja `boolean`;
- se E_1 e E_2 são duas expressões aritméticas, ou variáveis ou constantes do mesmo tipo, então as seguintes são expressões lógicas:
  - $E_1 = E_2$
  - $E_1 <> E_2$
  - $E_1 > E_2$
  - $E_1 >= E_2$
  - $E_1 < E_2$
  - $E_1 <= E_2$
- Se L_1 e L_2 são duas expressões lógicas, então as seguintes também são:
  - L_1 `or` L_2
  - L_1 `and` L_2
  - L_1 `xor` L_2
  - `not`(L_1)
  - (L_1)
- Somente são expressões aritméticas as que podem ser construídas com as regras acima.

Os símbolos `=`, `<>`, `>`, `>=`, `<`, `<=` são operadores relacionais binários. Podem ter como argumento expressões aritméticas ou então variáveis ou constantes do tipo, mas sempre resultam em tipo `boolean`.

Os símbolos `and`, `or`, `xor` e `not` são operadores booleanos, os dois primeiros são binários e o último é unário. Significam, respectivamente, as operações lógicas de conjunção, disjunção, disjunção exclusiva e negação.

A tabela abaixo explica cada um destes símbolos.

operador	significado
=	igual
<>	diferente
>=	maior ou igual
<	menor
<=	maior ou igual
and	conjunção
or	disjunção
xor	disjunção exclusiva
not	negação

O tipo do retorno sempre é `boolean` e o resultado da operação segue as regras da lógica clássica que são resumidas nas tabelas verdade apresentada na figura 1.

Expressão	Resultado
`false and false`	`false`
`false and true`	`false`
`true and false`	`false`
`true and true`	`true`

(a) AND

Expressão	Resultado
`false or false`	`false`
`false or true`	`true`
`true or false`	`true`
`true or true`	`true`

(b) OR

Expressão	Resultado
`false xor false`	`false`
`false xor true`	`true`
`true xor false`	`true`
`true xor true`	`false`

(c) XOR

Expressão	Resultado
`not false`	`true`
`not true`	`false`

(d) NOT

Figura 1: Conjunto de tabelas verdade dos operadores lógicos

As expressões podem ser utilizadas nos seguintes casos:

- em comandos de atribuição (ver seção 7.1);

- em comandos que exigem um condicional (ver seção 7);
- como argumento de funções ou procedimentos (ver seção 4.1.5).

6.2.1 Exemplos

São exemplos de expressões booleanas bem formadas:

- `1 = 2`
- `'A' = 'a'`
- `5 < 2`
- `3 <= 3`
- `'JOAO' > 'JOSE'`
- `2 + 3 <> 5`
- `'comp' <> 'COMP'`
- `11 > 4`
- `11 > 4`
- `conhece_pascal and conhece_C`
- `reprovouNota xor reprovouPresenca`
- `(media >= 7) and (percentualFaltas <= 0.75)`
- `(idade >= 65) or (tempo_servico >= 25)`
- `not(chovendo) and ((temperatura >= 30) or tem_cerveja)`
- `(a and not(b)) or (b and not(a))`

6.2.2 Ordem de precedência

Assim como acontece com as expressões aritméticas (seção 6.1.2), as expressões lógicas também precisam de uma ordem de precedência entre os operadores para evitar ambigüidades na interpretação da expressão.

- As operações entre parênteses ocorrem primeiro;
- Em seguida, os operadores unários (`not`, `-` (unário));
- Em seguida, os operadores multiplicativos (`*`, `/`, `div`, `mod`, `and`);
- Em seguida, os operadores aditivos (`+`, `-`, `or`, `xor`);

- Em seguida, os operadores relacionais (=, <>, >, >=, <, <=);
- Finalmente, realiza operações da esquerda para a direita, em geral.

Observação Segundo o guia de referência, em sua versão de dezembro de 2008, o compilador *Free Pascal*, contrariamente ao *Turbo Pascal*, não consegue garantir a precedência quando avalia expressões para acontecerem da esquerda para a direita, informando que o compilador decidirá qual avaliar primeiro baseado em regras de otimização. Assim, na expressão seguinte:

```
a := g(3) + f(2);
```

$f(2)$ pode ser executada antes de $g(3)$. Se o programador quiser garantir que $g(3)$ ocorra antes, deve escrever seu código assim:

```
e1 := g(3);
a  := e1 + f(2);
```

Sempre que for feita a construção de expressões lógicas contendo expressões aritméticas em conjunto com operadores relacionais, as sentenças contendo operadores relacionais precisam estar entre parênteses, ou o compilador vai gerar erro de execução. Por exemplo, na expressão:

```
a < 0 or b < 0
```

o operador `or` possui maior precedência em relação aos operadores relacionais. Assim, o compilador tenta realizar primeiro a expressão `0 or b`. O operador `or` não foi construído para operar com dados numéricos, mas apenas com dados do tipo `boolean`, o que gera erro.

A maneira correta de construir a expressão é utilizar parênteses para separar as sentenças relacionais, da seguinte maneira:

```
(a < 0) or (b < 0)
```

Deste modo, como os parênteses possuem maior precedência em relação ao `or`, as sentenças relacionais são reduzidas a valores lógicos antes da utilização do operador.

7 Comandos da linguagem *Pascal*

Existem dois tipos de comandos em *Pascal*, os comandos simples e os compostos. Neste texto vamos apresentar apenas os comandos compatíveis com o nível da disciplina básica de programação de computadores ou de introdução aos algoritmos.

Por este motivo, recomendamos fortemente que o leitor consulte o guia de referência do compilador utilizado para conhecer todos os comandos disponíveis na linguagem.

Os comandos que serão apresentados são os seguintes:

- **Comandos Simples**
  - Comando de atribuição (`:=`);
  - Comando de desvio incondicional (`goto`);
- **Comandos Compostos**
  - Comandos de desvio condicional (`if-then-else`, `case`);
  - Comandos de repetição (`while-do`, `repeat-until`, `for`);
  - Delimitadores de bloco.

7.1 Comando de atribuição

O comando de atribuição é usado exclusivamente para se atribuir um valor a uma variável. O valor previamente armazenado naquela posição de memória é substituído pelo novo.

Sintaxe:

```
<var> := <expressao>;
```

Com isto, ocorre a atribuição do resultado da avaliação da expressão para a variável indicada lado esquerdo do operador de atribuição (`:=`).

Em função da linguagem ser tipada, as atribuições só funcionam se o tipo do valor atribuído for o mesmo da variável que recebe o novo valor. Quando ocorre uma atribuição de tipos diferentes, o programa aborta com uma mensagem do tipo *type mismatch*. Há vezes em que o próprio compilador detecta incompatibilidades de tipos.

Exemplo:

```
program exemplo_atribuicao;
 var
   x, alpha: real;
   m, n, fatorial: integer;
   nome: string;
   letra: char;
begin
   x := 10.75;
```

```
    alpha := sqrt (ln (y) + beta) - y * (y + 1);
    m := m + n * n DIV (fatorial - 1);
    nome := 'Fulano de Tal';
    letra := 'A';
end.
```

7.2 Comandos de entrada

Os comandos de entrada são usados para se carregar um determinado valor fornecido pelo usuário em memória. Este valor pode ser digitado ou fornecido através de arquivo.

Os comandos de entrada da linguagem *Pascal* são `read` e `readln`, cujas sintaxes respectivas são apresentadas na seqüência[11].

Sintaxe:

```
read(<dispositivo>, <lista de variaveis>);
readln(<dispositivo>, <lista de variaveis>);
```

em que `<lista de variáveis>` é uma lista de identificadores de variáveis separados por vírgula, e `<dispositivo>` é o nome de um dispositivo de entrada (ver seção 5.8). Caso o nome do dispositivo seja omitido, então se considera a entrada padrão (teclado).

As variáveis devem ser de um tipo básico da linguagem, isto é, tipos ordinais, reais, `char`, `string` ou `boolean`.

A diferença entre as duas formas do comando `read` só existe no caso de leitura de `strings`: `readln` acrescentará o CR (*carriage return*) ao final da `string` lida, como parte da mesma.

Exemplo:

```
program exemplo_leitura;
 var
    x, alpha: real;
    m, n, fatorial: integer;
    nome: string;
    letra: char;
    arq: file of real;
 begin
    readln(nome);
    read(x);
    read(arq,alpha);
    read(m,n,fatorial,letra);
 end.
```

[11] O guia de referência do *Free Pascal*, em sua versão de dezembro de 2008, não contém informações sobre os comandos de leitura, apenas menciona que são *modificadores*.

7.3 Comandos de saída

Os comandos de saída são usados para se exibir conteúdos de variáveis, bem como mensagens que devem ser fornecidas para o usuário do programa.

Os comandos de saída da linguagem *Pascal* são `write` e `writeln`, conforme sintaxes seguintes[12]:

Sintaxe:

```
write(<dispositivo>, <lista>);
writeln(<dispositivo>, <lista>);
```

em que `<lista>` é uma lista de elementos separados por vírgula, em que cada elemento pode ser um identificador de variável, uma expressão aritmética ou uma `string` (entre aspas simples) e `<dispositivo>` é o nome de um dispositivo de saída (ver seção 5.8). Tanto o dispositivo quanto a lista de variáveis podem ser vazios.

Caso o nome do dispositivo seja omitido, então se considera a saída padrão (tela). Um dispositivo interessante é o `lst`, que remete à impressora padrão.

As variáveis devem ser de um tipo básico da linguagem, isto é, tipos ordinais, reais, `char`, `string` ou `boolean`.

A diferença entre as duas formas do comando `write` é que `writeln` acrescentará o CR (*carriage return*) ao final do conteúdo impresso. Isto faz com que o cursor seja levado à próxima linha do dispositivo de saída.

Exemplo:

```
program exemplo_leitura;
 var
    x, alpha: real;
    m, n, fatorial: integer;
    nome: string;
    letra: char;
    arq: file of real;
 begin
     // Trecho com a leitura das variaveis omitido

     writeln('Nome do usuario: ', nome);
     write(x, ' = ', f(2) + ln (alpha));
     write(arq, alpha);
     write(lst, m, ' + ', n, '     ', fatorial, letra);
     write ('       ');
     writeln;
 end.
```

7.4 Comando de desvio incondicional (`Goto`)

O comando de desvio incondicional é o `goto` e serve para desviar o fluxo de execução do programa incondicionalmente para outro ponto do código. Este outro ponto deve

[12] O guia de referência do *Free Pascal*, em sua versão de dezembro de 2008, não contém informações sobre os comandos de saída, apenas menciona que são *modificadores*.

ser indicado por um rótulo previamente declarado na seção de rótulos (ver seção 4.1.1).

Sintaxe:

```
goto <label>
```

em que `<label>` é um rótulo declarado com o comando `label`.

Atenção: é preciso observar que o comando de desvio incondicional não faz parte dos comandos aceitos no contexto de programação estruturada. Sempre é possível reconstruir o código de maneira mais elegante (embora talvez menos eficiente) e obter o mesmo resultado.

Outra observação é que o uso do `goto` de dentro para fora de laços ou vice-e-versa não é algo muito razoável de se fazer, a chance de o programador estar fazendo algo errado é bastante alta.

Exemplo:

```
program alomamae;
 label
    alo, 100, FIM;
 begin
    goto 100;

    alo : writeln ('Alo mamae!');
          goto fim;

    100 : writeln ('Inicio!');
          goto alo;
    FIM :
 end.
```

Este programa inicia pelo comando que desvia o código para o rótulo `100`, que imprime a mensagem "Inicio" na tela. Em seguida executa o desvio para o rótulo `alo`, que imprime na tela a mensagem "Alo mamae!". Finalmente, desvia o código para o rótulo `fim` e o programa termina. Observe que o exemplo seguinte cumpre exatamente com a mesma finalidade.

Exemplo:

```
program alomamae_melhorado;
 begin
    writeln ('Inicio!');
    writeln ('Alo mamae!');
 end.
```

7.5 Comandos de desvio condicional

Os comandos de desvio condicional servem para desviar o fluxo de execução dos comandos com base em uma condição definida pelo programador.

Existem dois comandos de desvio condicional em *Pascal*. Um deles possibilita duas formas de uso:

1. `if-then`;
2. `if-then-else`;
3. `case-of`.

Cada forma de uso será apresentada nas seções subseqüentes.

7.5.1 Desvio condicional simples (`if-then`)

Nesta primeira variante, é possível executar um comando desejado apenas se uma condição booleana for satisfeita, ou seja, for avaliada como `true` (verdadeira). Caso contrário (o resultado seja `false`), o comando não é executado.

Sintaxe:

```
if <expressao booleana> then
   comando>;
```

Note que as palavras `if` e `then` são reservadas. O comando a ser executado é único. Se o programador precisar de uma seqüência de comandos, então os delimitadores de bloco devem ser empregados (ver seção 7.7). Observe também que, como em *Pascal* todo comando termina por ponto-e-vírgula e, neste caso, o comando `if-then` termina somente após `<comando>` aninhado, então o ponto-e-vírgula somente aparece no final de toda a construção.

Exemplo:

```
program exemplo_condicional_1;
 var
  n: integer;
 begin
  writeln('Informe um numero inteiro positivo: ');
  read(n);
  if (n mod 2) = 0 then
     writeln('O numero ', n,' eh par.');

    writeln ('fim');
 end.
```

O programa anterior imprime a mensagem "O numero n eh par." apenas e tão somente se o resto da divisão inteira de `n` por 2 for nulo. Em ambos os casos a mensagem "fim" é impressa na tela.

Caso o programador queira imprimir também a mensagem "fim" apenas no caso da avaliação do condicional, o programa deveria ter sido escrito desta outra forma que segue.

Exemplo:

```
program exemplo_condicional_2;
 var
  n: integer;
 begin
  writeln('Informe um numero inteiro positivo: ');
  read(n);
  if (n mod 2) = 0 then
        begin
           writeln('O numero ', n,' eh par.');
           writeln ('fim');
        end;
 end.
```

7.5.2 Desvio condicional completo (`if-then-else`)

Nesta segunda e última variante, é possível executar um comando desejado apenas se uma condição booleana for satisfeita, ou seja, for avaliada como `true` e ao mesmo tempo garantir que um segundo comando seja executado tão somente se a condição booleana for avaliada como `false`.

Sintaxe:

```
if <expressao booleana> then
   <comando_1>
else
   <comando_2>;
```

Note que as palavras `if`, `then` e `else` são reservadas. O comando a ser executado é único, tanto o primeiro quanto o segundo. Se o programador precisar de uma seqüência de comandos, então deve usar um delimitador de bloco (ver seção 7.7). Observar também que, como em *Pascal* todo comando termina por ponto-e-vírgula e, neste caso, o comando `if-then-else` termina somente após `<comando_2>`, então o ponto-e-vírgula aparece apenas no final, e não no meio da construção, o que é um erro comum para principiantes.

Exemplo:

```
program exemplo_condicional_3;
 var
  n: integer;
 begin
  writeln('Informe um numero inteiro positivo: ');
  read(n);
  if (n mod 2) = 0 then
        writeln('O numero ', n,' eh par.')
    else
        writeln('O numero ', n,' eh impar.');

    writeln ('fim');
 end.
```

O programa anterior imprime a mensagem "O numero n eh par." apenas e tão somente se o resto da divisão inteira de `n` por 2 for nulo. Caso contrário, a mensagem "O numero n eh impar." será impressa. Independentemente do desvio realizado, a mensagem "fim" é exibida na tela.

Assim, uma maneira de distingüir o desvio condicional simples do completo, é visualizar a `else` como parte opcional do comando `if`.

No exemplo a seguir, o programa gera um *menu* de opções. Existe uma outra forma mais adequada de fazer o mesmo *menu*, utilizando o comando **case**, que será visto na seção 7.5.3.

Exemplo:

```
program par_impar;
 var
  opc: integer;
 begin
  writeln('------ M E N U ------');
  writeln(' 1 - Cadastro');
  writeln(' 2 - Busca');
  writeln(' 3 - Ajuda');
  writeln(' 4 - Sair');
  writeln('--------------------');
  read(opc);

  if (opc = 1) then
       writeln('Cadastrando...')
  else if (opc = 2) then
     writeln('Buscando...')
  else if (opc = 3) then
       writeln('Mostrando ajuda...')
  else if (opc = 4) then
       writeln('Saindo do programa...')
  else
       writeln('Opcao Invalida');
 end.
```

7.5.3 Comando de desvio condicional por caso (`case`)

O comando `case` serve para facilitar a visualização de vários comandos `if-then-else` aninhados, com a diferença de que, no lugar de avaliar expressões booleanas, são avaliadas expressões cujo resultado seja de algum tipo enumerado ou ordinal (exceto `boolean` e `char`).

O comando é constituído por uma *expressão* e por *constantes* de caso. As constantes devem ser conhecidas em tempo de compilação. Todas as constantes devem ser do mesmo tipo.

O compilador avalia a expressão e, caso haja o casamento de uma das constantes com o valor da expressão, a sentença que segue esta constante é executada. O programa continua então após o `end` final do comando.

Na situação em que não há casamento de nenhuma constante com o valor da expressão, executam-se as instruções aninhadas em `else`.

O trecho do `else` é opcional e executado apenas no caso de não haver qualquer casamento de constante com o valor da expressão. Ainda neste caso, se não houver este trecho, o fluxo de execução seguirá para o comando que sucede o `end`.

Sintaxe:

```
case <expressao do case> of
   cte1: comando_1;
   cte2: comando_2;
   cte3: comando_3;
   //...
   cteN: comando_N;
else
   comando_do_else;
end;
```

Observe que `case`, `of` e `else` são palavras reservadas. As constantes do comando são `cte1`, `cte2`, ..., `cteN`. Conforme explicado anteriormente, estas constantes podem ser também de tipos enumeráveis, isto é, da forma `A..B`. No *Free Pascal* não é permitido haver duplicidade nas constantes.

Analogamente aos comandos `if`, aqui somente um único comando é executado em cada caso. Quando necessário mais de um comando, deve-se utilizar os delimitadores de bloco (ver seção 7.7).

Ao contrário do que ocorre com o `if`, aqui deve existir o ponto-e-vírgula na instrução que precede o `else`.

Em suma, o comando `case` funciona baseado em casamentos de padrões e não é tão genérico quando o comando `if`. Este, por sua vez, trabalha com expressões booleanas que podem ser tão complexas quanto se precise. Vale frisar que todo trecho que utilize `case` pode ser reescrito como um aninhamento de comandos `if-then-else`, embora o código fique menos legível.

Exemplo:

```
program exemplo_case_of;
 var
    a, b : integer;
    resposta, escolha : byte;
 begin
    writeln('Digite dois inteiros:');
    read(a, b);
    writeln('Escolha a operacao a ser realizada: ');
    writeln('(1) Soma, (2) Subtracao, (3) Multiplicacao, (4) divisao)');
    read(escolha);

    case escolha of
       1 : writeln(a, ' + ', b, ' = ', (a+b));
       2 : writeln(a, ' + ', b, ' = ', (a-b));
       3 : begin
              resposta := a * b;
```

```
            writeln(a, ' + ', b, ' = ', resposta);
         end;
      4: writeln(a, ' + ', b, ' = ', (a/b));
      5..255:
         writeln ('Escolha errada!');
   else
         writeln('Entrada incorreta.');
end.
```

O código anterior, escrito com o `case` equivale a este outro código, que utiliza `if`.

```
program operacoes;
 var
    a, b : integer;
    resposta : integer;
    escolha : integer;
 begin
    writeln('Digite dois inteiros:');
    read(a, b);
    writeln('Escolha a operacao a ser realizada: ');
    writeln('(1) Soma, (2) Subtracao, (3) Multiplicacao, (4) divisao)');
    read(escolha);

    if escolha = 1 then
       writeln(a, ' + ', b, ' = ', (a+b))
    else
       if escolha = 2 then
          writeln(a, ' + ', b, ' = ', (a-b))
       else
          if escolha = 3 then
             begin
                resposta := a * b;
                writeln(a, ' + ', b, ' = ', resposta);
             end
          else
             if escolha = 4 then
                writeln(a, ' + ', b, ' = ', (a/b))
             else
                if (escolha > 5) and (escolha <= 255) then
                   writeln('Escolha errada!')
                else
                   writeln('Entrada incorreta.');
end.
```

O próximo exemplo explora mais as possibilidades do `case`.

```
program exemplo_case_sofisticado;
 var
    numero : integer;
 begin
    writeln('Digite um numero:');
    read(numero);
```

```
    case numero of
       1 : writeln('Numero um.');
       2..3, 5, 7, 11, 13, 17, 19, 23, 29:
           writeln('Numero esta entre os 10 primeiros primos.');
       4, 8, 12, 16, 20, 24, 28:
           writeln('Numero menor do que 30 e divisivel por quatro');
    else
       writeln('Nada de especial com esse numero');
    end;
end.
```

No exemplo acima, toda vírgula dentro da especificação das constantes é considerada um `or`. Sempre que forem encontrados dois pontos entre constantes, lê-se **até**, pois especifica uma escala de possíveis valores.

7.6 Comandos de repetição

Os comandos de repetição servem para desviar o fluxo de execução de maneira que um determinado trecho do código possa ser repetido por um número determinado de vezes. A este desvio, dá-se o nome de *laço* ou *loop*.

O *Pascal* dispõe dos seguintes comandos de repetição:

1. `while-do`;
2. `repeat-until`;
3. `for-do`.

Todos estes serão apresentados na seqüência.

7.6.1 Repetição condicional com teste no início (`while-do`)

Em *Pascal*, o `while-do` é o comando de repetição condicional com teste no início. Usa-se o `while-do` para repetir um comando enquanto a avaliação de sua expressão booleana resulte em verdade (`true`). Tal expressão é avaliada antes do comando ser executado.

Sintaxe:

```
while <expressao booleana> do
   comando>;
```

As palavras `while` e `do` são reservadas. O comando a ser executado é único. Caso o programador deseje executar uma seqüência de comandos, o delimitador de blocos deve ser utilizado (ver seção 7.7).

Como, em *Pascal*, todo comando termina por ponto-e-vírgula e, neste caso, o comando `while-do` termina somente após o `<comando>`, então o ponto-e-vírgula somente aparece no final, e não no meio da construção.

Exemplo:

```
program exemplo_while_do_1;
 var
    n : integer;
 begin
    writeln('Informe um numero inteiro positivo: ');
    read(n);

    while n > 0 do
       begin
          writeln(n);
          n := n - 1;
       end;
    writeln ('Fim.');
 end.
```

O programa imprime, de maneira decrescente, todos os números entre `n` e `1`. Se o número lido for negativo ou nulo, então a mensagem "Fim." é impressa imediatamente, isto é, nenhum número aparece na tela. Isto ocorre porque a avaliação da expressão condicional ocorre antes do bloco de instruções ser executado.

Como o `while-do` repete um bloco de instruções enquanto determinada condição for verdadeira, é possível definir um laço de execução infinita caso a expressão avaliada nunca se altere para falsa.

No código a seguir, se um número negativo ou nulo for digitado como entrada, o programa termina sem nunca ter impresso número algum na saída. Contudo, se o número entrado for positivo e não nulo, será impresso infinitamente.

Exemplo:

```
program exemplo_while_do_2;
 var
    n : integer;
 begin
    writeln('Informe um numero inteiro positivo: ');
    read(n);

    while n > 0 do
       writeln(n);
    writeln ('Fim.');
 end.
```

7.6.2 Repetição condicional com teste no final (`repeat-until`)

O `repeat-until` é o comando de repetição condicional com teste no final. Funciona de modo contrastante ao `while-do`, ou seja, repete até que uma dada expressão booleana resulte em verdadeiro. Esta expressão é avaliada *após* o bloco de instruções ter sido executado.

Sintaxe:

```
repeat
```

```
    <comandos>;
until <expressao booleana>;
```

As palavras `repeat` e `until` são reservadas. Contrariamente ao comando `while-do`, o `repeat-until` pode executar vários comandos sem a delimitação de bloco por `begin` e `end` (ver seção 7.7).

Como, em *Pascal*, todo comando termina por ponto-e-vírgula e, neste caso, o comando `repeat-until` termina somente após a expressão booleana, então o ponto-e-vírgula somente aparece no final, e não no meio da construção.

Exemplo:

```
program exemplo_repeat_until_1;
 var
    n : integer;
 begin
    writeln('Informe um numero inteiro positivo: ');
    read(n);

    repeat
       writeln(n);
       n := n - 1;
    until n <= 0;
    writeln ('Fim.');
end.
```

O programa imprime de maneira decrescente todos os números entre `n` e `1`. Contudo, aqui reside a principal diferença entre o `repeat-until` e o `while-do`: se o número lido for negativo ou nulo, o número será impresso seguido da mensagem de "Fim.". Isto ocorre porque a avaliação da expressão somente é feita *após* a execução do bloco de comandos.

Dependendo da avaliação da expressão booleana, é possível definir um laço de execução infinita, tal como apresentado no exemplo a seguir. Um inteiro positivo, não nulo, lido do teclado é imprimido infinitamente. Caso um número nulo ou negativo seja lido, o programa termina após imprimir o número uma única vez e exibir a mensagem "Fim.".

Exemplo:

```
program exemplo_repeat_until_2;
 var
    n : integer;
 begin
    writeln('Informe um numero inteiro positivo: ');
    read(n);

    repeat
       writeln(n);
    until n <= 0;
    writeln ('Fim.');
 end.
```

7.6.3 Repetição por enumeração (`for-do`)

O `for-do` é o comando de repetição por enumeração do *Pascal*. Usa-se para repetir um bloco de comandos por um número fixo e *predeterminado* de vezes. O controle é feito por uma variável, denominada *variável de controle*, a qual respeita os limites estabelecidos que dependem de um valor inicial e de um valor final. Tais limites são definidos antes do comando ser executado e não podem ser alterados.

Sintaxe:

```
for <variavel_de_controle> := <EAO_1> to <EAO_2> do
   <comando>;

for <variavel_de_controle> := <EAO_1> downto <EAO_2> do
   <comando>;
```

As palavras `for`, `do`, `to` e `downto` são reservadas. O comando a ser executado é único, devendo ser utilizados os delimitadores de bloco (ver seção 7.7) no caso de haver mais de um.

Como, em *Pascal*, todo comando termina por ponto-e-vírgula e, neste caso, o comando `for-do` termina somente após `<comando>`, então o ponto-e-vírgula somente aparece no final, e não no meio da construção.

O comportamento do `for-do` é o seguinte:

1. o valor de `<EAO_2>` é avaliado;
2. a variável de controle recebe o valor da avaliação de `<EAO_1>`;
3. caso tenha sido usada a forma com `to`, então o valor da variável de controle é comparado com `<EAO_2>` e, se for menor ou igual, o comando é executado e a variável de controle é incrementada de uma unidade;
4. caso tenha sido usada a forma com `downto`, então o valor da variável de controle é comparado com `<EAO_2>` e se for maior ou igual, o comando é executado e a variável de controle é decrementada de uma unidade;
5. o ciclo se repete até que o valor da variável de controle ultrapasse o limite estabelecido em `<EAO_2>`, no caso da forma com `to` ou se torne inferior a `<EAO_2>` no caso da forma com `downto`.

É imperativo dizer que `<variavel_de_controle>` deve ser de tipo ordinal e que `<EAO_1>` e `<EAO_2>` são expressões aritméticas cujos valores de retorno também são de um tipo ordinal. Estas duas expressões são avaliadas *antes* que o comando seja executado, sendo que é proibido que se modifique o valor da variável de controle pelo comando. Finalmente, não adianta alterar variáveis que mudem o valor de `<EAO_2>`, uma vez que esta expressão foi avaliada *antes* do início do laço.

Exemplo:

```
program exemplo_for_do_1;
var
   i, n: integer;
begin
   writeln('Informe um numero inteiro positivo: ');
   read(n);

   for i:= 1 to n do
      writeln(i);
  writeln ('Fim.');
end.
```

O programa imprime de maneira crescente todos os números entre `1` e `n`. Se o número lido for negativo ou nulo então a mensagem "Fim." é impressa imediatamente, isto é, nenhum número aparece na tela.

Exemplo:

```
program exemplo_for_do_2;
var
   i, n: integer;
begin
   writeln('Informe um numero inteiro positivo: ');
   read(n);

   for i:= n downto 1 do
      writeln(n);
   writeln('Fim.');
end.
```

O programa anterior imprime de maneira decrescente todos os números entre `n` e `1`. Se o número lido for negativo ou nulo então a mensagem "Fim." é impressa imediatamente, isto é, nenhum número aparece na tela. Este programa é absolutamente equivalente ao primeiro exemplo apresentado na seção 7.6.1.

Exemplo:

```
program tabuada_1_ao_10;
 const
    MAX = 10;
 var
  i, j : integer;
 begin
  for i:= 1 to MAX do
     begin
          writeln('Tabuada do ', i);
          for j := MAX downto 1 do
             writeln(i, ' x ', j, ' = ', i * j);
      writeln;
     end;
end.
```

Este último exemplo mostrou como comandos do tipo `for-do` podem ser alinhados para se obter a impressão de todas as tabuadas do 1 ao 10.

Não é possível definir um laço de execução infinita usando-se o comando `for-do`.

7.7 Delimitadores de bloco

Os delimitadores de bloco serve para agrupar comandos a fim de que sejam visto como um conjunto de instruções encapsulado. Isto é, aglomera mais de um comando em um bloco que é encarado pelo compilador como um único comando.

O modo de usar é simples, basta escrever os comandos desejados entre um `begin` e um `end;`.

O seguinte trecho de programa ilustra um uso deste tipo de comando.

```
begin
   a := b + 1;
   b := a - 1;
   if a > b then
      writeln ('A maior do que B');
end;
```

8 Legibilidade do código

Esta seção apresenta algumas técnicas para se escrever códigos elegantes, de fácil leitura e compreensão por quem os lê. Apesar de ter a linguagem *Pascal* como foco, o conteúdo aqui descrito pode ser projetado para qualquer outra linguagem de programação cujo compilador ignore espaços, tabulações ou linhas em branco. Em particular, também se apresentam considerações para as linguagens que não são *case sensitive*, isto é, que não diferenciam maiúsculas de minúsculas.

As próximas seções descrevem convenções e sugestões que ajudam a padronizar o código e que devem ser levadas em consideração.

As idéias aqui apresentadas podem não fazer muito sentido para um programador aprendiz, mas são fundamentais na arte da programação, em particular quando se escrevem programas com algumas milhares de linhas.

8.1 Espaços e linhas em branco são ignorados

Ambos os exemplos geram rigorosamente o mesmo código executável, inclusive com o mesmo tamanho em bytes, pois o compilador ignora linhas em branco e espaços. Contudo, a legibilidade da primeira versão é privilegiada.

Versão 1:

```
program OiPessoal;
begin
   write('Oi ');
   writeln('Pessoal!');
end.
```

Versão 2:

```
program OiPessoal; begin write('Oi '); writeln('Pessoal!'); end.
```

Versão 3:

```
program
OiPessoal
;
begin

write

('Oi '
          )
;
writeln('Pessoal!')
;
end
.
```

Versão 4:

```
program OiPessoal          ;
begin                                          write('Oi ');
writeln('Pessoal!');          end.
```

Em resumo, é preferível escrever um comando por linha, preferencialmente deixando espaços em branco de maneira regular. Usar uma linha em branco entre blocos com o mesmo sentido, também é boa prática: inicialização de variáveis, leitura de variáveis, processamento das informações, por exemplo.

É muito importante que o programador mantenha a consistência do padrão por toda a redação do programa.

8.2 Não há distinção entre maiúsculas e minúsculas

Embora ambas as versões seguintes gerem o mesmo programa executável, a versão com comandos e identificadores em maiúsculas tem a legibilidade prejudicada.

Versão 1:

```
program calcular_imc;
var
   nome : string;
   altura, peso, imc : real;
begin
   write('Digite o nome do paciente: ');
   readln(nome);
   write('Digite a altura em metros (Ex.: 1.78): ');
   readln(altura);
   write('Digite o peso em quilogramas (Ex.:93): ');
   readln(peso);
   writeln;

   imc := peso/(altura*altura);
   writeln('O paciente ', nome, ' tem o I.M.C. igual a ', imc:4:2);

   write('Pressione qualquer tecla para sair.');
end.
```

Versão 2:

```
PROGRAM CALCULAR_IMC;
 VAR
    NOME : STRING;
    ALTURA, PESO, IMC : REAL;
 BEGIN
    WRITE('DIGITE O NOME DO PACIENTE: ');
    READLN(NOME);
    WRITE('DIGITE A ALTURA EM METROS (EX.: 1.78): ');
    READLN(ALTURA);
```

```
    WRITE('DIGITE O PESO EM QUILOGRAMAS(EX.: 93): ');
    READLN(PESO);
    WRITELN;

    IMC := PESO/(ALTURA*ALTURA);
    WRITELN('O PACIENTE ', NOME, ' TEM O IMC IGUAL A ', IMC:4:2);

    WRITE('PRESSIONE QUALQUER TECLA PARA CONTINUAR.');
END.
```

A leitura do código fica mais fluente quando letras minúsculas são utilizadas.

8.3 Comentários

A maior parte das linguagens de programação modernas permite que o programador insira textos que não façam parte do programa propriamente dito, são comentários adicionais que facilitam a compreensão por parte de quem lê o código.

Conforme vimos na seção 5.3, em *Pascal* isto é possível com o uso equilibrado dos símbolos: `(*` e `*)`; { e }; e `+`
`+`.

Exemplo:

```
(* Este programa faz o calculo do MDC pelo metodo de Euclides *)
(* Feito por: Marcos Castilho                                  *)
(* Data: 12/05/2009                                            *)
program MDCporEuclides;
 var
    a, b, resto: longint; // 2 bytes eh pouco para numeros grandes

 begin
    read (a,b); // os valores lidos devem ser nao nulos
    if not ((a = 0) or (b = 0)) then
       begin
          resto:= a mod b;
          while resto <> 0 do // quando o resto for zero temos o MDC em
             b
             begin
                a:= b;
                b:= resto;
                resto:= a mod b;
             end;
             writeln ('MDC = ', b);
       end
    else
       writeln('O algoritmo nao funciona para entradas nulas');
end.
```

Algumas dicas para uso dos comentários no código:

- Ausência de comentários é ruim, mas o excesso também é! Use o bom senso para

decidir os lugares relevantes no código que merecem comentários. Se o código precisa de muitos comentários, provavelmente está mal escrito;

- Não escreva comentários longos demais, a idéia é manter o código o mais limpo possível;
- Descrever o comportamento de funções e procedimentos é interessante;
- Indicar pontos no código que são fundamentais para o bom funcionamento do programa, também é.

8.4 Identação ou alinhamento de código por colunas

Identação é uma palavra que vem do inglês *identation* e significa, neste contexto, o alinhamento de comandos em colunas de acordo com o nível hierárquico de aninhamento destes comandos. Identação é uma prática em programação estruturada e é viável em compiladores que ignoram espaços ou linhas em branco.

Em geral, é boa prática de programação identar blocos de comandos de maneira que, visualmente, o programa fique mais claro e mostre quais linhas de código são executadas por um determinado comando em um certo trecho do programa.

O objetivo é permitir que programas fiquem melhor apresentados e mais fáceis de se entender.

Sempre que um bloco de comandos estiver no escopo de um comando (significa estar sob o controle do comando), a identação passa a ser interessante. A técnica elementar é deslocar este bloco mais a direita do que o comando principal. Isto facilita muito a visualização de diferentes escopos no programa.

Os dois exemplos abaixo ilustram esta situação, o primeiro está bem identado, o segundo não.

Exemplo:

```
program exemplo_identacao_boa;
 var
    i: integer;

 begin
    for i:= 1 to 10 do
       write (i);
    writeln ('Fim.');
 end.
```

Observe como o comando `for` foi deslocado 3 espaços para a direita com relação ao alinhamento `begin-end` do programa, bem como o comando `write` foi deslocado outros 5 espaços com relação a coluna do comando `for` (e portanto 6 com relação a ao `begin-end`). Já o comando que imprime a mensagem de fim ficou alinhado na mesma coluna do comando `for`, mostrando que está fora do seu escopo, isto é, consiste em um comando independente do `for`.

Exemplo:

```
program exemplo_identacao_ruim;
 var
    i: integer;
 begin
    for i:= 1 to 10 do
       write (i);
       writeln ('Fim.');
 end.
```

O motivo da identação confusa do segundo exemplo é que, visualmente, quem lê o código tem a falsa impressão de que o comando que imprime o texto "Fim." está no escopo do comando `for` e portanto será repetido 10 vezes, o que não é verdade!

Se esta fosse de fato a intenção do programador, então ele deveria ter escrito da maneira como é mostrado no exemplo abaixo:

Exemplo:

```
program corrigindo_identacao_ruim;
 var
    i: integer;

begin
    for i:= 1 to 10 do
       begin
          write (i);
          writeln ('fim');
       end;
end.
```

Isto é, introduzindo o `begin` e o `end` que o programador provavelmente tinha esquecido.

O número de espaços do deslocamento depende da preferência do programador. Costuma-se utilizar entre 3 e 5 espaços em branco. Há programadores que utilizam a tabulação (`tab`), porém esta pode ser desconfigurada conforme o editor.

O importante é manter o mesmo número de espaços em todo o código. Deve-se também ter cuidado para não se avançar mais do que a última coluna da direita, pois o resultado também é confuso, conforme exemplo seguinte.

Exemplo:

```
program outro_exemplo_identacao_ruim;
 var
          i : integer;
 begin
          for i:= 1 to 10 do
                   for j:= 1 to 10 do
                            for k:= 1 to 10 do
                                     for l:= 1 to 10 do
                                              write (i, j, k, l);
 end.
```

8.5 Realce de sintaxe

O realce de sintaxe (*syntax highlighting*) [Syn] é uma funcionalidade presente na maioria dos editores de texto para programação. Essa funcionalidade permite que cada categoria de termos da linguagem seja apresentada na tela com um realce (formato ou cor) específico.

Geralmente é utilizada coloração para destacar os termos, mas também podem ser utilizados negrito, itálico ou outras fontes.

Dessa forma, comentários, tipos básicos da linguagem, números e comandos são realçados, facilitando a leitura do código fonte.

É importante lembrar que o realce não afeta o significado do código fonte, e sua única função é facilitar a leitura e edição. Afinal de contas, o compilador funciona sobre um arquivo totalmente escrito usando-se os caracteres visíveis da tabela ASCII.

8.6 Simplicidade e clareza

Simplicidade e clareza são dois atributos muito importantes para aumentar a legibilidade do código-fonte. A clareza torna o programa inteligível e a simplicidade evita que a leitura do programa seja enfadonha.

Entretanto, esses atributos são relativamente subjetivos e a habilidade de escrever programas simples, claros e eficientes só é alcançada com muita prática, o que exige dedicação e tempo. É necessário que o programador tenha um bom conhecimento em algoritmos, bem como doss recursos que a linguagem de programação fornece, pois isto ajuda na tarefa de alcançar o bom senso necessário para se escrever códigos elegantes.

8.7 Nomes dos identificadores

Em linguagens de programação, os identificadores são trechos de texto que nomeiam entidades da linguagem. Em *Pascal*, essas entidades são os nomes das variáveis, constantes, dos tipos de dados, procedimentos e funções.

Assim, ao se programar em *Pascal* é necessário escolher um nome para cada uma dessas entidades, com exceção dos tipos básicos da linguagem. A escolha do nome dos identificadores é muito importante na legibilidade do código-fonte, pois não ajuda muito ter um código identado, comentado e com a sintaxe realçada, se os nomes dos identificadores complicarem a leitura do código.

Por exemplo, se uma determinada variável for armazenar o CPF de algum usuário, um bom nome para esta variável é `cpf`. Observe como outras escolhas poderiam ir contra a legibilidade do código:

- `numero_cadastro_pessoa_fisica`;
- `cadastro`;
- `numero`;
- `campo02`.

A regra geral é a simplicidade e a clareza, como mencionado anteriormente. O ideal é que a leitura do código seja intuitiva, e se pareça mais com um texto em língua nativa do que com um código de máquina.

Referências

[Car82] S. Carvalho. *Introdução à Programação com Pascal*. Editora Campus, 1982.

[FBF+86] Harry Farrer, Christiano Gonçalves Becker, Eduardo Chaves Faria, Frederico Ferreira Campos Filho, Helton Fábio de Matos, Marcos Augusto dos Santos, and Miriam Lourenço Maia. *Pascal Estruturado*. Livros Técnicos e Científicos Editora S.A., 2nd edition, 1986.

[Gui] Manuais on-line do freepascal. Disponíveis juntamente com o compilador em `http://www.freepascal.org`.

[Rin92] Roberto Rinaldi. *Turbo Pascal 6.0 - Comandos e Funções*. Livros Érica Editora Ltda, 1992.

[Syn] `http://en.wikipedia.org/wiki/Syntax_highlighting`. Acessado em 7 de Maio de 2009.

[VC07] Michael Van Canneyt. Free pascal: Reference guide. `http://www.freepascal.org/docs.var`, August 2007.

[Wir78] N. Wirth. *Programação Sistemática em PASCAL*. Editora Campus, 1978.

A Funções e procedimentos predefinidos no compilador

Nesta seção são brevemente descritas os principais procedimentos e funções predefinidos pelo compilador *Free Pascal*. Existem funções adicionais para finalidades específicas que exigem o uso de bibliotecas (`Units`), contudo tal tópico excede a finalidade deste guia rápido.

Funções específicas para se trabalhar com entrada e saída em arquivos são descritas na seção 5.8.

Um manual pormenorizado de todos os procedimentos e funções providos pelo compilador *Free Pascal* pode ser encontrado em: `http://www2.toki.or.id/fpcdoc/ref/refse64.html#x151-15700013.3`

A.1 Abs

Cabeçalho

```
function abs(x :  qualquerTipoNumerico) :  qualquerTipoNumerico;
```

Descrição

Retorna o valor absoluto de `x`. O resultado da função é do mesmo tipo que o parâmetro dado, que pode ser de qualquer tipo numérico.

Exemplo

```
program exemploAbs;
 var
    r : real;
    i : integer;
 begin
    r := abs(-1.0);   // r:=1.0
    i := abs(-21);    // i:=21
 end.
```

A.2 ArcTan

Cabeçalho

```
function arctan(x :  real) :  real;
```

Descrição

Retorna o arco tangente de `x`, que pode ser de qualquer tipo real. O resultado é dado em radianos.

Exemplo

```
program exemploArctan;
 var
```

```
    r : real;
 begin
    r := arctan(0);      // r := 0
    r := arctan(1)/pi;   // r := 0.25
end.
```

A.3 Break

Cabeçalho

`procedure break;`

Descrição

Salta para a instrução seguinte ao final do comando de repetição corrente. Tudo o que estiver entre o `break` e o fim do comando de repetição é ignorado. A expressão booleana do comando de repetição não é avaliada.

Pode ser usado com os comandos `while-do`, `repeat-until` e `for-do`. Ressalta-se que pode ser muito danoso ao código, pois possui efeito semelhante ao `goto` (ver seção 7.4);

Exemplo

```
program exemploBreak;
 var
    i : longint;
 begin
    i := 0;
    while i < 10 do
       begin
          inc(i);
          if i > 5 then
             break;
          writeln(i);
       end;
 end.
```

A.4 Chr

Cabeçalho

`function chr(x : byte) : char;`

Descrição

Retorna o caractere correspondente ao valor `x` na tabela ASCII.

Exemplo

```
program exemploChr;
 begin
```

```
    write(chr(65));  // Escreve 'A'
end.
```

A.5 Concat

Cabeçalho

`function concat(s1, s2 [, s3, ...  , sn]) :  string;`

Descrição

Concatena as `strings` `s1`, `s2`, ..., `sn` em uma única `string`. O truncado para 255 caracteres. O mesma resultado pode ser obtido com o operador `+`.

Exemplo

```
program exemploConcat;
 var
    s : string;
 begin
    s := concat('Seria mais facil',' com ',' o operador +.');
 end.
```

A.6 Continue

Cabeçalho

`procedure continue;`

Descrição

Salta para o fim do comando de repetição corrente. O código entre `continue` e o fim do comando de repetição é ignorado.

Pode ser usado com os comandos `while-do`, `repeat-until` e `for-do`. Ressalta-se que pode ser muito danoso ao código, pois possui efeito semelhante ao `goto` (ver seção 7.4);

Exemplo

```
program exemploContinue;
 var
    i : longint;
 begin
    i := 0;
    while i < 10 do
       begin
          inc(i);
          if i < 5 then
             continue;
          writeln(i);
       end;
 end.
```

A.7 Copy

Cabeçalho

```
function copy(const s :  string; indice :  integer;
quantidade :  integer) :  string;
```

Descrição

Retorna uma `string` que é uma cópia da `quantidade` de caracteres de `s`, iniciando na posição `indice`. Se isto ultrapassar o tamanho de `s`, então o resultado é truncado. Se `indice` extrapolar os limites de `s`, então uma `string` vazia é retornada.

Exemplo

```
program exemploCopy;
 var
    s, t : string;
 begin
    t := '1234567';
    s := copy (t, 1, 2);  // s := '12'
    s := copy (t, 4, 2);  // s := '45'
    s := copy (t, 4, 8);  // s := '4567'
end.
```

A.8 Cos

Cabeçalho

```
function cos(x :  real) :  real;
```

Descrição

Retorna o cosseno de `x`, onde `x` é um ângulo em radianos.

Exemplo

```
program exemploCos;
 var
    r : real;
 begin
    r := cos(Pi);     // r := -1
    r := cos(Pi/2);   // r := 0
    r := cos(0);      // r := 1
 end.
```

A.9 Dec

Cabeçalho

```
procedure dec(var x :  qualquerTipoOrdinal
[; decremento :  qualquerTipoOrdinal]);
```

Descrição

Decrementa de `x` o valor de `decremento`. Se `decremento` não for especificado, é tido como 1.

Exemplo

```
program exemploDec;
 var
    i : integer;
 begin
    i := 1;
    dec(i);      // i := 0
    dec(i, 2);   // i := -2
 end.
```

A.10 Delete

Cabeçalho

`procedure delete(var s : string; indice : integer; quantidade : integer);`

Descrição

Remove de `s` a `quantidade` de caracteres iniciando na posição `indice`. Todos os caracteres depois daqueles deletados são trazidos para imediatamente após o `indice` e o tamanho de `s` é ajustado.

Exemplo

```
program exemploDelete;
 var
    s : string;

 begin
    s := 'This is not easy !';
    delete(s, 9, 4);  // s := 'This is easy !'
 end.
```

A.11 Exp

Cabeçalho

`function exp(var x : real) : real;`

Descrição

Retorna $e^{\texttt{x}}$.

Exemplo

```
program exemploExp;
 begin
    writeln(exp(1):8:2);   // Deve imprimir 2.72
 end.
```

A.12 Frac

Cabeçalho

`function frac(x : real) : real;`

Descrição

Retorna a parte não inteira de `x`.

Exemplo

```
program exemploFrac;
 begin
    writeln(frac(123.456):0:3);    // Imprime  0.456 }
    writeln(frac(-123.456):0:3);   // Imprime -0.456 }
 end.
```

A.13 Inc

Cabeçalho

`procedure inc(var x : qualquerTipoOrdinal`
`[; incremento : qualquerTipoOrdinal]);`

Descrição

Incrementa em `x` o valor de `incremento`. Se `incremento` não for especificado, é tido como 1.

Exemplo

```
program exemploInc;
 var
    i : integer;
 begin
    i := 1;
    inc(i);      // i := 2
    inc(i, 2);   // i := 4
 end.
```

A.14 Insert

Cabeçalho

```
procedure insert(const s1 :  string; var s2 :  string;
indice :  integer);
```

Descrição

Insere a `string` `s1` em `s2` na posição `indice`, empurrando todos os caracteres depois de `indice` para a direita. O resultado é truncado para 255 caracteres.

Exemplo

```
programa exemploInsert;
 var
    s : string;
 begin
    s := 'Free Pascal is difficult to use !';
    insert('NOT ', s, pos('difficult',s));
    writeln(s);
 end.
```

A.15 Int

Cabeçalho

```
function int(x :  real) :  real;
```

Descrição

Retorna a parte inteira de um `x` real.

Exemplo

```
program exemploInt;
 begin
    writeln(int(123.456):0:1);    // Imprime 123.0
    writeln(int(-123.456):0:1);   // Imprime -123.0
 end.
```

A.16 Length

Cabeçalho

```
function length(s :  string) :  integer;
```

Descrição

Retorna o tamanho de `s`, limitado a 255. Se `s` estiver vazio, 0 é retornado.

Exemplo

```
program exemploLength;
 var
    s : string;
    i : integer;
 begin
    s := '';
    for i := 1 to 10 do
       begin
          s := s + '*';
          writeln(length(S):2, ' : ', s);
       end;
 end.
```

A.17 Log

Cabeçalho

```
function ln(x : real) : real;
```

Descrição

Retorna o logaritmo natural de um `x` positivo.

Exemplo

```
program exemploLog;
 begin
    writeln(ln(1));        // Prints 0
    writeln(ln(Eep(1)));   // Prints 1
 end.
```

A.18 Lowercase

Cabeçalho

```
function lowercase(c : charOuString) : charOuString;
```

Descrição

Retorna o argumento `c` convertido para minúsculas. O retorno é `char` ou `string`, conforme o tipo de `c`.

Exemplo

```
program exemploLowercase;
 var
    i : longint;
 begin
    for i := ord('A') to ord('Z') do
       write (lowercase(chr(i)));
```

```
    writeln;
  writeln (lowercase('ABCDEFGHIJKLMNOPQRSTUVWXYZ'));
end.
```

A.19 Odd

Cabeçalho

```
function odd(x :  longint) :  boolean;
```

Descrição

Retorna `true` se `x` for ímpar, `false` caso o contrário.

Exemplo

```
program ExemploOdd;
 begin
    if odd(1) then
       writeln('Tudo certo com o 1!');
    if not(odd(2)) then
    Writeln ('Tudo certo com o 2!');
 end.
```

A.20 Ord

Cabeçalho

```
function ord (x :  qualquerTipoOrdinal) :  longint;
```

Descrição

Retorna o valor ordinal (ASCII) do caractere `x`. A função `ord` não é exclusiva do tipo `char` e pode ser utilizada em outros tipos, porém foge do escopo deste guia.

Exemplo

```
program type_char;
 var
  a, c, d: char;
  b: integer;
 begin
  // armazena 65 na variavel a, que equivale a 'A'
  a := 'A';
  // obtem o codigo ASCII de 'A' (65) e soma a 2 (b eh um inteiro)
  b := ord('A') + 2;
  // obtem o caractere correspondente ao codigo ASCII 67 (valor de
     c)
  c := chr(b);
  d :=  #68;
  // Retorna a=A b=67 c=C
```

```
  writeln('Variaveis: a=', a, ' b=', b, ' c=', c, ' d=', d);
end.
```

A.21 Pi

Cabeçalho

```
function pi :  real;
```

Descrição

Retorna o valor de pi (3.14159265358979323 85).

Exemplo

```
program exemploPi;
 begin
    writeln (pi);  // 3.1415926}
    writeln (sin(pi));
 end.
```

A.22 Pos

Cabeçalho

```
function pos(const substring :  string; const s :  string)
:  integer;
```

Descrição

Retorna o índice da `substring` em `s`, se esta a contiver. Caso contrário, 0 é retornado. A busca é `case-sensitive`.

Exemplo

```
program exemploPos;
 var
    s : string;
 begin
    s := 'The first space in this sentence is at position : ';
    writeln(s, pos(' ', s));
    s := 'The last letter of the alphabet doesn''t appear in this
       sentence ';
    if (pos ('Z', s) = 0) and (pos('z', s) = 0) then
       writeln (s);
 end.
```

A.23 Power

Cabeçalho

```
function power(base, expon :  real) :  real;
```

Descrição

Retorna `base`$^{\texttt{expoente}}$.

Exemplo

```
program exemploPower;
 begin
    writeln(power(2, 3)); // Deve imprimir 8
 end.
```

A.24 Pred

Cabeçalho

```
function pred(x : qualquerTipoOrdinal) : mesmoTipo;
```

Descrição

Retorna o elemento que precede `x`.

Exemplo

```
program exemploPred;
 begin
    writeln(pred(3)); // Deve imprimir 2
 end.
```

A.25 Random

Cabeçalho

```
function random [(x : longint)] : longIntOuReal;
```

Descrição

Retorna um número aleatório maior do que 0 e menor do que `x`.

Exemplo

```
program exemploRandom;
 begin
    randomize;
    writeln(random(3));  // Imprimira 0, 1 ou 2
 end.
```

A.26 Randomize

Cabeçalho

```
procedure randomize;
```

Descrição

Inicializa o gerador de números aleatórios do *Pascal*, garantindo que a seqüência de números gerados mude a cada execução.

Exemplo

Vide função anterior.

A.27 Round

Cabeçalho

```
function round(x :  real) :  longint;
```

Descrição

Arredonda o real `x` para o inteiro mais próximo.

Exemplo

```
program exemploRound;
 begin
    writeln(round(1234.56));   // Imprime 1235
 end.
```

A.28 Sin

Cabeçalho

```
function sin(x :  real) :  real;
```

Descrição

Retorna o seno de `x`, onde `x` é um ângulo em radianos.

Exemplo

```
program exemploSin;
 var
    r : real;
 begin
    r := sin(Pi);     // r := 0
    r := sin(Pi/2);   // r := 1
 end.
```

A.29 Sqr

Cabeçalho

```
function sqr(x :  real) :  real;
```

Descrição

Retorna o quadrado de `x`.

Exemplo

```
program exemploSqr;
 var
    i : Integer;
 begin
    for i:=1 to 10 do
       writeln(sqr(i):3);
 end.
```

A.30 Sqrt

Cabeçalho

```
function sqrt(x : real) : real;
```

Descrição

Retorna a raiz quadrada de um `x` positivo.

Exemplo

```
program exemploSqrt;
 var
    i : Integer;
 begin
    writeln(sqrt(4):0:3);  // Imprime 2.000
    writeln(sqrt(2):0:3);  // Imprime 1.414
 end.
```

A.31 Succ

Cabeçalho

```
function succ(x : qualquerTipoOrdinal) : mesmoTipo;
```

Descrição

Retorna o elemento que sucede `x`.

Exemplo

```
program exemploSucc;
 begin
    writeln(pred(3)); // Deve imprimir 4
 end.
```

A.32 Trunc

Cabeçalho

```
function trunc(x :  real) :  longint;
```

Descrição

Trunca o real `x`. O resultado sempre será igual ou menor ao valor original.

Exemplo

```
program exemploRound;
 begin
    writeln(trunc(1234.56));  // Imprime 1234
 end.
```

A.33 Upcase

Cabeçalho

```
function upcase(c :  charOuString) :  charOuString;
```

Descrição

Retorna o argumento `c` convertido para maiúsculas. O retorno é `char` ou `string`, conforme o tipo de `c`.

Exemplo

```
program exemploUpcase;
 begin
    writeln (upcase('abcdefghijklmnopqrstuvwxyz'));
 end.
```

A.34 Val

Cabeçalho

```
procedure val (const s :  string; var v; var codigo :  word);
```

Descrição

Converte o valor representado em `s` para um valor numérico e armazena este conteúdo na variável `v`, que pode ser dos tipos `longint`, `real` ou `byte`. Caso a conversão não seja bem-sucedida, o parâmetro `codigo` conterá o índice do caractere de `s` que impossibilitou a conversão. Admite-se espaços no início de `s`.

Exemplo

```
programa exemploVal
 var
    i, codigo : integer;
 begin
    val ('2012', i, codigo);
    if (codigo <> 0) then
       writeln ('Erro na posicao ', codigo,'.')
    else
       writeln ('Valor: ', i);
 end.
```