6

Expressões e Comandos

"Decide. Will you share the labor, share the work?"

—Sophocles

Discutimos até o momento a importância dos tipos e variáveis como elementos responsáveis pela modelagem dos dados dos problemas reais. A partir dessa modelagem, ou representação, dos dados reais, precisamos de formas de processamento que transformem os dados originais em novos dados produzidos como resultado dos programas. Essas transformações são efetivadas por expressões e comandos. Neste capítulo mostramos expressões e comandos comuns em linguagens de programação responsáveis pela transformação dos dados. Inicialmente discutimos a idéia de estado de um programa e a computação de um programa como sucessivas transformações de estados. Em seguida, as expressões e comandos responsáveis por tais transformações são discutidos.

6.1 O PROGRAMA COMO MÁQUINA ABSTRATA

Como já visto no Capítulo 2, os programas são máquinas abstratas responsáveis pela transformação de dados. Vimos também que a solução computacional para um problema requer que este tenha os seus dados modelados de forma computacional. A partir de um modelo desses dados, o programa faz processamentos que os transformam em dados resultantes, ou seja, o resultado computacional do programa.

Considerando a computação de um programa como uma máquina de transformação, podemos distinguir dois **estados** primordiais do programa: o **estado inicial** – quando nenhuma transformação sobre os dados ainda foi realizada – e o **estado final**, após todas as transformações realizadas pelo programa. A maioria dos programas, no entanto, é formada por sucessivas transformações de dados, por suas expressões e comandos. Dessa forma, podemos subdividir o programa em elementos de transformação sucessiva.

Ainda nos resta definir o que representa o estado de um programa. Se considerarmos que um programa é representado por uma função, como nas linguagens funcionais, os dados de entrada representam o estado inicial, o processamento da função representa o programa, e o resultado da função representa o estado final. Para as linguagens que possuem variáveis para "guardar"valores de dados a serem computados pelo programa, o estado é refletido pelo conjunto de variáveis (todos os seus atributos: identificador, conteúdo, etc.) e o seu fluxo de controle. O estado inicial de um programa é, portanto, caracterizado pelo conjunto de suas variáveis, com os seus respectivos conteúdos, quando a computação do programa é iniciada. Após uma primeira transformação, temos um novo estado do programa, e assim sucessivamente até o final da computação, chegando ao estado final do programa. Para programas seqüenciais, a computação de um programa é, na realidade, uma transformação sucessiva de estados.

As expressões são o recurso disponível nas linguagens de programação para a transformação de dados, alguns dados são fornecidos e um resultado obtido a partir da computação desses dados. Os comandos, por sua vez, atuam sobre o fluxo de controle e estados do programa. Nas seções que seguem, mostramos as expressões e comandos

mais comuns nas linguagens de programação, os quais atuam na transformação de dados e estados dos programas.

6.2 EXPRESSÕES

As expressões são os elementos de transformação de dados em um programa: a partir de valores, uma transformação é aplicada e um valor resultado é produzido. Note que as expressões atuam primordialmente sobre valores: elas recebem e produzem valores. A seguir, mostramos como as expressões comuns nas linguagens de programação representam valores e como elas são avaliadas.

6.2.1 Expressões como Valores

Na maioria das linguagens de programação, podemos tratar valores por meio expressões. Podemos, por exemplo, calcular somas ou subtrações de dois números inteiros pelos operadores aritméticos predefinidos nas linguagens. A seguir, enumeramos as várias formas de expressões que aparecem em linguagens de programação e como elas representam valores.

6.2.1.1 Literais Os literais nas linguagens de programação são expressões que denotam um valor fixo de algum tipo. As linguagens de programação possuem formas de representação para valores de tipos predefinidos. Exemplos triviais de literais são:

5 denota o valor cinco do tipo inteiro;

5.0 denota o valor cinco do tipo real;

‘a‘ denota o caracter a na linguagem C, por exemplo, etc.

Para linguagens que permitem a construção de tipos enumerados, também são considerados como literais os novos valores inseridos para o tipo.

6.2.1.2 Agregação de Valores Na maioria das linguagens, existem expressões que constroem valores compostos a partir de outros valores mais simples.

Exemplo 6.1 – Em C, por exemplo, podemos agregar valores para formar um *array* com um tipo e tamanho definidos:

```
int n[5] = {32,45,66,23,58};
```

O conjunto de valores é usado como expressão, e o *array* `n` possui 5 elementos com os respectivos valores do conjunto. □

A linguagem Perl permite agregação de valores de uma forma ainda mais flexível, no qual não se tem necessariamente predefinidos o tipo e a quantidade de elementos dos *arrays*.

Exemplo 6.2 – Um *array* com dois elementos que associam uma *string* a um valor inteiro pode ser definido em Perl como segue:

```
%salarios = (``Maria'' => 750, ``Pedro'' => 100);
```

Neste caso, o tipo e tamanho do *array* são inferidos a partir dos valores, estes elementos aparecem sem uma declaração prévia e os atributos de tamanho e tipo podem ser modificados ao longo da computação do programa. □

6.2.1.3 Aplicação de Funções

A chamada de uma função em um programa computa um resultado pela aplicação de uma abstração a um conjunto de argumentos. De uma forma geral, podemos escrever a aplicação de uma função como $F(PA)$, onde F é um identificador de função e PA é o parâmetro atual. Na maioria das linguagens, PA é permitido ser apenas valores específicos. Em linguagens funcionais, estes parâmetros podem ser também outras funções.

É importante salientar que os operadores predefinidos sobre tipos primitivos podem ser vistos como aplicações de funções:

$$3 + 4 \equiv soma(3, 4)$$

Assim, as expressões aritméticas e lógicas predefinidas na maioria das linguagens de programação são aplicações de funções predefinidas sobre tipos predefinidos nas linguagens.

6.2.1.4 Expressões Condicionais

As expressões condicionais são aquelas que assumem valores diferentes dependendo de uma condição. Em linguagens funci-

onais, que são orientadas a expressões como SML, temos condicionais que produzem valores.

Exemplo 6.3 – Uma função definida em SML que recebe dois valores inteiros e devolve o maior deles pode ser definida como segue:

```
fun maior (x:int, y:int) = if x > y then x else y;
```

Neste caso, podemos ter como resultado tanto o valor de `x` quanto o valor de `y`, depende do resultado da condição da expressão. Note que o resultado é o próprio valor, sem haver qualquer atribuição a variáveis. □

Em Pascal, não há expressão condicional, mas apenas comandos condicionais, já que os mesmos não produzem um valor, mas um efeito sobre o estado do programa pelas atribuições. Nas linguagens C, C++ e Java, as expressões condicionais aparecem da seguinte forma:

```
<expressão1> ? <expressão2> : <expressão3>
```

onde a `<expressão1>` é uma condicional e, mediante sua avaliação, ter-se-á como resultado a `<expressão2>` (se a condição for verdadeira) ou a `<expressão3>` (se a condição for falsa).

Exemplo 6.4 – Um exemplo de uso de expressões condicionais em C:

```
int n, x, y;
  n = (x > y) ? x : y
```

Aqui, à variável `n` é atribuído o valor resultado da expressão condicional: o valor de `x` se a expressão lógica `(x > y)` for verdadeira e o de `y` se a expressão for falsa. □

Note o efeito sobre o estado do programa na expressão acima, enquanto no exemplo dado em SML não há qualquer efeito sobre "variáveis" do programa.

6.2.1.5 Valores associados a identificadores

Na maioria das linguagens de programação, podemos definir valores constantes associados a nomes (as constantes), bem como variáveis. Os valores associados às constantes são calculados

diretamente, já que na maioria dos casos as constantes são substituídas pelos próprios valores (quando a linguagem realiza vinculação estática de valores). Por outro lado, as variáveis têm valores atuais associados. Desta forma, uma variável deve ter seu valor recuperado na hora em que é usada; depende do estado atual do programa. O acesso ao valor de cada constante ou variável é na verdade o cálculo de uma expressão; um valor a partir de um identificador.

6.2.2 Avaliação de Expressões

Da avaliação de uma expressão resulta o cálculo da expressão dados os valores. Para literais, por exemplo, o cálculo resultante da expressão é o valor correspondente do literal. Para variáveis, o cálculo da expressão corresponde ao conteúdo da variável. Para as expressões aritméticas e lógicas, contudo, o cálculo só pode ser realizado mediante as regras de precedência dos próprios operadores ou impostas pelos parênteses, a ordem de avaliação dos operandos e ainda as regras da linguagem quando a sobrecarga de operadores é permitida. A sobrecarga de operadores foi discutida na Seção 5.3.2, na seções que seguem tratamos a ordem de avaliação e os efeitos colaterais nas expressões.

6.2.2.1 Ordem de Avaliação A ordem de avaliação depende primordialmente da precedência dos operadores, bem como dos parênteses, e a ordem de avaliação dos operandos. Nas expressões aritméticas, por exemplo, temos em geral os operadores binários de multiplicação e divisão com a mesma prioridade, mas com prioridade maior que os operadores de soma e subtração. Os operadores unários têm prioridade mais alta que os binários aqui citados. Para operadores associativos definidos com a mesma prioridade, o resultado da expressão independe da ordem em que eles são avaliados.

Exemplo 6.5 – Considere a seguinte expressão:

```
A + B - C
```

A expressão acima pode ser calculada pela aplicação tanto da soma e depois da subtração, quanto da subtração e depois a soma, por causa da associatividade dos operadores de soma e subtração.

```
(A + B) - C = A + (B - C)
```

□

Para operadores com prioridades diferentes, primeiro o operador de mais alta prioridade será avaliado, e depois o de menor prioridade.

Exemplo 6.6 – Considere a seguinte expressão:

```
A + B * C
```

A expressão acima realizará primeiro o cálculo da multiplicação para então proceder com a soma.

```
A + (B * C)
```

□

Quando se deseja dar prioridade a operadores explicitamente, usamos os parênteses para denotar a ordem de avaliação de uma expressão: as subexpressões nos parênteses mais internos são calculadas, depois os imediatamente mais externos e assim sucessivamente até completar o cálculo de toda a expressão.

Exemplo 6.7 – Considere a expressão:

```
(A + B) * C
```

Esta expressão realizará primeiro o cálculo da soma (os parênteses mais internos) para, então, proceder com a multiplicação. □

As expressões lógicas seguem o mesmo raciocínio. Na maioria das linguagens o operador lógico de maior prioridade é o unário `NOT`, seguido do binário `AND`, seguido do `OR` (e o `OR` exclusivo em alguns casos). Outras linguagens, tal como Ada, tratam todos os operadores lógicos com a mesma prioridade, e assim as expressões devem ser parentetizadas porque estes operadores não são associativos.

Além da precedência natural dos operadores e a ordem de avaliação imposta pelos parênteses, devemos notar que a maioria das linguagens avalia as expressões da esquerda para a direta da ordem de escrita (algumas vezes determinado pelo projeto da linguagem e em outras vezes pela implementação do processador). Vários dos

processadores atuais utilizam o recurso de **avaliações truncadas**, principalmente em expressões lógicas.

Exemplo 6.8 – Considere a expressão lógica seguinte:

```
(x > 0) AND ((y/x) > 2)
```

Se a expressão for avaliada por completo, haverá um erro de execução quando `x` tiver o valor 0 (divisão por zero). Contudo, se a expressão `(x > 0)` for falsa, já podemos concluir que a expressão completa também será falsa. Vários processadores de linguagens truncam a avaliação da expressão no estágio em que o resultado final pode ser inferido. Nessa avaliação truncada, a expressão acima não ocasionaria erro. Para isso, o programador deve ter conhecimento se a linguagem usa este recurso ou não. □

Linguagens mais modernas, como por exemplo C, C++, Java e Modula-2 oferecem avaliações de expressões lógicas truncadas. Um recurso análogo poderia ser usado para algumas expressões aritméticas (multiplicação por zero, por exemplo), mas as linguagens atuais não oferecem tal recurso.

6.2.2.2 Efeitos Colaterais Expressões denotadas por literais, constantes ou variáveis têm apenas acesso a um valor diretamente. A avaliação dessas expressões não provocam quaisquer modificações sobre o estado do programa. Da mesma forma, expressões aritméticas que envolvem apenas os operadores predefinidos na linguagem juntamente com variáveis e literais não provocam quaisquer mudanças no estado do programa. No entanto, quando a expressão contém uma aplicação de função, algumas operações internas à função podem acarretar mudanças no estado do programa.

Exemplo 6.9 – Considere a definição esquemática de um programa na linguagem C:

```
int x = 10, y = 20;
int fun1 ()
 { x = x + 30;
   return 10;
 } /* fun1 */
```

```
void main ()
 { y = y + fun1();                              (1)
   ...
   if ((y > 50) && ((fun1() + y) > 70))         (2)
   ...
 } /* main */
```

No programa acima, a função `fun1` modifica o valor da variável `x`. Então, a aplicação desta função acarreta um efeito colateral sobre o estado do programa. A avaliação da expressão em (1) acima ocasionará uma mudança de estado do programa. □

Uma outra forma de efeito colateral das expressões pode ser notada quando uma variável usada em uma expressão tem seu valor modificado por uma função na mesma expressão. Suponha a seguinte expressão (referente ao exemplo acima), quando `x` tem o valor 10:

```
x + fun1();
```

se a expressão for avaliada da esquerda para a direita, o resultado da expressão acima será 20 e `x` passa a ter o valor 40 ao final da avaliação. Contudo, se a função for avaliada primeiro, o valor de `x` será modificado e a expressão terá como resultado o valor 50.

Existem algumas alternativas de solução para se ter uma ordem precisa de avaliação de expressões e conseqüentemente um controle dos efeitos colaterais. Primeiro, o projetista pode impedir que a avaliação de uma função afete o valor da expressão, evitando assim o efeito colateral no resultado da expressão. Uma segunda alternativa poderia ser a definição, no projeto da linguagem, da ordem de avaliação das expressões, e os processadores devem obedecer a tal ordem. Na linguagem Java, por exemplo, as expressões são avaliadas da esquerda para a direita (definido no projeto da linguagem). Uma terceira alternativa ainda seria a proibição, nas expressões, do uso de funções que modifiquem valores de variáveis existentes na própria expressão. Ignorar os efeitos colaterais no projeto de uma linguagem compromete o seu uso, uma vez que cada processador adotará uma forma de avaliação particular.

Outro aspecto a ser observado é a conseqüência de efeitos colaterais quando a avaliação truncada é permitida. Ainda no exemplo acima, temos em (2) o uso de uma expressão lógica com o operador AND (`&&`). Se a expressão for avaliada da esquerda para a direita e a primeira parte da expressão `(y > 50)` for falsa, o efeito colateral da aplicação de `fun1` na segunda parte da expressão não ocorrerá. Mas se a expressão for avaliada por completo, o efeito colateral ocorrerá mesmo que a expressão lógica seja falsa. Note que todos esses aspectos relacionados à avaliação das expressões devem ser conhecidos pelo programador para o uso adequado da linguagem.

6.3 COMANDOS

Assim como as expressões são responsáveis pela transformação de dados, os comandos são responsáveis pelas mudanças de estados nas linguagens dos programas. Podemos dividir os comandos mais simples em dois grandes grupos: os comandos de atribuição, os quais fazem mudanças de conteúdo de variáveis (estado) do programa diretamente; e as estruturas de controle no nível de instrução que direcionam os passos subseqüentes da computação do programa.

6.3.1 Atribuição

Os comandos de atribuição são responsáveis pela mudança explícita do estado do programa, onde uma ou mais variáveis têm seus valores modificados. As linguagens de programação que possuem variáveis, possuem também comandos de atribuição. Formas triviais desses comandos são:

Atribuição simples: nesta forma de atribuição, uma expressão é avaliada e o valor resultado atribuído a uma única variável. Exemplos deste tipo de atribuição estão presentes nas mais diversas linguagens de programação, como na linguagem C:

```
x = y;
x = y + 15;
```

O efeito da atribuição em ambos os casos acima é a modificação do conteúdo da variável `x` para o valor resultado das respectivas expressões. Algumas abreviações do comando de atribuição são permitidas em linguagens como C, C++ e Java:

```
x ++;       equivalente a   x = x + 1;          (1)
x + = y     equivalente a   x = x + y;          (2)
x = y ++;   equivalente a   y = y + 1; x = y;   (3)
x = ++ y;   equivalente a   x = y; y = y + 1;   (4)
```

No exemplo acima, as variáveis `x` (em (1) e (2)) e `y` (em (3) e (4)) são usadas tanto como operando da expressão quanto como variável a ter seu conteúdo modificado. Abreviações análogas existem para os operadores de subtração, multiplicação e divisão nessas linguagens. Essas abreviações são em geral usadas por programadores experientes, mas em muitos casos prejudicam a legibilidade dos programas.

Atribuição múltipla: algumas linguagens permitem que o valor resultado da expressão seja atribuído a diversas variáveis. Exemplos de atribuição múltipla podem ser encontrados nas linguagens PL/I, C, C++ e Java. Em PL/I a atribuição é realizada sobre as variáveis simultaneamente.

```
x, y = 10        em PL/I
```

O efeito desse tipo de atribuição é o cálculo da expressão mais à direita e a modificação do valor das variáveis à esquerda. Em C, C++ e Java um efeito semelhante é conseguido quando usamos atribuições inseridas nas expressões (o item seguinte trata essas atribuições) e as variáveis são do mesmo tipo.

```
float x, y;
  x = y = 10.2       em C, C++ e Java
```

Aqui o valor 10.2 é atribuído à variável `y` e depois o valor de `y` é atribuído à variável `x`. Note que a ordem de atribuição difere do exemplo anterior. Além

disso, como são realizadas atribuições sucessivas, coerções de tipos podem ser aplicadas sucessivamente.

Atribuição inserida em expressões: vimos na Seção 6.2.2.2 que podemos ter efeitos colaterais nas expressões quando nelas existem funções que modificam variáveis globais ou parâmetros formais. Além dessa forma de efeitos colaterais, expressões nas linguagens C, C++ e Java permitem que quaisquer expressões contenham atribuições. Podemos ter exemplos do tipo:

```
x = a + (y=z / b++) equivalente a   b = b + 1;
                                    y = z;
                                    x = a + (y/b);
```

Este comando de atribuição não só tem efeito sobre a variável `x`, mas também sobre as variáveis `y` e `b`, as quais realizam atribuições dentro da expressão.

Essas atribuições podem inclusive ser realizadas em expressões lógicas:

```
if (((x = y) > 10)...   equivalente a   x = y;
                                        if (x > 10)...
```

Mais uma vez, tal flexibilidade, ou abreviação, de comandos de atribuição acarretam um comprometimento da legibilidade dos programas, principalmente porque a igualdade relacional (comparação entre valores) é escrita nestas linguagens como "==". É comum erros em programas que realizam uma atribuição (=) dentro de uma expressão lógica quando na realidade deveriam fazer uma comparação (==), e muitas vezes passa despercebido aos programadores.

6.3.2 Instruções Compostas e Blocos

Vimos na Seção 5.2 que podemos ter uma coleção de comandos (delimitados pelo seu início e fim) abstraídos de forma a serem tratados como um único comando. Esta coleção não é um conjunto de instruções que podem ser executadas em uma ordem arbitrária, mas em uma seqüência determinada de instruções, e a estas seqüências

chamamos de **instruções compostas**. A ordem dos comandos é determinada pelo comando seqüencial que em algumas linguagens, como C, C++ e Java, aparecem com separador ";", e em outras apenas a seqüência de escrita.

Para que essas seqüências de instruções sejam tratadas como únicas elas precisam ter seu começo e fim delimitados. A idéia de instruções compostas surgiu com Algol 60, a qual usa os delimitadores `begin` e `end` para início e final respectivamente. A linguagem Pascal usa esses mesmos delimitadores.

Em outras linguagens, como C, C++ e Java, além das instruções compostas temos também os **blocos de comando**, os quais podem conter declarações de variáveis locais seguido de uma seqüência de instruções. Essas linguagens usam chaves para delimitar tanto as instruções compostas quanto os blocos de comandos. Em algumas outras linguagens como Pascal, apenas os próprios programas ou as subrotinas são aceitos como blocos de comando (só neles é permitido fazer declarações de variáveis).

6.3.3 Condicionais

As instruções de seleção permitem escolher entre duas ou mais seqüências de instruções a serem executadas nos programas e são fundamentais às linguagens de programação. Existem dois grande grupos de seleção: a bidirecional e a n-direcional, estudadas a seguir.

6.3.3.1 Seleção bidirecional A seleção bidirecional está presente em todas as linguagens de programação atuais baseadas em comandos. Na linguagem Pascal, essa instrução aparece como:

```
if (<expressão_lógica>)
  then <instrução_composta1>
  else <instrução_composta2>
end;
```

Um exemplo de uso dessa instrução de seleção é como segue:

Exemplo 6.10 – Uma instrução que atribui à variável `maior` o maior valor entre as variáveis `x` e `y` pode ser definida por:

```
if (x > y) then
   begin
      maior := x      (1)
   end
else
   begin
      maior := y      (2)
   end
end;
...
```

A expressão `(x > y)` é avaliada, e quando o valor resultante é o verdadeiro, (1) será executado, caso contrário, (2) será executado. □

No comando de seleção acima apenas uma instrução é dada para cada um dos caminhos alternativos, e, por isso, não há necessidade do uso dos delimitadores, eles poderiam ser retirados dessas instruções sem qualquer prejuízo. Um uso restrito da seleção bidirecional é a seleção unidirecional, em que instruções são executadas apenas quando a expressão lógica é verdadeira.

Exemplo 6.11 –

```
if (x > y) then
   begin
      maior := x      (1)
   end
end;
...
```

A instrução em (1) é executada apenas quando a expressão `(x > y)` resulta no valor verdadeiro, e nada é executado neste comando quando esta expressão resulta o valor falso. □

Construções similares são encontradas nas linguagens C , C++ e Java, exceto que blocos de comandos podem ser executado como alternativas em vez de apenas as

instruções compostas. Instruções com o mesmo significado dos exemplos anteriores podem ser definidos na linguagem C como segue:

Exemplo 6.12 – O exemplo anterior pode ser escrito em C da seguinte forma:

```
 if (x > y)
    { maior := x;}   (1)
 else
    { maior := x;}   (2)
...

 if (x > y)
    { maior := x;}   (3)
...
```

Como (1), (2) e (3) podem ser blocos de comandos, poderíamos ainda ter declarações de variáveis locais ao bloco. □

Vale ressaltar que podemos ter comandos de seleção aninhados (um comando como parte de outro) uma vez que cada instrução composta (ou bloco) pode ter um outro comando de seleção.

6.3.3.2 Seleção n-direcional A seleção n-direcional é uma generalização da bidirecional, em que várias (n) seqüências de instruções podem ser seguidas em vez de apenas duas. Uma forma mais antiga do uso de múltiplos seletores pode ser encontrada na Algol-W, na qual n instruções podem ser escolhidas alternativamente, mediante o resultado (no intervalo de inteiros de 1 a n) de uma expressão aritmética. Formas menos restritivas podem ser encontradas em linguagens mais modernas.

Na linguagem Pascal, por exemplo, a seleção n-direcional é denotada pelo comando `case`:

```
case <expressão> of
  <lista_de_literais1> : <instrução_composta1>;
  ...
  <lista_de_literaisn> : <instrução_compostan>;
  [else <instrução_composta_n+1>]
```

```
end
```

Nesse comando a `expressão` é de um dos tipos primitivos: inteiro, booleano, caracter ou enumeração. A `expressão` é avaliada e o valor resultado da expressão é comparado com os literais das `listas_de_literais`, e a `instrução_composta` correspondente (do lado direito do literal) é executada quando o valor da expressão é igual a um dos literais na lista. Obviamente os literais devem ser do mesmo tipo, o qual deve ser o mesmo do valor da expressão, e devem ser mutuamente exclusivos (um mesmo literal não pode ser repetido em listas diferentes), mas a lista não precisa ser exaustiva. Quando nenhum dos literais é igual ao valor da expressão, a opção `else` é usada[1]. Ao final da execução da instrução correspondente, o comando é finalizado. Ou seja, uma vez encontrado o valor da expressão em uma das listas de literais, as instruções correspondentes são executadas e não há mais comparações com os elementos das outras listas. Um outro fator importante é que a cláusula `else` é opcional. Para comandos que não incluem esta cláusula, o programa executa a próxima instrução sem ter executado qualquer instrução do `case`.

Exemplo 6.13 – Um exemplo de uso do comando `case` em Pascal pode ser visto como a contagem de conjuntos de vogais:

```
case  letra  of
  'a', 'e' : begin
              contae := contae + 1                (1)
             end;
  'i', 'o' : begin
              contio := contio + 1                (2)
             end;
  'u'      : begin
              contu  := contu  + 1                (3)
             end;
  else       begin
```

[1] Algumas versões iniciais da linguagem não incluíam a cláusula `else`.

```
                contconsoantes := contconsoantes +1   (4)
 end
```

Neste exemplo, temos contadores para os respectivos conjuntos de vogais, os quais serão incrementados quando o valor da expressão `letra` é igual a uma das constantes da lista correspondente. Por exemplo, quando `letra` tem o valor `'a'` o (1) é executado. Quando o valor da expressão `letra` não é uma vogal, o contador de consoantes será incrementado (opção `else`). □

A linguagem Ada, assim como Pascal, permite que as constantes sejam de um tipo primitivo discreto, mas permite ainda o uso de intervalos, [1 .. 10] por exemplo. As linguagens C, C++ e Java, contudo, possuem um comando de seleção n-direcional (`switch`) mais restritivo que o da linguagem Pascal, onde a expressão só pode ser do tipo inteiro. A definição do comando é como segue:

```
switch (expressão) {
  case <expressão_constante1> : <bloco1>;
  ...
  case <expressão_constanten> : <blocon>;
  [default                    : <blocon+1>]
  }
```

Nesse comando, a `expressão` deve ser do tipo inteiro, assim como as `expressoes_constante` em cada uma das alternativas `case` do comando. A `expressão` é avaliada, o seu valor é comparado com a primeira `expressão_constante` e se forem iguais a instrução correspondente é executada, e assim sucessivamente até que se tenha comparado/executado todas as alternativas. Aqui, as constantes não precisam ser mutuamente exclusivas, mais de um conjunto de instruções pode ser executado, e todas as alternativas serão comparadas, uma vez que não há um desvio para o final do comando como no `case` da linguagem Pascal. A cláusula `default` tem a mesma semântica do `else` do Pascal.

Em grande parte das vezes que usamos as multescolhas, colocamos alternativas exclusivas, e certamente não gostaríamos que o comando prosseguisse com as comparações uma vez que a alternativa correta já foi encontrada. Então, é comum o

uso do comando `switch` com um desvio para o final do comando (`break`) após a execução das instruções correspondentes:

```
switch (expressão) {
  case <expressão_constante1> : <bloco1>; break;
  ...
  case <expressão_constanten> : <blocon>; break;
  [default                    : <blocon+1>]
  }
```

Quando o comando de desvio é usado, as alternativas seguintes não serão comparadas, o que se torna imprescindível, por exemplo, quando usamos a cláusula `default`. Vale ressaltar que alguns livros sobre a linguagem C já incluem o desvio como parte do próprio comando `switch`.

Exemplo 6.14 – A contagem de uma variável como par ou ímpar, no intervalo de números inteiros de 1 a 2, pode ser implementada como segue:

```
switch  (numero) {
   case 1: contimpar = contimpar + 1;
           break;
   case 2: contpar = contpar + 1;
           break;
   default:  printf("fora do intervalo");
   }
```

Quando nenhuma das alternativas constante é encontrada igual ao valor da expressão, o programa imprime a mensagem acima através da cláusula `default`. □

Todas essas seleções n-direcionais podem ser reescritas com o comando bidirecional, mas as n-direcionais melhoram a legibilidade do programa. Na maioria das linguagens, não é permitido o uso de expressões relacionais ou lógicas como alternativas (os exemplos acima mostram o uso de constantes). Para essas linguagens, os comandos bidirecionais aninhados devem ser usados.

Todos os comandos de seleção descritos acima sempre executam as instruções correspondentes quando encontrada a constante de mesmo valor. No caso do `switch`,

vários conjuntos de instruções podem ser executados, desde que as constantes sejam iguais aos valores da expressão. Essa é uma forma de processamento **determinística**, onde todas as instruções correspondentes serão executadas. Existem ainda os comandos n-direcionais **não determinísticos**, definidos por Dijkstra [26] e implementados na linguagem Ada. Neste, tem-se um `if` com várias expressões lógicas alternativas (em paralelo), e cada uma seguida dos conjuntos de instruções correspondentes. Se mais de uma expressão for verdadeira, uma escolha não determinística é feita para a execução de apenas um dos conjuntos de instruções correspondente. Apesar de valioso em programação, este comando é mais difícil de ser implementado e raramente encontrado nas linguagens.

6.3.4 Iterativos

Os comandos iterativos fazem com que uma instrução, ou um conjunto de instruções, seja executada zero, uma ou várias vezes. Esses comandos possuem basicamente o **corpo do comando** (o conjunto de instruções as serem executadas) que deve ser executado repetidas vezes, e mais uma expressão de controle que determina quando o corpo do comando deve ser executado, o **controlador do laço**. Podemos classificar os comandos iterativos em relação ao número de iterações em dois grandes grupos: número predefinido de iterações, e número indeterminado de iterações. Nas seções que seguem apresentamos os comandos existentes segundo esta classificação.

6.3.4.1 Número predefinido de Iterações

Nesses comandos o número de vezes que o conjunto de instruções deve ser executada é determinado *a priori*. Eles são caracterizados por uma **variável de controle**, inserida na expressão controladora do laço, e esta determina o número de vezes que as instruções devem ser executadas mediante o seu valor. Um exemplo é o comando `for` da linguagem Pascal:

```
for <variável> := <expressão1> (to | downto)
                  <expressão2> do
                     <corpo_comando>
```

A `expressão1` é avaliada e atribuída à `variável`. Quando o `to` é usado, a variável é comparada se menor ou igual à `expressão2` (ou maior ou igual quando

o `downto` é usado), e em caso positivo o `corpo_comando` é executado. O comando prossegue com o incremento/decremento (to/downto) 1 da `variável` e execução do corpo do comando até que o valor da `variável` seja maior/menor que o valor da `expressão2`. Em outras palavras, a `variável` recebe um valor no intervalo de `expressão1` até `expressão2` e o conjunto de instruções é executado para cada valor da variável. Essa variável de controle não pode ser modificada no conjunto de instruções e ao final do comando o seu valor fica indefinido. A variável de controle pode ser de quaisquer dos tipos primitivos discretos, em que o incremento 1 tem o significado de o próximo valor.

Exemplo 6.15 – O seguinte comando `for` na linguagem Pascal imprime as letras minúsculas em sua ordem:

```
for  c := 'a'  to 'z' do
  write(c);
```

□

Algumas outras linguagens permitem a definição do incremento. Dos projetos mais antigos, as linguagens como Algol e PL/I são exemplos:

```
for <variável> := <expressão1> to <expressão2>
                                  by <expressão3> do
    <corpo_comando>
```

Várias das linguagens mais modernas permitem a determinação do incremento. As linguagens C, C++ e Java, além da definição do incremento, permitem também que mais de uma variável de controle seja usada:

```
for (<expressão1>; <expressão2>; ,expressão3>){
    <corpo_comando>
    }
```

onde a `expressão1` é um conjunto de atribuições iniciais às variáveis de controle, a `expressão2` representa o conjunto de expressões lógicas sobre as variáveis de controle, e a `expressão3` representa a progressão das variáveis. As atribuições existentes na `expressão1` são inicialmente executadas, a expressão lógica da

`expressão2` é avaliada e, caso seja verdadeira, o corpo do comando é executado. Após o seu término, os incrementos das variáveis são realizados de acordo com a `expressão3`.

Exemplo 6.16 – Um comando `for` em C, que incrementa contadores com os valores de duas variáveis, uma do tipo inteiro e outra do tipo real com incrementos definidos pelo programador, pode ser implementado como segue:

```
somai = ...;
somar = ...;
for(i= 10, r= 2.0 ; (i < 100 || r < 20.0); i=i+2, r=r*2.5)
 {somai = somai + i;
  somar = somar + r;
 }
```

□

O fato de ter uma expressão lógica que controla as iterações definidas pelo usuário (e não predefinida pela linguagem como nos casos acima) faz com que este comando seja mais abrangente que os outros apresentados. Essas linguagens, no entanto, não coibem a modificação das variáveis de controle no corpo do comando, apesar de ser uma prática de programação não recomendada. Da mesma forma, o comando `for` pode ser usado como um comando com um número indefinido de iterações (apresentado a seguir) quando as expressões 1 e 3 são omitidas. Todas estas observações não são consideradas como boa prática de programação apesar de permitidas pelas linguagens.

6.3.4.2 Número indefinido de iterações

Os comandos de iteração com um número indefinido de repetições são controlados por uma expressão lógica em vez de uma (ou um conjunto de) variável com uma expressão de progressão predefinida na expressão de controle.

A maioria das linguagens modernas inclui comandos iterativos, que fazem o teste da expressão lógica e só executam o corpo de laço após este teste, e outros que

executam pelo menos o corpo de comando uma vez e depois fazem o teste. Nas linguagens C, C++ e Java, estes comandos aparecem com os seguintes formatos:

```
   --while--                        --do-while--
while (<expressão_lógica>)    do{
 {corpo_comando               corpo_comando
 }                              }while(<expressão_lógica>)
```

No comando `while`, a `expressão_lógica` é avaliada e se verdadeira, o `corpo_comando` é executado. Esses passos são repetidos até que a `expressão_lógica` passe a ter o valor falso. Neste caso, o próximo comando do programa é executado. As variáveis que aparecem na expressão lógica devem ter seus valores predefinidos por comandos anteriores do programa e podem ser modificadas no corpo do comando. O comando `do-while` executa inicialmente o `corpo_do_comando` e faz o teste da expressão lógica, prosseguindo com as repetições apenas quando esta é verdadeira.

Exemplo 6.17 – Um comando `while` em C que incrementa contadores mediante os valores de duas variáveis, como no exemplo anterior, pode ser implementado como segue:

```
somai = ...;
somar = ...;
i = 10 ;
r = 2.0
while(i < 100 || r < 20.0)
 {somai = somai + i;
  somar = somar + r;
  i = i + 2 ; r = r * 2.5;
 }
```

As variáveis usadas na expressão lógica são modificadas no corpo do comando. □

Comandos semelhantes existem nas outras linguagens modernas. Pascal possui versões com semântica semelhante para o `while` e para o `do-while`, o `repeat-until`, só que neste último a repetição é realizada enquanto a expressão tiver o valor falso.

A linguagem Ada possui apenas o `while`, mas isso não a torna menos expressiva porque os comandos iterativos com o pós-teste (`do-while`) podem ser facilmente implementados com o `while`.

6.3.5 Desvio Incondicional

Os desvios incondicionais transferem o controle de execução de um programa para um determinado local. Estes comandos foram implementados nas primeiras linguagens de programação como Fortran (o `goto`) e necessário por causa da limitação das estruturas de controle.

Para a execução do comando `goto`, por exemplo, é necessário o uso de rótulos que determinem locais do programa para os quais o desvio pode enviar o controle. Contudo existem sérias restrições ao uso dos desvios porque eles desconsideram as outras estruturas de controle do programa, e isso pode ocasionar erros no programa. Outro problema relacionado ao `goto` é a legibilidade dos programas. Ele pode ser usado para desvios tanto para locais posteriores quando anteriores ao local atual da computação, e o uso excessivo do mesmo dificulta o entendimento dos programas.

Algumas linguagens modernas foram projetadas sem o `goto`, como a Modula-2 e Java. A linguagem C, apesar de moderna, conservou este desvio incondicional. Formas de desvios para os finais de comandos, como o `break`, também são incluídos nestas linguagens. Este último contudo é, em geral, usado de forma disciplinada por existir o local específico do desvio.

6.4 LEITURA RECOMENDADA

O resultado teórico de que os comandos de controle de seleção, seqüência e o de iteração com um pré-teste (`while`) são o suficiente para expressar os comportamentos seqüenciais está descrito em [18].

Os comandos condicionais não determinísticos foram definidos por Dijkstra em [26], juntamente com uma disciplina de programação. Uma apresentação destes comandos de forma resumida também pode ser encontrada em [64].

Para os leitores interessados em aspectos mais formais sobre linguagens de programação consultar [42].

Os comandos e expressões específicos a cada linguagem podem ser obtidos através dos manuais de programação ou de definição.

6.5 EXERCÍCIOS

1. Alguns programadores são favoráveis aos efeitos colaterais em expressões. Dê um exemplo no qual o bom uso de efeitos colaterais em expressões pode simplificar a programação. Descreva os aspectos positivos e negativos de efeitos colaterais em expressões.

2. Verificar se as expressões têm sua avaliação truncada ou não na sua linguagem favorita. Para isso, leia os manuais de definição da linguagem e experimente em alguns pequenos programas. Caso as expressões sejam truncadas, verifique se é proveniente da definição da linguagem ou do processador que você usa.

3. Estude os comandos de atribuição existentes em sua linguagem favorita. É possível fazer atribuições múltiplas. Todas são feitas ao mesmo tempo ou são atribuições sucessivas?

4. Dê argumentações contra e a favor de se ter expressões de atribuição na linguagem C. Para isso, discuta os aspectos de uniformidade e legibilidade da linguagem.

5. Mostre como os comandos `do-while` e o `repeat-until` podem ser facilmente re-escritos na semântica do `while`.

6. Liste os comandos de seleção da sua linguagem favorita e mostre como cada um deles pode ser reescrito usando o comando de seleção bidirecional (if-then-else).