4

Variáveis

"For the Snark was a Boojum, you see."

—Lewis Carroll

Como visto no capítulo anterior, os valores são agrupados em tipos para que determinemos um tratamento uniforme. Além disso, vimos que os tipos predefinidos nas linguagens de programação são formados por tipos ou estruturas matemáticas conhecidas, as quais usamos quando da solução de problemas reais. Contudo, para que as soluções dos problemas sejam concretizadas, precisamos, na maioria das vezes, de mecanismos para armazenar valores de forma que estes possam ser usados *a posteriori*. Em grande parte das linguagens de programação os valores são armazenados em **variáveis**, as quais permitem que os mesmos sejam usados em diversos pontos do programa.

Este capítulo tem como objetivo mostrar quais recursos dispomos nas linguagens de programação atuais para armazenar valores, de forma que possam ser acessados

e/ou modificados em passos posteriores da computação do programa. Para tanto, abordamos o uso das variáveis nos programas, classificamo-nas quanto ao armazenamento e acesso a valores, bem como quanto à sua existência como elemento ativo dentro do programa, ou seja, como pode ser feito o acesso às variáveis ao longo da execução do programa.

4.1 O PAPEL DAS VARIÁVEIS NOS PROGRAMAS

Para a solução dos problemas reais, devemos inicialmente escolher valores que denotem os valores reais do problema. A representação desses conjuntos de valores se dá por tipos, os quais podem ser tão simples como os tipos primitivos, quanto mais elaborados como os tipos compostos e todas as suas combinações (tipos construídos).

A solução para cada problema é uma transformação sucessiva dos valores originais até que obtenhamos os resultados desejados. Uma forma de abstrair da representação efetiva de cada valor é por um 'nome fantasia' que possa ter como conteúdo qualquer valor aceitável do tipo escolhido. Nas linguagens de programação, denominamos de **variáveis** os 'nomes fantasia' utilizados nos programas para representar valores que podem ser transformados (manipulados) ao longo da computação do programa.

Fazendo uma analogia com os seres humanos, cada um de nós poderia ser identificado pelo código genético individual. Além da identificação genética, cada um tem um conhecimento individual do mundo, o conteúdo de cada ser humano. Apesar da identificação genética não sofrer modificação ao longo do tempo (não em curto prazo da existência individual), os nossos conhecimentos são modificados ou renovados; adquirimos novos conhecimentos e/ou renovamos os existentes. Certamente não gostaríamos de identificar cada pessoa pela sua representação genética, e uma forma de contornar este problema é pelo registro de um nome para cada um dos seres humanos, um 'nome fantasia'. Assim, para cada ser humano, identificado pelo seu 'nome fantasia', podemos associar um conhecimento sobre um determinado assunto, o qual pode ser transformado ao longo do tempo.

De forma análoga, na execução de um programa, uma **variável** é um objeto identificado por um 'nome fantasia' que contém um valor (o seu conteúdo), o qual pode ser

consultado ou modificado tantas vezes quanto necessário ao longo da computação do programa. Essas variáveis são então usadas para modelar os objetos reais de forma que seus valores possam ser consultados ou modificados.

O armazenamento das variáveis é feito pelo uso de células de memória e o número de células usadas para o armazenamento de uma variável depende do tipo de dados associado e da linguagem de programação usada. É importante salientar que para cada variável de um programa temos associado: um identificador (o 'nome fantasia'), um endereço da(s) célula(s) de memória aonde o valor é armazenado, e mais o conteúdo (valor) no endereço de memória.

identificador → endereço de memória | conteúdo |

Quando criamos variáveis em um programa pela sua declaração (implícita ou explícita), estamos na realidade criando um espaço de memória que pode acomodar qualquer valor do tipo declarado. Além disso, criamos um 'nome fantasia' que queremos tratar, para não termos que lidar com o endereço físico da memória reservada.

Exemplo 4.1 – A declaração de variáveis e comandos de atribuição mostram como variáveis podem ser "criadas" e ter valores associados. O exemplo abaixo mostra um trecho de código na linguagem C:

```
int n;       {aloca uma célula de memória p/ o
              identificador n -valor indefinido}
n = 0;       {associa o valor zero como conteúdo}
n = n + 1;   {avalia a expressão ``n + 1'' com o valor
              atual de n e depois modifica o conteúdo
              mediante o resultado da expressão}
```

□

No exemplo acima, temos o identificador `n` associado a uma célula de memória a ser alocada pelo compilador e um conteúdo que é inicialmente indefinido. Na segunda linha de comando, `n` passa a ter como conteúdo o valor "zero", e posteriormente o valor "um". Note que a criação (declaração) de uma variável possui três subtarefas: associar um tipo à variável, alocar um espaço de memória apropriado, e associar a este espaço um identificador de forma que a partir desse ponto do programa o endereço da memória

reservada e o identificador sejam indistingüíveis. Contudo, o valor associado ao conteúdo da variável não é determinado na declaração da mesma, mas pela *atribuição*[1] de valores ao conteúdo das variáveis, como mostrado no Exemplo 4.1. Dessa forma, o conteúdo de uma variável pode ser modificado ao longo da computação do programa.

4.2 ARMAZENAMENTO E ACESSO A VALORES

Assim como os tipos de dados definidos no Capítulo 3, as variáveis podem tanto armazenar um conteúdo atômico quanto composto. Quando o armazenamento e acesso ao conteúdo da variável só pode ser realizado sobre o valor como um todo, dizemos que estas são **variáveis simples**. Por outro lado, vimos que para alguns tipos, os valores podem ser decompostos em valores mais simples, e certamente queremos armazenar e ter acesso aos valores mais simples individualmente nas variáveis das linguagens de programação. São chamadas de **variáveis compostas** aquelas que podem ser desmembradas em elementos mais simples. Nas seções que seguem, mostramos os conceitos sobre variáveis simples e compostas, como elas aparecem nas linguagens de programação e a relação entre os tipos de dados vistos e as variáveis.

4.2.1 Variáveis Simples

As variáveis simples denotam espaços de memória que podem ter seus conteúdos armazenados e acessados atomicamente nas linguagens de programação. Os tipos primitivos, por exemplo, têm valores atômicos e os conteúdos associados a cada variável que representa um tipo primitivo são armazenados e modificados como um todo; não há como armazenar apenas parte dos valores. Isso não significa que um mesmo espaço de memória se faz necessário para os vários tipos primitivos (isso depende da linguagem e do hardware em uso), mas que seu armazenamento é realizado de forma atômica. A relação entre as variáveis e os tipos de dados apresentados no Capítulo 3 (tipos primitivos, compostos e recursivos) é mostrada nas seções que seguem.

[1] Os comandos base de programação serão vistos em detalhes no Capítulo 6.

4.2.1.1 Tipos Primitivos Os valores dos tipos primitivos são sempre referidos como elementos atômicos, mesmo que sejam armazenados internamente na máquina por elementos que possam ser eventualmente desmembrados. Um número inteiro, por exemplo, é representado internamente, no mínimo, pelo seu sinal (positivo ou negativo) mais um valor. Contudo, nas linguagens de programação, nos referimos aos valores inteiros, os quais podem ser um número inteiro positivo ou negativo, e não os desmembramos em sinal e valor. Da mesma forma, os números reais são representados internamente por um expoente e uma parte fracionária, mas esses elementos não aparecem desmembrados nas linguagens de programação; os números reais são armazenados como um valor de ponto-flutuante. Assim, os tipos primitivos numéricos estudados são armazenados em variáveis simples.

Exemplo 4.2 – Dado que os tipos numéricos primitivos são predefinidos nas linguagens de programação, variáveis podem ser declaradas como de um dado tipo predefinido. Na linguagem C, por exemplo, podemos declarar variáveis do tipo inteiro e ponto-flutuante de forma que possamos acessar e armazenar valores do tipo correspondente como conteúdo.

```
int i1, i2;   {variáveis i1 e i2 declaradas       }
float r;      {variável r declarada               }
...
i1 = 10;      {valor 10 armazenado na variável i1 }
i2 = i1 - 20; {valor da variável i1 é acessado e
               o valor resultado da expressão é
               armazenado na variável i2          }
r = 20.5;     {valor 20.5 armazenado na variável r}
```

Note que para os inteiros, o acesso e o armazenamento dos valores são realizadas de forma atômica, independentemente se o valor inteiro é negativo ou positivo. Os números de ponto-flutuante têm acesso e armazenamento de forma semelhante. □

Da mesma forma que para os tipos numéricos, os valores caracteres e booleanos também são armazenados em variáveis simples. Vale lembrar que o tipo booleano não é um tipo predefinido na linguagem C. Contudo, este tipo pode ser construído

através da enumeração, como mostra o exemplo a seguir, e assim tratado em variáveis simples como quaisquer dos tipos enumerados.

Exemplo 4.3 – O tipo booleano pode ser definido como uma enumeração de valores na linguagem C como segue:

```
enum Bool {false, true};
...
 Bool b;                  (1)
...
 b = true ;               (2)
...
 if (b == true) then ... (3)
```

Nesta linguagem, o primeiro elemento da enumeração corresponde ao valor "zero" e o segundo ao valor "um". Além disso, um valor enumerado é armazenado (2) em uma variável simples do mesmo tipo da enumeração, e o acesso à variável (3) é como em quaisquer outras variáveis simples, acesso direto ao valor correspondente. Neste exemplo, o valor da variável é comparado com um dos valores da enumeração definida. □

Os tipos enumerados da linguagem Pascal também têm seus valores armazenados em variáveis simples. Nesta linguagem, podemos ter a enumeração explícita dos valores, assim como a definição de um novo tipo como subintervalo de um tipo existente. Em ambos os casos são usadas as variáveis simples para o armazenamento desses valores.

Outro tipo primitivo comum nas linguagens de programação são os apontadores, os quais têm como valores endereços de memória aonde se encontram os conteúdos (valores) desejados. Os apontadores têm uma característica especial de armazenar um endereço de memória e a partir deste temos acesso aos valores armazenados no endereço de memória indicado. Assim, apesar de variáveis deste tipo armazenarem um único valor, e por isso usamos variáveis simples, podemos ter acesso também ao conteúdo do endereço de memória indicado via operações predefinidas nas linguagens.

Os exemplos abaixo mostram o uso de apontadores nas linguagens Pascal e C.

Exemplo 4.4 – Na linguagem Pascal, um apontador para um valor inteiro pode ser definido por:

```
var int_ptr1, int_ptr2 : ^ Integer;
                      {apontadores para inteiros       }
    i : Integer;
...
 New(int_ptr1);       {cria um espaço para armazenar
                       um inteiro apontado por int_ptr1}
 i:= 10;
 int_ptr2 := @i;      {armazena em int_ptr2 o endereço
                       da variável i                    }
...
 int_ptr1^ := i;      {armazena o valor de i no espaço
                       apontado por int_ptr1            }
...
 Dispose(int_ptr1);   {libera o espaço reservado para
                       armazenar o inteiro apontado por
                       int_ptr1                         }
...
```

□

Exemplo 4.5 – De forma análoga, na linguagem C, um apontador para um valor inteiro pode ser definido por:

```
int *int_ptr1, *int_ptr2;  {apontadores para inteiros}
int  i;
...
 int_ptr1= (int*) malloc(sizeof(int))
                      {cria um espaço para armazenar
                       um inteiro apontado por int_ptr1}
 i= 10;
```

```
 int_ptr2 = &i;       {armazena em int_ptr2 o endereço
                       da variável i                  }
...
 *int_ptr1 = i;       {armazena o valor de i no espaço
                       apontado por int_ptr1          }
...
 free(int_ptr1);      {libera o espaço reservado para
                       armazenar o inteiro apontado por
                       int_ptr1}
...
```

□

Na linguagem C, os apontadores são mais flexíveis e operações aritméticas podem ser realizadas sobre apontadores. Dessa forma, os programadores têm acesso direto aos elementos de memória e um cuidado extra se faz necessário no tratamento de apontadores. Na linguagem Pascal, os apontadores são usados apenas para acessar variáveis anônimas alocadas dinamicamente, o que é uma forma mais restrita de uso. Nas duas linguagens temos operações para reservar dinamicamente os espaços de memória, assim como liberar os respectivos espaços. Isso se faz necessário para a introdução da idéia de administração de espaços de memória pelos próprios programadores. Grande parte das linguagens ainda hoje não fornecem mecanismos para liberação automática de espaços de variáveis alocadas dinamicamente, dado o custo de processamento de tal mecanismo. O tratamento de coleta de espaços de memória não utilizados está relacionado à implementação de processadores de linguagens, assunto não tratado neste livro (ver leitura recomendada no final do capítulo).

É importante salientar que apesar de podermos ter acesso ao endereço de memória e ao conteúdo associado, essa não é uma variável estruturada (composta), pois o valor a ser tratado é unicamente o conteúdo associado.

4.2.1.2 Tipos Compostos A maioria dos tipos compostos tem seus valores tratados de forma seletiva e por isso armazenados em variáveis composts, como será visto na Seção 4.2.2. Os conjuntos, contudo, como implementados em Pascal e

Modula-3, são tratados como um único elemento e não existe acesso direto aos seus componentes (ao i-ésimo componente, por exemplo). Variáveis do tipo conjunto podem apenas ser manipuladas por operações predefinidas sobre conjuntos. Apesar de ser um tipo composto, o acesso e armazenamento dos conjuntos são feitos de forma não seletiva, o que justifica o uso de variáveis simples para tanto.

Exemplo 4.6 – O conjunto das cores primárias e todas as outras cores formadas a partir destas pode ser descrito na linguagem Pascal como segue:

```
type Cores = (vermelho, azul, amarelo);
     NovasCores = set of Cores;
...
var conjcor: NovasCores;          (1)
...
conjcor := [vermelho, amarelo];   (2)
...
if (vermelho in conjcor)          (3)
...
```

os conjuntos são tratados como um todo: definido por inteiro (2). Não há como acessar o i-ésimo elemento do conjunto, contudo operações como pertinência (3) união, intersecção e igualdade de conjuntos também estão definidas nestas linguagens. □

As operações sobre conjuntos estão predefinidas tanto na linguagem Pascal quanto na Modula-3. Apesar de representarem um tipo de dados importante para a modelagem de problemas reais, existem algumas limitações do uso de variáveis do tipo conjunto devido a implementações. As variáveis do tipo conjunto, à semelhança das variáveis do tipo enumeração, não podem ser entrada nem saída de funções tanto em Pascal quanto em Modula-3. Além disso, existe um limite máximo para os tamanhos de conjuntos na maioria das implementações; a maioria limita o máximo menor que 100 elementos. Isso se dá porque a maioria implementa os conjuntos e suas operações sobre *strings* de bits que cabem em uma única palavra de máquina por razões de eficiência.

4.2.1.3 Tipos Recursivos

As listas são o tipo recursivo mais comum nas linguagens de programação atuais. Na maioria das linguagens, as listas são representadas por variáveis que selecionam um valor e um apontador para o próximo elemento da lista, as quais serão mostradas na Seção 4.2.2.4. Algumas outras linguagens possuem listas como tipo predefinido.

Linguagens que embutem listas como tipo predefinido, na sua maioria as linguagens funcionais tais como Lisp, SML e Miranda, não fazem acesso e armazenamento seletivos dos seus elementos. Em SML, por exemplo, existem operações predefinidas comuns ao tipo lista: primeiro elemento, inserir um elemento, retirar o primeiro elemento, concatenar listas, etc (mostrado na Seção 3.4). Contudo, as listas são tratadas como elementos únicos, não podemos ter acesso ao i-ésimo elemento da lista, exceto via operações sucessivas.

Exemplo 4.7 – Na linguagem SML, uma lista de inteiros pode ser construída como segue:

```
5::3::2::nil =  5::3::[2] =  5::[3,2] = [5,3,2]
```

Apesar de variáveis não serem comumente usadas em definições nas linguagens funcionais[2], elas podem ser usadas para simplificar a aplicação de operações.

```
val list1 =  [5,3,2]; {definição da variável list1XS}
> [5,3,2] : int list
  rev list1;          {lista em ordem inversa}
> [2,3,5] : int list
  hd list1;           {primeiro elemento da lista}
> 5: int
  tl list1;           {descarta o primeiro elemento}
> [3,2] : int list
  hd tl list1;        {segundo elemento da lista}
> 3: int
```

[2]Linguagens funcionais puras são orientadas a expressões em vez de comandos, como será visto no Capítulo 6.

```
  hd rev list1;       {último elemento da lista}
> 2: int
```

Veja que o segundo elemento de `lista` só pode ser acessado pelas operações sobre listas, não há seleção direta do elemento. Da mesma forma, o último elemento também só pode ser obtido por operações (ou mediante novas definições de funções). □

De forma semelhante, as listas são implementadas em outras linguagens funcionais e em linguagens lógicas, como será visto na Parte II.

4.2.2 Variáveis Compostas

As variáveis compostas possuem componentes que podem ser tratados seletivamente. Como visto no Capítulo 3, os tipos compostos possuem, em geral, acesso seletivo aos seus subcomponentes. Vimos nas Seções 4.2.1.2 e 4.2.1.3 que os conjuntos e as listas, apesar de serem tipos compostos, são representados em algumas linguagens por variáveis simples. Contudo, a maioria dos tipos compostos é representada por variáveis compostas. Aqui seguem os tipos compostos definidos anteriormente e que tipo de variáveis se faz necessário para o seu armazenamento e acesso nas linguagens de programação.

4.2.2.1 Produto Cartesiano Os produtos cartesianos são representados através de registros na linguagem Pascal (`record`) e estruturas (`struct`) na linguagem C.

Exemplo 4.8 – Como visto no capítulo anterior, podemos definir um novo tipo para representar datas na linguagem C como segue[3]:

```
enum MesesC {jan, fev, mar, abr, mai, jun,
             jul, ago, set, out, nov, dez};
enum DiasC {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
            16,17,18,19,20,21,22,23,24,25,26,27,
            28,29,30,31};
struct DataC {
```

[3] Aqui repetimos tipos já vistos em exemplos anteriores para facilitar a leitura.

```
              DiasC  d ;
              MesesC m
             };
...
DataC umadata;      (1)
...
umadata.m = mar;    (2)
umadata.d = 7;
...
```

Note que cada um dos elementos que constitue a variável `umadata` (1), definida como do tipo `DataC`, pode ser armazenado e acessado seletivamente através dos nomes que discriminam os subelementos (`m` e `d`), o que as caracteriza como variáveis compostas. □

Na linguagem Pascal, as variáveis definidas por tipos que incluem um `record` têm um tratamento semelhante às estruturas (`struct`) da linguagem C; cada subelemento do `record` pode ser armazenado e acessado de forma direta como no exemplo acima. A grande maioria das linguagens de programação atuais fornecem definição de variáveis que acomodam produto cartesiano. É importante salientar que cada variável só pode armazenar um valor por vez em cada um de seus subelementos, contudo o universo de valores que podem ser representados em cada uma das variáveis é dado pelo produto cartesiano.

4.2.2.2 União disjunta A união disjunta provida nas linguagens de programação são caracterizadas por mais de uma variável compartilhando um mesmo espaço de memória. Contudo, apenas uma das variáveis deve ser acessível por vez. Apesar de termos apenas uma das variáveis acessível em um dado instante, podemos acessar este espaço de memória de forma seletiva, o que caracteriza a variável como composta. A seguir, há alguns exemplos de variáveis que representam união disjunta nas linguagens C e Pascal.

Exemplo 4.9 – Como definido no Exemplo 3.16, podemos ter números que para certos propósitos do problema deve ser do tipo inteiro, enquanto em outras

partes do problema deve ser tratado como de ponto-flutuante. Um número com tais características pode ser definido na linguagem C como:

```
union NumeroC {
               int   ival;
               float rval;
              };
...
 NumeroC valor;
...
 valor.ival = 5;                 (1)
 valor.ival = valor.ival + 10;   (2)
...
 valor.rval = 5.87;              (3)
 valor.rval = valor.rval + 20.0; (4)
...
```

A variável `valor` pode ter um valor inteiro ou real, depende do valor armazenado; se o `valor.ival` ou o `valor.rval`. Quando um valor é atribuído a `valor.ival` (1) apenas esta variável deve ser usada (2). A variável `valor.rval` não possui qualquer valor no estágio (2) do programa e, portanto, não deveria ser usada. Da mesma forma, no estágio (3) do programa não há qualquer valor associado à variável `valor.ival`, e esta não poderia ser usada no passo seguinte (4) da computação do programa. □

No exemplo acima, as variáveis alternativas foram usadas devidamente. Contudo, o acesso à `valor.rval` quando o último valor armazenado foi para a `valor.ival` é permitido pela linguagem apesar de ser um uso incorreto (o valor depende da implementação). É de responsabilidade do programador o uso devido das variáveis união na linguagem C.

Além da união livre existente na linguagem C, algumas outras linguagens, tal como Pascal, possuem variáveis que representam a união disjunta com um discriminador, a união discriminada.

Exemplo 4.10 – Com o mesmo propósito do `NumeroC` acima, podemos criar na linguagem Pascal o tipo `NumeroP`, previamente definido no Exemplo 3.15:

```
type Precisao = {exato, aprox};
     NumeroP = record
                case prec : Precisao of
                    exato : (ival : Integer);
                    aprox : (rval : Real)
               end;
var num : NumeroP;
    i   : Integer;
    r   : Real;
...
 case num.prec of
   exato: i := num.ival;
   aprox: r := num.rval
 end
...
```

A variável `num.prec` terá exclusivamente um dos valores do tipo `Precisao` e, dependendo desse valor, apenas uma das variáveis `num.ival` ou `num.rval` poderá ser acessada. Este é o uso correto do tipo `NumeroP`, onde o discriminador `prec` é verificado antes que um dos valores alternativos seja usado. □

Vale salientar que para variáveis do tipo `NumeroP`, definido como um `record`, é permitido o acesso direto a quaisquer dos seus elementos. Então, se tivermos em um dado estado o valor `exato` associado à `num.prec` e tentarmos acessar `num.rval`, ocorrerá um erro em tempo de execução. Por isso, recomenda-se o uso de variáveis de união disjunta com a prévia verificação do discriminador.

Da mesma forma que na união da linguagem C, a união disjunta na Pascal requer uma variável composta para sua representação. Contudo, faz-se necessário nesta última um elemento a mais, o discriminador, como visto no exemplo acima.

4.2.2.3 Mapeamentos Nos mapeamentos usuais das linguagens de programação (os *arrays*), temos acesso tanto aos elementos do domínio (índices) quanto aos elementos imagem. Na maioria das linguagens de programação, temos apenas acesso aos valores imagem mediante os índices; dado um índice podemos ter acesso ao valor imagem associado àquele índice.

Exemplo 4.11 – Suponha um mapeamento com 16 elementos definido sobre os conjuntos domínio e imagem de valores inteiros como segue:

```
int mapintC[16], n;
...
mapintC[0] = 10;
mapintC[15] = 14;      (1)
n = mapintC[15];       (2)
...
```

Por definição de projeto na linguagem C, quando um *array* é declarado com 16 elementos tem-se os índices (elementos do domínio) variando de 0 (zero) a 15 do tipo inteiro. O armazenamento (1) e acesso (2) a cada um dos elementos imagem é realizado de forma direta mediante o índice. □

Exemplo 4.12 – Na linguagem Pascal, uma variável que representa um mapeamento de inteiros no intervalo $[0\,..\,15]$ para números inteiros representáveis na linguagem pode ser definido por:

```
var mapintP:  array [0..15] of Integer;
...
mapintP[0] := 10;
mapintP[15] := 14;      (1)
n := mapintP[15];       (2)
...
```

Note que os índices devem ser definidos explicitamente na linguagem, e não há obrigatoriedade que estes sejam inteiros a partir de 0 como na linguagem C. □

Assim, o tipo associado aos domínios dos mapeamentos (índices) podem ser tanto predefinidos pela própria linguagem, como nas linguagens C, C++ [5] e Java [34] (inteiros a partir de 0), quanto definidos pelo programador, como definido em Pascal, Modula-3 e Ada (inteiros ou tipo enumerado). Na maioria das linguagens, o tipo da imagem pode ser um tipo predefinido na linguagem assim como tipos criados pelo usuário.

Ainda em relação aos índices, podemos ter apenas um índice ou vários dependendo da linguagem. A grande maioria das linguagens admite *arrays* multidimensionais, com vários índices. A linguagem Fortran, por exemplo, limitou a três o número de índices por razões de eficiência. Por outro lado, a linguagem C admite apenas um índice, mas cada um dos elementos, por sua vez, pode ser um *array* (princípio da ortogonalidade), e temos como resultado os *arrays* multidimensionais da linguagem.

Em geral, a atribuição aos valores imagem é realizado um a um, como visto nos exemplos anteriores. Contudo, algumas linguagens, tal como C, permitem o armazenamento dos valores de um *array* tanto de forma seletiva, como no exemplo acima, quanto de forma integral, o que é o caso do exemplo abaixo.

Exemplo 4.13 – Aqui um *array* de inteiros com 5 elementos é definido, assim como os valores associados a cada um dos elementos:

```
int n[5] = {32,45,66,23,58};
...
n[3] = 40;                          (1)
...
```

Note que o fato de poder armazenar todos os elementos de um *array* de uma única vez não o descaracteriza como uma variável composta, já que seus subcomponentes podem tanto ser armazenados (1) quanto acessados de forma seletiva. □

Além do armazenamento integral de um *array*, algumas linguagens permitem que em um *array* bidimensional (uma matriz), por exemplo, uma linha inteira seja armazenada de uma única vez. Variações de fatias de *arrays* podem ser armazenados

em algumas linguagens, tal como na Fortran. Outras linguagens, tais como SML, Ada [29] e Perl, também permitem o armazenamento de variáveis compostas de forma integral ou parcial.

4.2.2.4 Tipos Recursivos Os tipos recursivos mais conhecidos nas linguagens de programação são as listas. Na maioria das linguagens, são criadas pelo usuário como uma nova estrutura de dados. Desta forma, os elementos da estrutura são tratados seletivamente, assim como em qualquer das estruturas criadas pelo usuário. Contudo, as listas predefinidas das linguagens de programação funcionais são tratadas como um único elemento (assim como os conjuntos em Pascal), não se tendo acesso seletivo aos seus elementos, apenas via operações predefinidas (Seção 4.2.1).

Como já visto, os tipos recursivos podem ser definidos através um novo tipo que tem um de seus subelementos declarado como do seu próprio tipo. Um dos exemplos clássicos são as listas que podem ter seus tipos definidos por apontadores para uma lista do mesmo tipo. Assim, as variáveis destes tipos compostos são também variáveis compostas que podem ter acesso tanto aos valores quanto às listas.

Exemplo 4.14 – Na linguagem C, uma lista de números inteiros pode ser definida por apontadores como segue:

```
typedef struct NoIntlist* IntListC ;
struct  NoIntlist {
                    int valor;
                    IntListC *prox;
                   }
...
IntlistC prim_linteiros, ult_linteiros;
...
prim_linteiros = malloc(sizeof *prim_linteiros);       (1)
prim_linteiros->valor = 10;                            (2)
prim_linteiros->prox = malloc(sizeof *prim_linteiros)  (3)
ult_linteiros = *prim_linteiros.prox                   (4)
ult_linteiros->valor = 20;                             (5)
```

```
ult_linteiros->prox = NULL;                                    (6)
...
```

`prox` é um apontador para uma estrutura do tipo `IntlistC`. Como o novo tipo é recursivo, um dos elementos da célula é um apontador para um outro elemento do mesmo tipo. Uma variável do tipo lista terá acesso seletivo a `valor` (2), a qual é um inteiro, e a `prox` (3) que é um apontador para uma lista de inteiros. `prim_linteiros` é um apontador, então deve ter um espaço de memória alocado (1) para um célula do tipo `NoIntlist` antes de ser usado. Da mesma forma, uma nova célula é alocada e apontada por `prim_linteiros->prox` (3) e posteriormente também apontada por `ult_linteiros` (4). Por fim, um valor inteiro é atribuído ao elemento `valor` desta nova célula (5) e o elemento `prox` recebe `NULL` (6), indicando que não há mais elementos na lista. □

4.3 AS VARIÁVEIS E SUA EXISTÊNCIA

Uma outra característica importante das variáveis está relacionada à ativação da variável ao longo da execução do programa; **quando** os valores das variáveis podem ser acessados. Assim, além de classificar as variáveis sob o ponto de vista de acesso e armazenamento de valores (variáveis simples e compostas), é preciso classificá-las quanto a sua existência quando da execução dos programas (o "tempo de vida"das variáveis).

Uma variável existe desde o momento em que é associada (vinculada) a uma célula de memória (a chamada **alocação**) da variável, e cessa sua existência quando o dado espaço de memória é disponibilizado (**desalocação**). Nas linguagens de programação atuais, existem as variáveis que têm sua alocação antes que se inicie a execução dos blocos de comando dos programas (**variáveis globais e locais**), as variáveis que são criadas em tempo de execução (**variáveis *heap***), e as **variáveis persistentes**, as quais existem mesmo depois da execução dos programas.

A vinculação ao armazenamento das variáveis determina sua existência na computação de um programa. O caráter de uma linguagem de programação é, em grande parte, determinado pelo projeto das vinculações de armazenamento para as suas variáveis. No

Capítulo 5, as vinculações em linguagens de programação e sua importância nos projetos das linguagens serão discutidas. Nas seções que seguem, apresentamos apenas uma classificação para as variáveis quanto à sua existência, e problemas relacionados à existência das variáveis.

4.3.1 Variáveis Globais e Locais

As variáveis **globais** são aquelas vinculadas às células de memória antes da execução do programa e assim permanecem até que a execução do programa se encerre. O tempo de vida das variáveis **locais** está relacionado aos blocos de execução nos quais elas estão inseridas. Dessa forma, as variáveis locais têm seu escopo de existência delimitado pela ativação do bloco de execução no qual está inserida. Esses blocos individuais podem ser ativados sucessivas vezes durante a execução de um programa, contudo, os valores das variáveis locais aos blocos são novos a cada ativação do bloco na maioria das linguagens. Ou seja, o valor das variáveis locais não são retidos após o término da ativação do bloco. Uma discussão sobre blocos de comandos das linguagens atuais será conduzida no Capítulo 6.

Linguagens como Pascal, Modula-3, Ada e C, possuem blocos de comandos nos quais podemos ter definidas estas variáveis locais, que são acessíveis apenas no tempo de execução dos blocos. C, C++ e Java, contudo, permitem que valores de variáveis locais sejam retidos, quando as variáveis são declaradas como static (um atributo de variáveis que as torna globais).

4.3.2 Variáveis Intermitentes (*Heap*)

Algumas variáveis são criadas e destruídas em tempo de execução, as variáveis intermitentes. Os apontadores existentes na maioria das linguagens de programação atuais são exemplos clássicos deste tipo de variável. Para esse tipo de variável, as células de memória são associadas mediante um comando de criação da variável (alocação). Da mesma forma, a destruição das mesmas também é feita via comando específico (desalocação), ou através de uma coleta de lixo após a execução do programa nas linguagens mais modernas.

A existência temporária destas variáveis não deve ser confundida com as variáveis locais. O que difere substancialmente estas variáveis das locais é o controle explícito da sua existência pelos comandos de criação e destruição, enquanto as variáveis locais têm sua existência relacionada a blocos de execução.

Tanto as variáveis globais e locais, quanto as *heap* são variáveis com existência limitada ao tempo de execução dos programas.

4.3.3 Variáveis Persistentes

O armazenamento de dados não pode estar restrito apenas ao tempo de execução dos programas. É comum utilizarmos dados já previamente processados ou simplesmente armazenados. Para dados que são utilizados várias vezes, utilizamos arquivos (ou bancos de dados) como meio de armazenamento. Para que se tenha acesso a esta informação através dos programas, devemos ter variáveis do tipo arquivo. A maioria das linguagens de programação possui variáveis do tipo arquivo e operações predefinidas sobre eles para que se tenha acesso às informações.

Na maioria das linguagens não há como criar ou destruir arquivos, Ada é uma exceção a essas linguagens. Em C, a criação de arquivos não faz parte da biblioteca padrão.

4.3.4 Problemas Relacionados à Existência das Variáveis

A priori, variáveis só devem ter seus valores acessados quando estão ativas em tempo de execução. Não obstante, alguns problemas podem ocasionar erros por acessos indevidos a valores de variáveis.

Exemplo 4.15 – Considere um pseudoprograma na linguagem Pascal aonde temos um procedimento e um programa que chama este procedimento:

```
var r: ^ Integer;
procedure P;
 var v: Integer:
 begin
  r:= &v;
```

```
 end;

begin
 P ;
 r^ := 1;
end
```

Note que no exemplo acima `r` é uma referência para a variável local `v`, a qual é não existente quando da execução do programa principal. Isso certamente acarretaria um erro: acesso a uma variável não existente. □

Pascal evita esse tipo de problema com a não permissão de referência a variáveis locais (referências a variáveis locais devem sempre ser passadas como parâmetros). Em outras linguagens, a atribuição de referências só pode ser feita para variáveis que tenham um tempo de vida menor (o que requer verificações em tempo de execução).

Em algumas linguagens funcionais, quando é feita uma atribuição de referência a uma variável, a variável referida só pode ser destruída quando não há mais referências para ela. Isso requer um tratamento em tempo de execução, o que degrada a eficiência da linguagem.

4.4 LEITURA RECOMENDADA

Um estudo sobre variáveis temporárias e persistentes pode ser encontrado em [14].

Métodos de implementação para tipos de dados são descritos de forma resumida em [54], e de forma mais sistemática em livros de implementação de linguagens como [3] e [65].

Vários estudos foram desenvovidos para a coleta de lixo das variáveis *heap*, dada a sua importância nas linguagens modernas. Uma discussão sobre o assunto pode ser encontrada em [22]. Bons estudos para os leitores interessados no assunto também podem ser encontrados em [70, 27] e [22].

4.5 EXERCÍCIOS

1. Algumas linguagens de programação permitem que variáveis sejam definidas sem um tipo. Quais as vantagens e desvantagens de termos variáveis sem um tipo associado em uma linguagem de programação?

2. Listas com criação dinâmica de elementos podem ser implementadas em Pascal pelos apontadores. Considere que uma extensão de Pascal inclui o tipo lista:

   ```
   var <nome-variável> : list of <nome-tipo>
   ```

   sobre o qual estão definidas operações de atribuição entre listas do mesmo tipo, construção de listas (um elemento e uma lista são os parâmetros), concatenação de listas, *head, tail* e *length*. As listas podem conter elementos do tipo primitivo ou composto, e na operação de atribuição, quando uma variável lista é atribuída à outra, é criada apenas uma referência de forma que as duas variáveis acessam a mesma lista de valores.

   ```
   type IntList = ÎntNode;
        IntNode = record elem: Integer;
                         prox: IntList
                  end;
   var ll : list of Integer
       lp : IntList
   ```

   Compare as `ll` e `lp` listas acima quanto a:

   (a) o conjunto de valores que elas podem representar;

   (b) a forma de acesso aos elementos da lista (o primeiro, e um elemento qualquer da lista); e

   (c) o mecanismo de atribuição para cada um dos tipos das listas.

3. Classifique as variáveis `ll` e `lp` do exemplo acima quanto ao armazenamento e acesso a valores (Seção 4.2), e quanto à sua existência (Seção 4.3).

4. As *strings* são representadas de diversas formas nas linguagens de programação atuais. Para as linguagens de programação que você usa, classifique as variáveis que representam *strings* quanto ao acesso e armazenamento (Seção 4.2), e explique as razões da classificação.

5. Analise o uso de apontadores em C++ e as variáveis de referência Java para variáveis *heap* sob os pontos de vista de segurança e facilidade de uso. Para complementar esta análise, faça uma breve discussão do ganho e perdas na decisão dos projetistas de Java da não inclusão de apontadores na linguagem.

6. Faça um paralelo sobre as vantagens e desvantagens de se ter liberação de memória ocupada pelas variáveis *heap* de forma manual (realizada explicitamente pelo programador, como em C++), ou de forma automática, como em Java.

7. Em geral, os tipos primitivos são armazenados em variáveis simples, enquanto tipos compostos, em variáveis compostas. Determine, para as suas linguagens favoritas, os tipos predefinidos e quais tipos de variáveis que são necessários para cada tipo predefinido.

8. Apenas algumas linguagens possuem conjuntos como elementos predefinidos na linguagem. Enuncie alguns problemas para os quais a representação de conjuntos facilita a manipulação dos dados.

9. A linguagem C, por exemplo, não possui conjuntos predefinidos. Faça uma representação para conjuntos usando a linguagem C (dica: lembre-se de que algumas operações sobre conjuntos se faz necessário - ex: união).

10. Os índices dos *arrays* são predefinidos como números naturais na linguagem C.

    (a) Cite vantagens e desvantagens sobre esta representação se comparada a outras linguagens que exigem a definição dos tipos dos índices.

    (b) Cite problemas para os quais você gostaria de ter a representação dos índices diferente do domínio dos números naturais (ex: caracteres, enumerados, etc).

(c) Para os problemas citados no item anterior, como você representaria os dados usando a linguagem C?

11. Várias linguagens modernas permitem a criação de tipos enumerados (novos valores determinados pelo programador). Contudo, esses valores são associados, internamente, a valores inteiros e, em algumas linguagens, operações sobre inteiros podem ser efetuadas sobre os valores enumerados. Discuta o uso de valores enumerados como inteiros à luz dos requisitos de flexibilidade, clareza e facilidade de escrita desejáveis às linguagens de programação atuais.