7

Abstrações

"Reach what you cannot."

—Nikos Kazantzakis

Abstração é o princípio pelo qual nos concentramos nos aspectos essenciais de um problema em vez de seus detalhes. Sob o ponto de vista de linguagens de programação, as abstrações são usadas para resolver os problemas de uma forma genérica, e não resolver apenas problemas particulares. A idéia de se ter um programa como solução de uma classe de problemas reflete de uma certa forma as abstrações. Um programa tem como princípio a solução de um problema computacional e pode ser aplicado a diferentes dados. Ele é, portanto, uma forma abstrata de resolver um problema e a sua execução com dados específicos é uma solução particular de um problema.

A solução dos problemas computacionais por um programa de forma monolítica (um único programa, sem subprogramas) tornou-se insuficiente à medida que aumentou a complexidade dos problemas a serem resolvidos. Este é apenas conveniente

quando os problemas são muito simples. Para que os programas sejam subdivididos em unidades de processamento que agrupem conjuntos de comandos, precisamos de elementos que denotem estas unidades de processamento. Além disso, usando como princípio a uniformidade, cada unidade de processamento deve ser uma abstração da solução de um problema, da mesma forma que os programas.

Neste capítulo, abordaremos a idéia de abstrações relativas a comportamento, variáveis e tipos encontradas nas linguagens de programação.

7.1 TIPOS DE ABSTRAÇÕES

Quando falamos de abstrações, referimo-nos a soluções computacionais que podem ser vistas sob diferentes perspectivas: quem constrói e quem usa a abstração. O programador que constrói a abstração precisa do conhecimento de **como** o problema é resolvido. Enquanto o que usa a abstração precisa apenas conhecer **o que** ela faz. Este último precisa apenas saber usar o programa com os dados adequados, abstraindo-se de como o programa foi implementado.

Para as abstrações computacionais, precisamos reconhecer de quais elementos nos abstraímos nas subunidades de programas disponíveis nas linguagens de programação atuais. A definição (ou representação) de uma função nas linguagens de programação, por exemplo, embute uma expressão a ser avaliada e o programador usuário da função não precisa saber como tal expressão foi implementada, mas o que ela faz. Dessa forma, uma função embute, na verdade, o **processo** computacional usado na solução da função.

De uma forma geral, encontramos nas linguagens de programação as abstrações de **processos**, e as abstrações de **tipos**. As seções que seguem tratam em detalhes cada um desses tipos de abstrações e de que forma elas aparecem nas linguagens de programação atuais.

7.2 ABSTRAÇÃO DE PROCESSOS

Sob o ponto de vista computacional, uma abstração de processos deve "esconder"os passos computacionais necessários para solucionar o problema. Abstrações sobre

mapeamentos de domínios podem ser representadas por funções, assim como comportamentos com efeito colateral podem ser abstraídos por procedimentos.

7.2.1 Funções

Computacionalmente, uma função é representada por uma expressão que denota o comportamento do mapeamento entre um domínio e a imagem da função. A aplicação de uma função resulta em um valor: o elemento da imagem relacionado com um dado elemento do domínio. O usuário de uma função observa apenas o valor resultado, em vez de passos de avaliação da expressão.

Exemplos de abstrações de funções podem ser encontrados nas várias linguagens de programação imperativa, funcional e orientadas a objetos. Nelas, a abstração de uma função é construída pela definição da função. Na linguagem Pascal, por exemplo, a definição de uma função aparece da seguinte forma:

```
function F(P1; ... ; Pn) : T; <bloco>
```

onde `F` é o identificador da função, `Pi` são os parâmetros formais, `T` é o tipo do valor resultado e `bloco` é um bloco de comandos que contém as declarações locais e os comandos que descrevem o comportamento da função. Dentro desse bloco de comandos deve haver pelo menos o comando `F := E`, o qual denota o resultado da função.

Exemplo 7.1 – Uma função que calcula a potência de um número real a um expoente inteiro pode ser definida em Pascal como segue:

```
function pot (x:Real; n:Integer): Real;
 begin (*assumimos n>0*)
  if n = 1 then pot := x
           else pot := x * pot(x, n-1)
 end
```

A chamada (aplicação) dessa função com o número real 10.0 elevado a 2 é escrita por "`pot (10,2)`". □

Na função acima, usada-se uma pseudovariável com o mesmo nome da função (`pot`). Esta é uma expressão não-pura, característica de várias linguagens impera-

tivas. Aparentemente, estamos apenas usando a recursividade, mas uma variável é criada a cada chamada recursiva e posta em uma pilha de solução da função. Os valores dessas variáveis são atribuídos à medida que as chamadas recursivas da função são resolvidas.

A linguagem C possui uma forma semelhante para a definição de funções com uma sintaxe um pouco diferente:

```
T F (P1, ..., Pn) <bloco>
```

`T` representa o tipo do valor resultado da função, `F` o identificador da função, `Pi` os parâmetros formais e `bloco` os comandos ou expressão que implementam o comportamento da função.

Exemplo 7.2 – A função potência mostrada no exemplo anterior pode ser definida na linguagem C da seguinte forma:

```
float pot (float x, int n)
 { /*assumimos n>0*
  float potencia;
  if (n = 1)
       {potencia = x ;}
  else {potencia = x * pot(x, n-1);}
  return (potencia);
 }
```

Aqui, uma nova variável foi criada para acumular o resultado da função, não há uma pseudo-variável como em Pascal. A cada chamada recursiva é criada uma nova variável `potencia`. Esta função, em particular, poderia ainda ser definida na linguagem C com o uso de uma expressão, sem a criação da variável local:

```
float pot (float x, int n)
 { /*assumimos n>0*
  if (n = 1)
       {return (x) ;}
  else {return (x * pot(x, n-1));}
```

}

Note que nesta última forma de definição não foi usado o recurso de efeito colateral sobre o estado do programa, mas apenas a avaliação de expressões (o que é conhecido como expressão pura). □

Para este tipo particular de funções, não há necessidade de variáveis intermediárias para guardar os valores; as expressões são empilhadas nas chamadas recursivas.

A maioria das linguagens funcionais, como SML, também usa expressões puras para prover funções.

Exemplo 7.3 – O mesmo exemplo acima pode ser definido em SML como segue:

```
fun pot (x: real, n:int) = if n=1 then x
                                  else x * power(x,n-1)
```

□

As linguagens funcionais, a serem discutidas no Capítulo 8, têm como princípio a solução de problemas pelas expressões puras. Das linguagens imperativas, C é uma das que permite algumas expressões puras, apesar de admitir os efeitos colaterais, como mostrado acima.

7.2.2 Procedimentos

As abstrações de procedimentos são realizadas por uma série de comandos que provoca mudanças sucessivas nos valores de suas variáveis. Da mesma forma que as funções, elas são denotadas pelas suas definições e classificadas como abstrações de processos por descreverem um comportamento computacional. O usuário destas não precisa conhecer os passos internos de sua computação, mas apenas a mudança de estado do programa provocado por elas. A chamada de um procedimento é observado pelo usuário apenas como uma mudança no estado do programa influenciada pelos argumentos fornecidos por ele.

A maioria das linguagens de programação imperativas possui formas de abstrair procedimentos, uma vez que elas são essencialmente baseadas em comandos. Temos, por exemplo em Pascal, os procedimentos definidos da seguinte forma:

```
procedure P (PF1, ..., PFn); <bloco>
```

`P` é o identificador do procedimento, `Pi` os parâmetros formais, e `bloco` é o bloco de comandos que descreve o comportamento do procedimento. Note que aqui não há um valor como resultado porque esta é uma abstração de comandos, os quais têm como princípio os efeitos colaterais sobre o estado do programa.

Em C, os procedimentos e funções aparecem em uma forma unificada, todos são funções a princípio. Os procedimentos são denotados pelo tipo void como tipo do resultado da função:

```
void P (PF1, ..., PFn) <bloco>
```

Em SML, não existem procedimentos explicitamente, já que nas linguagens funcionais, funções recursivas são usadas para as soluções dos problemas. Contudo, tal recurso pode ser alcançado artificialmente através do uso de ref, variáveis de referência.

7.3 PARÂMETROS

Discutimos acima que as abstrações de processos são realizadas sobre expressões ou comandos. Para que possamos usar tais abstrações, precisamos fornecer alguns argumentos, os dados que serão usados na aplicação da abstração. A definição dos dados necessários à aplicação da abstração está embutida na definição da abstração pelos seus parâmetros. A expressividade das abstrações estão diretamente relacionadas com os seus parâmetros. Considere os exemplos abaixo.

Exemplo 7.4 – O cálculo da potência de um número real com um expoente inteiro quando estes números são valores fixados no programa:

```
float pot_restrita ()
 { /*assumimos n>0*
  float potencia;
  if (n = 1)
       {potencia = x ;}
  else {n = n - 1;
```

```
        potencia = x * pot_restrita();}
  return (potencia);
 }
 ...
 void main()
  {float x, pot10_2;
   int n;
   ...
   x = 10;
   n = 2;
   pot10_2 = pot_restrita();
   ...
```

A função definida acima depende de valores fixados no programa e o seu uso requer que o programador saiba quais variáveis são usadas internamente na função, não apenas o que ela deve computar. Tal restrição é obtida porque não foram definidos parâmetros para o uso da função, elementos que permitem fornecer dados para instanciar a função sem conhecer o seu comportamento interno. Ao passo que, se tivermos a redefinição dessa função em que os dados a serem considerados são passados como parâmetros, precisamos apenas fornecer os dados, em vez de também precisar conhecer as suas variáveis internas.

```
float pot (float x, int n)
 { /*assumimos n>0*
  float potencia;
  if (n = 1)
       {potencia = x ;}
  else {potencia = x * pot(x, n-1);}
  return (potencia);
 }
```

No uso desta função, faz-se necessário apenas fornecer um valor do tipo real para o primeiro parâmetro e um valor do tipo inteiro para o segundo parâmetro (por exemplo,

`pot(10.0, 2)`). Não há necessidade de conhecer os identificadores das variáveis internas. □

A abstração de comportamentos computacionais só podem ser obtidos pela parametrização dos mesmos. Chamamos de **parâmetros formais** os identificadores usados nas abstrações para denotar os argumentos, enquanto que a expressão associada a cada parâmetro formal no uso da abstração é chamada de **parâmetro atual**. No exemplo acima, `(float x, int n)` são os parâmetros formais da função `pot`, enquanto no uso da função `pot(10.0, 2)`, `(10.0, 2)` são os parâmetros atuais. Os valores dos parâmetros atuais das abstrações são chamados de **argumentos** e os tipos de valores permitidos em abstrações pode variar de linguagem para linguagem.

Na aplicação das abstrações, cada parâmetro formal deve ser associado a um argumento. Contudo, a relação dos parâmetros de uma abstração com o programa depende dos mecanismos de passagem de parâmetros como veremos a seguir.

Os parâmetros apresentados nesta seção, apenas sobre as abstrações de processos, são válidos também para as abstrações de tipos a serem apresentadas neste capítulo.

7.3.1 Mecanismos de Passagem de Parâmetros

Existem diferentes maneiras de associar os parâmetros formais de uma abstração com os parâmetros atuais usados na aplicação da função. Os mecanismos de passagem de parâmetros determinam como os parâmetros atuais e formais são associados computacionalmente.

Cada linguagem de programação determina suas formas de passagem de parâmetros com as respectivas variações de implementação. Encontramos nas linguagens os parâmetros de entradas, os parâmetros de saída (ou resultado), os quais podem ainda ser valores constantes, variáveis, etc com tipos e implementações diversas. Contudo, essas várias formas podem ser resumidas, sob o ponto de vista conceitual, em apenas duas: **mecanismo de cópia** e o **mecanismo de referência a valores**. Estes dois mecanismo são usados tanto para os parâmetros de entrada quanto para os parâmetros resultado.

7.3.1.1 Cópia de Valores O mecanismo de cópia de valores permite que os parâmetros da abstração sejam criados como variáveis locais, de forma que os argumentos passados para a abstração no momento de sua aplicação sejam copiados para estes parâmetros. No momento em que a abstração é aplicada, as expressões passadas como parâmetros atuais são avaliadas e copiadas para os parâmetros formais da abstração, os quais são variáveis locais da mesma. Dado que os valores são copiados, não há associação entre os parâmetros atuais do programa e os parâmetros formais da abstração. Quando a computação da abstração é finalizada, os parâmetros atuais do programa não sofrem as mudanças eventualmente efetuadas sobre os parâmetros formais da abstração.

A maioria das linguagens de programação admite formas de passagem de parâmetros como valores. Para os parâmetros de entrada, os valores das expressões são calculados e copiados para os parâmetros formais. Quanto aos parâmetros de resultado, estes são criados como variáveis locais e terão um valor associado apenas ao final da computação da função.

Na linguagem Pascal, por exemplo, os parâmetros resultado de funções são passados por valor, e os parâmetros de entrada também podem ser passados por valor, efetuando assim o mecanismo de cópia.

Exemplo 7.5 – Considere a definição da função fatorial e um trecho de programa que usa esta função:

```
program X
 var num, calculo: Integer;
 function fat (n:Integer): Integer;
  begin (*assumimos n>=0*)
   fat := 1;
   while (n > 1)
    begin
     fat := fat * n;
     n := n - 1
    end
  end
```

```
...
 begin
  num := 3;                      (1)
  calculo := 2 * fat(num) ;      (2)
  ...
```

No trecho acima do programa, a variável num recebe o valor 3, em (1), e é usada como parâmetro atual da função fat em (2). Note que dentro da função o parâmetro formal n sofre mudanças de valores. Como no caso acima é feita a passagem de parâmetro por valor, o mecanismo de cópia é usado e ao final do comando em (2) a variável num permanece com o seu valor inalterado (valor 3). Isso é possível porque uma cópia do seu valor foi realizada para a variável n local à função fat. □

Esse mesmo mecanismo é utilizado nas linguagens C e C++. O exemplo acima pode ser definido em C como segue.

Exemplo 7.6 – O exemplo acima pode ser definido em C como segue:

```
int fat (int n)
 {(*assumimos n>=0*)
  fatorial = 1;
  while (n > 1)
   {fatorial = fatorial * n;
    n = n - 1
   }
  return (fatorial);
 }
...
void main ()
{int num, calculo;
 num = 3;                       (1)
 calculo = 2 * fat(num) ;       (2)
 ...
```

Da mesma forma que no programa em Pascal, o valor da variável `num` permanece inalterada porque o mecanismo de cópia é usado. □

7.3.1.2 Referência a Valores A passagem de parâmetros por referência requer que os argumentos sejam referenciados em vez de copiados. Neste mecanismo, o parâmetro formal é vinculado diretamente ao parâmetro atual: e o resultado é que os parâmetros atual e formal compartilham as vinculações de memória e conteúdo.

Então, quando da aplicação de uma abstração, todas as consultas ou modificações de conteúdo efetuadas sobre o parâmetro formal dentro da abstração corresponde a uma consulta ou modificação, respectivamente, ao parâmetro atual do programa.

A maioria das linguagens modernas possui formas de denotar a passagem de parâmetros por referência. A linguagem Pascal, por exemplo, utiliza a palavra reservada `var` sobre os parâmetros da função para indicar que estes devem usar o mecanismo de passagem por referência.

Exemplo 7.7 – Considere uma nova definição, em Pascal, da função fatorial com passagem de parâmetro por referência:

```
program X
 var num, calculo: Integer;
 function fat (var n:Integer): Integer;
  begin (*assumimos n>=0*)
   fat := 1;
   while (n > 1)
    begin
     fat := fat * n;
     n := n -1
    end
  end
...
 begin
  num := 3;                    (1)
  calculo := 2 * fat(num) ;    (2)
```

```
...
```

No trecho acima do programa, a variável `num` recebe o valor 3, em (1), e é usada como parâmetro atual da função `fat` em (2). O parâmetro formal `n` é passado por referência (`var n:Integer`) e dentro da função ele sofre mudanças de valores. Como o mecanismo de passagem de parâmetro por referência é usado, o parâmetro formal `n` será associado diretamente às vinculações de memória e conteúdo de `num` quando `fat(num)` é aplicada em (2).

Então, todas as modificações efetuadas sobre `n` dentro da função são refletidas sobre a variável correspondente `num` no programa. Ao final do comando em (2), a variável `num` terá último valor associado à `n` dentro da função (valor 1). □

Nas linguagens C e C++, não há um modificador que denote explicitamente o mecanismo de passagem de parâmetro a ser usado, todos os parâmetros são passados por valor. Para se conseguir o efeito de passagem de parâmetro por referência nessas linguagens, deve-se usar um apontador (ou referência) para o argumento ao invés do valor. Dessa forma, o tipo do argumento deve ser modificado para apontador.

Exemplo 7.8 – Considere a função fatorial modificada para que o parâmetro formal seja compartilhado com o atual do programa:

```
int fat (int *n)                    (Note o ``*''!)
 {(*assumimos n>=0*)
  fatorial = 1;
  while (*n > 1)
   {fatorial = fatorial * *n;
    *n = *n - 1;
   }
  return (fatorial);
 }
...
void main ()
{int num, calculo;
 num = 3;                      (1)
```

```
calculo = 2 * fat(&num) ;     (2)
...
```

Note que o parâmetro formal da função é definido, agora, como um apontador para um valor inteiro (`int *n`). Isso acarreta obviamente uma modificação na aplicação da função, que deve ser agora o endereço de memória da variável `num`, denotado por `&num`. Assim, a variável `num` e o parâmetro `n` compartilham o mesmo espaço de memória resultando no efeito de passagem de parâmetro por referência como no exemplo acima em Pascal. □

Em C++ os parâmetros podem usar o modificador `const` o qual indica que o valor do parâmetro formal não deve ser modificado. Modula-2 tem uma passagem de parâmetros similar à Pascal e ambas também podem ter constantes como parâmetros. Na linguagem funcional SML, todos os parâmetros usam o mecanismo de referência, sejam eles valores simples ou funções.

7.3.2 Parametrização de Tipos

Reutilização é um dos princípios que corrobora com a produtividade, e as abstrações em linguagens de programação são formas de fornecer reutilização. Os parâmetros são essenciais para que possamos abstrair os dados, mas além desses podemos ainda ter algoritmos genéricos que podem ser aplicados a tipos diferentes.

As abstrações relacionadas a tipos referem-se a declarações que podem ter tipos genéricos. Discutimos no Capítulo 5 que polimorfismo aumenta a capacidade de expressão das linguagens, e a parametrização de tipos nas abstrações proporciona polimorfismo das mesmas. Esse recurso não é comum para a maioria das linguagens de programação.

Exemplo 7.9 – Considere o seguinte algoritmo, em Ada, de ordenação de vetores (com, no máximo, 100 elementos) de um tipo genérico:

```
generic
  type tipo_elem is private;
  type vetor     is array (1..100) of tipo_elem;
  procedure ord_generica(lista: in out vetor);
```

```
procedure ord_generica(lista: in out vetor) is
    ...
end ord_generica;
```

Aqui, `tipo_elem` é um tipo genérico usado na definição de `vetor`. A abstração `ord_generica` ordena um vetor do tipo `vetor`. Para usar essa abstração é necessário adotar (instanciar) um tipo existente para este declarado como genérico.

```
procedure ord_inteiros is new ord_generica(Integer);
```

Esta nova abstração instancia a anterior com o tipo `Integer` para o tipo genérico `tipo_elem`. □

Em Ada, existe uma forma especial de declaração que parametriza tipos abstratos. Estes serão vistos posteriormente quando encapsulamento for discutido. C++ também possui abstrações em declarações através dos `templates`.

7.3.3 Ordem de Avaliação

Vimos no Capítulo 6 que a ordem de avaliação de uma expressão influencia o resultado da mesma quando efeitos colaterais são permitidos em expressões. Da mesma forma, a ordem de avaliação dos parâmetros das abstrações pode influenciar os resultados obtidos nas suas aplicações.

Existem basicamente duas formas majoritárias de ordem de avaliação dos parâmetros: avaliação sob demanda e avaliação de todos os argumentos **a priori**. Na estratégia de avaliação dos argumentos **a priori**, quando uma abstração é usada, todos os argumentos são avaliados por completo para então iniciar o processamento da abstração. Por outro lado, na avaliação sob demanda, quando uma abstração é aplicada, o seu processamento é iniciado e os argumentos são avaliados à medida que são usados no processamento. A diferença dessas estratégias podem ser observadas no exemplo a seguir.

Exemplo 7.10 – Considere uma função que tem como resultado o segundo ou terceiro argumento, dependendo do valor do primeiro:

```
fun escolha (b1: bool, x:int, y:int) =
```

```
if b1 then x else y ;
```

Suponha que tenhamos a seguinte aplicação desta:

```
val a = 0;
val b = 10;
escolha (b>50, b/a, a/b);
```

Considerando as estratégias de avaliação:

- Se todos os parâmetros forem avaliados **a priori** na aplicação acima, a mesma acusará um **erro** quando a expressão `b/a` for avaliada (divisão por zero) e a aplicação não será processada.

- Se a avaliação **sob demanda** for usada, os parâmetros atuais serão avaliados apenas no momento em que são usados. No processamento da função `escolha`, o argumento `b>50` será avaliado primeiro por causa da expressão `if`, e como esta é avaliada com o valor `false`, a expressão `a/b` será avaliada e a aplicação `escolha (b>50, b/a, a/b)` terá como resultado o valor 0 (zero). O argumento `b/a` não será processado nesta estratégia.

Note que a aplicação da função conduziu a resultados distintos com as diferentes estratégias de avaliação dos parâmetros. □

No exemplo acima, o uso das estratégias conduziu a resultados distintos porque a função tem um comportamento não estrito, no qual o resultado depende de uma avaliação parcial dos parâmetros (ou o segundo, ou o terceiro parâmetro precisa ser avaliado). Para processamentos que dependem da avaliação de todos os parâmetros (comportamento estrito), os resultados serão idênticos.

A maioria das linguagens de programação utiliza a estratégia de avaliação a priori, apenas algumas linguagens funcionais, tais como Miranda, Lazy ML e Haskell, usam a estratégia sob demanda. Este tipo de avaliação será discutido em mais detalhes no Capítulo **??**.

7.4 ABSTRAÇÃO DE TIPOS

As abstrações de processos estão presentes em projetos de linguagens desde as mais antigas até as mais modernas. À medida que aumentou a complexidade dos problemas a serem resolvidos, veio a necessidade de não apenas ter abstrações de processos como subunidades de programas, mas também conjuntos de abstrações de processos. Com isso surgiu o conceito de módulos (em Modula-2, por exemplo), os quais podem conter os vários tipos de abstrações: de dados e de processos (tipos, variáveis, constantes, funções, etc). Os módulos (ou pacotes) passam a ser considerados como unidades de programas e também unidades de compilação, o que faz que, sob o ponto de vista prático, sejam de grande utilidade.

Junto com a idéia de agrupar as abstrações vem o conceito de **encapsulamento**. Dizemos que um módulo encapsula todas as definições existentes nele. Além disso, em vários problemas práticos, alguns elementos definidos dentro de um módulo devem ser usados apenas como auxílio ao funcionamento de outras abstrações, e não devem ser conhecidos pelos usuários do módulo. Para diferenciar as abstrações que devem ser usadas apenas dentro dos módulos das que devem ser usadas externamente surgiu o conceito de **ocultação de informação**. Assim, os módulos devem agrupar abstrações, e além disso, ter formas de distinguir as informções visíveis das não-visíveis externamente.

Linguagens como Modula-2 e Ada, por exemplo, possuem formas de denotar os módulos (ou pacotes), e de definir as interfaces externas de cada um deles. Essas linguagens definem os módulos (ou pacotes) em duas partes distintas: a interface, a qual contém apenas os cabeçalhos das abstrações que podem ser usadas externamente; e o corpo do módulo (ou pacote) contendo a implementação de todas as suas abstrações. O usuário do módulo deverá ter acesso apenas à interface do módulo, a qual define os elementos necessários ao uso, o corpo do módulo é ocultado do usuário.

Essa idéia de módulos, juntamente com ocultação de informação, faz com que tenhamos unidades maiores com interface de uso bem definida, pois as informações internas e externas são diferenciadas. Apesar de nessas linguagens podermos ter módulos como conjuntos de quaisquer abstrações, podemos especializar tal conjunto

de forma a conter apenas dados abstratos e formas de transformação desses dados abstratos. Assim, módulos junto com formas de ocultação de informação podem ser usados para definir **tipos abstratos**.

7.4.1 Tipos Abstratos

Vimos no Capítulo 3 que existem os tipos predefinidos nas linguagens de programação. O tipo inteiro, por exemplo, denota um conjunto ordenado de valores, e além disso, temos operações predefinidas para tratar (ou manipular) esses valores: as operações aritméticas e as de comparação que estão relacionadas à ordem dos valores. Um tipo de dados abstrato é denotado por dados elementares do tipo e um conjunto de operações para transformação dos dados.

Quando definimos novos tipos nos programas por meio apenas declaração de um tipo a partir dos pré-existentes, não definimos as formas como esses dados devem ser tratados. Quando uma estrutura de pilha de inteiros é definida por um vetor, podemos realizar quaisquer operações permitidas a vetores sobre a dada estrutura. Com isso, poderíamos, por exemplo, acidentalmente retirar um elemento do meio da pilha, mesmo sabendo que tal operação é inválida para tal estrutura (da qual só deveríamos retirar o elemento do topo). Como a declaração de novos tipos não coibe operações equivocadas sobre o tipo e nem a confusão deste com outros tipos de declarações semelhantes, houve a necessidade de se criar mecanismos para construir os tipos abstratos.

Os tipos abstratos definem os dados abstratos e todas as operações permitidas sobre eles, de forma que quaisquer outras operações sobre elementos definidos como daquele tipo são rejeitadas. Essa definição do tipo e todas as operações associadas devem estar em uma unidade de programação, e tal implementação não deverá ser acessível pelo usuário de objetos desse tipo, para quem é permitido apenas o uso do tipo e as respectivas operações. Linguagens como Ada, SML e as linguagens orientadas a objetos provêem mecanismos para a criação de tipos abstratos pelos usuários.

Um exemplo de tipos abstratos pode ser visto com o uso da estrutura de pilhas. As operações definidas para uma estrutura de pilha denotam o comportamento da mesma:

cria(pilha)	cria uma pilha vazia
vazia(pilha)	testa se uma pilha está vazia
insere(elem, pilha)	insere o elemento no topo da pilha
retira(pilha)	retira o elemento do topo da pilha
topo(pilha)	consulta o elemento do topo da pilha e cria uma cópia

Note que para termos um elemento inicial de pilha, precisamos de uma representação para a pilha vazia. A partir daí, podemos construir e destruir pilhas pelas operações base, além de colocar ou retirar elementos da pilha. O usuário de pilhas precisa apenas conhecer tais operações e não como elas são implementadas e representadas.

Quando as implementações não são acessíveis (informação oculta), evita-se qualquer operação equivocada sobre o tipo pilha. Por outro lado, para cada objeto do tipo pilha, todas as operações base definidas para o tipo estão disponíveis aos usuários do objeto (encapsulamento).

Na linguagem Ada, por exemplo, o tipo abstrato pilha pode ser definido através de pacotes (módulos discutido acima), associado ao mecanismo de ocultação de informação (cláusula `private`). O pacote deve então conter uma interface, a qual define todos os elementos que devem ser visíveis, e os que são internos (ocultos), e mais a parte de implementação (`body`).

Exemplo 7.11 – Um pacote que define de forma esquemática o tipo abstrato pilha na linguagem é como segue:

```
package pilha_int is
    type Pilha is limited private;
    procedure cria   (p: out Pilha);
    function  vazia  (p: in Pilha);
    procedure insere (elem: in Integer;
                      p: in out Pilha);
    procedure retira (p: in out Pilha);
```

```
    function  topo   (p: i Pilha)
                      return Integer;

private
    type NoPilha;
    type Pilha   is access NoPilha;
    type NoPilha is record
                      elemento  : Integer;
                      restopilha: Pilha;
                    end record;
end pilha_int;

package body pilha_int is
    procedure cria   (p: out Pilha);
    begin
      p := null;
    end;

    function  vazia  (p: in Pilha);
    ...
    procedure insere (elem: in Integer;
                      p: in out Pilha);
    ...
    procedure retira (p: in out Pilha);
    ...
    function  topo   (p: i Pilha)
                      return Integer;
    ...
end pilha_int;
```

O pacote define, na parte de interface, todos os elementos que são visíveis pelo usuário da pilha e mais os elementos que não são visíveis (entre `private` e `end` da primeira

parte). Note que a `Pilha` é declarada como `limited private`. Isso significa que este novo tipo só pode ser manipulado pelas operações definidas na interface[1]. A implementação segue em um pacote separado (`body`) e deve conter pelo menos todas as operações de manipulação do tipo contidas na interface. Algumas operações auxiliares poderiam ser definidas na parte de implementação, mas seriam acessíveis apenas localmente e não pelo usuário do tipo.

Para usar o tipo abstrato acima, deve ser incluída no programa a cláusula que indica os pacotes a serem usados e a partir daí o novo tipo pode ser usado como qualquer outro predefinido na linguagem. Como não há acesso à representação do tipo (foi declarada como `private`), o usuário tem acesso apenas às operações sobre os elementos do tipo definido (da mesma forma que não temos acesso à representação dos números inteiros, apenas os usamos mediante as operações predefinidas).

```
with  pilha_int;      {importa o pacote a ser usado}
use   pilha_int;
elem_int: Integer;
p_int   : Pilha;      {declara p_int do tipo pilha}
...
cria(p_int);          {aplica operações predefinidas}
insere(1,p_int);
...
if  not vazia(p_int)  then ...
...
```

□

A linguagem Modula-2 permite que tipos abstratos sejam definidos de forma semelhante à de Ada por seus módulos. Uma diferença fundamental é que em Modula-2 os tipos cuja representação estão ocultas nos módulos devem ser apontadores; os tipos abstratos são sempre representados por apontadores. A linguagem SML provê

[1]Com a cláusula `limited` as comparações e atribuições, inclusive, de elementos devem estar definidas no próprio pacote.

mecanismos de definição de tipos abstratos de forma explícita pela palavra-reservada `abstype`.

Exemplo 7.12 – O tipo abstrato pilha de inteiros pode ser definido em SML como segue:

```
abstype  pilha_int =   p_vazia
                     | pilha of (int * pilha_int)
with
    fun cria   = p_vazia
    fun vazia  (p: pilha_int) = (p = p_vazia)
    fun insere (elem: int, p: pilha_int) = pilha(elem,p)
    fun retira (p: pilha_int) = ...
    fun topo   (p: pilha_int) = ...
end
```

A pilha de inteiros é declarada e sua representação é dada por `p_vazia`, representa a pilha vazia, ou um elemento inteiro seguido de uma pilha de inteiros. As operações são então definidas para este tipo, e elementos definido com ele só podem ser tratados via estas operações predefinidas.

```
fun pilhadois (x:int,y:int) = insere(x,insere(y,p_vazia));
>val pilhadois = fn: int * int -> pilha_int
```

Esta função cria uma pilha com dois elementos. Primeiro é colocado o elemento `y`, e depois o elemento `x`, o qual é o elemento do topo. □

As linguagens orientadas a objetos têm como princípio a criação de tipos abstratos por classes, as quais serão vistas a seguir.

7.4.2 Classes

O conceito de **objetos** em linguagens de programação consiste de uma variável oculta juntamente com um conjunto de operações sobre esta variável. O objeto então é responsável pelo seu próprio estado e exporta as operações que podem ser aplicadas sobre os seus dados. As **classes**, elementos de definição das linguagens orientadas

a objetos, podem ser vistas como tipos abstratos, uma vez que elas embutem uma representação de dados abstratos e as operações (métodos) que podem manipular tais dados. As classes são os elementos de definição, enquanto os objetos são os elementos de uso das classes (instâncias).

As linguagens orientadas a objetos, tais como Smalltalk, C++ e Java, dão suporte a essa forma de definição de tipos abstratos.

Exemplo 7.13 – O tipo abstrato pilha de inteiros pode ser definido como uma classe na linguagem C++ (esquemático):

```
class pilha {
  private:            (elementos ocultos)
    int *ptr_pilha;
    int elemento;

  public:             (elementos disponíveis ao usuário)
    pilha(){...}      (cria pilha)
    ~pilha(){...}     (destrói pilha)
    void vazia(){...}
    void insere(int elem){...}
    void retira(){...}
    int topo(){...}
}
```

A classe tem a sua representação de dados ocultada (`private`) e um conjunto de operações que estão disponíveis aos usuários. Aqui, a criação e destruição da `pilha` são definidas por `pilha` e `~ pilha`. As outras operações são definidas como nos exemplos anteriores, exceto que os dados abstratos são usados dentro dos métodos sem que tenham sido passados como parâmetros porque cada método é explicitamente aplicado sobre o objeto.

Para o uso do tipo pilha, considere o seguinte trecho de programa:

```
void main(){
  int elem_int;
```

```
    pilha p_int;        (1) (cria uma instância de classe pilha)
    ...
    p_int.insere(9);    (2)
    ...
    if  (p_int.vazia) ...
    ...
}
```

`p_int` é um objeto do tipo `pilha` (em (1)). A partir da sua declaração, cada operação é aplicada ao objeto declarado (`p_int.insere(9)`, `p_int.vazia`). □

Cada objeto declarado tem consigo todas as operações que podem ser efetuadas sobre ele. Dessa forma, cada objeto tem as suas próprias transformações, enquanto que nos tipos abstratos anteriores temos uma única definição para todas as instâncias do tipo. Além disso, os objetos aparecem de forma explícita como pré-fixo de cada uma das suas operações, diferindo também da forma de uso dos tipos predefinidos nas linguagens.

Devemos ressaltar que na maioria das linguagens a associação de tipos abstratos com classes não é válida de forma geral. Alguns problemas surgem quando tentamos associar subtipos com subclasses. As restrições para tal associação serão discutidas no Capítulo 10.

7.4.3 Tipos Abstratos Genéricos

A definição de tipos abstratos em linguagens aumenta sua capacidade de expressão, facilidade de escrita e leitura dos programas. A combinação de tipos abstratos e parametrização de tipos aumenta esta capacidade para que possamos construir tipos abstratos genéricos, os quais podem ser instanciados para tipos abstratos. Em Ada, por exemplo, podemos definir esses tipos abstratos genéricos.

Usando ainda o exemplo da estrutura de pilhas, podemos notar que os mecanismo para construir e manipular as pilhas independem do tipo de dados dos elementos da pilha. Assim podemos criar pilhas genéricas e instanciá-las para os tipos desejados.

Exemplo 7.14 – Um pacote que define de forma esquemática o tipo abstrato genérico pilha na linguagem Ada é como segue [64]:

```
generic
    type tipo_elem is private;
package pilha_gen is
    type Pilha is limited private;
    procedure cria   (p: out Pilha);
    function  vazia  (p: in Pilha);
    procedure insere (elem: in tipo_elem;
                      p: in out Pilha);
    procedure retira (p: in out Pilha);
    function  topo   (p: i Pilha)
                      return tipo_elem;

private
    type NoPilha;
    type Pilha   is access NoPilha;
    type NoPilha is record
                      elemento  : tipo_elem;
                      restopilha: Pilha;
                    end record;
end pilha_gen;

package body pilha_gen is
    procedure cria   (p: out Pilha);
    begin
      p := null;
    end;
```

```
    function  vazia  (p: in Pilha);
    ...
    procedure insere (elem: in tipo_elem;
                      p: in out Pilha);
    ...
    procedure retira (p: in out Pilha);
    ...
    function  topo   (p: i Pilha)
                      return tipo_elem;
    ...
end pilha_gen;
```

o tipo do elemento da pilha é genérico (`tipo_elem`) e pode ser instanciado com um tipo existente na linguagem

```
package pilha_int   is new  pilha_gen(Integer);
package pilha_carac is new  pilha_gen(Character);
```

Aqui, os tipos abstatos pilhas de inteiros e de caracteres são criados como instâncias do tipo `pilha_gen`. □

A linguagem C++ tem um mecanismo semelhante de definição de classes genéricas por `templates`. Como SML é essencialmente polimórfica, os seus tipos abstratos também podem ser generalizados.

7.5 LEITURA RECOMENDADA

A idéia de encapsulamento foi inicialmente definida por D. Parnas em [45]. Uma classificação sobre abstrações em linguagens de programação com o objetivo de reutilização pode ser vista em [67].

O poder de expressão conseguido com parametrização em linguagens de programação é discutido por Goguen em [30]. Leitores interessados em uma definição mais acurada sobre tipos e abstração de dados vejam [21] e [61].

7.6 EXERCÍCIOS

1. Em projetos de linguagens mais antigas como Fortran, todos os parâmetros eram passados por referência. Discuta as vantagens e desvantagens.

2. Na linguagem C, todos os *arrays* são passados por referência. Pesquise os argumentos dos projetistas dessa linguagem para tal decisão de projeto. Existem argumentos contra?

3. A maioria das linguagens permite apenas que valores simples sejam resultados de funções. Algumas outras linguagens, como SML, admitem que o valor resultado seja inclusive uma função. Dê um exemplo no qual seria conveniente um valor resultado como um valor composto.

4. Projete o tipo abstrato pilha em uma linguagem como C ou Pascal, a estrutura de dados e as operações necessárias, todos em um único programa.

5. Discuta o que se faz necessário ao programador usuário do tipo pilha definido na questão anterior para o uso adequado na linguagem de um tipo abstrato.

6. Qual a diferença de se ter em uma linguagem o recurso de definir um novo apenas tipo por declarações de tipos preddefinidos, de outra que admite a definição de tipos abstratos?

7. Complemente a definição esquemática do tipo pilha mostrado neste capítulo.

8. Defina o tipo abstrato lista ligada em uma das linguagens que dá suporte a tipos abstratos, tais como Modula-2, Ada, C++ ou Java. Mostre a relação, semelhanças e diferenças, entre o tipo pilha definido na questão anterior e o tipo lista ligada aqui definido.