1.5.3.3. Tipos Abstratos de Dados

Linguagens de programação modernas fornecem alguns tipos de dados elementares e várias maneiras de definir novos tipos a partir deles. A maneira mais simples consiste na enumeração (seção 1.5.3.2). Já os tipos agregados permitem a composição de estruturas de dados complexas a partir dos tipos elementares da linguagem. As LP mais novas também permitem que a construção de dados agregados seja nomeada como um novo tipo. Através da atribuição de um nome à esta estrutura, é possível declarar quantas variáveis desse tipo forem necessárias através de declarações simples. Por exemplo, depois da declaração em C que introduz um novo tipo

```
typedef struct {
        float real_part, imaginary_part;
} complex;
```

qualquer número de variáveis pode ser definido:

```
complex a, b, c, ...;
```

A habilidade de atribuir um nome de tipo a uma estrutura de dado definida pelo usuário é somente o primeiro passo em direção ao suporte aos tipos abstratos de dados. As vantagens da introdução de novos tipos em uma linguagem são classificação e proteção. Assim é permitido que os dados sejam organizados como uma coleção de grupos distintos, e protegidos contra manipulações indesejáveis através de duas propriedades:

- Especificação exata de quais operações são permitidas com o novo tipo;
- Esconder a representação concreta do novo tipo das unidades que usam este tipo.

Destas duas propriedades, somente a primeira é alcançada através de um tipo definido pelo usuário. Para fornecer proteção, um novo tipo deve ser definido para satisfazer as propriedades. Mais precisamente, é necessário um construtor para definir tipos abstratos de dados. Um tipo abstrato de dado é um novo tipo para o qual é possível definir as operações a serem usadas para manipulação de instâncias, enquanto a estrutura de dado que implementa o tipo é escondida dos usuários. Exemplos de linguagens que possuem tipos abstratos de dados são: *C++*,*Eiffel*,*Ada*,*CLU*,*Pascal Concorrente*,*Módula-2*,*Smalltalk*,*MESA*,*Euclid*,*Simula 67*,etc.

O conceito de tipos abstratos de dados vem do princípio mais geral de esconder informação: um trecho de programa implementando um tipo abstrato de dados é um exemplo de módulo que esconde informação. Tipos abstratos de dados escondem detalhes de representação e direcionam o acesso aos objetos abstratos por meio de procedimentos. A representação está protegida contra qualquer tentativa de manipulá-la diretamente. Uma troca da implementação de um tipo abstrato de dado fica confinada ao trecho de programa que descreve a implementação e não afeta o resto do programa. Pode-se dizer, então, que os tipos abstratos de dados encapsulam tanto a representação interna dos tipos, como as operações concretas em unidades especiais de programação [GUE 87, GHE 97].

Tipos abstratos de dados podem ser definidos em ***C++*** através da construção *class*. Uma classe reúne a definição de um novo tipo e fornece explicitamente as operações que podem ser invocadas para o correto uso das instâncias do tipo. O exemplo abaixo mostra a definição de um tipo abstrato de dado para o ponto.

```
class point {
  public:
       point(int a, int b) { x=a; x=b; } //Inicializa as coordenadas de um novo ponto
      void moveX(int a) { x += a; } // Move o ponto horizontalmente
      void moveY(int b) { y += b; } // Move o ponto verticalmente
      void reset() { x=0; y=0; } // Move o ponto para a origem
  private:
     int x, y;
};
```

Uma classe pode ser vista como uma extensão das estruturas (ou registros), onde os campos podem ter dados e rotinas. A diferença é que somente campos declarados como *public* são acessíveis de fora da classe.

Campos não *public* são escondidos dos usuários e podem ser declarados como *private* explicitamente, que é o *default* caso não sejam declarados nem como *public* nem como *private*. No exemplo, a classe encapsula a definição da estrutura de dados para representar os pontos (são os dois números inteiros "x" e "y") e as operações fornecidas para manipular os pontos. As operações são descritas através de rotinas, chamadas funções membro. A estrutura de dados que define o ponto não é acessada diretamente, os pontos são manipulados através das operações definidas como rotinas públicas, como mostra o próximo trecho de programa.

```
:
point p1(1,3); // instancia p1 e inicializa seus valores
point p2(55,0); // instancia p2 e inicializa seus valores
point* p3= new point(0,0); // p3 aponta para a origem
p1.moveX(3); // move p1 horizontalmente
p2.moveY(99); // move p2 verticalmente
p1.reset(); // posiciona p1 na origem
:
```

No trecho de programa anterior, pode-se observar também que as operações são invocadas através de ".", isto é, escrevendo-se "nome_do_objeto.nome_da_função_pública". A única exceção é a invocação de construtores e destrutores. O primeiro é usado para fazer a inicialização do tipo definido pelo usuário, e possui o mesmo nome do tipo definido. Ele é invocado automaticamente quando um objeto da classe é alocado. O destrutor é reconhecido por ter o nome da classe precedido por "~", e seu objetivo é "limpar" as variáveis depois da última utilização do objeto.

Eiffel também fornece uma construção *class* para implementar tipos abstratos de dados. Na seqüência, o tipo abstrato de dado "ponto" é descrito em *Eiffel*.

```
class POINT export
    moveX, moveY, reset
creation
    make_point
feature
    x, y: INTEGER;
    moveX(a: INTEGER) is
            -- move o ponto horizontalmente
    do
        x := x + a
    end; --moveX
    moveY(b: INTEGER) is
        --move o ponto verticalmente
    do
        y := y + b
    end; --moveY
    reset is
        -- move o ponto para a origem
    do
        x := 0;
        y := 0
    end; – reset
    make_point(a, b: INTEGER) is
        -- determina a coordenada inicial do ponto
    do
        x := a;
        y := b
    end – make_point
end; -- POINT
```

Em outra classe, pode-se declarar referências para objetos do tipo "*POINT*" da seguinte forma:
*p1, p2: POINT**;***
Objetos podem ser criados e amarrados para tais referências usando construtores, que são especificados na criação da classe. Os objetos criados podem então ser manipulados de acordo com o tipo de operação, como mostra o próximo exemplo (a chamada ao construtor é sempre precedida por "*!!*").

```
!!p1.make_point(4,7);
!!p2.make_point(55,0);
p1.moveX(3);
p2.moveY(99);
p1.reset();
```

Em *C++* instâncias de um tipo abstrato de dado podem ser associadas com variáveis automáticas ou ser dinamicamente alocadas e referenciadas por ponteiros. Em *Eiffel*, todos os objetos são implicitamente alocados no *heap* e são acessíveis através de ponteiros. No exemplo, "*p1*" e "*p2*" são referências para "*POINT*", que são alocadas e inicializadas através da invocação da operação de criação. Mais tarde, *Eiffel* introduziu a possibilidade de declarar uma classe como expandida, o que significa que os possíveis valores de tempo de execução não são uma referência para o objeto, mas são o próprio objeto. Declarando "*POINT*" como expandida, "*p1*" e "*p2*" poderão considerar um objeto "*POINT*" como um valor, e a instrução de criação pode tornar-se desnecessária. Entretanto, classes expandidas raramente são utilizadas em *Eiffel*. A função "*make_point*" é análoga ao construtor *C++*, mas deve ser chamada explicitamente para criar o objeto [GHE 97].

Em ***Ada*** o mecanismo de encapsulamento é o pacote (ou *sharing*). Uma unidade de programa tem uma estrutura aninhada típica, que é alcançada através de declarações: uma declaração de um subprograma ou um pacote pode conter uma declaração de variáveis, procedimentos e pacotes (ou *package*) locais. O aninhamento também pode ser obtido pela definição de novos blocos em seqüências de instruções. Pacotes podem ser usados para várias aplicações, desde a declaração de um conjunto de entidades comuns (variáveis, constantes e tipos) até agrupar um conjunto de subprogramas relacionados ou descrever tipos abstratos de dados. Um exemplo do primeiro caso é o seguinte:

```
package NUMEROS_COMPLEXOS is
   type COMPLEXO is
      record
         RE: INTEGER;
         IM: INTEGER;
      end record;
   TABELA: array(1..500) of COMPLEXO;
end NUMEROS_COMPLEXOS
```

Esta declaração é processada como se fosse declaração das suas variáveis e tipos internos; portanto, tais variáveis e tipos têm o mesmo escopo e tempo de vida que as variáveis e tipos declarados na parte declarativa, onde aparece a declaração do pacote "*NUMEROS_COMPLEXOS*". Os nomes declarados no pacote podem ser usados dentro do escopo do mesmo por meio da notação de ponto, como em:
NUMEROS_COMPLEXOS.TABELA (K)

Quando se agrupa um conjunto de programas relacionados ou se descreve um tipo abstrato de dado, freqüentemente é necessário esconder algumas entidades locais dentro do pacote. No caso do conjunto de programas relacionados, as entidades locais ocultas poderiam ser variáveis e/ou procedimentos locais; já no caso de tipos abstratos de dados elas incluirão a representação concreta e talvez algumas variáveis e procedimentos auxiliares. A estrutura geral do pacote se compõe de duas partes: a especificação do pacote e o corpo do pacote. A

especificação contém exatamente a informação que é exportada pelo módulo, enquanto que o corpo do pacote contém todos os detalhes ocultos da implementação e uma seção de inicialização que é executada quando da ativação da unidade que contém a declaração do pacote.

Para exemplificar o próximo pacote define o tipo abstrato de dados número complexo. Neste exemplo, o tipo complexo exportado pelo módulo é *private*, isto é, os detalhes da representação incluídos dentro da porção "*private...end NÚMEROS_COMPLEXOS*" da especificação do pacote não são visíveis fora do pacote. Variáveis de tipo "*COMPLEXO*" só podem ser manipuladas por meio dos subprogramas "*INICIALIZA*" e "*SOMA*" exportados pelo pacote. As operações pré-definidas de atribuição e teste de igualdade/desigualdade também são permitidas. A criação é automaticamente efetuada quando as unidades que declaram variáveis do tipo "*COMPLEXO*" são ativadas. Porém o pacote "*NUMEROS_COMPLEXOS*" contém um procedimento explícito para inicialização. Os parâmetros dos procedimentos são especificados como "*in*" ou "*out*". A especificação "*in*" denota um parâmetro de entrada não modificável, e "*out*" denota um parâmetro de saída cujo valor é dado pelo procedimento.

```
package NÚMEROS_COMPLEXOS is
    type COMPLEXO is private;
    procedure INICIALIZA (A, B: in REAL; X: out COMPLEXO);
    function SOMA (A, B: in COMPLEXO) return COMPLEXO;
private
    type COMPLEXO is
        record R, I: REAL;
        end record;
end NÚMEROS_COMPLEXOS
package body NÚMEROS_COMPLEXOS is
    procedure INICIALIZA (A, B: in REAL; X: out COMPLEXO) is
    begin X.R := A;
            X.I := B;
    end INICIALIZA;
    function SOMA (A, B: in COMPLEXO) return COMPLEXO is
    TEMP: COMPLEXO;
    begin TEMP.R := A.R + B.R;
            TEMP.I := A.I + B.I;
            return TEMP;
    end SOMA
end NÚMEROS_COMPLEXOS;
```

1.5.4. Controle de Fluxo de Execução

Uma LP deve fazer mais do que simplesmente especificar as ações que um computador pode tomar. Ela também deve especificar a ordem em que estas ações podem ocorrer. Devido a natureza seqüencial da execução da linguagem de máquina, virtualmente todas as linguagens imperativas seguem o mesmo padrão adotando a execução seqüencial dos comandos. Isto significa que depois de um comando ter completado sua execução, o *default* para escolha do próximo comando a ser executado é o próximo comando "físico" do programa. Um dos benefícios óbvios da LP é a sua habilidade de modificar a ordem de execução dos comandos através da apresentação de alternativas ao modo seqüencial. As facilidades de uma linguagem que permitem isso são chamadas de estruturas de controle [DER 90].

Nesta seção é feita uma análise de como os processamentos são estruturados em uma LP em termos de controle de fluxo entre os diferentes componentes de um programa, que é feito através de estruturas de controle. Tais mecanismos descrevem a ordem na qual instruções ou grupos de instruções devem ser executados, facilitando assim o entendimento da lógica utilizada pelo programador e, principalmente, permitindo uma maior legibilidade e facilidade de manutenção dos programas. As estruturas de controle podem ser classificadas em:

- Estruturas de controle a nível de instrução, usadas para ordenar a ativação de instruções individuais.
- Estruturas de controle a nível de unidades de programação, empregadas para ordenar a ativação de unidades de programa.

Antes de descrever as estruturas de controle de acordo com esta classificação [GHE 87], o que é feito nas próximas seções, deve-se considerar os elementos constituintes de qualquer programa: expressões (que possuem um papel fundamental em todas as LP, incluindo as lógicas e funcionais) e comandos (que são típicos das linguagens convencionais baseadas em instruções).

Expressões definem como um valor pode ser obtido através da combinação de outros valores com operadores. Operadores que fazem parte de uma expressão, denotando as suas funções matemáticas, são caracterizados pela sua aridade (número de operandos requerido, tal como unário – aplicado somente a um operando, e binário – aplicado a dois operandos). Considerando-se a notação do operador, pode-se distinguir entre infixada, pré-fixada e pós-fixada. A primeira é a notação mais comum, onde o operador é escrito entre dois operandos (por exemplo: x + y). Na pré-fixada o operador aparece antes e os operadores depois (por exemplo: + x y). E finalmente na pós-fixada os operadores aparecem antes e são seguidos pelos operandos correspondentes (por exemplo: x y +).

Para ilustrar, em *C*, os operadores unários de incremento e decremento podem ser escritos nas notações pré e pós-fixada. Porém, a semântica é diferente, isto é, eles denotam dois operadores distintos. Considerando "++k" e "k++", ambas as expressões consistem em incrementar "k" em uma unidade. Porém, em "++k" primeiro o valor armazenado de "k" é incrementado e depois o novo valor de "k" é fornecido como o valor da expressão. Em "k++" o valor da expressão é o valor de "k" antes do incremento.

Notação infixada é a mais comum para operadores binários, desde que esta permite que os programas sejam escritos como expressões matemáticas convencionais. Apesar do programador poder usar parênteses para agrupar explicitamente sub-expressões que devem ser avaliadas primeiro, as LP tornam-se mais complexas através da introdução das suas próprias convenções para associação e precedência de operadores. Na realidade, isto é feito para facilitar a tarefa do programador, reduzindo redundâncias.

LP funcionais são fortemente baseadas em expressões. Em tais linguagens, o programa em si é uma expressão, definida através de uma função aplicada a operadores. Linguagens convencionais, por outro lado, fazem os valores das expressões visíveis como uma modificação do estado de processamento através da atribuição de expressões a variáveis. Um comando de atribuição, tal como "*x := y + z*" em Pascal, muda o estado associando um novo valor a "x", resultado de "y+z". O resultado da expressão é então armazenado na localização de memória usando o valor de "x". Desde que as atribuições mudam o estado de processamento, o comando que executa depois opera em um novo estado. Freqüentemente o novo comando a ser executado é o que segue textualmente aquele que acabou de ter sua execução completada, gerando assim, uma seqüência de comandos [GHE 97].

1.5.4.1 Estruturas de Controle: Nível de Instrução

Estruturas de controle neste nível contribuem sensivelmente para a legibilidade e manutenção de programas. Tais estruturas podem ser definidas em três categorias: seqüência, seleção e repetição. A **seqüência** é o mecanismo mais simples para mostrar o fluxo de um programa, sendo utilizada para indicar que um comando "B" ocorre após um comando "A". Por exemplo: "A; B", onde o ";" indica o operador de seqüência. Algumas LP que possuem um formato orientado a linhas, como o *Fortran*, usam o final de linha para separar as instruções e impor um mecanismo de seqüência entre elas. Várias linguagens permitem agrupar comandos em seqüência para formar um único comando composto, utilizando sempre algum mecanismo (explícito ou implícito) específico. Este mecanismo corresponde, por exemplo, a "*Begin ... End*" em *Pascal* e a "*{ ... }*" em *C* [SIL 88].

Estruturas de controle de seqüência implícita são aquelas definidas através da linguagem para serem seguidas, a menos que o programador a altere através de uma estrutura de controle explícita. Por exemplo, a maioria das linguagens definem a seqüência física dos comandos em um programa como o controle da seqüência na qual os comandos são executados. Estruturas de controle de seqüência explícita são aquelas que o programador

pode opcionalmente usar para modificar a seqüência implícita de operações, como por exemplo o uso de parênteses em expressões [PRA 75].

A categoria de **seleção** compreende os mecanismos que possibilitam especificar que uma escolha deve ser feita entre um certo número de alternativas. Assim, estruturas de controle condicionais determinam o próximo bloco de comandos a ser executado com base no resultado de um teste ou de uma seqüência de testes. A partir de agora serão descritas quatro formas de tais comandos.

A forma mais simples de uma estrutura condicional executa um único teste cujo resultado é usado para determinar quando ou não um bloco específico de programas deve ser executado. As duas partes destas estrutura são a expressão boolean que define a condição, e o bloco de comandos que será executado se a avaliação da expressão for verdadeira. A forma típica desta condição é:

if <expressão boolean> then <bloco de comandos>

A extensão desta condição simples do comando *if* é feita na maioria da linguagens através de uma estrutura de duas alternativas, levando a seguinte forma:

if <expressão boolean> then <bloco de comandos>
else <bloco de comandos>

A adição do *else* possui a desvantagem de levar a um problema de ambigüidade no caso de comandos condicionais aninhados. Por exemplo, considerando o seguinte trecho de programa em *Pascal*:

```
if x>0 then
   if x<10 then
         x := x + 1
   else
        x := x – 1;
```

O *else* neste comando pode pertencer tanto ao primeiro *if* como ao segundo. A indentação sugere que o *else* pertence ao segundo *if*, mas indentação e outras convenções de formatações são irrelevantes na interpretação da semântica. Uma solução para este problema consiste em associar marcadores de bloco que no caso do exemplo em *Pascal* são *begin* e *end*.

Duas formas da estrutura condicional de múltiplos níveis são encontradas nas LP. A primeira, e mais genérica, permite que qualquer seqüência de expressões boolean seja testada, com a primeira expressão verdadeira especificando o bloco de comandos escolhido para execução. A segunda forma, fornecida por um comando *case* ou *select*, avalia a expressão e determina o bloco a ser executado de acordo com o valor obtido. Desta maneira, é feita uma seleção entre várias alternativas mutuamente exclusivas [DER 90]. Para ilustrar a utilização destes comandos, um mesmo trecho de programa é apresentado nas linguagens *C++* e *Ada*, respectivamente [GHE 97].

```
switch (operator) {
        case '+': result = operand1 + operand2;
                  break;
        case '*': result = operand1 * operand2;
                  break;
        case '-': result = operand1 - operand2;
                  break;
        case '/': result = operand1 / operand2;
                  break;
        default: // não faz nada
                  break;
};
```

```
case OPERATOR is
when '+' => result := operand1 + operand2;
when '*' => result := operand1 * operand2;
when '-' => result := operand1 - operand2;
when '/' => result := operand1 / operand2;
when others => null;
end case;
```

Uma extensão importante da condição de múltiplas alternativas, sugerida por Dijkstra em 1975, corresponde a um mecanismo de seleção não determinístico. A forma geral proposta é:

if <condição 1> -> <bloco de comandos 1>
when <condição 2> -> <bloco de comandos 2>
...
when <condição n> -> <bloco de comandos n>
fi

Esta construção avalia todas as condições e quando mais de uma das condições é verdadeira, a alternativa cuja seqüência de comandos é executada é escolhida não deterministicamente. Isto significa que não há regra para escolha entre várias possibilidades, e qualquer uma delas pode ser escolhida. Se nenhuma das condições é verdadeira, o comando é considerado errado. As condições são chamadas de guarda (*guards*), e a construção como um todo é chamada de comandos guardados e foi implementada em *Ada* para programação concorrente (seção 4.2) [DER 90].

Estruturas de **repetição** são usadas para permitir que um bloco de comandos seja executado repetidamente. A maioria das LP fornecem diferentes tipos de construtores de laço para definir a iteração de certas ações (chamadas de corpo do laço). Freqüentemente há uma distinção entre laços onde o número de repetições é conhecido no início, e laços onde o corpo é executado repetidamente até uma condição ser satisfeita. O primeiro tipo é geralmente chamado de *for* e o segundo de *while*.

Os laços *for* definem uma variável de controle que recebe todos os valores de uma seqüência pré-definida, um após o outro. Para cada valor o corpo do laço é executado. *Pascal* permite iterações onde as variáveis de controle podem ser de qualquer tipo ordinal (*integer*, *boolean*, *character*, *enumeration* ou *subranges*). Um laço, então, tem a seguinte aparência:

for <variável de controle> := <limite inferior> to <limite superior> do <comandos>

A variável de controle assume todos os seus valores do limite inferior ao superior. A linguagem especifica que a variável não será alterada no laço, e seu valor será indefinido fora do laço. O código abaixo ilustra a utilização desta estrutura em Pascal.

```
type day = (sun, mon, tue, wed, thu, fri, sat);
var week_day: day;
...
for week_day := mon to fri do ...
```

O laço *for* em *C++*, por exemplo, pode ser escrito da seguinte maneira:

for (int i = 0; i < 10; i++) { ... }

Neste caso, o corpo do laço é executado para todos os valores de “i” de 0 a 9. A parte de controle do comando é claramente composta de três partes: uma inicialização, um teste feito antes de cada iteração que faz com que o laço seja suspenso se a expressão for igual a zero (falso), e uma expressão de incremento que é realizada depois de cada iteração. No exemplo anterior, o comando *for* também declara a variável “i”, cujo escopo estende-se até o final do bloco. É importante ressaltar que as três partes do comando podem ser omitidas, da seguinte maneira:

for (; ;) { ... }

Neste caso, tem-se um laço infinito.

O laço *while* descreve qualquer número de iterações do corpo do laço, incluindo zero. Geralmente sua forma é a seguinte:

while <condição> do <comandos>

Por exemplo, o código em *Pascal* para encontrar o máximo divisor comum de duas variáveis é:

```
while a<>b do
  begin
     if a>b then
        a := a – b
     else
        b := b – a
  end
```

A condição de laço é avaliada antes da execução do corpo do laço, que é suspenso se "a" for igual a "b" e neste caso contém o máximo divisor comum. Em *C++* o comando *while* é similar:

while (<expressão>) <comandos>

Geralmente as linguagens fornecem outro tipo de laço onde a variável de controle do laço é verificada no final do corpo. Neste caso, o corpo do laço é executado pelo menos uma vez. Em Pascal, esta construção tem a seguinte forma:

repeat
    <comandos>
until <condição>

Neste laço há iteração enquanto a condição for falsa, isto é, o laço termina quando a condição for satisfeita. Analogamente, *C++* fornece o seguinte comando [GHE 97]:

do <comandos> while (<expressão>);

Uma estrutura de controle que gera muita controvérsia, é a de desvio incondicional, isto é, que permite desviar a execução de qualquer unidade de programa sem restrição. Tais estruturas são geralmente conhecidas como *goto*, e seu uso é de valor questionável. Todavia, todas as linguagens imperativas populares, com exceção de *Modula-2*, fornecem um comando *goto*, cujo formato em geral é [DER 90]:

goto <label do comando>