Capítulo II – Amarrações

"... se antes de cada ato nosso nos puséssemos a prever todas as conseqüências dele, a pensar nelas a sério, primeiro as imediatas, depois as prováveis, depois as imaginárias, não chegaríamos sequer a mover-nos de onde o primeiro pensamento nos tivesse feito parar."

José Saramago

Amarração (`binding`) é um conceito amplamente utilizado no estudo de LPs. Em termos gerais, uma amarração é uma associação entre entidades de programação, tais como entre uma variável e seu valor ou entre um identificador e um tipo.

Nesse capítulo se discute o conceito de amarração enfocando especialmente as associações feitas entre identificadores e símbolos da LP com entidades de programação, tais como constantes, variáveis, procedimentos, funções e tipos.

Inicialmente apresentam-se os diversos momentos nos quais podem ocorrer amarrações. Em seguida, discutem-se as propriedades relacionadas com identificadores. Abordam-se também os ambientes de amarração e as noções de escopo das entidades de programação. Por fim, discute-se como podem ser feitas definições e declarações dessas entidades.

2.1 Tempos de Amarração

Existem inúmeros tipos de amarrações, as quais podem ocorrer em momentos distintos. O momento no qual uma amarração é realizada é conhecido como tempo de amarração. A seguir, são apresentadas descrições de diferentes tempos de amarração juntamente com exemplos.

- **Tempo de Projeto da LP:** Ao se projetar uma LP é necessário definir os símbolos e alguns identificadores que poderão ser usados para a construção de programas, bem como amarrá-los às entidades que representam. Por exemplo, a escolha do símbolo * para denotar a operação de multiplicação em C foi feita durante o projeto da linguagem.
- **Tempo de Implementação do Tradutor:** Algumas amarrações são efetuadas no momento em que se implementa o software responsável por traduzir o código da LP (em geral, o compilador). Por exemplo, a definição do intervalo de inteiros associado ao tipo *int* de C é realizada durante a implementação do compilador. Isso sig-

nifica que diferentes compiladores podem adotar diferentes intervalos para o tipo *int*.

- **Tempo de Compilação:** Um grande número de amarrações ocorre no momento em que o programa é compilado. São exemplos desse tipo de amarração em C a associação de uma variável a um tipo de dados e a associação, em uma expressão do programa, do operador * à operação que denota.
- **Tempo de Ligação:** Amarrações também ocorrem no momento em que vários módulos previamente compilados necessitam ser integrados (ou, no termo mais usado, ligados) para formar um programa executável. Por exemplo, a amarração entre a chamada de uma função da biblioteca padrão de C (tal como, *printf*) e o código compilado correspondente a essa função é realizado em tempo de ligação.
- **Tempo de Carga:** Outro momento onde ocorrem amarrações é durante o carregamento do programa executável na memória do computador. Nesse momento, são associadas áreas de memória às variáveis globais e constantes que serão usados pelo programa, assim como são substituídas várias referências no código executável por endereços absolutos de memória.
- **Tempo de Execução:** Outro grande número de amarrações ocorre durante a própria execução do programa. Exemplos desse tipo de amarração são a associação de um valor a uma variável ou a associação de áreas de memória às variáveis locais de uma função em C.

Costuma-se afirmar também que uma amarração é **estática** se ela ocorre antes da execução do programa e permanece inalterada ao longo de toda a execução. Já se a amarração ocorre ou é alterada durante a execução do programa, ela é chamada de amarração **dinâmica**.

O entendimento sobre amarrações e seus respectivos tempos colabora muito para o entendimento da semântica de LPs. Como durante a programação só se realizam amarrações entre identificadores e entidades de computação, vamos nos concentrar nelas a partir de agora. Começaremos discutindo identificadores, abordaremos ambientes de amarração e concluiremos estudando declarações e definições.

2.2 Identificadores

Identificadores são cadeias de caracteres definidos pelos programadores para servirem de referência às entidades de computação. A escolha apro-

priada de identificadores e símbolos facilita o entendimento dos programas.

O uso de identificadores também possibilita definir uma entidade em um ponto do programa e posteriormente utilizar aquele identificador para se referir àquela entidade em vários outros lugares. Além de aumentar a redigibilidade dos programas, isso faz com que o programa seja mais facilmente modificável, uma vez que, se a implementação da entidade deve ser alterada, a mudança afeta apenas a parte do programa na qual ela foi amarrada ao identificador e não as partes nas quais o identificador foi usado.

A sintaxe para formação de identificadores em LPs pode variar. Uma forma comum é apresentada a seguir (em BNF):

<identificador> ::= <letra> | <letra><sequencia>
<sequencia> ::= <caracterid> | <caracterid><sequencia>
<caracterid> ::= <letra> | <digito> | <sublinha>

Exemplo 2. 1 - Regras Sintáticas para Formação de Identificadores

Algumas LPs limitam o número máximo de caracteres que podem ser utilizados. Outras não impõem limites ou permitem que se criem nomes com tamanho ilimitado, mas fazem distinções apenas até um número determinado de caracteres. Algumas LPs definem o limite na sua definição, enquanto outras deixam para o implementador do compilador ou interpretador da LP definir o tamanho máximo.

LPs podem ser `case sensitive`, isto é, podem fazer distinções entre identificadores escritos com letras maiúsculas e minúsculas (C, MODULA-2, C++ e JAVA) ou não (PASCAL). As LPs que adotam a abordagem não sensitiva permitem que uma mesma entidade seja referenciada pelo mesmo nome escrito de várias maneiras distintas. Isto tende a provocar programas menos legíveis, visto que o programador pode abusar do uso das variações, dificultando o reconhecimento da amarração entre o identificador e a entidade que referencia. Por outro lado, as LPs que adotam a abordagem sensitiva permitem que um mesmo nome identifique várias entidades distintas, podendo gerar confusões no entendimento do programa, além de forçar o programador a lembrar como descreveu a entidade em termos do formato de seu identificador.

Identificadores devem ser formados por nomes significativos, isto é, devem prover informação a respeito do significado das entidades que denotam. Em particular, identificadores devem refletir o significado das entidades do domínio do problema e não da forma como elas são implementadas. Por exemplo, é sempre melhor definir um identificador chamado

palavra do que um chamado *lista_de_caracteres*, mesmo que o identificador esteja associado a uma entidade implementada dessa forma.

Identificadores que sejam visíveis ao longo de partes substanciais do programa, e que não sejam muito usados, devem ter significado óbvio e podem ser relativamente longos. Identificadores visíveis apenas em pequenos trechos do programa, mas muito usados, podem ser curtos. Tipicamente, nesses casos, deve-se usar abreviações ou acrônimos ou nomes convencionais, tais como, *i*, *j* e *p*.

Em geral, deve-se evitar formar identificadores que se diferem de forma sutil, como por exemplo, identificadores que se diferem apenas pela escrita em letra maiúscula e minúscula ou que se diferem apenas pelo uso da letra o maiúscula e o dígito zero. Além disso, é importante tentar manter um estilo consistente na formação de identificadores, embora isso nem sempre seja possível, uma vez que programas são frequentemente compostos por fragmentos de diferentes origens.

2.2.1 Identificadores Especiais

Alguns identificadores podem ter significado especial para a LP [SEBESTA, 1998]. Alguns podem ser vocábulos reservados, isto é, são símbolos da LP que não podem ser usados pelo programador para a criação de identificadores de entidades. Os identificadores *int*, *char*, *float*, *if*, *break* de C são exemplos de vocábulos reservados.

Outros identificadores podem ser vocábulos chave, isto é, só são símbolos da LP quando usados em um determinado contexto. Esse tipo de identificador pode ser muito ruim para a legibilidade de programas. O vocábulo *INTEGER* é um vocábulo chave em FORTRAN (veja o exemplo 2.2).

```
INTEGER R
INTEGER = 7
```

Exemplo 2. 2 - Vocábulos Chave

Na primeira linha do exemplo 2.2, o identificador *INTEGER* é usado para declarar a variável *R* como do tipo inteiro. Já na segunda linha, esse mesmo identificador denota uma variável.

Por fim, identificadores podem ser vocábulos pré-definidos, isto é, tem significados pré-definidos, mas podem ser redefinidos pelo programador. Por exemplo, as funções *fopen* e *fclose* são exemplos de identificadores pré-definidos na biblioteca padrão de C, mas que podem ser redefinidos pelo programador.

2.3 Ambientes de Amarração

A interpretação de comandos e expressões, tais como *a = 5* ou g*(a + 1)*, depende do que denotam os identificadores utilizados nesses comandos e expressões. A maioria das LPs permite que um mesmo identificador seja declarado em várias partes do programa denotando, presumivelmente, diferentes entidades.

O conceito de ambiente (`environment`) é utilizado para determinar o que os identificadores denotam ao longo do programa. Um ambiente corresponde a um conjunto de amarrações. Cada expressão e comando é interpretado num determinado ambiente e todos os identificadores que ocorrem devem ter amarrações nesse ambiente.

Expressões e comandos idênticos em diferentes partes de um programa podem ser interpretados diferentemente se seus ambientes são distintos. Por outro lado, em geral, só é permitida uma amarração por identificador dentro de um determinado ambiente. Uma exceção a essa última regra ocorre em C++, como ilustrado no exemplo 2.3.

```
int a = 13;
void f() {
  int b = a;
  int a = 2;
  b = b + a;
}
```

Exemplo 2. 3 - Amarração de Identificador a Duas Entidades Distintas no Mesmo Ambiente

Enquanto o identificador *a* na primeira linha da função *f* do exemplo 2.3 designa a variável global, esse mesmo identificador nas segunda e terceira linhas designa a variável local.

Apenas em uma LP muito elementar todas amarrações afetam o ambiente de todo o programa. Em geral, uma amarração tem um determinado escopo de visibilidade, isto é, a região do programa onde a entidade amarrada é visível. O escopo de visibilidade de uma LP pode ser estático ou dinâmico.

No escopo estático, o ambiente de amarração é determinado pela organização textual do programa. Assim, de maneira geral, as entidades de computação são amarradas em tempo de compilação.

No escopo dinâmico, o ambiente de amarração é determinado em função da sequência de ativação (chamada) dos módulos do programa, a qual só é determinada em tempo de execução. Em outras palavras, é o fluxo de controle do programa que determina as amarrações às entidades de computação.

2.3.1 Escopo Estático

O conceito de bloco é fundamental para o entendimento do escopo estático. Um bloco delimita o escopo de qualquer amarração que ele possa conter. Normalmente, um bloco é um subprograma ou um trecho de código delimitado através de marcadores, tais como, as chaves (*{* e *}*) de C, C++ e JAVA ou os vocábulos *begin* e *end* introduzidos por ALGOL-60 e adotados por PASCAL e ADA.

A estrutura de blocos de uma LP é a relação textual entre blocos. A figura 2.1 ilustra uma classificação dos tipos de estruturas, tal como apresentada por Watt [WATT, 1990]:

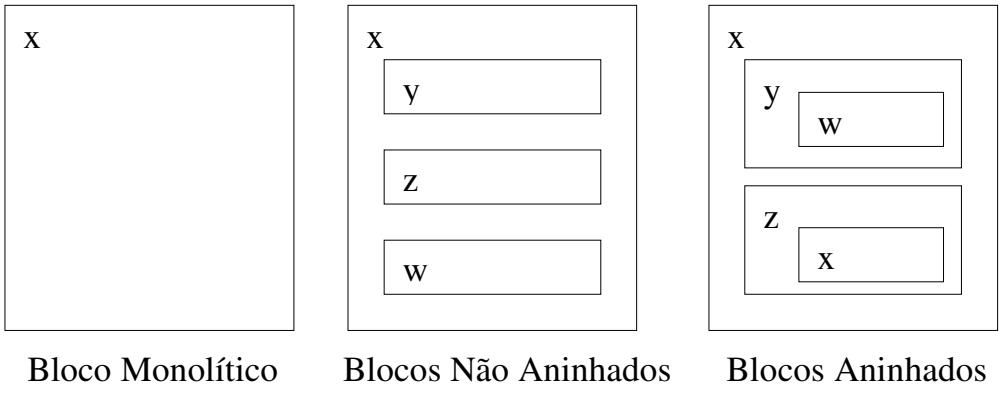

Figura 2. 1 - Estruturas de Blocos (adaptada de Watt [WATT, 1990])

Na estrutura monolítica todo o programa é composto por um único bloco. Todas as amarrações têm como escopo de visibilidade o programa inteiro. Essa estrutura de blocos é a mais elementar possível e não é apropriada para programas grandes, uma vez que todas as amarrações de identificadores devem ser agrupadas num mesmo lugar. Isso faz com que o programador tenha de interromper frequentemente a análise de trechos do programa e desviar sua atenção para o lugar onde pode consultar o significado dos identificadores usados nesses trechos. Mais ainda, isso dificulta o trabalho simultâneo de vários programadores em um mesmo programa visto que os identificadores criados por um deles devem ser necessariamente distintos dos identificadores criados pelos outros. Versões antigas de BASIC e COBOL adotam essa estrutura.

A estrutura de blocos não aninhada é considerada um avanço em relação à estrutura de blocos monolítica, uma vez que o programa é dividido em vários blocos. Nessa estrutura, o escopo de visibilidade dos identificadores é o bloco onde foram criados. Esses identificadores são chamados de locais. Os identificadores criados fora do bloco são chamados de globais, uma vez que seu escopo de visibilidade é todo o programa. Uma desvantagem associada à estrutura de blocos não aninhada é que qualquer identi-

ficador que não pode ser local é forçado a ser global e ter todo o programa como escopo, mesmo que seja acessado por poucos blocos. Outra desvantagem é a exigência de que todos os identificadores globais tenham identificadores distintos. FORTRAN adota esse tipo de estrutura. Todos subprogramas são separados e cada um atua como um bloco.

A estrutura aninhada é considerada um avanço ainda maior. LPs como PASCAL, MODULA-2 e ADA adotam essa estrutura. São, por isso, chamadas LPs ALGOL-`like` uma vez que foi ALGOL a primeira linguagem a utilizá-la. Qualquer bloco pode ser aninhado dentro de outro bloco e localizado em qualquer lugar que seja conveniente. Identificadores podem ser amarrados dentro de cada bloco. Normalmente, para descobrir qual entidade está amarrada a um identificador, deve-se procurar a declaração da entidade no bloco onde é usada. Se não a encontrar, deve-se procurar no bloco mais externo imediato e assim por diante.

Entidades podem se tornar inacessíveis quando se usa o mesmo identificador para denotar diferentes entidades em blocos aninhados. Isso ocorre em C, como mostra o exemplo 2.4.

```
main() {
  int i = 0,  x = 10;
  while (i++ < 100) {
       float x = 3.231;
       printf("x = %f\n", x*i);
  }
}
```

Exemplo 2. 4 - Ocultamento de Entidade em Blocos Aninhados

No exemplo 2.4, a variável *x* inteira criada no bloco mais externo não é visível dentro do bloco mais interno porque neste bloco foi criada uma outra variável de tipo ponto flutuante com o mesmo identificador *x*.

Com o intuito de evitar confusões, JAVA não permite que um mesmo identificador seja utilizado para designar entidades distintas em blocos aninhados. Já em algumas LPs, como ADA, é permitido usar o nome do bloco para acessar a entidade que fica oculta quando outra entidade é associada ao mesmo identificador em um bloco aninhado interno. Isso é chamado de referência seletiva. O exemplo 2.5 ilustra essa situação.

```
procedure A is
       x : integer;
       procedure B is
            y : integer;
            procedure C is
                 x : integer;
```

```
            begin
                  x := A.x;
            end C;
      begin
            null;
      end B;
 begin
      null;
end A;
```

Exemplo 2. 5 - Referência Seletiva em ADA

No exemplo 2.5, a referência seletiva *A.x* dentro do bloco *C* permite que a variável *x* do bloco *A* seja acessada. Observe que a referência *x* dentro de *C* designa a variável *x* do próprio bloco *C*, como seria de se esperar.

Em certas situações, contudo, a estrutura aninhada pode requerer que uma variável seja declarada globalmente, embora seja usada por poucos blocos.

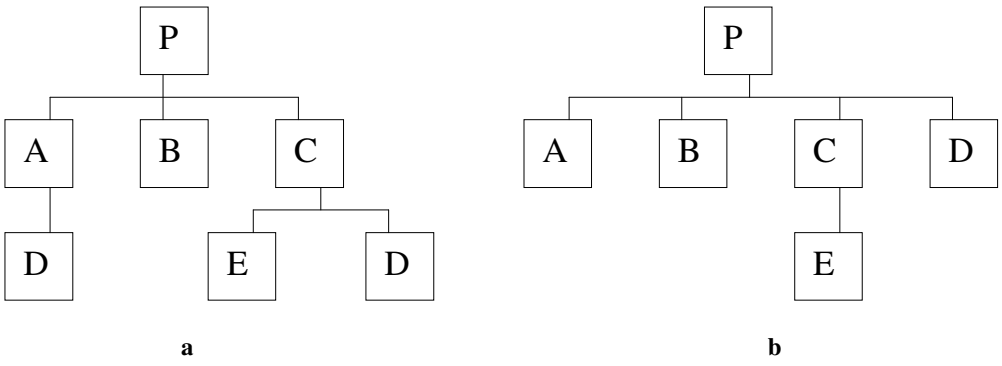

Figura 2. 2 - Aninhamento de Blocos

Na figura 2.2.a, o bloco D deve ser repetido dentro dos blocos A e C para que ele possa ser usado única e exclusivamente por esses dois blocos. Uma alternativa para a não repetição do código seria elevar o bloco D para o mesmo nível de amarração de A e C (ver figura 2.2.b). Nesse caso, D seria visível por esses blocos. Contudo, D também passaria a ser visível por P e B, o que pode não ser interessante. Outra opção seria permitir declarar que um bloco faz uso de outro bloco criado dentro de um terceiro. Por exemplo, poder-se-ia declarar que o bloco A utiliza o bloco D criado dentro do bloco C.

C utiliza uma abordagem mista na qual os blocos definidos por funções adotam uma estrutura não aninhada e os blocos internos às funções adotam uma estrutura aninhada. Nesse caso, o que pode ser reusado (as funções) ou compartilhado (as variáveis globais) se torna visível para todos.

Já o que não pode ser reusado (os blocos internos) são aninhados. Observe como isso é feito no exemplo 2.6.

```
int x = 10;
int y = 15;
void f() {
  if (y – x) {
    int z = x + y;
  }
}
void g() {
  int w;
  w = x;
}
main() {
  f();
  x = x + 3;
  g();
}
```

Exemplo 2. 6 - Estrutura de Blocos de C

Observe no exemplo 2.6 que qualquer função pode usar a variável global *x* e chamar qualquer outra função. Observe também que dentro do bloco definido na função *f* existe um bloco interno aninhado associado ao comando *if*. Observe que mesmo nessa abordagem, mais uma vez, as variáveis globais e funções se tornam visíveis para blocos que não as utilizam. Pelas regras de escopo de C, a função *g* teria permissão para utilizar a variável global *y* e chamar a função *f*.

2.3.2 Escopo Dinâmico

No escopo dinâmico as entidades são amarradas aos identificadores de acordo com o fluxo de controle do programa. APL, SNOBOL4 e versões iniciais de LISP adotam este tipo de escopo. Considere o exemplo 2.7, escrito numa LP hipotética:

```
procedimento sub() {
    inteiro x = 1;
    procedimento sub1() {
       escreva( x);
    }
    procedimento sub2() {
      inteiro x = 3;
      sub1();
    }
```

```
    sub2();
    sub1();
}
```

Exemplo 2. 7 - Escopo Dinâmico

Quando *sub* chama *sub2* e este chama *sub1*, o valor escrito de *x* por *sub1* é *3*, isto é, *x* é uma referência à variável criada em *sub2*. Quando *sub* chama *sub1* diretamente, o valor escrito de *x* por *sub1* é *1*, isto é, *x* é uma referência à variável criada em *sub*.

LPs que adotam o escopo dinâmico apresentam os seguintes problemas:

1. perda de eficiência pois a checagem de tipos tem de ser feita durante a execução;
2. legibilidade do programa é reduzida pois a sequência de chamadas de subprogramas deve ser conhecida para determinar o significado das referências a variáveis não locais;
3. acesso menos eficiente às variáveis porque é necessário seguir a cadeia de chamadas de subprogramas para identificar as referências não locais;
4. confiabilidade do programa é reduzida pois variáveis locais podem ser acessadas por qualquer subprograma chamado subsequentemente no processo.

Como consequência desses problemas, a maioria das LPs atuais não adotam a abordagem de escopo dinâmico.

2.4 Definições e Declarações

Definições e declarações são frases de programa elaboradas para produzir amarrações. Definições produzem amarrações entre identificadores e entidades criadas na própria definição. Declarações produzem amarrações entre identificadores e entidades já criadas ou que ainda o serão.

Algumas LPs, tais como C, C++ e JAVA, permitem que sejam feitas declarações e definições dentro de blocos. Enquanto C requer que elas sejam feitas imediatamente após o marcador (*{*) de início do bloco e antes de qualquer comando executável, C++ e JAVA permitem que elas sejam feitas em qualquer ponto do bloco. Já PASCAL não permite que sejam feitas declarações e definições dentro dos blocos internos, apenas na área destinada a amarrações no programa e nos subprogramas. O exemplo 2.8 mostra uma função *f* em C++ na qual são definidas as variáveis *a* e *b*.

```
void f() {
  int a = 1;
  a = a + 3;
  int b = 0;
```

```
    b = b + a;
}
```

Exemplo 2. 8 - Localização de Definições de Variáveis em C++

Observe que a variável *b* é definida após a realização do comando de atribuição à variável *a.* Note que um compilador de C acusaria erro ao compilar essa função, uma vez que em C todas as amarrações devem ocorrer no início do bloco.

2.4.1 Declarações de Constantes

Uma declaração de constante amarra um identificador a um valor pré-existente que não pode ser alterado ao longo da execução do programa. Isso pode ser feito em C++ tal como na seguinte linha de código:

```
const float pi = 3.14;
```

Nessa linha é criada uma constante de nome *pi*. Essa mesma declaração é válida em C. Contudo, a definição de C só requer que os compiladores avisem ao programador de tentativas de alterações de constantes, permitindo assim que eles ignorem a definição e aceitem alterações de valores das constantes!!! Por essa razão, programadores C continuam utilizando o mecanismo tradicional de macros quando querem criar constantes em C:

```
#define pi 3.14
```

Com esse mecanismo todas as referências a *pi* no código serão substituídas por *3.14*, antes do início da compilação, tendo efeito equivalente a definição de *pi* como constante.

No entanto, pode ser vantajoso usar *const* ao invés de *define* em C porque esse último mecanismo não reconhece regras de escopo (a constante será reconhecida do ponto de declaração até o final do programa). Em caso de uso de *const*, a constante só será reconhecida dentro do escopo de visibilidade definido pelo seu ambiente de amarração.

Algumas linguagens, como PASCAL e MODULA-2, requerem que constantes tenham seus valores definidos estaticamente (em tempo de compilação). Já ADA, C++ e JAVA permitem que sejam usados valores calculados dinamicamente (em tempo de execução do programa). Essa é uma outra vantagem do uso de *const* em relação a *define*. O mecanismo de macros somente permite a declaração de constantes estáticas. JAVA utiliza a palavra *final* para declarar constantes, tal como ilustrado no exemplo 2.9[2.1].

[2.1] A palavra *static* utilizada no exemplo 2.9 é utilizada para indicar que a constante é um atributo de classe (isto é, um atributo comum para todos os objetos da classe). As declarações sem *static* provocam a criação de atributos constantes individuais para cada objeto da classe.

```
final int const1  = 9;
static final int const2  = 39;
final int const3 = (int)(Math.random()*20);
static final const4 = (int)(Math.random()*20);
```

Exemplo 2. 9 - Declaração de Constantes em JAVA

Enquanto as duas primeiras linhas do exemplo 2.9 ilustram a criação de constantes estáticas, as duas últimas ilustram a criação de constantes dinâmicas. *Math.random* é uma função calculada em tempo de execução que gera um número aleatório.

JAVA também permite a inicialização de constantes em ponto distinto da declaração. No exemplo 2.10, é mostrada uma situação onde isso ocorre. Nesse exemplo, a constante *j* é inicializada no método *Construtor*.

```
final int j;
Construtor () {
   j = 1;
}
```

Exemplo 2. 10 - Inicialização de Constante em JAVA

2.4.2 Definições e Declarações de Tipos

Uma definição de tipo amarra um identificador a um tipo criado na própria definição. As definições de *struct*, *union*, *enum* em C são definições de tipo, tal como visto no exemplo 2.11.

```
struct data {
        int d, m, a;
};
```

```
union angulo {
        int graus;
        float rad;
};
```

```
enum dia_util {
   seg, ter, qua, qui, sex
};
```

Exemplo 2. 11 - Definições de Tipos em C

Uma declaração de tipo amarra um identificador a um tipo definido em outro ponto do programa. A primeira linha do exemplo 2.12 (em C) indica que o identificador *data* está amarrado a um tipo estrutura definido em outro ponto do programa. As duas linhas restantes mostram o uso de *typedef* na produção de declarações de tipo, amarrando respectivamente os identificadores *curvatura* e *aniversario* aos tipos *union angulo* e *struct data*.

```
struct data;
typedef union angulo curvatura;
typedef struct data aniversario;
```

Exemplo 2. 12 - Declarações de Tipos em C

Observe que, com o uso de *typedef*, não foi criado um novo tipo, ou seja, produz-se simplesmente um novo identificador para designar tipos previamente definidos. Por exemplo, é possível usar uma variável do tipo *union angulo* onde se espera uma variável do tipo *curvatura* e vice-versa.

2.4.3 Definições e Declarações de Variáveis

Definições de variáveis são as mais comuns em LPs. Uma definição de variável é um trecho de programa onde ocorre uma amarração determinando a criação de uma variável, isto é, a alocação de um conjunto de células de memória e sua associação à variável definida naquele trecho. Veja alguns exemplos de definições em C:

```
int k;          union angulo ang;          struct data d;
```

Variáveis podem ser definidas em uma mesma frase para economizar digitação e espaço do programa, aumentando a redigibilidade. A seguinte linha em C mostra um exemplo de como isso pode ser feito:

```
int *p, i, j, k, v[10];
```

Observe nessa linha que são definidas três variáveis inteiras (*i, j, k*), uma variável ponteiro para inteiro (*p*) e um vetor de inteiros com 10 elementos (*v*). Se, por um lado, construções como essa aumentam a redigibilidade do programa, por outro lado, ao misturar a definição de variáveis de diferentes tipos, elas reduzem a legibilidade e podem provocar erros. Portanto, é aconselhável que só se façam definições de variáveis de um único tipo em uma mesma frase do programa.

Uma vantagem de se definir variáveis em frases separadas, mesmo que de um único tipo, é permitir a inserção de comentários explicativos a respeito das variáveis ao lado de cada definição, aumentando a legibilidade.

Uma variável pode ser inicializada durante a sua definição. O exemplo 2.13 mostra isso sendo feito em C.

```
int i = 0;
char virgula = ', ';
float f, g = 3.59;
int j, k, l = 0, m=23;
```

Exemplo 2. 13 - Inicialização de Variáveis em C Durante a Definição

As duas últimas linhas do exemplo 2.13 mostram como combinar definições em uma mesma frase nas quais algumas variáveis são inicializadas (no caso: *g, l* e *m*) e outras não (no caso: *f, j* e *k*).

Certas linguagens oferecem inicialização implícita (também conhecida como inicialização `default`) de variáveis. Em C e C++, variáveis alocadas estaticamente (isto é, variáveis globais) são inicializadas com zero

do tipo apropriado. Isso significa que, se as variáveis definidas no exemplo 2.13 forem globais, as variáveis *j* e *k* serão inicializadas automaticamente com o valor *0* e a variável *f* será inicializada com o valor *0.0*. Já variáveis alocadas dinamicamente (isto é, variáveis locais e variáveis alocadas explicitamente na memória livre) não são inicializadas, ou seja, têm valores indefinidos (lixo). JAVA adota uma outra política. Variáveis definidas como atributos de classe podem ser inicializadas implicitamente e variáveis locais aos métodos, não.

Também é comum se permitir inicializar explicitamente variáveis através do uso de expressões. De modo similar ao mecanismo de inicialização implícita, em C e C++, as variáveis alocadas estaticamente só permitem a inicialização com expressões estáticas, isto é, expressões que podem ter seu valor calculado em tempo de compilação. Já as variáveis alocadas dinamicamente podem ser inicializadas com expressões calculadas em tempo de execução. O único requisito exigido é que os valores necessários para o cálculo da expressão sejam conhecidos no momento da criação da variável. O exemplo 2.14 mostra a inicialização, em C++, de uma variável (*k*) através do uso de uma expressão dinâmica.

```
void f(int x) {
  int i;
  int j = 3;
  i = x + 2;
  int k = i * j * x;
}
```

Exemplo 2. 14 - Inicialização com Uso de Expressão Dinâmica

Variáveis compostas também podem ser inicializadas em sua definição. No caso de C e C++, os valores usados na inicialização devem ser listados em ordem para que haja correspondência com os elementos da variável composta. A linha seguinte mostra a inicialização explícita de um vetor de inteiros em C.

```
int v[3] = { 1, 2, 3 };
```

Uma declaração de variável serve para indicar que a variável correspondente àquele identificador é definida em outro módulo de código ou para amarrar um identificador a uma variável existente. C utiliza declarações de variáveis para indicar que uma variável é definida em um módulo externo. Isso significa que o compilador não deve gerar código para alocar espaço de memória para essa variável (isso será feito pelo módulo externo), mas poderá utilizar a declaração para verificar o uso apropriado da variável externa nas operações do programa.

```
extern int a;
```

C++ permite que declarações de variáveis sejam feitas amarrando um identificador a uma variável já existente. Uma desvantagem dessa abordagem é permitir a produção de sinônimos (`aliases`), tornando mais difícil entender os programas e podendo provocar erros de programação. O exemplo 2.15 mostra uma situação na qual isso ocorre. Observe que o incremento da variável *j* implica em alterar implicitamente o valor da variável *r* para *11*.

```
int r = 10;
int &j = r;
j++;
```

Exemplo 2. 15 - Declaração de Variável

2.4.4 Definições e Declarações de Subprogramas

Subprogramas se compõem de cabeçalho e corpo. No cabeçalho são especificados o identificador do subprograma, sua lista de parâmetros e o tipo de retorno (se for o caso). No corpo é definido o algoritmo que implementa o subprograma e também é especificado o valor de retorno (se for o caso).

Uma definição de subprograma contém a especificação do cabeçalho e do corpo do subprograma. O exemplo 2.16 ilustra um subprograma em C:

```
int soma (int a, int b) {
  return a + b;
}
```

Exemplo 2. 16 - Definição de Subprograma em C

Uma declaração de subprograma contém apenas o cabeçalho do subprograma. Ela indica que a definição do subprograma ocorre em outro trecho do código ou em outro módulo. Declarações de subprogramas são usadas para permitir a verificação do uso apropriado do subprograma durante a compilação. O exemplo 2.17 mostra o uso de uma declaração de subprograma em C.

```
int incr (int);
void f(void) {
  int k = incr(10);
 }
int incr (int x) {
  x++;
  return x;
}
```

Exemplo 2. 17 - Declaração de Subprograma em C

A primeira linha de código do exemplo 2.17 é uma declaração do subprograma *incr*. Em C, só é necessário dar nomes aos parâmetros na definição do subprograma, bastando especificar o seu tipo na declaração. Observe que no corpo da função *f* ocorre uma chamada a *incr*, que só é definida posteriormente. O compilador pode verificar se a chamada a *incr* está correta em *f* porque a declaração de *incr* foi antecipada[2.2].

2.4.5 Composição de Definições

Definições podem ser compostas a partir de outras definições ou a partir delas mesmas. Definições compostas podem ser sequenciais ou recursivas.

2.4.5.1 Definições Sequenciais

Definições sequenciais se utilizam de outras definições estabelecidas anteriormente no programa. Elas permitem que as amarrações produzidas em uma definição sejam usadas nas demais. O exemplo 2.18 mostra a ocorrência de definições sequenciais em um código C.

```
struct funcionario {
  char nome [30];
  int matricula;
  float salario;
};
struct empresa {
  struct funcionario listafunc [1000];
  int numfunc;
  float faturamento;
};
int m = 3;
int n = m;
```

Exemplo 2. 18 - Definições Sequenciais em C

No exemplo 2.18, a definição do tipo *struct empresa* utiliza a definição do tipo *struct funcionario* e a definição da variável *n* utiliza a definição da variável *m*.

Definições sequenciais de subprogramas normalmente envolvem chamadas a outros subprogramas. Contudo, algumas LPs fornecem uma forma

---

[2.2] No caso específico do exemplo 2.17, onde a definição do subprograma se encontra no mesmo arquivo onde o subprograma é usado, um compilador poderia dispensar a declaração de *incr*. Para tanto, o compilador deveria varrer previamente o código identificando todos os cabeçalhos de subprogramas para posteriormente realizar as verificações de uso.

menos comum de definição sequencial de subprogramas. Nessas LPs, o lado direito da definição sequencial pode ser qualquer expressão que produza um valor subprograma. ML é uma linguagem que oferece esse tipo de recurso. O exemplo 2.19 mostra a definição sequencial das funções *impar* e *jogo* em ML.

```
val par = fn (n: int) => (n mod 2 = 0)
val negacao = fn (t: bool) => if t then false else true
val impar = negacao o par
val jogo = if x < y then par else impar
```

Exemplo 2. 19 - Definições Sequenciais em ML

Observe que a função *impar* é definida a partir das funções *par* e *negacao*, já definidas anteriormente. Observe também que a definição de *impar* não envolve a chamada dos subprogramas *negacao* e *par*. De fato, a definição de *impar* é dada através da especificação de uma expressão que utiliza o operador de composição de funções (*o*) aplicado sobre os valores *par* e *negacao*. Note, por fim, que a função *jogo* é definida de maneira similar. Se no momento da criação de *jogo*, o valor associado a *x* for inferior ao valor associado a *y*, *jogo* se referirá à função *par*. Caso contrário, se referirá à função *impar*.

Tais possibilidades são eliminadas em LPs onde definições de funções e procedimentos são a única forma de amarração de um identificador a uma função ou procedimento.

2.4.5.2 Definições Recursivas

Definições recursivas são aquelas que utilizam as próprias amarrações que produzem. Algumas versões de LPs antigas (tal como, FORTRAN e COBOL) não suportam recursão e, como conseqüência, se enfraquecem. As LPs mais modernas suportam recursão, em geral, restringindo-a a tipos e definições de procedimentos e funções, que são, de fato, os modos mais úteis de recursão.

Idealmente, tipos recursivos devem ser oferecidos pelas LPs de maneira indiscriminada. Contudo, em linguagens que utilizam explicitamente o conceito de ponteiros (tal como C), a definição de tipos recursivos é restrita, através de uma regra sintática, a tipos que envolvem esses elementos. O exemplo 2.20 mostra um tipo recursivo *struct lista* sendo definido em C:

```
struct lista {
  int elemento;
  struct lista * proxima;
};
```

Exemplo 2. 20 - Tipo Recursivo em C

Definições de funções também podem ser recursivas em C. O exemplo 2.21 mostra a definição recursiva da função *potencia* em C.

```
float potencia (float x, int n) {
        if (n == 0) then {
          return 1.0;
        } else if (n < 0) {
          return 1.0/ potencia (x, -n);
        } else {
          return x * potencia (x, n - 1);
        }
}
```

Exemplo 2. 21 - Definição Recursiva de Função em C

Definições de funções em C também podem ser mutuamente recursivas. O exemplo 2.22 mostra as definições mutuamente recursivas, em C, das funções *primeira* e *segunda*.

```
void segunda (int);
void primeira (int n) {
  if (n < 0) return;
  segunda (n – 1);
}
void segunda (int n) {
  if (n < 0) return;
  primeira (n – 1);
}
```

Exemplo 2. 22 - Definições Mutuamente Recursivas em C

Existem desvantagens em tratar declarações como automaticamente recursivas. Suponha que se queira redefinir a função *strcmp* de C. O programador poderia tentar redefinir essa função utilizando a versão original de *strcmp*, tal como ilustrado no exemplo 2.23.

```
int strcmp (char *p, char *q) {
    return !strcmp (p, q);
}
```

Exemplo 2. 23 - Erro em Definição de Função *strcmp* em C

A função definida no exemplo 2.23 não alcança o resultado esperado. A chamada *strcmp (p, q)* será recursiva, enquanto o desejo do nosso programador era chamar a função da biblioteca! Seria melhor se o programador pudesse escolher se a declaração é recursiva ou não.

Em ML, é possível definir se uma declaração é recursiva ou não. O exemplo 2.24 ilustra a definição recursiva em ML da função *mdc*. Observe

que a palavra *rec* colocada após a palavra *val* indica que essa é uma função recursiva, isto é, referências a *mdc* no corpo da função significam chamadas recursivas.

```
val rec mdc = fn ( m:int, n: int) = >
        if m > n then mdc (m – n, n)
        else if m < n then mdc (m, n – m)
        else m
```

Exemplo 2. 24 - Explicitação de Recursividade em Função ML

2.5 Considerações Finais

Nesse capítulo foi apresentado e discutido o conceito de amarração. Mostrou-se que as amarrações podem ocorrer em diferentes momentos, desde o instante de criação da linguagem ao tempo de execução dos programas. O enfoque do capítulo se concentrou na amarração de identificadores a entidades dos programas e no estudo dos conceitos de ambientes de amarração e escopo de visibilidade.

Mostrou-se ainda que as amarrações de identificadores às entidades dos programas podem ser feitas através de definições ou declarações. Nesse contexto, abordou-se as amarrações de identificadores a constantes, tipos, variáveis e subprogramas.

Estudos mais aprofundados dos conceitos de tipos de dados, variáveis e constantes e subprogramas serão vistos, respectivamente, nos capítulos 3, 4 e 6 desse livro.

2.6 Exercícios

1. Liste pelo menos cinco diferentes tipos de amarrações que ocorrem no seguinte trecho de código C.

   ```
   float j = 3.2;
   j = j – 1.7;
   ```

2. Especifique as regras de formação de identificadores de C, C++ e JAVA. Responda ainda se existem limites no número máximo de caracteres que podem ser usados e quais tipos de identificadores especiais são considerados.

3. Considere o seguinte trecho de código em ADA:

   ```
   procedure A is
           u : integer;
           procedure B is
                   v : integer;
                   procedure C is
                           x : integer;
   ```

```
                    procedure D is
                        u : integer;
                    begin
                        null;
                    end D;
                    procedure E is
                        v : integer;
                    begin
                        u := 7;
                    end E;
                begin
                    null;
                end C;
                procedure F is
                    y : integer;
                    procedure G is
                        x : integer;
                    begin
                        null;
                    end G;
                begin
                    u := 10;
                end F;
    begin
                null;
    end B;
begin
        null;
end A;
```

Identifique quais variáveis e subprogramas são visíveis em cada um dos subprogramas desse trecho de código. Suponha que novos requisitos do problema demandem que a variável *u* de *D* possa ser acessada por *G*. Quais modificações necessitariam ser feitas no programa? Cite erros que poderiam ocorrer em situações como essa.

4. Indique qual valor será escrito pelo trecho de código seguinte no caso da linguagem de programação utilizada adotar escopo estático e no caso dela adotar escopo dinâmico.

```
procedimento sub() {
    inteiro x = 1;
   inteiro y = 1;
   procedimento sub1() {
        se (x = 1 & y = 1) então
            sub2();
        senão
```

```
            sub3();
    }
    procedimento sub2() {
       inteiro x = 2;
       y = 0;
       sub1();
    }
    procedimento sub3() {
        escreva( x);
    }
   sub1();
}
```

Cite e explique os problemas de legibilidade do trecho de código acima quando se adota o escopo estático e o escopo dinâmico.

5. Compare, em termos de legibilidade, as opções de C e C++ relativas à localização das definições e declarações nos programas.

6. Identifique o problema que ocorre no seguinte trecho de código C. Explique porque ele ocorre e indique como poderia ser resolvido.

```
void circulo () {
    #define pi 3.14159
    float  raio = 3;
    float area = pi * raio * raio;
    float perimetro = 2 * pi  * raio;
}
void pressao () {
    float pi = 3.2, pf = 5.3;
    float variacao;
    variacao = pf – pi;
}
```

7. Indique quais valores serão escritos pelo seguinte programa em C. Explique sua resposta e discuta a postura da linguagem em termos de ortogonalidade e de potencialidade para indução de erros de programação.

```
int i;
main () {
    printf ("%d\n", i);
   f();
}
void f () {
    int i;
    printf ("%d\n", i);
}
```

8. Uma declaração de função é um segmento de código contendo apenas a sua assinatura (isto é, um segmento de código com o cabeçalho da fun-

ção, mas sem seu corpo). Apresente uma situação na qual a declaração de funções é útil (ou necessária) em C. Justifique sua resposta explicando para que o compilador utiliza a declaração.