5.5 ESCOPO

Um dos fatores mais importantes para o entendimento das variáveis é o escopo. O **escopo** de uma variável é a faixa de sentenças nas quais ela é visível. Uma variável é **visível** em uma sentença se ela pode ser referenciada nessa sentença. As regras de escopo de uma linguagem determinam como uma ocorrência em particular de um nome é associada com uma variável. Em particular, regras de escopo determinam como referências a variáveis declaradas fora do subprograma ou bloco em execução são associadas com suas declarações e, logo, com seus atributos (os blocos são discutidos na Seção 5.5.2). Logo, um claro entendimento dessas regras para uma linguagem é essencial para a habilidade de escrever ou ler programas nela.

Conforme definido na Seção 5.4.3.2, uma variável é local a uma unidade ou a um bloco de programa se ela for lá declarada. As variáveis **não locais** de uma unidade ou de um bloco de programa são aquelas visíveis dentro da unidade ou do bloco de programa, mas não declaradas nessa unidade ou nesse bloco.

Questões de escopo para classes, pacotes e espaços de nomes são discutidas no Capítulo 11.

5.5.1 Escopo estático

O ALGOL 60 introduziu o método de vincular nomes a variáveis não locais, chamado de **escopo estático**[8], copiado por muitas linguagens imperativas subsequentes (e por muitas linguagens não imperativas). O escopo estático é chamado assim porque o escopo de uma variável pode ser determinado estaticamente – ou seja, antes da execução. Isso permite a um leitor de programas humano (e um compilador) determinar o tipo de cada variável.

Existem duas categorias de linguagens de escopo estático: aquelas nas quais os subprogramas podem ser aninhados, as quais criam escopos estáticos aninhados, e aquelas nas quais os subprogramas não podem ser aninhados. Nessa última categoria, os escopos estáticos também são criados para subprogramas, mas os aninhados são criados apenas por definições de classes aninhadas ou de blocos aninhados.

Ada, JavaScript, Fortran 2003 e PHP permitem subprogramas aninhados, mas as linguagens baseadas em C não.

Nossa discussão sobre o uso de escopo estático nesta seção foca nas linguagens que permitem subprogramas aninhados. Inicialmente, assumimos que *todos* os escopos são associados com unidades de programas e que todas as variáveis não locais referenciadas são declaradas em outras unidades de programa[9]. Neste capítulo, assumimos que o uso de escopos é o único método de acessar variáveis não locais nas linguagens em discussão, o que não é verdade para todas as linguagens. Não é verdade nem mesmo para todas as linguagens que usam escopo estático, mas tal premissa simplifica a discussão aqui.

Quando o leitor de um programa encontra uma referência a uma variável, os atributos dessa variável podem ser determinados por meio da descoberta da sentença na qual ela está declarada. Em linguagens de escopo estático com subprogramas, esse processo pode ser visto da seguinte forma. Suponha que uma referência é feita a uma variável `x` em um subprograma `Sub1`. A declaração correta é encontrada primeiro pela busca das declarações do subprograma `Sub1`. Se nenhuma declaração for encontrada para a variável lá, a busca continua nas declarações do subprograma que declarou o subprograma `Sub1`, chamado de **pai estático**. Se uma declaração de `x` não é encontrada lá, a busca continua para a próxima unidade maior que envolve o subprograma onde está sendo feita a busca (a unidade que declarou o pai de `Sub1`), e assim por diante, até que uma declaração para `x` seja encontrada ou a maior unidade de declaração tenha sido buscada sem sucesso. Nesse caso, um erro de variável não declarada é detectado. O pai estático do subprograma `Sub1`, e seu pai estático, e assim por diante até (e incluindo) o maior subprograma que envolve os demais, são chamados de

[8] O escopo estático é chamado algumas vezes de *escopo léxico*.

[9] Variáveis não locais não definidas em outras unidades de programa são discutidas na Seção 5.5.4.

ancestrais estáticos de Sub1. Técnicas de implementação reais para escopo estático, discutidas no Capítulo 10, são muito mais eficientes do que o processo que acabamos de descrever.

Considere o seguinte procedimento em Ada, chamado Big, no qual estão aninhados os procedimentos Sub1 e Sub2:

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin  -- de Sub1
    ...
    end;  -- de Sub1
  procedure Sub2 is
    begin  -- de Sub2
    ...X...
    end;  -- de Sub2
  begin  -- de Big
  ...
  end;  -- de Big
```

De acordo com o escopo estático, a referência à variável X em Sub2 é para o X declarado no procedimento Big. Isso é verdade porque a busca por X começa no procedimento no qual a referência ocorre, Sub2, mas nenhuma declaração para X é encontrada lá. A busca continua no pai estático de Sub2, Big, onde a declaração de X é encontrada. O X declarado em Sub1 é ignorado, porque ele não está nos ancestrais estáticos de Sub2.

A presença de nomes pré-definidos, descritos na Seção 5.2.3, complica esse processo. Em alguns casos, um nome pré-definido é como uma palavra-chave, que pode ser redefinida pelo usuário. Em tais casos, um nome pré-definido é usado apenas se o programa de usuário não contém uma redefinição.

Em linguagens que usam escopo estático, independentemente de ser permitido o uso de subprogramas aninhados ou não, algumas declarações de variáveis podem ser ocultadas de outros segmentos de código. Por exemplo, considere mais uma vez o procedimento Big em Ada. A variável X é declarada tanto em Big quanto em Sub1, aninhado dentro de Big. Dentro de Sub1, cada referência simples para X é para o X local. Logo, o X externo está oculto de Sub1.

Em Ada, variáveis ocultas de escopos ancestrais podem ser acessadas com referências seletivas, que incluem o nome do escopo ancestral. Em nosso

NOTA HISTÓRICA

A definição original do Pascal (Wirth, 1971) não especifica quando a compatibilidade de tipo por estrutura ou por nome deve ser usada. Isso é altamente prejudicial à portabilidade, porque um programa correto em uma implementação pode ser ilegal em outra. O Pascal Padrão ISO – ISO Standard Pascal (ISO, 1982) informa as regras de compatibilidade de tipos da linguagem, as quais não são completamente por nome nem completamente pela estrutura. A estrutura é usada na maioria dos casos, enquanto o nome é usado para parâmetros formais e em algumas outras poucas situações.

procedimento de exemplo anterior, o `X` declarado em `Big` pode ser acessado em `Sub1` pela referência `Big.X`.

5.5.2 Blocos

Muitas linguagens permitem que novos escopos estáticos sejam definidos no meio do código executável. Esse poderoso conceito, introduzido no ALGOL 60, permite uma seção de código ter suas próprias variáveis locais, cujo escopo é minimizado. Tais variáveis são dinâmicas da pilha, de forma que seu armazenamento é alocado quando a seção é alcançada e liberado quando a seção é abandonada. Tais seções de código são chamadas de **blocos**; daí a origem da frase **linguagem estruturada em blocos**.

As linguagens baseadas em C permitem que quaisquer sentenças compostas (uma sequência de sentenças envoltas em chaves correspondentes – {}) tenham declarações e, dessa forma, definam um novo escopo. Tais sentenças compostas são blocos. Por exemplo, se `list` fosse um vetor de inteiros, alguém poderia escrever

```
if (list[i] < list[j]) {
  int temp;
  temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

Os escopos criados por blocos, que podem ser aninhados em blocos maiores, são tratados exatamente como aqueles criados por subprogramas. Referências a variáveis em um bloco e que não estão declaradas lá são conectadas às declarações por meio da busca pelos escopos que o envolvem (blocos ou subprogramas), por ordem de tamanho (do menor para o maior).

Considere o seguinte esqueleto de função em C:

```
void sub() {
  int count;
  ...
  while ( ... ) {
    int count;
    count++;
    ...
  }
  ...
}
```

A referência a `count` no laço de repetição **`while`** é para o `count` local do laço. Nesse caso, o `count` de `sub` é ocultado do código que está dentro do laço **`while`**. Em geral, uma declaração de uma variável efetivamente esconde quaisquer

declarações de variáveis com o mesmo nome no escopo externo maior[10]. Note que esse código é legal em C e C++, mas ilegal e Java e C#. Os projetistas de Java e C# acreditavam que o reúso de nomes em blocos aninhados era muito propenso a erros para ser permitido.

5.5.3 Ordem de declaração

Em C89, como em algumas outras linguagens, todas as declarações de dados em uma função, exceto aquelas em blocos aninhados, devem aparecer no início da função. Entretanto, algumas linguagens – como C99, C++, Java e C# – permitem que as declarações de variáveis apareçam em qualquer lugar onde uma sentença poderia aparecer em uma unidade de programa. Declarações podem criar escopos não associados com sentenças compostas ou subprogramas. Por exemplo, em C99, C++ e Java, o escopo de todas as variáveis locais é de suas declarações até o final dos blocos nos quais essas declarações aparecem. Entretanto, em C# o escopo de quaisquer variáveis declaradas em um bloco é o bloco inteiro, independentemente da posição da declaração, desde que ele não seja aninhado. O mesmo é verdade para os métodos. Note que C# ainda requer todas as variáveis declaradas antes de serem usadas. Logo, independentemente do escopo de uma variável se estender da declaração até o topo do bloco ou subprograma no qual ela aparece, a variável ainda não pode ser usada acima de sua declaração.

As sentenças **`for`** de C++, Java e C# permitem a definição de variáveis em suas expressões de inicialização. Nas primeiras versões de C++, o escopo de tal variável era de sua definição até o final do menor bloco que envolvida o **`for`**. Na versão padrão, entretanto, o escopo é restrito à construção **`for`**, como é o caso de Java e C#. Considere o esqueleto de método:

```
void fun() {
   ...
   for (int count = 0; count < 10; count++){
      ...
   }
   ...
}
```

Em versões posteriores de C++, como em Java e C#, o escopo de `count` é a partir da sentença **`for`** até o final de seu corpo.

[10] Conforme discutido na Seção 5.5.4, em C++ tais variáveis globais ocultas podem ser acessadas no escopo interno usando o operador de escopo (::).

5.5.4 Escopo global

Algumas linguagens, incluindo C, C++, PHP e Python, permitem uma estrutura de programa que é uma sequência de definição de funções, nas quais as definições de variáveis podem aparecer fora das funções. Definições fora de funções em um arquivo criam variáveis globais, potencialmente visíveis a essas funções.

C e C++ têm tanto declarações quanto definições de dados globais. Declarações especificam tipos e outros atributos, mas não causam a alocação de armazenamento. As definições especificam atributos *e* causam a alocação de armazenamento. Para um nome global específico, um programa em C pode ter qualquer número de declarações compatíveis, mas apenas uma definição.

Uma declaração de uma variável fora das definições de funções especifica que ela é definida em um arquivo diferente. Uma variável global em C é implicitamente visível em todas as funções subsequentes no arquivo, exceto aquelas que incluem uma declaração de uma variável local com o mesmo nome. Uma variável global definida após uma função pode ser tornada visível na função declarando-a como externa, como:

```
extern int sum;
```

Em C99, as definições de variáveis globais sempre têm valores iniciais, mas as declarações de variáveis globais nunca têm. Se a declaração estiver fora das definições de função, ela pode incluir o qualificador **extern**.

Essa ideia de declarações e definições é usada também para funções em C e C++, onde os protótipos declaram nomes e interfaces de funções, mas não fornecem seu código. As definições de funções, em contrapartida, são completas.

Em C++, uma variável global oculta por uma local com o mesmo nome pode ser acessada usando o operador de escopo (`::`). Por exemplo, se `x` é uma variável global que está oculta em uma função por uma variável local chamada `x`, a variável global pode ser referenciada como `::x`.

Programas em PHP são geralmente embutidos em documentos XHTML. Independentemente se estiverem embutidos em XHTML ou em arquivos próprios, os programas em PHP são puramente interpretados. Sentenças podem ser interpoladas com definições de funções. Quando encontradas, as sentenças são interpretadas; as definições de funções são armazenadas para referências futuras. As variáveis em PHP são implicitamente declaradas quando aparecem como alvos de sentenças de atribuição. Qualquer variável implicitamente declarada fora de qualquer função é global; variáveis implicitamente declaradas em funções são variáveis locais. O escopo das variáveis globais se estende de suas declarações até o fim do programa, mas pulam sobre quaisquer definições de funções subsequentes. Logo, as variáveis

globais não são implicitamente visíveis em nenhuma função. As variáveis globais podem ser tornadas visíveis em funções em seu escopo de duas formas. Se a função inclui uma variável local com o mesmo nome da global, esta pode ser acessada por meio do vetor `$GLOBALS`, usando o nome da variável como o índice do vetor. Se não existe uma variável local na função com o mesmo nome da global, esta pode se tornar visível com sua inclusão em uma sentença de declaração global. Considere o exemplo:

```
$day = "Monday";
$month = "January";

function calendar() {
  $day = "Tuesday";
  global $month;
  print "local day is $day <br />";
  $gday = $GLOBALS['day'];
  print "global day is $gday <br \>";
  print "global month is $month <br />";
}

calendar();
```

A interpretação desse código produz:

```
local day is Tuesday
global day is Monday
global month is January
```

As regras de visibilidade para variáveis globais em Python não são usuais. As variáveis não são normalmente declaradas, como em PHP. Elas são implicitamente declaradas quando aparecem como alvos de sentenças de atribuição. Uma variável global pode ser referenciada em uma função, mas uma variável global pode ter valores atribuídos a ela apenas se tiver sido declarada como global na função. Considere os exemplos:

```
day = "Monday"

def tester():
  print "The global day is:", day

tester()
```

A saída desse *script*, como as variáveis globais podem ser referenciadas diretamente nas funções, é:

```
The global day is: Monday
```

O *script* a seguir tenta atribuir um novo valor a variável global `day`:

```
day = "Monday"

def tester():
  print "The global day is:", day
  day = "Tuesday"
  print "The new value of day is:", day

tester()
```

O *script* cria uma mensagem de erro do tipo `UnboundLocalError`, porque a atribuição a `day` na segunda linha do corpo da função torna `day` uma variável local – o que a referência a `day` na primeira linha do corpo da função se tornar uma referência ilegal para a variável local.

A atribuição a `day` pode ser para a variável global se `day` for declarada como global no início da função. Isso previne que a atribuição de valores a `day` crie uma variável local, como é mostrado no *script* a seguir:

```
day = "Monday"

def tester():
  global day
  print "The global day is:", day
  day = "Tuesday"
  print "The new value of day is:", day

tester()
```

A saída desse *script* é:

```
The global day is: Monday
The new value of day is: Tuesday
```

A ordem de declaração e as variáveis globais também são problemas que aparecem na declaração e nos membros de classes em linguagens orientadas a objetos. Esses casos são discutidos no Capítulo 12, "Suporte à Programação Orientada a Objetos".

5.5.5 Avaliação do escopo estático

O escopo estático fornece um método de acesso a não locais que funciona bem em muitas situações. Entretanto, isso não ocorre sem problemas. Primeiro, na maioria dos casos, ele fornece mais acesso tanto a variáveis quanto a subprogramas do que o necessário – uma ferramenta pouco refinada para especificar concisamente tais restrições. Segundo, e talvez mais importante, é um problema relacionado à evolução de programas. Sistemas de software são altamente dinâmicos – programas usados regularmente mudam com frequência. Essas mudanças normalmente resultam em reestruturação, destruindo a estrutura inicial que restringia o acesso às variáveis e aos subprogramas. Para evitar a complexidade de manter essas restrições de acesso, os desenvolvedores normalmente descartam a estrutura quando ela começa a atrapalhar. Logo, tentar contornar as restrições do escopo está-

tico pode levar a projetos de programas que mantêm pouca semelhança ao seu original, mesmo em áreas do programa nas quais mudanças não foram feitas. Os projetistas são encorajados a usar muito mais variáveis globais do que o necessário. Todos os subprogramas podem terminar aninhados no mesmo nível, no programa principal, usando variáveis globais em vez de níveis mais profundos de aninhamento[11]. Além disso, o projeto final pode ser artificial e de difícil manipulação, e pode não refletir o projeto conceitual subjacente. Esses e outros defeitos do escopo estático são discutidos em detalhes em Clarke, Wileden e Wolf (1980). Uma alternativa ao uso do escopo estático para controlar o acesso às variáveis e aos subprogramas é o uso de construções de encapsulamento, incluídas em muitas das novas linguagens. Construções de encapsulamento são discutidas no Capítulo 11, "Tipos Abstratos de Dados e Construções de Encapsulamento".

5.5.6 Escopo dinâmico

O escopo de variáveis em APL, SNOBOL4 e nas primeiras versões de LISP era dinâmico. Perl e COMMON LISP também permitem que as variáveis sejam declaradas com escopo dinâmico, apesar de o mecanismo de escopo padrão dessas linguagens ser estático. O **escopo dinâmico** é baseado na sequência de chamadas de subprogramas, não em seu relacionamento espacial uns com os outros. Logo, o escopo pode ser determinado apenas em tempo de execução.

Considere mais uma vez o procedimento `Big` da Seção 5.5.1:

```
procedure Big is
  X : Integer;
  procedure Sub1 is
  X : Integer;
    begin  -- de Sub1
    ...
    end;  -- de Sub1
  procedure Sub2 is
    begin  -- de Sub2
     ...X...
    end;  -- de Sub2
  begin  -- de Big
  ...
  end;  -- de Big
```

Assuma que as regras de escopo dinâmico se aplicam a referências não locais. O significado do identificador `X` referenciado em `Sub2` é dinâmico – ele não pode ser determinado em tempo de compilação. Ele pode referenciar a qualquer uma das declarações de `X`, dependendo da sequência de chamadas.

Uma maneira pela qual o significado correto de `X` pode ser determinado em tempo de execução é iniciar a busca com as variáveis locais. Essa também

[11] Parece muito com a estrutura de um programa em C, não parece?

é a maneira pela qual o processo começa no escopo estático, mas é aqui que a similaridade entre os dois tipos de escopo termina. Quando a busca por declarações locais falha, as declarações do pai dinâmico, o procedimento que o chamou, são procuradas. Se uma declaração para `X` não é encontrada lá, a busca continua no pai dinâmico desse procedimento chamador, e assim por diante, até que uma declaração de `X` seja encontrada. Se nenhuma for encontrada em nenhum ancestral dinâmico, ocorre um erro em tempo de execução.

Considere as duas sequências de chamadas diferentes para `Sub2` no exemplo anterior. Primeiro, `Big` chama `Sub1`, que chama `Sub2`. Nesse caso, a busca continua a partir do procedimento local, `Sub2`, para seu chamador, `Sub1`, onde uma declaração de `X` é encontrada. Logo, a referência a `X` em `Sub2`, nesse caso, é para o `X` declarado em `Sub1`. A seguir, `Sub2` é chamado diretamente por `Big`. Nesse caso, o pai dinâmico de `Sub2` é `Big`, e a referência é para o `X` declarado em `Big`.

Note que se o escopo estático fosse usado, em qualquer uma das sequências de chamadas discutidas, a referência a `X` em `Sub2` seria o `X` de `Big`.

O escopo dinâmico em Perl não é usual – na verdade, ele não é exatamente conforme discutimos nesta seção, apesar de a semântica ser geralmente ser aquela do escopo dinâmico tradicional (veja o Exercício de Programação 1).

5.5.7 Avaliação do escopo dinâmico

O efeito do escopo dinâmico na programação é profundo. Os atributos corretos das variáveis não locais visíveis a uma sentença de um programa não podem ser determinados estaticamente. Além disso, uma referência ao nome de tal variável nem sempre é para a mesma. Uma sentença em um subprograma que contém uma referência para uma variável não local pode se referir a diferentes variáveis não locais durante diferentes execuções do subprograma. Diversos tipos de problemas de programação aparecem por causa do escopo dinâmico.

Primeiro, durante o período de tempo iniciado quando um subprograma começa sua execução e terminado quando a execução é finalizada, as variáveis locais do subprograma estão todas visíveis para qualquer outro subprograma sendo executado, independentemente de sua proximidade textual ou de como a execução chegou ao subprograma que está executando atualmente. Não existe uma forma de proteger as variáveis locais dessa acessibilidade. Os subprogramas são *sempre* executados no ambiente de todos os subprogramas previamente chamados que ainda não completaram suas execuções. Assim, o escopo dinâmico resulta em programas menos confiáveis do que aqueles que usam escopo estático.

Um segundo problema com o escopo dinâmico é a impossibilidade de verificar os tipos das referências a não locais estaticamente. Esse problema resulta da impossibilidade de encontrar estaticamente a declaração para uma variável referenciada como não local.

O escopo dinâmico também faz os programas muito mais difíceis de serem lidos, porque a sequência de chamadas de subprogramas deve ser conhecida para determinar o significado das referências a variáveis não locais. Essa tarefa é praticamente impossível para um leitor humano.

Por fim, o acesso a variáveis não locais em linguagens de escopo dinâmico demora muito mais do que acesso a não locais quando o escopo estático é usado. A razão para isso é explicada no Capítulo 10.

Por outro lado, o escopo dinâmico tem seus méritos. Em alguns casos, os parâmetros passados de um subprograma para outro são variáveis definidas no chamador. Nada disso precisa ser passado em uma linguagem com escopo dinâmico, porque elas são implicitamente visíveis no subprograma chamado.

Não é difícil entender por que o escopo dinâmico não é tão usado como o escopo estático. Os programas em linguagens com escopo dinâmico são mais fáceis de ler, mais confiáveis e executam mais rapidamente do que programas equivalentes em linguagens com escopo dinâmico. Por essas razões, o escopo dinâmico foi substituído pelo escopo estático na maioria dos dialetos atuais de LISP. O Capítulo 10 discute métodos de implementação tanto para escopo estático quanto para escopo dinâmico.

5.6 ESCOPO E TEMPO DE VIDA 

Algumas vezes, o escopo e o tempo de vida de uma variável parecem ser relacionados. Por exemplo, considere uma variável declarada em um método Java que não contém nenhuma chamada a método. O escopo dessa variável é de sua declaração até o final do método. O tempo de vida dessa variável é do período de tempo que começa com a entrada no método e termina quando a execução do método chega ao final. Apesar de o escopo e o tempo de vida da variável serem diferentes, devido ao escopo estático ser um conceito textual, ou espacial, enquanto o tempo de vida é um conceito temporal, eles ao menos parecem ser relacionados nesse caso.

Esse relacionamento aparente entre escopo e tempo de vida não se mantém em outras situações. Em C e C++, por exemplo, uma variável declarada em uma função usando o especificador **`static`** é estaticamente vinculada ao escopo dessa função e ao armazenamento. Logo, seu escopo é estático e local à função, mas seu tempo de vida se estende pela execução completa do programa do qual ela é parte.

O escopo e o tempo de vida também não são relacionados quando chamadas a subprogramas estão envolvidas. Considere as funções C++ de exemplo:

```
void printheader() {
  ...
}  /* Fim de printheader */
```

```
void compute() {
  int sum;
  ...
  printheader();
 }  /* Fim de compute */
```

O escopo da variável `sum` é completamente contido na função `compute`. Ele não se estende ao corpo da função `printheader`, apesar de `printheader` executar no meio da execução de `compute`. Entretanto, o tempo de vida de `sum` se estende por todo o período em que `printheader` é executada. Qualquer que seja a posição de armazenamento com a qual `sum` está vinculada antes da chamada a `printheader`, esse vínculo continuará durante e após a execução de `printheader`.