第五讲　符号表

2020-10

1. 符号表的作用

符号表是编译程序用到的最重要的数据结构之一，几乎在编译的每个阶段每一遍都要涉及到符号表。

符号表自创建后便开始被用于收集符号（标识符）的属性信息，不同阶段会有不同的信息。例如，编译程序在分析处理到下述两个说明语句

```
int A ;
float B[5] ;
```

则在符号表中收集到关于符号 A 的属性是一个整型变量，关于符号 B 的属性是具有 5 个浮点型元素的一维数组。

在语义分析中，符号表所登记的内容是进行上下文语义合法性检查的依据。同一个标识符可能在程序的不同地方出现，而有关该符号的属性是在不同情况下收集的，特别是在多遍编译及程序分段编译（以文件为单位）的情况下，更需检查标识符属性在上下文中的一致性和合法性。通过符号表中属性记录可进行这些语义检查。例如： 在 C 语言中同一个标识符可作引用说明也可作定义说明：

```
┇
int i [3, 5]；//　定义说明 i
┇
extern float i;//　引用说明 i
┇
```

按编译过程，符号表中首行先建立标识符 i 的属性是 3 × 5 个整型元素的数组，而后在分析第二个说明时标识符属性是浮点型简单变量。通过符号表的语义检查可发现其不一致错误。

又例如，在一个 C 语言程序中出现

```
┇
int i [3, 5]；
┇
float i [4, 2]；
┇
int i [3, 5]；
┇
```

编译过程首先在符号表中记录了标识符 i 的属性是 3 × 5 个整型元素的数组，而后在分析第二、第三这两个定义说明时编译系统可通过符号表检查出标识符 i 的二次重定义冲突错误。本例还可以看到不论在后二句中 i 的其它属性与前一句是否完全相同，只要标识符名重定义，就将产生重定义冲突的语义错误。

在目标代码生成阶段，符号表是对符号名进行地址分配的依据。程序中的变量符号由它被定义的存储类别和被定义的位置等来确定将来被分配的存储位置。首先要根据存储类别确

定其被分配的存储区域。例如，在C语言中需要确定该符号变量是分配在公共区（extern）、文件静态区（extern static）、函数静态区（函数中的 static）、还是函数运行时的动态区（auto）等。其次是根据变量出现的次序（一般来说是先声明的在前）来决定该变量在某个区中所处的具体位置，这通常使用在该区域中相对于起始位置的偏移量来确定。而有关区域的标志及相对位置都是作为该变量的语义信息被收集在该变量的符号表属性中。

此外，符号表的组织与结构还需要体现符号的作用域与可见性信息。

2. 符号的常见属性

符号表中可以存放不同的属性，以便在编译的不同阶段使用。不同的语言定义的标识符属性不尽相同，但下列几种通常都是需要的。

- 符号的名字。这是每个符号不可缺少的属性。在符号表中，符号名常用作查询相应表项的键字，一般不允许重名。根据语言的定义，程序中出现的重名标识符定义将按照该标识符在程序中的作用域和可见性规则进行相应的处理。在一些允许操作重载的语言中，函数名、过程名是可以重名的，对于这类重载的标识符要通过它们的参数个数和类型以及函数返回值类型来区别，以达到它们在符号表中的唯一性。
- 符号的类别。比如，符号可分为常量符号，变量符号，过程/函数符号，类名符号等不同的类别。
- 符号的类型。各类符号一般会有类型，如常量符号、变量符号对应有数据类型。函数/过程符号也可以有类型（比如由参数类型和返回值类型复合而成的函数类型）。符号的类型属性决定了该符号所标识的内容在存储空间的存储格式，还决定了可以对其施加的运算操作。
- 符号的存储类别和存储分配信息。存储类别信息如：该符号对应的存储是在数据区还是代码区，是在静态数据区还是动态数据区，是动态分配在栈区还是在堆区，等等。存储分配信息如：该符号数据单元的大小（字节数），相对某个存储基地址的偏移位置，等等。
- 符号的作用域与可见性。（参见第 4 节）

其他属性。体现不同数据对象的符号属性特征可能差异较大，可以将这些属性分开组织。如对于数组，符号的属性包含数组内情向量；对入结构体或类，则符号的属性应包含成员信息；对于函数/过程，需要包含形参的信息。

3. 符号表的实现

和其它关于表的数据结构类似，针对符号表的操作通常包含：

- 创建符号表。通常在编译开始或进入一个作用域时调用创建符号表操作。
- 插入表项。在遇到新的符号声明时进行，通常是插入到当前作用域所对应的符号表。
- 查询表项。在引用符号时进行。
- 修改表项。在获得新的语义值信息时进行。

- 删除表项。在符号成为不可见/不再需要它的任何信息时进行。
- 释放符号表空间。在编译结束前或退出一个作用域时进行。

符号表的实现需要选择适当的数据结构，除了需要体现符号表的功能和作用，通常也需要考虑符号表操作的方便性和高效性，有时还需要考虑节省内存空间（如某些运行在低端嵌入设备的即时编译程序会有这样的需求）。

以下是实现符号表的几种常用数据结构：

- 一般的线性表。如：数组，链表，等。
- 有序表。访问时较无序表快，比如可以使用折半查找算法。
- 二叉搜索树。
- Hash 表。

实现高效的符号表组织与管理对于编译程序来说非常重要，因为它在各阶段都要被频繁访问。但本书的重点不在于此，故不对此进行更深入的讨论。

最后，我们简要讨论一下编译器何时创建符号表。符号表至少应该在静态语义分析之前已经创建，最常见的情况是在语法分析的同时创建。如果词法分析程序单独作为一遍，则一般不可能承担符号表创建的任务（因为不能获得作用域信息）。如果词法分析程序是被语法分析器调用，则符号表的相应表项既可由词法分析程序也可由语法分析程序写入。通常是后者，因为可以在同时写入更多的属性信息，并且知道是否是正在声明的符号（如果是，就创建新的表项，否则只是更新表项的属性）。当由词法分析程序写入时，则加入到当前作用域对应的符号表中（符号表指针需要语法分析程序告知），可以包含符号名、属性值、位置信息等。符号表在语法分析之后而在语义检查之前创建也是很常见的，这种方法容易获得符号的更多属性，也容易处理同一作用域内随处声明的符号。

4. 符号表体现作用域与可见性

充分理解符号表与作用域之间的关系，是这一讲应该重点掌握的内容。

我们先来讨论一下作用域与可见性的问题。每一个符号在程序中都有一个确定的有效范围。拥有共同有效范围的符号所在的程序单元就构成了一个**作用域**（*scope*）。作用域之间可以嵌套，即一个作用域可以被另一个作用域包围，称为嵌套的作用域（*nested scopes*）。但作用域之间不会交错，也就是说，两个作用域要么嵌套（一个包含另一个），要么不相交。相对于程序的某一特殊点而言，该点所在的作用域称为**当前作用域**。当前作用域与包含它的程序单元所构成的作用域称为**开作用域**（open scopes）。不属于开作用域的作用域称为**闭作用域**（close scopes）。**可见性**（visibility）是指在程序的某一特定点，哪些符号是可访问的（即可见的）。程序语言中常用的可见性规则（visibility rules）为：

- 在程序的任何一点，只有在该点的开作用域中声明的符号才是可访问的。
- 若一个符号在多个开作用域中被声明，则把离该符号的某个引用最近的声明作为该引用的解释。
- 新的声明只能出现在当前作用域。

值得注意的是，这里讲到的“作用域”是指在静态作用域规则下的含义。多数常用语言都是采用静态作用域规则。

多数情况下，每个作用域都有各自的符号表，我们称之为**多符号表组织**。但我们也可以使所有嵌套的作用域共用一个全局符号表，称之为**单符号表组织**。

4.1 作用域与单符号表组织

通常，单符号表组织具有以下特点：

- 所有嵌套的作用域共用一个全局符号表。
- 每个作用域都对应一个作用域号
- 仅记录开作用域中的符号。
- 当某个作用域成为闭作用域时，从符号表中删除该作用域中所声明的符号。

下面，我们举一个单符号表组织的例子。该例子中，我们采用一个 Hash 表的结构来组织全局符号表。

如下是某语言（PL/0）的一段代码：

```
const a=25;
var x,y;
procedure    p;
      var    z;
      begin
            ……
      end;
procedure    r;
      var x, s;
      procedure t;
            var    x;
            begin
                ……    /*here*/
            end;
      begin
            ……
      end;
begin
      ……
end.
```

当处理到程序位置 /*here*/ 时，符号表的当前状态如图 1 所示。

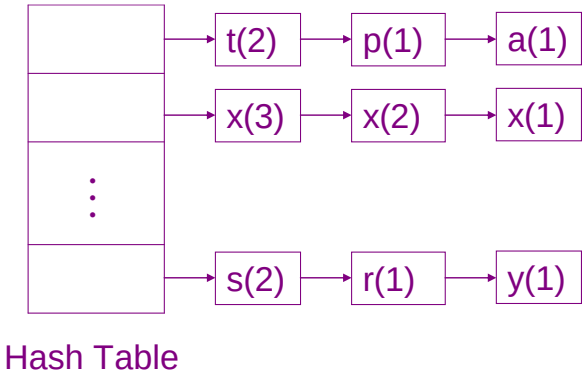

图 1　单符号表组织

图 1 中，各符号的散列值是我们随意假设的。针对程序位置 /*here*/，当前的开作用域包括：第 1 层的全局作用域，含符号 a(1), x(1), y(1), p(1), r(1)；第 2 层的由过程 r 所辖的作用域，含符号 x(2), s(2), t(2)；以及第 3 层的由过程 t 所辖的作用域，含符号 x(3)。这里，各符号所附加的数字代表符号所在的层次号。注意，图 1 的符号表中不包含闭作用域中的符号，如过程 p 所辖的作用域（第 2 层）中的符号 z 没有出现在符号表中。

在图 1 的符号表中插入一个符号对应的表项时，我们假定是插入到各分表的表头位置。这样，当某个符号在符号表中出现多个副本时，那么离该符号的某个引用最近声明的副本应作为该引用的解释，它应该处于分表中最靠前的位置。例如，在程序位置 /*here*/处，引用符号 x，其含义是指向符号 x(3) 所对应的表项。

图 2 是单符号表组织的另一个例子，这次我们不采用 Hash 表结构，而采用线性表结构。同时，我们展示了这一全局符号表中表项属性的设计，以接近实际的应用。

图 2 中，符号表项的 KIND 域标识符号的种类（变量符号 VARIABLE，常量符号 CONSTANT，过程符号 PROCEDURE）；VAL/LEVEL 域对于常量符号来讲标识是具体的数值，对变量或过程符号来讲是指声明这些符号时的层号（如果最外层声明的层号为 LEV，那么过程 r 内部声明的层号就为 LEV+1，过程 t 内部声明的层号就为 LEV+2）；ADDR 域仅对变量符号有效，DX+n 中的 n 表示该变量在分配存储时相对于该过程存储区基址的偏移量（我们不考虑变量的类型，并假设每个变量占一个单元的存储；若当前过程基址为 DX，则该变量的存储地址将是 DX+n）；SIZE 域仅对过程符号有效，标识该过程在被调用（激活）时，应该申请的初始化空间的大小（CX 表示过程活动记录中控制单元的数目，因此 SIZE 的取值是该过程所声明局部变量的数目加上 CX）。

当处理到图 2 右边的程序位置 /*here*/ 时，某个符号是可见的，当且仅当它对应的表项出现在符号表中（这是一个全局符号表）。实际上，所有嵌套的作用域共享着这张符号表（嵌套层次即为层号 LEVEL）。图 2 所表示的符号表状态实际上刻画了当前的开作用域有两个，一个作用域对应的层号为 LEV，另一个作用域对应的层号为 LEV+1。

右边程序在处理到*here*/时
的符号表如下所示
（请注意：作用域是如何体现的）

NAME	KIND	VAL / LEVEL	ADDR	SIZE
a	CONSTANT	25		
x	VARIABLE	LEV	DX	
y	VARIABLE	LEV	DX+1	
p	PROCEDUR	LEV		CX+1
r	PROCEDUR	LEV		CX+2
x	VARIABLE	LEV+1	DX	
s	VARIABLE	LEV+1	DX+1	
t	PROCEDUR	LEV+1		CX+3

```
const a=25;
var x,y;
 procedure  p;
   var  z;
   begin
      ……
   end;
 procedure  r;
   var x, s;
    procedure t;
       var   v, x, y;
       begin
          ……
       end;
       begin    /*here*/
       ……
   end;
begin
   ……
end.
```

图 2　某编译器的线性单符号表组织与作用域的关系

4.2 作用域与多符号表组织

通常，多符号表组织具有以下特点：

- 每个作用域都有各自的符号表。
- 需要维护一个作用域栈，每个开作用域对应栈中的一个入口，当前的开作用域出现在该栈的栈顶
- 当一个新的作用域开放时，新符号表将被创建，并将其入栈。
- 在当前作用域成为闭作用域时，从栈顶弹出相应的作用域。

下面，我们举一个多符号表组织的例子。该例子使用如下代码段：

```
const a=25;
var x,y;
procedure    p;
      var    z;
      begin
            ……
      end;
procedure    r;
      var x, s;
      procedure t;
            var    v;
```

```
        begin
            ……
        end;
    begin              /*here*/
        ……
    end;
begin
    ……
end.
```

当处理到程序位置 /*here*/ 时，符号表的当前状态如图 3 所示。

从图 3 可见，这一代码段包含 4 个作用域，分别对应符号集合{a, x, y, p, r}，{x, s, t}，{z}，以及 [v]。这里，前两个作用域为开作用域，它们都出现在当前的作用域栈上；后两个作用域为闭作用域，不出现在当前的作用域栈上。

当某个符号在开作用域中出现多次，那么离该符号的某个引用最近声明的副本应作为该引用的解释，它应该是指离栈顶最近的作用域中的符号。例如，在程序位置 /*here*/处，引用符号 x，其含义是指作用域 {x, s, t} 中的符号 x。

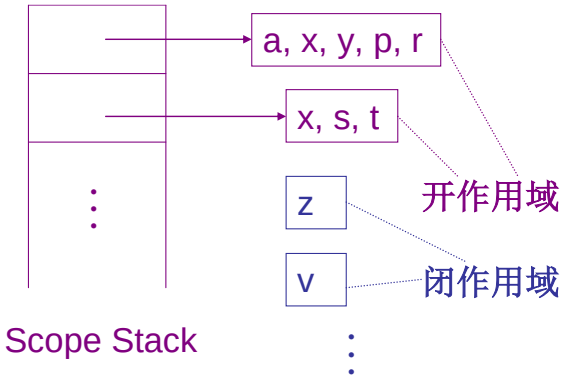

图 3　多符号表组织

课后作业

1　（可忽略）对于下列 Decaf/Mind 程序，根据第二阶段实验建立符号表的过程，当处理到图中所标出的 Print 语句时，当前作用域栈中含哪些开作用域？它们对应的符号表中分别包含哪些符号？

```
class Fruit
{
    int price;
    string name;
    void init(int p, string s)  {price=p; name=s;}
    void print(){ Print(” The price of ", name, " is ",price,"\n");}
}
class Apple extends Fruit
{
    string color;
    void setcolor(string c)  {color=c;}
    void print(){
        Print(“The price of ",color," ",name," is ", price,"\n");
    }
}
class Main {
{
    static void main() {
        class Apple a;
        a=new Apple();
        a.setcolor("red");
        a.init(100,"apple");
        a.print();
    }
}
```

2 （可忽略）对于下列 Decaf/Mind 程序，根据第二阶段实验所建立的符号表过程，当处理到图中所标出的 if 语句时，（1）当前作用域对应的符号表中包含哪些符号？（2）所有开作用域有哪些？它们对应的符号表中分别包含哪些符号

```
class Math {

    static int pow(int a, int b) {
        int i;
        int result;
        ……
    }

    static int log(int a) {
        if (a < 1) {
            return -1;
        }
        int result;
        result = 0;
        while (a > 1) {
            result = result + 1;
            a = a / 2;
        }
        return result;
```

```
        }
    }

    class Main {
        static void main() {
            ……
        }
    }
```

3 PL0 编译器的符号表如 4.1 节所述的那样，采用一个全局的单符号表栈结构。对于下列的 PL0 程序片断，当 PL0 编译器在处理到第一个 call p 语句（第 7 行）以及第二个 call p 语句（第 t 行，即过程 q 的第 4 行）时，试分别列出每个开作用域中的符号。

```
（1） var a,b;
（2） procedure   p ;
（3）         var s;
（4）         procedure   r ;
（5）                 var v;
（6）                 begin
（7）                         call p;
  .                                 ……
  .                         end;
  .          begin
  .                  If   a < b   then   call r ;
  .                         ……
  .          end ;
  .    procedure   q ;
  .          var x,y;
  .          begin
（t）                  call p ;
  .                         ……
  .          end ;
  .    begin
  .          a := 1;
  .          b := 2;
  .          call q;
  .                 ……
  .    end .
```

4 阅读 PL/0 编译程序的符号表数据结构（struct tablestruct table[tmax]）以及查表（int positon(…)）、添加符号（void enter (…)）等操作 ，读懂与符号表相关的代码，包括理解作用域是如何体现的（注意函数 void enter (…) 的形参 lev）。

（非书面作业，可选）