第四讲　符号表

2024-10

第四讲的基本要求以课堂讲稿（ppt）为准，本文档中不属于 slide04.ppt 的内容均作为这一讲的选讲内容。

1 符号表的作用

高级语言程序中有许多符号（*symbols*）或者名字（*names*）对应于标识符（*identifiers*），用来标识各种程序对象，如变量（*variables*）、类型（*types*）、类（*classes*）、函数（*functions*），等等。这些标识符的基本属性及其在程序中的上下文环境属性，能够充分体现程序的语义信息。

符号表是编译设计中最重要的数据结构之一，用于登记和追踪所声明（*declarations*）的符号及其所绑定的信息。几乎在编译的每个阶段每一遍都要涉及到符号表。

符号表的组织与管理是编译程序的重要组成部分，负责符号表的创建和维护，以声明内容和位置等信息为基础，收集和查询标识符的基本属性，同时体现标识符在程序中的上下文环境，例如作用域（*scope*），可见性（*visibility*）和生存期（*lifetimes*）等。

符号表自创建后便开始被用于收集符号（标识符）的基本属性，不同阶段会有不同的信息，后续根据需要进一步更新和维护。例如，编译程序分析处理到下述两个声明语句：

```
float A ;
int B[10] ;
```

则在符号表中收集到关于符号 A 的属性中表明该符号是一个浮点型变量，关于符号 B 的属性中表明该符号是具有 10 个整型元素的一维数组。

编译程序的前端需要对程序的合法性进行检查。语法（含词法）是否合法主要是通过检查上下文无关文法（含正规文法）的符合性，而上下文无关文法不可能刻画程序的上下文相关特性（或称上下文语义合法性），后者通常是在语义分析阶段进行检查。

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别编译（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。可以通过符号表中所记录的属性进行这些语义检查。例如，在 C 语言中对同一个标识符可作引用声明也可作定义声明：

```
┆
int x[2][5];     //   定义声明 x
┆
extern float x;     //   引用声明 x
┆
```

按编译过程，符号表中先设置标识符 x 的属性，表明该符号是一个含有 2× 5 个整型元素的二维数组变量，而后在分析另一个声明语句时，发现标识符 x 是一个浮点型简单外部变量（位

于不同文件）。这样，通过基于符号表的语义检查可发现二者类型不一致的语义错误。

作用域（*scope*），通常指静态作用域（*static scope*）或词法作用域（*lexical scope*），是与符号表密切相关的静态程序结构。作用域内部可以声明一个或多个变量或其他类别的标识符。以作用域为单位构建符号表是常规选项。同一个作用域中一般不允许标识符被多重定义，例如，在同一个作用域所涵盖的程序片段中若包含

```
┆
int y [2, 5];
┆
float y [4, 3];
┆
int y [2, 5];
┆
```

编译过程首先在符号表中记录了标识符 y 是 2× 5 个整型元素的数组，而后在分析该变量的第二、第三后两个定义声明时编译系统可通过符号表检查出标识符 y 的两次重定义冲突错误。只要标识符名重定义，不论在 y 的定义声明与前面是否完全相同，一般都会报告重定义冲突的语义错误。

许多语言中，作用域是可以嵌套的。嵌套作用域中，变量（或其他标识符）的可见性会遵循一定的规则，如最内嵌套作用域的可见性规则。在实现符号表的组织与管理时应能够体现所遵循的规则。

通过符号表中符号的属性信息还可以检查在上下文中符号使用的合法性。例如，对于下列代码片段：

```
┆
int foo() { …; return 1; }
┆
int main() { …;   return foo (1); }
┆
```

编译程序在符号表中记录了标识符 foo 是一个无参函数，然而在处理 main 函数 return 语句的表达式时，发现 foo 的调用使用了参数，可以报告函数调用的实参数量与函数声明的形参数量不匹配的语义错误。

除了借助符号表实现语义分析和静态检查，如上述检查符号声明的一致性以及使用的合法性，编译程序所采用的运行时存储策略以及代码生成过程也要频繁访问符号表中符号的各种属性，涉及符号表的组织与管理机制。

程序中的变量符号由它被定义的存储类别、类型和被定义的位置等来进一步确定将来被分配的运行时存储空间。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（文件中最外层的 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。还有，变量的类型用于确定其分配的存储空间大小（字节数）。而有关存储区域的标志、类型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。

程序变量的作用域与可见性属性，体现在程序执行期间就对应于该变量的生存期（*lifetimes*）。比如，在嵌套作用域最外层声明的变量，其生存期会贯穿于程序执行的全过程，而对于内层声明的非静态局部变量，其生存期则是不连续的（即不同实例的分段生存期）。编译程序借助于符号表的组织与管理机制获取标识符在程序中的上下文环境属性，得到作用域、可见性和生存期等相关信息。

此外，符号表在与编译程序密切相关的伙伴程序（用于程序开发工具/环境）中也可能会用到。比如，调试程序需要使用标识符在源程序中的定位信息（行号/列号），可将其记录在符号表中该标识符的相关属性中。

2 符号的常见属性

符号表中可以存放符号的不同属性，以便在编译的不同阶段使用。因源语言以及实现方式的多样性，所涉及的标识符属性信息不尽相同，有些适合直接在符号表中登记的信息，有些可以通过已登记的信息间接得到，还有一些可以通过符号表的组织与管理来体现。以下列举几种常见的符号属性。

- 符号的名字

名字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，在共同的上下文环境中一般不允许重名。

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。同一作用域中的符号通常是不允许重名的，若存在多个作用域共享符号表的情形，需要通过作用域来区分同名标识符。

在一些允许运算重载的语言中，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所对应表项的唯一性。

- 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。类别不同，符号表中其他属性的组成和含义通常也会有所差异。当体现不同数据对象的属性特征差异较大时，可将这些属性以分表的形式分开组织。如对于数组，符号的属性通常会包含数组的内情向量；对于结构体或类，符号的属性应包含成员信息；对于函数/过程，则需要包含形参的信息。

- 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/过程符号也对应有由参数类型和返回值类型复合而成的函数类型。由类（class）声明的实例变量符号拥有相应 class 所定义的类型。

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。某些拥有高阶类型的符号，可以对应存储空间及其运算操作的模板，例如C++的模板（template）可以声明一个拥有参数化类型的类。

有时，符号的类型需要根据使用的上下文借助类型推导才能获得。

另外，符号表中所记录的是符号的静态类型。有些符号在实际运行时的类型（动态类型）是随进行环境变化的，一般可以兼容符号表中所登记的静态类型，比如，运行时的动态类型是该静态类型的子类型。

- 符号的存储类别和存储分配信息

符号的存储类别信息，用于决定该符号对应的存储是在数据区还是代码区，数据区是静态分配还是动态分配，静态分配具体是在哪一块静态数据区，动态分配是在栈区还是在堆区，等等。

符号的存储分配信息，如符号的数据单元大小（字节数），相对某个存储基地址的偏移量，函数/过程符号的初始化栈帧空间（活动记录）的大小，等等。

- 符号在程序中的上下文环境信息

符号的上下文环境属性，比如，符号在程序中的行/列位置，相对某个存储基地址的偏移量（也见符号的存储分配信息），以及作用域、可见性和生存期等信息。作用域与可见性是符号在上下文环境中的基本属性，一般是体现在符号表的组织与管理中，同时通过存储分配和代码生成环节体现在程序执行期间该符号的生存期。

上述属性中，有些基本属性可以直接从符号的声明中得到。例如，当遇到 C/C++ 语言程序中的如下声明语句：

```
static int x[2][5];
```

编译程序会在当前符号表中添加符号名为 x 的表项，并直接登入其类别属性为变量，类型属性为整型数组，且进一步借助内情向量（*dove vector*）信息栏将其登记为拥有两个维度的上界 2 和 5 的二维数组，同时标记其具有 static 属性。

有时，可以从声明语句中间接得到有用的信息，必要时也可以作为符号属性或附加信息记录在符号表中。如，从以上声明中的上下界信息，可以计算出数组 x 的元素个数（10）以及存储字节数（以 32 位整数为例，需分配 40 个字节的存储空间），以及其他有用的内情向量信息。这些派生出来的信息，可在后续阶段（如代码生成）直接使用。

根据上述声明语句，x 具有 static 属性，可知该数组将会分配在静态数据区。然而，具体是哪一静态数据区，则需要从上述 x 的声明在程序中的上下文环境推断。若是声明在源文件（.c 或者.cpp 文件）内的 static 全局变量，则会将 x 分配在文件静态数据区。若是声明在函数/方法内的 static 局部变量，则会将 x 分配在相应的函数静态数据区。若是声明在某个类中的 static 数据成员（或称 x 为类变量），则会将 x 分配在相应的类静态数据区，在该类的实例对象中不会为 x 分配对象存储空间。

在编译的不同阶段，符号表中的属性信息会有所不同。比如，在编译后端，可能包含符号的寄存器之类的低层次属性。

作用域与可见性对符号表的组织和管理影响较大。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。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）。

例如，对如下 C 代码片段 Prog1：

```
int a = 0;

int foo (int a) {
    int b = 1;
    return a + b;
}

int main ()
{
     int a=1, b=1, c;
     …                                  /*  不含其他声明语句  */
     {                                   /*  语句块 1  开始  */
          int b=0, c;
           …                                  /*  不含其他声明语句  */
     }                                   /*  语句块 1  结束  */
     {                                   /*  语句块 2  开始  */
          int d;
           …                                  /*  不含其他声明语句  */
          {                                    /*  语句块 3  开始  */
               int b=2, e;
               e = foo (a+b);                  /* here */
          }                                    /*  语句块 3  结束  */
     }                                   /*  语句块 2  结束  */
 }
```

对于代码片段 Prog1，可以认为包含如下 8 个静态作用域：（1）全局作用域，辖符号集合{a, foo, main}；（2）foo 函数形参作用域，辖符号集合{a}；（3）foo 函数体作用域，辖符号集合{b}；（4）main 函数形参作用域，辖符号集合{}；（5）main 函数体作用域，辖符号集合{a, b, c}；（6）main 函数内语句块 1 作用域，辖符号集合{b, c}；（7）main 函数内语句块 2 作用域，辖符号集合{d}；（8）main 函数内语句块 3 作用域，辖符号集合{b, e}。

在代码片段 Prog1 的/*here*/处，可见的符号 a 是 main 函数体作用域中的 a，而全局作用域中的 a 是不可见的符号。

针对如 Prog1 这种具有块结构静态嵌套作用域的语言，其符号表的组织与管理有经典的方法，本讲后续将专门讨论。

再看一个例子。设有如下简单面向对象语言（类 Java）程序片段 Prog2：

```
class Computer {
    int cpu;
    void Crash(int numTimes) {
        int i;
        for (i = 0; i < numTimes; i = i + 1)
            Print("sad\n");
    }
```

```
}

class Mac extends Computer {
    int mouse;
    void Crash(int numTimes) {
        Print("ack!");                                    /* here */
    }
}

class Main {
    static void main() {
        class Mac powerbook;
        powerbook = new Mac();
        powerbook.Crash(2);
    }
}
```

对于代码片段 Prog2，可划分如下 10 个静态作用域：（1）全局作用域，辖符号集合{Computer, Mac, Main}；（2）类 Computer 作用域，辖符号集合{cpu, Crash}；（3）类 Computer 方法 Crash 形参作用域，辖符号集合{this[1], numTimes}；（4）类 Computer 方法 Crash 的方法体作用域，辖符号集合{i}；（5）类 Mac 作用域，辖符号集合{mouse, Crash}；（6）类 Mac 方法 Crash 形参作用域，辖符号集合{this, numTimes}；（7）类 Mac 方法 Crash 的方法体作用域，辖符号集合{}；（8）类 Main 作用域，辖符号集合{main}；（9）类 Main 方法 main 形参作用域，辖符号集合{}[2]；（10）类 Main 方法 main 的方法体作用域，辖符号集合{powerbook}。

在代码片段 Prog2 的/*here*/处，由于继承，类 Computer 作用域中的符号 cpu 是可见的；而由于重载，类 Computer 作用域中的符号 Crash 是不可见的。

在代码片段 Prog2 类 Main 的 main 方法中，虽然最后一行 powerbook.Crash(2)有方法 Crash 的调用，但不能认为类 Mac 作用域的符号 Crash 是可见的。这里，可见的符号是 powerbook，powerbook.Crash(2)是发消息 Crash(2) 给 powerbook。

3 符号表的结构与实现

符号表中的各个表项均对应唯一的符号。关于表项内容的组织，可以采用单级或者多级的符号表结构。下面借助例子予以说明。

单级的符号表结构，是指符号表表项的属性不会关联其他的符号表，这意味着表项中描述相对独立的符号属性。例如，对于第 2 节的代码片段 Prog1，我们为每个作用域设计一个符号表，它们是结构相似的单级符号表。图 1，图 2 和图 3 分别对应全局作用域（辖符号集合{a, foo, main}），main 函数体作用域（辖符号集合{a, b, c}）和 main 函数内语句块 3 作用域（辖符号集合{b, e}）的符号表结构。

---

[1] 注：this 变量，以当前类作为类型，为隐式形参。参考 C++或 Java 中的 this 变量。

[2] 注，main 是静态方法，不含隐式形参 this。

名字	嵌套层号	类别	类型	偏移量	初始栈帧空间
a	LEV	variable	int	DX	
foo	LEV	function	int		CX+2
main	LEV	function	int		CX+6

图 1　符号表结构示例（1）

名字	嵌套层号	类别	类型	偏移量	初始栈帧空间
a	LEV+1	variable	int	DX	
b	LEV+1	variable	int	DX+1	
c	LEV+1	variable	int	DX+2	

图 2　符号表结构示例（2）

名字	嵌套层号	类别	类型	偏移量	初始栈帧空间
b	LEV+3	variable	int	DX+4	
e	LEV+3	variable	int	DX+5	

图 3　符号表结构示例（3）

图 1，图 2 和图 3 所刻画的符号表，采用了一致的符号属性。对于不同“类别”的符号，可能某些属性不使用，留作空白。例如，如“偏移量”属性对 variable 类别的符号有效，而对 function 类别的符号无效。类似地，“初始栈帧空间”属性仅适用于 function 类别的符号。“偏移量”属性中，DX 表示当前栈帧中数据信息基址的偏移量，目标代码生成时可经专用寄存器换算得到。

“初始栈帧空间”属性中，CX 表示函数栈帧所需要的控制单元数目，且此例中假设 int 型数占用 1 个栈帧存储单元。函数 foo 的初始栈帧大小 CX+2，参数 a 和局部变量 b 各占一个存储单元。函数 main 的初始栈帧大小 CX+6，运行栈上所分配的局部变量最多时有 6 个，main 函数体作用域中的 a，b 和 c，main 函数内语句块 2 作用域中的 d，以及 main 函数内语句块 3 作用域中的 b 和 e。

“嵌套层号”属性中，LEV 表示当前全局作用域所对应的嵌套层次。若当前全局作用域是最外层（可以考虑有更外层的作用域，比如 extern 变量构成的作用域），可令 LEV=0。这一属性在采用单表结构进行符号表的组织和管理时用于区分不同的作用域（参见第 4 节），也可用来实现基于静态访问链的非局部量访问。

多级的符号表结构是指符号表表项的某些属性会关联其他的符号表。例如，对于第 2 节的程序片段 Prog2，我们采用如图 4 所示的多级结构的符号表组织。该符号表结构中，共有 4 种类型的作用域，即全局（Global）作用域，类（Class）作用域，形参（Formal）作用域，以及局部（Local）作用域。为简明，图 4 只是 Prog2 符号表结构的示意图，其中省略了部分信息。

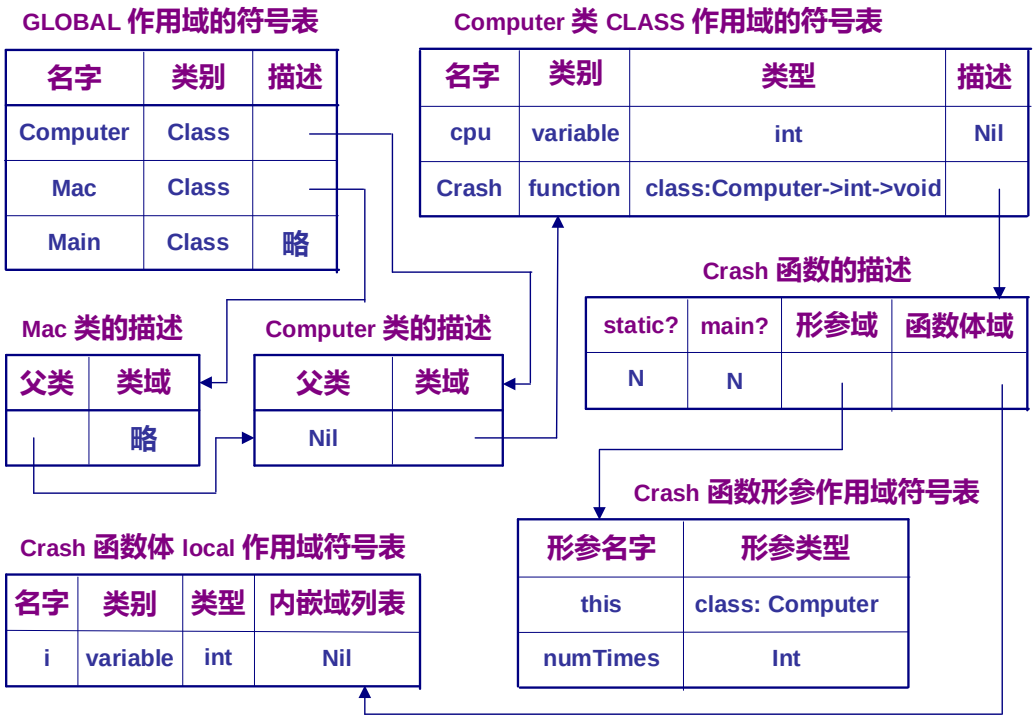

图 4　符号表结构示例（4）

图 4 中，全局作用域（辖类别为 class 的符号 Computer, Mac 和 Main）构成一个 GLOBAL 作用域的符号表，其中各个符号的“描述”属性引申出“父类”和“类域”属性，后者又引申出 Class 作用域的符号表，体现了多级结构的符号表组织。图中省略了 class Main 的相关描述；对 class Mac 也仅刻画了对 class Computer 的继承关系，省略了进一步的描述；对于 class Computer，则从 Class 作用域的符号表开始进行了较为详细的描述。

在 Computer 类 Class 作用域的符号表中，包含 cpu 和 Crash 两个符号的表项，分别对应 Computer 类的实例变量和成员函数（或方法）。可以看出，从符号 Crash 的“描述”属性引申出的“形参域”和“函数体域”，又分别引申出函数 Crash 的 Formal 作用域符号表和 Crash 的函数体 Local 作用域符号表。注意 Crash 的类型为 Computer->int->void 的函数类型，它的第一个形参为自动加入的隐含参数 this，后者的类型为 Computer，这也可以在函数 Crash 的 Formal 作用域符号表中体现，其中含有形参名字 this。对于非静态方法，我们假定均会自动加入隐含参数 this。

下面，我们简单讨论关于符号表的实现。与其它基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式设备的即时编译程序会有这样的需求）。

以下列举几种实现符号表的常见数据结构：

- 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。但本课的重点不在于此，故不对此进行更深入的讨论。一方面，相关内容可以在其他相关课程得到训练。另一方面，由于计算机软硬件系统的大幅进步，早年所关注的一些以节省编译程序运行时间和空间的技术，现如今早已不是主流。

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不排除可由词法分析程序写入，但通常是不会这样，不如直接由语法分析程序来做，因为后者可以同时写入更多的属性信息，并且知道是否为正在声明的符号（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。另外，符号表在语法分析之后而在语义检查之前创建是很常见的选择，这种方法容易获得符号的更多属性（假定所需要的属性在抽象语法树中有完整记录），也容易处理同一作用域内随处声明的符号。

4 块结构语言基于静态嵌套作用域的符号表组织与管理

允许名字作用域嵌套的语言称为块结构语言（*block-structured languages*）。自 Algol60 之后，绝大多数程序设计语言均支持静态嵌套的作用域，遵循静态作用域规则。支持动态作用域规则的语言很少，如不专门说明，我们提到 “作用域”默认是指在静态作用域规则下的含义。后续会提及静态和动态两种作用域规则的区别。

本节专门讨论这种块结构语言基于静态嵌套作用域的符号表组织与管理，相关方法可以体现最基本和最常用的作用域和可见性规则。

第 2 节结合例子介绍了作用域与可见性的基本含义，为进一步讨论符号表组织与管理机制，我们再引入几个相关术语。

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（*current scope*）。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*）。

基于静态嵌套作用域的典型可见性规则（*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。

多数情况下，每个作用域都有各自的符号表（*individual table for each scope*），称之为多符号表（*multiple symbol tables*）组织。但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（*single symbol table*）组织。下面分开讨论基于这两种方式的符号表组织与管理常规方法。

4.1 单符号表的组织与管理

通常，单符号表的组织与管理具有以下特点：

- 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指明作用域的嵌套层次；
- 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。

由于是所有作用域共享同一张符号表，所以针对某个作用域的符号进行访问或维护操作时，需要区分出来是否为这个作用域的符号。显然，这可以通过作用域号来做到。然而，由于表中仅记录开作用域中的符号，每个当前开作用域的嵌套层次是唯一的，因此也可以通过嵌套层次编号达到这一目标。后者更为常用，因其还可以用于其他方面，如实现基于静态访问链的非局部量访问。嵌套层次编号，在第 3 节的例子中称为“嵌套层号”。

下面看一个单符号表组织与管理的例子。针对第 2 节的代码片段 Prog1，编译程序维护一个线性表结构的全局符号表，相当于一个栈结构。当编译程序处理到 Prog1 的程序位置/*here*/时，当前全局符号表的内容如图 5 所示，其间经历如下过程：

（1）开始时，全局作用域符号表（见图 1）中的符号 a 与 foo 的表项相继入栈；假设最外嵌套层号 LEV=0，则这两个符号的嵌套层号为 0；

（2）然后，foo 函数形参作用域的参数符号 a（嵌套层号为 1）的表项入栈，以及 foo 函数体作用域的局部符号 b（嵌套层号为 1）的表项入栈；注意，这里将函数形参作用域的符号与函数体作用域的符号设置了相同的嵌套层次，将来在处理到函数结束位置时统一将它们退栈，这是可适用于多数语言的常规做法；

（3）扫描至退出 foo 函数体作用域时，将当前嵌套层号（1）的所有符号 b 和 a 的表项

依次从栈中删除；

（4）接下来开始扫描函数 main 的代码，将符号 main 的表项入栈（嵌套层号为 0），与当前栈中的嵌套层号同为 0 的符号 a 与 foo 一起，在栈顶形成完整的全局作用域符号表（如图 1）；

（5）随后，main 函数体作用域局部符号 a，b 和 c 的表项入栈，其嵌套层号均为 1；注意，main 函数形参作用域中没有符号，直接过渡到处理 main 函数体作用域；

（6）接下来，main 函数内语句块 1 作用域的符号 b 和 c 的表项相继入栈，这两个符号的嵌套层号为 2；

（7）退出 main 函数内语句块 1 作用域时，将当前嵌套层号（2）的所有符号 c 和 b 的表项依次从栈中删除；

（8）随后是 main 函数内语句块 2 作用域的符号 d（嵌套层号为 2）的表项入栈；

（9）接着，main 函数内语句块 3 作用域的符号 b 和 e 入栈，二者的嵌套层号为 3；

（10）接下来扫描至/*here*/，当前全局符号表的内容如图 5 所示。

名字	嵌套层号	类别	类型	偏移量	初始栈帧空间
a	LEV	variable	int	DX	
foo	LEV	function	int		CX+2
main	LEV	function	int		CX+5
a	LEV+1	variable	int	DX	
b	LEV+1	variable	int	DX+1	
c	LEV+1	variable	int	DX+2	
d	LEV+2	variable	int	DX+3	
b	LEV+3	variable	int	DX+4	
e	LEV+3	variable	int	DX+5	

图 5　某编译器的线性单符号表组织

在 Prog1 的程序位置/*here*/的语句需要访问符号 a 和 b，根据可见性规则，从全局符号表当前栈顶开始在栈内查询，发现最近的 b 层号为 3，而最近的 a 层号为 1，即为实际所访问的符号。

为改进查询效率，可以采用更加高效的表数据结构。比如，若采用一个 Hash 表的结构来组织全局符号表，同样针对第 2 节的代码片段 Prog1，当处理到程序位置/*here*/时，符号表的当前状态如图 6 所示。

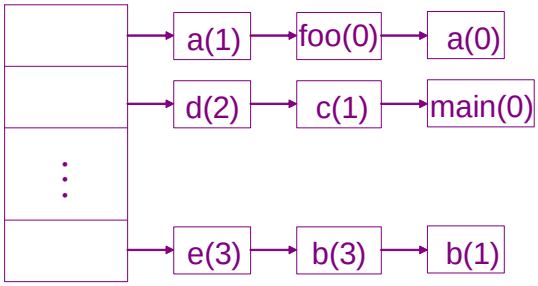

图 6　采用 Hash 表的某编译器的单符号表组织

图 6 中，各符号的散列值是我们随意假设的。针对 Prog1 的程序位置/*here*/，当前的开作用域包括：第 0 层的全局作用域，含符号 a(0)，foo(0)，main(0)；第 1 层的 main 函数体作用域，含符号 a(1)，b(1)，c(1)；第 2 层的 main 函数内语句块 2 作用域，含符号 d(2)；以及第 3 层的 main 函数内语句块 3 作用域，含符号 b(3)和 e(3)。这里，各符号所附加的数字代表符号所在作用域的嵌套层号。

在如图 6 所示的 Hash 表中插入一个符号对应的表项时，假设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 a，其含义是指向符号 a(1)所对应的表项；引用符号 b，则指向符号 b(3)所对应的表项。

在单符号表的组织与管理方式下，唯一的符号表中仅包含当前开作用域中已经扫描到的符号，已关闭的作用域所有符号均不在表中。这种方式适合用于单遍编译器的符号表组织，但也常被于函数局部作用域中所有嵌套的块作用域共享同一张符号表。

另外，无论是哪种场合用到，单符号表的组织与管理方式无法支持前向引用（*forward reference*）的特性，遇到之前未声明的符号总会报错，不管该符号后续是否会被声明。例如，以下是某简单语言的一个程序片段 Prog3：

```
（1）    var x;
（2）    procedure   p ;
（3）        var x;
（4）        procedure   r ;
（5）            var y;
（6）            begin
  .                  ……      /*不含任何 call 语句和声明语句*/
（l1）               call q;
  .                  end;
  .          begin
（l2）           call r ;
  .                ……      /*不含任何 call 语句和声明语句*/
  .          end ;
  .    procedure   q ;
  .          var y;
```

```
.           begin
.               ……      /*不含任何 call 语句和声明语句*/
（l3）          call p ;
.           end ;
.     begin
.           ……      /*不含任何 call 语句和声明语句*/
.           call p;
.     end .
```

该语言中不包含数据类型的声明，所有变量的类型默认为整型，变量声明以保留字 var 开头；语句块的括号为 ‘begin’和 ‘end’组合；过程声明保留字为 procedure。该语言支持嵌套的过程声明（类似 Pascal 语言），但只能定义无参过程，且没有返回值。

若实现该语言的编译器采用某种全局的单符号表结构，比如图 5 或图 6 所示的那样，在分析至语句（l_1）时会报告语义错误（符号 q 不存在）。因为当前符号表中尚无符号 q 的表项，虽然 q 是某个开作用域（最外层作用域）中的符号。与此对照，在分析至语句（l_2）时，r 已出现在当前开作用域（p 过程体作用域）中，因而在符号表中有相应表项，不会报告类似的错误。

如需支持前向引用，不适合采用单符号表的组织方式来实现，因为后者只能记录当前扫描过的符号，即使是采用多遍扫描，效果也是一样。

C 语言中不允许前向引用，调用一个函数之前，必须先定义或先申明过这个函数。这样规定后，就给采用单符号表的组织方式来实现提供了机会。

4.2 多符号表的组织与管理

通常，多符号表的组织与管理具有以下特点：

- 每个作用域都有各自的符号表。
- 需要维护一个作用域栈（*scope stack*），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。

下面举两个多符号表组织的例子。

第一个例子，同样针对第 2 节的代码片段 Prog1。当处理到程序位置/*here*/时，符号表的当前状态如图 7 所示。如第 2 节所述，这一代码段包含 8 个作用域。当前开作用域如图 2 所示，从作用域栈的栈顶到栈底依次为:（1）main 函数内语句块 3 作用域，辖符号集合{b, e}；（2）main 函数内语句块 2 作用域，辖符号集合{d}；（3）main 函数体作用域，辖符号集合{a, b, c}；（4）全局作用域，辖符号集合{a, foo, main}。其余作用域当前状态下为闭作用域。

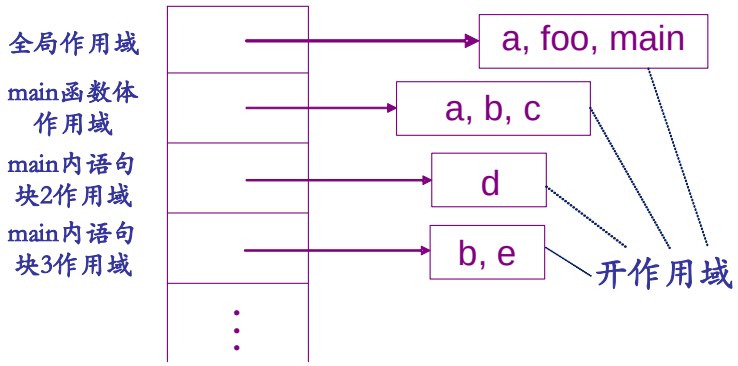

图 7　多符号表组织示例（1）

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 b，其含义是指当前栈顶作用域中的符号 b；引用符号 a，是指次次栈顶作用域中的符号 a。

另一个例子，是针对第 2 节的代码片段 Prog2，为方便，也将代码列于图 8 左边。如图 8 所示，当处理到图中唯一的 for 语句时，当前作用域栈中包含 4 个开作用域：（1）GLOBAL 作用域；（2）类 Computer 的 CLASS 作用域；（3）类 Computer 方法（成员函数）Crash 的 FORMAL（形参）作用域；（4）类 Computer 方法 Crash 的函数体 LOCAL 作用域。图 8 右边的图中，一些省略掉的内容可参见图 4。

```
class Computer {
   int cpu;
   void Crash(int numTimes) {
      int i;
      for (i = 0; i < numTimes; i = i + 1)
         Print("sad\n");
   }
}

class Mac extends Computer {
   int mouse;
   void Crash(int numTimes) {
      Print("ack!");
   }
}

class Main {
   static void main() {
      class Mac powerbook;
      powerbook = new Mac();
      powerbook.Crash(2);
   }
}
```

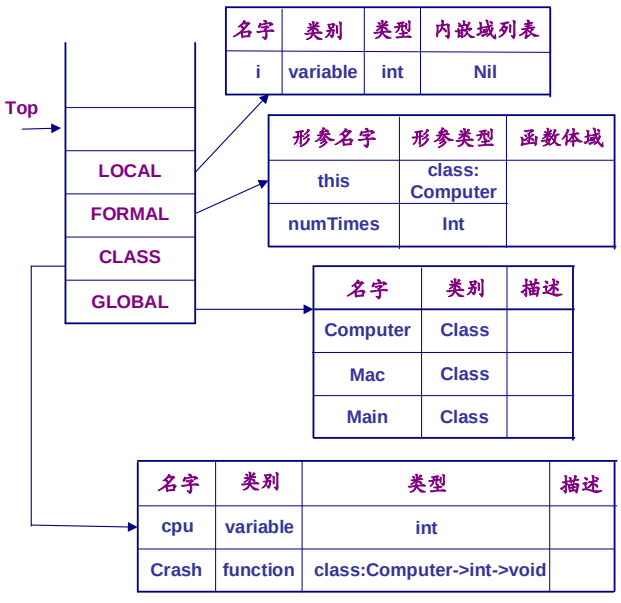

图 8　多符号表组织示例（2）

这里一个有关作用域与可见性的问题。对于上述代码 Prog2，在类 Mac 的 Crash 方法中可以访问由父类继承下来的属性 cpu。但进入这个 Crash 方法的 LOCAL 作用域后，cpu 并

不在某个开作用域中（开作用域 GLOBAL 中只有父类标识符）。然而，能否直接引用父类继承下来的属性是取决于 extends 的实现，通常是可直接引用的。这说明，虽然作用域与可见性二者是密切关联的，但二者有着本质的不同。从嵌套层次来看，将 cpu 看作开作用域中的符号是不恰当的，然而，继承性决定了 cpu 的可见性。

若采用多符号表的组织与管理，可以方便支持前向引用的实现。比如，可以采用单独一遍扫描来构建所有符号表。当后续扫描时，出现在作用域栈中的作用域所对应的符号表，已包含该作用域中的全部符号，因此后面定义的符号也可以访问得到。

例如，对于 4.1 中程序片段 Prog3 的简单语言，若采用多符号表组织，并假设在语义分析遍之前已经构建好虽有的符号表，则在分析至语句（l_1）时不会报告语义错误，因为符号 q 已经包含在最外层作用域对应的符号表中。

5 影响符号访问规则与查找方式的其它典型语言特性

第 4 节讨论了经典的基于静态嵌套作用域的符号表组织与管理，适合多数语言都具有的块结构特性。除块结构语言特性之外，还有其他各种语言特性会影响到符号表的结构、组织及管理方式。下面主要列举一些对符号的访问规则与查找方式影响较大的典型语言特性，从而也对符号表的结构、组织及管理方式有较大的影响。

（1）记录（Record）与结构体（Struct）

多数编程语言均支持记录/结构体的特性。每个记录/结构体是由多个域（*fields*）的数据构成的数据集合体，如下列 C 语言定义的结构体 a：

```
struct {
        char name[20];
        int   age;
        b:    struct {
                char subject[10];
                int rank;
                c:   struct {…. ;   int b;   int c;   int d; …} ;
        }
}   a;
```

结构体的定义可以嵌套，如上述结构体 a 的域 b 本身是一个结构体，而结构体 b 中又定义了一个结构体 c 的域。结构体 a 中域 b 的数据可通过 a.b 访问，而 b 中的结构体 c 可由 a.b.c 访问，以及通过 a.b.c.d 可以访问上述定义中的数据域 d。嵌套的结构体定义中，只要保证同一层次的域名符号不冲突即可，不同层次之间可以有相同名字的域名。例如上述结构体中最内层的结构体 c 中包含名字为 b 和 c 域，使用 a.b.c.b 和 a.b.c.c 所访问的是哪两个数据单元是无异议的。

多数语言，如 C，Ada 以及 Pascal，对于嵌套表定义的记录/结构体均采用类似上述的域名符号访问方式。也有少数例外，如语言 COBOL 和 PL/I，可以在无歧义的情况下略去中间层次的域名符号，如允许使用 a.d 代替上述的 a.b.c.d。但后者很少见，实现起来较为复杂，而且使用也并不方便。

记录/结构体符号，可登记为其所在作用域对应的符号表中的一个表项，对应的描述可

实现为一棵树的根结点，其域名符号的描述作为该结点的孩子，本身为结构体的域可作为相应子树的根节点。也可以将所有域名符号看作一个作用域并对应自己的符号表，本身为结构体的域对应新一层嵌套的作用域，这样就形成一个多级结构的符号表。

有些语言可以使用开域语句，如 Pascal 的 with 语句，可以改变记录/结构体中域名符号的访问方式，从而也影响到符号表的组织方式。例如，与上述 C 语言定义的结构体变量 a 对应，下面是在 Pascal 语言中定义的记录变量 a：

```
var a: record
        name: array [1..20] of char;
        age: integer;
        b: record
            subject: array [1..10] of char;
            rank: integer;
            c: record
                ….;   int d; …;
                end
           end
     end
```

下列 Pascal 程序片段中，使用了 with 语句：

```
with a do
    begin
        …… ;      //   可直接使用  name, age, b, b.c, b.c.d, …
        with b do
            begin
                …… ;      //   可直接使用  subject, rank, c, c.d, …
                with c do
                    begin
                        ….;        //   可直接使用  d,
                    end
            end
        …… ;      //   可直接使用  name, age, b, b.c, b.c.d, …
    end
```

在这种开域语句的作用范围内，可以直接访问域名符号，使用时无需将相应记录/结构

体的名字作为前缀。由于开域语句可以嵌套，如上述 Pascal 程序片段，每一层 with 语句均对应不同的作用域。若是采用单符号表的组织方式，需要使用有相应的作用域号/层号。若是采用多符号表的组织方式，则需要维护作用域栈。

通常，记录/结构体可以关联一个类型名，如在 C 语言中，可以使用 typedef 为结构体定义一个别名的类型。

（2）类层次（Class Hierarchies）、继承（Inheritance）与重载（Overloading）

面向对象语言（如 C++和 Java）中，类（*classes*）之间的继承关系形成了类层次结构，也对应了一种特殊的类型层次结构。

单纯看对象中属性与方法符号的访问，面向对象语言中类结构的符号表组织与管理同记录/结构体有相似之处，然而，类层次结构、继承与重载等特性对符号的访问规则和查找方式有很大影响。

继承性使得不在当前类作用域中的父类符号也具有了可见性，这些符号可通过类层次结构进行查找。例如，前面提到，对于第 2 节或图 8 左边的代码片段 Prog2，在类 Mac 的 CLASS 作用域中可以访问从父类 Computer 继承的属性符号 cpu。同时，在 Crash 方法的 LOCAL 作用域中，符号 cpu 也是可见的。

在一些面向对象语言中，由继承性带来的可见性规则变化，会受到某些符号修饰符的影响。如在 C++语言中，修饰符 private、protected 和 public 对类成员符号的可见性有如下影响：被 private 和 protected 修饰的是私有符号，外部对象无法访问；private 的符号仅在本类对象内部可访问；protected 的符号可以被继承，在子类对象中也可以访问；而 public 的符号在当前类及子类的对象中可访问，同时在外部也可以访问（访问方式类似于对记录/结构体的域名符号的访问）。

面向对象语言（如 C++和 Java 等）中，子类可以重载（*overloading*）父类的方法，相应符号在子类中不可见。通常，重载的方法不只是名称相同，方法的签名（*signature*）也应相同，即输入和输出参数的个数相同，且对应的类型也应匹配。

某些语言，如 C++和 Ada，允许运算符重载。比如，运算+遇到整数作为参数和遇到字符串作为参数，其含义完全不同。此类语言的符号表组织与管理方式，要能够体现运算符在各个作用域场景下的具体含义。

（3）符号的导出（Export）和导入（Import）

符号的导出规则允许将某个内部作用域中的符号可以在该作用域的外部可见。符号的导入规则则是限定哪些外部符号可以在某个作用域的内部可见。导出规则有悖于如第 4 节所述的经典块结构语言的可见性规则。导入规则可以认为是对经典可见性规则的限定，以便对外部可见符号进行有序化梳理。

大型软件系统设计，离不开模块化、封装、和信息隐蔽等机制的支持，在程序设计语言层面，相应出现了如 Modula-2 的 module、Ada 的 package、以及 C++和 Java 的 Class 等语言特性。符号的导入和导出规则在这些语言特性中有明确的体现。如 Modula-2 的 module 定义中，可显式地使用修饰符 IMPORT 和 EXPORT 指明所导入和导出的符号。

前面提到，在 C++语言中，可以使用修饰符 private、protected 和 public 对类成员符号的可见性进行规定，相当于针对类成员符号的导出和导入规则。进一步，C++中还可以使用修

饰符 friend 定义友元类，可看作是定义了特性化的导入规则。

在 C 语言文件中，全局定义的符号默认可以导出（export）给其他文件使用，后者使用时需要使用修饰符 extern 导入相应符号的定义（对于当前文件中定义的符号，可以省略 extern）。若是不允许其他文件导入的符号，定义该符号时需加 static 修饰符。另外，还可以使用编译指导命令#include 导入其他文件中定义的符号，这相当于将这些符号的定义原样拷贝并入当前编译的文件，因此在使用时需要注意是否符合导入导出规则的要求。

在 Java 语言中，可以使用指导命令 import 描述当前编译单元中可以访问的包和类。Ada 中的指导命令 use 也有类似的作用。

（4）隐式声明（Implicit Declaratios）

有时，某些符号看似没有声明就被使用，即隐式声明的符号。例如，Fortran 变量未经声明就可以使用，其类型取决于该符号名字串的第一个字符。Basic 变量使用前也不用声明，其所呈现的数据类型可以从使用的上下文中推导出来。还有，C 以及多数语言中的标号不声明就可直接使用。这些隐式声明的例子对符号表的组织与管理影响不大。

对符号表的组织有较大影响的一个例子比如 for-循环中的循环变量，多数语言都不需先声明后使用。For-循环变量的类型一般默认与其范围表达式的类型一致。循环变量一般与 for 语句所在作用域中的变量无关联，因此通常会为 for-循环体创建新的作用域，将循环变量纳入该作用域中，而 for 语句所在作用域为嵌套的直接外层作用域。

（5）前向引用（Forward References）

前向引用是指允许引用声明在后的符号。否则，如果不支持该特性，当从前向后扫描，遇到之前未声明的符号时会报告如“符号未声明”之类的语义错误。参考第 4 节的讨论，实现对前向引用特性的支持，可采用多符号表的组织与管理，并进行至少两遍扫描。

（6）动态作用域（Dynamic Scope）规则（相关内容也见第八讲）

本讲之前对符号表设计、组织与管理的讨论均是基于静态作用域规则，即程序某处所使用名字的声明之处是可以静态确定的。在动态作用域（*dynamic scope*）规则下，符号的作用域与可见性取决于执行历史中的声明信息，只有在程序执行时才能确定程序某处所使用名字的声明位置。

为理解动态作用域规则和静态作用域规则的主要差异，我们来看一个简单的例子。设有 Pascal 程序片断 Prog4：

```
var    r: real
procedure    show;
      begin
            write(r:5:3)       //  以长度为 5 小数位数为 3 的格式显示实型量 r 的值
      end;
procedure    small;
      var      r: real;
      begin
            r:=0.125;    show
      end;
begin
```

```
        r:=0.25;
        show; small; writeln;
        show; small; writeln;
    end.
```

若采用静态作用域规则，无论在哪个上下文中执行，过程 show 中的变量 r 总是指全局声明的 r，因此执行结果是：

0.250　0.250

0.250　0.250

若采用动态作用域规则，则在不同的上下文中执行，过程 show 中的变量 r 会被认为是最近的调用过程所声明的 r，因此执行结果是：

0.250　0.125

0.250　0.125

由此例可以看出，若遵循动态作用域规则，非局部符号（如过程 show 使用的非局部符号 r）的查找/定位，与程序的运行时执行历史密切相关，无通用办法可以在编译期间进行，实际上，需要通过运行时的动态链属性实现查找/定位，参见第 6 章的相关内容。

早期的 Lisp（Common Lisp 之前）、Snobol 以及 Perl 等少数语言采用动态作用域规则。现代编程语言中，仅有 Common Lisp 和 Perl 支持动态作用域规则，而它们同时也支持静态作用域规则。可见，现代语言基本上已抛弃动态作用域规则。

相比静态作用域规则，动态作用域规则有不少弊端，比如：并非所有的非局部符号都可以静态确定，从而影响到静态类型检查工作；函数的本地变量对于被调用函数而言都是可见的，也是可修改的，不符合信息隐蔽原则，潜在的可靠性风险很大；运行时非局部量的访问需要沿调用链反向查找，效率较低。

本课中，若非特别指明，所讨论的内容均假定是针对基于静态作用域规则的语言。

课后作业

1. 思考与讨论：在哪个阶段创建符号表？

2. 思考与讨论：符号的常见属性在哪个阶段、如何收集？在哪个阶段、有何种用途？

3. 思考与讨论：相比多符号表组织，单符号表组织的适用范围受限，试探讨单符号表组织可适用于哪些场景？

4. 思考与讨论：对于自己熟知的对作用域和可见性有影响的语言特性，可能采用什么样的符号表组织？

5. 对于第2节的程序片段Prog1，本讲给出了若干作用域所对应的符号表（参见图1，图2和图3）。试基于同样的符号表结构，补充Prog1中其他作用域所对应的符号表。

6. 对于第2节的程序片段Prog2，本讲采用了如图4所示的多级结构的符号表。试补齐图4中所省略的关于类Mac和Main的符号表结构。

7. 如下是某语言的一段代码，若该语言编译器的符号表如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 .
```

8. 以下是某简单语言的一段代码。该语言中不包含数据类型的声明，所有变量的类型默认为整型，变量声明以保留字var开头；语句块的括号为 ‘begin’和 ‘end’组合；过程声明保留字为procedure；赋值号为 ‘:=’，不等号为‘<>’。该语言支持嵌套的过程声明（类似于Pascal语言），但只能定义无参过程，且没有返回值。

```
（1） var a0, b0, a2;
（2） procedure fun1 ;
（3）         var a1, b1;
（4）         procedure fun2 ;
（5）                 var a2;
（6）                 begin
（7）                         a2 := a1 + b1;
（8）                         if(a0 <> b0) then call fun3;
  .                          …...    /*不含任何 call 语句和声明语句*/
```

```
.                   end;
.           begin
                    a1 := a0 - b0;
                    b1 := a0 + b0;
（x）               If   a1 < b1   then   call fun2 ;
.                   …...      /*不含任何 call 语句和声明语句*/
.           end ;
.   procedure fun3 ;
.           var a3;
.           begin
      a3 := a0*b0 ;
（y）               if(a2 <> a3) call fun1 ;
.                      …...    /*不含任何 call 语句和声明语句*/
.           end ;
.   begin
.           a0 := 1;
.           b0 := 2;
.           a2 := a0 - b0 ;
.           call fun3;
.           …...    /*不含任何 call 语句和声明语句*/

.   end .
```

(a) 若该语言编译器的符号表类似于图 5 所示的那样，采用一个全局的单符号表栈结构。试指出：在分析至语句（x）和（y）时，当前开作用域分别有几个？各包含哪些符号？在分析至语句（y）时，所访问的 a2 是在哪行语句声明的？

(b) 若实现该语言时采用多符号表的组织和管理方式，即每个静态作用域均对应一个符号表。假定采用多遍扫描机制，在静态语义检查之前每个作用域中的所有表项均已生成。实现静态语义分析时，为了体现作用域信息，须要维护一个作用域栈，假设栈中的元素是指向某个作用域的指针。试指出：在分析至语句（x）和（y）时，当前开作用域分别有几个？各包含哪些符号？

9. 设有如下简单面向对象语言（类Java）程序片段：

```
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();
    }
}
```

若实现该语言的编译器采用多符号表的组织和管理方式，使用一个作用域栈来记录当前的开作用域，参考图 8。符号表设计为类似于图 4 所示的多级结构，有 4 种类型的作用域：全局（Global）作用域，类（Class）作用域，形参（Formal）作用域，以及局部（Local）作用域。当处理到图中所标出的 Print 语句时，当前作用域栈中包含哪些开作用域？它们对应的符号表中分别包含哪些符号？

10. 思考与讨论：对于自己熟知的对作用域和可见性有影响的语言特性，可能采用什么样的符号表结构、组织及管理方式？

11. 思考与讨论：大多数语言均采用静态作用域规则，而不是动态作用域规则，你认为遵循动态作用域规则会有哪些弊端？不限于第5节所提到的。