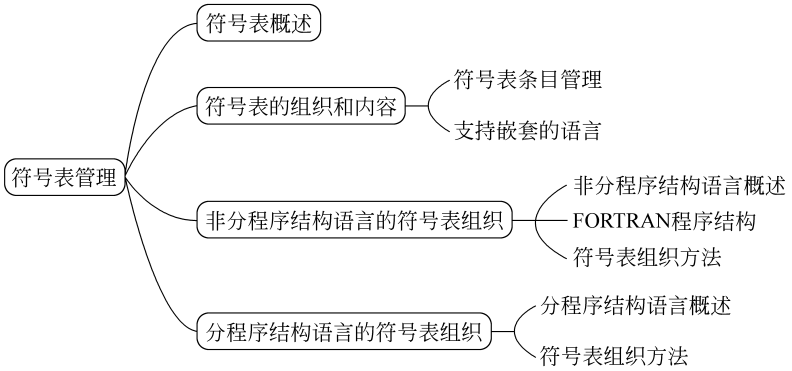
第 5 章　符号表管理



符号表是编译器中用来管理程序中的标识符的数据结构，它保存了程序中各个变量名、函数名、常量名等标识符的信息。一般来说，符号表包含两个主要部分：符号表入口和符号表内容。其中，符号表入口是用来访问符号表内容的主要方法，而符号表内容则包含了每个标识符的相关信息。

符号表的组织方式主要有线性表和哈希表两种，这两种方式各有优劣。线性表通常采用顺序表或链表实现，其优点在于实现简单且占用内存较少，但不适用于大规模的程序；哈希表则可以通过哈希函数将标识符映射到固定的位置上，从而快速地查找标识符。哈希表具有查找速度快的优点，但需要占用更多的内存。此外，非分程序和分程序的符号表组织也不尽相同。

符号表管理是编译器中非常重要的一部分，符号表合理的组织方式和内容可以大大提高编译器的效率和质量。本章对符号表的组织和内容进行介绍。

5.1　概述

编译器通过符号表跟踪源程序中使用的变量和其他命名实体。符号表是一个关键的数据结构，用于存储每个标识符（如变量名、函数名、常量名等）及其相关属性信息。符号表可以帮助编译器在分析源代码时进行语义分析和类型检查，同时也可以在代码生成阶段生成相应的汇编代码。

符号表可以采用各种不同的数据结构，例如哈希表、树和链表等。对于较小的程序，使用简单的线性结构就足够了；但对于大型程序，更高效的数据结构可以更好地支持快速查找和更新符号表。

符号表中每个记录条目通常包含以下字段：标识符名称、数据类型、存储位置、作用域、是否初始化等信息。对于过程或函数名，还需要记录参数数量、类型、返回类型等信息。符号表的一个重要作用是确保在程序中使用的每个变量都已经被声明，并且在正确的作用域

内使用。

编译器会在词法分析和语法分析过程中不断更新符号表。在遇到新的标识符时，编译器会将其添加到符号表中。在变量被引用时，编译器会查找符号表以确定它的属性信息，以便在后续的语义分析和代码生成阶段中使用。

一个语句的翻译过程伴随着符号表检查和更新的过程，如图 5.1 所示。

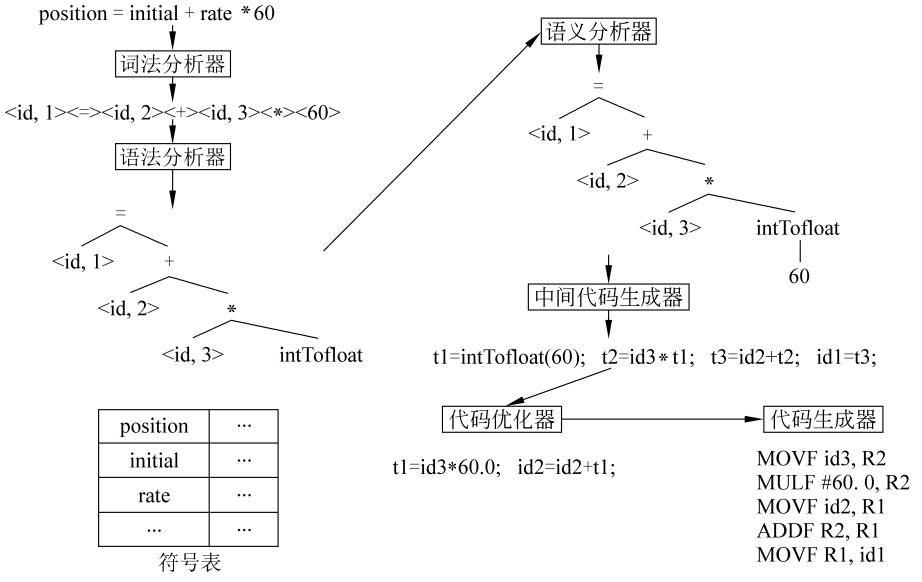


图 5.1　一个语句的翻译过程

5.2　符号表的组织和内容

符号表是一种供编译器用于保存有关源程序构造的各种信息的数据结构。这些信息在编译器的分析阶段被逐步收集并放入符号表，它们在综合阶段用于生成目标代码。符号表的每个条目中包含与一个标识符相关的信息，例如它的字符串（或者词素）、类型、存储位置和其他相关信息。符号表通常需要支持同一标识符在一个程序中的多重声明。

一个声明的作用域是指该声明起作用的那一部分程序。编译器将为每个作用域建立一个单独的符号表。每个带有声明的程序块都会有自己的符号表，这个程序块中的每个声明都在此符号表中有一个对应的条目。这种方法对其他能够设立作用域的程序设计语言构造同样有效。例如，每个类也可以拥有自己的符号表，它的每个域和方法都在此表中有一个对应的条目。

符号表条目是在分析阶段由词法分析器、语法分析器和语义分析器创建并使用的。因为语法分析器知道一个程序的语法结构，因此相对于词法分析器而言，语法分析器通常更适合创建符号表条目。它可以更好地区分一个标识符的不同声明。

在有些情况下，词法分析器可以在它遇到组成一个词素的字符串时立刻创建一个符号表条目。但是在更多的情况下，词法分析器只能向语法分析器返回一个词法单元，例如 id，以及指向这个词素的指针。只有语法分析器才能够决定是使用已经创建的符号表条目还是

为这个标识符创建一个新条目。

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的情况下收集的，特别是多遍编译以及程序分段编译(以文件为单位)的情况下，更需要检查标识符属性在上下文中的一致性和合法性。通过符号表中的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既可作为定义说明，也可作为引用说明：

```
int index;                  //定义
extern double i;            //引用
```

在编译过程中，符号表中首先建立标识符i，其属性是int型变量；而后在扫描到第二行代码时，标识符i的属性是double型变量。通过符号表的语义检查可发现其不一致的错误。

另外，符号表还可用于优化目的。例如，编译器可能利用符号表的信息减少目标代码或者提高代码执行效率。编译器还可以利用符号表来检测和消除未使用的变量和未引用的函数等无效代码，从而提高程序的性能和效率。

符号表的实现通常采用哈希表、树和链表等数据结构。哈希表是一种快速查找的数据结构，通过使用哈希函数将每个条目映射到哈希表中的一个位置。树和链表是有序的数据结构，它们可以用于在符号表中进行插入和查找操作。在实现符号表时，需要考虑到查找和插入的时间复杂度以及空间复杂度等方面的因素。

符号表是编译器中非常重要的数据结构，它为编译器提供了存储、管理和检索程序中标识符相关信息的能力，对于编译器的正确性、效率和可维护性等方面都有着至关重要的作用。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确定将来被分配的存储位置。首先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、文件静态区、函数静态区还是函数运行时的动态区等。其次根据变量出现的次序(一般来说是先声明的在前)决定该变量在某个区域中所处的具体位置，这通常使用在该区域中相对于起始位置的偏移量确定。而有关区域的标志及相对位置都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行信息。

一个标识符的使用范围称为作用域。标识符的作用域实际上指的是标识符的某个声明的作用域。术语作用域(scope)本身是指一个或多个声明起作用的程序部分。作用域是非常重要的，因为在程序的不同部分可能会出于不同的目的而多次声明相同的标识符。像x和i这样常见的名字会被多次使用。再如，子类可以重新声明一个方法名字以覆盖父类中的相应方法。如果程序块可以嵌套，那么同一个标识符的多次声明就可能出现在同一个程序块中。当stmts能生成一个程序块时，下面的语法规则会产生嵌套的块：

block→'{'decls stmts'}'

这个语法规则中的花括号使用了引号，这么做的目的是将它们和用于语义动作的花括号区分开来。在图5.2给出的文法中，decls生成一个可选的声明序列，stmts生成一个可选的语句序列。

更进一步，一条语句可以是一个程序块，所以该语言支持嵌套的程序块。而标识符可以在这些程序块中重新声明。

支持嵌套的语言的程序块可使用最近(most-closely)嵌套规则处理。最近嵌套规则是

```
program ⟶                  {top=null;}
          block
block   ⟶ '{'               {saved=top;
                              top=new Env(top);
                              print("{");}
          decls stmts'}'
decls     decls decl
          |ε
decl      type id;          {s=new Symbol;
                              s.type=type.lexeme;
                              top.put(id.lexeme,s);}
stmts   ⟶ stmts stmt
          |ε
stmt    ⟶ block
          |factor;          {print(;);}
factor  ⟶ id                {s=top.get(id.lexeme);
                              print(id.lexeme);
                              print("{");
                              print(s.type);}
```

图 5.2　使用符号表翻译带有程序块的语言

说一个标识符 x 位于最近的 x 声明的作用域中。也就是说，从 x 出现的程序块开始，从内向外检查各个程序块时找到的第一个对 x 的声明指定了 x 的作用域。

下面的伪代码用下标区分对同一标识符的不同声明：

```
|int x1;int y1;
|int w1;bool y2;int z2;
…w2…;…x1…;…y2…;…z2…;
|
…w0…;…x1…;…y1…;
|
```

下标并不是标识符的一部分，它实际上是该标识符对应的声明的行号。因此，x 的所有出现都位于第 1 行上声明的作用域中。第 3 行上出现的 y 位于第 2 行上 y 的声明的作用域中，因为 y 在内层块中被再次声明了。然而，第 5 行上出现的 y 位于第 1 行上 y 的声明的作用域中。

假设第 5 行上出现的 w 位于这个程序块之外某个 w 声明的作用域中，它的下标表示一个全局的或者位于这个程序块之外的声明。

最后，z 在最内层的程序块中声明并使用。它不能在第 5 行上使用，因为这个内嵌的声明只能作用于最内层的程序块。

实现语句块的最近嵌套规则时，可以将符号表链接起来，也就是使得内嵌程序块的符号表指向外围程序块的符号表。

图 5.3 对应于上面伪代码的符号表。

B_1 对应于从第 1 行开始的程序块。B_2 对应着从第 2 行开始的程序块。图 5.3 的顶端是符号表 B_0，它记录了全局的或由语言提供的默认声明。在分析第 2～4 行时，环境是由一个指向最下层的符号表(即 B_2 的符号表)的指针表示的。当分析第 5 行时，B_2 的符号表变得不可访问，环境指针转而指向 B_1 的符号表，此时可以访问上一层的全局符号表 B_0，但不能访问 B_2 的符号表。

程序块的符号表的优化可以利用作用域的最近嵌套规则实现。嵌套的结构确保可应用

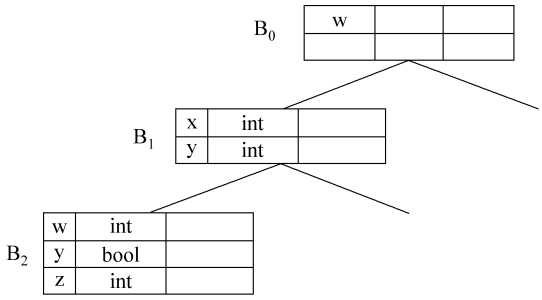


图 5.3 符号表链

的符号表形成一个栈。在栈的顶部是当前程序块的符号表。栈中这个符号表的下方是包含这个程序块的各个程序块的符号表。因此,符号表可以按照类似于栈的方式分配和释放。有些编译器维护了一个哈希表存放可访问的符号表条目。也就是说,存放那些没有被内嵌程序块中的某个声明覆盖的条目。这样的哈希表实际上支持常量时间的查询,但是在进入和离开程序块时需要插入和删除相应的条目。在从程序块 B 离开时,编译器必须撤销所有因为 B 中的声明而对此哈希表作出的修改。它可以在处理 B 的时候维护一个辅助的栈以跟踪对这个哈希表的修改。

5.3 非分程序结构语言的符号表组织

非分程序结构语言中每个可独立进行编译的程序单元是一个不包含子模块的单一模块。例如,FORTRAN 的程序构造如图 5.4 所示。

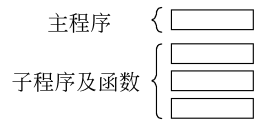


图 5.4 FORTRAN 的程序构造

在主程序和子程序中可以定义 COMMON 语句。FORTRAN 程序中各程序单元之间的数据交换可以通过虚实结合的方式实现,也可以通过建立公用区的方式实现。公用区有两种:无名公用区和有名公用区。任何一个程序中只能有一个无名公用区。一个程序中可以根据需要开辟任意多个有名公用区。无名公用区和有名公用区都通过 COMMON 语句建立。

在该语言系统中,标识符具有两种作用域:全局作用域和局部作用域。具有全局作用域的标识符包括子程序名、函数名和公用区名等,具有局部作用域的标识符包括程序单元中定义的变量等。

对于符号表的组织,要进行如下处理:

(1) 将子程序名、函数名和公用区变量名填入全局符号表。

(2) 在子程序(函数)声明部分读到标识符时,检查本程序单元局部符号表有无同名,若有则报错,若无则填入局部符号表。

(3) 在语句部分读到标识符时,检查本程序单元的局部符号表有无同名,若有则使用,若无则查全局符号表。此时若未查询到则报错,若查询到则使用。

(4) 程序单元结束时,释放该程序单元的局部符号表。

(5) 程序执行完成时,释放全部符号表。

符号表的组织方式分为无序符号表、有序符号表、哈希表。

5.4 分程序结构语言的符号表组织

分程序结构语言中的模块内可嵌入子模块。标识符局部作用于所定义的模块(最小模块),模块中定义的标识符的作用域是定义该标识符的子程序。下面是一个例子:

```
{                   //主程序块 1
    int a;          //1 内全局变量
    {               //程序块 2
        int b;      //2 内局部变量
        …
        int a;      //2 内局部变量,该重名合法,在此程序块中使用 a 时使用此局部变量
}

{                   //程序块 3
        int b;      //3 内局部变量,该重名合法
        …
}
    //在主程序块中使用 a 时使用 1 内全局变量
    //主程序块中无法跳回分程序块,也无法跳回循环体内
}
```

建立、查找符号表的要求如下:

(1) 建表时不能重复、遗漏。

(2) 查表时按标识符作用域查找。

基本处理方法如下:

(1) 在程序的声明部分读到新标识符时建表。

(2) 查表,若无同名,填入符号表;反之则报错。

(3) 在语句中读到引用标识符时查表。

(4) 检查本层符号表,如有同名,使用之;反之则查上层符号表;若最外层符号表查表失败,则报错。

sin、cos 等内置函数是标识符的子集,它们并不是关键字,但是名称和数量已知,应预先将该类标识符填入最外层符号表中。

分程序符号表的组织方式如下:

(1) 分层组织符号表的条目。各分程序符号表条目按照语法识别顺序连续排列在一起,不为其内层分程序的符号表条目所割裂。

(2) 用分程序表为各分程序符号表的信息建立索引。分程序表中的各条目是自左至右扫描源程序的过程中按分程序出现的顺序依次填入的,且对每一个分程序填写一个条目。分程序表条目序号隐含地表征各分程序的编号。

分程序表的结构如图 5.5 所示。其中:

- OUTERN 指明该分程序的直接外层分程序的编号。
- ECOUNT 记录该分程序符号表条目的个数。
- POINTER 指向该分程序符号表的起始位置。

嵌套的代码结构如图 5.6 所示。

OUTERN	ECOUNT	POINTER

图 5.5　分程序表的结构

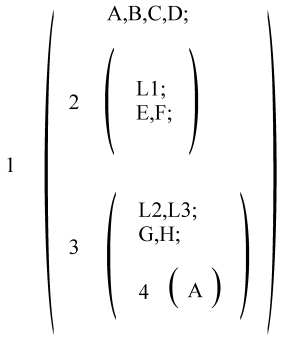


图 5.6　嵌套的代码结构

分程序索引表和符号表如图 5.7 所示。

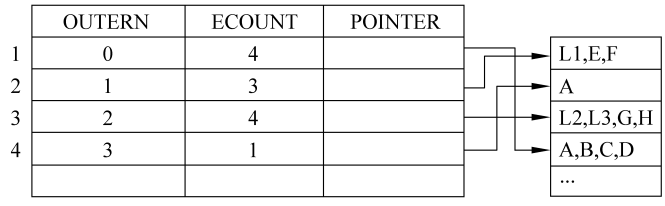


图 5.7　分程序索引表和符号表

分程序索引表的形成顺序与每个模块头的出现顺序一致。

分程序符号表形成顺序与每个模块(语法分析时的语法单位)识别顺序一致。

分程序符号表构造方法如下。为使各分程序的符号表连续地邻接在一起,并在扫描具有嵌套分程序结构的源程序时总是按先进后出的顺序扫描其中各个分程序,可设一个临时工作栈。每当进入一层分程序时,就在栈顶预构造该分程序的符号表;而当遇到该分程序的结束符(END)时,该分程序的全部条目已位于栈顶,再将该分程序的全部条目移至正式符号表中。

小结

本章主要介绍了符号表的相关概念,以下是主要内容:

- 符号表是一种供编译器保存有关源程序构造的各种信息的数据结构。这些信息在编译器的分析阶段被逐步收集并放入符号表,它们在综合阶段用于生成目标代码。
- 符号表条目是在分析阶段由词法分析器、语法分析器和语义分析器创建并使用的。通常语义分析器更适合创建符号表条目。
- 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确定将来被分配的存储位置。
- 对于支持嵌套的语言,使用最近嵌套规则处理其语句块。

- 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声明的符号。
- 非分程序结构语言的符号表组织方式分为无序符号表、有序符号表、哈希表。
- 对于分程序结构语言，其模块内可嵌入子模块，标识符局部作用于所定义的模块（最小模块），模块中定义的标识符的作用域是定义该标识符的子程序。
- 分程序结构语言的符号表组织方式分为分层组织符号表的条目和用分程序表为各分程序符号表的信息建立索引。

习题 5

5.1 给出编译下面程序的有序表：

```
main() {
    int m, n[5];
    real x;
    char name;
}
```

5.2 给出编译到下面程序 a、b、c 处的栈式符号表：

```
realx, y;
char str;
int fun1(int ind) {
    int x;
    x = m2(ind + 1);        //a
}
main(){
    char y;                 //b
    x = 2;                  //c
    y = 5;
}
```

5.3 以下说法中正确的是(　　)。

A. 符号表由词法分析程序建立，由语法分析程序使用

B. 符号表的内容在词法分析阶段填入并在以后各个阶段得到使用

C. 对一般的程序设计语言而言，其编译器的符号表应包含哪些内容及何时填入这些信息不能一概而论

D. “运算符与运算对象类型不符”属于语法错误

5.4 在目标代码生成阶段，符号表用于(　　)。

A. 目标代码生成

B. 语义检查

C. 语法检查

D. 地址分配

5.5 如何基于符号表构造访问链？

5.6 如何基于符号表访问非局部数据？

拓展阅读：Open64的符号表设计

Open64是一套针对Itanium及x86-64架构优化的编译器，它以GNU自由文档许可证发行。Open64源自一套SGI公司为MIPS R10000处理器开发的编译器MIPSPro，它于2000年首次发行并命名为Pro64，隔年特拉华大学将其改名为Open64并负责维护。目前Open64经常作为编译器以及计算机系统结构研究领域的研究平台。Open64使用WHIRL中介码，支持的语言包括C、C++、FORTRAN 77/95以及OpenMP。它可以进行高质量的过程间优化及分析、数据流分析、数据依赖性分析以及数组区域分析。Open64支持的操作系统包括Linux及类UNIX系统。Open64支持的处理器架构包括IA-32(x86)、x86-64、IA-64、龙芯(MIPS)及PowerPC。

Open64使用WHIRL指令集，WHIRL的符号表是基于哈希表设计的。哈希表是一种将键映射到值的数据结构，可以高效地进行插入、查找和删除操作。在WHIRL的符号表中，每个符号表条目都由一个哈希值和一个指向数据结构的指针组成。哈希值是根据符号表条目的键(即符号名)计算出来的，并且用于快速查找符号表条目。

WHIRL的符号表支持作用域嵌套和隐藏。每个符号表条目都包含了符号的名字、类型、存储位置和其他相关信息。在WHIRL中，符号表条目的键是由符号名和所在作用域的编号组成的，这样可以保证在不同的作用域中使用同样的符号名而不会发生冲突。当在一个作用域中声明一个符号时，符号表会在该作用域的符号表中创建一个新的条目，并将其添加到哈希表中。如果在一个作用域中使用了一个已经被声明过的符号名，符号表会查找符号表链表，以确定该符号名所对应的符号表条目。

在WHIRL中，符号表也支持符号的属性继承。当在一个作用域中声明一个新的符号时，如果该符号的类型和之前声明的某个符号的类型相同，那么该新符号将继承之前符号与类型相关的属性。例如，如果一个作用域中声明了一个整型变量，那么在该作用域中声明的所有整型变量都将继承这个变量的属性。

符号表是编译器中的一个重要模块，用于存放源代码中的各种符号(例如变量、函数)的一些重要信息，例如类型信息、初始化值等。并且该结构要能够区分不同作用域中的同名符号，例如：

```
int f(int a){                    //参数 a 在作用域 s0 中
    if(a> 0){
        char a = 1;              //变量 a 在作用域 s1 中
        int b = 2;               //变量 b 在作用域 s1 中
        return b;
    } else {
        int16_6 b = 3;           //变量 b 在作用域 s2 中
        return b;
    }
}
```

两个名为b的变量出现在两个不同的作用域中，应当在符号表中予以区分。然而，真实的工业编译器的符号表通常保存了更多详细的程序信息，用于后续的IR生成、检查以及优化。

在介绍 WHIRL 这个复杂的符号表之前，先介绍一种非常简单的符号表的设计，通常用于简单编译器的实现。以 C 语言为例，不考虑 struct/union 这种复杂语法，可以用一种合适的数据结构实现符号表，那就是哈希表。可以使用符号名作为哈希表的键，用 SymInfo 这个结构体作为哈希表的值，以保存符号的类型等信息。以程序中的函数 f 为例，作用域 s1 的符号表可以像表 5.1 这样。

表 5.1　作用域 s1 的符号表

Name	SymInfo::Type	SymInfo::Init	SymInfo::IsConst	…
a	char	1	false	…
b	int	2	false	…

可以把程序的作用域当作一个树进行处理，因为这里需要解决 3 个问题：

(1) 对于 C 语言而言，低层作用域中的符号可以覆盖高层作用域中的符号，例如上述代码中作用域 s1 中的 char a 变量会覆盖作用域 s0 中的 int a 参数。

(2) 高层作用域无法访问低层作用域中定义的符号，低层作用域却可以自由访问高层作用域中定义的符号。

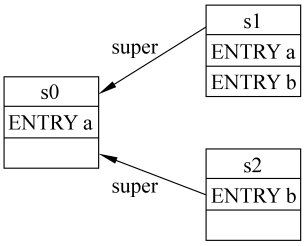


图 5.8　嵌套哈希表

(3) 同级作用域中的符号是互相屏蔽的。例如，在作用域 s2 中看不到作用域 s1 中定义的 a、b 两个符号。

因此，需要设计一种可以满足上述要求的简单数据结构。对于一个简单编译器而言，做完语义分析，确保没有类型错误，就可以把符号表丢掉了。这里可以使用嵌套哈希表实现，如图 5.8 所示。

在图 5.8 中，s0、s1、s2 均维护一个哈希表结构，同时维护一个 super 指针信息，当然 s0 的 super 指针是 NULL。用代码实现如下：

```
class SymTreeNode {
  unordered_map <string, SymInfo * > symbols;
  SymTreeNode * super;
public:
  SymInfo * GetSymInfo(const string &name , bool recursive){
    if(symbols.find(name) != symbols.end ())
    return symbols(name )
    if(recursive){
      return super -> GetSymInfo(name, true);
    }
    return nullptr;
  }
  void AddSymInfo(const string &name, SymInfo * info){
      symbols[name]=info;
  }
  //more methods...
};
```