讲稿（教学内容、步骤）

第 8 章 静态语义分析和中间代码生成

1. 符号表

- 符号表的作用

(1) **收集**和**记录**符号（标识符）的属性信息。
如：符号的名称、类型、数组维数、每一维长度等。

(2) 符号表中的内容是上下文**语义合法性检查**的依据。
(3) 符号表是目标代码生成阶段对符号名进行**地址分配的依据**。
(4) 符号表的组织与结构还体现了符号的**作用域**和**可见性**信息。

- 符号的常见属性

(1) 符号**名**
- ✓ 变量名
- ✓ 函数名 或 过程名等（重载通过参数个数、类型、返回值类型加以区别）。

(2) 符号**类别**
- ✓ 常量符号、变量符号、函数符号、类名符号等

(3) 符号**类型**
- ✓ 整型、实型、字符型、布尔性、位组型等，函数的类型看返回值。
- ✓ 类型决定了该变量的存储格式、还决定了可以施加的运算操作。

(4) 符号的**存储类别**和**存储分配**信息

(5) **作用域**：一个符号变量在程序中起作用的范围。

(6) **可视性**：变量可以出现的场合。
- ✓ 某变量作用域范围内，该变量是可视的

- ✓ 函数的形参影响变量可视性

【举例】

```
int f(int x, int y){
        int sum;
        sum = x + y;
        return sum;
}
int main(){
    int x = 1, y = 2;
    int a = 100, b = 200;

    cout << f( a, b ) << endl;
    return 0;
}
```

✓ 复合语句分程序结构影响变量可视性

【举例】

```
#include <iostream>
using namespace std;

int main(){
    int x = 10, y = 20;
    if(true){
            int x = 1, y = 2;
            cout << x+y << endl;
    }
    return 0;
}
```

(7) **其他属性**

- ✓ 数组的内情向量
- ✓ 结构体或类的成员信息
- ✓ 函数的形参

2. 符号的常见属性

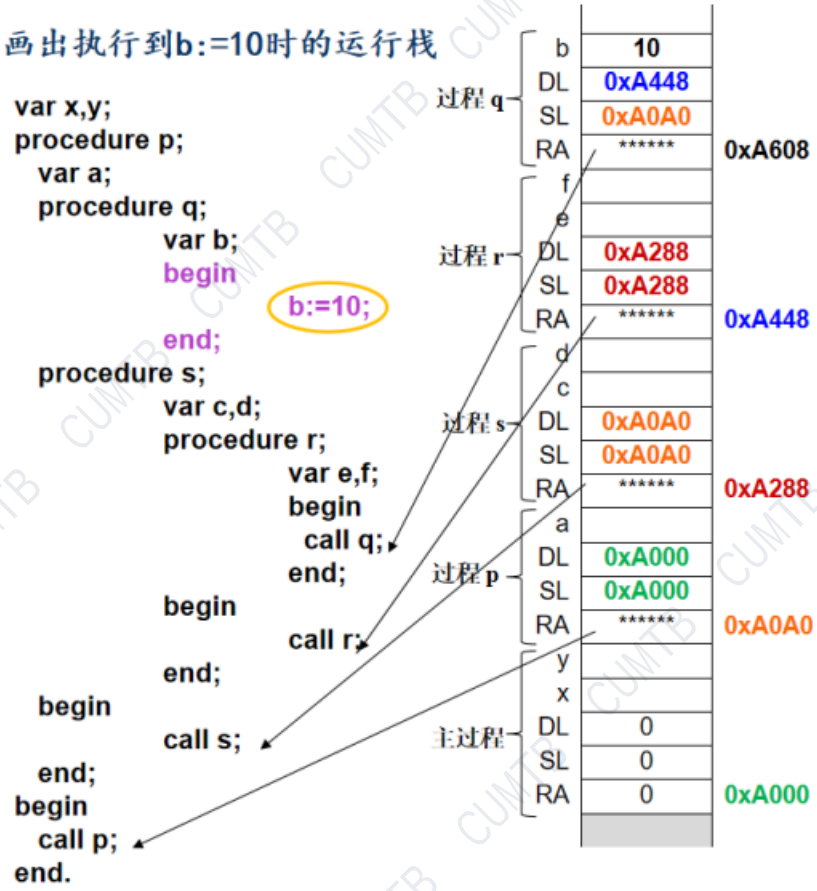
- 符号的存储类别和存储分配信息
  - ✓ 是编译过程语义处理、检查和存储分配的重要依据；
  - ✓ 还决定了符号变量的作用域、可视性和生命周期等。

- 目标代码运行时的存储分配
  - ✓ 目标代码运行时符号的存储分配区域

- ✓ **静态链 SL**（访问链）：指向**定义**该过程的直接外过程运行时最新的活动记录的基址。
- ✓ **动态链 DL**（控制链）：指向**调用**该过程的活动记录的基址。

- ✓ 目标代码运行时的存储分配过程

【举例】



3. 符号表的实现

- 符号表的操作
  - ① 创建符号表：编译开始或进入一个新的作用域时
  - ② 插入表项：遇到新符号声明时
  - ③ 查询表项：引用符号时
  - ④ 修改表项：获得新的语义值信息时
  - ⑤ 删除表项：在符号成为不可见或不再需要时
  - ⑥ 释放符号表空间：编译结束前或者退出一个作用域的时候

- 符号表的组织

(1) 总体组织

① 把属性完全相同的符号组织在一起（**分表制**）

优点：表项等长、单个表容易管理

缺点：表较多、总体管理较复杂

② 把所有符号放在一张表中（**一表制**）

优点：相同属性处理一致、总体管理简单

缺点：表项不等长、表项复杂、空间浪费

③ 根据属性相似的符号组织在一张表中（**折中**）

总体管理的复杂性和时空效率都取得了折衷的良好效果。

(2) 表项组织

① 线性组织

- ✓ 根据扫描到的符号先后顺序建立
- ✓ 优点：管理简单
- ✓ 缺点：运行效率低，符号表长度不能确定
- ✓ 适合：较少符号的组织

② 有序表

- ✓ 将符号按名称排序（词典排序）
- ✓ 优点：查找迅速（折半法查找）
- ✓ 缺点：算法复杂。如：搜索到新符号后，将其插入到表中，须采用链表组织

③ 二叉搜索树

④ Hash 表

- ✓ 用 hash 函数将符号名映射成整数
- ✓ 优点：运行效率高、算法复杂度比排序组织低
- ✓ 缺点：对变量符号很难找到高效的 hash 函数

4. 符号的作用域和可见性

- 作用域：每个符号变量在程序中都有确定的有效范围。拥有共同有效范围的符号所在的程序单元构成了一个作用域。
  - ✓ 作用域可以嵌套，但不能交错。
  - ✓ 开作用域：当前作用域与包含它的程序单元所构成的作用域。
  - ✓ 闭作用域：不属于开作用域的作用域。

【举例】

```
int x = 1, y = 2;
void f(int c, int d) {
    int sum;
    cout << x + y << endl;
}
int main( ) {
    int x = 3, y = 4;
    int a = 5, b = 6;
    f(a, b);
    cout << x + y << endl;
    return 0;
}
```

```
int x = 1, y = 2;
void f(int x, int y) {
    int sum;
    cout << x + y << endl;
}
int main( ) {
    int x = 3, y = 4;
    int a = 5, b = 6;
    f(a, b);
    cout << x + y << endl;
    return 0;
}
```

```
int x = 1, y = 2;

int main( ) {
    int x = 3, y = 4;
    if(true){
            int x = 1, y = 2;
            cout << x+y << endl;
    }
    cout << x+y << endl;
    return 0;
}
```

- 可见性：对于程序的某一特定点，哪些符号是可以访问的。

① 某变量作用域范围内，该变量是可视的

【举例】

```
int x = 1, y = 2;
void f(int c, int d) {
    int sum;
    cout << x + y << endl;
}
int main( ) {
    int x = 3, y = 4;
    int a = 5, b = 6;
    f(a, b);
    cout << x + y << endl;
    return 0;
}
```

② 函数形参影响变量可视性

【举例】

```
int x = 1, y = 2;
void f(int x, int y) {
    int sum;
    cout << x + y << endl;
}
int main( ) {
    int x = 3, y = 4;
    int a = 5, b = 6;
    f(a, b);
    cout << x + y << endl;
    return 0;
}
```

③ 复合语句分程序结构影响变量可视性

【举例】

```
int x = 1, y = 2;

int main( ) {
    int x = 3, y = 4;
    if(true){
            int x = 1, y = 2;
            cout << x+y << endl;
    }
    cout << x+y << endl;
    return 0;
}
```

- 根据作用域进行符号表组织（自学）
  ① 单符号表组织：所有嵌套的作用域，共用一个全局符号表
  ② 多符号表组织：每个作用域都有各自的符号表

5. 静态语义分析
   - 静态语义分析的主要任务

(1) 控制流检查

控制流语句必须使控制转移到合法的地方。

例如：

① 一个跳转语句跳转到的地方必须与跳转语句在同一个块内；

② break 语句必须有合法的语句包围它…

(2) 唯一性检查

一个对象在一个指定的上下文范围内只允许定义一次。

(3) 名字的上下文相关性检查

名字的出现应满足一定的上下文相关性。

例如：

① 变量在使用前必须声明；

② 外部不能访问私有变量；

③ 类声明和类实现之间要匹配；

④ 向对象发送消息时所调用的方法，使用相应类中合法的方法或继承方法…

(4) 类型检查

① 验证程序的结构是否匹配上下文所期望的类型

② 为代码生成阶段搜集及建立必要的类型信息

③ 实现某个类型系统

- 类型收集

【举例】**说明语句** 中各种变量的类型信息的语义规则：

(1)	D→TL	{ L.in := T.type }
(2)	T→char	{ T.type := char }
(3)	T→int	{ T.type := int }
(4)	T→float	{ T.type := float }
(5)	$L \rightarrow L^1, id$	{ L^1.in := L.in addtype(id.entry, L.in) }
(6)	L→id	{ addtype(id.entry, L.in) }

- 类型检查

【举例】表达式类型检查和求值：

（假设：类型不同的两个变量进行运算则语义错误。）

```
(1) L→E      { print(E.val); }
(2) E→E1+T   { if(E1.type==T.type)
               { E.type:=E1.type;
                 E.val:=E1.val+T.val; }
               else error(); }
(3) E→T      { E.type:=T.type;  E.val:=T.val }
(4) T→T1*F
(5) T→F
(6) F→(E)
(7) F→id     { getType(F.type, id.entry);
               F.val:=id.lexval; }
```

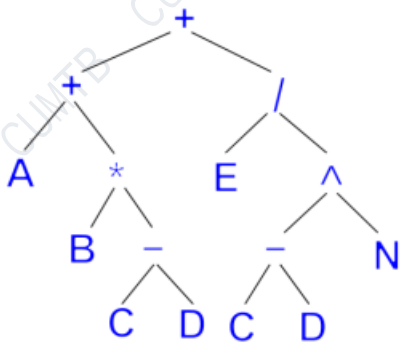
6. 中间代码的形式

- 中间代码的作用
  ① 源语言和目标语言之间的桥梁，使编译程序逻辑更加简单明确；
  ② 利于编译程序的重定向；
  ③ 利于进行与目标机无关的优化。

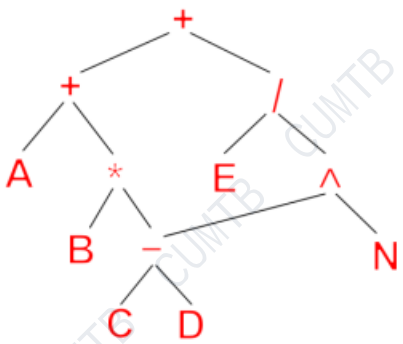
- 常见形式
  (1) **AST**(Abstract Syntax Tree，抽象语法树，简称语法树)
  DAG(Directed Acyclic Graph，有向无圈图) AST 改进形式
  (2) **TAC**(Three-Address Code，**三地址码**或**四元式**)
  (3) P-code(Pascal 编译器的输出)
  (4) Bytecode(Java 编译器的输出)
  (5) **SSA**(Static Single Assignment Form，静态单赋值形式)

- **AST、DAG**

【举例】A+B*(C-D)+E/(C-D)^N 的 AST 和 DAG 表示



AST



DAG

- **TAC——三地址码、四元式**

格式：(算符，第一运算对象，第二运算对象，结果)

特点：利于代码优化和目标代码生成

【举例】写出 A+B*(C-D)+E/(C-D)^N 的四元式表示

(-, C, D, T1)	或	T1 := C - D
(*, B, T1, T2)		T2 := B * T1
(+, A, T2, T3)		T3 := A + T2
(-, C, D, T4)		T4 := C - D
(^, T4, N, T5)		T5 := T4 ^ N
(/, E, T5, T6)		T6 := E / T5
(+, T3, T6, T7)		T7 := T3 + T6

【课堂练习】写出下式的四元式序列

(1) a*(-b+c)

(2) -a+b*(-c+d)/e

(3) A∨B∧¬ C∨D

(4) A∨B∧(¬ C∨D)

(5) (A∨B)∧(¬ C∨D)

答：

a*(-b+c)	T1 := - b T2 := T1 + c T3 := a * T2
-a+b*(-c+d)/e	T1 := - a T2 := - c T3 := T2 + d T4 := b * T3 T5 := T4 / e T6 := T1 + T5
A∨B∧¬ C∨D	T1 := ¬ C T2 := B ∧ T1 T3 := A ∨ T2 T4 := T3 ∨ D
A∨B∧(¬ C∨D)	T1 := ¬ C T2 := T1 ∨ D T3 := B ∧ T2 T4 := A ∨ T3
(A∨B)∧(¬ C∨D)	T1 := A ∨ B T2 := ¬ C T3 := T2 ∨ D T4 := T1 ∧ T3

- **SSA——静态单赋值形式**

单赋值：程序里的名字仅有一次赋值

SSA 的特点：使用一个名字时仅关联于唯一的“定值点”

SSA 的优势：有利于后续的程序分析和优化

【举例】

```
x := 5                    SSA:    x1 := 5
x := x - 3                        x2 := x1 - 3
if x < 3 then                     if x2 < 3 then
    y := x * 2                        y1 := x2 * 2
    w := y                            w1 := y1
else                              else
    y := x - 3                        y2 := x2 - 3
w := x - y                        y3 := Φ(y1, y2)
z := x + y                        w2 := x2 - y3
                                  z1 := x2 + y3
```

- 当前的编译程序多使用 AST，再从 AST 表示成 TAC。该过程也伴随着代码优化。

7. 生成三地址码
   (1) 赋值语句及算数表达式的翻译
   (2) 说明语句的翻译（自学）
   (3) 数组说明和数组元素引用的翻译（自学）
   (4) 布尔表达式的翻译（自学）
   (5) 控制语句的翻译（自学）
   (6) 拉链与代码回填（自学）
   (7) 过程调用的翻译（自学）

【举例】将如下**赋值语句**翻译成四元式的语义描述

① S → id := A
② A → id
③ A → int
④ A → real
⑤ A → A1 + A2
⑥ A → A1 * A2
⑦ A → -A1
⑧ A → (A1)

句子 y := 5.1 * x + b

S →id := A { S.code:=A.code || gen(id.place ':=' A.place); }

- ✧ S.code（综合属性）表示 S 的四元式语句序列；
- ✧ A.code（综合属性）表示 A 的四元式语句序列；
- ✧ A.place 是综合属性，表示存放 A 的值的存储位置；
- ✧ id.place 表示相应的符号名对应的存储位置；
- ✧ || 是四元式语句序列之间的链接运算；
- ✧ gen 是语义函数，用于生成一条四元式语句。

```
A→id {  A.place:=id.place;
        A.code:=" ";
     }
```

```
A→int { A.place:=newtemp;
        A.code:=gen(A.place ':=' int.val);
      }

A→real { A.place:=newtemp;
         A.code:=gen(A.place ':=' real.val);
       }
```

✧ newtemp 是语义函数，用于在符号表中新建一个从未使用过的名字，并返回该名字的存储位置。

$A \rightarrow A_1+A_2$
{ A.place:= newtemp;
A.code:= A_1.code || A_2.code || gen(A.place ':=' A_1.place '+' A_2.place);
}

$A \rightarrow A_1*A_2$
{ A.place:= newtemp;
A.code:= A_1.code || A_2.code || gen(A.place ':=' A_1.place '*' A_2.place);
}

$A \rightarrow -A_1$
{ A.place:= newtemp;
A.code:= A_1.code || gen(A.place ':=' 'uminus' A_1.place);
}

$A \rightarrow (A_1)$
{ A.place:= A_1.place;
A.code:= A_1.code;
}

8. 语义分析和中间代码生成的多遍方法

- 在实际编译过程中，静态语义分析和中间代码生成通常采用多遍的方法，例如：第一遍创建符号表，第二遍进行静态语义分析，第三遍生成四元式序列。

- 为实现对语法树的多次遍历，通常采用 visitor 设计模式来实现对所有结点进行同类处理、不同结点又可以有不同的行为。

【本章小结】

1. 符号表
   ① 符号属性
   ② 符号表的实现
      - 总体组织
      - 表项组织（线性表、有序表、二叉搜索树、Hash 表）
      - 表的操作（创建表、释放表；插入、删除、修改、查询表项）
   ③ 作用域和可见性
   ④ 目标代码运行时的存储分配

2. 静态语义分析的任务

3. 中间代码形式（逆波兰式、四元式）

4. 中间代码生成（赋值语句及算术表达式的翻译）