中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

编译器构造实验

Lab 2 – 语义分析

Teaching Team: 王焱林、伍嘉栋、黄炎贤、王岩立、张俊鹏

特别说明

- 本课程需要和理论课《编译原理》配套学习
- 作业迟交：每天扣10%的作业分
- 作业抄袭：0分

2.1 实验内容

2.1.1 实验要求

- 在本次实验中，对C--语言做如下假设
  - 假设1：整型（int）变量不能与浮点型（float）变量相互赋值或者相互运算。
  - 假设2：仅有int型变量才能进行逻辑运算或者作为if和while语句的条件；仅有int型和float型变量才能参与算术运算。
  - 假设3：任何函数只进行一次定义，无法进行函数声明。
  - 假设4：所有变量（包括函数的形参）的作用域都是全局的，即程序中所有变量均不能重名。
  - 假设5：结构体间的类型等价机制采用**名等价（Name Equivalence）**的方式。
  - 假设6：函数无法进行嵌套定义。
  - 假设7：结构体中的域不与变量重名，并且不同结构体中的域互不重名。

2.1.1 实验要求

- 你的程序需要对输入文件进行语义分析（输入文件中可能包含函数、结构体、一维和高维数组）并检查如下类型的错误：
  - 错误类型1：变量在使用时未经定义。
  - 错误类型2：函数在调用时未经定义。
  - 错误类型3：变量出现重复定义，或变量与前面定义过的结构体名字重复。
  - 错误类型4：函数出现重复定义（即同样的函数名出现了不止一次定义）。
  - 错误类型5：赋值号两边的表达式类型不匹配。
  - 错误类型6：赋值号左边出现一个只有右值的表达式。

2.1.1 实验要求

- 你的程序需要对输入文件进行语义分析（输入文件中可能包含函数、结构体、一维和高维 数组）并检查如下类型的错误：
  - 错误类型7：操作数类型不匹配或操作数类型与操作符不匹配（例如整型变量与数组变量相加减，或数组（或结构体）变量与数组（或结构体）变量相加减）。
  - 错误类型8：return语句的返回类型与函数定义的返回类型不匹配。
  - 错误类型9：函数调用时实参与形参的数目或类型不匹配。
  - 错误类型10：对非数组型变量使用“[…]”（数组访问）操作符。
  - 错误类型11：对普通变量使用“(…)”或“()”（函数调用）操作符。
  - 错误类型12：数组访问操作符“[…]”中出现非整数（例如a[1.5]）。

2.1.1 实验要求

- 你的程序需要对输入文件进行语义分析（输入文件中可能包含函数、结构体、一维和高维 数组）并检查如下类型的错误：
  - 错误类型13：对非结构体型变量使用“.”操作符。
  - 错误类型14：访问结构体中未定义过的域。
  - 错误类型15：结构体中域名重复定义（指同一结构体中），或在定义时对域进行初始化（例如struct A { int a = 0; }）。
  - 错误类型16：结构体的名字与前面定义过的结构体或变量的名字重复。
  - 错误类型17：直接使用未定义过的结构体来定义变量。

2.1.1 实验要求

- 注意三点：
  - 关于数组类型的等价机制，同C语言一样，只要数组的基类型和维数相同即认为类型是匹配的，例如int a[10][2]和int b[5][3]即属于同一类型；
  - 允许类型等价的结构体变量之间的直接赋值（见后面的测试样例），这时的语义是，对应的域相应赋值（数组域也如此，按相对地址赋值直至所有数组元素赋值完毕或目标数组域已经填满）；
  - 对于结构体类型等价的判定，每个匿名的结构体类型认为均具有一个独有的隐藏名字，以此进行名等价判定。

2.1.1 实验要求

- 可以选择完成以下部分或全部的要求：
  - 要求1：修改前面的C--语言假设3，使其变为“函数除了在定义之外还可以进行声明”。函数的定义仍然不可以重复出现，但函数的声明在相互一致的情况下可以重复出现。任一函数无论声明与否，其定义必须在源文件中出现。在新的假设3下，你的程序还需要检查两类新的错误和增加新的产生式
    - 错误类型18：函数进行了声明，但没有被定义。
    - 错误类型19：函数的多次声明互相冲突（即函数名一致，但返回类型、形参数量或者形参类型不一致），或者声明与定义之间互相冲突。
    - 由于C--语言文法中并没有与函数声明相关的产生式，因此你需要先对该文法进行适当修改。对于函数声明来说，我们并不要求支持像“int foo(int, float)”这样省略参数名的函数声明。在修改的时候要留意，你的改动应该以不影响其它错误类型 的检查为原则。

2.1.1 实验要求

- 可以选择完成以下部分或全部的要求：
  - 要求2：修改前面的C--语言假设4，使其变为“变量的定义受可嵌套作用域的影响，外层语句块中定义的变量可在内层语句块中重复定义（但此时在内层语句块中就无法访问到外层语句块的同名变量），内层语句块中定义的变量到了外层语句块中就会消亡，不同函数体内定义的局部变量可以相互重名”。在新的假设4下，完成错误类型1至17的检查。

2.1.1 实验要求

- 可以选择完成以下部分或全部的要求：
  - 要求3：修改前面的C--语言假设5，将结构体间的类型等价机制由名等价改为结构等价（Structural Equivalence）。例如，虽然名称不同，但两个结构体类型struct a { int x; float y; }和struct b { int y; float z; }仍然是等价的类型。注意，在结构等价时不要将数组展开来判断，例如struct A { int a; struct { float f; int i; } b[10]; }和struct B { struct { int i; float f; } b[10]; int b;}是不等价的。在新的假设5下，完成错误类型1至17的检查。

2.1.2 输入格式

- 输入是一个包含C--源代码的文本文件，该源代码中可能会有语义错误
- 接收一个输入文件名作为参数。例如，假设你的程序名为cc、输入文件名为test1、程序和输入文件都位于当前目录下，那么在Linux命令行下运行./cc test1即可获得以test1作为输入文件的输出结果。

2.1.3 输出格式

- 要求通过标准输出打印程序的运行结果
- 对于那些没有语义错误的输入文件，不需要输出任何内容;
- 对于那些存在语义错误的输入文件，输出相应的错误信息，包括错误类型、出错的行号以及说明文字，其格式为：
  - Error type [错误类型] at Line [行号]: [说明文字]
  - 说明文字的内容没有具体要求，但是错误类型和出错的行号一定要正确，因为这是判断输出的错误提示信息是否正确的唯一标准。
  - 严格遵守实验要求中给定的错误分类，否则将影响实验评分。

2.1.3 输出格式

- 输入文件中可能包含一个或者多个错误（但每行最多只有一个错误），需要将它们全部检查出来。
- 对于输入文件中的一个错误所产生的连锁反应，导致别的地方出现多个错误（例如，一个未定义的变量在使用时由于无法确定其类型，会使所有包含该变量的表达式产生类型错误），只考察是否报告了较本质的错误（如果难以确定哪个错误更本质一些，建议报告所有发现的错误）。
- 如果源程序里有错但没有报错或报告的错误类型不对，又或者源程序里没有错但报错，都会影响本次实验评分。

2.1.4 测试环境

- 你的程序将在如下环境中被编译并运行（同实验一）：
  - GNU Linux Release: Ubuntu 12.04, kernel version 3.2.0-29
  - GCC version 4.6.3
  - GNU Flex version 2.5.35
  - GNU Bison version 2.5
  - 可以使用其它版本的Linux或者GCC等

2.1.5 提交要求

- 实验二要求提交如下内容（同实验一）：
  - 提交链接：https://send2me.cn/wABEcf9S/Tz6EYvqLdQJUig
  - 截止时间：2023-10-26 23:59:59
  - Flex、 Bison以及C语言的可被正确编译运行的源程序。
  - 一份PDF格式的实验报告，内容包括：
    - 程序实现了哪些功能？简要说明如何实现这些功能
    - 程序应该如何被编译？可以使用脚本、makefile或逐条输入命令进行编译，请详细说明应该如何编译程序

2.1.6 样例

- 样例1：
  - 输入：

```
1  int main()
2  {
3    int i = 0;
4    j = i + 1;
5  }
```

  - 输出：
    - 样例输入中变量“j”未定义，因此你的程序可以输出如下的错误提示信息：

```
Error type 1 at Line 4: Undefined variable "j".
```

2.1.6 样例

- 样例2：
  - 输入：

```
1  int main()
2  {
3    int i = 0;
4    inc(i);
5  }
```

  - 输出：
    - 样例输入中函数“inc”未定义，因此你的程序可以输出如下的错误提示信息：

```
Error type 2 at Line 4: Undefined function "inc".
```

2.1.6 样例

- 样例3：
  - 输入：

```
1  int main()
2  {
3    int i, j;
4    int i;
5  }
```

  - 输出：
    - 样例输入中变量“i”被重复定义，因此你的程序可以输出如下的错误提示信息：

```
Error type 3 at Line 4: Redefined variable "i".
```

2.1.6 样例

- 样例4：

```
1  int func(int i)
2  {
3    return i;
4  }
5
6  int func()
7  {
8    return 0;
9  }
10
11 int main()
12 {
13 }
```

  - 输入：
  - 输出：
    - 样例输入中函数“func”被重复定义，因此你的程序可以输出如下的错误提示信息：

```
Error type 4 at Line 6: Redefined function "func".
```

2.1.6 样例

- 样例5：
  - 输入：

```
1  int main()
2  {
3    int i;
4    i = 3.7;
5  }
```

  - 输出：
    - 样例输入中错将一个浮点常数赋值给一个整型变量，因此你的程序可以输出如下的错误提示信息：

```
Error type 5 at Line 4: Type mismatched for assignment.
```

2.1.6 样例

- 样例6：
  - 输入：

```
1  int main()
2  {
3    int i;
4    10 = i;
5  }
```

  - 输出：
    - 样例输入中整数“10”出现在了赋值号的左边，因此你的程序可以输出如下的错误提示信息：

```
Error type 6 at Line 4: The left-hand side of an assignment must be a varia-
   ble.
```

2.1.6 样例

- 样例7：
  - 输入：

```
1  int main()
2  {
3    float j;
4    10 + j;
5  }
```

  - 输出：
    - 样例输入中表达式“10 + j”的两个操作数的类型不匹配，因此你的程序可以输出如下的 错误提示信息：

```
Error type 7 at Line 4: Type mismatched for operands.
```

2.1.6 样例

➢ 样例8：

- 输入：

```
1  int main()
2  {
3    float j = 1.7;
4    return j;
5  }
```

- 输出：
  - 样例输入中“main”函数返回值的类型不正确，因此你的程序可以输出如下的错误提示信息：

```
Error type 8 at Line 4: Type mismatched for return.
```

2.1.6 样例

- 样例9：
  - 输入：

```
1  int func(int i)
2  {
3    return i;
4  }
5
6  int main()
7  {
8    func(1, 2);
9  }
```

  - 输出：
    - 样例输入中调用函数“func”时实参数目不正确，因此你的程序可以输出如下的错误提示信息：

```
Error type 9 at Line 8: Function "func(int)" is not applicable for arguments
  "(int, int)".
```

2.1.6 样例

- 样例10:
  - 输入:

```
1  int main()
2  {
3    int i;
4    i[0];
5  }
```

  - 输出:
    - 样例输入中变量“i”非数组型变量，因此你的程序可以输出如下的错误提示信息:

```
Error type 10 at Line 4: "i" is not an array.
```

2.1.6 样例

- 样例11：
  - 输入：

```
1  int main()
2  {
3    int i;
4    i(10);
5  }
```

  - 输出：
    - 样例输入中变量“i”不是函数，因此你的程序可以输出如下的错误提示信息：

```
Error type 11 at Line 4: "i" is not a function.
```

2.1.6 样例

- 样例I2:
  - 输入:

```
1  int main()
2  {
3    int i[10];
4    i[1.5] = 10;
5  }
```

  - 输出:
    - 样例输入中数组访问符中出现了非整型常数“1.5”，因此你的程序可以输出如下的错误提示信息:

```
Error type 12 at Line 4: "1.5" is not an integer.
```

2.1.6 样例

- 样例13：
  - 输入：

```
1  struct Position
2  {
3    float x, y;
4  };
5
6  int main()
7  {
8    int i;
9    i.x;
10 }
```

  - 输出：
    - 样例输入中变量“i”非结构体类型变量，因此你的程序可以输出如下的错误提示信息：

```
Error type 13 at Line 9: Illegal use of ".".
```

2.1.6 样例

- 样例I4:
  - 输入:

```
1  struct Position
2  {
3    float x, y;
4  };
5
6  int main()
7  {
8    struct Position p;
9    if (p.n == 3.7)
10     return 0;
11 }
```

  - 输出:
    - 样例输入中结构体变量“p”访问了未定义的域“n”，因此你的程序可以输出如下的错误提示信息:

```
Error type 14 at Line 9: Non-existent field "n".
```

2.1.6 样例

- 样例15：
  - 输入：

```
1  struct Position
2  {
3    float x, y;
4    int x;
5  };
6
7  int main()
8  {
9  }
```

  - 输出：
    - 样例输入中结构体的域“x”被重复定义，因此你的程序可以输出如下的错误信息：

```
Error type 15 at Line 4: Redefined field "x".
```

2.1.6 样例

➢ 样例16：

- 输入：

```
1  struct Position
2  {
3    float x;
4  };
5
6  struct Position
7  {
8    int y;
9  };
10
11 int main()
12 {
13 }
```

- 输出：
  - 样例输入中两个结构体的名字重复，因此你的程序可以输出如下的错误信息：

```
Error type 16 at Line 6: Duplicated name "Position".
```

2.1.6 样例

- 样例I7：
  - 输入：

```
1  int main()
2  {
3    struct Position pos;
4  }
```

  - 输出：
    - 样例输入中结构体“Position”未经定义，因此你的程序可以输出如下的错误信息：

```
Error type 17 at Line 3: Undefined structure "Position".
```

2.1.7 样例

要求1：函数除了在定义之外还可以进行声明

- 样例I8：

```
1  int func(int a);
2
3  int func(int a)
4  {
5    return 1;
6  }
7
8  int main()
9  {
10 }
```

  - 输入：
  - 输出：
    - 如果你的程序需要完成要求1，这个样例输入不存在任何词法、语法或语义错误，因此不需要输出。
    - 如果你的程序不需要完成要求1，这个样例输入存在语法错误，因此你的程序可以输出如下的错误提示信息：

```
Error type B at Line 1: Incomplete definition of function "func".
```

2.1.7 样例

要求1：函数除了在定义之外还可以进行声明

- 样例I9：
  - 输入：

```
1  struct Position
2  {
3    float x,y;
4  };
5
6  int func(int a);
7
8  int func(struct Position p);
9
10 int main()
11 {
12 }
```

  - 输出：
    - 如果你的程序需要完成要求1，这个样例输入存在两处语义错误：一是函数“func”的两次声明不一致；二是函数“func”未定义，因此你的程序可以输出如下的错误提示信息：

```
Error type 19 at Line 8: Inconsistent declaration of function "func".
Error type 18 at Line 6: Undefined function "func".
```

    - 注意，我们对错误提示信息的顺序不做要求。
    - 如果你的程序不需要完成要求1，这个样例输入存在两处语法错误，因此你的程序可以输出如下的错误提示信息：

```
Error type B at Line 6: Incomplete definition of function "func".
Error type B at Line 8: Incomplete definition of function "func".
```

2.1.7 样例

要求2：变量的定义受可嵌套作用域的影响， 外层语句块中定义的变量可在内层语句块中重复定义，内层语句块中定义的变量到了外层语句块中就会消亡，不同函数体内 定义的局部变量可以相互重名

- 样例20：
  - 输入：

```
1  int func()
2  {
3    int i = 10;
4    return i;
5  }
6
7  int main()
8  {
9    int i;
10   i = func();
11 }
```

  - 输出：
    - 如果你的程序需要完成要求2，这个样例输入不存在任何词法、语法或语义错误，因此不需要输出。
    - 如果你的程序不需要完成要求2，样例输入中的变量“i”被重复定义，因此你的程序可以输出如下的错误提示信息：

```
Error type 3 at Line 9: Redefined variable "i".
```

2.1.7 样例

要求2：变量的定义受可嵌套作用域的影响， 外层语句块中定义的变量可在内层语句块中重复定义，内层语句块中定义的变量到了外层语句块中就会消亡，不同函数体内 定义的局部变量可以相互重名

- 样例21：
  - 输入：

```
1  int func()
2  {
3    int i = 10;
4    return i;
5  }
6
7  int main()
8  {
9    int i;
10   int i, j;
11   i = func();
12 }
```

  - 输出：
    - 如果你的程序需要完成要求2，样例输入中的变量“i”被重复定义，因此你的程序可以 输出如下的错误提示信息：

```
Error type 3 at Line 10: Redefined variable "i".
```

    - 如果你的程序不需要完成要求2，样例输入中的变量“i”被重复定义了两次，因此你的程序可以输出如下的错误提示信息：

```
Error type 3 at Line 9: Redefined variable "i".
Error type 3 at Line 10: Redefined variable "i".
```

2.1.7 样例

要求3：结构体间的类型等价机制由名等价改为结构等价

➢ 样例22：

- 输入：

```
1  struct Temp1
2  {
3    int i;
4    float j;
5  };
6
7  struct Temp2
8  {
9    int x;
10   float y;
11 };
12
13 int main()
14 {
15   struct Temp1 t1;
16   struct Temp2 t2;
17   t1 = t2;
18 }
```

- 输出：
  - 如果你的程序需要完成要求3，这个样例输入不存在任何词法、语法或语义错误，因此不需要输出。
  - 如果你的程序不需要完成要求3，样例输入中的语句“t1 = t2;”其赋值号两边变量的类型不匹配，因此你的程序可以输出如下的错误提示信息：

```
Error type 5 at Line 17: Type mismatched for assignment.
```

2.1.7 样例

要求3：结构体间的类型等价机制由名等价改为结构等价

➢ 样例23：

- 输入：

```
1  struct Temp1
2  {
3    int i;
4    float j;
5  };
6
7  struct Temp2
8  {
9    int x;
10 };
11
12 int main()
13 {
14   struct Temp1 t1;
15   struct Temp2 t2;
16   t1 = t2;
17 }
```

- 输出：
  - 如果你的程序需要完成要求3，样例输入中的语句“t1 = t2;”其赋值号两边变量的类型不匹配，因此你的程序可以输出如下的错误提示信息：

```
Error type 5 at Line 16: Type mismatched for assignment.
```

  - 如果你的程序不需要完成要求3，应该输出与上述一样的错误提示信息：

```
Error type 5 at Line 16: Type mismatched for assignment.
```

2.2 实验指导

2.2.1 属性文法

- 核心思想：为上下文无关文法中的每一个终结符或非终结符赋予一个或多个属性值。
- 对于产生式A → X1 … Xn来说，在自底向上分析中X1…Xn的属性值是已知的，这样语义动作只会为A计算属性值；而在自顶向下分析中，A的属性值是已知的，在该产生式被应用之后才能知道X1…Xn的属性值。
- 终结符号的属性值通过词法分析可以得到，非终结符号的属性值通过产生式对应的语义动作来计算。

2.2.1 属性文法

- 综合属性（Synthesized Attribute）：一个结点的综合属性值是从其子结点的属性值计算而来；
- 继承属性（Inherited Attribute）：一个结点的继承属性值则是由该结点的父结点和兄弟结点的属性值计算而来
- S属性文法：如果 对一个文法P，$\forall A \to X_1 \cdots X_n \in P$ 都有与之相关联的若干个属性定义规则，则称P为属性文法。如果属性文法P只包含综合属性而没有继承属性，则称P为S属性文法。
- L属性文法：如果每个属性定义规则中的每个属性要么是一个综合属性，要么是Xj的一个继承属性，并且该继承属性只依赖于X1…Xj−1的属性和A的继承属性，则称P为L属性文法。

2.2.1 属性文法

- 语法制导翻译（Syntax-Directed Translation或SDT）：在SDT中，属性文法中的属性定义规则用计算属性值的语义动作来表示，并用花括号“{”和“}”括起来，它们可被插入到产生式右部的任何合适的位置上，这是一种语法分析和语义动作交错的表示法。

2.2.2 符号表

- 符号表：记录源程序中各种名字的特性信息
- 所谓“名字”包括：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等
- 所谓“特性信息”包括：上述名字的种类、具体类型、维数、参数个数、数值及目标地址（存储单元地址）等

2.2.2 符号表

- 符号表操作
  - 填表：当分析到程序中的说明或定义语句时，应将说明或定义的名字，以及与之有关的特性信息填入符号表中
  - 查表：查表操作则使用得更为广泛，需要使用查表操作的情况有：
    - 填表前查表，包括检查在输入程序的同一作用域内名字是否被重复定义，检查名字的种类是否与说明一致，对于那些类型要求更强的语言，则要检查表达式中各变量的类型是否一致等；
    - 此外生成目标指令时，也需要查表以取得所需要的地址或者寄存器编号等

2.2.2 符号表

➢ 符号表组织方式

- 可以将程序中出现的所有符号组织成一张表，也可以将不同种类的符号组织成不同的表（例如，所有变量名组织成一张表，所有函数名组织成一张表，所有临时变量组织成一张表，所有结构体定义组织成一张表，等等）
- 可以针对每个语句块、每个结构体都新建一张表，也可以将所有语句块中出现的符号全部插入到同一张表中
- 符号表可以仅支持插入操作而不支持删除操作（此时如果要实现作用域则需要将符号表组织成层次结构），也可以组织一张既可以插入又可以删除的、支持动态更新的表

2.2.2 符号表

- 实现符号表的数据结构——线性链表：
  - 符号表里所有的符号（假设有n个，下同）都用一条链表串起来，插入一个新的符号只需将该符号放在链表的表头，其时间复杂度是O(1)。在链表中查找一个符号需要对其进行遍历，时间复杂度是O(n)。删除一个符号只需要将该符号从链表里摘下来，不过在摘之前由于我们必须要执行一次查找操作以找到待删除的结点，因此时间复杂度也是O(n)。
  - 缺点：查找和删除效率太低，一旦符号表中的符号数量较大，查表操作将变得十分耗时
  - 优点：它的结构简单，编程容易，可以被快速实现

2.2.2 符号表

- 实现符号表的数据结构——平衡二叉树：
  - 在一个典型的平衡二叉树实现（例如AVL树、红黑树或伸展树I等）上查找一个符号的时间复杂度是O(logn)。插入一个符号相当于进行一次失败的查找而找到待插入的位置，时间复杂度也是O(logn)。删除一个符号可能需要做更多的维护操作，但其时间复杂度仍然维持在O(logn)的级别。
  - 优点：较高的搜索效率（在绝大多数应用中O(logn)的搜索效率已经完全可以接受）以及较好的空间效率（它所占用的空间随树中结点的增多而增长，不像散列表那样每张表都需要大量的空间）。
  - 缺点：平衡二叉树的缺点是编程难度高

2.2.2 符号表

- 实现符号表的数据结构——散列表：
  - 散列表是一种可以达到搜索效率极致的数据结构。一个好的散列表实现可以让插入、查找和删除的平均时间复杂度都达到O(1)。
  - 散列表在代码实现上也很简单：申请一个大数组，计算一个散列函数的值，然后根据该值将对应的符号放到数组相应下标的位置即可。
    - 对于符号表来说，一个最简单的散列函数（即hash函数）可以把符号名中的所有字符相加，然后对符号表的大小取模
    - 更好的hash函数

```
1  unsignedinthash_pjw(char* name)
2  {
3    unsignedint val = 0, i;
4    for (; *name; ++name)
5    {
6      val = (val << 2) + *name;
7      if (i = val & ~0x3fff) val = (val ^ (i >> 12)) & 0x3fff;
8    }
9    return val;
10 }
```

2.2.2 符号表

- 实现符号表的数据结构——Multiset Discrimination：
  - 散列表在最坏情况下它会退化为O(n)的线性查找，而且几乎任何确定的散列函数都存在某种最坏的输入。
  - 散列表所要申请的内存空间往往比输入程序中出现的所有符号的数量还要多，较为浪费。
  - **Multiset discrimination4**：只为输入程序中出现的每个符号单独分配一个编号和空间。在词法分析部分，我们先统计输入程序中出现的所有符号（包括变量名、函数名等），然后把这些符号按照名字进行排序，最后申请一张与符号总数量一样大的符号表，查表功能可通过基于符号名的二分查找实现。

2.2.3 支持多层作用域的符号表

➢ Functional Style：维护一个符号表栈

- 出现被“{”和“}”包含的语句块则将f的符号表压栈，然后新建一个符号表，这个符号表里只有变量a的定义
- 当语句块中出现任何表达式使用到某个变量时，编译器先查找当前符号表，如果找到就使用这个符号表里的该变量，否则顺着符号表栈向下逐个符号表进行查找，使用第一个查找成功的符号表里的相应变量，如果查遍所有的符号表都找不到则报告当前语句出现了变量未定义的错误
- 离开某个语句块时须销毁当前的符号表，再从栈中弹一个符号表出来作为当前的符号表
- 最多申请d（语句块的最大嵌套层数）个符号表。比较适合于采用链表或红黑树数据结构的符号表实现

- 编译器不能在“int a = c * 2;”这个地方报错
- 语句“a = a + b;”中a的值应该取外层定义中a的值；
- 语句“b = b – a;”中的a的值应该是if语句内部定义的a的值
- b的值都应该取外层定义中b的值

```
1  …
2  int f()
3  {
4    int a, b, c;
5    …
6    a = a + b;
7    if (b > 0)
8    {
9      int a = c * 2;
10     b = b - a;
11   }
12   …
13 }
```

2.2.3 支持多层作用域的符号表

➢ Imperative Style：单个符号表上进行动态维护

- 当编译器发现函数中出现了一个被“{”和“}”包含的语句块，而在这个语句块中又有新的变量定义时，它会将该变量插入 f 的符号表里
- 当语句块中出现任何表达式使用某个变量时，编译器就查找f的符号表。
  - 查找失败：报告一个变量未定义的错误；
  - 查表成功：返回查到的变量定义
  - 变量既在外层又在内层被定义：返回最近的那个定义
- 离开某个语句块时，将这个语句块中定义的变量全部从表中删除
- 对符号表的数据结构有一定的要求

- 编译器不能在“int a = c * 2;”这个地方报错
- 语句“a = a + b;”中的a的值应该取外层定义中a的值，语句“b = b – a;”中的a的值应该是if语句内部定义的a的值，这两个语句中b的值都应该取外层定义中b的值

```
1  …
2  int f()
3  {
4    int a, b, c;
5    …
6    a = a + b;
7    if (b > 0)
8    {
9      int a = c * 2;
10     b = b - a;
11   }
12   …
13 }
```

2.2.3 支持多层作用域的符号表

- **Hash table**：将不同层中同名的变量串在一起。其中i、j这两个变量有同名情况，被分配到散列表的同一个槽内。
- **Stack**：将属于同一层作用域的所有变量都串起来。图中，a、x同属最外层定义的变量，i、j、var同属中间一层定义的变量，i、j同属最内层定义的变量。
- 向散列表中插入元素时：将新插入的元素放到该槽下挂的链表和该层所对应的链表的表头
- 每次查表时如果定位到某个槽，则按顺序遍历这个槽下挂的链表并返回这个槽中符合条件的第一个变量
- 每次进入一个语句块，需要为这一层语句块新建一个链表用来串联该层中新定义的变量
- 每次离开一个语句块，则需要顺着代表该层语句块的链表将所有本层定义变量全部删除。

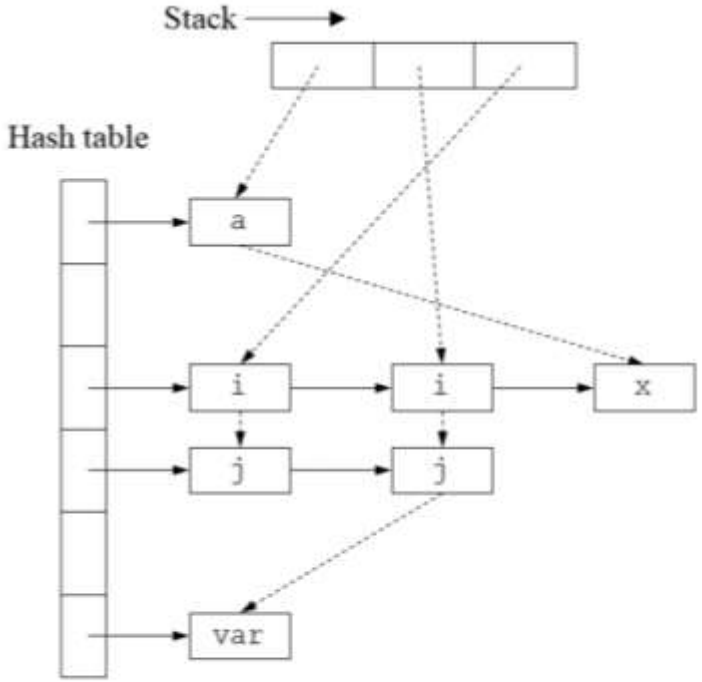

图1. 基于十字链表和open hashing散列表的符号表。

2.2.4 类型表示

- “类型”包含两个要素：一组值，以及在这组值上的一系列操作
- 一个典型程序设计语言的类型系统应该包含如下四个部分：
  - 1) 一组基本类型。在C--语言中，基本类型包括int和float两种。
  - 2) 从一组类型构造新类型的规则。在C--语言中，可以通过定义数组和结构体来构造新的类型。
  - 3) 判断两个类型是否等价的机制。在C--语言中，默认要求实现名等价，如果你的程序 需要完成要求2.3，则需实现结构等价。
  - 4) 从变量的类型推断表达式类型的规则。

2.2.4 类型表示

- 链表表示：多维数组的每一维都可以作为一个链表结点，每个链表结点存两个内容：数组元素的类型，以及数组的大小。例如，int a[10][3]可以表示为图2所示的形式。

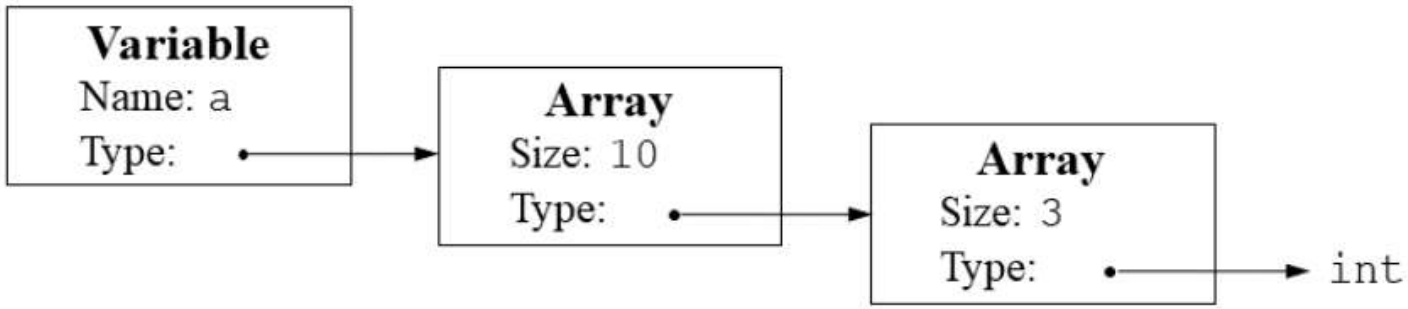

图2. 多维数组的链表表示示例。

2.2.4 类型表示

- 结构体同样也可以使用链表保存。例如，结构体struct SomeStruct { float f; float array[5]; int array2[10][10]; }可以表示为图3所示的形式。

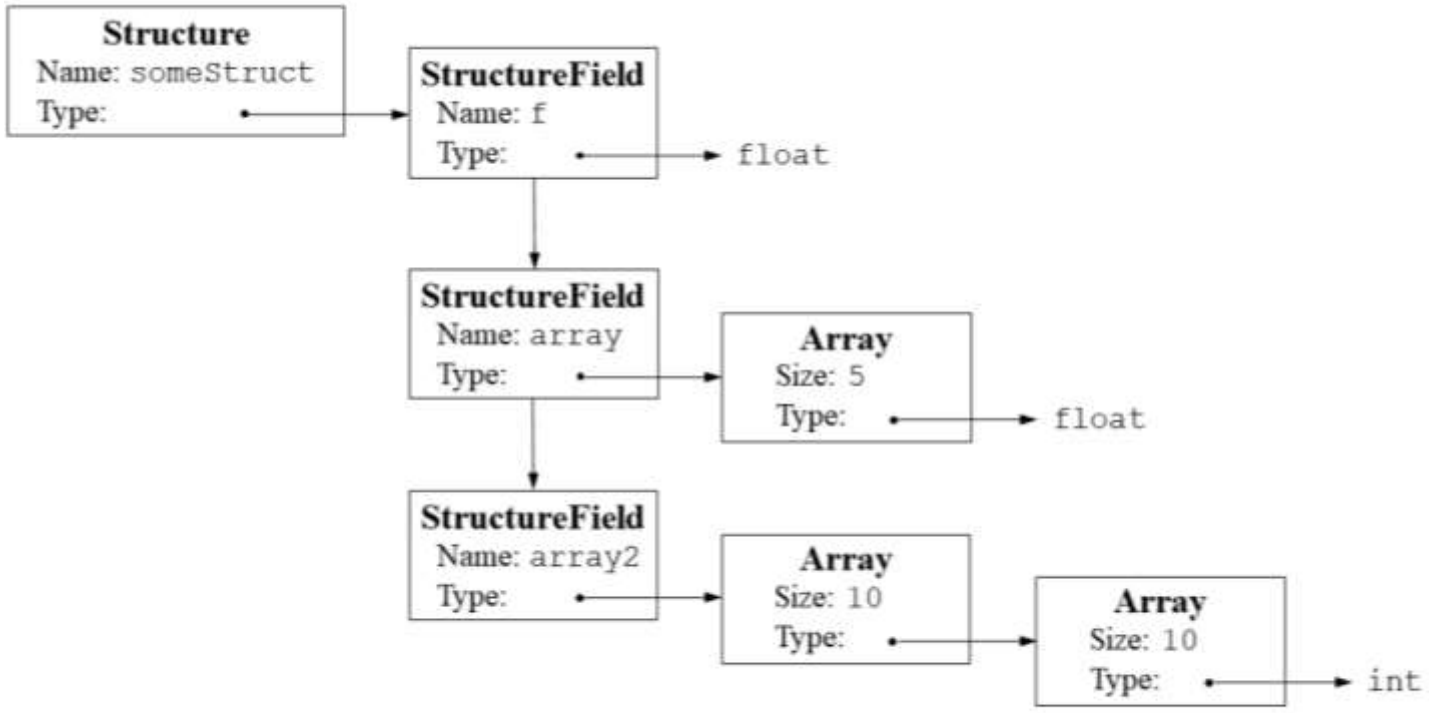

图3. 结构体的链表表示示例。

```
1  typedef struct Type_* Type;
2  typedef struct FieldList_* FieldList;
3
4  struct Type_
5  {
6    enum { BASIC, ARRAY, STRUCTURE } kind;
7    union
8    {
9      // 基本类型
10     int basic;
11     // 数组类型信息包括元素类型与数组大小构成
12     struct { Type elem; int size; } array;
13     // 结构体类型信息是一个链表
14     FieldList structure;
15   } u;
16 };
17
18 struct FieldList_
19 {
20   char* name;  // 域的名字
21   Type type;  // 域的类型
```

2.2.5 语义分析

- 实验二需要在实验一的基础上完成，特别是需要在实验一所构建的语法树上完成。
- 实验二仍然需要对语法树进行遍历以进行符号表的相关操作以及类型的构造与检查。
- 可以模仿SDT在Bison代码中插入语义分析的代码；更推荐的做法是，Bison代码只用于构造语法树，而把和语义分析相关的代码都放到一个单独的文件中去。

2.2.5 语义分析

- 每当遇到语法单元ExtDef或者Def，就说明该结点的子结点们包含了变量或者函数的定义信息，这时候应当将这些信息通过对子结点们的遍历提炼出来并插入到符号表里。
- 每当遇到语法单元Exp，说明该结点及其子结点们会对变量或者函数进行使用，这个时候应当查符号表以确认这些变量或者函数是否存在以及它们的类型是什么。
- 在发现一个语义错误之后不要立即退出程序，因为实验要求中有说明需要你的程序有能力查出输入程序中的多个错误。

2.2.5 语义分析

- 有关左值的错误：
  - 左值代表地址，可以出现在赋值号的左边或者右边
  - 右值代表数值，只能出现在赋值号的右边
  - 变量、数组访问以及结构体访问一般既有左值又有右值
  - 常数、表达式和函数调用一般只有右值而没有左值。
  - 例如，赋值表达式x = 3是合法的，但3 = x是不合法的；y = x + 3是合法的，但x + 3 = y是不合法的。
- 可以只从语法层面来检查左值错误：赋值号左边能出现的只有ID、Exp LB Exp RB以及Exp DOT ID，而不能是其它形式的语法单元组合。

2.2.5 语义分析

- 要求2.1与函数声明有关，函数声明需要你在语法中添加产生式，并在符号表中记录每个函数当前的状态：是被实现了，还是只被声明未被实现。
- 要求2.2涉及作用域，作用域的实现方法前文已经讨论过。
- 要求2.3为实现结构等价，对于结构等价来说，你只需要在判断两个类型是否相等时不是直接去比较类型名，而是针对结构体中的每个域逐个进行类型比较即可。