实验报告（二）

- 曹洋笛 161220004 2904428882@qq.com

一、数据结构

- 类型 Type

```
struct Type {
  Kind kind; // 类型，int/float/array/struct/undefined
  union {
    bool isRight; // int/float/undefined专用，是否是右值
    struct { Type* eleType; int length; } array; // array专用，元素类型和元素个数
    struct { TypeNode* node; char* name; } structure; // struct专用，域的链表和结构体名
  };};
```

- 函数 Function

```
struct Function {
  char* name; // 函数名
  int lineno; // 所在行数（函数名位置）
  bool isDefined; // 是否定义（一旦存在，默认已经声明）
  Type* returnType; // 返回类型
  TypeNode* paramNode; // 参数链表
};
```

- 临时变量 TypeNode（局部/全局变量，或结构体的一个域，或函数的一个参数）

```
struct TypeNode {
  Type* type; // 域/参数/变量的类型
  char* name; // 域/参数/变量的id
  int lineno; // 所在行数
  TypeNode* next; // 下一个节点
};
```

- 作用域 FieldNode

```
struct FieldNode {
  FieldType type; // 作用域种类，包括GLOBAL, FUNCTION, COND_LOOP(条件/循环), F_ANONY(匿名)
  FieldNode* parent; // 其外部作用域（一个）
  Function* func; // 对于函数作用域，记录函数信息（一个）
  int varListLen; // 本作用域内的变量符号表长度
  SymElem* varSymList; // 本作用域内的变量符号表（已按id排序的有序数组，便于二分查找）
};
```

- 符号表数组元素 SymElem（变量/函数/结构体符号表）

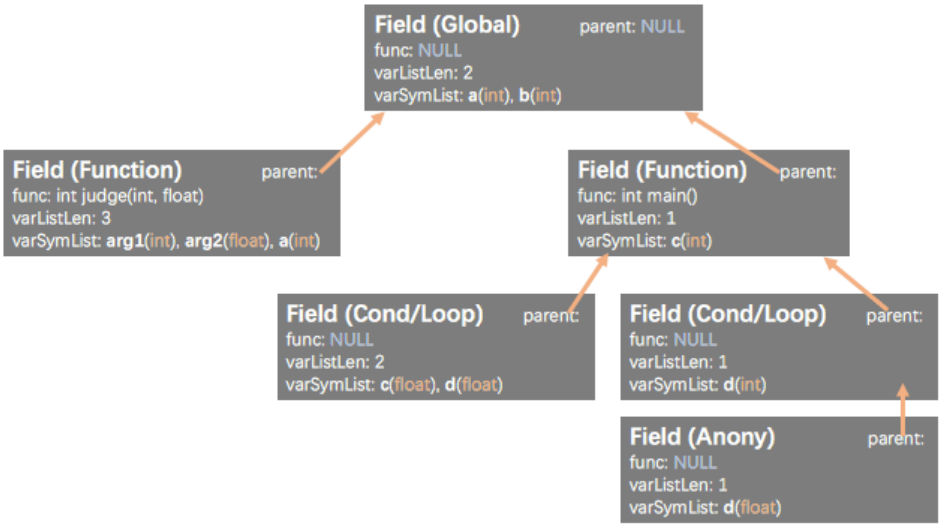
```
struct SymElem {
  bool isNull; // 是否为空
  char* name; // 变量名/函数名/结构体名
  union {
    Type* type; // 对于变量类型或结构体结构，其类型信息
    Function* func; // 对于函数，其函数信息
  };};
```

二、亮点

1. Field Tree —— 作用域的灵活处理

   以下面一段代码得到的作用域树为例：

```
int a, b;
int judge(int arg1, float arg2) { // 函数作用域
  int a = 3; // 局部的a，不是全局的a
  return ((arg1 < a) && (arg2 < b)); }
int main() { // 函数作用域
  int c; // 局部的c
  a = 1; b = 5; // 全局的a和b
  c = judge(2, 2.5); // True
  if (c) {
    float c = 2.5, d = c * 1.5; // 局部的c和d
  } else {
    int d = c + 3; // main的c，局部的d
    { // 匿名作用域
      float d = 0.5; // 局部的d
      a + b + c; // 全局的a和b，main的c
    }   }  }
```



   作用域的种类包括全局作用域（仅一个）、函数作用域（ExtDef -> Specifier FunDec **CompSt**）、条件/循环作用域（Stmt -> IF LP Exp RP **CompSt** ELSE **CompSt** | WHILE LP Exp RP **CompSt**），匿名作用域（Stmt -> **CompSt**），每个作用域维护自己的变量符号表（CompSt -> LC **DefList** StmtList RC 中的 **DefList**）。

   开始语义分析时就创建一个全局作用域的FieldNode，并把当前作用域的指针 `FieldNode* currentField` 指向全局作用域，分析时，当遇到新的局部作用域（如函数的CompSt，if/while的ComptSt），则新建一个当前作用域的子作用域，将currentField置为子作用域并分析，在分析完子作用域后再通过FieNode中的parent指针将当前作用域指针currentField置回父作用域。

   每个作用域都有一个按ID排序的变量符号表（数组），查询时可对currentField的变量符号表进行二分查找，若找到了就用，找不到就寻找其父作用域currentField->parent的变量符号表，以此类推直到全局作用域，若都找不到才判定该变量未定义。

2. 更广泛的右值判定

   这里的右值不仅仅是数字（如2, 4.5等等），还包括了一切不占用储存位置的值，例如：

   - 算术运算所得到的临时值（假设已经定义了 `int a; float b; int func(int);`）

     `a * a = 5;` `(a && a) = 1;` `(b > a) = 1` 均报错6：赋值号左边出现一个只有右值的表达式

   - 函数调用的返回值（假设已经定义了 `int func(int);`）

     `func(5) = 8` 报错6：赋值号左边出现一个只有右值的表达式

3. Undefined —— 避免连环报错的类型推定

   在分析表达式Exp运算时，为了防止一处错误导致连环报错，引入未定义类型Undefined：

   - 赋值语句：右侧是Undefined，则视为遗留错误（不报错），并把表达式的类型推测为赋值号左侧的类型

- 加减乘除运算：

  两个Exp操作数都是Undefined，则视为遗留错误（不报错），运算结果为Undefined（右值）；

  有且只有一个Exp操作数是Undefined，不报错，并推测其类型为另一个已定义的操作数的类型，并作为运算结果（设为右值后）返回

- 与或运算/关系运算：

  两个Exp操作数都是Undefined，则视为遗留错误（不报错），运算结果为 int（右值）；

  有且只有一个Exp操作数是Undefined，不报错，并推测其类型为另一个已定义的操作数的类型，若是 int 则返回 int（右值）的运算结果，不是 int 则报错7： 操作数类型不匹配

- 函数调用：调用未定义的函数或参数不匹配，报相应的错误并返回Undefined类型
- 结构体域访问：访问不存在的域，报相应的错误并返回Undefined类型
- 数组访问：对非数组使用数组访问，报相应的错误并返回Undefined类型

4. 交付给底层的分析模式

   在语义分析过程中，会从根节点遍历一次语法树。在每一个节点把需要完成的任务交给子节点完成，并获取子节点返回的结果，例如：对于 `int a[5];`，语法树呈：Def -> Specifier DecList SEMI，先调用函数Type* defType = Type* handleSpecifier(Node*) 获取Specifer对应的Type类型，再将结果defType作为继承属性传递给DecList，即调用函数handleDecList(Node*, Type* inhType, ...)，在这个函数中（DecList -> Dec | Dec COMMA DecList）每遇到一个变量的定义Dec，就把它添加到当前作用域的符号表中；其中会调用的VarDec的具体分析代码如下（VarDec -> ID | VarDec LB INT RB）：

   ```
   Type* handleVarDec(Node* varDecNode, Type* inhType)  {
     if (/* 子节点是ID */) { // 最后获得一个ID，直接返回前面的继承属性
       return inhType;
     } else { // 子节点是数组
       Type* arrayType = createArrayType(getCertainChild(varDecNode, 3)->ival, inhType);
       return handleVarDec(getCertainChild(varDecNode, 1), arrayType);
     } }
   ```

5. 二分查找加快查询速度

   对于函数符号表、结构体符号表以及每个作用域下的变量符号表，它们都是按照ID从小到大排列的有序数组。这个数组的长度是预先计算好的：对于全局的函数符号表、结构体符号表，在语义分析开始前就计算函数头FuncDec的个数和结构体关键字struct的个数作为数组长度（由于函数有的是声明、struct有的只是定义变量，这个长度大于等于真正需要的长度，但没什么影响），并用malloc开辟符号表数组。

   设符号表长度为n，加入符号表的时候遍历直到找到合适的插入位置的时间复杂度为 O(n)，查询时使用二分查找（最坏情况下从当前作用域开始查到全局作用域）的时间复杂度为 kO(log(n))，k为作用域嵌套层数。

三、注意事项（一些私设）

- 对于**函数参数重名**的错误，本人把它归类为**错误类型3**（详见下面的细节说明）
- 对于**条件判断**的类型错误（如：`if(1.5)...`、`while(0.5)...`），本人归类为**错误类型7**（详见细节说明）
- 报错出现的**顺序可能不与错误出现的行号顺序相同**（例如错误类型18永远是最后才判断的）
- 对于**同名结构体的嵌套是不允许**的（如：`struct A {int x; struct A next;};`），因为本程序中只有一个结构体完全定义结束后才会加入结构体符号表，对于同名嵌套将报错17
- 结构体内部**域名是允许与任何结构体名重名**的（如：`struct A {...}; struct B {int A;};` 不报错），因为实验要求中没有说需要报错（只有变量与前面定义过的结构体名字重复时才报错）
- 对于算术运算的类型匹配，认为：
  - 加减乘除/取负：仅int/float可参与，且参与运算的**操作数类型相同**（同为int或同为float）
  - 逻辑与或非/取反：参与运算的操作数必须**同为int**
  - 关系（==/</>/...）：仅int/float可参与，参与运算的**操作数类型可以不同**（如：`1 < 1.5` 不报错）
- 根据本程序**右值的判定**的方法，像 `x*y, func(4), -t` 等只要没有储存地址，出现在赋值语句左侧均报错6
- 由于Undefined的类型推测，表达式的一些错误不会连续性引发
- 为了避免输出奇怪的问题，当词法/语法分析不通过时（Lab1的错），不会进行Lab2的语义分析