# 编译原理实验报告

## 实验二：语义分析

**191220008 陈南疃**

924690736@qq.com

## 一、运行方式

```
$ make
$ ../parser testfile
$ make clean
```

## 二、实现功能

任务编号：11

选做部分：2.2

### （一）符号表

#### 1、基本数据结构

符号表的基本结构采用了开散列、闭地址的进行哈希的方式，因此添加、查找、删除的时间效率都比较高。整体上用一个结构数组 `symbolTable[SYMBOL_TABLE_SIZE]` 进行储存，相同哈希值的符号以链表的方式存储在同一个槽下。

#### 2、嵌套作用域

为了实现嵌套作用域，我在上述结构的基础上将单向链表修改为十字链表，横向为相同哈希值的符号，纵向为同一作用域的符号，并用一个栈（用链表实现，因此不用担心层数太大导致栈溢出）存储每个作用域的链表头，每当进入（退出）一个作用域时，只需对栈顶的进行操作，并添加（删除）对应作用域的符号即可，操作简便且效率高。

```
struct SymbolNode_ { // 符号结点
    ...
    SymbolNode hashNext; // 相同哈希值的下一个符号
    SymbolNode scopeNext; // 当前作用域中的下一个符号
};

struct ScopeStackHead_ { // 作用域链表头
    SymbolNode scopeNext; // 当前作用域的第一个符号
    ScopeStackHead scopeStackNext; // 上一层作用域的链表头
};
```

狭义上整个程序表面上只有一张符号表，但每个作用域对应一个链表头，所以实际上每个作用域广义上均有一张符号表，用全局变量 `scopeDepth` 表示当前作用域的层数，进入（退出）作用域时调用 enterScope 函数或 exitScope 函数：

```
void enterScope() { scopeDepth += 1; ··· } // 进入作用域
void exitScope() { scopeDepth -= 1; ··· } // 退出作用域
```

## （二）语义分析

### 1、语义分析方法

由于在实验一的语法分析过程中构建了一棵语法分析树，因此我直接基于对语法树的遍历来进行语义分析，所以我对每种非终结符都编写了一个同名的函数，该函数通过参数的传递来实现继承属性，通过返回值实现综合属性，然后根据非终结符的子结点进行语义分析和符号表的增删查，并报出对应类型的语义错误。

### 2、匿名结构体之间的等价判定

本实验的语法是支持匿名结构体的定义的，但由于没有结构体名，所以给匿名结构体变量之间的等价判定造成了困难。因此，我对每个结构体进行了额外的命名。首先用全局变量 `anonymousStructNum` 记录当前定义过的匿名结构体数量，在定义匿名结构体时，将 `anonymousStructNum` 转为字符串并于字符串 `“struct”` 拼接后作为该结构体的名字存储到符号表中（如：`”14struct“`）。由于合法的变量名不会以数字开头，所以这些匿名结构体的名字不会与任何变量重复。

```
sprintf(anonymouStructName, "%dstruct", anonymousStructNum);
```

### 3、如何将函数形参添加到符号表

在C语言中，函数形参的作用域是与函数体位于同一层的，需要用一张符号表来存储，但由于在我们C--的文法定义中，两者是分开的，因此我为两者各自建立了一张符号表，在形参定义结束后，以 FieldList 类型传给函数体，将形参导入函数体的符号表中去即可。

```
FieldList ParamDec(Node* node, Context context); // 形参以FieldList类型返回
void CompSt(Node* node, Type type, FieldList param); // 通过param传入函数体CompSt
```

### 4、Exp 的内容获取

在部分错误信息的输出中需要用到某个 Exp 结点的内容，如：

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

这里的 `i` 便是某个 Exp 结点中的内容，但有的 Exp 结点可能存在多个子结点，因此需要将每个子结点的内容拼接起来作为当前 Exp 的内容。我编写了一个 getExpContent 函数，在里面进行递归调用，利用许畅老师上课时讲到的递归下降实现L属性SDD的方法，边扫描边生成（将 Exp 内容作为返回值），到达递归的最外层时就得到了整个 Exp 的内容，如：

```
Error type 10 at Line 4: "i[2][3]" is not an array.
```

但由于有少数情况会导致 segmentation fault（与字符串拼接的方法不当有关），所以暂时将返回值设为” xxx“。

### 5、类型表示

手册中给出了 Type 和 FieldList 来表示类型，但在本实验中不太够用，我在 Type 的 kind 枚举类型中额外添加了 `FUNCTION`，`STRUCTTAG`，`STRUCTFIELD`，分别是函数、结构体名字、结构体域，函数需要存储参数和返回值类型，结构体名字借用结构体变量的存储方式，结构体域与变量存储方式相同。每种类型的具体表示方法较复杂且因人而异，我也是在草稿纸上手画了很多次才得到一个满意的表示方法，达到了尽可能少的信息冗余和尽可能高的查找效率。

此外，需要严格区分 `STRUCTURE` 和 `STRUCTTAG`，分别是结构体变量和结构体名字，实验前期由于没有将 `STRUCTTAG` 单独表示，走了不少弯路。

### 6、VarDec 的分类讨论

非终结符 VarDec 可能在多种语句块内出现，不同的语句块内需做不同的处理。我将不同的语句块分为四类，并用枚举类型 Context 表示：

```
typedef enum { CONTEXT_EXT, CONTEXT_LOCAL, CONTEXT_STRUCT, CONTEXT_PARAM }
Context;
```

在不同非终结符的处理函数以参数的形式进行传递，最终传到 VarDec 函数中：

```
FieldList VarDec(Node* node, Type type, Context context) { ··· }
```

### 7、发生语义错误时 Type 的处理

当发生语义错误时，原本的 Type 类型的返回值可能需要手动设置（比如符号表查询失败时需要置为 NULL，Exp AND Exp 类型错误时需要置为 BASIC_INT 类型……）。这可以尽可能的减少因当前语义错误而导致后续代码错误的误报。

## 三、心得和建议

实验二总体来说很繁杂，需要处理的细节非常多，因此良好、清晰的数据结构和函数接口能够为程序的编写提供有效的帮助，为后期的 debug 提供便利，所以我在实验前期先用了不合理的数据结构编写后，发现代码很容易出问题，且难以与后期的代码兼容，所以我花了许多时间在草稿纸上画出了程序的关键数据结构和数据流向，确认无误才进行了代码实现，所以实验后期相对愉快，能够利用 gdb 很快发现并修复 bug。

实验中使用了许多很少用到的库函数，虽然在使用过程中碰到了不少问题，但都加深了对库函数的理解。

手册上各种背景知识和实验指导交杂，导致我实验最初一头雾水，不知道语义分析该怎么用代码实现，向其他同学咨询后才有所思路，希望手册上能够多提示一下实现方法（类似实验一）。