# 实验二 语义分析 实验报告

匡亚明学院 陈劭源 161240004

## 文件夹结构

注意：Makefile在根目录下（而不是在Code文件夹内）

```
.
├── Code                    // 源代码文件
│   ├── ast                 // 抽象语法树相关代码
│   │   └── ...
│   ├── error.c             // 错误处理代码
│   ├── lexical.l           // flex词法文件
│   ├── memory.c            // 内存管理代码
│   ├── symtbl.c            // 符号表代码
│   ├── main.c              // 主程序
│   └── syntax.y            // bison语法文件
├── Include                 // 头文件
│   ├── ast                 // 抽象语法树相关
│   │   └── ...
│   ├── cmm.h
│   ├── memory.h
│   ├── symtbl.h
│   ├── container           // 包含链表等数据结构
│   │   └── ...
│   ├── cst.h
│   └── error.h
├── Makefile                // Makefile文件
├── parser                  // 语法分析器可执行文件
├── README.txt              // README文件
├── report.md               // 本实验报告的源代码
├── report.pdf              // 本实验报告
├── Test
│   ├── sample              // 提供的测试用例
│   │   └── ...
│   └── secret              // 自行构造的测试用例
│       └── ...
└── testrun.sh              // 测试用例运行脚本
```

## 编译和运行方法

### 编译环境

- OS: Ubuntu 18.04.2 LTS
- gcc: gcc 7.3.0
- flex: flex 2.6.4

- bison: GNU Bison 3.0.4
- make: GNU Make 4.1
- shell: GNU bash 4.4.19

## 编译方法

切换到根目录，输入

```
make
```

即可从源代码生成可执行文件parser（位于根目录）。

## 运行方法

输入

```
./parser
```

或

```
make run
```

即可运行语法分析器。语法分析器默认从标准输入读入c--源代码，可以通过参数指定从文件读入：

```
./parser source_file
```

运行

```
make clean
```

可以清除所有中间文件和目标文件。

为便于测试，根目录包含了一个测试脚本，可按如下方式运行：

```
./testrun.sh Test/sample/semantics/*
```

脚本会将源代码和分析器的输出用`less`打印在屏幕上。

# 完成的功能点

1. 检查17种语义错误
2. （选做）实现函数声明功能
3. （选做）变量定义受嵌套作用域影响，外层语句块中的变量可以在内层重复定义
4. （选做）实现结构等价，即两个结构体成员数目相等，且对应成员类型等价，则整个结构体类型等价

# 实现方法

本次实验主要实现方法是，在语法分析的过程中，直接构造抽象语法树（不再构造具体语法树），并且在构造抽象语法树的过程中检查各种语义错误。

抽象语法树中的节点类型主要有：

- 语句节点（statement）；
- 表达式节点（expression）；
- 函数节点（function）；
- 类型节点（type）；
- 变量节点（variable）。

其中，根据语句和表达式的具体类型，语句节点和表达式节点还可细分为多种子类型。每种节点都有对应的构造函数，在构造该节点的同时检查是否有相关的语义错误。为了避免多趟扫描，各节点的属性都尽量设计成了综合属性。

为了实现符号的保存和查找，程序还需维护符号表。这里符号表采用链表实现。变量作用域则是通过栈的方式实现的：

- 每当进入一个新的作用域时，就创建一张新的符号表并将其压入栈中；
- 查找符号时，从栈顶向下依次查找；
- 添加符号时，添加到栈顶的符号表中；
- 离开作用域时，弹出栈顶的符号表。

为了方便内存管理，本次实验还实现了简单的内存分配器（在memory.c）中，实现内存的分配和集中释放。

# 实验总结

本次实验在上一次实验的基础上，添加了语义分析功能，生成的分析器能够根据输入的c--代码构造抽象语法树，并能识别并报告c--源代码中的语义错误。