

# 编译原理Lab1实验报告

191870271 赵超懿

## 实现功能

使用Flex实现了词法分析，注释解析

整个Flex文件包括了一些头文件，变量定义，正则表达式及其对应的动作，一些辅助函数。

使用Bison实现了语义分析，多种错误恢复

Bison文件包括了token等的定义以及解决移入规约问题的描述，第二部分是附录中的产生式

## 如何编译

使用makefile,在Code目录下，使用make parse，可以在当前目录下获得一个parse文件，直接使用./parse test来使用，test是测试文件

## 程序特点

### Flex部分

在写正则表达式时，一个问题是int和float两种类型的数有时候前一部分是相同的，例如21.3343中的21，即21可以被识别为整数，为了解决这个问题，我在int的表达式后面增加了匹配一个非.(小数点)即[^\.]的匹配项，这样可以区分int和float的前半部分。当识别为int类型时，使用讲义中提到的unput()函数将多匹配的字符放回，保证不会误匹配

### Bison部分

错误恢复，在这一部分我主要通过syntax.output文件中内容来寻找问题，并增加了很多error且不产生规约冲突

### 语法树

采用了函数指针实现OOP写法，将所用的定义及函数用一个.c文件和一个.h文件写完，并且将函数多层封装，使得在Bison中非error和非空项均可以使用一个函数完成语义动作

```
$$ = Operator($$, string, @$$.first_line, 1, $1);  
//string是一个表示$$的字符串, 1指后面的参数的个数, Operator参数的个数是可变的
```

这样的写法使得Bison十分整齐简洁, 如下

```
Exp : Exp ASSIGNOP Exp          { $$ =  
  Operator($$, "Exp", @$$.first_line, 3, $1, $2, $3); }  
    | Exp AND Exp              { $$ =  
  Operator($$, "Exp", @$$.first_line, 3, $1, $2, $3); }  
    | Exp OR Exp               { $$ =
```

```
Operator($$, "Exp", @$.first_line, 3, $1, $2, $3); }
```

...均类似于这样的写法

## 整个Lab的调试

通过简单修改Makefile和各种宏来完成一键编译测试。

调试的输出全部使用debug.h中定义的各种函数来进行区别，并通过各种宏控制是否开启