编译原理Lab1实验报告

任务号：11
组长：董启轩 181860016
组员：田浩 181860088
联系方式：1273440689@qq.com

一、编译及测试环境

虚拟机：Ubuntu16.04.5

版本： gcc 5.4.0
flex 2.6.0
bison 3.0.4

编译方式：使用提供的makefile编译

分工： 田浩：语法树及基本语法
董启轩：词法分析器以及错误恢复部分

二、实验部分

(一)词法分析器

必做部分：

根据Appendix_A中所描述的Token以及实验手册的指导，完成基本的誊抄，返回值根据语法部分做修改。

其中int型数为0或者由0-9组成的不以0开头的整数。float型数整数部分照抄int，小数部分为任意0-9数字的组合。ID根据要求实现即可。此处由于科学计数法和八进制十六进制数在选做1.1和1.2中，不在本组范畴所以就没有考虑。在INT、FlOAT和ID传值时分别使用atoi( )，atof( )和char*完成yytext向语法分析器的传递。

此外，为了便于后续的语法树书写，将\n单独拎出来分析，同时将RELOP的六个操作符分开按照特定的顺序排列分开向语法分析器传递在一开始硬编码的六个值，但返回值仍为RELOP。TYPE也使用char*传递具体类型。

选做部分：

支持单行和多行注释，根据Appendix_A，单行注释将由词法分析器抛掉，同理多行注释也可以由词法分析器抛掉。单行注释部分照抄手册的代码；多行注释模拟单行注释实现即可，匹配最先遇到的*/，当未查找到*/时将在/*所在处报语法错误。

(二)语法分析器

1、基本语法和语法树

必做部分：

在bison中写下C--语法的所有产生式，按照运算优先级与运算的结合性设置结合性。特殊地，负号应当为左结合，同时应该声明一个不存在的符号SUB的结合性，然后在产生式EXP->EXP MINUS EXP的后面加上%prec SUB，表明这条产生式的优先级等同于SUB的优先级，用以处理MINUS同时作为负号和减号带来的问题，从语法上区分MINUS是负号还是减号。

有关语法分析树的构建，语法分析树的相关定义如下：

```
union TreeVal{
    int i;
    float f;
    char *str;
};
struct GrammarTree;
struct ListNode{
    struct GrammarTree *val;
    struct ListNode *next;
};
struct GrammarTree{
    int line;
    union TreeVal val;
    int type;
    struct ListNode *head;
};
```

语法分析树的结点类型为GrammarTree，其中的成员line为结点所在行号，type表示其类型，val类型为int，float，char*的联合体，存放结点的数据，根据type的不同，存放的数据类型不同，head为子女结点的链表头。

type既包含了词法单元的类型也包含了语法单元的类型，词法单元的类型已经在yytokentype当中定义，按照在syntax.y中出现顺序从258开始排列。用enum定义语法单元的类型如下：

```
enum token{
    Program = 0, ExtDecList, ExtDef, Specifier, FunDec, CompSt,
    VarDec, ExtDefList, StructSpecifier,
    OptTag, DefList, Tag, VarList, ParamDec,
    StmtList, Stmt, Exp, Def, DecList, Dec, Args,
    error
};
```

事先需要开启行号的维护。

为bison中的产生式添加动作，创建产生式的头对应的语法分析树结点，创建产生式的体中的词法单元对应的树结点，因为词法单元对应为语法分析树的叶子结点。如果词法单元属于ID，TYPE，INT,FLOAT，RELOP，还应该将词法分析器传来的值存入语法分析树的结点。然后往产生式的头对应的树结点的子女链表中插入产生式的体的所有单元对应的结点。为了实现值从词法分析器到语法分析器的传输，词法单元和语法单元的类型type_treenode实际上是一个包含了int，float，char *，struct GrammarTree*的联合类型，词法分析器将值写入联合体中，语法分析器按照需要将其取出。特殊地，为了判断RELOP的种类，在词法分析器和语法分析器中都define了不同RELOP对应的值，然后用int类型进行传递。为了代码的可维护性与可读性，这些产生式动作对应的代码主要用两个封装好的函数实现，定义分别为struct GrammarTree *createnode(int type,int line,void *value)与void insertall(struct GrammarTree *parent,int num,...)，前者传入创建树结点需要的所有信息，返回创建的树结点指针，后者使用可变参数，传入父节点子女个数与需要插入父节点的所有树结点指针，进行插入操作。最后将最上层的产生式的树结点的指针传给全局变量root，就完成了语法分析树的生成。

有关语法分析树的打印，使用DFS实现。DFS过程中记录层数，用来控制缩进。用switch语句判断树结点的类型，然后决定打印的信息。通过树结点里面存储的类型值得到要打印的类型名的字符串用一个定义好的token_map数组实现。具体如下：

```
char *token_map[] = {"Program","ExtDecList","ExtDef","Specifier","FunDec","CompSt",
                     "VarDec","ExtDefList","StructSpecifier",
                     "OptTag","DefList","Tag","VarList","ParamDec",
                     "StmtList","Stmt","Exp","Def","DecList","Dec","Args","error"};
char *token2_map[] = {"SEMI","COMMA","ASSIGNOP","PLUS",
                      "MINUS","STAR","DIV","AND","OR","NOT","DOT","TYPE","LP","RP","LB","RB" ,"LC","RC","STRUCT","RETURN",
                      "IF","ELSE","WHILE","INT","FLOAT","ID","RELOP","SUB","LOWER_THAN_ELSE"};
```

前者根据语法单元的类型得到对应字符串，后者根据词法单元类型得到对应字符串。因为词法单元类型为yytokentype，yytokentype从258开始，所以再define一个BASE_NUM，为常数258，使用token2_map的时候应当用类型值减去BASE_NUM从数组中索引得到对应的字符串。

选做部分：

注释已交给词法过滤，此处无需做任何工作。

2、错误恢复

必做部分：

此处添加了新的token：error，用于在错误的地方创建error节点，便于输出语法树查看error覆盖了哪些部分。同时添加新的结构ErrorNode用于构建按错误行号非递减的语法错误链表。在有错误时将按顺序报错。

由于不可能解决所有的错误因此针对给出的样例写出了如下错误产生式，注释内容为其可以解决的错误：

```
Exp -> Exp LB error RB                  //数组元素访问时，括号中格式错误
Stmt -> IF LP Exp RP error ELSE Stmt    //if-else的条件错误，但针对给出样例将由下式报错
Stmt -> Exp error                       //一条代码的末尾缺少“;”
```

此外经过许久的摸索添加了如下尽可能少的发生冲突、覆盖重复错误少的几个错误产生式，以下是举例：

```
Stmt -> RETURN Exp error                //return缺少“;”
VarDec -> VarDec error RB               //数组变量声明时括号中格式使用错误或缺少“[”
```

经过多个样例的测试，上述错误产生式基本能够独立分别报错。

虽然在“尽少的冲突，尽可能独立”的前提下写出了足够多的错误产生式，但很明显的，这些错误产生式并不能按描述一样精确的认出所有该类型的错误。其仍会因为某些错误忽略掉过多的词法单元导致接下来的错误无法报出。同时也仍有可能一错多报。

所以这一部分只能完成至此(受制于错误产生式的局限性，不太可能尽善尽美)。

选做部分：

针对注释，只需要考虑多行注释的嵌套错误和符号不匹配问题，根据词法分析器的结构，一般只会出现独立的*/，但这是更加棘手的问题，因此多行注释可以出现在程序代码中的任何地方。那么遇到*/时，语法分析器可能处于很多可能的状态。而这些，部分会由之前填写的error报错，部分将不能被识别。因此需要写一个覆盖面尽可能广的error产生式，如下：

```
ExtDef -> Specifier error
```

这样基本可以实现注释的报错，但一个极大的缺点是一旦由ExtDef报错，很多错误将直接被忽略。因此也是一个不太能尽善尽美的报错。