实验二 语法分析器

一、实验目的

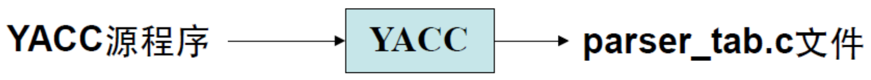
1. 理解编译器的工作机制，掌握编译器的构造方法
2. 掌握语法分析器的生成工具 bison 的用法

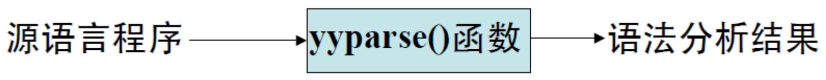
二、实验内容

1. YACC 简介

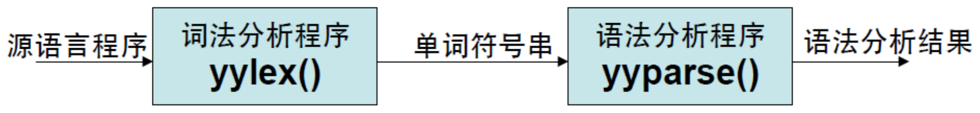
YACC = Yet Another Compiler Compiler

YACC是一个语法分析程序的自动产生系统

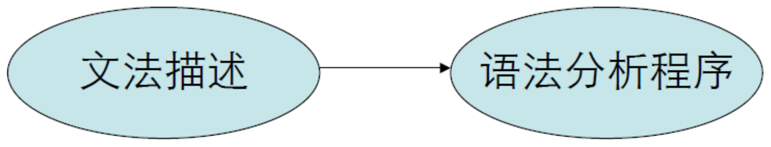




词法分析程序与语法分析程序的关系



YACC的工作原理：



YACC的处理能力：可以用LALR(1)文法表示的上下文无关文法。

2. PL/0 语言的 EBNF 范式

详见词法分析实验指导书

3. 实验内容

1）用 bison 工具生成一个 PL/0 语言的语法分析程序，对 PL/0 源程序进行语法分析。

- 输入：pl/0 源程序
- 输出：
  - 按归约顺序用到的语法规则
  - 语法单位的层次结构关系

5）实验环境

- Windows & C
- 语法分析器生成工具：bison

三、实验步骤

1. 思路说明：

对于第一个输出，按规约顺序输出用到的语法规则，思路比较简明，yyparse 函数执行规约的时候会按照程序本身的顺序进行规约，即直接在对应产生式规约后执行的动作中加上将该产生式输出到输出文件的操作即可。对于第二个输出，建立一个栈，在词法每次进行匹配的时候，将匹配到的终结符入栈。同时建立一个树，左节点表示孩子节点，右节点表示兄弟节点。在语法文件进行规约的时候，在规约动作中，将产生式右边的对应的终结符或者非终结符出栈，并建立新节点作为产生式左部的非终结符，将产生式右部的第一个符号作为产生式左部的孩子节点，并以此次将右部其他节点作为其左边节点的兄弟节点，最后将产生式左部的节点压入栈中。程序执行完成规约后，对建立的树进行先序遍历，对应层次越深缩进越多，以此来表示层次结构。

2. 实验步骤：

a.将 flex.exe 文件、bison 相关程序文件、写好的 lex 程序、写好的语法的.y 程序和待输入的 PL/0 程序源代码放到同一个文件夹

b.在当前目录下打开 cmd

c.在 cmd 下输入命令 flex lex.l 将 lex 程序用 flex 工具生成 lex.yy.c 文件，输入命令 bison Syntax.y -d 将.y 程序用 bison 工具生成 Syntax.tab.c 和 Syntax.tab.h 文件（提前将 Syntax.tab.h 文件在 lex.l 声明部分中引用）

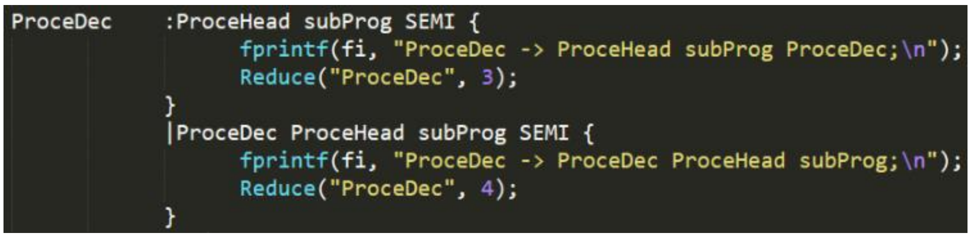
d.在 cmd 下用 gcc 命令将 Syntax.tab.c 和 lex.yy.c 文件联合编译：

```
gcc  －o Syntax.tab.exe Syntax.tab.c lex.yy.c
```

e.输入命令 Syntax.tab.exe ＜ test-syn.pl0 对 test-syn.pl0 文件进行对应输出

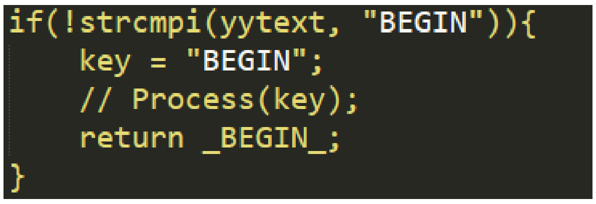
3.对代码的说明

下面对部分相关代码进行说明：



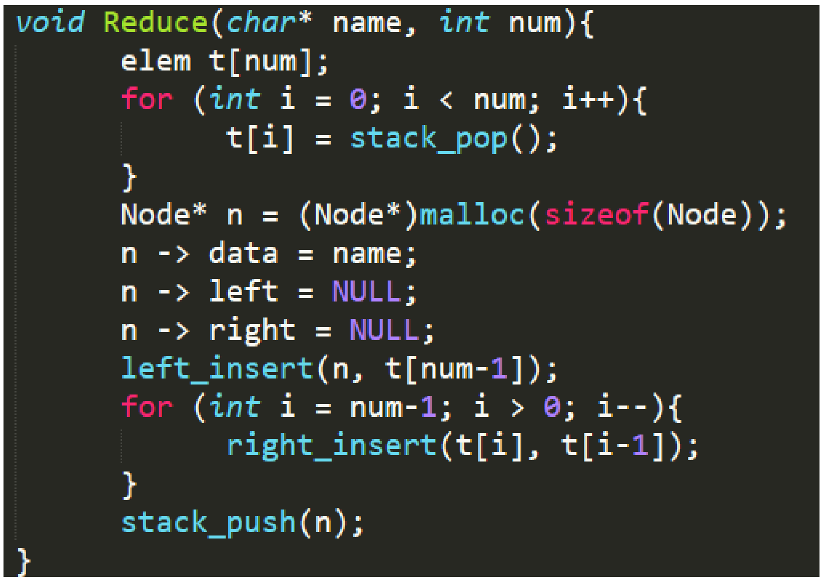
```
ProceDec    :ProceHead subProg SEMI {
                fprintf(fi, "ProceDec -> ProceHead subProg ProceDec;\n");
                Reduce("ProceDec", 3);
            }
            |ProceDec ProceHead subProg SEMI {
                fprintf(fi, "ProceDec -> ProceDec ProceHead subProg;\n");
                Reduce("ProceDec", 4);
            }
```

- 上图是过程并列的部分产生式定义，采用左递归方式，对应规约动作为输出该产生式，并调用 Reduce 函数进行出栈入栈和连接节点操作



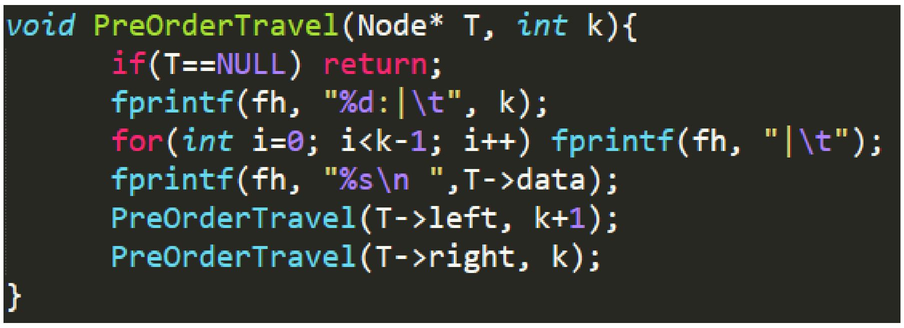
```
if(!strcmpi(yytext, "BEGIN")){
    key = "BEGIN";
    // Process(key);
    return _BEGIN_;
}
```

- 上图为词法.l 文件中的代码，匹配到 BEGIN 终结符后，将该终结符压入栈中（由于 bison 会提前多看一个符号，所以将具体的 Process 函数即入栈操作放到了 Syntax.tab.c 中移进操作之前），并将其 return 到语法分析程序中进行处理



```
void Reduce(char* name, int num){
        elem t[num];
        for (int i = 0; i < num; i++){
                t[i] = stack_pop();
        }
        Node* n = (Node*)malloc(sizeof(Node));
        n -> data = name;
        n -> left = NULL;
        n -> right = NULL;
        left_insert(n, t[num-1]);
        for (int i = num-1; i > 0; i--){
                right_insert(t[i], t[i-1]);
        }
        stack_push(n);
}
```

- 上图是语法规约动作中对应的规约操作，即产生式左的节点入栈，右边的节点进行连接

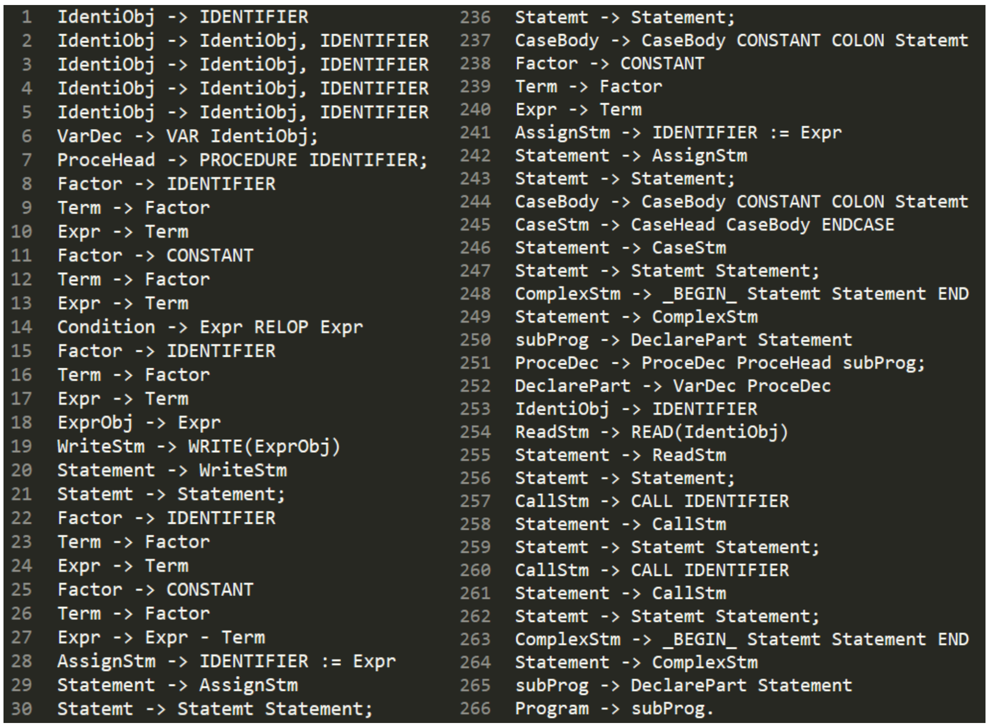


```
void PreOrderTravel(Node* T, int k){
        if(T==NULL) return;
        fprintf(fh, "%d:|\t", k);
        for(int i=0; i<k-1; i++) fprintf(fh, "|\t");
        fprintf(fh, "%s\n ",T->data);
        PreOrderTravel(T->left, k+1);
        PreOrderTravel(T->right, k);
}
```

- 上图是进行先序遍历的函数,k 表示递归的层数,T 是传进来的节点，最终规约完后只剩 Program 一个节点，即树的根节点，从其开始先序遍历即可得到对应的层次结构

四、实验结果

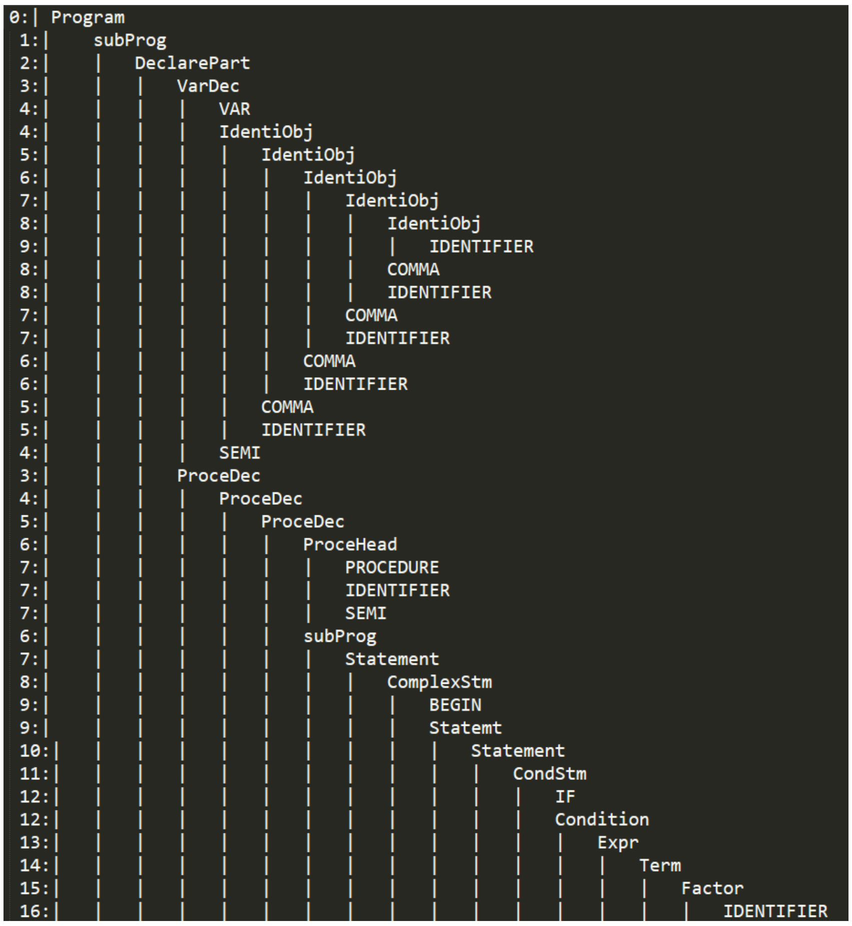
按规约顺序输出的语法规则如下：



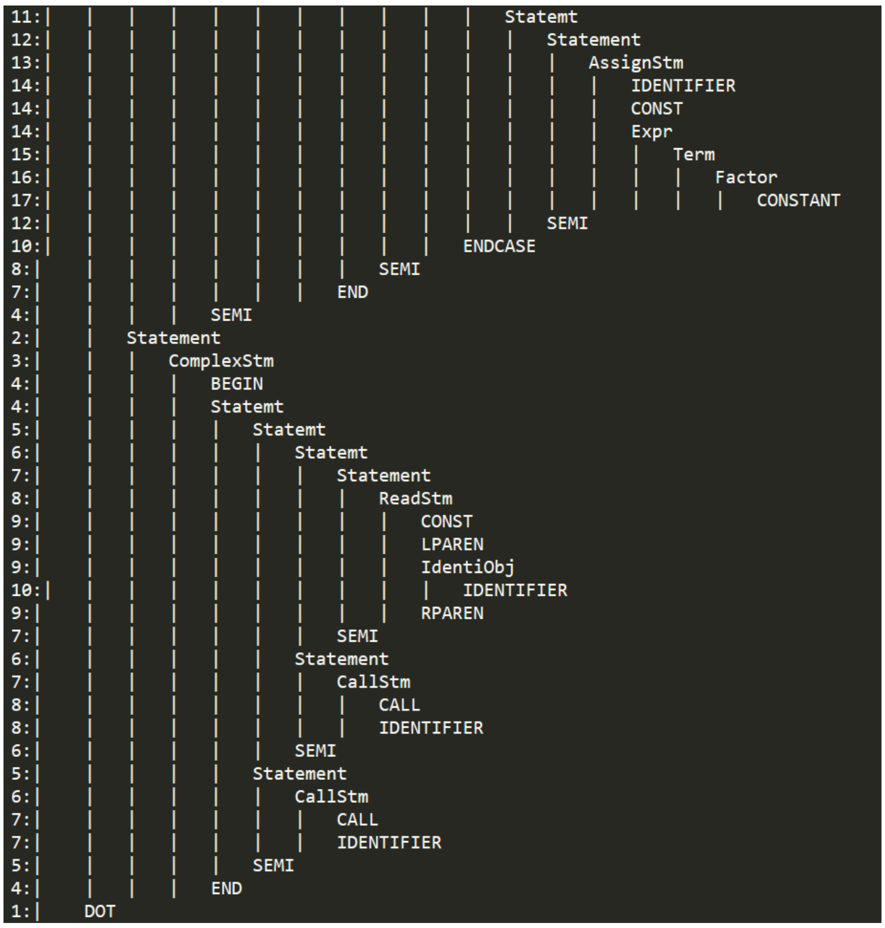
```
IdentiObj -> IDENTIFIER
IdentiObj -> IdentiObj, IDENTIFIER
IdentiObj -> IdentiObj, IDENTIFIER
IdentiObj -> IdentiObj, IDENTIFIER
IdentiObj -> IdentiObj, IDENTIFIER
VarDec -> VAR IdentiObj;
ProceHead -> PROCEDURE IDENTIFIER;
Factor -> IDENTIFIER
Term -> Factor
Expr -> Term
Factor -> CONSTANT
Term -> Factor
Expr -> Term
Condition -> Expr RELOP Expr
Factor -> IDENTIFIER
Term -> Factor
Expr -> Term
ExprObj -> Expr
WriteStm -> WRITE(ExprObj)
Statement -> WriteStm
Statemt -> Statement;
Factor -> IDENTIFIER
Term -> Factor
Expr -> Term
Factor -> CONSTANT
Term -> Factor
Expr -> Expr - Term
AssignStm -> IDENTIFIER := Expr
Statement -> AssignStm
Statemt -> Statemt Statement;
```

```
Statemt -> Statement;
CaseBody -> CaseBody CONSTANT COLON Statemt
Factor -> CONSTANT
Term -> Factor
Expr -> Term
AssignStm -> IDENTIFIER := Expr
Statement -> AssignStm
Statemt -> Statement;
CaseBody -> CaseBody CONSTANT COLON Statemt
CaseStm -> CaseHead CaseBody ENDCASE
Statement -> CaseStm
Statemt -> Statemt Statement;
ComplexStm -> _BEGIN_ Statemt Statement END
Statement -> ComplexStm
subProg -> DeclarePart Statement
ProceDec -> ProceDec ProceHead subProg;
DeclarePart -> VarDec ProceDec
IdentiObj -> IDENTIFIER
ReadStm -> READ(IdentiObj)
Statement -> ReadStm
Statemt -> Statement;
CallStm -> CALL IDENTIFIER
Statement -> CallStm
Statemt -> Statemt Statement;
CallStm -> CALL IDENTIFIER
Statement -> CallStm
Statemt -> Statemt Statement;
ComplexStm -> _BEGIN_ Statemt Statement END
Statement -> ComplexStm
subProg -> DeclarePart Statement
Program -> subProg.
```

语法单位的层次结构关系如下，行首的数字表示节点（递归）的层数：



```
 0:| Program
 1:|    subProg
 2:|    |   DeclarePart
 3:|    |   |   VarDec
 4:|    |   |   |   VAR
 4:|    |   |   |   IdentiObj
 5:|    |   |   |   |   IdentiObj
 6:|    |   |   |   |   |   IdentiObj
 7:|    |   |   |   |   |   |   IdentiObj
 8:|    |   |   |   |   |   |   |   IdentiObj
 9:|    |   |   |   |   |   |   |   |   IDENTIFIER
 8:|    |   |   |   |   |   |   |   COMMA
 8:|    |   |   |   |   |   |   |   IDENTIFIER
 7:|    |   |   |   |   |   |   COMMA
 7:|    |   |   |   |   |   |   IDENTIFIER
 6:|    |   |   |   |   |   COMMA
 6:|    |   |   |   |   |   IDENTIFIER
 5:|    |   |   |   |   COMMA
 5:|    |   |   |   |   IDENTIFIER
 4:|    |   |   |   SEMI
 3:|    |   |   ProceDec
 4:|    |   |   |   ProceDec
 5:|    |   |   |   |   ProceDec
 6:|    |   |   |   |   |   ProceHead
 7:|    |   |   |   |   |   |   PROCEDURE
 7:|    |   |   |   |   |   |   IDENTIFIER
 7:|    |   |   |   |   |   |   SEMI
 6:|    |   |   |   |   |   subProg
 7:|    |   |   |   |   |   |   Statement
 8:|    |   |   |   |   |   |   |   ComplexStm
 9:|    |   |   |   |   |   |   |   |   BEGIN
 9:|    |   |   |   |   |   |   |   |   Statemt
10:|    |   |   |   |   |   |   |   |   |   Statement
11:|    |   |   |   |   |   |   |   |   |   |   CondStm
12:|    |   |   |   |   |   |   |   |   |   |   |   IF
12:|    |   |   |   |   |   |   |   |   |   |   |   Condition
13:|    |   |   |   |   |   |   |   |   |   |   |   |   Expr
14:|    |   |   |   |   |   |   |   |   |   |   |   |   |   Term
15:|    |   |   |   |   |   |   |   |   |   |   |   |   |   |   Factor
16:|    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   IDENTIFIER
```



```
11:|   |   |   |   |   |   |   |   |   |   |   Statemt
12:|   |   |   |   |   |   |   |   |   |   |   |   Statement
13:|   |   |   |   |   |   |   |   |   |   |   |   |   AssignStm
14:|   |   |   |   |   |   |   |   |   |   |   |   |   |   IDENTIFIER
14:|   |   |   |   |   |   |   |   |   |   |   |   |   |   CONST
14:|   |   |   |   |   |   |   |   |   |   |   |   |   |   Expr
15:|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   Term
16:|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   Factor
17:|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   CONSTANT
12:|   |   |   |   |   |   |   |   |   |   |   |   SEMI
10:|   |   |   |   |   |   |   |   |   |   ENDCASE
8:|   |   |   |   |   |   |   |   SEMI
7:|   |   |   |   |   |   |   END
4:|   |   |   |   SEMI
2:|   |   Statement
3:|   |   |   ComplexStm
4:|   |   |   |   BEGIN
4:|   |   |   |   Statemt
5:|   |   |   |   |   Statemt
6:|   |   |   |   |   |   Statemt
7:|   |   |   |   |   |   |   Statement
8:|   |   |   |   |   |   |   |   ReadStm
9:|   |   |   |   |   |   |   |   |   CONST
9:|   |   |   |   |   |   |   |   |   LPAREN
9:|   |   |   |   |   |   |   |   |   IdentiObj
10:|   |   |   |   |   |   |   |   |   |   IDENTIFIER
9:|   |   |   |   |   |   |   |   |   RPAREN
7:|   |   |   |   |   |   |   SEMI
6:|   |   |   |   |   |   Statement
7:|   |   |   |   |   |   |   CallStm
8:|   |   |   |   |   |   |   |   CALL
8:|   |   |   |   |   |   |   |   IDENTIFIER
6:|   |   |   |   |   |   SEMI
5:|   |   |   |   |   Statement
6:|   |   |   |   |   |   CallStm
7:|   |   |   |   |   |   |   CALL
7:|   |   |   |   |   |   |   IDENTIFIER
5:|   |   |   |   |   SEMI
4:|   |   |   |   END
1:|   DOT
```

五、实验体会

本次实验是关于编译器制作中的第二步，即语法分析器。通过本次实验，进一步理解了编译器的工作机制，尤其是语法处理过程中的规约规则和其执行顺序，掌握了语法分析器的生成工具 bison 的用法，同时对产生式和规约的层次结构有了更深的理解，有待在后续实验中进一步熟悉和掌握。