

## 1. 应用背景

C 语言子集的编译器，可将该语言的源程序翻译成其他形式的中间语言的目标程序，并且能够执行相应的语义动作（如计算复杂的表达式的值，if-else 语句的跳转，条件的判断，变量的定义与赋值，循环语句）得到结果，实现了 C 语言的基本功能。

## 2. 语法描述

### 2.1 主要分为词法分析，语法分析，语义分析三个部分。

词法分析包括：代码块 block，if-else 语句，while 语句，switch-case 语句，do-while 语句，变量定义（包括 char, real, int, string），赋值语句，表达式，定义结构体，字面值的识别。（在 Adder. jj 文件中，我还定义了结构体，函数，指针等语法规则，将在后期实现）

语法分析包括：代码块 block，if-else 语句，while 语句，变量定义与赋值语句，赋值语句，表达式，条件语句。

语义分析包括：代码块 block，if-else 语句，while 语句，变量定义与赋值语句，赋值语句，表达式，条件语句。

### 2.2 详细语法规则

2.2.1 代码块 block：以 “{” 为开始，“}” 为结束，中间可嵌套代码块或是 C 语言语句。

规则如下：

```
"{"(Statement() | Block())+"}"
```

2.2.2 变量定义语句：以 int, char, string 等为识别开始符号，可逗号分割定义多个变量，并为变量赋值。赋值语句与之规则类似。

规则如下：

```
((<INT>
| <REAL>
| <CHAR>)[ "["Integer()" ]"
)Identifier()(<EQUAL> Expression())*(<COMMA>Identifier() (<EQUAL>
Expression())*)*<SEMICOLON>
//赋值语句
Identifier()<EQUAL>Expression()<SEMICOLON>
```

2.2.3 if-else 语句：if 为开始符号，然后识别左括号，条件表达式，右括号，然后识别具体的执行语句，可以有 else 语句或省略 else。

规则如下：

```
(<IF><LEFTPARENTHESSES>Expr()<RIGHTPARENTHESSES>Statement()[<ELSE>Statement()]+
```

2.2.4 while 语句：while 为开始符号，然后识别左括号，条件表达式，右括号，执行语句。

规则如下：

```
<WHILE><LEFTPARENTHESSES>Expr()<RIGHTPARENTHESSES>Statement()
```

2.2.5 条件表达式 Expr()：

条件表达式的优先级从低到高顺序为：采用嵌套方式定义，  
||, &&, >|<|>=|<=|==|，括号表达式。

```
void Expr():{}{Expr0()(<OR> Expr0())*}  
void Expr0():{}{Condition()(<AND> Condition())*}
```

```
void Condition():{}  
Expression()("=="  
| "<"  
| "<"  
| ">"  
| ">="  
| "<=")Expression()  
}
```

2.2.6 表达式 Expression()：嵌套定义规则，运算优先级从低到高顺序为  
+|-,\*|/，运算项(因子)Factor()。其中 Factor() 又细化为变量，括号表达式，  
整型数值，字符型数值，实数型数值。

规则如下：

```
void Expression():{}  
Term()(((<PLUS>  
| <MINUS>)Term())*  
}  
void Term():{}  
Factor()(((<TIMES>  
| <DIVIDE>)Factor())*  
}  
void Factor():{}  
Identifier()  
| <LEFTPARENTHESSES>Expression()<RIGHTPARENTHESSES>  
| Integer()  
| Real()  
| String()  
}
```

2.2.7 终结符定义：

主要为 Integer(), Real(), String() 类型定义，便于出错时修改。

规则如下：

```
void Identifier():{{<IDENTIFIER>}}
void Integer():{{<INTEGER_LITERAL>}}
void Real():{{<REAL_LITERAL>}}
void String():{{<STRING>}}
```

## 2.2.8 结构体定义

Member\_list 为具体的成员变量列表。

```
/*define struct*/
StructNode defstruct(): {Token t;String n;List<Slot> membs;}
{t=<STRUCT> n=name() membs=member_list() ";"}
/*System.out.println("发现结构体!");*/
return new StructNode(location(t), new StructTypeRef(n),n,membs);//稍后添加 StructNode}}
List<Slot> member_list():
{List<Slot> membs = new ArrayList<Slot>();Slot s;}
{"{" (s=slot() ";") { membs.add(s); }}* "}" //结构体代码块
{return membs;}}

Slot slot():{TypeNode t;String n;}
t=type() n=name() { return new Slot(t, n); }//结构体成员
}
```

## 2.2.9 Return 与 Break 定义

LOOKAHEAD 超前扫描返回值是否为空。

```
BreakNode break_stmt(): {Token t;}
{t=<BREAK> ";"} { return new BreakNode(location(t)); }
ReturnNode return_stmt():
{Token t;ExprNode expr;}
{LOOKAHEAD(2) t=<RETURN> ";"} { return new ReturnNode(location(t), null); }/*函数无返回值
*/{t=<RETURN> expr=expr() ";"} { return new ReturnNode(location(t), expr); }
}
```

## 2.2.10 函数定义

Params () 为函数语法规则，LOOKAHEAD 超前扫描函数是否有返回值。

Fixedparams () 为函数参数列表，超前扫描函数是否有多个形参。

Param () 为函数体。

```
Params params():
{Token t;Params params;}
```

```

{LOOKAHEAD(<VOID> "}") t=<VOID> //无返回值
{return new Params(location(t),new ArrayList<CBCParameter>());}
| params=fixedparams() [" "..." { params.acceptVarargs();}]
{return params;}
}
Params fixedparams():
{List<CBCParameter> params = new ArrayList<CBCParameter>();
CBCParameter param, param1;}
{param1=param() { params.add(param1); }
( LOOKAHEAD(2) " " param=param() { params.add(param); } ) *
{return new Params(param1.location(), params);}}

```

```

CBCParameter param():
{TypeNode t;String n;}
{t=type() n=name() { return new CBCParameter(t, n); } //形式参数}
*/
BlockNode block():
{Token t;List<DefinedVariable> vars;List<StmtNode> stmts;}
{t="{" vars=defvar_list() stmts=stmts() "}" //函数体代码块{return new
BlockNode(location(t),vars,stmts);}}

```

## 2.2.11 冲突处理

在扫描器扫描时，可能出现选择冲突，例如，定义变量与定义函数时，前两个终结符相同，此时扫描器无法判断应该选择定义函数还是选择定义变量。可以利用 JavaCC 的 LOOKAHEAD 超前扫描，LOOKAHEAD 的规则是在相同终结符个数的基础上加一，也就是说当识别到左括号或等号时，扫描器就知道这是一个函数定义还是整数定义了。所以可以用 LOOKAHEAD 向前看三个符号，再作判断。

```

(defun=defun()          {decls.addDefun(defun);}
| LOOKAHEAD(3)
  defvars=defvars()      {decls.addDefvars(defvars);}

```

### 3. 单词类别定义

3.1 保留字定义：在 JAVACC 中采用<符号：字面值>方式定义。

```
TOKEN: { /*reserved word*/
<VOID : "void">
| <CHAR : "char">
| <SHORT : "short">
| <INT : "int">
| <LONG : "long">
| <STRUCT : "struct">
/* | <UNION : "union"> */
| <ENUM : "enum">
| <STATIC : "static">
/* | <EXTERN : "extern"> */
| <CONST : "const">
| <SIGNED : "signed">
| <UNSIGNED : "unsigned">
| <IF : "if">
| <ELSE : "else">
| <SWITCH : "switch">
| <CASE : "case">
/* | <DEFAULT : "default"> */
| <WHILE : "while">
| <DO : "do">
/* | <FOR : "for"> */
| <RETURN : "return">
| <BREAK : "break">
| <CONTINUE : "continue">
/* | <GOTO : "goto"> */
| <TYPEDEF : "typedef">
| <IMPORT : "import"> //代替 C 语言中 include
| <SIZEOF : "sizeof">
}
```

### 3.2 变量定义

```
TOKEN: {<IDENTIFIER : ["a"-"z", "A"-"Z", "_"] ([["a"-"z", "A"-"Z", "_", "0"-"9"]])*>}
```

### 3.3 整数值定义

```
TOKEN: {<INTEGER : ["1"-"9"] ([["0"-"9"]]* ~["a"-"z"] ("U")? ("L")? /*10*/
| "0" ["x", "X"] ([["0"-"9", "a"-"f", "A"-"F"])+ ("U")? ("L")? /*16*/
| "0" ([["0"-"7"]]* ("U")? ("L")? /*8*/
```

```
>}
```

### 3.4 实数定义

```
TOKEN: {<REAL_LITERAL:(<INTEGER>)+ | (<INTEGER>)+ "." | (<INTEGER>)+ "."(<INTEGER>)+  
| "."(<INTEGER>)+}
```

### 3.5 符号定义

```
TOKEN: /*定义符号*/  
{<UNDERSCORE: "_">  
| <COMMA: ",">  
| <SEMICOLON: ";">  
| <COLON: ":">  
| <L: "(">  
| <R: ")">  
| <EQUAL: "=">  
| <PLUS: "+">  
| <MINUS: "-">  
| <TIMES: "*">  
| <DIVIDE: "/">  
| <FL: "{">  
| <FR: "}">  
}
```

### 3.6 字符串字面值与字符字面值定义

```
/*字符串字面值*/  
MORE: {<"": IN_STRING> IN_STRING MORE: {(<~["'", "\"", "\n", "\r"]>)+ | <"\""  
(["0"-"7"]){3}>  
| <"\" ~[]>}  
<IN_STRING> TOKEN: {<STRING: "">: DEFAULT}  
/*字符字面值*/  
MORE: {<"": IN_CHARACTER}>  
<IN_CHARACTER> MORE: {  
<~["'", "\"", "\n", "\r"]>: CHARACTER_TERM  
| <"\" ([\"0\"-\"7\"]){3}>: CHARACTER_TERM  
| <"\" ~[]>: CHARACTER_TERM  
> <CHARACTER_TERM> TOKEN: {<CHARACTER: "">: DEFAULT}
```

### 3.7 忽略空格注释等定义

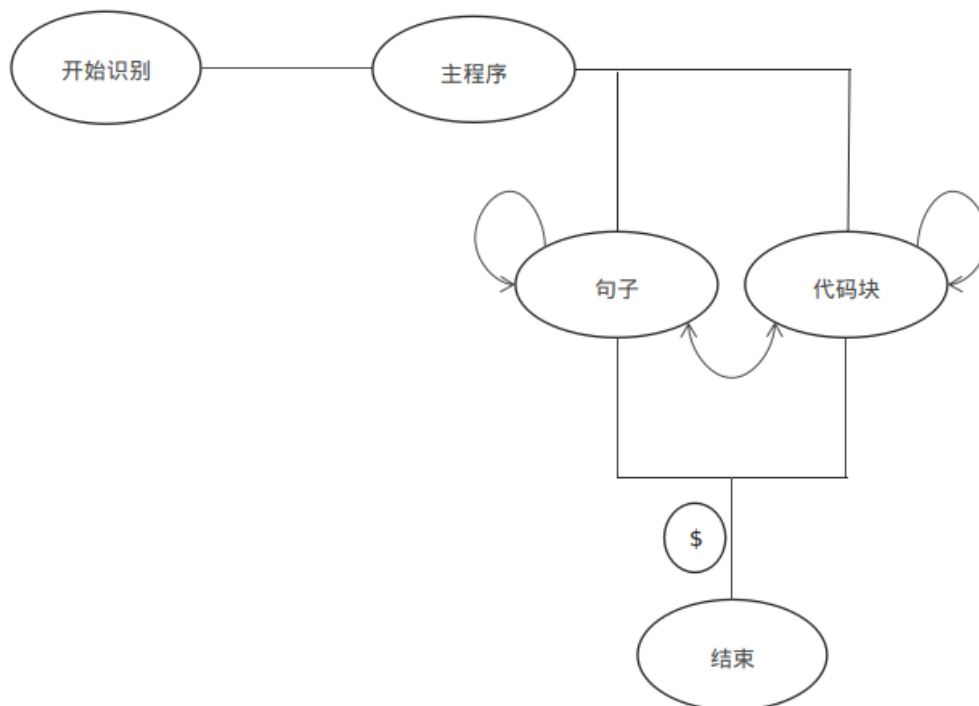
```
/*忽略空格等格式符号，但要保存记录*/  
SPECIAL_TOKEN: {<SPACES: ([\" \"\t\"\\n\"\\r\"\\f\"])+>}  
/*忽略注释，同时避免最长匹配引发的错误*/
```

```
MORE: { "<\"/*\"> : IN_BLOCK_COMMENT }  
<IN_BLOCK_COMMENT> MORE: { "<^[]> }  
<IN_BLOCK_COMMENT> SPECIAL_TOKEN: { "<BLOCK_COMMENT: \"*/\"> : DEFAULT }
```

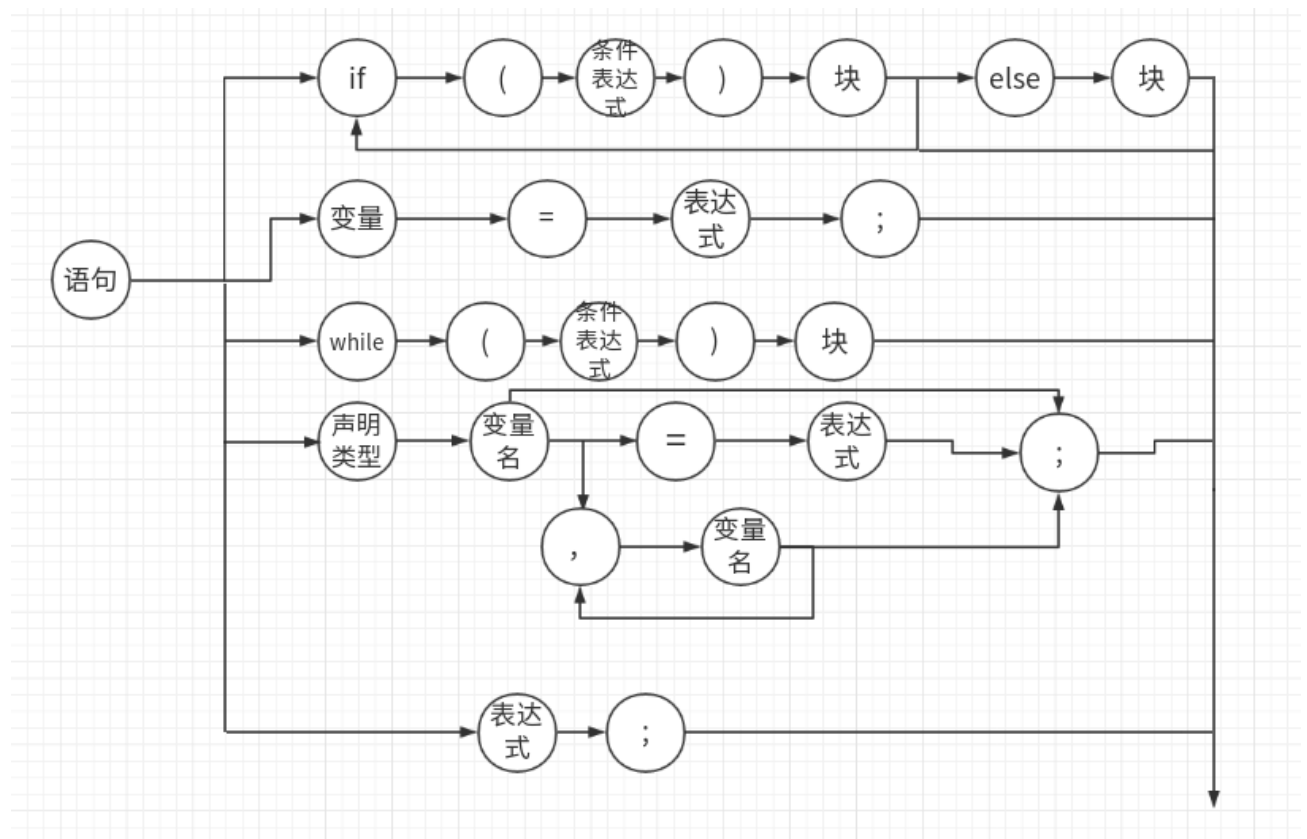
在语法分析语义部分对定义有所修改，但与词法分析单词类别定义类似，所以不再本文中列出。

## 4. 程序流程图

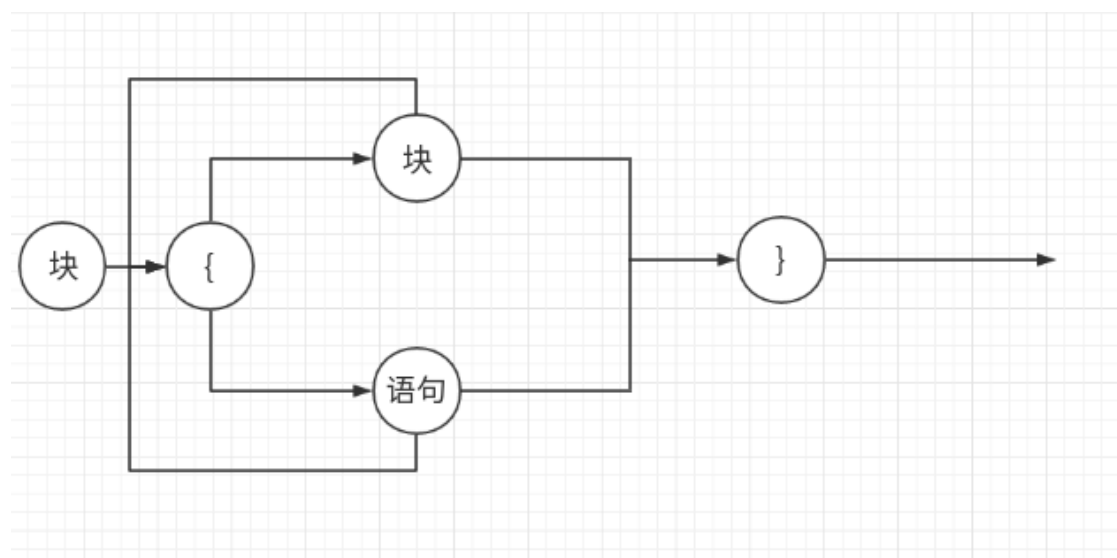
### 4.1 主程序



## 4.2 语句



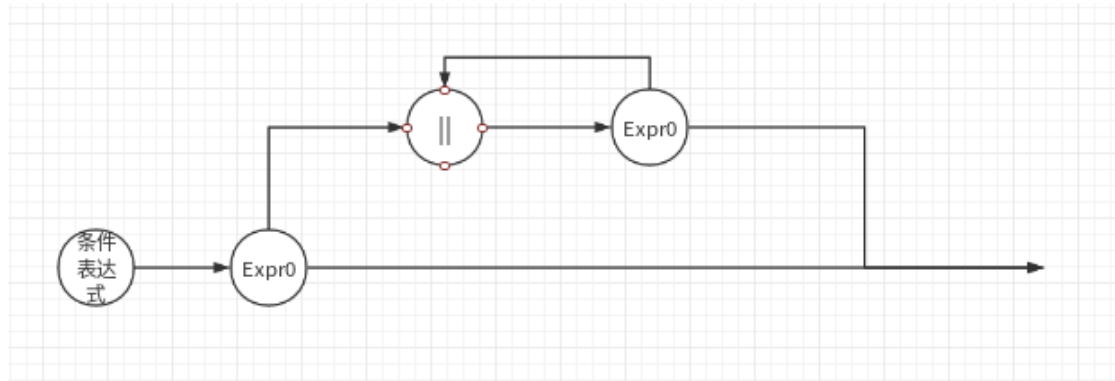
## 4.3 块



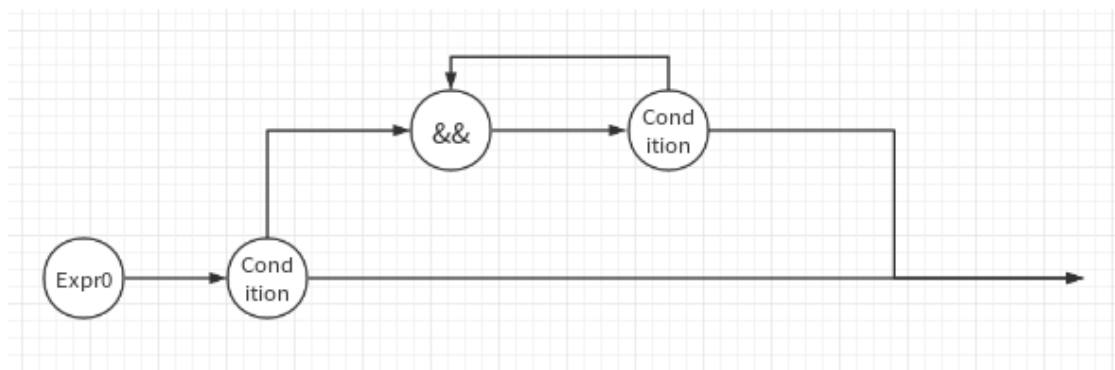


## 4.4 条件表达式

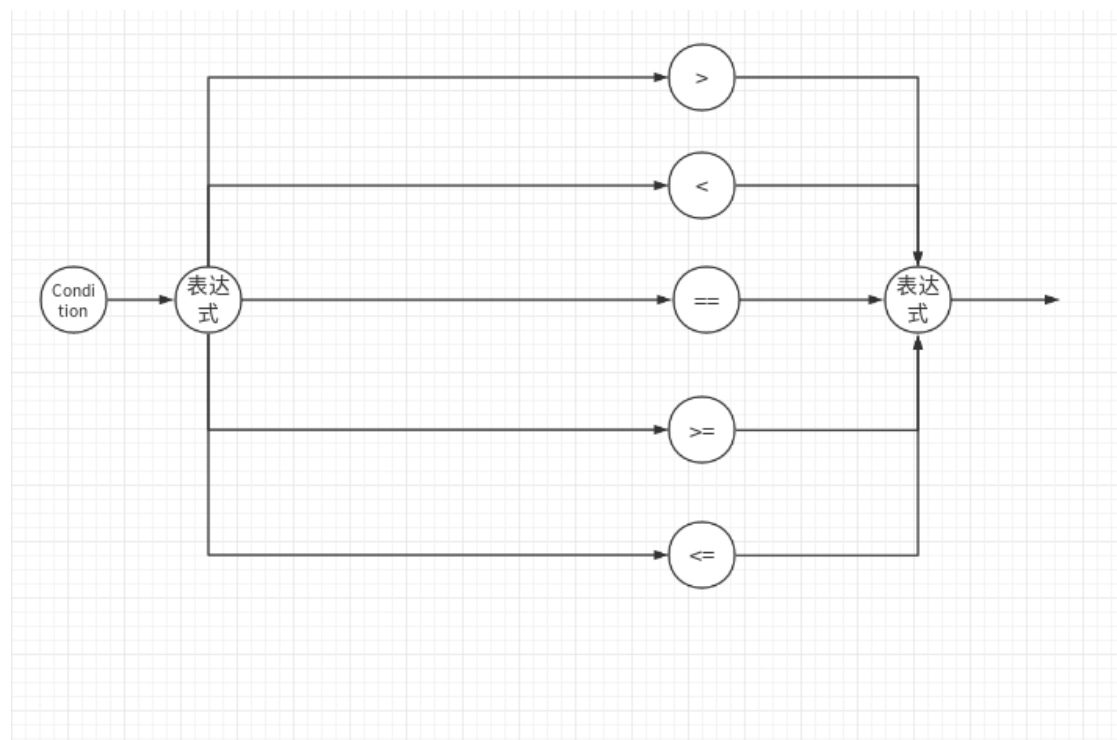
### 4.4.1 或



### 4.4.2 与

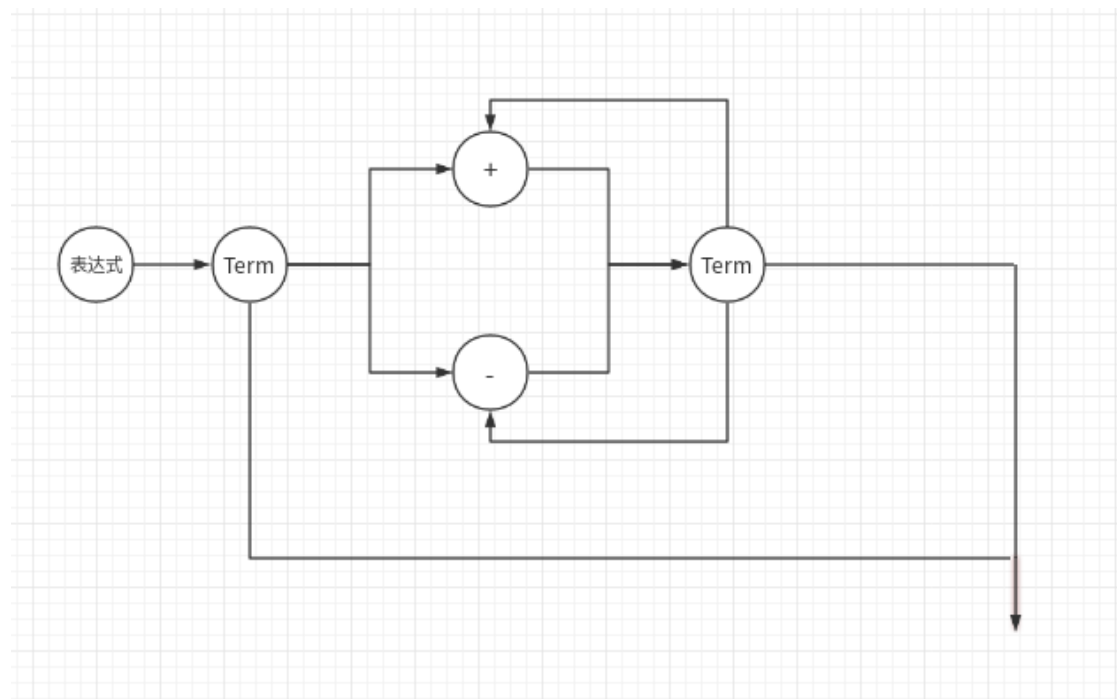


### 4.4.3 比较

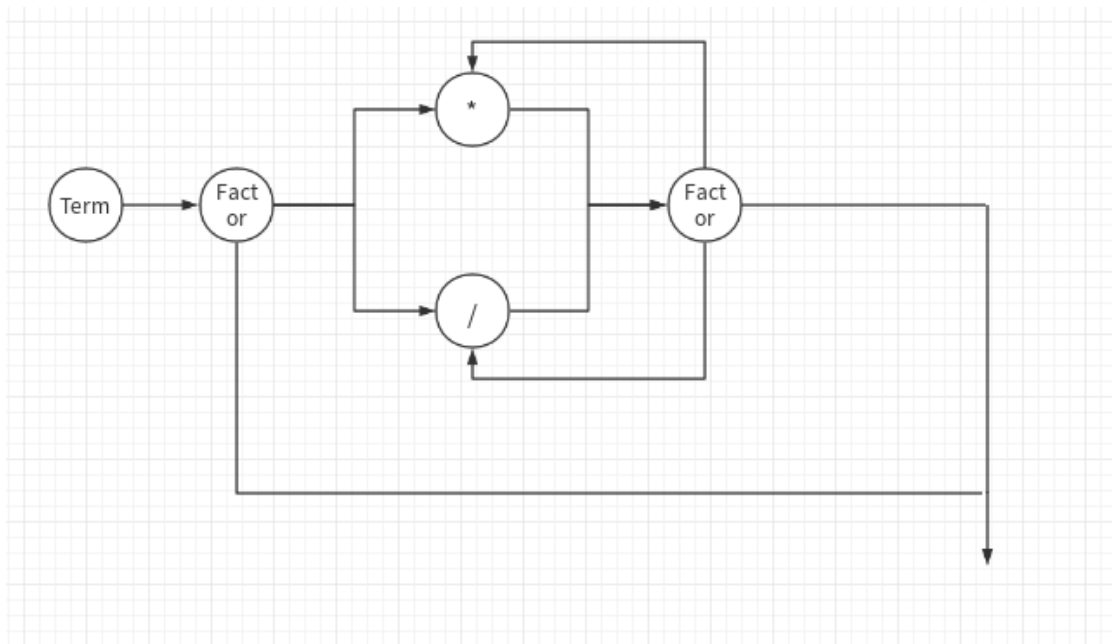


## 4.5 表达式

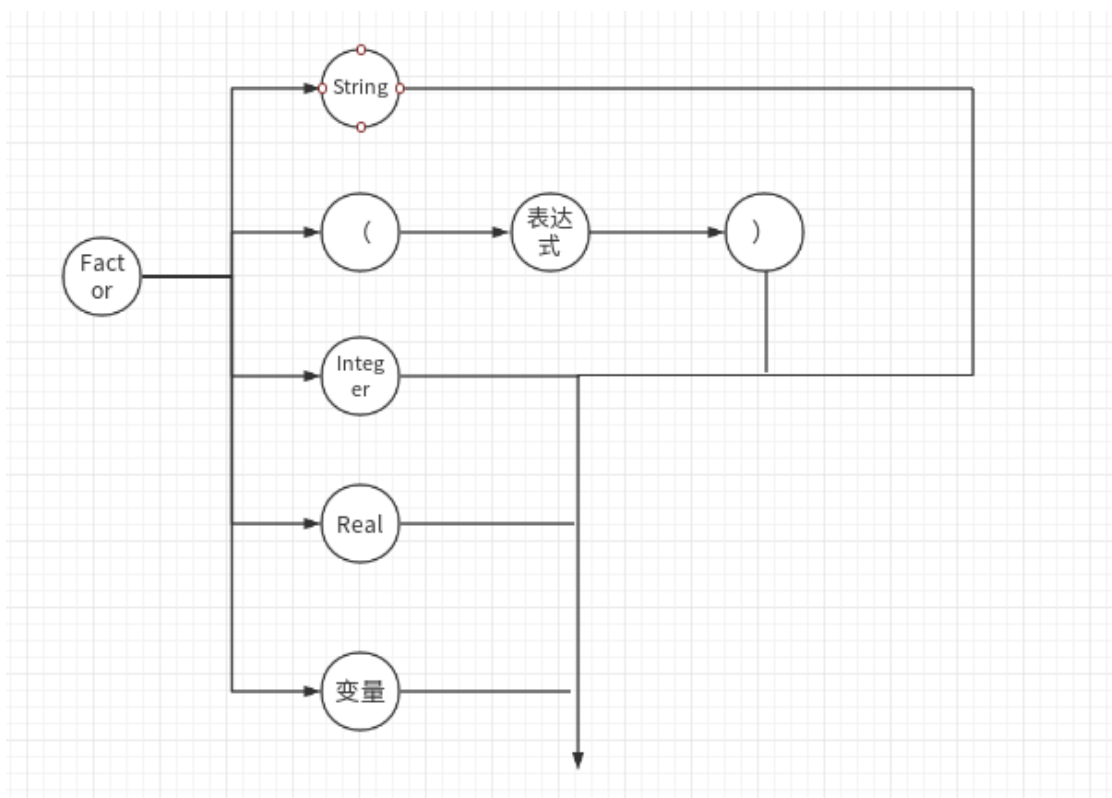
### 4.5.1 加减



### 4.5.2 乘除



#### 4.6 項



## 5. 程序源代码解读与分析

### 5.1 JAVACC 代码结构说明

5.1.1 Option{...} 用于设置 JAVACC 解析器的属性。

5.1.2 PARSER\_BEGIN()

JAVA 代码;

PARSER\_END()

为 JAVA 语言解析器，主要用于执行整个编译器程序。

5.1.3 TOKEN:....

SKIP:....

为扫描器定义部分。

5.1.4 Start(可以传入参数): {定义代码块需要使用的局部变量}

{

进行语句的识别与赋值

{被花括号包围的 JAVA 语句都可以直接执行}

} 该部分为狭义解析器

### 5.2 词法分析

Args 为命令行参数，用于获得命令行传入的文件。三个 catch 语句分别用于捕获词法分析异常，未找到输入的文件异常，输入输出异常。

```

static public void main(String[] args){
    for(String arg: args){
        try{

            System.out.println("词法分析完成");

            BufferedReader cfile = new BufferedReader(new FileReader(arg));
            String data = cfile.readLine();//一次读入一行，直到读入null为文件结束
            while( data!=null){
                findInt(data);
                data = cfile.readLine(); //接着读下一行

            }

        }
        catch(ParseException ex){
            System.err.println(ex.getMessage());
        }
        catch(FileNotFoundException ex1){
            System.err.println(ex1.getMessage());
        }
        catch(IOException ex2){
            System.err.println(ex2.getMessage());
        }
    }
}

```

findInt 函数传入读取的文件内容，然后调用 expr() 启动词法分析。

```

static public void findInt(String src) throws ParseException{
    Reader reader = new StringReader(src);
    new c1Lex(reader).expr();
}

```

JAVACC 中的 Token 类属性中的 Image 可以得到对应终结符号的 Token 字面值。通过 x=终结符号可以直接对 Token 类型变量赋值，然后新建写缓冲对象，将终结符号与字面值写入文件 aaa 中。

FileWriter 的第二个参数为 true 则可以连续写入文件内容。

当扫描器扫读到一个整数时，就会执行花括号内的 JAVA 语句。

对扫描器中定义的所有终结符号调用下图所示的方法，便可以扫描记录所有的<终结符号，字面值>二元组。

```

void expr():{
    Token x;

}
{
    (
    x=<INTEGER>
    {

        /*将x,y的类型与字面值写入文件中*/
        try{
            FileWriter fw = new FileWriter("aaa.txt",true);

            fw.write("\r");
            fw.write("INTEGER ");
            fw.write(x.image);

            fw.flush();

            fw.close();
        }
        catch(IOException ex3){
            System.err.println(ex3.getMessage());
        }

    }

}
}

```

## 5.3 语法分析

### 5.3.1 JJTree 简介

JJTree 是 JavaCC 的伙伴工具，JavaCC 不直接生成分析树或抽象语法树（AST），但提供建立分析树或 AST 生成的预处理器 JJTree，JJTree 采用压栈出栈的递归方法生成分析树，为 JavaCC 的输入进行预处理。调用 JJTree 特有的 SimpleNode 类的 dump() 方法，就可以打印语句的语法树。

### 5.3.2 主要代码解读

新建一个 c1JJTree 的对象，从键盘输入数据，然后新建一个 SimpleNode 类，用程序的开始结点对其初始化。然后调用 SimpleNode 的 dump() 方法，打印语法树，catch 语句用于检查用户输入的 C 语言子集语句是否有语法错误，并提示错误的位置与错误的修改方法。

```

System.out.println( "使用 \ $ ( 以表示输入结束) ",
new c1JJTree(System.in);
try {
    SimpleNode n = c1JJTree.Start();
    n.dump("");
    System.out.println("语法正确");
}
catch (Exception e){
    System.out.println("啊哦，语法出错啦！");
    System.out.println(e.getMessage());
}
}
}

```

语法分析程序的开始结点，用户输入” \$” 代表输入结束，然后 Start 返回一个 SimpleNode 的对象 jjtThis。JJTree 会按照（5. 程序流程图）所示的嵌套结构输出抽象语法树。

```

/*主程序，以$结束*/
SimpleNode Start():{}{
    Procedure() "$"{
        return jjtThis;
    }
}
/*分程序：语句与块*/
void Procedure():{}{
    (Statement()
    | Block()) +
}

```

## 5.4 语义分析与语义动作执行

### 5.4.1 变量定义与赋值

Var 为变量类，分为 double, int, String 类型，getx 为得到变量字面值，setx 为变量赋值。

```

class var{
    double x=1;
    int y=1;
    String varname="a";

    //int type var
    int gety(){
        return y;
    }
    // var(int var_y){
    //     y=var_y;
    // }
    void sety(int var_y){
        y=var_y;
    }
    //double type var
    double getx(){
        return x;
    }
    // var(double var_x){
    //     x=var_x;
    // }
    void setx(double var_x){
        x=var_x;
    }
    //var name
    String getn(){
        return varname;
    }
    void setn(String var_n){

```

定义变量函数，var newv 用于存储新定义的变量对象，首先新建一个变量对象，varname 接收 Identifier() 返回的变量名，然后对变量 newv 初始化。Double x 接收表达式的返回值，如果对变量赋值，则调用 new, setx() 函数存储赋值。然后打印出 newv 对象的变量名与变量值。

```

/*变量声明*/
void defvar():
{
    var newv;//新定义的变量类
    double x;
    String varname;//变量明
}
{
    {
        newv=new var();
    }
    (<INT>
    | <REAL>
    | <CHAR>)[["Integer()"]]"// [ ]also can be array
    )varname=Identifier()
    {
        newv.setn(varname);
    }
    (<EQUAL> x=Result()
    {
        newv.setx(x);
        System.out.println("新定义了一个变量: ");
        System.out.println(newv.getn());
        System.out.println("赋值为:");
        System.out.println(newv.getx());
    }
    )*
    (<COMMA>Identifier() (<EQUAL> Expression())*<SEMICOLON> //可连续定义变量，并且可以用表达式赋值
}

```



扫描器扫描到变量之后，赋值给 Token 类型变量，返回 String 类型的变量名（n.image 字面值）。

```
/*变量*/
String Identifier():
{
    Token n;
    String name;
}
{
    n=<IDENTIFIER>
    {
        name=n.image;
        return name;
    }
}
```

#### 5.4.2 while 循环语句

由于网上关于 JavaCC 的资料较少，我还没有思考出如何实现 while 语句的语义动作。目前的思路是将 statement() 的返回值定义为一个类，包括 statement() 的具体操作。条件表达式 Condition() 返回值是一个类，包括了条件表达式的左值/条件符号/右值/条件为真或假，然后利用 java 的循环语句按照条件和语句进行循环。

```
| <WHILE><LEFTPARENTHESES>Expr()<RIGHTPARENTHESES>Statement()
{
    System.out.println("while sentence");
    return 0;
}
```

#### 5.4.3 if-else 语句

注释部分的代码是针对只输入了单独的 if 语句没有 else 语句的情形。

条件表达式 Expr() 返回值为 1 代表条件为真，返回值为代表条件为假。

Statement() 返回语句块执行之后的值。

在这里我以 if-else 语句为例，t 接收紧跟 if 之后的语句的返回值，f 接收 else 语句的返回值。在花括号内执行 JAVA 的 if 语句根据 Condi 条件值的真假选择输出哪条分支的结果。

```

|(<IF><LEFTPARENTHESSES>condi=Expr()<RIGHTPARENTHESSES>t=Statement()
// this for if alone to use
// {

//   if(condi==1)//t
//   {
//       System.out.println(t);
//       return t;
//   }
//   else
//   {
//       System.out.println("this if will not be done");
//       return 0;
//   }
// }
[<ELSE>f=Statement()
{
    if(condi==1)//t
    {
        System.out.println(t);
        return t;
    }
    else
    {
        System.out.println(f);
        return f;
    }
}
]

)+

```

条件表达式 Expr()

首先从最底层的 Condition() 函数分析。Token 类型的 condi 用于存储扫描到的条件运算符, double 类型的 L 接收条件运算符的左值, R 接收条件运算符的右值。

```

/*条件运算*/
int Condition():
{
    double l,r;
    Token condi;
}
{
    l=Expression()(
    condi=="
    //| "<"
    | condi="<"
    | condi=">"
    | condi=">="
    | condi="<="
    )r=Expression()

```

花括号内为执行 JAVA 语句,根据 condi 的值,执行对应的 if 语句并返回给 Expr0()  
int 类型的条件运算值, 1 为真, 0 为假。

```

{
    //for test
    // System.out.println(l);
    // System.out.println(r);
    // System.out.println(condi.image);
    if(condi.image=="<"){
        if(l<r)return 1;
        else return 0;
    }
    if(condi.image==">"){
        if(l>r)return 1;
        else return 0;
    }
    if(condi.image=="<="){
        if(l<=r)return 1;
        else return 0;
    }
    if(condi.image==">="){
        if(l>=r)return 1;
        else return 0;
    }
    if(condi.image=="=="){
        if(l==r)return 1;
        else return 0;
    }
}
}

```

x 为与与运算的左值，y 为右值，用于获得条件表达式返回的逻辑值，花括号内执行 JAVA 运算，根据与运算的规律，在这里我用乘法代替与运算，因为当有一个值为 0 时，条件表达式的值就为 0，然后立即返回 0。若不为 0，由于是乘法，直接返回 x 的字面值即可。

```
//与 主要用于条件表达式
int Expr0()://乘法代替与
{
    int x,y;
}
{
    x=Condition()
    (
        <AND> y=Condition()
        {
            x*=y;
            if(x==0)return 0;
        }
    ) *
    {
        return x;
    }
}
```

x 为或运算的左值，y 为右值，用于获得条件表达式返回的逻辑值，花括号内执行 JAVA 运算，根据或运算的规律，在这里我用加法代替或运算，因为只有所有值全为 0 时，条件表达式的值才为 0，由于是加法运算，返回 x 即可。若不为 0，直接返回 1。

```

//或 主要用于条件表达式
int Expr()://加法代替或
{
    int x,y;
}{
    x=Expr0()
    (
        <OR> y=Expr0()
        {
            x+=y;
            if(x>0) return 1;
        }
    )*
    {
        return x;
    }
}

```

#### 5.4.4 表达式

当扫描语句为表达式时，返回 double 类型的表达式计算值。

```

}
| t=Result() <SEMICOLON>
{
    return t;
}

```

Result() 接收表达式的计算值，并返回一个 double 类型值。

```

double Result():
{
    double x;
}
{
    x=Expression()
    {
        return x;
    }
}

```

Expression() 进行加减运算，x 接收 Term() 返回的运算符左值，y 接收 Term() 返回的运算符右值。当识别到加法符号时，花括号执行加法运算，识别到减法时，执行减法运算，当所有的加减运算都执行完毕之后，return 返回计算出的 x 值。

```
double Expression():
{
    double x,y,z;
}
{
    (x=Term()
    (<PLUS> y=Term()
    {
        x+=y;
    }
    |
    <MINUS> z=Term()
    {
        x-=z;
    }
    ) *
    )
    {
        return x;
    }
}
```

Term() 进行加减运算，x 接收 Facor() 返回的运算符左值，y 接收 Facor() 返回的运算符右值。当识别到乘法符号时，花括号执行乘法运算，识别到除法时，执行除法运算，当所有的乘除运算都执行完毕之后，return 返回计算出的 x 值。

```

double Term():
{
    double x,y,z;
}
{
    (x=Factor()
    (<TIMES> y=Factor()
    {
        x*=y;
    }
    |
    <DIVIDE> z=Factor()
    {
        x/=z;
    }
    )*
    )
    {
        return x;
    }
}
}

```

Factor() 函数，当扫描到整型数值时，返回 int 类型的值，扫描实数类型数值时，返回 double 类型。

```

/*运算项*/
double Factor():
{
    double x;
}
{
    // Identifier()
    (
    //<LEFTPARENTHESES>x=Expression()<RIGHTPARENTHESES>
    x=Integer()
    | x=Real()
    )
    {
        return x;
    }
    // | String()
}
}

```

整型与实数型，分为两种情况，一种是扫描到 x 存储整（实数）型值，并返回整（实数）型的字面值。另一种情况为识别到表达式，sum 获得表达式的返回值（整型或实数型），并将 sum 返回给上一层的 Factor()。

```
double Integer():
{
    Token x;
    double sum;
}
{
    x=<INTEGER_LITERAL>
    {
        return Integer.parseInt(x.image);
    }
    |
    <LEFTPARENTHESES>sum=Expression()<RIGHTPARENTHESES>
    {
        return sum;
    }
}
/*实数*/
double Real():
{
    Token x;
    double sum;
}
{
    x=<REAL_LITERAL>
    {
        return Double.parseDouble(x.image);
    }
    |
    <LEFTPARENTHESES>sum=Expression()<RIGHTPARENTHESES>
    {
        return sum;
    }
}
```

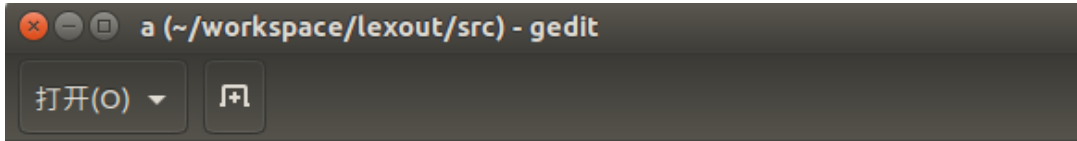


## 6. 程序结果

### 6.1 词法分析

#### 6.1.1 正确运行

程序输入：文件内容如下。



```
/*lex by wangiyyi*/
int main(){

    int _x,y;
    int z=1.1;
    int xy;
    string c="stringtest";
    char d='w';
    if(x=y)
    {
        x=x*1;
    }
    else y=y-1;
    while(a=1)
    {
        x=x+1;
    }
    switch(a)
    {
        case 1:a+1;break;
        case 2:b-1;break;
    }
    return 0;
}
/**/
```

运行方式，首先进入 c1Lex. jj 文件所在的工作区，用终端打开该文件夹。

通过 Javacc c1Lex. jj 命令，对. jj 文件进行编译。

```

oneone@oneone-Inspiron-7548:~/workspace/lexout/src$ javacc c1Lex.jj
Java HotSpot(TM) 64-Bit Server VM warning: Insufficient space for shared memory
file:
27942
Try using the -Djava.io.tmpdir= option to select an alternate temp location.

Java Compiler Compiler Version 5.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file c1Lex.jj . . .
Warning: "_" cannot be matched as a string literal token at line 131, column 5.
It will be matched as <IDENTIFIER>.
File "TokenMgrError.java" is being rebuilt.
File "ParseException.java" is being rebuilt.
File "Token.java" is being rebuilt.
File "SimpleCharStream.java" is being rebuilt.
Parser generated with 0 errors and 1 warnings.

```

编译后生成相关的 java 文件。

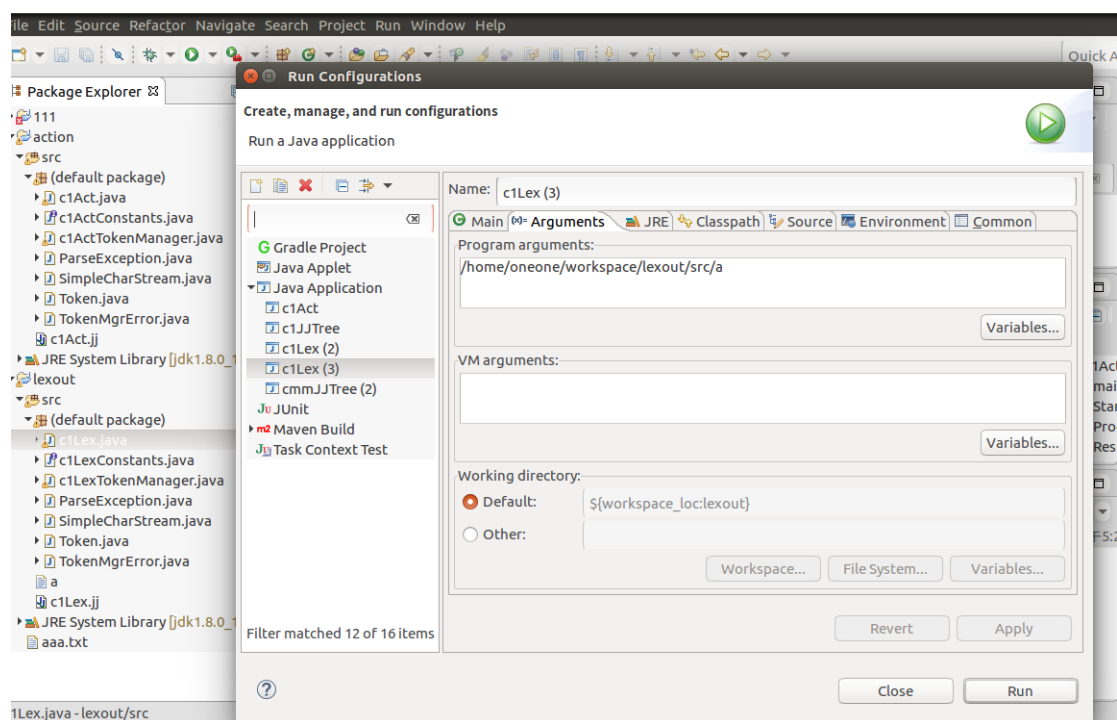
```

oneone@oneone-Inspiron-7548:~/workspace/lexout/src$ ls
a
c1Lex.class          c1LexTokenManager.class  Token.class
c1LexConstants.class c1LexTokenManager.java   Token.java
c1LexConstants.java  ParseException.class     TokenMgrError.class
c1Lex.java           ParseException.java       TokenMgrError.java
c1Lex.jj             SimpleCharStream.class
SimpleCharStream.java

```

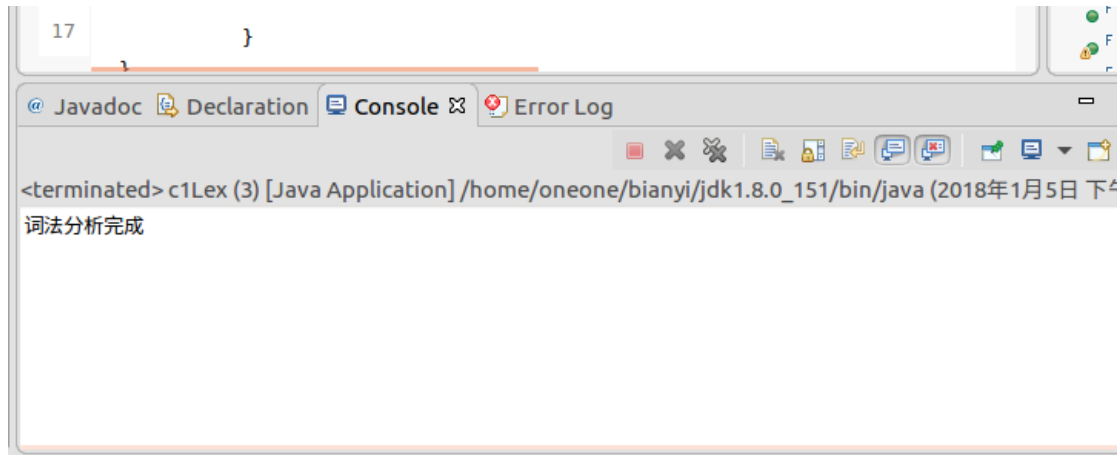
再用 `javac *.java` 生成可执行的 `c1Lex.class` 类。

由于用 Eclipse 运行更方便，于是刷新工作区，编译运行 `c1Lex.java`。由于是从文件中读取数据，所以在运行前，需要点击 Run configuration 的 arguments 设定要读取文件的绝对路径，如下图所示。在这里我读取的是工作区中的 `a` 文件。

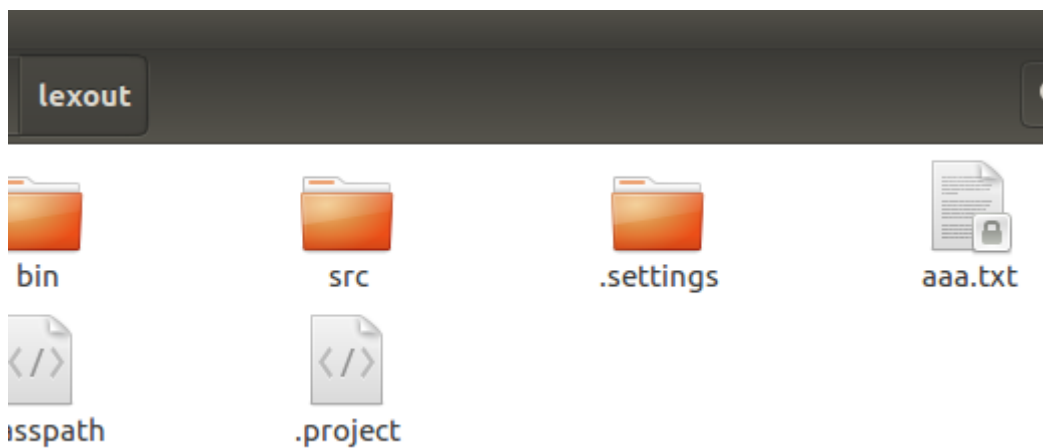


为了快速观察结果，在代码中我直接将输出结果写在工作区的 aaa 文件中。（输出文件也可以改为用户手动输入路径然后保存。）

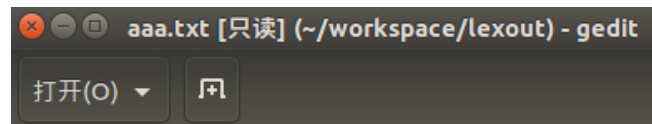
运行成功后，会提示，词法分析完成。如下图所示：



然后我们到工作区目录下找到 aaa 文件。



打开可以看到，输出结果为定义的单词与对应的源程序代码的字面值的二元组。



```
INT int
IDENTIFIER main
L (
R )
FL {
INT int
IDENTIFIER _x
COMMA ,
IDENTIFIER y
SEMICOLON ;
INT int
IDENTIFIER z
EQUAL =
REAL_LITERAL 1.1;
INT int
IDENTIFIER xy
SEMICOLON ;
IDENTIFIER string
IDENTIFIER c
EQUAL =
STRING "stringtest"
SEMICOLON ;
CHAR char
IDENTIFIER d
EQUAL =
CHARACTER 'w'
SEMICOLON ;
IF if
L (
IDENTIFIER x
EQUAL =
IDENTIFIER y
R )
FL {
IDENTIFIER x
```

```

IDENTIFIER a
EQUAL =
IDENTIFIER x
TIMES *
INTEGER 1;
FR }
ELSE else
IDENTIFIER y
EQUAL =
IDENTIFIER y
MINUS -
INTEGER 1;
WHILE while
L (
IDENTIFIER a
EQUAL =
INTEGER 1)
FL {
IDENTIFIER x
EQUAL =
IDENTIFIER x
PLUS +
INTEGER 1;
FR }
SWITCH switch
L (
IDENTIFIER a
R )
FL {
CASE case
INTEGER 1:
IDENTIFIER a
PLUS +
INTEGER 1;
BREAK break
SEMICOLON ;

```

```

IDENTIFIER b
MINUS -
INTEGER 1;
BREAK break
SEMICOLON ;
FR }
RETURN return
INTEGER 0
SEMICOLON ;
FR }

```

在结果中我们可以看出，扫描程序跳过了空格注释等内容，同时避免了由于最长匹配导致的在两个注释中间的代码也被忽略的错误。

### 6.1.2 错误运行报错举例

(1) 输入文件中，我将内容修改为未闭合的注释。

```

        case 2:b-1;break;
    }
    return 0;
}
/*

```

执行后报错。

```

Exception in thread "main" TokenMgrError: Lexical error at line 1, column 3. Encountered: <
    at c1LexTokenManager.getNextToken(c1LexTokenManager.java:1493)
    at c1Lex.jj_ntk(c1Lex.java:701)
    at c1Lex.expr(c1Lex.java:53)
    at c1Lex.findInt(c1Lex.java:43)
    at c1Lex.main(c1Lex.java:14)

```

(2) 输入文件中，我将标识符定义为数字开头的（在 C 语言中不允许，将报错）。

```

/*lex by wangyiyi*/
int main(){
    int _x,y;
    int z=1.1;
    int 6yiyi;
}

```

```

<terminated> c1Lex (3) [Java Application] /home/oneone/bianyi/jdk1.8.0_151/bin/java (2018年1月5日 下午9:
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 14. Encountered:
    at c1LexTokenManager.getNextToken(c1LexTokenManager.java:1493)
    at c1Lex.jj_ntk(c1Lex.java:701)
    at c1Lex.expr(c1Lex.java:53)
    at c1Lex.findInt(c1Lex.java:43)
    at c1Lex.main(c1Lex.java:14)

```

## 6.2 语法分析

为了便于调试与观察结果，语法分析的输入输出都置于屏幕上，如果改为文件输入输出，只需按照词法分析的代码修改即可。

在语法分析部分，我使用了 JavaCC 中自带的 JJTree，它能够根据我们定义的语法规则，自动生成对应的语法树。

具体的运行方法为，首先在命令行中使用 `jtree c1JJtree.jjt` 语句编译 `jtree` 文件，然后会生成对应的 `c1JJtree.jj`，接下来的运行方式就与 `JavaCC` 文件一样了。

(1) 输入定义变量语句，输出对应结构的语法树，并显示语法正确的提示。

```
***欢迎使用C1-C语言子集语法分析器***
请输入C语言子集语句:
例如, 您可以输入
"int a;" "a = 3;" "if(a==0)int b;" "while(a>0) b=b+1;"
使用 "$" 以表示输入结束>>
int a;
$
|start
|  Procedure
|    Statement
|      defvar
|        Identifier
语法正确
```

如果输入错误的 C 语言语句则会提示错误的位置，并且给出应当如何修改 C 语言语句。例如我想声明一个变量，却忘记给出变量名称。

```
***欢迎使用C1-C语言子集语法分析器***
请输入C语言子集语句:
例如, 您可以输入
"int a;" "a = 3;" "if(a==0)int b;" "while(a>0) b=b+1;"
使用 "$" 以表示输入结束>>
int ;
啊哦, 语法出错啦!
Encountered " ";" ";" "" at line 1, column 5.
Was expecting one of:
    <IDENTIFIER> ...
    "[" ...
```

(2) 输入表达式语句。

```

***欢迎使用C1-C语言子集语法分析器***
请输入C语言子集语句:
例如, 您可以输入
"int a;" "a = 3;" "if(a==0)int b;" "while(a>0) b=b+1;"
使用 "$" 以表示输入结束>>
((1+3)*5+4)/6;
$

```

生成的语法树如下图所示。

```

Start
  Procedure
    Statement
      Expression
        Term
          Factor
            Expression
              Term
                Factor
                  Expression
                    Term
                      Factor
                        Integer
                    Term
                      Factor
                        Integer
                  Factor
                    Integer
                Term
                  Factor
                    Integer
            Factor
              Integer
          Term
            Factor
              Integer
        Factor
          Integer
      语法正确

```

如果输入表达式有误, 则会报错。例如我在计算表达式的适合没有闭合括号。



\*\*\*欢迎使用C1-C语言子集语法分析器\*\*\*

请输入C语言子集语句:

例如, 您可以输入

```
"int a;" "a = 3;" "if(a==0)int b;" "while(a>0) b=b+1;"
```

使用 "\$" 以表示输入结束>>

```
(1+3+5;
```

啊哦, 语法出错啦!

Encountered " ";" " at line 1, column 7.

Was expecting one of:

)" ...

+" ...

-" ...

\*" ...

/" ...

(3) 输入符合语法规则的 while 语句。

```
while(a>1) x=x+1;
```

```
$
```

Start

Procedure

Statement

Expr

Expr0

Condition

Expression

Term

Factor

Identifier

Expression

Term

Factor

Integer

Statement

Identifier

Expression

Term

Factor

Identifier

Term

Factor

Integer

语法正确

输入错误的 while 语句，例如我们平时写 C 语言代码常犯的一个错误，就是条件表达式不小心将==写成=赋值，则会出现如下图所示的报错。

```
***欢迎使用C1-C语言子集语法分析器***
请输入C语言子集语句：
例如，您可以输入
"int a;" "a = 3;" "if(a==0)int b;" "while(a>0) b=b+1;"
使用 "$" 以表示输入结束>>
while(a=1) x=x+1;
啊哦，语法出错啦！
Encountered " =" "=" at line 1, column 8.
Was expecting one of:
    "+" ...
    "-" ...
    "*" ...
    "/" ...
    "==" ...
    "<>" ...
    "<" ...
    ">" ...
    ">=" ...
    "<=" ...
```

(4) 输入赋值语句。

```
***欢迎使用C1-C语言子集语法分析器***
请输入C语言子集语句：
例如，您可以输入
"int a;" "a = 3;" "if(a==0)int b;" "while(a>0) b=b+1;"
使用 "$" 以表示输入结束>>
a=1;
$
Start
  Procedure
    Statement
      Identifier
      Expression
      Term
      Factor
      Integer
语法正确
```

(5) 输入 if-else 语句。

```
***欢迎使用C1-C语言子集语法分析器***
```

```
请输入C语言子集语句:
```

```
例如, 您可以输入
```

```
"int a;" "a = 3;" "if(a==0)int b;" "while(a>0) b=b+1;"
```

```
使用 "$" 以表示输入结束>>
```

```
if(a==0) b=b+1; else b=b-1;
```

```
$
```

```
L. .
```

Start

Procedure

Statement

Expr

Expr0

Condition

Expression

Term

Factor

Identifier

Expression

Term

Factor

Real

Statement

Identifier

Expression

Term

Factor

Identifier

Term

Factor

Integer

```

Statement
  Identifier
  Expression
  Term
  Factor
  Identifier
  Term
  Factor
  Integer
语法正确

```

同样如果在 if 语句的条件表达式中出现等号，会报错。

```

***欢迎使用C1-C语言子集语法分析器***
请输入C语言子集语句：
例如，您可以输入
"int a;" "a = 3;" "if(a==0)int b;" "while(a>0) b=b+1;"
使用 "$" 以表示输入结束>>
if(a=1)x=x+1;
啊哦，语法出错啦！
Encountered " "=" "= " at line 1, column 5.
Was expecting one of:
    "+" ...
    "-" ...
    "*" ...
    "/" ...
    "==" ...
    "<>" ...
    "<" ...
    ">" ...
    ">=" ...
    "<=" ...

```

如果写完 If 语句之后，没有写对应的执行代码，也会报错。

(6) 如果写代码时，忘记加分号也会报错。

\*\*\*欢迎使用C1-C语言子集语法分析器\*\*\*

请输入C语言子集语句:

例如, 您可以输入

```
"int a;" "a = 3;" "if(a==0)int b;" "while(a>0) b=b+1;"
```

使用 "\$" 以表示输入结束>>

```
int a=1
```

```
$
```

啊哦, 语法出错啦!

Encountered " "\$" "\$ "" at line 2, column 1.

Was expecting one of:

```
"," ...  
";" ...  
"=" ...
```

\*\*\*欢迎使用C1-C语言子集语法分析器\*\*\*

请输入C语言子集语句:

例如, 您可以输入

```
"int a;" "a = 3;" "if(a==0)int b;" "while(a>0) b=b+1;"
```

使用 "\$" 以表示输入结束>>

```
if(a==1)
```

```
$
```

啊哦, 语法出错啦!

Encountered " "\$" "\$ "" at line 2, column 1.

Was expecting one of:

```
"if" ...  
"while" ...  
"int" ...  
"real" ...  
"char" ...  
<INTEGER_LITERAL> ...  
<REAL_LITERAL> ...  
"(" ...  
<IDENTIFIER> ...  
"\" ...
```

(7) 将所有 C 语句结合成完整的 C 语言代码, 得到的结果如下。

\*\*\*欢迎使用C1-C语言子集语法分析器\*\*\*

请输入C语言子集语句:

例如, 您可以输入

"int a;" "a = 3;" "if(a==0)int b;" "while(a>0) b=b+1;"

使用 "\$" 以表示输入结束>>

```
{int a=6;    if(a>1) x=x+1;else x=x-1;    while(a<10)b=((3+4)*5+8)/6;}  
$  
|
```

Start

Procedure

Block

Statement

defvar

Identifier

Expression

Term

Factor

Integer

Statement

Expr

Expr0|

Condition

Expression

Term

Factor

Identifier

Expression

Term

Factor

Integer

```

Statement
  Identifier|
  Expression
    Term
      Factor
        Identifier
      Term
        Factor
          Integer
Statement
  Identifier
  Expression
    Term
      Factor
        Identifier
    Term
      Factor
        Integer

```

```

Statement
Expr
Expr0
Condition
Expression
Term
Factor
Identifier
Expression
Term
Factor
Integer

```

```

Statement
Identifier
Expression
Term
Factor
Expression
Term
Factor
Expression
Term
Factor
Integer
Term
Factor
Integer

```

```

Factor
Integer
Term
Factor
Integer
Factor
Integer

```

语法正确

## 6.3 语义分析及执行语义动作



(1) 识别复杂的表达式，并计算表达式的值。

```
***欢迎使用C1-C语言子集语义分析器***
请输入C语言子集语句 例如:(int a = 8; if(1>2) 1+8; else 1+1; while(1>3) 2+2;):
((1+3)*5+8)/6 ;
$
4.666666666666667
```

(2) 识别 if 语句，根据条件表达式的结果判断执行哪条分支。

为了便于观察结果，条件表达式我直接输入了显示的数值。可以看到结果显示，由于表达式的值为 false 执行了 else 分支，并输出了表达式的计算结果。

```
***欢迎使用C1-C语言子集语义分析器***
请输入C语言子集语句 例如:(int a = 8; if(1>2) 1+8; else 1+1; while(1>3) 2+2;):
if(1>2) (1+5)*2 ; else 1+1;
2.0
```

将条件改为小于之后，条件表达式为 true，执行的是紧跟在 if 后的分支。

```
c1Act [Java Application] /home/oneone/bianyi/jdk1.8.0_151/bin/java (2018年1月5日 下午9
***欢迎使用C1-C语言子集语义分析器***
请输入C语言子集语句 例如:(int a = 8; if(1>2) 1+8; else 1+1; while(1>3) 2+2;):
if(1<2) (1+5)*2 ; else 1+1;
12.0
```

(3) 识别 while 语句，由于网上关于 JavaCC 的资料太稀有，所以我还没有弄清楚如何写一个 while 语句的执行动作。因此，在这里没有具体实现 while 循环，只是打印出识别到 while 语句的提示，循环功能将在后期实现。

```
c1Act [Java Application] /home/oneone/bianyi/jdk1.8.0_151/bin/java (2018年1月5日 下午9:11)
***欢迎使用C1-C语言子集语义分析器***
请输入C语言子集语句 例如:(int a = 8; if(1>2) 1+8; else 1+1; while(1>3) 2+2;):
while(1>3) (1+3)*5;
while sentence
```

(4) 定义变量与赋值语句。定义变量时，新建了一个变量类 var，在类中保存有新建变量的名称，赋值后的字面值。变量定义分为 int, real, string, char 四种，下图为定义 int 类型的演示，其它类型与之类似。单独的赋值语句与变量赋值类似，也不再演示。

```
c1Act [Java Application] /home/oneone/bianyi/jdk1.8.0_151/bin/java (2018年1月5日 下午9:11)
***欢迎使用C1-C语言子集语义分析器***
请输入C语言子集语句 例如:(int a = 8; if(1>2) 1+8; else 1+1; while(1>3) 2+2;):
int yiyi=100;
新定义了一个变量:
yiyi
赋值为:
100.0
```

## 7. 存在的问题与解决方法

工程问题：由于网上关于 JAVACC 与 ECLIPSE 结合编译的资料太少，不知道为什么 JAVACC 插件在 ECLIPSE 中没有编译代码的权限，始终不知如何设置，所以只能通过命令行进行编译再在 ECLIPSE 中运行了。

直接在命令行中使用 JAVA 命令报错:无法加载类，由于网上的解决方法无效，不熟悉 LINUX 系统，至今未找到原因。

只能使用 ECLIPSE 运行 JAVACC 生成的文件，同时要注意如果 .jj 文件中没有声明 Package 那就不能在 ECLIPSE 的包中打开，会报错，必须要直接在 SRC 文件夹下生成 .java 文件。

### 7.1 词法分析程序

7.1.1 写入文本文件应该使用宏定义的方法，否则不易修改。

7.1.2 STRING 识别代替了字符串的识别。可以将字符规定为单引号，或特殊加上一些特殊识别标志。

7.1.3 在扫描的过程中相同的标识符重复识别了多次,能否不重复，扫描一次之后保存即可，不必再重复写入。

7.1.4 识别函数 `expr()` 的代码可以用一个函数代替，降低代码量。以识别指针与识别单元的字面值作为函数的参数。

7.1.5 检查到非法标识符，比如数字开头的标识符时，只能提示错误的位置，无法打印错误的原因（比如非法标识符这句话）。

7.1.6 按照《自制编译器》书中所讲，用 `parserConstants.tokenImage[token.kind]` 可以获得 `token` 的“类型”常量，但我在代码中加入该语法后，报错：没有 `parserConstants` 类。由于目前确实没有在网上查阅到 `parserConstants` 类的相关资料，所以有待考究。

## 7.2 语法分析程序

7.2.1 如果将所有运算符（条件表达式与算数表达式）按照优先级嵌套到一起使用，会发生歧义，需要用到 JavaCC 的 LOOKAHEAD 函数解决。

7.2.2 在 JJTREE 中使用 `()*` 会报错→可以为空串，在网上查了一下但是资料甚少，以至于目前还未找到解决办法，只能将所有的语句都使用 `()+`，这样定义的缺点就是无法接收空语句必须输入定义的语句，否则就会报错。

对于循环语句 `while()`，还没有实现嵌套的循环语句规则。

## 7.3 语义分析程序

7.3.1 `if else` 与单独只有 `If` 语句在句子识别时，需要区分开，在代码中我还没有区分，只能识别 `if-else`，如果不区分并直接为单独的 `If` 语句添加动作，将会重复两遍 `if` 的语义动作然后报错。

7.3.2 `statement()` 必须要有返回值，而且返回值需要定义为很多类型，目前我仅仅把返回值定义为 `double` 只对数字类型的变量起作用，这就导致了当执行的代码块是 `statment()` 中的定义变量语句的时候，会在正确输出新建的变量之后报错。如果加返回值则这个返回值没有任何意义。

目前我想到的解决方法就是把起始语句 `statment()` 的返回类型定义为类，然后再在类中逐渐添加各种变量的类型。由于所剩时间较短所以只实现了对数字类型有效的语句。

7.3.3 语义分析中，我没有单独实现赋值语句，直接在变量定义中实现了赋值语句，只以定义一个变量并且赋值为例，多个变量定义并赋值的代码与之类似。

7.3.4 WHILE 语义动作的实现：将 `statement()` 的返回值定义为一个类，包括 `statement()` 的具体操作。条件表达式 `Condition()` 返回值是一个类，包括了条件表达式的左值/条件符号/右值/条件为真或假，然后利用 java 的循环语句按照条件和语句进行循环。

7.3.5 关于语义分析的最终结果，没有实现打印语句的四元组功能。而我实现的功能是将语义要执行的动作通过 JAVA 语句实际的执行。四元组的含义是语句的执行顺序含义与值的存储，这个部分我在实现语义动作时就已经实现了，因为在语法规则定义时便确定了优先级与嵌套结构，将每一条识别出的语句的值进行存储并返回给上一级。如果把四元组比作一个公式，那么我实现的则是利用公式解决问题并得到结果。但是本门课程的要求是打印四元组，输出四元组与实现语义执行动作相比更加简单，我将在以后制作 C 语言编译器的过程中实现。

## 参考资料

《自制编译器》 [日]青木峰郎

JAVACC 官方 HTML 文档

<http://cs.lmu.edu/~ray/notes/javacc/> Using JavaCC

<http://digital.cs.usu.edu/~allanv/cs4700/javacc/javacc.html>

<https://www.codeproject.com/Articles/35748/An-Introduction-to-JavaCC>

<https://www.javaworld.com/article/2076269/learn-java/build-your-own-languages-with-javacc.html>

<https://javacc.org/>

[https://www.cnblogs.com/Gavin\\_Liu/archive/2009/03/07/1405029.html](https://www.cnblogs.com/Gavin_Liu/archive/2009/03/07/1405029.html)  
JavaCC 研究与应用