

1. Application background

CA compiler for a subset of a language that can translate source programs in that language into target programs in other forms of intermediate languages, and can perform corresponding semantic actions (such as calculating the value of complex expressions, if-else statement jumps, conditional judgments, variable definitions and assignments, loop statements) to obtain results, realizing the basic functions of the C language.

2. Syntax description

2.1 is mainly divided into three parts: lexical analysis, syntax analysis, and semantic analysis.

Lexical analysis includes: code block, if-else statement, while statement, switch-case statement, do-while statement, variable definition (including char, real, int, string), assignment statement, expression, definition structure, word Identification of face value. (In the Adder.jj file, I also defined grammatical rules for structures, functions, pointers, etc., which will be implemented later)

Syntax analysis includes: code blocks, if-else statements, while statements, variable definitions and assignment statements, assignment statements, expressions, and conditional statements.

Semantic analysis includes: code blocks, if-else statements, while statements, variable definitions and assignment statements, assignment statements, expressions, and conditional statements.

2.2 Detailed grammar rules

2.2.1 Code block: starts with "{" and ends with "}". Code blocks or C language statements can be nested in the middle.

The rules are as follows:

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

2.2.2 Variable definition statement: Use int, char, string, etc. as the starting symbols for identification, and you can define multiple variables separated by commas and assign values to the variables. Assignment statements have similar rules.

The rules are as follows:

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

2.2.3 if-else statement: if is the start symbol, then identify the left bracket, conditional expression, right bracket, and then identify the specific execution statement, which can have an else statement or omit else.

The rules are as follows:

```
(<IF><LEFTPARENTHESES>Expr()<RIGHTPARENTHESES>Statement()[<ELSE>Statement()])+
```

2.2.4 while statement: while is the start symbol, and then identifies the left bracket, conditional expression, right bracket, and execution statement.

The rules are as follows:

```
<WHILE><LEFTPARENTHESES>Expr()<RIGHTPARENTHESES>Statement()
```

2.2.5 Conditional expression Expr():

The priority of conditional expressions from low to high is: defined in a nested manner, |, &&, >|<|>=|<=|==|, bracket expressions.

```
voidExpr():{}{Expr0()(<OR>Expr0())*}  
voidExpr0():{}{Condition()(<AND>Condition())*}
```

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

2.2.6 Expression(): Nested definition rules, the order of operation priority from low to high is + |-,*|/, operation item (factor) Factor(). Among them, Factor() is further refined into variables, bracket expressions, integer values, character values, and real values.

The rules are as follows:

```
voidExpression():{}{  
Term()((<PLUS>  
| <MINUS>)Term())* }  
  
voidTerm():{}{  
Factor()((<TIMES>  
| <DIVIDE>)Factor())* }  
  
voidFactor():{}{  
Identifier()  
| <LEFTPARENTHESES>Expression()<RIGHTPARENTHESES>  
| Integer()  
| Real()  
| String()  
}
```

2.2.7 Terminal symbol definition:

It is mainly defined for Integer(), Real(), and String() types to facilitate modification when errors occur.

The rules are as follows:

```

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

```

2.2.8 Structure definition

Member_list is a specific list of member variables.

```

/*define struct*/
StructNode defstruct(): { Token t; String n; List<Slot> membs; }
{ t = <STRUCT> n = name() membs = member_list(); }
/*System.out.println("Structure found!");*/
return new StructNode(location(t), new StructTypeRef(n), n, membs); //add later StructNode } } List<
Slot> member_list():
{ List<Slot> membs = new ArrayList<Slot>(); Slots; }
{ "({s=slot()},{ membs.add(s); })" } //Structure code block {
return membs; } }

Slots slot(): { TypeNode t; String n; }
t = type() n = name() { return new Slot(t, n); } //Structure members }

```

2.2.9 Definition of Return and Break

Whether the return value of LOOKAHEAD scan ahead is empty.

```

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); } /*Function has no return value
*/ | t = <RETURN> expr = expr() "; " { return new ReturnNode(location(t), expr); }
}

```

2.2.10 Function definition

Params() is a function syntax rule. LOOKAHEAD scans ahead to see if the function has a return value.

Fixedparams() is a function parameter list, and it scans ahead whether the function has multiple formal parameters.

Param() is the function body.

```

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

```

```

{LOOKAHEAD(<VOID>)"") t=<VOID>//No return value
{return newParams(location(t),newArrayList<CBCParameter>());}
| params=fixedparams() ["", "..."{params.acceptVarargs();}]
{return params;}
}

Paramsfixedparams():
{List<CBCParameter> params= newArrayList<CBCParameter>();
CBCParameterparam, param1;}
{param1=param() { params.add(param1); }
(LOOKAHEAD(2)", "param=param() { params.add(param); } ) *
{return newParams(param1.location(), params);}}

```

```

CBCParameterparam():
{TypeNode t;String n;}
{t=type()n=name() {return newCBCParameter(t, n); }//Formal parameters}
*/

BlockNodeblock():
{Tokent;List<DefinedVariable> vars;List<StmtNode> stmts;}

{t="{vars=defvar_list() stmts=stmts()}"//Function body code block{return new
BlockNode(location(t),vars,stmts);}}

```

2.2.11 Conflict handling

When the scanner is scanning, selection conflicts may occur. For example, when defining a variable and defining a function, the first two terminal symbols are the same. At this time, the scanner cannot determine whether to choose to define a function or a variable. You can use JavaCC's LOOKAHEAD to scan ahead. The rule of LOOKAHEAD is to add one to the number of the same terminal symbols. That is to say, when the left bracket or equal sign is recognized, the scanner will know whether it is a function definition or an integer definition. . Therefore, you can use LOOKAHEAD to look ahead three symbols and then make a judgment.

```

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

```

3. Word category definition

3.1 Definition of reserved words: It is defined in JAVACC using the <symbol: literal value> method.

```
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"> //replaceCin languageinclude
| <SIZEOF:"sizeof"> }
```

3.2 Variable definition

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

3.3 Definition of integer values

```
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 Definition of real numbers

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

3.5 Symbol definition

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

3.6 Definition of string literals and character literals

```
/*String literal*/  
MORE:{<"\" :IN_STRING><IN_STRING>MORE:{(<~["\"","\\","\\n","\\r"]>+> | <"\\"  
(["0"-"7"]){3}>  
| <"\"~[]>}  
<IN_STRING>TOKEN:{<STRING:"\"\"> :DEFAULT}  
  
/*character literal */  
MORE:{<""> :IN_CHARACTER} <  
IN_CHARACTER>MORE:{  
<~["\"","\\","\\n","\\r"]> :CHARACTER_TERM | <  
"\"(["0"-"7"]){3}> :CHARACTER_TERM | <"\"  
~[]> :CHARACTER_TERM  
><CHARACTER_TERM>TOKEN:{<CHARACTER:"\"> :DEFAULT}
```

3.7 Ignore space comments and other definitions

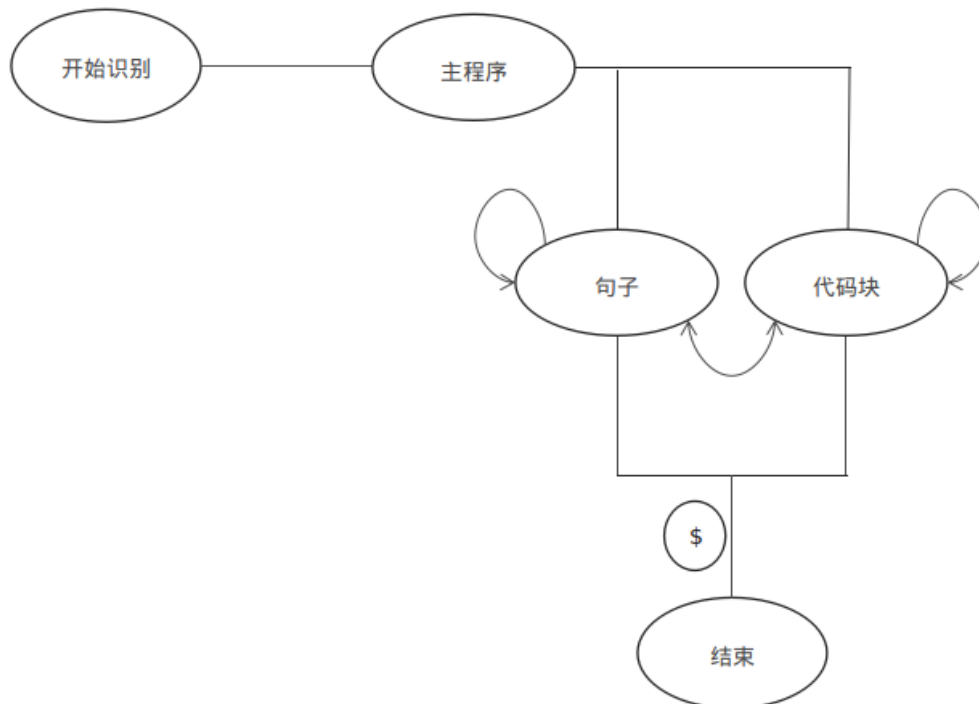
```
/*Ignore formatting symbols such as spaces, but save the record*/  
SPECIAL_TOKEN:{<SPACES:([" ","\\t","\\n","\\r","\\f"]>+>}  
  
/*Ignore comments while avoiding errors caused by longest match */
```

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

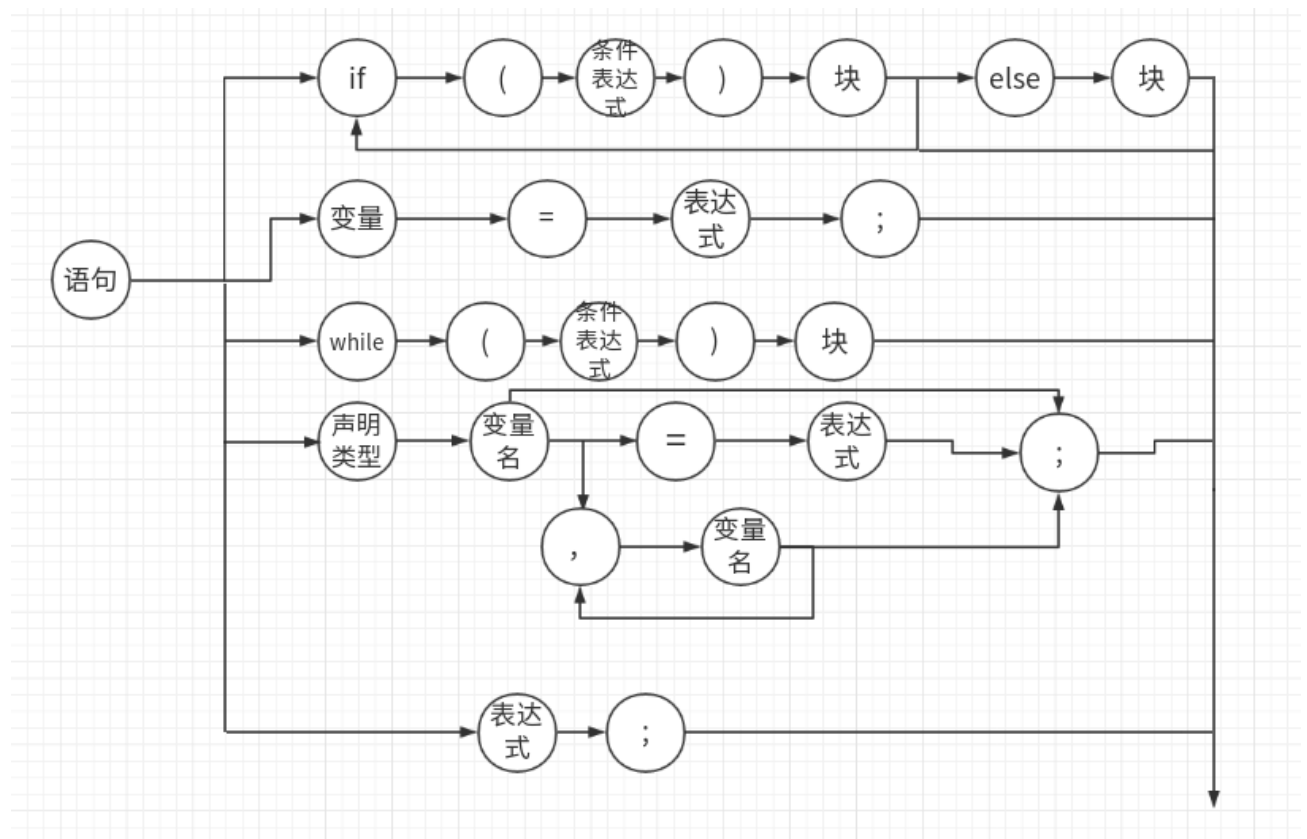
The definition has been modified in the semantic analysis part, but it is similar to the word category definition of lexical analysis, so it is not listed in this article.

4. Program flow chart

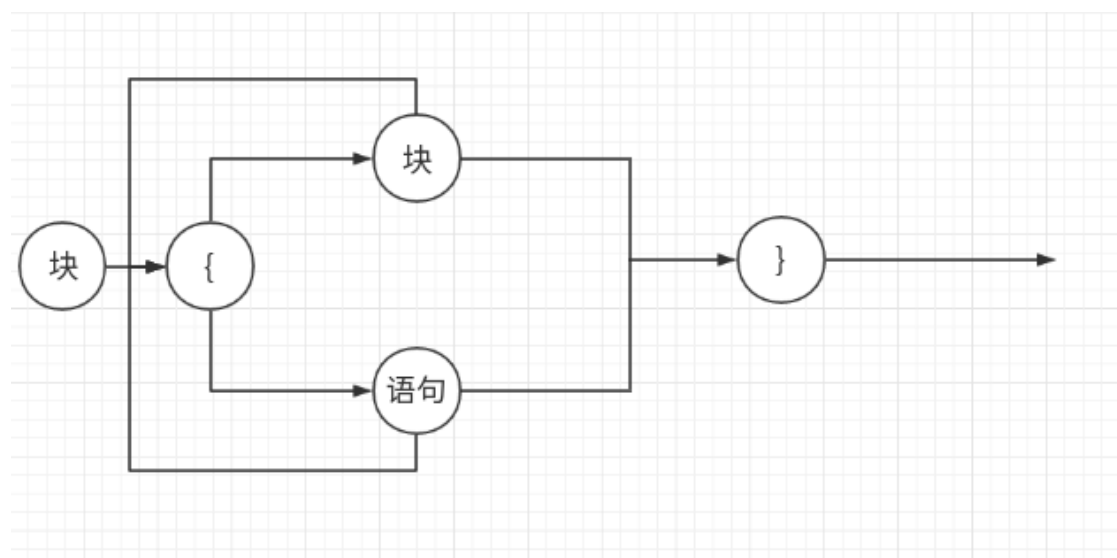
4.1 Main program



4.2 Statements

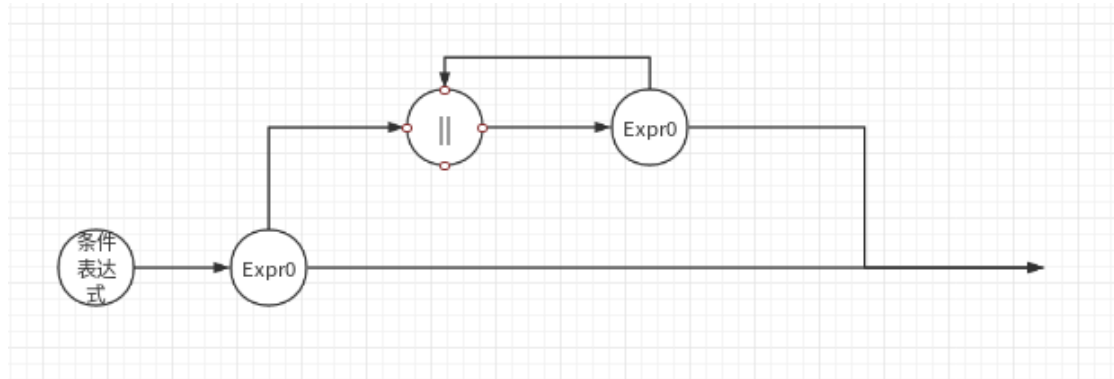


4.3 blocks

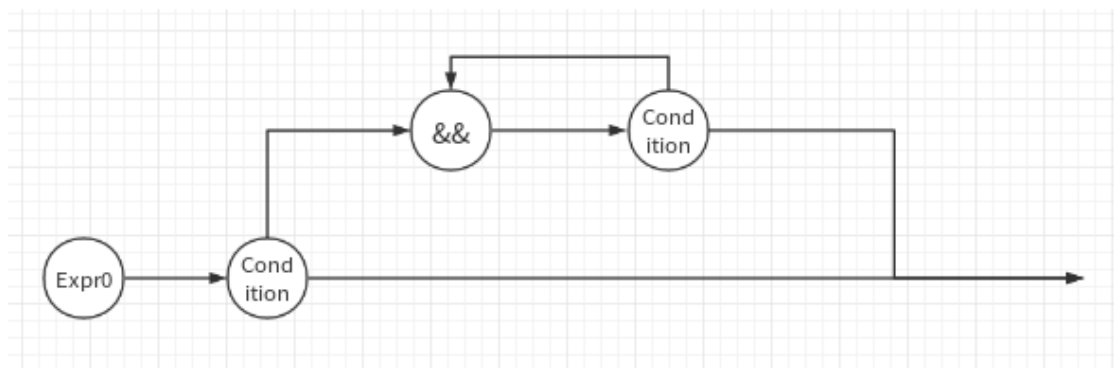


4.4 Conditional expression

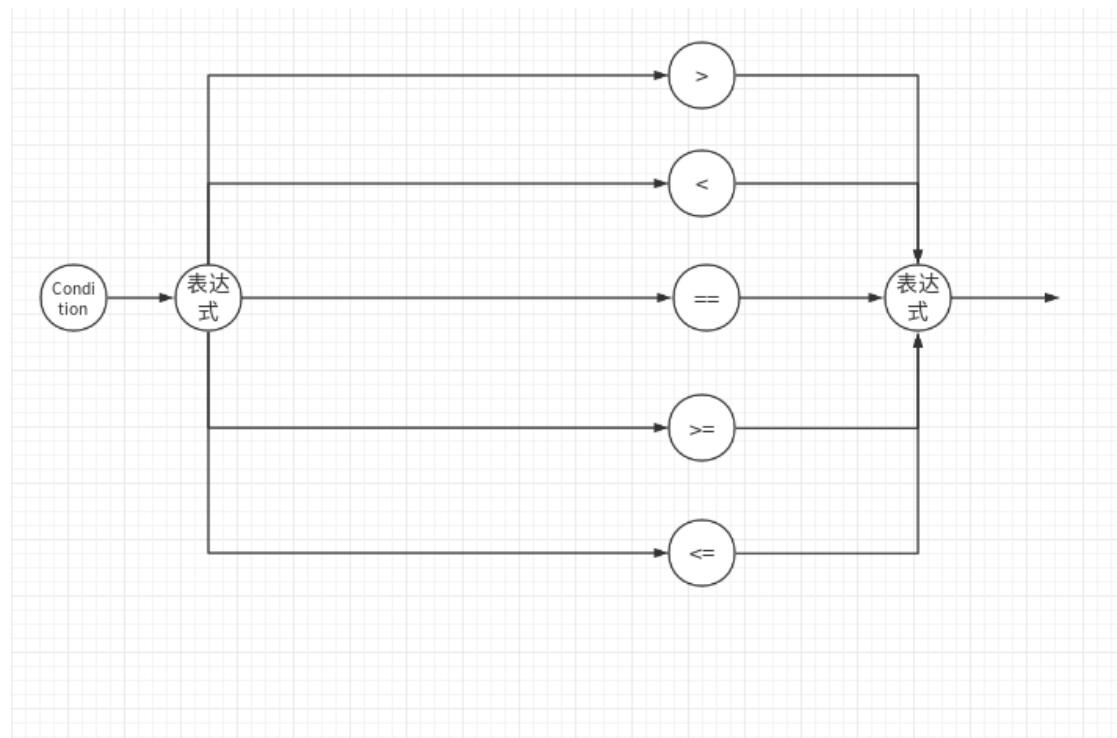
4.4.1 or



4.4.2 With

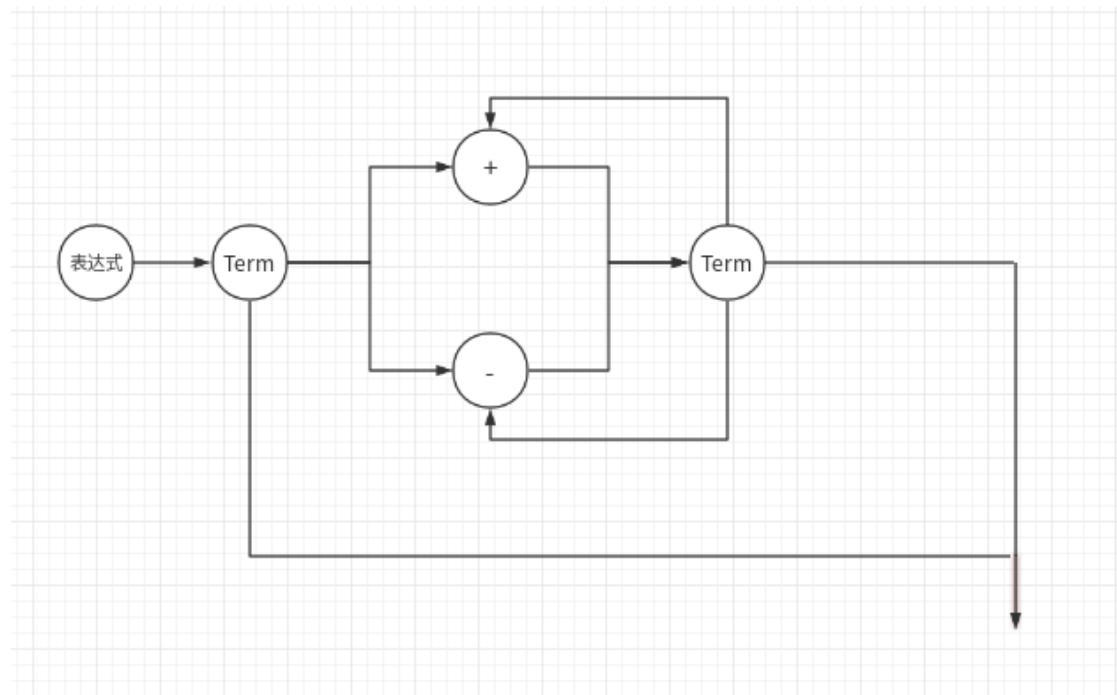


4.4.3 Comparison

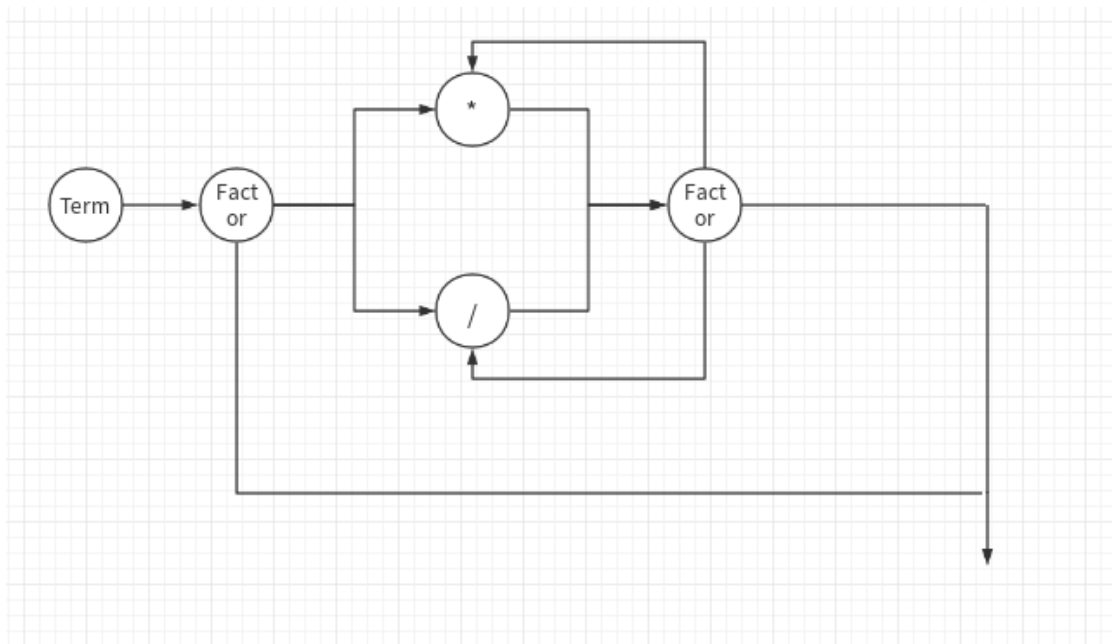


4.5 Expressions

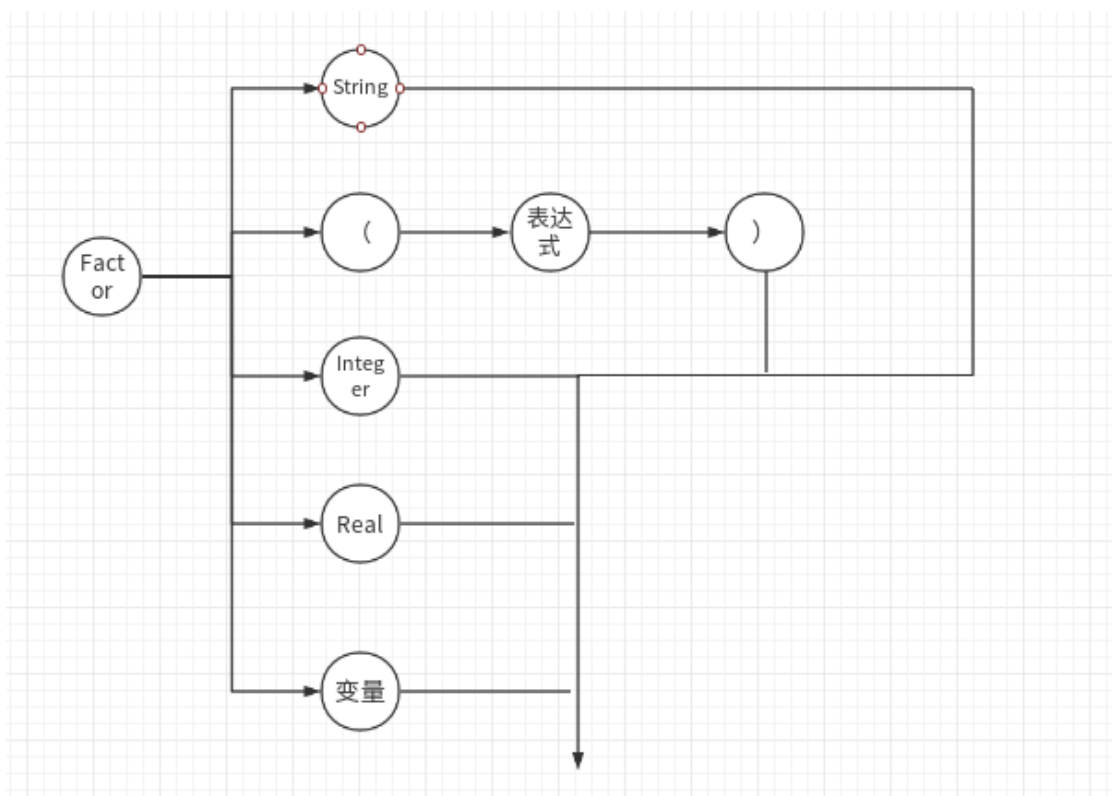
4.5.1 Addition and subtraction



4.5.2 Multiplication and division



4.6 items



5. Interpretation and analysis of program source code

5.1 JAVACC code structure description

5.1.1 Option{...} is used to set the properties of the JAVACC parser.

5.1.2 PARSER_BEGIN()

JAVA code;

PARSER_END()

It is a JAVA language parser, mainly used to execute the entire compiler program.

5.1.3 TOKEN:....

SKIP:....

Define the section for the scanner.

5.1.4 Start (can pass in parameters): {Define local variables that need to be used in the code block}

{

Identify and assign statements

{JAVA statements surrounded by curly braces can be executed directly}

}This part is a narrow parser

5.2 Lexical analysis

Args is the command line parameter, used to obtain the file passed in from the command line. The three catch statements are used to capture lexical analysis exceptions, input file not found exceptions, and input and output exceptions.

```

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());
        }
    }
}

```

The findInt function passes in the read file content, and then calls expr() to start lexical analysis.

```

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

```

The Image in the Token class attribute in JAVACC can get the Token literal value corresponding to the terminal symbol. Through x=terminal symbol, you can directly assign a value to the Token type variable, then create a new write buffer object, and write the terminal symbol and literal value to the file aaa.

If the second parameter of FileWriter is true, the file content can be written continuously.

When the scanner reads an integer, the JAVA statement within the curly braces is executed.

By calling the method shown in the figure below for all terminal symbols defined in the scanner, you can scan and record all <terminal symbol, literal value> tuples.

```

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 Syntax analysis

5.3.1 JJTree Introduction

JJTree is a partner tool of JavaCC. JavaCC does not directly generate parse trees or abstract syntax trees (AST), but provides JJTree, a preprocessor for building parse trees or AST generation. JJTree uses a recursive method of pushing and popping the stack to generate parsing trees, which is the basis for JavaCC. The input is preprocessed. By calling the dump() method of the SimpleNode class unique to JJTree, you can print the syntax tree of the statement.

5.3.2 Main code interpretation

Create a new c1JJTree object, enter data from the keyboard, then create a new SimpleNode class and initialize it with the start node of the program. Then call the dump() method of SimpleNode to print the syntax tree. The catch statement is used to check whether there are syntax errors in the C language subset statements entered by the user, and prompts the location of the error and the method of modifying the error.

```

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());
}
}
}

```

The starting node of the syntax analysis program. The user inputs "\$" to indicate the end of the input, and then Start returns a SimpleNode object jjtThis. JJTree will output the abstract syntax tree according to the nested structure shown in (5. Program flow chart).

```

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

```

5.4 Semantic analysis and semantic action execution

5.4.1 Variable definition and assignment

Var is the variable class, which is divided into double, int, and String types. getx is to get the variable literal value, and setx is to assign the variable value.

```

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){

```

Define the variable function, var newv is used to store the newly defined variable object, first create a new variable object, varname receives the variable name returned by Identifier(), and then initializes the variable newv. Double x receives the return value of the expression. If a variable is assigned a value, the new and setx() functions are called to store the assigned value. Then print out the variable name and variable value of the newv object.

```

/*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
}

```


After the scanner scans the variable, it assigns it to the Token type variable and returns the String type variable name (n.image literal value).

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

5.4.2 while loop statement

Since there is little information about JavaCC on the Internet, I haven't figured out how to implement the semantic actions of the while statement. The current idea is to define the return value of statement() as a class, including the specific operations of statement(). The return value of the conditional expression Condition() is a class, including the lvalue/conditional symbol/rvalue/condition of the conditional expression is true or false, and then use java's loop statement to loop according to the conditions and statements.

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

5.4.3 if-else statement

The code in the comment part is for the case where only a single if statement is entered without an else statement.

The return value of the conditional expression Expr() is 1, which means the condition is true, and the return value is 1, which means the condition is false.

Statement() returns the value after the statement block is executed.

Here I take the if-else statement as an example. t receives the return value of the statement immediately following if, and f receives the return value of the else statement. The if statement of JAVA executed within the curly braces selects which branch result to output based on the true or false value of the Condi condition value.

```

|(<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;
    }
}
]

)+

```

Conditional expressionExpr()

First, analyze the Condition() function at the lowest level. The condi of Token type is used to store the scanned conditional operator, the L of double type receives the lvalue of the conditional operator, and R receives the rvalue of the conditional operator.

```

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

```

The curly braces are for executing JAVA statements. According to the value of condi, the corresponding if statement is executed and returned to Expr0() as a conditional operation value of int type. 1 is true and 0 is false.

```

{
    //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 is the lvalue of the AND operation, y is the rvalue, which is used to obtain the logical value returned by the conditional expression. JAVA operations are performed within the curly braces. According to the rules of the AND operation, I use multiplication here instead of the AND operation, because when there is When a value is 0, the conditional expression evaluates to 0 and returns 0 immediately. If it is not 0, since it is multiplication, just return the literal value of x directly.

```
//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 is the lvalue of the OR operation, y is the rvalue, used to obtain the logical value returned by the conditional expression. JAVA operations are performed within the curly braces. According to the rules of the OR operation, here I use addition instead of the OR operation, because only all values When all are 0, the value of the conditional expression is 0. Since it is an addition operation, x can be returned. If it is not 0, it returns 1 directly.

```

//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 Expressions

When the scan statement is an expression, the calculated expression value of double type is returned.

```

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

```

Result() receives the calculated value of an expression and returns a double type value.

```

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

```

E

return

Execute

The operator lvalue returned by rm(), when y receives Term(), the curly braces perform addition operations, and when subtraction is recognized, After the row is complete, return returns the calculated x value.

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

Term() performs addition and subtraction operations, x receives the operator lvalue returned by Facor(), and y receives the operator rvalue returned by Facor(). When the multiplication symbol is recognized, the curly braces perform the multiplication operation. When the division operation is recognized, the division operation is performed. When all the multiplication and division operations are completed, return returns the calculated x value.

```

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

```

The Factor() function returns a value of type int when an integer value is scanned, and a double type when a real number type value is scanned.

```

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

```

Integer type and real number type are divided into two situations. One is to scan to x, store the integer (real number) type value, and return the integer (real number) type literal value. Another case is when an expression is recognized, sum obtains the return value of the expression (integer or real number), and returns sum to the Factor() of the previous layer.

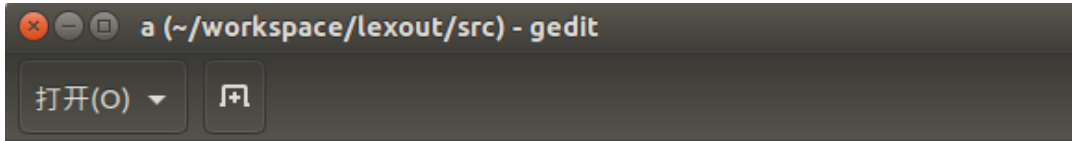
```
double Integer():
{
    Token x;
    double sum;
}
{
    x=<INTEGER_LITERAL>
    {
        return Integer.parseInt(x.image);
    }
    |
    <LEFTPARENTHESES>sum=Expression()<RIGHTPARENTHESES>
    {
        return sum;
    }
}
/*实数*/
double Real():
{
    Token x;
    double sum;
}
{
    x=<REAL_LITERAL>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


6. Program results

6.1 Lexical analysis

6.1.1 Correct operation

Program input: The file content is as follows.



```
/*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;

}
/**/
```

To run, first enter the workspace where the c1Lex.jj file is located, and use the terminal to open the folder.

Compile the .jj file through the Javacc c1Lex.jj command.

```

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.

```

After compilation, the relevant java files are generated.

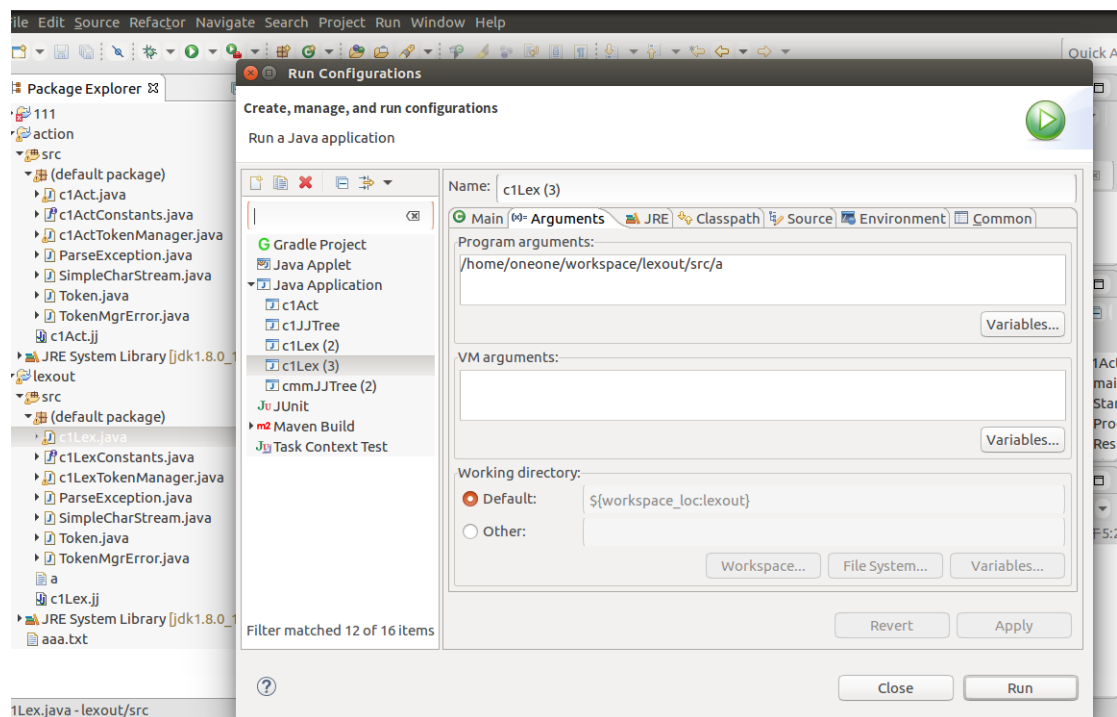
```

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

```

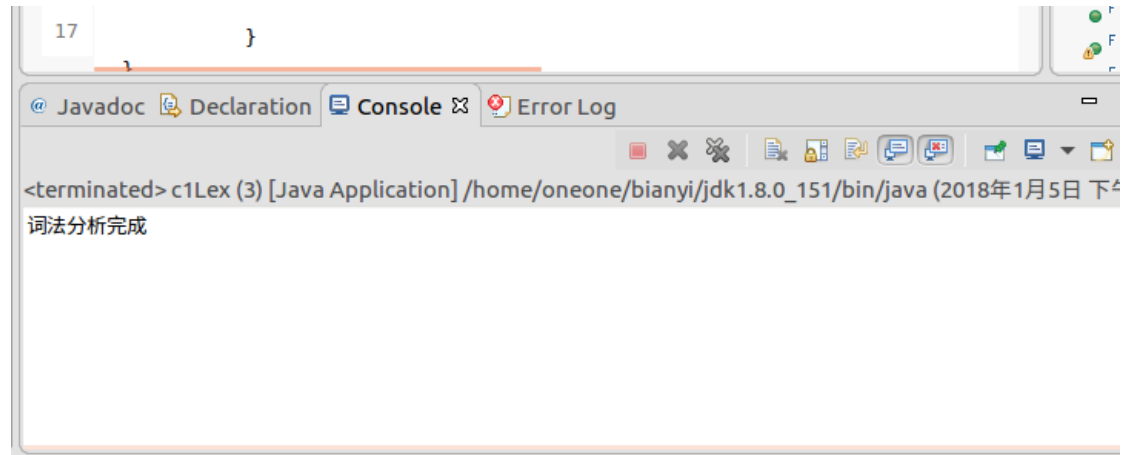
Then use `javac *.java` to generate the executable `c1Lex.class` class.

Since it is more convenient to run with Eclipse, I refresh the workspace, compile and run `c1Lex.java`. Since data is read from a file, before running, you need to click the arguments of the Run configuration to set the absolute path of the file to be read, as shown in the figure below. What I am reading here is a file in the workspace.

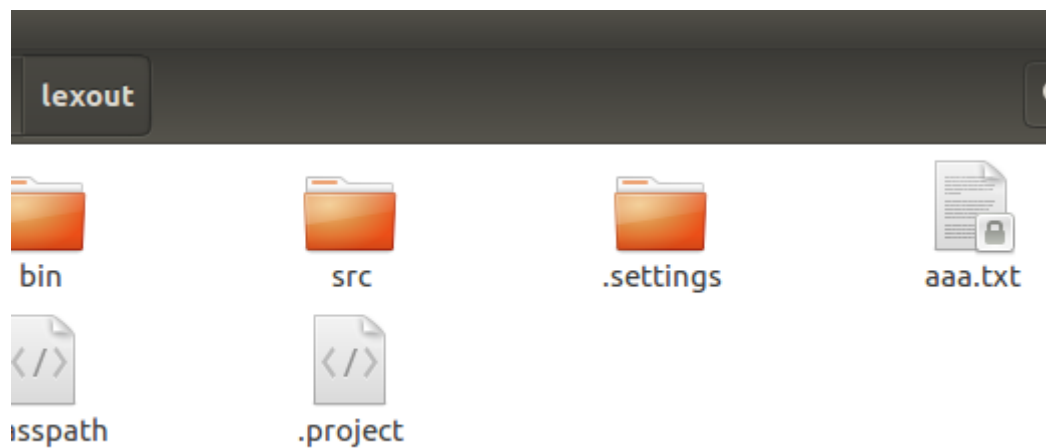


for
out

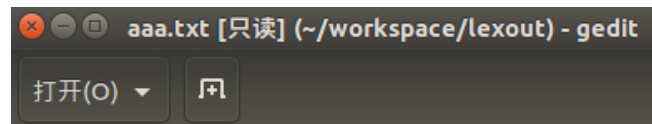
transport



Then we go to the workspace directory and find the aaa file.



Open it and you can see that the output result is a tuple of the defined word and the literal value of the corresponding source program code.



```
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 }

```

We can see in the results that the scanner skips content such as white space comments, while avoiding errors where code between two comments is also ignored due to the longest match.

6.1.2 Example of error reporting during incorrect operation

(1) In the input file, I modified the content into unclosed comments.

```

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

```

An error is reported after execution.

```

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) In the input file, I defined the identifier to start with a number (not allowed in C language, an error will be reported).

```

/*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 Syntax analysis

In order to facilitate debugging and observation of results, the input and output of syntax analysis are placed on the screen. If it is changed to file input and output, just modify it according to the code of lexical analysis.

In the syntax analysis part, I used the JJTree that comes with JavaCC, which can automatically generate the corresponding syntax tree based on the syntax rules we defined.

The specific running method is to first use jtree in the command line c1Jtree.jjt statement compilation jtree file, and then the corresponding c1Jtree.jj will be generated. The next operation method is the same as the JavaCC file.

(1) Enter the variable definition statement, output the syntax tree of the corresponding structure, and display a prompt for correct syntax.

```
***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
```

If an incorrect C language statement is entered, the error location will be prompted and how to modify the C language statement will be given. For example, I want to declare a variable but forget to give the variable name.

```
***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) Enter the expression statement.

```

***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;
$

```

The generated syntax tree is shown in the figure below.

```

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

```

If the input expression is incorrect, an error will be reported. For example I am evaluating the fit of an expression without closing parentheses.

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) Enter the while statement that conforms to the grammatical rules.

```
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

If you enter an incorrect while statement, for example, a common mistake we often make when writing C language code is to accidentally write == as = assignment in the conditional expression, an error will appear as shown in the figure below.

```
***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) Enter the assignment statement.

```
***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) Enter the if-else statement.

```
***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

```

Similarly, if an equal sign appears in the conditional expression of an if statement, an error will be reported.

```

***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:
    "+" ...
    "-" ...
    "*" ...
    "/" ...
    "==" ...
    "<>" ...
    "<" ...
    ">" ...
    ">=" ...
    "<=" ...

```

If after writing the If statement, the corresponding execution code is not written, an error will also be reported.

(6) If you forget to add a semicolon when writing code, you will also get an error.

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) Combine all C statements into complete C language code, and the results obtained are as follows.

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

(1) Recognize complex expressions and calculate the value of the expression.

```
***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) Identify the if statement and determine which branch to execute based on the result of the conditional expression.

In order to facilitate the observation of the results, I directly entered the displayed numerical value in the conditional expression. You can see the result display. Since the value of the expression is false, the else branch is executed and the calculation result of the expression is output.

```
***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
```

After changing the condition to less than, the conditional expression is true, and the branch immediately following if is executed.

```
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) Identify the while statement. Since the information about JavaCC on the Internet is too scarce, I have not yet figured out how to write the execution action of a while statement. Therefore, there is no specific implementation of the while loop here. It just prints out the prompt that the while statement is recognized. The loop function will be implemented later..

```
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) Define variables and assignment statements. When defining a variable, a new variable class var is created, and the name of the newly created variable and the assigned literal value are stored in the class. Variable definitions are divided into four types: int, real, string, and char. The following figure shows a demonstration of defining the int type. Other types are similar. A separate assignment statement is similar to variable assignment and will not be demonstrated again.

```
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. Existing problems and solutions

Engineering problem: Since there is too little information on the combined compilation of JAVACC and ECLIPSE on the Internet, I don't know why the JAVACC plug-in does not have the permission to compile code in ECLIPSE. I still don't know how to set it up, so I can only compile it through the command line and run it in ECLIPSE.

When using the JAVA command directly in the command line, an error message appears: The class cannot be loaded. Since the online solutions are invalid and I am not familiar with the LINUX system, the reason has not yet been found.

You can only use ECLIPSE to run files generated by JAVACC. At the same time, please note that if there is no Package declared in the .jj file, it cannot be opened in the ECLIPSE package and an error will be reported. The .java file must be generated directly in the SRC folder.

7.1 Lexical analyzer

7.1.1 The macro definition method should be used when writing text files, otherwise it will be difficult to modify.

7.1.2 STRING recognition replaces string recognition. Characters can be specified as single quotes, or with some special identification marks.

7.1.3 During the scanning process, the same identifier is recognized multiple times. Is it possible to avoid repeating it? Just scan it once and save it. There is no need to write it again.

7.1.4 The code for identifying function `expr()` can be replaced by a function to reduce the amount of code. Take the identification pointer and the literal value of the identification unit as parameters of the function.

7.1.5 When an illegal identifier is detected, such as an identifier starting with a number, it can only prompt the location of the error, but cannot print the reason for the error (such as the sentence "illegal identifier").

7.1.6 According to the book "Homemade Compiler", you can use `parserConstants.tokenImage[token.kind]` to get the "type" constant of token. However, after I added this syntax to the code, I got an error: There is no `parserConstants` class. Since there is indeed no relevant information on the `parserConstants` class online, it remains to be studied.

7.2 Parser

7.2.1 If all operation symbols (conditional expressions and arithmetic expressions) are nested together according to priority, ambiguity will occur, which needs to be resolved by using the LOOKAHEAD function of JavaCC.

7.2.2 Using `()*` in JJTREE will report an error -> can be an empty string. I checked online but there is very little information, so I haven't found a solution yet. I can only use `()+` in all statements. The disadvantage of this definition is that it cannot accept empty statements and must enter the defined statement, otherwise an error will be reported.

For the loop statement `while()`, the rules for nested loop statements have not been implemented.

7.3 Semantic analysis program

7.3.1 If else and separate If statements need to be distinguished when recognizing sentences. I have not distinguished them in the code. I can only recognize if-else. If I do not distinguish and directly add actions for separate If statements, it will be repeated. Go through the semantic actions of if twice and then report an error.

7.3.2 `statement()` must have a return value, and the return value needs to be defined in many types. At present, I only define the return value as double, which only works on numeric type variables. This results in when the executed code block is a statement When defining a variable statement in `()`, an error will be reported after the newly created variable is correctly output. If a return value is added, the return value has no meaning.

The solution I am currently thinking of is to define the return type of the starting statement `statement()` as a class, and then gradually add various variable types to the class. Since the remaining time is short, only statements valid for numeric types are implemented.

7.3.3 In the semantic analysis, I did not implement the assignment statement separately. I implemented the assignment statement directly in the variable definition. I only took defining one variable and assigning values as an example. The code for defining and assigning multiple variables is similar.

7.3.4 Implementation of WHILE semantic action: define the return value of statement() as a class, including the specific operations of statement(). The return value of the conditional expression Condition() is a class, including the lvalue/conditional symbol/rvalue/condition of the conditional expression is true or false, and then use java's loop statement to loop according to the conditions and statements.

7.3.5 Regarding the final result of semantic analysis, the quadruple function of the print statement is not implemented. The function I implemented is to actually execute the semantic actions through JAVA statements. The meaning of the four-tuple is the execution order of the statement and the storage of the value. I have already implemented this part when implementing the semantic action, because the priority and nesting structure are determined when the grammar rules are defined, and each item is identified. The value of the statement is stored and returned to the previous level. If the quadruple is compared to a formula, then what I achieve is to use the formula to solve the problem and get the result. However, the requirement of this course is to print quadruples. Outputting quadruples is simpler than implementing semantic execution actions, which I will implement in the future when making a C language compiler.

References

"Homemade Compiler" [Japanese] Aoki Minero

JAVACC official HTML documentation

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

<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 JavaCC research and application