编译原理研讨课 PR001 实验报告

小组成员：

杜旭蕾 2022K8009929003　王锦如 2022K8009929022

一. 实验任务

1.熟悉 ANTLR 的安装和使用：

了解 ANTLR 工具生成词法-语法源码的能力。

掌握 ANTLR 生成 lexer 和 parser 的流程。

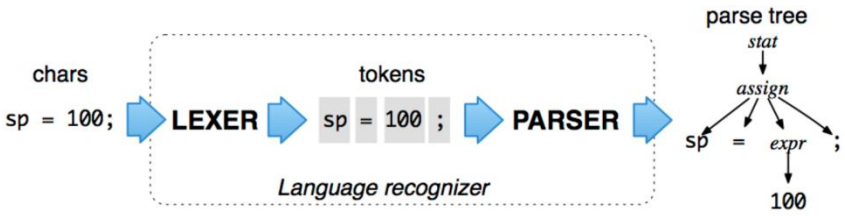
2.完成词法和语法分析：

根据 CACT 文法规范编写 ANTLR 文法文件(.g4)，并通过 Antlr 生成 CACT 源码的词法-语法分析。

覆写 ANTLR 默认的文法错误处理机制，能检查出源码中的词法语法错误。

二. 设计思路

1. Hello.g4



如图，Hello.g4 主要分成 Lexer 和 Parser 两部分。Lexer 的主要作用是将输入转化为 token，位于 Hello.g4 的后半段；Parser 的主要作用是将 token 转化为语法树，位于 Hello.g4 的前半段。Hello.g4 的代码主要根据 cact25-spec.pdf 给出的规则经过修改和 debug 即可。

2. main 文件

main 文件主要由 main 函数和 Analysis 函数构成，main 函数通过读取出入的 cact 文件产生语法树并调用 Analysis 函数对每一个节点进行遍历，查找错误节点并输出，如果没有错误节点则返回 0，有错误节点则返回 1。这一部分代码在原有的 main 函数基础上通过使用 ANTLR 提供的遍历器遍历语法树，然后调用 ANTLR 提供的 ParseTreeListener 接口监听语法树遍历事件即可。

三. 代码实现

1. Lexer 实现

Lexer 的主要作用是将输入转化为 token。由于 ANTLR4 在词法分析时，使用最长匹配优先和规则定义顺序优先的原则，因此首先需要对讲义中给出的 12 个关键字进行识别匹配，这可以使得这些关键字不被识别为 Ident，具体代码如下。

```
/****** lexer  ******/
/*Key Word*/
CONST_KW : 'const' ;
INT_KW : 'int' ;
DOUBLE_KW : 'double' ;
CHAR_KW : 'char' ;
FLOAT_KW : 'float' ;
VOID_KW : 'void' ;
IF_KW : 'if' ;
ELSE_KW : 'else' ;
WHILE_KW : 'while' ;
BREAK_KW : 'break' ;
CONTINUE_KW : 'continue' ;
RETURN_KW : 'return' ;
```

除了关键字的识别还需要对运算符和逗号、分号、小数点和大括号进行匹配，其中由于规则定义顺序优先的原则，运算符需要按照文件给出的优先级从上到下排序，代码如下。

```
/*operator*/
LeftBracket : '[' ;
RightBracket : ']' ;
LeftParen : '(' ;
RightParen : ')' ;
Plus : '+' ;
Minus : '-' ;
Not : '!' ;
Star : '*' ;
Div : '/' ;
Mod : '%';
LessEqual : '<=' ;
Less : '<' ;
GreaterEqual : '>=' ;
Greater : '>' ;
Equal : '==' ;
NotEqual : '!=' ;
AndAnd : '&&' ;
OrOr : '||' ;
Semi : ';' ;
Comma : ',' ;
Assign : '=' ;
Dot : '.' ;
```

```
LeftBrace : '{' ;
RightBrace : '}' ;
```

在这之后进行终结符的匹配，这些都需要放在关键字的识别之后来确保关键字不会被识别为 Ident。其中，相比于文件中给出的终结符匹配，我们化简了十进制、八进制和十六进制的常量写法，只用一行实现这些常量的匹配，具体代码如下。在这里中括号加星号表示可以匹配 0 个或多个方括号内包括的字符，而加号则表示至少要匹配一个方括号内的内容。因此在这部分形如'0x'这样的十六进制数是不合法的。

```
/*** 终结符 ***/
Ident: [a-zA-Z_] [a-zA-Z_0-9]*;
IntConst : DecimalConst | OctalConst | HexadecConst ;
HexadecConst : ('0x'|'0X') [0-9a-fA-F]+ ;
OctalConst : '0' [0-7]+ ;
DecimalConst : '0' | [1-9] [0-9]*;
```

有关 FloatConst 的识别文件中没有给出，参考 ISO/IEC 9899 文档的以下内容:

floating-constant:
decimal-floating-constant
hexadecimal-floating-constant

decimal-floating-constant:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

hexadecimal-floating-constant:
hexadecimal-prefix hexadecimal-fractional-constant
binary-exponent-part floating-suffix$_{opt}$
hexadecimal-prefix hexadecimal-digit-sequence
binary-exponent-part floating-suffix$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ **.** *digit-sequence*
digit-sequence **.**

exponent-part:
e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

sign: one of
+ **-**

digit-sequence:
digit
digit-sequence digit

hexadecimal-fractional-constant:
hexadecimal-digit-sequence$_{opt}$ **.**
hexadecimal-digit-sequence
hexadecimal-digit-sequence **.**

binary-exponent-part:
p *sign*$_{opt}$ *digit-sequence*
P *sign*$_{opt}$ *digit-sequence*

hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit

floating-suffix: one of
f **l** **F** **L**

从以上内容中去掉十六进制的部分即可转化为以下代码。其中，对浮点数的匹配分为两种，可以是小数部分+指数部分（可选）+后缀（可选），或者整数部分+指数部分（必须）+后缀（可选）。以下代码中'?'就表示了可以没有或只能有一个。小数部分小数点之前可以有数字，也可以没有数字。指数部分可以以 E 或 e 开头，可以有指数的符号。浮点数的后缀是 f 或 F。

```
FloatConst
    : FractionalConstant ExponentPart? FloatingSuffix?
    | DigitSequence ExponentPart FloatingSuffix?
    ;

//分数部分
fragment FractionalConstant
    : DigitSequence? '.' DigitSequence
    | DigitSequence '.'
```

```
    ;
// 指数部分
fragment ExponentPart : [eE] Sign? DigitSequence ;
// 符号
fragment Sign : [+-] ;
// 数字序列
fragment DigitSequence : [0-9]+ ;
// 浮点数后缀
fragment FloatingSuffix : [fF] ;
```

需要跳过的内容需要放在 Lexer 的最后，主要包括换行符、空白符、单行注释和多行注释，如果遇到这些字符编译器会直接跳过。其实现与 cact 文件中给出的基本一致，具体代码如下。

```
/****** skips  ******/
NewLine :( '\r' '\n'?| '\n' ) -> skip ;
WhiteSpace : [ \t\r\n]+ -> skip;
LineComment : ('//' ~[\r\n]* ) -> skip;
BlockComment :('/*' .*? '*/') -> skip;
```

2. Parser 实现

Praser 的主要作用是匹配已经分割好的token 组成 parse tree。其主要分为声明和定义、以及语句和表达式两部分。在声明和定义部分需要将 cact 文件中给出的规则的非终结符改为小写开头，还需要将规则中带单引号的部分换成已经分割好的 token name 即可，对应代码如下。

```
/****** parser ******/
/*** 声明和定义 ***/
//所有{}换成()*,因为 g4 不支持，任意次（可能 0 次，也可能多次）
//所有[]换成()?，因为 g4 不支持，整个括号内容是可选的（0 次或 1 次）

compUnit : ( decl | funcDef )+ EOF ;

decl : constdecl | vardecl ;

constdecl : CONST_KW bType constDef ( Comma constDef )* Semi ;

bType : INT_KW | FLOAT_KW | CHAR_KW;

constDef : Ident ( LeftBracket IntConst RightBracket )* Assign constInitVal ;

constInitVal : constExp | LeftBrace ( constInitVal ( Comma constInitVal )* )? RightBrace ;
/*
vardecl : bType varDef ( Comma varDef )* Semi ;
varDef : Ident ( LeftBracket IntConst RightBracket )* ( Assign constInitVal )?;
//以上内容转化为以下部分
*/
```

```
//尝试解决数组初始化问题，询问助教后发现也可以不改
//解决 int a[4] = 4;非法
vardecl : bType (varDef | varDefArray) ( Comma (varDef | varDefArray) )* Semi ;
varDefArray : Ident LeftBracket IntConst RightBracket ( LeftBracket IntConst RightBracket )* ( Assign
constInitValArray )?;
constInitValArray : LeftBrace ( constInitVal ( Comma constInitVal )* )? RightBrace ;
varDef : Ident ( Assign constInitVal )?;

funcDef : funcType Ident LeftParen (funcFParams)? RightParen block ;

funcType : VOID_KW | INT_KW | FLOAT_KW | CHAR_KW ;

funcFParams : funcFParam? ( Comma funcFParam )* ;

funcFParam : bType Ident ( LeftBracket (IntConst)? RightBracket ( LeftBracket IntConst
RightBracket )* )? ;
```

语句和表达式部分也需要将 cact 文档中的规则进行类似处理，其中 constExp 与 cact 文档中 constExp 的定义存在区别，写在 debug 部分。

```
/*** 语句和表达式 ***/
block : LeftBrace (blockItem)* RightBrace ;

blockItem : decl | stmt ;

stmt :  lVal Assign exp Semi
       | ( exp )? Semi
       | block
       | RETURN_KW exp? Semi
       | IF_KW LeftParen cond RightParen stmt ( ELSE_KW stmt )?
       | WHILE_KW LeftParen cond RightParen stmt
       | BREAK_KW Semi
       | CONTINUE_KW Semi ;

exp : addExp ;

//constExp : addExp ;// 使用的 Ident 必须是常量
//显式初始化：CACT 限制初值表达式必须是常数
constExp : number ;

cond : lOrExp ;

lVal : Ident (LeftBracket exp RightBracket)* ;

number : IntConst | FloatConst ;

funcRParams : exp (Comma exp)* ;

primaryExp : LeftParen exp RightParen | lVal | number ;

unaryExp : primaryExp | (Plus | Minus | Not) unaryExp
```

```
          | Ident LeftParen ( funcRParams )? RightParen ; //'!'仅出现在条件表达式中

mulExp : unaryExp | mulExp (Star | Div | Mod) unaryExp ;

addExp : mulExp | addExp (Plus|Minus) mulExp ;

relExp : addExp | relExp (Less | Greater | LessEqual | GreaterEqual) addExp ;

eqExp : relExp | eqExp (Equal | NotEqual) relExp ;

lAndExp : eqExp | lAndExp AndAnd eqExp ;

lOrExp : lAndExp | lOrExp OrOr lAndExp ;

/*
1. 常量表达式不在语句块中出现。
2. 条件表达式只在 If 和 While 的条件语句中出现，不出现在赋值语句中。
3. 连加（a+b+c）会出现，但诸如连等(a=b=c)、连续比较(a==b==c，a < b < c)则不会出现，不必考
虑。
*/
```

3. main 文件实现

main 文件的主要作用是读取输入的 cact 文件，产生语法树并检验其是否正确，如果正确则返回 0，有错误则返回 1。

对于 main 函数，在原本的基础上增加对输入的读取使其能够处理不同的 cact 文件。在解析输入之后，调用 praser 解析的 compUnit 规则，返回这个顶层规则之下的语法树 tree。主机后将这个树结构打印出来。

在这之后创建 Analysis 监听器，通过 ANTLR 提供的默认遍历器 tree::ParseTreeWalker::DEFAULT 遍历 tree 语法书，对每一个节点调用 listener 进行处理。如果输入的 cact 文件是合法的则向 output.txt 中写入 0，方便批量处理 test 文件。具体代码实现如下。

```
int main(int argc, const char* argv[]) {
    if (argc != 2) {
        return -1;
    }
  std::ifstream stream;
  stream.open(argv[1]);

  ANTLRInputStream   input(stream);
  HelloLexer        lexer(&input);
  CommonTokenStream  tokens(&lexer);
  HelloParser       parser(&tokens);

  tree::ParseTree *tree = parser.compUnit();
  std::cout << "-------" << std::endl;
```

```
  std::cout << tree->toStringTree(&parser) << std::endl;
  Analysis listener;
  tree::ParseTreeWalker::DEFAULT.walk(&listener, tree);
  std::cout << "0" << std::endl;

  std::ofstream outfile("output.txt", std::ios::app);
    if (outfile.is_open()) {
        outfile << "0" << std::endl;
        outfile.close();
    } else {
        std::cerr << "无法打开 output.txt 文件" << std::endl;
        return -1;
    }
  return 0;
}
```

Analysis 监听器的代码如下，其继承 ParseTreeListener,用于监听语法树遍历事件。代码在进入任何语法规则、访问终结符（token）、退出语法规则时不做任何处理，如果解析时遇到错误节点则向 output.txt 中写 1 并退出程序返回 1，具体代码如下。

```
class Analysis : public antlr4::tree::ParseTreeListener {
public:
    virtual void enterEveryRule(ParserRuleContext* ctx) override {}
    virtual void visitTerminal(tree::TerminalNode* node) override {}
    virtual void visitErrorNode(tree::ErrorNode* node) override {
        std::cout << "1" << std::endl;
        std::ofstream outfile("output.txt", std::ios::app);
        if (outfile.is_open()) {
            outfile << "1" << std::endl;
            outfile.close();
        } else {
            std::cerr << "无法打开 output.txt 文件" << std::endl;
        }
        exit(1);}
    virtual void exitEveryRule(ParserRuleContext* ctx) override {}
};
```

4. 运行脚本实现

本次实验我们写了两个运行脚本，build.sh 用来在修改 Hello.g4 文件之后编译，batch_run.sh 用来在编译之后批量运行 test 文件。

build.sh 主要将讲义给出的编译代码整合使其可以一起运行，具体代码如下。

```
#!/bin/bash

# 设置错误处理，遇到错误时终止执行
set -e

# 进入语法文件目录
```

```
echo "切换到 grammar 目录..."
cd ../grammar/

# 运行 ANTLR 生成 C++ 代码
echo "运行 ANTLR 生成 C++ 代码..."
java -jar ../deps/antlr-4.13.1-complete.jar -Dlanguage=Cpp -no-listener -visitor ./Hello.g4

# 切换到 build 目录
echo "切换到 build 目录..."
cd ../build/

# 运行 CMake 配置项目
echo "运行 CMake..."
cmake ..

# 编译项目
echo "开始编译..."
make -j$(nproc)

echo "编译完成! "
```

batch_run.sh 首先先清空输出文件 output.txt（用于存放每一个 test 的正确性，0 为正确 1 为错误），然后从 file_list.txt 中读取全部需要运行的文件名并批量的运行，具体代码如下。

```
#!/bin/bash

output_file="output.txt"

# 确保文件存在
touch "$output_file" 2>/dev/null || {
    echo "无法创建/访问输出文件: $output_file"
    exit 1
}

# 清空文件
> "$output_file"

while read filename
do
  echo "正在处理文件: $filename"
  ./compiler "../test/samples_lex_and_syntax/$filename"
done < file_list.txt
```

四. Debug 过程

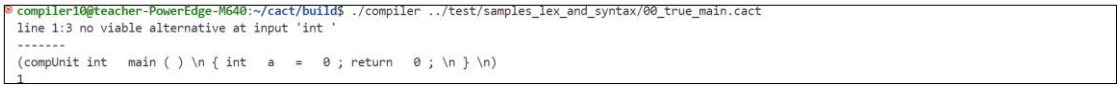
1. 规则首字母大小写问题

由于一开始写的时候直接复制的 cact25-spec.pdf 中的内容，没有将 Parser 中的非终结

符改为小写导致无法正确运行，首字母修改为小写即可解决。

2. Lexer 解析空格问题

在测试 00_true_main.cact 是遇到如下问题，Lexer 在解析时没有跳过空格，将'int'解析为了'int '（int+空格）。



查看 Lexer 代码发现将 Blank 等加入了规则中，因此使得在生成 token 的过程中无法删除空格，去掉前面的这部分问题解决。

```
SlashR : '\r' ;
SlashN : '\n' ;
SlashT : '\t' ;
Blank : ' ' ;
Remark : '//' ;
LeftRemark : '/*' ;
RightRemark : '*/' ;

…

/****** skips  ******/
NewLine :( '\r' '\n'?| '\n' ) -> skip ;
WhiteSpace : [ \t\r\n]+ -> skip;
LineComment : ('//' ~[\r\n]* ) -> skip;
BlockComment :('/*' .*? '*/') -> skip;
```

3. constExp 定义问题

当 constExp 取以下代码中第一行的定义时（来源于 cact25-spec.pdf），会将 19_false_val_init.cact 文件判定为正确。

```
//constExp : addExp ;// 使用的 Ident 必须是常量
//显式初始化：CACT 限制初值表达式必须是常数
constExp : number ;
```

19_false_val_init.cact 内容如下，代码错误的原因是 c 在初始化时，初值表达式必须是常数。

```
int a = 2;
const int b = 3;
int main()
{
     int c = b;
     // c 在初始化时，初值表达式必须是常数
     return c;
}
```

查看错误输出如下，可以发现 Praser 将 b 识别为了 constExp 导致识别正确。由于 cact25-spec.pdf 中关于初始化的定义为限制初值表达式必须是常数，因此直接将 constExp 的定义改为 number 即可。

```
(compUnit
    (decl
        …
    )
    (decl
       …
    )
    (funcDef
        (funcType int)
        main
        (funcFParams)
        (block {
            (blockItem
                (decl
                    (vardecl
                        (bType int)
                        (varDef c =
                            (constInitVal
                                (constExp
                                    (addExp
                                        (mulExp
                                            (unaryExp
                                                (primaryExp (lVal b))
                                            )
                                        )
                                    )
                                )
                            )
                        )
                    ;)
                )
            (blockItem
                …
            )
        })
    )
    <EOF>
)
```

4. 数组初始化

发现代码不能对以下内容报告非法，因为其赋值时将数组类型和其他类型没有做出区分。

```
int a[4] = 4; //非法，因为数组赋值不能为 int 类型
```

因此进行以下修改，将 varDef 划分为 varDef 和 varDefArray，其中 varDef 不包括定义数组，varDefArray 的等号右边是 constInitValArray，其必须要有大括号在最外层，可以解决这一问题。

```
/*
vardecl : bType varDef ( Comma varDef )* Semi ;
varDef : Ident ( LeftBracket IntConst RightBracket )* ( Assign constInitVal )?;
//以上内容转化为以下部分
*/

//尝试解决数组初始化问题
//解决 int a[4] = 4;非法
vardecl : bType (varDef | varDefArray) ( Comma (varDef | varDefArray) )* Semi ;
varDefArray : Ident LeftBracket IntConst RightBracket ( LeftBracket IntConst RightBracket )* ( Assign
constInitValArray )?;
constInitValArray : LeftBrace ( constInitVal ( Comma constInitVal )* )? RightBrace ;
varDef : Ident ( Assign constInitVal )?;
```

五. 实验结果

运行 27 个 cact 文件，27 个测试点全部通过。且运行正确的输入文件，输出 0；运行有错误的输入文件，输出 1。

```
正在处理文件: 00_true_main.cact
-------
(compUnit (funcDef (funcType int) main ( funcFParams ) (block { (blockItem (decl (vardecl (bType int)
(varDef a = (constInitVal (constExp (number 0)))) ;))) (blockItem (stmt return (exp (addExp (mulExp
(unaryExp (primaryExp (number 0)))))) ;)) })) <EOF>)
0
正在处理文件: 01_false_hex_num.cact
line 3:13 extraneous input 'x' expecting {';', ','}
-------
(compUnit (funcDef (funcType int) main ( funcFParams ) (block { (blockItem (decl (vardecl (bType int)
(varDef a = (constInitVal (constExp (number 0)))) x ;))) (blockItem (stmt return (exp (addExp (mulExp
(unaryExp (primaryExp (lVal a)))))) ;)) })) <EOF>)
1
正在处理文件: 02_true_octo.cact
-------
(compUnit (funcDef (funcType int) main ( funcFParams ) (block { (blockItem (decl (vardecl (bType int)
(varDef a = (constInitVal (constExp (number 077)))) ;))) (blockItem (stmt return (exp (addExp (mulExp
(unaryExp (primaryExp (number 0)))))) ;)) })) <EOF>)
0
正在处理文件: 03_false_bracket.cact
line 2:11 mismatched input '}' expecting ']'
line 2:13 extraneous input '=' expecting {<EOF>, 'const', 'int', 'char', 'float', 'void'}
-------
(compUnit (funcDef (funcType int) main ( funcFParams ) (block { (blockItem (decl (vardecl (bType int)
(varDef a [ 3) <missing ';'>))) })) = ] 2 ] ; return 0 ; } <EOF>)
1
```

```
……


正在处理文件: 25_false_nested_func_def.cact
line 2:13 mismatched input '(' expecting {';', ','}
line 2:14 mismatched input ')' expecting {'(', '+', '-', '!', Ident, IntConst, FloatConst}
line 2:15 missing ';' at '{'
-------
(compUnit (funcDef (funcType int) main ( funcFParams ) (block { (blockItem (decl (vardecl (bType float)
(varDef foo)))) (blockItem (stmt (exp (addExp (mulExp (unaryExp (primaryExp ( (exp (addExp (mulExp
unaryExp))) )))))) <missing ';'>)) (blockItem (stmt (block { (blockItem (stmt return (exp (addExp
(mulExp (unaryExp (primaryExp (number 1.0f)))))) ;)) }))) (blockItem (stmt return (exp (addExp (mulExp
(unaryExp (primaryExp (number 0)))))) ;)) })) <EOF>)
1
正在处理文件: 26_true_multi_dim_const.cact
-------
(compUnit (decl (constdecl const (bType float) (constDef a [ 1 ] [ 2 ] [ 2 ] = (constInitVal
{ (constInitVal { (constInitVal { (constInitVal (constExp (number 1.0f))) , (constInitVal (constExp
(number 2.0f))) }) , (constInitVal { (constInitVal (constExp (number .3f))) , (constInitVal (constExp
(number 4.0f))) }) }) })) ;)) (funcDef (funcType int) main ( funcFParams ) (block { (blockItem (stmt
(exp (addExp (mulExp (unaryExp print_float ( (funcRParams (exp (addExp (mulExp (unaryExp (primaryExp
(lVal a [ (exp (addExp (mulExp (unaryExp (primaryExp (number 0)))))) ] [ (exp (addExp (mulExp (unaryExp
(primaryExp (number 0)))))) ] [ (exp (addExp (mulExp (unaryExp (primaryExp (number
0)))))) ])))))) ))))) ;)) (blockItem (stmt return (exp (addExp (mulExp (unaryExp (primaryExp (number
0)))))) ;)) })) <EOF>)
0
```

六. 思考题

1.如何设计编译器的目录结构?

本实验中，CACT 的目录结构如下:

```
cact/
├── build/
├── deps/
├── grammar/
├── src/
├── test/samples_lex_and_syntax/
├── CMakeLists.txt
……
```

build/：存放脚本文件和编译后生成的可执行文件、目标文件、中间产物。

deps/：存放依赖项，例如 ANTLR 运行时库、第三方工具等。

grammar/：存放 ANTLR 文法文件(.g4)、词法分析器、语法分析器。

src/：存放编译器的 C++源代码，包括语法树构建、语义分析、中间代码生成等模块。

test/samples_lex_and_syntax/：存放测试文件(.cact)，用于验证词法和语法分析的功能。

CMakeLists.txt：CMake 配置文件。

2.如何把表达式优先级体现在文法设计中？

2.1 词法分析中的优先级处理：

词法分析是编译器的第一步，负责将源代码转化为一个个的“token”。由于编译器是按顺序匹配规则的，所以我们需要把优先级较高的匹配子规则写在前面，把匹配长度较长的子规则写在前面。如果高优先级的规则写在后面，编译器可能会先匹配到较低优先级的规则，从而产生错误的 token。例如，在处理<和<=时，必须确保<=的规则在<之前，否则编译器会误将<=匹配成两个独立的<符号和=符号。

2.2 语法分析中的优先级处理：

在语法分析阶段，我们会根据文法来构造抽象语法树（AST）。语法分析的表达式优先级可以分为表达式内部的优先级和表达式之间的优先级。在本实验中，表达式的语法结构采用了递归式文法规则以体现各类操作符之间的优先级关系。具体做法是：按照从低优先级到高优先级的顺序，逐层构造表达式的语法规则，且每一层的规则均采用右递归的形式，以满足 ANTLR4 对语法规则结构的要求。

对于表达式之间的优先级，为了正确地处理不同算符的优先级，我们通常需要从低优先级到高优先级的顺序构造文法规则。例如，对于包含加减、乘除、括号的表达式，因为括号的优先级最高，加减的优先级最低，以下是其优先级处理方法：

```
addExp : mulExp | addExp (Plus | Minus) mulExp ;

mulExp : unaryExp | mulExp (Star | Div | Mod) unaryExp ;

unaryExp : primaryExp | (Plus | Minus | Not) unaryExp
        | Ident LeftParen ( funcRParams )? RightParen ;

primaryExp : LeftParen exp RightParen | lVal | number ;
```

对于比较运算符（==, !=, <, >, <=, >=）和逻辑运算符（&&, ||），lOrExp（||）优先级最低，因此它位于最底层；lAndExp（&&）次之；eqExp（==, !=）的优先级高一些；relExp（<, >, <=, >=）优先级最高。我们需要按优先级从低到高设计相应的规则：

```
lOrExp : lAndExp
      | lOrExp '||' lAndExp ;

lAndExp : eqExp
       | lAndExp '&&' eqExp ;

eqExp : relExp
```

```
      | eqExp '==' relExp
      | eqExp '!=' relExp ;

relExp : addExp
       | relExp '<' addExp
       | relExp '>' addExp
       | relExp '<=' addExp
       | relExp '>=' addExp ;
```

在这个规则中，括号中的 expr 拥有最高优先级。当多个运算符组合在一起时，必须通过括号来控制优先级。例如，(a + b) * c，括号使得加法运算先进行，乘法运算后进行。括号在文法中被处理为原子表达式：

```
primaryExp : LeftParen expr RightParen
           | number
           | lVal ;
```

我的 g4 语法通过自底向上的方式逐步构建表达式树，优先级从高到低依次是：

```
1. primaryExp         -> ((), 数组访问或变量, 数值常量)
2. unaryExp           -> (+, -, !, 函数调用)
3. mulExp            -> (*, /, %)
4. addExp            -> (+, -)
5. relExp            -> (<, >, <=, >=)
6. eqExp             -> (==, !=)
7. lAndExp           -> (&&)
8. lOrExp            -> (||)
```

对于表达式内部的优先级，需要在每一条语法规则当中，把匹配概率高的放在前面，匹配概率低的放在后面。并且对于存在递归的子规则，本实验中使用了右递归的方式分层表达各级表达式优先级。

3.如何设计数值常量的词法规则?

3.1 本实验中，使用非终结符 number 统一表示整形常量和浮点常量。数值常量的定义如下：

```
number : IntConst | FloatConst ;
```

3.2 整形常量(IntConst)：

整型常量支持三种进制表示方式：

十进制常量 DecimalConst：由非零数字开头（1-9），后接任意位数字，或单独为“0”；

八进制常量 OctalConst：以 0 开头，后接一个或多个 0~7 的数字；

十六进制常量 HexadecConst：以 0x 或 0X 开头，后跟一个或多个十六进制数字（0-9,a-f,A-F）。

这些规则的设计参考了 C 语言的标准整型字面量形式。词法规则如下

```
IntConst : DecimalConst | OctalConst | HexadecConst ;

DecimalConst : '0' | [1-9] [0-9]* ;
OctalConst : '0' [0-7]+ ;
HexadecConst : ('0x'|'0X') [0-9a-fA-F]+ ;
```

3.3 浮点常量(FloatConst):

浮点常量遵循以下规则：包含小数点.，可选指数部分 e/E。若带有 f 或 F 后缀，识别为 float 类型；若无后缀，则默认为 double 类型。规则中对小数部分与指数部分进行了拆分识别，以支持灵活的格式组合。

实验中，我们未专门定义 DoubleConst，所有浮点常量统一使用 FloatConst，并允许无 f/F 后缀的情况。相关规则如下:

```
FloatConst
    : FractionalConstant ExponentPart? FloatingSuffix?
    | DigitSequence ExponentPart FloatingSuffix?
    ;

fragment FractionalConstant
    : DigitSequence? '.' DigitSequence
    | DigitSequence '.'
    ;

fragment ExponentPart : [eE] Sign? DigitSequence ;

fragment Sign : [+-] ;

fragment DigitSequence : [0-9]+ ;

fragment FloatingSuffix : [fF] ;
```

4.如何替换 ANTLR 的默认异常处理方法?

4.1 ANTLR 的默认异常处理机制:

在使用 ANTLR4 构建语法分析器时，如果输入的源代码不符合文法规则，ANTLR 会在语法树中插入错误节点（ErrorNode），并通过默认的方式进行错误处理。

4.2 本实验中，我们利用 ParseTreeListener 的 visitErrorNode 来处理错误。

我们在 Analysis.h 中写了一个自定义的 ParseTreeListener，重写了其中的 visitErrorNode 方法，用来捕捉语法分析阶段出现的错误节点。

```
void visitErrorNode(tree::ErrorNode* node) override {
    std::cout << "1" << std::endl;
    std::ofstream outfile("output.txt", std::ios::app);
```

```
    if (outfile.is_open()) {
        outfile << "1" << std::endl;
        outfile.close();
    } else {
        std::cerr << "无法打开 output.txt 文件" << std::endl;
    }
    exit(1);
}
```

这段代码表示：只要在语法树中出现错误节点 ErrorNode，就立刻退出并输出 1 到文件中。通过 tree::ParseTreeWalker::DEFAULT.walk() 遍历语法树时就能自动触发该方法，实现对默认错误处理机制的替代。

七. 实验总结

通过本次实验，我们深入学习并掌握了使用 ANTLR4 构建词法分析器与语法分析器的完整流程，初步理解了编译器前端的基本组成结构。在文法编写过程中，我们系统梳理了 CACT 语言的语法规则，逐步熟悉了巴科斯范式的表达方式，并通过反复调试和样例验证，不断提升了文法的正确性和覆盖能力。

本实验还涉及语法树的遍历与错误处理机制。我们采用 Listener 模式对语法树进行遍历，通过重写 visitErrorNode 方法，实现了自定义的语法错误检测和响应逻辑。尽管 ANTLR 同时支持 Visitor 模式，本实验中暂未使用该机制，未来可以在功能扩展阶段进一步比较两种遍历模式的差异与适用场景，从而加深对 ANTLR 遍历机制和错误处理策略的理解。

此外，我们还调试并分析了实验框架中的 main.cpp 与 Analysis.h，在实践中掌握了 ANTLR4 与 C++接口的基本用法，包括输入流构造、词法与语法分析器对象的创建、语法树的生成与遍历等关键流程。通过运行不同输入样例并观察输出结果，我们对编译器前端的整体工作流程有了更直观、系统的认识。

总的来说，这次实验不仅帮助我们掌握了使用 ANTLR4 编写文法文件并构建词法分析器与语法分析器的基本方法，也提升了我们在调试与错误处理方面的能力，为后续的编译原理实验打下了坚实基础。