编译原理研讨课CACT实验PR001报告

成员组成

第16小组：

彭睿思 2020K8009915018

严哲虞 2020K8009907030

游昆霖 2020K8000026006

赵若雯 2020K8009929031

任务说明

本实验需要完成的任务包括：

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：
   - 参考demo样例，并按照CACT的文法规范编写ANTLR文法文件，通过ANTLR工具生成CACT源码的词法-语法分析器和访问接口listener、visitor，使之能对.cact文件进行词法和文法分析
   - 修改ANTLR中默认的文法错误处理机制，对于符合词法和文法规范的.cact文件返回值为0，否则返回非零值

实验过程

设计思路

1. ANTLR文法文件

   对于ANTLR文法文件，需要参照和补全实验讲义给出的CACT语言规范，并结合编写ANTLR代码的语法规则来定义词法表达式和文法表达式

2. main文件

   - main函数的设计思想是生成CACTLexer和CACTParser实例Lexer和Parser，并将被测试的.cact程序打包成ANTLRInputStream类作为输入字符流，进行词法分析和语法分析。在进行语法分析时，首先通过调用`parser.compUnit()`构建语法树，然后选择listener或visitor模式遍历语法树，以便检查源.cact程序是否存在语法错误。
   - listener遍历模式设计思想：

     由于listener方法会被ANTLR提供的遍历器对象（比如`ParseTreeWalker`）自动调用，因此只需将构建的监听器和语法分析树传入walk方法，该方法就会自动遍历语法树并触发回调。

     ```
     Analysis listener;
     tree::ParseTreeWalker::DEFAULT.walk(&listener, tree);
     ```

   - visitor遍历模式设计思想：

     在构建语法分析树之后，可以使用访问器对树中各个节点进行遍历，检查是否出现语法错误。与listener不同的是，listener方法会被遍历器自动调用，但在visitor方法中，程序设计者必须显式调用visit方法来访问各个子节点。在使用visitor模式时，需要在运行antlr构建工具时通过指定命令参数`-visitor`和`-no-listner`来生成包含访问器的头文件和基础父类。然后我们可以在main函数中手动构建visitor分析类，递归地访问语法分析树中的每一个子节点。

3. 实验辅助工具

   针对多人同步协作，我们可以编写脚本忽略不同本地依赖之间的冲突，进行个人分支和master主分支之间的进度同步。同时，为了便于对测试样例实现批量测试和结果统计，也可以通过脚本，执行批量测试，并对打印信息和测试结果进行整合，帮助我们定位错误。

代码实现

环境配置

1. 个人分支和master分支的同步

   - 创建分支

     ```
     git checkout -b xxx
     //第一次提交为本地分支创建相应远程分支
     git push --set-upstream origin xxx
     //之后的提交
     git push origin xxx
     ```

- 运行脚本的要求
  - 在个人分支下完成git add .和git commit
  - cact目录下运行`sh s2m.sh xxx`将个人分支进度同步到master
  - cact目录下运行`sh m2s.sh xxx`将master分支进度同步到个人分支

2. 在bash终端提示符中显示当前分支

将`~/.bashrc`中的

```
if [ "$color_prompt" = yes ]; then
  PS1='${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\[\033[00m\]:\
[\033[01;34m\]\w\[\033[00m\]\$ '
else
  PS1='${debian_chroot:+($debian_chroot)}\u@\h:\w\$ '
fi
unset color_prompt force_color_prompt
```

更改为

```
show_git_branch() {
 git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/(\1)/'
}
if [ "$color_prompt" = yes ]; then
  PS1='${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\[\033[00m\]:\
[\033[01;34m\]\w\[\033[01;32m\]$(show_git_branch)\[\033[00m\]\$ '
else
  PS1='${debian_chroot:+($debian_chroot)}\u@\h:\w$(show_git_branch)\$ '
fi
unset color_prompt force_color_prompt
```

3. 忽视本机修改依赖对远程仓库的影响

   notes目录下执行

   ```
   chmod +x ignores-deps.sh
   ./ignore-deps.sh
   ```

   - 原理：通过`git update-index --skip-worktree <filename>`可以让git认为这个文件始终处于最新提交状态（跟踪文件状态，不跟踪具体状态）。因此，不会在checkout切换分支或者git push 提交时产生影响。
   - 注意：该方法要求远程仓库的deps不会进行修改，否则在git pull时会更新到远程仓库的内容。这需要保证每个人在配置自己环境时都先执行本脚本。

4. 在本机产生cact项目编译相关依赖

   notes目录下执行

   ```
   chmod +x myenv.sh
   ./myenv.sh
   ```

   - 注意：先执行`./ignore-deps.sh`
   - 原理：实验一在服务器目录下执行的cmake使用了绝对路径，使得本机的依赖路径不同。该脚本在本机路径下执行cmake和make，使得本机可以编译cact项目。
   - 测试：将cact项目删除build文件夹和grammar文件夹中除Hello.g4外文件，然后重复PR001实验说明的第三步跑通demo。

5. 查看打了标识的文件

   ```
   git ls-files -v | grep -i ^S
   ```

6. 其他说明：

不同电脑上由于依赖不同，编译cact项目时的build也会有一定差异，不过一方面该文件夹小，改动消耗少，而且随着.g4文件的修改也会频繁改动，因此不需要使用上述方法进行忽视。

ANTLR文法代码实现

在CACT.g4文件中定义规则，包括词法规则和语法规则，每条规则都是`key: value`的形式，并且以`;`结尾。文法实现采用EBNF范式，用`|`表示分支选项，用`*`表示匹配前一个匹配项0次或者多次，用`+`表示匹配前一个匹配项1次或者多次。对于变量名的命名，在lexer中首字母要大写，parser中首字母小写。

对于CACT语言规范给出定义的规则，我们需要翻译以使其符合ANTLR文件编写规则，例如语言规范文件中的[]，{}分别表示匹配0或1次、重复任意次，按照ANTLR文件编写规则应该对应的改为()?、()*。与此同时，我们还需要调整产生式右部每个规则的优先级。对于词法而言，需要保证不会因为前一个规则的部分识别影响后一个规则的识别；对于语法而言，我们需要调整递归子规则排序在后以提高程序性能。

对于课程讲义中没有给出的词法表达式，如`DecimalConst` `OctalConst` `HexadecConst` `FloatConst`和`DoubleConst`，需要进行规则补全。例如，对于`IntConst`的三种可能，我们分别使用`fragment`分片规则进行描述；对于`FloatConst`和`DoubleConst`，区别只在于后缀，因此我们使用`fragment`规则描述其共同前缀，该前缀可分为一般小数形式和含指数的科学计数法形式，可进一步简化为含小数点形式和不含小数点、含有指数的形式。由此，我们给出如下定义:

```
IntConst
    : DecimalConst
    | OctalConst
    | HexadecConst
    ;

fragment DecimalConst
    : '0'
    | [1-9] [0-9]*
    ;

fragment OctalConst
    : '0' [0-7]+
    ;

fragment HexadecConst
    : HexPrefix [0-9a-fA-F]+
    ;

fragment HexPrefix
    : '0x'
    | '0X'
    ;

FloatConst
    : PreFloatDouble ('f' | 'F')
    ;

DoubleConst
    : PreFloatDouble
    ;
```

```
fragment PreFloatDouble
    : Fraction Exponent?
    | [0-9]+ Exponent
    ;
fragment Fraction
    : [0-9]+ '.' [0-9]+
    | [0-9]+ '.'
    | '.' [0-9]+
    ;

fragment Exponent
    : ('E' | 'e') (ADD | SUB)? [0-9]+
    ;
```

在编写上述代码的过程中需要注意的一项是，在词法规则中那些不会被语法规则直接调用的词法规则要用一个`fragment`关键字来标识，而被`fragment`标识的规则只能Wie其他词法规则提供基础。例如，在parser中会直接调用`IntConst`

```
constDef
: Ident (LeftBracket IntConst RightBracket)* ASSIGN constInitVal
;
```

因此`InstConst`不能用`fragment`修饰，但在定义`InstConst`这个词法规则时，还需要使用`DecimalConst OctalConst`和`HexadecConst`这三个token的词法规则，但它们都不会被parser直接调用，因此在定义这三个词法规则时都需要用`fragment`来修饰

main.cpp实现

我们的代码支持选择listener或visitor模式，为此针对这两个模式分别编写一个`main.cpp`。这两个模式的`main.cpp`都由`class Analysis`的声明以及`main`函数两部分构成。

1. `class Analysis`

在定义类`Analysis`时，我们需要继承ANTLR已有的类，并重写父类的`virtual`方法。这一部分有两种写法：

- 如果按照demo中的写法，我们可以继承自`CACTVisitor`或`CACTListener`类。由于ANTLER生成的`CACTBaseVisitor.h`/`CACTBaseListener`.h中提供了一个空实现的示例，可以直接复制它的写法，然后添加`visitErrorNode`方法。
- 如果想省略大量对于`virtual`方法的`override`声明，

  可以参照`CACTVisitor`或`CACTListener`的实现，继承自依赖中的类，并重写基本的一些方法。

我们在visitor模式的`main.cpp`中使用上面的第一种写法，并添加`visitErrorNode`方法，它会在判断一个结点为`ErrorNode`时被调用，负责识别错误，打印错误信息，并通过`exit(1)`退出，符合返回非零值的实验要求。对于类中的其他方法，只需仿照`CACTBaseVisitor.h`重写，访问子树后返回。

```
class  Analysis : public CACTVisitor {
public:

  virtual antlrcpp::Any visitErrorNode(tree::ErrorNode *node) override{
```

```
    std::cout << "Error:" << node->getText() <<std::endl;
    exit(1);
  }
  virtual antlrcpp::Any visitCompUnit(CACTParser::CompUnitContext *ctx) override
{
    return visitChildren(ctx);
  }

  virtual antlrcpp::Any visitDecl(CACTParser::DeclContext *ctx) override {
    return visitChildren(ctx);
  }

  virtual antlrcpp::Any visitConstDecl(CACTParser::ConstDeclContext *ctx)
override {
    return visitChildren(ctx);

  ......

}
```

在listener中，我们采用上面的第二种写法，这种方法与demo中的写法不同，胜在简洁。

```
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 {exit(1);}
    virtual void exitEveryRule(ParserRuleContext* ctx) override {}
};
```

我们利用ANTLR4工具提供的接口`ParseTreeListener`，它会在语法分析器（Parser）生成语法树（Parse Tree）时监听节点的进入和退出。我们从`ParseTreeListener`类继承`Analysis`，在`visitErrorNode`中用`exit(1)`退出，并重写其余的方法。查看`ParseTreeListener.h`中的定义可知，这一父类一共只有4个方法，因此在继承并重写时更加方便。

2. `main`函数

listener和visitor模式中`main`函数的代码大体相同，这里先以listener模式为例，介绍`main`函数的实现。

首先定义了一个`std::ifstream`类型的流对象`stream`，并通过打开`argv[1]`指定的文件，将文件内容传入`ANTLRInputStream`类的构造函数中，生成ANTLR输入流对象`input`，再从中分别生成词法分析器`lexer`和语法分析器`parser`。

随后，调用`parser.compUnit()`构建语法树`tree`，并输出AST的内容，便于debug。

最后，从我们上面实现的`Analysis`类中实例化一个对象`visitor`，通过`walk`方法隐式遍历AST。

```
int main(int argc, const char* argv[]) {
    std::ifstream stream;
    stream.open(argv[1]);
    ANTLRInputStream input(stream);
    CACTLexer lexer(&input);
```

```
    CommonTokenStream tokens(&lexer);
    CACTParser parser(&tokens);

    tree::ParseTree *tree = parser.compUnit();

    std::cout << "----------------------------Print AST:-----------------------
----" << std::endl;
    std::cout << tree->toStringTree(&parser) << std::endl;

    Analysis listener;
    tree::ParseTreeWalker::DEFAULT.walk(&listener, tree);
    return 0;
}
```

而在visitor模式中，`main`函数唯一的区别在于最后遍历AST时。这里也是从`Analysis`类实例化对象，但调用的是`visit`方法。并且`Analysis`类中会通过显示地调用`visitChildren`来遍历AST，这一部分已经在定义类时实现。

```
  Analysis visitor;
  visitor.visit( tree );
```

rebuild和test脚本实现

为了在实验过程中方便构建和测试，我们编写了`rebuild.sh`和`test.sh`脚本。

1. `rebuild.sh`脚本

首先在终端输出提示符，输入`l`表示选择`listener`模式，输入`v`或直接换行为`visitor`模式。

```
echo "Input Mode:"
echo "  l       : Listener"
echo "  v       : Visitor"
echo "  default: Visitor"
read choice
if [ "$choice" == "l" ];then
    echo "Mode: Listener"
    cp -r src_listener/ src/
else
    echo "Mode: Visitor"
    cp -r src_visitor/ src/
fi
```

随后，它会根据选择的模式，将源文件（本实验中为`main.cpp`）拷贝到src目录下，然后重新生成grammer文件夹下除.g4外其余文件，并重新编译程序。这里在使用visitor模式时，需要额外指定命令参数`-visitor`和`-no-listner`，以此生成包含访问器的头文件和基础类。

```
cd grammar
find . ! -name "$keep_file" -type f -delete
if [ "$choice" == "l" ];then
    java -jar ../deps/antlr-4.8-complete.jar -Dlanguage=Cpp "$keep_file"
else
```

```
    java -jar ../deps/antlr-4.8-complete.jar -Dlanguage=Cpp -no-listener -
visitor "$keep_file"
fi
cd ..
rm -rf build
mkdir build
cd build
cmake ..
make
cd ..
cp ./build/compiler ./
rm -rf src/
```

其运行时使用的命令如下。

```
# 第一次使用前
chmod +x ./rebuild.sh

# 后续使用
# Listerer 模式
./rebuild
...     # 脚本提示
l
# Visitor 模式
./rebuild
...     # 脚本提示
v       # 或直接换行
```

2. `test.sh`脚本

`test.sh`代码如下。其功能为：

1. 批量测试samples
2. 输出对每个样例执行时的打印信息(Output)及返回码（Exit code），并通过返回码和名字匹配判断是否通过
3. 统计出错样例(Fail list)和通过率(Pass rate)

```
#!/bin/bash

# 指定目录和命令
dir="samples/samples_lex_and_syntax"
# dir="samples/demo"
command="./compiler"

total_cnt=0
pass_cnt=0
fail_list=()
# 遍历目录中的所有文件
for file in "$dir"/*
do
    # 检查文件是否存在
    if [ -f "$file" ]
    then
```

```
        echo ">>> Test on file: $file"

        ((total_cnt=total_cnt+1))
        # 执行命令并获取输出信息和返回值
        output=$(eval "$command \"$file\"")
        exit_code=$?

        # 打印输出信息和返回值
        echo ">>>> Output:"
        echo "$output"
        echo ">>>> Exit code: $exit_code"

        if [ $exit_code -ne 0 ] && [[ $file == *"false"* ]]
        then
                echo "Pass"
                ((pass_cnt=pass_cnt+1))
        elif [ $exit_code -eq 0 ] && [[ $file == *"true"* ]]
        then
                echo "Pass"
                ((pass_cnt=pass_cnt+1))
        else
                echo "Fail"
                substr=$(echo $file | awk -F"/" '{print $NF}' | awk -F"_"
'{print $1}')
                fail_list+=($substr)
        fi
        echo -e "\n"
    fi
done

echo "Fail list: ${fail_list[@]}"
echo "Pass rate: $pass_cnt / $total_cnt "
```

运行时使用的命令如下。

```
# 第一次使用前
chmod +x ./test.sh

# 批量测试
./test.sh
```

由于脚本用的是bash语法，所以不能使用`sh`命令。上述命令也可以替换为`bash test.sh`。

debug过程

使用未经修改的文法文件可以通过27/28个测试点，失败的测试用例`27_true_mixed_sample.cact`的输出处理可见notes文件夹下`sample27_debug`。在main中的最后一个else中的部分推导应该由

```
unaryExp
  print_bool
  (
  false  //wrong here
  )
```

修改为

```
unaryExp
  print_bool
  (
  funcRParams
    exp
      false
  )
```

出现上述错误是因为`BoolConst`未被识别为终结符。

这是因为，在写关键字列表时，已经为"true"和"false"指定了终结符，在写`BoolConst->TURE|FALSE`产生式时，因为右部为终结符，左部应当为非终结符。但首字母大写不符合非终结符的定义规则。

解决该错误有下述方法

- 将`BoolConst`更名为`boolConst`，即认为是非终结符
- 为`TRUE`和`FALSE`添加`fragment`分片说明，则`BoolConst`仍可被认为是终结符。
- `BoolConst`直接产生`true`和`false`

综合考虑下，认为第二种方法更为合理。

上述错误的发现还启发了下面这种写法的不合理性：

```
FloatConst
    : DoubleConst ('f' | 'F')
    ;

DoubleConst
    : ...
```

此处`FloatConst`和`DoubleConst`都是需要被语法规则调用的终结符，因此应该使用一个中间表示来避免上述情形。

修改后该处文法应当如下：

```
FloatConst
    : PreFloatDouble ('f' | 'F')
    ;

DoubleConst
    : PreFloatDouble
    ;

fragment PreFloatDouble
    : ...
```

过程思考

1. 如何设计编译器的目录结构?

从总体上说，编译器的目录结构可以按功能或者层次进行划分：

按功能划分：可以将源代码、词法分析器、语法分析器、语义分析器、中间代码生成器、代码生成器等功能划分为不同的目录或模块，这样设计可以帮助维护者快速找到相关代码，并提高代码的模块化和可维护性。

按层次划分：可以将编译器的不同层次，如前端和后端，划分为不同的目录和模块，这种划分方式可以防止跨层次的代码耦合，能更容易理解编译器的整体架构

在本实验中，编译器的目录结构主要按照功能以及实现步骤来构建：编译器的主体代码位于`/src`当中，文法文件、词法分析器和语法分析器位于`/grammar`当中，antlr工具的环境安装在`/deps`当中，测试用例位于`/samples`，最后我们使用cmake进行构建，利用脚本辅助测试

2. 如何把表达式优先级体现在文法设计中?

对于词法分析，如果匹配了一个词法规则前面的token，就无法再匹配该词法规则后面的token，因此在每条词法规则中，要把优先级高的匹配子规则写在前面，要把匹配长度较长的子规则写在前面，如"<="先于"<"。

对于语法分析，表达式的优先级可以被分为表达式之间的优先级和表达式内部的优先级，而表达式之间的优先级主要取决于算符的优先级。以括号、乘除和加减表达式为例，三个算符的优先级顺序是由高到低，在处理优先级问题时，应该从优先级低的算符到优先级高的算符的顺序构造文法规则。因为括号拥有三个算符当中的最高优先级，因此由括号括起的复合表达式可以作为原子表达式。构造的文法规则如下：

```
expr : term
      | expr '+' term
      | expr '-' term
term : fator
     | term '*' factor
     | term '/' factor

factor : NUMBER
       |'('expr')''
```

然后再以比较和逻辑运算为例，只涉及'>'，'<'，'=='，'!='和逻辑运算符'&&'和'||'。由于所有比较运算符都具有相同的优先级，逻辑运算符中'&&'优先级要比'||'高，并且比较运算符优先级高于逻辑运算符所以与四则运算的文法类似，先构建低优先级的复合表达式，最后构建最高优先级的原子表达式。因此要首先是描述||的规则，然后是&&，最后是原子表达式：

```
compexpr : andexpr
         | compexpr '||' andexpr
andexpr : factor
         | andexpr '&&' factorfactor
factor : ID '>' NUMBER
       | ID '<' NUMBER
       | ID '==' NUMBER
       | ID '!=' NUMBER
       | '(' compexpr ')'
```

根据上述四则运算和比较运算的文法，我们可以总结具有优先级的表达式文法的构造原则是：

- 按照从低优先级到高优先级的顺序构造规则。
- 每个复合表达式都有一个只包含下一个优先级的表达式作为解析选项。
- 括号具有高优先级，被括号括起的符合表达式要作为原子表达式。

处理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，概率低的放在后面。并且对于存在递归的子规则，例如`exp : exp '+' term`，排序在后更有利于提升语法分析的性能。

根据上述构造原理，可以设计出如下语法规则：

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

number
    : IntConst
    | DoubleConst
    | FloatConst
    ;

unaryExp
    : primaryExp
    | (ADD | SUB | NOT) unaryExp
    | Ident LeftParen (funcRParams)? RightParen
    ;

funcRParams
    : exp (COMMA exp)*
    ;

mulExp
    : unaryExp
    | mulExp (MUL | DIV | MOD) unaryExp
    ;
```

```
addExp
    : mulExp
    | addExp (ADD | SUB) mulExp
    ;

relExp
    : addExp
    | relExp (LEQ | GEQ | LT | GT) addExp
    | BoolConst
    ;

eqExp
    : relExp
    | eqExp (EQ | NEQ) relExp
    ;

lAndExp
    : eqExp
    | lAndExp AND eqExp
    ;

lOrExp
    : lAndExp
    | lOrExp OR lAndExp
    ;
```

3. 如何设计数值常量的词法规则?

数常量分为整型常量、单精度浮点常量和双精度浮点常量。

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

对于整型常量`IntConst`，分十进制、八进制、十六进制三种情况考虑。十进制包括0，以及其它所有以1~9为开头的数；八进制的所有数需要在开头以0标记，后面是由0-7组成的数；十六进制有0x和0X两种标记，为此专门设置一个符号`HexPrefix`，其后是由0-9、a-f或A-F组成的数。

```
IntConst
    : DecimalConst
    | OctalConst
    | HexadecConst
    ;

fragment DecimalConst
    : '0'
    | [1-9] [0-9]*
    ;

fragment OctalConst
```

```
    : '0' [0-7]+
    ;

fragment HexadecConst
    : HexPrefix [0-9a-fA-F]+
    ;

fragment HexPrefix
    : '0x'
    | '0X'
    ;
```

对于32位浮点数常量`FloatConst`和64位浮点常量`DoubleConst`，通过后面有无f或F区分。使用`PreFloatDouble`生成前面的数值部分。

```
FloatConst
    : PreFloatDouble ('f' | 'F')
    ;

DoubleConst
    : PreFloatDouble
    ;
```

数值部分分两种情况考虑：以小数开头和以整数开头。对于以小数开头的情形，又分为普通形式（只有小数本身）和指数形式两种情况。

```
fragment PreFloatDouble
    : Fraction Exponent?
    | [0-9]+ Exponent
    ;
fragment Fraction
    : [0-9]+ '.' [0-9]+
    | [0-9]+ '.'
    | '.' [0-9]+
    ;

fragment Exponent
    : ('E' | 'e') (ADD | SUB)? [0-9]+
    ;
```

4. 如何替换ANTLR的默认异常处理方法?

我们在listener和visitor模式下采用不同的方法替换默认异常处理方法。

在listener模式下，调用`walk`方法时从根开始，遍历语法树中的节点。查看位于deps/antlr4-runtime/runtime/src/tree/ParseTreeWalker.cpp中`walk`的源码可知，在结点被判定为`ErrorNode`时，会调用`visitErrorNode`方法访问该节点，而`visitErrorNode`方法负责的正是对于错误的处理。于是，我们通过继承`antlr4::tree::ParseTreeListener`并重写其`visitErrorNode`方法，输出错误结点的信息并返回1。

而在visitor模式下，我们仿照demo中的写法，继承自`CACTVisitor`类，重写其所有方法，在`visitErrorNode`方法中输出错误结点的信息并返回1。

总结

实验结果总结

在listen和visitor模式下，我们都通过了全部28个测试点。

```
./test.sh
>>> Test on file: samples/samples_lex_and_syntax/00_true_main.cact
>>>> Output:
--------------------------Print AST:-----------------------------
(compUnit (funcDef (funcType int) main ( ) (block { (blockItem (decl (varDecl
(bType int) (varDef a = (constInitVal (constExp (number 0)))) ;))) (blockItem
(stmt return (exp (addExp (mulExp (unaryExp (primaryExp (number 0)))))) ;)) }))
<EOF>)
>>>> Exit code: 0
Pass


>>> Test on file: samples/samples_lex_and_syntax/01_false_hex_num.cact
line 3:13 extraneous input 'x' expecting {';', ','}
>>>> Output:
--------------------------Print AST:-----------------------------
(compUnit (funcDef (funcType int) main ( ) (block { (blockItem (decl (varDecl
(bType int) (varDef a = (constInitVal (constExp (number 0)))) x ;))) (blockItem
(stmt return (exp (addExp (mulExp (unaryExp (primaryExp (lVal a)))))) ;)) }))
<EOF>)
>>>> Exit code: 1
Pass

......

>>>> Exit code: 0
Pass


Fail list:
Pass rate: 28 / 28
```

通过本次实验，我们熟悉了ANTLR工具的使用，也在编写文法文件的过程中初步掌握了CACT语言规范。我们利用antlr生成了符合CACT语法标准的词法和语法分析器，支持在listen和visitor两种模式下遍历AST，并实现了词法和语法错误处理功能。

分成员总结

彭睿思：

在本次实验中，我对测试失败的样例打印信息进行了整理，使用换行缩进方式替代了原输出的括号以表示推导，使得可读性进行了很好的提升，帮助了错误的发现和调试。在错误信息的整理过程中，我也对CACT语言规范和ANTLR工具有了更深入的认识和了解。

严哲虞：

通过本次实验，我对编译器的组织形式和工作方式有了初步的认识。通过阅读源码等方式，我对ANTLR4工具的接口和作用有了一定了解，尤其了解了vistor和listener模式的使用方式。在实验过程中，我也逐渐熟悉了CACT语言的规范，为后续实验打下基础。

游昆霖：

本次实验我主要负责了脚本和文法等文件的编写。通过编写脚本，可以支持多人在自己的电脑上协同开发，并高效的进行项目重新编译和样例批量测试，提高了实验效率。在文法和源文件的编写中，我更加明晰了CACT语言规范和ANTLR工具的编写规则，并初步掌握了CACT编译器在词法和语法分析阶段的实现细节和项目整体步骤流程。

赵若雯：

通过本次实验，我主要了解了CACT语言的文法规则，学习了巴科斯范式规范。在研读demo源码框架的过程中，我学习了antlr4工具的使用以及visitor、listener两种遍历模式的原理、区别和构建方法，同时也从对sample的测试中进一步体会了编译的整体流程。