# 编译原理第一次实验报告

小组成员：王琦琪 胡贤哲 李彦辰

## 课上练习

### 拉取仓库

由于一些未知的原因，5号服务器上已经存在了cact文件夹，我们也不敢乱动它。于是我们新建了文件夹demo-teacher,并在其中拉去仓库

```
mkdir demo-teacher
cd demo-teacher
git clone http://124.16.71.61:8000/Teacher/cact.git
```

### 使用antler4生成词法语法分析文件

在grammar文件夹下根据Hello.g4生成相关词法语法分析文件

```
cd cact/grammar
java -jar ../deps/antlr-4.13.1-complete.jar -Dlanguage=Cpp Hello.g4 -visitor -
no-listener
```

### 编译并运行测试

编译出可执行文件compiler

```
cd ..
mkdir -p build
cd build
cmake ..
make -j
```

注意到test.hello文件的内容是"hell maria"，将其改为"hello world"(实际文件内容没有引号)，然后执行compiler。

```
./compiler
```

# 课后实验

## 实验目标

### 一、Preliminary 前置知识

1. 了解扩展巴科斯范式(EBNF)规范，正确运用扩展巴科斯范式描述计算机文法。

扩展巴科斯-瑙尔范式(Extended Backus-NaurFor,EBNF)是一种用于描述计算机编程语言等正式语言的与上下文无关语法的元语法(metasyntax)符号表示法。简而言之，它是一种描述语言的语言。它是基本巴科斯范式(BNF)元语法符号表示法的一种扩展。

EBNF的基本语法形式如下，这个形式也被叫做production:

左式(LeftHandSide)=右式(RightHandSide).

左式也被叫做非终端符号，而右式则描述了其的组成。

终端符号:形成所描述的语言的最基本符号。所描述语言的标点符号(不是EBNF自己的)会被左右加引号(它们也是终端符号)，而其他终端符号会用粗体打印。

非终端符号:是用于描述语法的变量，它必须被定义在一个production中。或说，它们必须出现在某个地方的producfion的左式中。

```
扩展巴科斯范式EBNF:

 = : 定义

, : 串接

 ; : 终止
```

```
 | : 或

[...]: 可选，0或1次

 {...} : 重复，0或n次

 (...) : 分组

 '...' : 终结符

 "..." : 终结符，同 '...'

(*...*) : 注释，说明性文本，不表示任何语法

 ?...? : 特殊序列

\- : 排除，除去;
```

2. 熟悉ANTLR的安装和使用。

ANTLR是一个功能强大的解析器生成器，用于读取，处理，执行或翻译结构化文本或二进制文件。它被广泛用于构建语言、工具和框架。从语法中，ANTLR 生成一个解析器，该解析器可以构建和遍历解析树。使用Java语言编写，基于LL(*)解析方式，使用自上而下的递归下降分析方法。ANTLR v4极大地简化了匹配算术表达式语法结构的文法规则。对于传统的自顶向下的语法分析器生成器来说，识别表达式的最自然的文法是无效的，ANTLR v4可以自动地将左递归规则重写为非左递归等价物，唯一的约束是左递归必须是直接的。

3. 掌握ANTLR生成lexer和parser的流程。

ANTLR的语法识别过程一般分为词法分析和语法分析两个阶段：词法对应的分析程序叫做lexer，负责将符号（token）分组成符号类（token class or token type）。语法分析对应的程序叫做parser，根据词法，构建出一棵分析树（parse tree）或叫语法树（syntax tree）。

## 二、实验代码前的了解与准备

Antlr语法每个规则采用key:value的形式，分号结尾。词法分析器中变量名称始终以大写字母开头。语法解析器规则名称始终以小写字母开头。对于CACT语言规范的定义，需要根据ANTLR的语法规则进行适当调整。

**注意到的一些细节(与讲义不同点，非常重要！)**:

1. 起始规则是解析器首先使用的规则;它是语言应用程序调用的规则函数。描述整个输入文件的规则应引用特殊的预定义令牌**EOF**。反映在文档中，即讲义中CompUnit->CompUnit，写为compUnit: (decl | funcDef)+ EOF;
2. 表达式优先级在文法设计中的体现

在 Antlr 的规则文件中，越是前面声明的规则，优先级越高。所以，我们把关键字的规则放在 ID 的规则前面。算法在执行的时候，会首先检查是否为关键字，然后才会检查是否为 ID，也就是标识符。通过 文法的设计可以实现表达式的优先级。我们需要实现的优先级为

| 优先级 | 运算符 | 含义 | 结合方向 |
| --- | --- | --- | --- |
| 1 | [...] | 数组下标 | 左到右 |
| | (...) | 圆括号 | |
| 2 | - | 单目负号 | **右到左** |
| | + | 单目正号 | |
| | ! | 逻辑非 | |
| 3 | * | 乘法 | 左到右 |
| | / | 除法 | |
| | % | 取余 | |
| 4 | + | 双目加法 | |
| | - | 双目减法 | |
| 5 | <= | 小于等于 | |
| | >= | 大于等于 | |
| | < | 小于 | |
| | > | 大于 | |
| 6 | == | 等于 | |
| | != | 不等于 | |
| 7 | && | 逻辑与 | |
| 8 | \|\| | 逻辑或 | |

注：赋值、逗号不作为运算符。

1. 赋值语句用等号'='，不支持连续赋值。
2. 逗号使用范围受限，只出现在变常量声明、初始值、函数声明&调用。

具体实现见实现代码(cact.g4)中的line137-line162， 以及primaryExp,unaryExp和mulExp。

### 3. 数值常量的语法规则

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。

整型规则为

| 整型常量 | **IntConst** | → | **DecimalConst** \| **OctalConst** \| **HexadecConst** |
| --- | --- | --- | --- |
| 十进制常量 | **DecimalConst** | → | nonzero-digit \| digit |
| 八进制常量 | **OctalConst** | → | '0' \| OctalConst octal-digit |
| 十六进制常量 | **HexadecConst** | → | hexadecimal-prefix hexadecimal-digit \| HexadecConst hexadecimal-digit |
| 8位字符常量 | **CharConst** | → | """character""" |
| 十进制非零数字 | nonzero-digit | → | '1' \| '2' \| '3' \| '4' \| '5' \| '6' \| '7' \| '8' \| '9' |
| 十进制数字 | digit | → | '0' \| nonzero-digit |
| 八进制数字 | octal-digit | → | '0' \| '1' \| '2' \| '3' \| '4' \| '5' \| '6' \| '7' |
| 十六进制前缀 | hexadecimal-prefix | → | '0x' \| '0X' |
| 十六进制数字 | hexadecimal-digit | → | '0'...'9' \| 'a'...'f' \| 'A'...'F' |

浮点规则为

**Syntax**

*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**

具体实现见line202-line243。

4. 替换main函数中的错误处理过程

如图所示。

```
    std::any visitErrorNode(tree::ErrorNode *node) override {
        std::cout << "visit error node!" << std::endl;
        exit(1);
        //return nullptr;
    }
};
```

对于main.cpp中的错误节点，直接exit(1)即可。

### 测试与debug过程

本次实验多数g4代码可以直接照搬cact词法和语法规范的内容，所以基本上没有bug。测试流程例如：

```
int a = 0, b = 1;

int foo(int a, int b)
{
  return a * a;
}

int main()
{
  int c;
    c = foo(a, b);
  return c;
}
```

上面是测试用例22_true_func.cact。执行一下测试命令：

```
./compiler ../test/samples_lex_and_syntax/22_true_func.cact
```

得到输出：

```
enter rule [CompUnit]!
```

说明没有词法和语法错误。而应对以下用例例如：

```
int main()
{
    int a = 0x;
    return a;
}
```

此为测试用例01_false_hex_num.cact。运行测试命令后得到：

```
line 3:13 extraneous input 'x' expecting {',', ';'}
enter rule [CompUnit]!
visit error node!
```

检查出词法错误。

## 三、实验心得

王琦琪：

在本次实验中，我完成了CACT.g4和main.cpp文件的编写，并对代码进行了一部分debug。这是对理论课上学到的知识点一次实践，让我对CACT语言规范和ANTLR工具有了更深入的认识和了解。同时，main.cpp文件的编写也让我有契机学习面向对象编程里的继承关系。不过，我现在依然对编译知之甚少，很多东西都需要进一步的学习。

胡贤哲：

本次实验中，我对编译器进行词法和语法分析的过程和antlr代码的编写有了一定的了解，除给出的测试用例以外编写了一些测试用例进行测试和debug，但是由于没有区分好一些错误是否能在该阶段检查出来，导致测试过程朝错误的方做了一些无用功，同时也让我对编译器不同步骤的工作有了更深一步的了解，希望能在之后的实验中把这些错误的检查都实现出来。

李彦辰：
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。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。