

编译原理研讨课实验PR001实验报告

[author]: 高梓源 官奕琳 [Stu.Num]: 2019K8009929026 2019K8009907026

编译原理研讨课实验PR001实验报告

任务说明

成员组成

实验设计

设计思路

CACT语法规则

终结符

非终结符

错误捕获

错误获取函数寻找

关于错误获取函数

实验实现

规则文件完善

Ident标识符表示

单/双精度浮点常量

消除左递归

主函数修改

其它

多平台适配

相关debug过程

语义规则书写

main函数处理错误

环境安装过程

总结

实验结果总结

分成员总结

官奕琳

高梓源

附录

27个测试样例输出

任务说明

本次实验为CACT编译器设计的第一个实验，主要完成的任务如下：

- 实验环境的配置，包括：
 - 熟悉ANTLR4工具的安装和使用
 - 服务器和本地编译环境的搭建

- 完成CACT相关词法的设计
 - 根据CACT_specification.pdf完善CACT.g4，实现对语法产生式规则的描述
- 能够捕获词法分析阶段和语法分析阶段的错误并进行相应的输出处理
 - 若无错误返回0，存在错误返回非0值
- 进行编译测试，正确实现所有语法的识别

成员组成

- 高梓源，2019K8009929026
- 官奕琳，2019K8009907026

实验设计

设计思路

CACT语法规则

终结符

终结符中需要自行设计的为Ident, BoolConst, IntConst, FloatConst, DoubleConst

其中BoolConst过于简单，在此不再赘述；IntConst的表达在start code中也已详细给出，不再赘述；

- Ident标识符：
 - 标识符可以由大小写字母、数字以及下划线组成，但必须以字母或者下划线开头。
 - 由于必须以字母或者下划线开头，所以必须以IdentNondigit开头
 - IdentNondigit：表示下划线或各大小写字母，即 $_|a|b|\dots|z|A|B|\dots|Z$
 - 从第二个字符（若有）开始，标识符可以由大小写字母、数字以及下划线组成，即为 $[a-zA-Z_0-9]^*$

综上，标识符的正则表达式为： $Ident \rightarrow IdentNondigit(|a|b|\dots|z|A|B|\dots|Z)^*$

- 单/双精度浮点常量：

CACT中，浮点型字面值分为double与float类型，若字面值后面跟'F'或'f'后缀，则为float型，否则默认为double型。所以先考虑构造double类型的正则表达式。

有符号浮点型常数有如下几个类型：

- 普通形式的浮点数：其正则表达式为 $DoubleConst \rightarrow FractionalConst$
 - 浮点数由一串数字与一个小数点组成，小数点的前或后必须有数字出现，有如下三类情况
 - DigitSequence.DigitSequence
 - DigitSequence.
 - .DigitSequence
 - 合并可得

$$FractionalConst \rightarrow DigitSequence?'.' DigitSequence \mid DigitSequence'$$

- 指数形式：包括基数、指数符号和指数三个部分，且三部分依次出现
 - 基数：
 - 可带小数点：即为 *FractionalConst*
 - 可不带小数点：即为 *DigitSequence*
 - 指数符号和指数： *ExponentPart*
 - 指数符号为“E”或“e”
 - 指数部分由可带“+”或“-”（也可不带）的一串数字（0-9）组成，“+”或“-”（如果有）必须出现在数字串之前。
 - 据上描述，可得正则表达式

$$ExponentPart \rightarrow ('e'|'E')('+'|'-')?DigitSequence$$

- 综上，指数形式的正则式可以描述为

$$FractionalConst \mid ExponentPart \mid DigitSequence \mid ExponentPart$$

- 结合普通形式和指数形式的浮点数，可以得到double类型的正则表达式

$$DoubleConst \rightarrow FractionalConst \mid ExponentPart? \mid DigitSequence \mid ExponentPart$$

- 在CACT中，float类型即为double类型字面值后加"f" 或"F"，所以产生式为

$$FloatConst \rightarrow DoubleConst ('f'|'F')$$

非终结符

相比于CACT_specification中所给出的文法，我们在本次实验中集中修改了表达式部分的文法，主要是因为该部分文法中含有左递归。即使ANTLR允许直接的左递归，在仍未了解底层实现的情况下，默认还需要进行操作以消除左递归，为了让表达式更加直观明了，并且减小编译器工作量，对一些产生式中的左递归进行消除。

- 左递归消除的形式化过程如下：

$$A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \cdots \mid \beta_n$$

消除左递归得：

$$A \rightarrow \beta_1 A' \mid \cdots \mid \beta_n A' \mid A' \rightarrow \alpha_1 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

- 值得注意的是，CACT的各个表达式的产生文法还隐含了不同运算的优先级。每一个产生式的右部只可包含自己，或比自己优先级更高的非终结符。具体来说，优先级为：

一元运算 > 乘除模运算 > 加减运算 > 关系运算 > 相等性运算 > 逻辑与运算 > 逻辑或运算

可结合实验实现·规则文件完善部分的代码理解。

错误捕获

错误获取函数寻找

首先需要寻找获取错误信息的函数，可以进行全局搜索正则表达式`get(\w)*Error`，从而快速找到以下错误获取函数：

- `getTokenErrorDisplay`
- `getErrorDisplay`
- `getNumberOfSyntaxErrors`

前两者是在运行过程中对于词法语法unmatched情况进行即刻打印，因此仅剩的第三个正好是获取错误数量的函数，在语法、词法分析完成后进行调用判断即可。

关于错误获取函数

关于getNumberOfSyntaxErrors更详细的信息：

根据查找结果，在CACTLexer的父类Lexer中和CACTParser的父类Parser中分别找到了对应函数的实现：

```
1 // size_t Parser::getNumberOfSyntaxErrors()
2 size_t Lexer::getNumberOfSyntaxErrors() {
3     return _syntaxErrors;
4 }
```

_syntaxErrors是两个类中的私有变量，而产生错误对其进行修改的只有：

```
1 void Lexer::notifyListeners(const LexerNoViableAltException & /*e*/) {
2     ++_syntaxErrors;
3     // ...
4 }
```

继续回溯同一类中，调用notifyListeners的函数只有：

```
1 std::unique_ptr<Token> Lexer::nextToken() {
2     // ...
3     do{
4         // ...
5         try{
6             //...
7         } catch (LexerNoViableAltException &e) {
8             notifyListeners(e); // report error
9             // ...
10        }
11    } while (type == MORE);
12 }
```

希望获取下一个token即nextToken的调用，在同一类中只有std::vector<std::unique_ptr<Token>> Lexer::getAllTokens()，而该函数是一个末端函数，不存在调用者。而注意到nextToken是public的，因此很有可能被从外部访问，这部分讨论在实验实现部分展开。

因此，想要对_syntaxErrors进行修改只有通过notifyListeners这一函数，但注意到该函数设置为public，可能有外部调用时出现错误进而进行汇报的考量，但是封装性仍有值得商榷的地方，事实上并没有直接调用的案例，因此笔者更倾向于设置为protected甚至是private。

实验实现

规则文件完善

Ident标识符表示

```
1 Ident
2   : IdentNondigit [a-zA-Z_0-9]*
3   ;
4
5 fragment
6 IdentNondigit
7   : [a-zA-Z_]
8   ;
```

单/双精度浮点常量

```
1 DigitSequence
2   : Digit+
3   ;
4
5 FloatConst
6   : DoubleConst ('f' | 'F')
7   ;
8
9 DoubleConst
10  : FractionalConst ExponentPart?
11  | DigitSequence ExponentPart
12  ;
13
14 fragment
15 FractionalConst
16   : DigitSequence? '.' DigitSequence
17   | DigitSequence '.'
18   ;
19
20 fragment
21 ExponentPart
22   : ('e' | 'E') ('+' | '-')? DigitSequence
23   ;
```

消除左递归

■ 编译单元：

CACT_specification中所给出的文法为： $\text{CompUnit} \rightarrow [\text{CompUnit}] (\text{Decl} \mid \text{FuncDef})$

上述文法等价于 $\text{CompUnit} \rightarrow \text{CompUnit} (\text{Decl} \mid \text{FuncDef}) \mid \text{Decl} \mid \text{FuncDef}$

但该文法为左递归的，需要消除左递归，得到：

$\text{CompUnit} \rightarrow (\text{Decl} \mid \text{FuncDef})^+$

由于文件最后一定会有EOF表示文件结束，所以compUnit代码为：

```
1 compUnit
2     : (decl | funcDef)+ EOF
3     ;
```

- 乘除模表达式 MulExp:

CACT_specification中所给出的文法为: $MulExp \rightarrow UnaryExp \mid MulExp ('*' \mid '/' \mid '%')$
 $UnaryExp$

消除左递归:

```
1 1. MulExp  $\rightarrow$  UnaryExp MulExp'
2 2. MulExp'  $\rightarrow$  ('*' | '/' | '%') UnaryExp MulExp' |  $\epsilon$ 
```

而产生式2等价于

```
1 MulExp'  $\rightarrow$  (('*' | '/' | '%') UnaryExp)*
```

综上, $MulExp \rightarrow UnaryExp (('*' \mid '/' \mid '%') UnaryExp)^*$

对应antlr代码为

```
1 mulExp
2     : unaryExp ( ('*' | '/' | '%') unaryExp )*
3     ;
```

- 加减表达式 AddExp:

CACT_specification中所给出的文法为: $AddExp \rightarrow MulExp \mid AddExp ('+' \mid '-') MulExp$

消除左递归:

```
1 1. AddExp  $\rightarrow$  MulExp AddExp'
2 2. AddExp'  $\rightarrow$  ('+' | '-') MulExp AddExp' |  $\epsilon$ 
```

而产生式2等价于:

```
1 AddExp'  $\rightarrow$  (('+' | '-') MulExp)*
```

综上, $AddExp \rightarrow MulExp (('+' \mid '-') MulExp)^*$

对应antlr代码为:

```
1 addExp
2     : mulExp (('+' | '-') mulExp)*
3     ;
```

- 关系表达式 RelExp :

CACT_specification中所给出的文法为: $\text{RelExp} \rightarrow \text{AddExp} \mid \text{RelExp} (< \mid > \mid \leq \mid \geq) \text{AddExp} \mid \text{BoolConst}$

消除左递归后的antlr代码为:

```
1 relExp
2     : addExp (('<' | '>' | '<=' | '>=') addExp)*
3     | BoolConst
4     ;
```

- 相等性表达式 EqExp:

CACT_specification中所给出的文法为: $\text{EqExp} \rightarrow \text{RelExp} \mid \text{EqExp} (== \mid !=) \text{RelExp}$

消除左递归后的antlr代码为:

```
1 eqExp
2     : relExp (('==' | '!=') relExp)*
3     ;
```

- 逻辑与表达式 LAndExp:

CACT_specification中所给出的文法为: $\text{LAndExp} \rightarrow \text{EqExp} \mid \text{LAndExp} \&\& \text{EqExp}$

消除左递归后的antlr代码为:

```
1 lAndExp
2     : eqExp (('&&') eqExp)*
3     ;
```

- 逻辑或表达式 LOrExp:

CACT_specification中所给出的文法为: $\text{LOrExp} \rightarrow \text{LAndExp} \mid \text{LOrExp} \mid\mid \text{LAndExp}$

消除左递归后的antlr代码为:

```
1 lOrExp
2     : lAndExp (('||') lAndExp)*
3     ;
```

主函数修改

经过设计思路中的相关讨论,我们了解到只有通过notifyListeners这一函数才能够对语法错误次数进行修改。之所以要了解这一点,是因为需要确定在main.cpp插入错误获取函数最早的位置。此前我们通过回溯的方法进行,但事实上只能将搜索范围限定在本类,现在通过正向调用来clarify程序分析、错误产生的流程。

在main.cpp中,需要确定获取lexer和parser错误最早的准确位置,有以下可能的位置:

```

1  CACTLexer lexer(&input);
2  // get lex error pos 1
3  CommonTokenStream tokens(&lexer);
4  // get lex error pos 2
5  CACTParser parser(&tokens);
6  // get lex error pos 3
7  // get parser error pos 1
8  tree::ParseTree *tree = parser.compUnit();
9  // get lex error pos 4
10 // get parser error pos 2
11 tree::ParseTreeWalker::DEFAULT.walk(&listener, tree);
12 // get lex error pos 5
13 // get parser error pos 3

```

但是很快就能排除前三条语句，能够获取错误的时机一定是词法分析与语法分析进行完成，而对前三条语句，只需要查看派生类和父类的构造函数便可知初始化阶段只设置实例化类的相关参数，并不进行分析动作。以Lexer为例，派生类CACTLexer以及父类Lexer构造函数如下：

```

1  // class relation
2  class CACTLexer : public antlr4::Lexer
3  // CACTLexer constructor:
4  CACTLexer::CACTLexer(CharStream *input) : Lexer(input) {
5      _interpreter = new atn::LexerATNSimulator(this, _atn, _decisionToDFA,
        _sharedContextCache);
6  }
7  // LexerATNSimulator constructor:
8  LexerATNSimulator::LexerATNSimulator(Lexer *recog, const ATN &atn,
        std::vector<dfa::DFA> &decisionToDFA,
9      PredictionContextCache &sharedContextCache)
10 : ATNSimulator(atn, sharedContextCache), _recog(recog),
        _decisionToDFA(decisionToDFA) {
11     InitializeInstanceFields();
12 }
13 void LexerATNSimulator::SimState::InitializeInstanceFields() {
14     index = INVALID_INDEX;
15     line = 0;
16     charPos = INVALID_INDEX;
17 }
18 /*==== NO SIGN OF ANALYSIS ACTION =====*/
19 //Lexer constructor:
20 Lexer::Lexer(CharStream *input) : Recognizer(), _input(input) {
21     InitializeInstanceFields();
22 }
23 void Lexer::InitializeInstanceFields() {
24     _syntaxErrors = 0;
25     token = nullptr;
26     _factory = CommonTokenFactory::DEFAULT;
27     tokenStartCharIndex = INVALID_INDEX;
28     tokenStartLine = 0;
29     tokenStartCharPositionInLine = 0;
30     hitEOF = false;

```



```

31     channel = 0;
32     type = 0;
33     mode = Lexer::DEFAULT_MODE;
34 }
35 /*=== NO SIGN OF ANALYSIS ACTION ===*/

```

其余构造函数类似，只是设置private参数的作用

于是焦点来到了compUnit函数，事实上，通过简单的想法，walk函数一定是为最后输出语法分析树服务的，那么在此之前一定已经做好了词法、语法分析，那么也就是在4,2位进行错误检测即可。

但为了深入ANTLR，对compUnit进行进一步了解，我们直接指出核心点：

```

1  CACTParser::CompUnitContext* CACTParser::compUnit() {
2      _la = _input->LA(1);
3  }

```

于是就需要了解_input，该属性继承自父类Parser，对他的赋值如下：

```

1  // declaration
2  /*
3   * The input stream
4   * see also
5   * @getInputStream
6   * @setInputStream
7   */
8  TokenStream *_input;
9  // in construction function
10 Parser::Parser(TokenStream *input) {
11     InitializeInstanceFields();
12     setInputStream(input);
13 }
14 void Parser::setInputStream(IntStream *input) {
15     setTokenStream(static_cast<TokenStream*>(input));
16 }
17 void Parser::setTokenStream(TokenStream *input) {
18     _input = nullptr; // Just a reference we don't own.
19     reset();
20     _input = input;
21 }

```

可以总结为在构造函数中，Parser将tokens保存在_input私有变量中，事实上tokens的类CommonTokenStream继承自BufferedTokenStream，后者继承自TokenStream，因此此处运用了面向父类编程的思想进行抽象。

了解了_input之后，对_input调用的操作可以看作调用token的动作，那么此处LA()函数正是此意，于是锁定在BufferTokenStream类中函数：

```

1  size_t BufferedTokenStream::LA(ssize_t i) {
2      return LT(i)->getType();
3  }
4  Token* BufferedTokenStream::LT(ssize_t k) {

```

```

5    // ...
6    sync(i);
7    // ...
8 }
9 bool BufferedTokenStream::sync(size_t i) {
10     // ...
11     size_t fetched = fetch(n);
12     // ...
13 }
14 size_t BufferedTokenStream::fetch(size_t n) {
15     // ...
16     std::unique_ptr<Token> t(_tokenSource->nextToken());
17     // ...
18 }

```

LT函数中有一个很不起眼的关键词句`sync(i)`，通过它我们最终找到了`nextToken`函数的调用，这里`_tokenSource`字面意即位token处理的源头，事实上的确如此：

```

1  // class var declaration
2  protected:
3      /**
4       * The {@link TokenSource} from which tokens for this stream are fetched.
5       */
6      TokenSource *_tokenSource;
7
8  BufferedTokenStream::BufferedTokenStream(TokenSource *tokenSource) :
9      _tokenSource(tokenSource){
10     InitializeInstanceFields();
11 }
12 // Lexer
13 class ANTLR4CPP_PUBLIC Lexer : public Recognizer, public TokenSource

```

`TokenSource`事实上是接口(interface)，`Lexer`类对其进行了实现，因此此处体现了面向接口编程的思想，通过使用`nextToken`函数，按照设计思路处的分析，能够进行词法分析并且根据出错打印出错信息并且修改`_syntaxErrors`。

事实上语法分析也是在`compUnit`这个函数调用实现的，最终生成语法树，因此在此后进行错误获取是合理的。

具体设计时，若出错统一跳转到函数末尾，此外复用`errorNum`变量节省时间空间：

```

1  // main.cpp
2  int main(int argc, const char* argv[]) {
3      // ...
4      CACTLexer lexer(&input);
5      CommonTokenStream tokens(&lexer);
6      CACTParser parser(&tokens);
7      tree::ParseTree *tree = parser.compUnit();
8      SemanticAnalysis listener;
9      if (errorNum = lexer.getNumberOfSyntaxErrors()) {
10         std::cout << "[Error]> Lexer reported " << errorNum << " errors." <<
std::endl;

```

```

11         goto exit;
12     }
13     if (errorNum = parser.getNumberOfSyntaxErrors()) {
14         std::cout << "[Error]> Parser reported " << errorNum << " syntax errors."
<< std::endl;
15         goto exit;
16     }
17     // normal output
18 exit:
19     return errorNum;
20 }

```

之所以会将SemanticAnalysis listener;的声明提前在后面会提到。

其它

多平台适配

由于服务器无法使用super user创建ssh公钥，进行git提交极不方便，因此使用本地手段，由于使用MacOS Monterey而非Linux，开虚拟机不很方便，于是搭建Mac下ANTLR环境。

注意需要下载匹配版本的antlr-complete.jar和antlr-runtime环境，最好是和Linux中相同的4.8版本（4.10版本测试失败），按照STFW获取的方法进行安装，需要能够使用antlr4命令。

antlr-runtime相关文件包括预编译需要的头文件，运行时动态链接的库文件，后者在linux下是.so结尾，Mac下为.dylib结尾，并且由于仍然未能解决的cmake问题，在运行时Mac无法找到本来添加至路径的动态链接，而是固定的访问几个路径下的对应文件，笔者暂时放置在/usr/local/lib/目录下。

而后需要做的就是修改cmake，使其能够根据运行平台自动适配，识别平台的常量是\${CMAKE_SYSTEM_NAME}，Linux下值为Linux，目前Mac下值为Darwin，使用比较操作STREQUAL即可判断平台，将两者runtime相关文件放在不同之处，include_directories操作时进行修改即可，制定编译器根据机器而异。详见根目录下[CMakeLists.txt]

相关debug过程

语义规则书写

- 经历过最多的bug是关于继承属性和综合属性的locals少加入，课件中指出constExp需要加入int basic_or_array_and_type，事实上还有number，constInitVal需要加入；
- 此外的问题还有Parser终结符首字母大写，Lexer非终结符首字母小写；
- fragment的使用也曾带来问题，fragment相当于C语言inline函数，起到直接替换、使代码易读的作用，并不是单独的终结符。

main函数处理错误

若在此之前讨论的2,4处使用goto跳转到结尾进行错误处理返回，那么会跳过SemanticAnalysis listener;的声明，这在cpp中不被允许，报错。

按照面向过程的思想，尽早进行错误判断能避免资源的浪费和开销。但是在OOP，需要类的良好排布和预编译的内存分配支持，因此不便如此考虑，空间不会被节省，因此可以后移到声明之后进行错误处理。

环境安装过程

经过一系列错误才能正确使用mac进行实验，首先是CMakeLists.txt中\${CMAKE_SYSTEM_NAME}变量并非任何时候都可以使用，需要先对PROJECT进行配置。在配置PROJECT时，服务器Linux环境下无法识别HOMEPAGE_URL字段，未找到原因与解决办法，遂删除。

总结

实验结果总结

按照上述内容完成的语法规则文件和编译器源码，经过编译、测试，对于公开的27个测试样例能够正确通过，res/log.txt中error log为空，结果见附录，可见能够正确识别程序中的语法，对于错误也能过正确报告并退出。

通过实验，我们掌握了ANTLR的基本使用方法，感觉到它的强大。在寻找接口的同时通过阅读Antlr生成的词法、语法分析器的源码，并参考其文档，我们大致理解了它们的工作流程，为以后进行更深层次的调用完善奠定基础。

分成员总结

官奕琳

完成了CACT语法规则的录入，完善CACT.g4并撰写相应部分实验报告。通过对语言进行准确的语法定义，更深入的了解了值得玩味的语法规范。而debug过程根据位置报错对连锁的产生式都需要进行修改，考验耐心细致程度，为以后更复杂的情况做铺垫。

高梓源

主要完成了mac环境的搭建、main.cpp中错误处理、运行脚本及测试调试相关工作，以及撰写相应部分的实验报告。此番实验对于编译环境的理解和代码阅读的能力提升非常大，上学期面向对象课程所学又派上了用场。

OOP是好文明

附录

27个测试样例输出

请按照项目根目录README.md进行编译运行，结果见res/pass_out.txt