编译原理研讨课实验PR001实验报告

任务说明

本次实验主要包含以下内容：

1. 熟悉 Antlr 的安装和使用：安装 Antlr，了解如何编写文法文件并生成词法分析器和语法分析器；
2. 完成词法和语法分析：根据 CACT 规范编写文法文件，通过 Antlr 生成分析器，对 CACT 源文件进行词法和语法分析，修改默认的错误处理机制

成员组成

孙彬：词法分析部分

郑天羽：语法分析部分

代瀚堃：修改错误处理机制、调试

实验设计

设计思路

1. 词法分析

   阅读 CACT 文法规范，比对已有代码，Lexer 的浮点数部分仍需完善。根据CACT规范自行设计浮点型常数并添加到 .g4 文件中

2. 语法分析

   阅读CACT文法规范，在.g4文件中补全已有代码，加入函数声明、函数定义、各种表达式等识别内容。

3. 错误处理

   Antlr 在 grammar 目录下生成了访问节点的接口声明，在 src 目录的 semanticAnalysis 文件中实现这些接口，就能定义进入和退出一个语法树中节点时所做的语义动作。其中，`visitErrorNode` 方法在访问错误节点时触发，在该方法中调用 `exit` 函数，以实现在遇到词法和语法错误时返回非零值的要求

实验实现

1. 词法分析

   根据CACT语言规范，浮点常量分为单精度(float)和双精度(double)两种类型，他们的区别在于单精度有f或F的后缀，没有则默认为双精度。因此我们可以根据后缀来区别二者：

   ```
   FloatConst
       : DoubleConst FloatSuffix
       ;

   fragment
   FloatSuffix
       : 'F' | 'f'
       ;
   ```

   而双精度浮点常量又有2种形式，小数形式和科学计数法形式。前者必须含有小数点且必须存在数字，后者则必须有指数部分。
   当然我们可以就按这2种形式划分，但是考虑到科学计数法形式允许基数不含小数点，我们可以选择更简洁的划分：

   ```
   DoubleConst
       : DecimalFraction Exponent?
       | Digit+ Exponent
       ;

   fragment
   DecimalFraction
       : Digit+ '.' Digit+
       | '.' Digit+
       | Digit+ '.'
       ;
   ```

   指数部分`Exponent`按照CACT规范，含有'e'或'E'的标识，指数可以带有符号:

   ```
   fragment
   Exponent
       : ExponentPrefix Sign? Digit+
       ;

   fragment
   ExponentPrefix
       : 'E' | 'e'
       ;

   fragment
   Sign
       : '+' | '-'
       ;
   ```

2. 语法分析

将CACT语法规范中的条目翻译成正则表达式即可，但要注意两者使用的符号不同。原文中使用的符号， [...] 表示方括号内包含的为可重复0次或1次的项，{...} 表示花括号内包含的为可重复0次或多次的项。在正则表达式中对应的表达方法为 (...)? 与 (...)* , 例如：

FuncDef → FuncType Ident '(' [FuncFParams] ')' Block

就可翻译为

```
funcDef
    : funcType Ident '(' (funcFParams)? ')' block
    ;
```

另外，

FuncFParams → FuncFParam { ',' FuncFParam }

可翻译为

```
funcFParams
   : funcFParam (','funcFParam)*
   ;
```

3. 错误处理

   查看 main.cpp 中的执行流可以得知，调用 `walk` 方法时从根 `*tree` 开始，遍历语法树中的节点，而 `walk` 方法在 ParseTreeWalker.cpp 中定义，当判断一个节点为 `ErrorNode` 时，在 listener 模式下会调用 `visitErrorNode` 方法访问该节点，可以推断，错误处理是在该函数中完成的，该函数在 CACTBaseListener.h 中声明为虚函数，其重载在 semanticAnalysis.cpp 中，我们仿照该文件中其他访问节点的方法，实现该函数，在其中调用 `exit` 函数，返回值为1即可完成本次实验的错误处理，代码如下：

   ```
   #define ERROR 1
   void SemanticAnalysis::visitErrorNode(antlr4::tree::ErrorNode * node){
     exit(ERROR);
   }
   ```

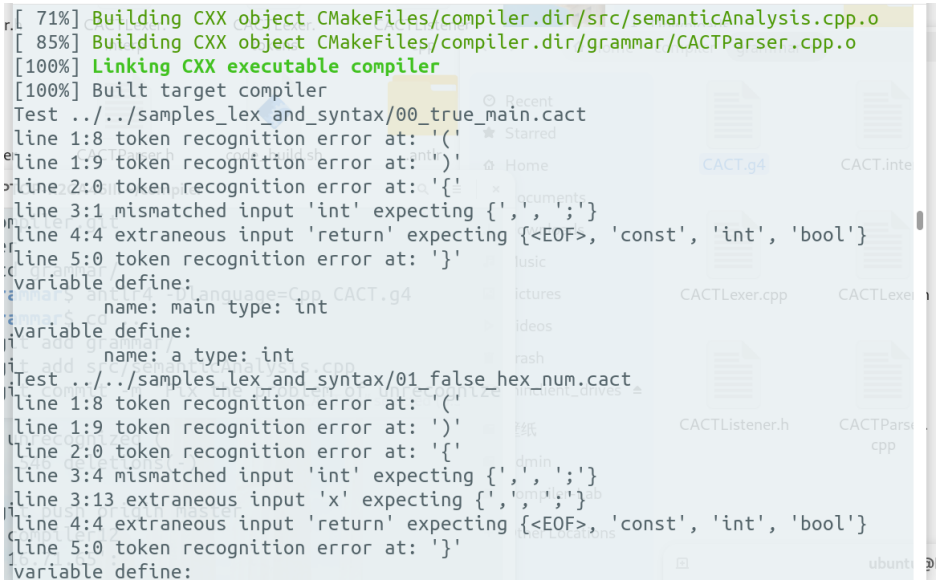
   并在头文件中修改其声明。

其它

调试

各小组成员提交自己的代码后，我们进行了测试，通过用例 21/27，我们认真比对了 CACT 文法规范中的产生式和 .g4 文件中的规则，发现有几条语法规则表示错误，改正后提交，重新在 material 目录下 clone 和测试，仍然只通过了 21 个用例，我们反复比对语法规范仍无法找出错误，也尝试参考 Antlr 的 Github 仓库中提供的 C 文法，都未成功。而单步调试时，由于 Antlr 的代码过于庞杂，短时间内难以读懂，也只好作罢。于是我们决定从发现语法/文法错误时输出的信息入手来分析。

查看终端打印的错误信息，发现无论我们怎么改，都无法识别函数定义，在读到左括号时报错：



其中 variable define 这一行是在 `exitVarDecl` 方法中打印的，也就是说，分析器误把 main 函数的定义当成了变量声明，我们尝试把第一个产生式中的 `decl` 删去，这样一来，分析器将不能再识别变量声明。然而重新测试，这一错误仍然存在，所以我们怀疑分析器可能没有更新，这时才想起来在编译前需要用 `antlr4` 命令重新生成分析器。重新提交 grammar 目录下的各个文件，再次测试，通过了 27 个测试用例。

总结

实验结果总结

在本次实验中，我们了解了 Antlr 的使用，编写 .g4 文件并利用 Antlr 生成符合 CACT 文法规范的词法、语法分析器，修改了默认的错误处理机制，最后通过了各个测试用例。

在调试阶段遇到用例不通过时，我们首先想到的是 .g4 文件中的规则可能编写错误，于是只局限于此，一直反复 commit 和测试，都未能解决问题，而忽略了 commit 前需要重新生成分析器这一重要的步骤，花费了不少时间来调试，也算是为我们之后的实验提了个醒。

分成员总结

1.孙彬

如何把产生式设计得简洁明了是我在这次实验里收获的经验，当然你也可以一股脑把所有的表达式全都写进去，但那样就会显得比较臃肿且思路混乱，通过宏定义的形式，我们能够用简洁的符号代表复杂的形式，这在之后的实验中也能起到作用，尤其是在小组成员相互沟通的时候就显得更为重要。

2.郑天羽

编写语法分析部分只需要按照CACT规范，将对应的条目翻译成正则表达式即可，唯一需要注意的地方是parser部分的产生式头需要小写字母开头。

3.代瀚堃

修改错误处理在实现上比较简单，需要读懂如何利用 Antlr 生成的接口访问节点，定义其中的动作代码，实现正确的错误处理机制。