北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

本科生《编译原理》课程实践报告

题　　目： 语法分析实验

学　　院： 徐特立学院

专业名称： 计算机科学与技术

姓　　名： 陈照欣-1120191086

实验目的

1. 熟悉 C 语言的语法规则，了解编译器语法分析器的主要功能；
2. 熟练掌握典型语法分析器构造的相关技术和方法，设计并实现具有一定复杂度和分析能力的 C 语言语法分析器；
3. 了解 ANTLR 的工作原理和基本思想，学习使用工具自动生成语法分析器；
4. 掌握编译器从前端到后端各个模块的工作原理，语法分析模块与其他模块之间的交互过程。实验内容该实验选择 C 语言的一个子集，基于 BIT-MiniCC 构建 C 语法子集的语法分析器，该语法分析器能够读入词法分析器输出的存储在文件中的属性字符流，进行语法分析并进行错误处理，如果输入正确时输出 JSON 格式的语法树，输入不正确时报告语法错误。

实验过程与方法

1. 概述

   本实验使用了两种方法实现了语法分析器，分别是基于 ANTLR 自动生成工具和修改 ExampleParser。对于 ANTLR，首先是根据.g4 文件的格式编写语法以及词法规则 MyCGrammar。通过工具将文件进行编译得到词法分析器 MyCGrammarParser 以及 MyCGrammarListener 等。同时为了使生成的 json 文件能够接近于实验要求中给出的 json 格式，我修改了 MyCGrammarListener，使得生成的语法树更接近于 AST。对于 ExampleParser 的修改，我设计了另一套较为简单的文法，同时消除了左递归和左公因子，使其更符合 LL(1)文法。

2. ANTLR 工具介绍

   ANTLR 是基于 LL 算法实现的语法解析器生成器，用 java 语言编写，采用的

是自上而下的递归下降 LL 解析器方法。如同一般的词法分析器和语法分析器，ANTLR 可以用来产生树状分析器，ANTLR 文法定义使用类似 EBNF 的定义方式，比较形象，十分简洁直观。ANTLR 本身是使用 switch-case 来匹配 token，形成记号序列记号流，而旧式的 Yaac 则利用符号表。ANTLR 是完全 exception-driven，LL（K）语法比目前流行的 LR 解析器要更强大，更可以有效的避免 LR 解析器有的归约移位冲突，以及移位移位冲突。产生的代码更清晰。

3. 配置框架

倘若使用 ANTLR，则难点之一为语法分析器的嵌入。框架执行的逻辑顺序为 BitMiniCC.java 作为主类，调用 MiniCCompiler.java 根据 config.xml 的内容，选择在每个阶段调用不同的方法。由于 ANTLR 直接生成的 MyCGrammarParser 中没有 run 方法，且需要调用新的 MyListener，我新建了一个 MyParser.java 作为过渡，在其中完成了对两个文件的调用。

4. 文法：

文法基于 C99 文法标准编写，删减了部分不常用的语法格式，同时消除了左递归和公共左因子，文法包括全局变量声明，循环语句、分支语句、函数调用语句以及 switch 语句等。具体文法可见 MyCGrammar.g4 文件。

5. AST 生成

通过我们文法规则生成的语法树(Parser Tree)，也可以被叫做 CST(Concrete Syntax Tree)——它非常“忠实”(concrete)地还原了我们的构造的文法规则。CST 完全对应于文法规则，每一节点的分叉少但树的深度大，冗余信息多。AST 是 CST 的简化，每一个节点分叉多，树更为扁平，使得后

续过程能方便的获取更多信息。根据框架中已经提供的 AST 节点，我们将 AST 节点互相连接，生成一个"JAVA 对象构成的 AST 树"，最后将这棵树的根节点送入 jackson 中自动生成 json 字符串并进行输出，所得结果即为所需。

为了将 AST 节点相互连接，我们需要以某种方式遍历 antlr 的语法树，并在遍历语法树的过程中按某种规则挂载 AST 节点，构造对应的抽象语法树。

算法步骤如下：

Step1：进入语法树的节点时，新建对应的 AST 节点，将 AST 节点推入栈中

Step2：离开语法树本节点时，对应 AST 节点出栈（并维护一个对它的引用）。将本 AST 节点挂载到栈顶的节点（即父语法节点）对应位置。

进入节点和离开节点分别对应于 listener 中文法的每一个节点的 Enter 和 Exit 函数。AST 父语法节点的选择由 AST 父类决定，主要根据 AST 类中定义的参数。AST 的节点主要包括：程序定义 ASTCompilationUnit，声明语句 ASTDeclaration，函数定义 ASTFunctionDefine，声明 ASTDeclarator，表达式 ASTExpression，句子 ASTStatement。其中 ASTDeclarator，ASTExpression，ASTStatement 为抽象类，被其它具体的节点所继承。具体文法见 MyCGrammar.g4。

6. ExampleParser 的编写

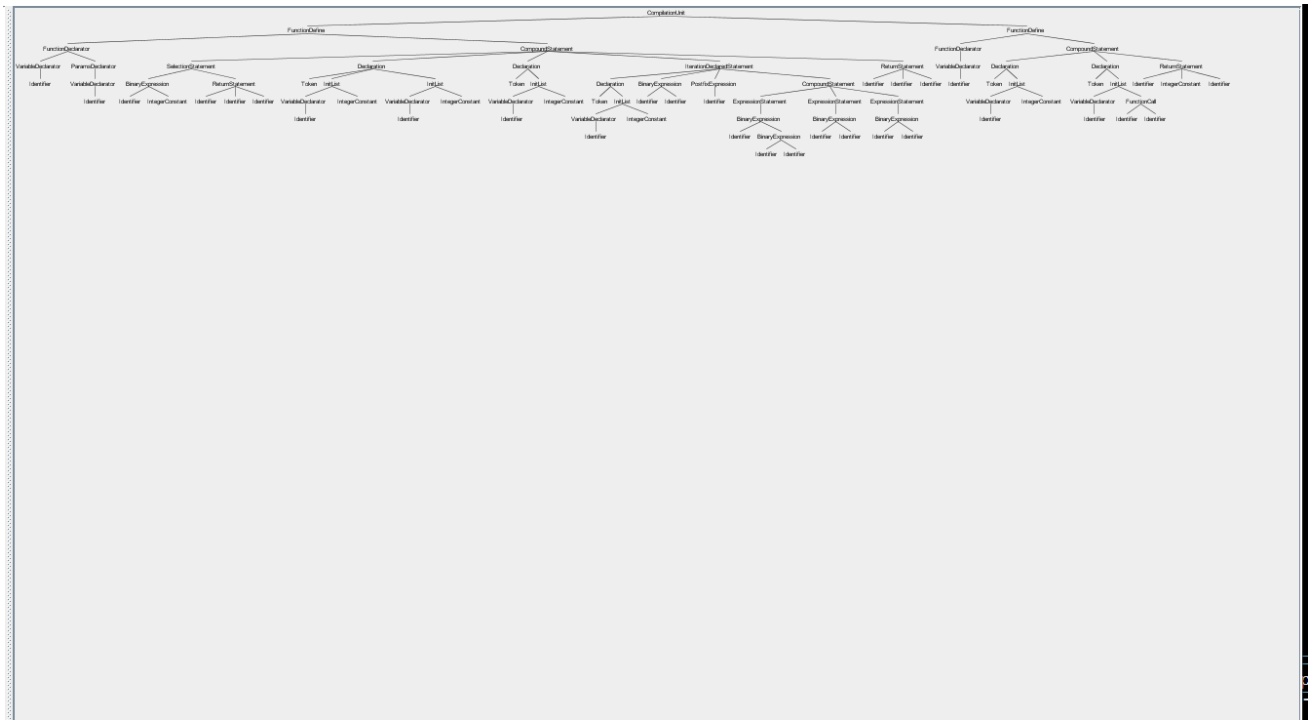
ExampleParser 的编写主要工作除了创建文法外，还在于左递归的消除和左因子的提取，为递归下降分析器的设计提供了一定便利，但由于实验要求实现的语法树的节点定义与设计的文法存在一定的差异，最终生成的语法分析树和分析文件必须按照实验要求设计，故编码设计完成的对非终结符进行匹配的函数可能与递归下降分析器中的分析子程序存在一定差异。

实验结果

测试样例 1:

```
int fibonacci ( int n ) {
    if ( n < 2 )
        return n;
    int f = 0 , g = 1 ;
    int result = 0 ;
    for ( int i = 0 ; i < n ; i ++ ) {
        result = f + g ;
        f = g ;
        g = result ;
    }
    return result ;
}
int main ( ) {
  int a = 10;
  int res = fibonacci ( a ) ;
  return 0 ;
}
```

测试结果:



编译器运行结果如下：

```
D:\Java\jdk1.8.0_321\bin\java.exe ...
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
Start to compile ...
2. LexAnalyse finished!
3. Parsing Finished!
Compiling completed!

Process finished with exit code 0
```

Json 文件部分截图如下：

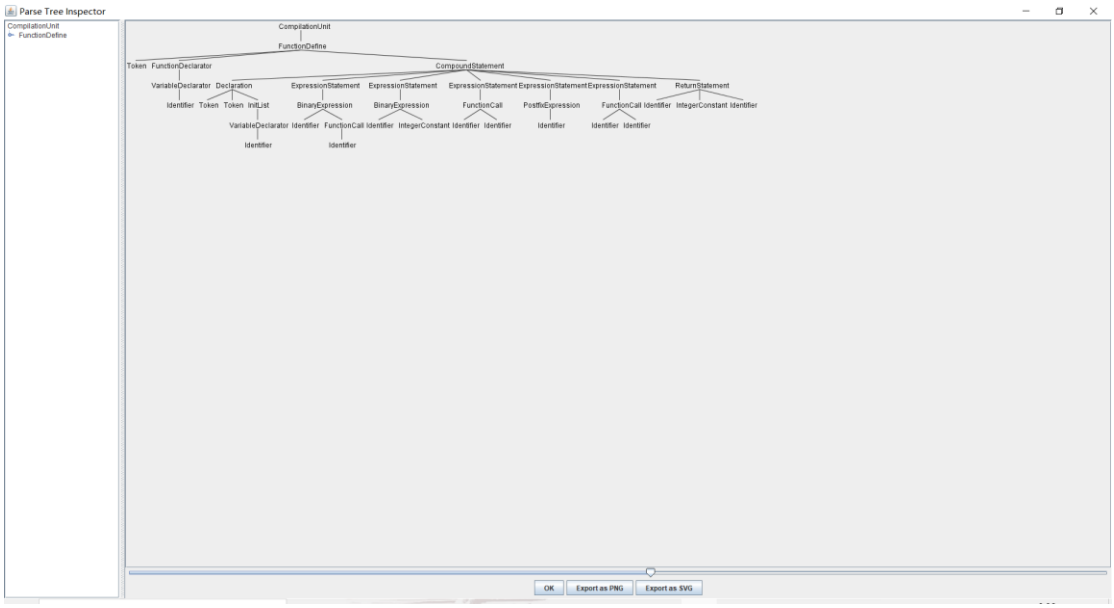
具体结果见附件

测试样例 2:

```
int main() {

  int a;


  a = MARS_GETI();

  a *= 10;
```

```
  MARS_PUTI(a);
  a ++;
  MARS_PUTI(a);
  return 0;
}
```

测试结果

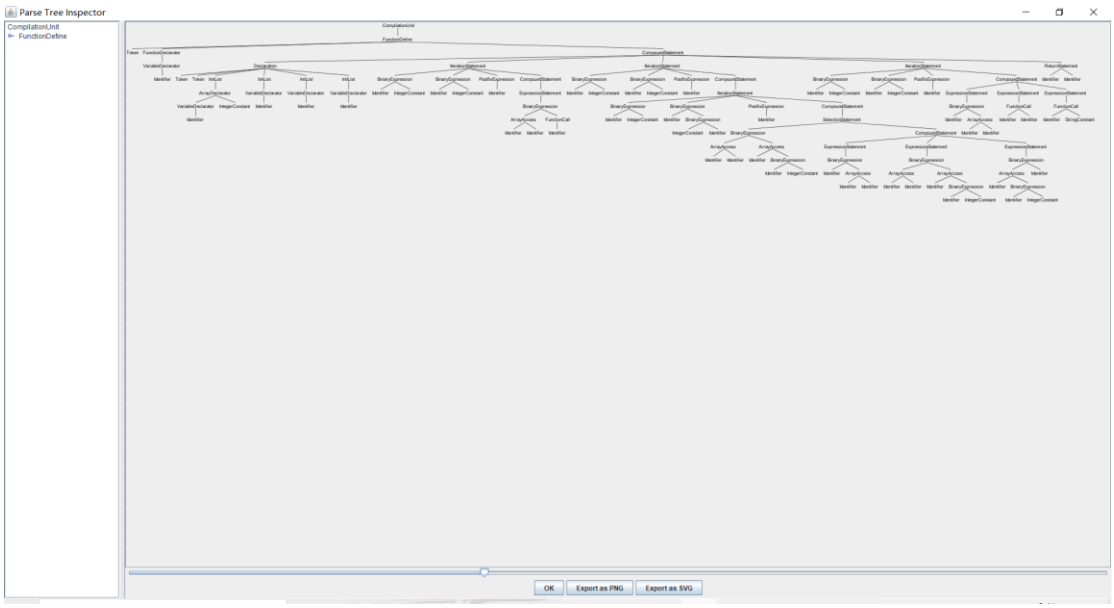


测试样例 3:

```
int main() {
  int a[5], i, j, t;
  for(i = 0; i < 5; i++) {
    a[i] = MARS_GETI();
  }
  for(i = 0; i < 4; i++) {
    for(j = 0; j < 5 - i; j++) {
      if(a[j] > a[j+1]) {
```

```
        t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
      }
    }
  }
  for(i = 0; i < 5; i++) {
    t = a[i];
    MARS_PUTI(t);
    MARS_PUTS("\n");
  }
  return;
}
```

测试结果:



实验心得

在选择这次实验具体采用的分析方法时，笔者是非常矛盾的。如果直接采用 antlr 生成语法分析器，则不能充分利用之前编写的词法分析器；又比较缺乏构建 LL、LR 分析器的经验，难以将这类分析器很好地嵌入到实验的框架中；递归下降分析法主体思路成熟，但具体到细节的分析又显得复杂，并非是一种通用的分析方法。幸而时间较为充沛，使用了两种不同的方法完成了实验，较为完整地实现了语法分析的过程，并将分析器完美嵌入到框架中。

同时，通过阅读 C 语言标准文法，我对于 C 语言的使用方法又有了一些不同的心得和理解，希望以后能够更好地运用到实际编程中。