编译原理 Lab5: 语法分析实验

郑子帆 1120200822

北京理工大学 计算机学院 07112002 班

日期：2023 年 5 月 4 日

摘　　要

本文为北京理工大学《编译原理与设计 2023》课程的 Lab5 实验报告。在本文中，我们对于简单的 C 语言程序设计了一个精简的文法，并在 Bitminicc 中编写了 MyParser 类，实现了以通过 Scanner 得到的属性字符流为输入、通过递归下降法分析的语法分析。本实验最终我们可以对于 parse_test 中的 C 程序进行语法分析并得到一个表示抽象语法树的 JSON 文件。

注：因为本人五一放假期间不幸感染新冠，一直高烧不退，直到此实验 ddl 当天仍然发烧 39.5 度，所以这次实验主要就是完成了一个可以得到正确的 JSON 文件的代码，报告并没有写的很详细，请评阅老师见谅。

1　实验简介 [1]

1.1　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。

1.2　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。

2 实验过程

在本次实验中，我对比了可以成功完成本实验的几种方法。

对于 ANTLR4 而言，我们可以轻松的得到一个语法分析器的 java 代码，但是其生成的 CST(Concrete Syntax Tree) 需要用很麻烦的方法转化为 AST(Abstract Syntax Tree)，所以最后若想要生成的 JSON 文件与闭源生成的一样，需要使用 ANTLR 中的 Listener 或 Visitor 遍历整棵树 [3]。

而对于直接扩写 *ExampleParser.java*，因为示例程序已经给出了一些 AST 节点类的使用方法，仅需要“照葫芦画瓢”一下就可以生成我们想要的 AST(当然代码量还是很大的)。

所以通过比较上述两种方法，我最后选择了后者。即，设计一个足够能识别 parse_test 文件夹中的 C 程序的文法，然后新建一个 MyParser 类，依照 *ExampleParser.java* 中的写法将设计好的文法进行代码编写，最后测试结果。

2.1 设计文法

虽然在 Lab4[2] 中我们设计了一个在 C11 标准下的文法，但是由于该文法太过于复杂，不方便进行修改和接下来程序的编写，故我们在本实验中设计了一个新的、精简版的文法。

首先我们对于 parse_test 中的测试程序进行分析，列出这些程序所用到的有关于 C 语言的语法和特性，大致如下：

- 声明语句
- 表达式语句
- 选择语句 (在这里仅支持 if 即可)
- 循环语句 (在这里仅支持 for 即可)
- 返回 (return) 语句
- 赋值语句

对于声明语句，还有如下几个细节：

- 全局普通、一维数组变量声明
- 局部普通、一维数组变量声明
- 声明时支持初始值赋值，初始值可以是常数、表达式、函数返回值、变量。
- 函数的声明

对于表达式，我们需要注意运算符 (因为测试程序中不涉及单目运算符，故此实验不考虑单目运算符) 的优先级。如"[]", "++" 的优先级高于"*", "/"；"*", "/" 的优先级高于"+", "-" 等等。

根据上述的总结，我们可以得到一个精简版的文法的“初稿”。因为我们用到的语法分析方法是**递归下降法**，所以还需要处理一下“初稿”中左递归和左公因子的问题

对于左递归的消除，我们以下面这一部分文法为例：

```
exp_postfix -->
      exp_postfix '[' exp ']'
    | exp_postfix '(' exp ')'
    | exp_postfix '(' ')'
    | exp_postfix '++'
    | exp_postfix '--'
    | exp_unit
```

可以发现这是很明显的左递归。使用课上学习过的方法消除左递归，即可得到下面的文法：

```
exp_postfix -->
      exp_unit exp_postfix1
exp_postfix1 -->
      e
    | '[' exp ']' exp_postfix1
    | '(' exp ')' exp_postfix1
    | '(' ')' exp_postfix1
    | '++' exp_postfix1
    | '--' exp_postfix1
```

当然，“初稿”中的含有左公因子的情况比左递归多了很多，以下面这一段文法为例：

```
arg_list -->
      ctype declarator ',' arg_list
    | ctype declarator
```

我们可以将 ctype declarator 提取出来，得到下面提取左公因子后的文法：

```
arg_list -->
      arg ',' arg_list
    | arg
arg -->
      ctype declarator
```

在完成上述处理后，我们可以得到最终的“定稿”文法，完整的文法详见A。

2.2 代码编写

我们仿照 *ExampleParser.java*，创建了 MyParser 类。

2.2.1 错误纠正

首先我们发现，对于成员函数 *loadTokens*，我们发现对于 ScannerToken 类的实例对象的 lexme 属性的赋值有错误，源代码中会在正确的 lexme 属性值上外加一对引号。对于解决方案，把 *substring* 中的头、尾参数各向里少取一位即可。更改后的代码如下：

```
st.lexme = segs[1].substring(
    segs[1].indexOf("=") + 2,segs[1].length() - 1);
st.type  = segs[2].substring(
    segs[2].indexOf("<") + 1, segs[2].length() - 1);
```

2.2.2 写入 JSON 文件

我们首先调用 jackson 包中的 ObjectMapper 类，创建一个实例化对象用于将 AST 树写入指定的文件路径名中，具体如下述代码：

```
ObjectMapper mapper = new ObjectMapper();
mapper.writeValue(new File(oFile), root);
```

2.2.3 AST 的可视化

导入 ANTLR 中的 TreeViewer 类，进行可视化：

```
String[] nonsense = new String[20];
TreeViewer Tviewer = new TreeViewer(Arrays.asList(nonsense), root);
Tviewer.open();
```

2.2.4 文法转换

这部分其实就是“照葫芦画瓢”，一下对于部分成员函数进行展示。

对于文法 stmt::=code_block | select_stmt | iteration_stmt | return_stmt | exp_stmt，编写的代码如下；

```
public ASTStatement stmt() {
    nxtToken = tknList.get(tokenIndex);
    if(nxtToken.type.equals("'{'")) {
        return code_block();
    }
```

```
    else if(nxtToken.type.equals("'if'")) {
        return select_stmt();
    }
    else if(nxtToken.type.equals("'for'")) {
        return iteration_stmt();
    }
    else if(nxtToken.type.equals("'return'")) {
        return return_stmt();
    }
    else{
        return exp_stmt();
    }
}
```

对于文法 iteration_stmt::='for' '(' exp ';' exp ';' exp ')' stmt | 'for' '(' decl ';' exp ';' exp ')' stmt，编写的代码如下：

```
public ASTStatement iteration_stmt() {
    matchToken("'for'");
    matchToken("'('");

    if(pd_ctype(tknList.get(tokenIndex).type)) {^^I// 走 decl
        ASTIterationDeclaredStatement is = new ASTIterationDeclaredStatem
        ASTDeclaration id = decl();
        LinkedList<ASTExpression> iter2;
        LinkedList<ASTExpression> iter3;

        iter2 = new LinkedList<>(exp());
        matchToken("';'");
        iter3 = new LinkedList<>(exp());
        matchToken("')'");
        ASTStatement s = stmt();

        is.init = id;
        is.cond = iter2;
        is.step = iter3;
        is.stat = s;
        is.children.add(id);
```

```
        is.children.addAll(iter2);
        is.children.addAll(iter3);
        is.children.add(s);

        return is;
    }
    else {
        ASTIterationStatement is = new ASTIterationStatement();
        LinkedList<ASTExpression> ie;
        LinkedList<ASTExpression> iter2;
        LinkedList<ASTExpression> iter3;

        ie = new LinkedList<>(exp());
        matchToken("';'");
        iter2 = new LinkedList<>(exp());
        matchToken("';'");
        iter3 = new LinkedList<>(exp());
        matchToken("')'");
        ASTStatement s = stmt();

        is.init = ie;
        is.cond = iter2;
        is.step = iter3;
        is.stat = s;
        is.children.addAll(ie);
        is.children.addAll(iter2);
        is.children.addAll(iter3);
        is.children.add(s);

        return is;
    }
```

完整的代码详见 src 文件夹中的 *MyParser.java* 文件。

3 实验结果

3.1 配置 *config.xml*

配置 *config.xml*，具体内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<config name="config.xml">
  <phases>
    <phase>
      <phase skip="true" type="java" path="" name="preprocess" />
      <phase skip="false" type="java" path="" name="scan" />
      <phase skip="false" type="java" path="bit.minisys.
      minicc.parser.MyParser.java" name="parse" />
      <phase skip="true" type="java" path="" name="semantic" />
      ...
    </phase>
  </phases>
</config>
```

注意，因为我们需要用到词法分析器输出的 tokens 文件作为语法分析器的输入，所以我们要设置 skip=false。

3.2 AST 的可视化展示

下图截选自对于测试程序 *3_parser_test2.c* 的语法分析。

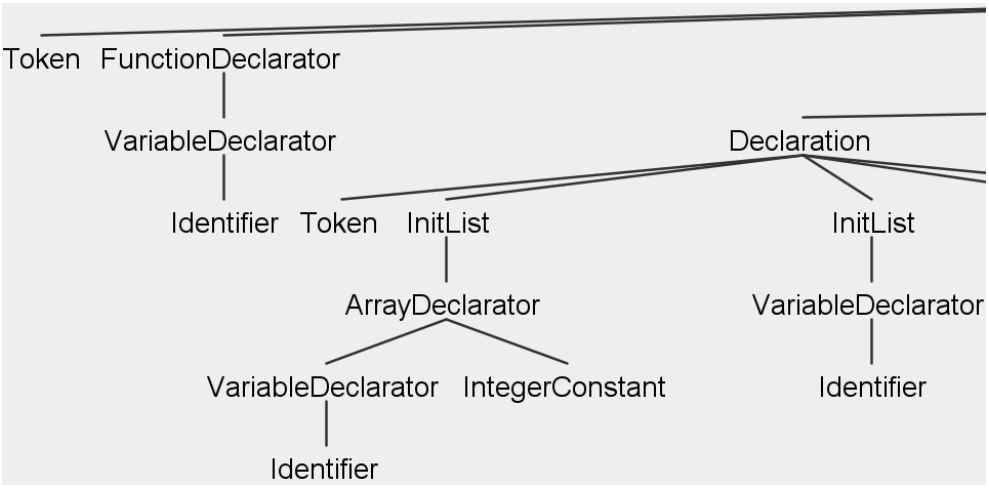

图 1: *3_parser_test2.c* 的抽象语法树 (截选)

3.3 JSON 文件内容对比

因为生成的 JSON 文件在 IDEA 中打开后内容为一行，所以我们需要对 JSON 文件中的内容进行格式化，然后再将其与 Bitminicc 中的闭源语法分析器得到的 JSON 文件内容做对比。

具体地，我们用网站https://www.bejson.com/进行 JSON 文件内容的格式化，用网站https://www.jq22.com/textDifference来比较 MyParser 和闭源语法分析器的结果。

为了实验方便，我们将 4 个测试程序放在一起，合成一个 *overall_parser_test.c* 文件，这样我们仅需比较一个即可。具体的结果截图如下：

图 2: *overall_parser_test.c* 的 JSON 格式化

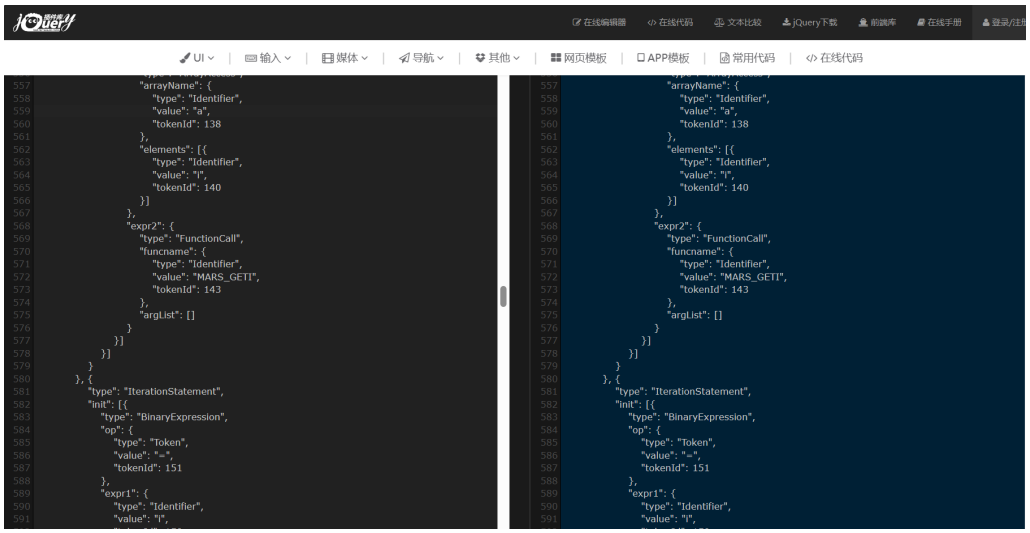

图 3: *overall_parser_test.c* 的 JSON 内容对比

可以看到，我们实现的 MyParser 和闭源程序得到的结果一样，实验圆满完成。

4 实验心得与体会

在本次实验中,我从温习 Lab4 的语法规则,到消除左递归,再到扩展 ExampleParser,构造我自己的语法分析器，过程中学到了很多。不仅将课堂上学习的消除左递归方法运用到了 C 语言的实际问题中，还在学习和编写代码的过程中收获了举一反三能力的提高。在细心琢磨语法规则，运用相应的 AST 类的接口来实现分析器时，我也苦于其复杂性，并且，在五一小长假期间，我感染了新冠，持续的高烧也让完成实验变得遥不可及，但我依然坚持下来，在实验完成后，我很期待下一个实验的开展，因为在每个实验中，我的收获远远大于过程中的煎熬。

参考文献

[1] *Lab 5 语法分析说明及要求.pdf*. zh. 2023.

[2] *Lab4-C 语言语法文法设计与验证实验.pdf*. zh. 2023.

[3] *如何优雅地展平你的语法树.pdf*. zh. 2021.

A 完整的文法

```
program -->
      func_list
func_list -->
      e
    | func func_list
func -->
      ctype declarator '(' args ')' code_block
ctype -->
      e
    | cytpe1 ctype
args -->
      e
    | arg_list
arg_list -->
      arg ',' arg_list
    | arg
arg -->
      ctype declarator
code_block -->
      '{' stmts '}'
stmts -->
      e
    | stmt stmts
    | decl stmts
decl -->
      ctype init_declarator_list ';'
init_declarator_list -->
      init_declarator
    | init_declarator ',' init_declarator_list
init_declarator -->
      declarator
    | declarator '=' exp_assign
declarator -->
      identifier post_declarator
```

```
post_declarator -->
      e
    | '[' ']'^^Ipost_declarator
    | '[' exp_assign ']' post_declarator
stmt -->
      code_block
    | select_stmt
    | iteration_stmt
    | return_stmt
    | exp_stmt
select_stmt -->
      'if' '(' exp ')' stmt
iteration_stmt -->
      'for' '(' exp ';' exp ';' exp ')' stmt
    | 'for' '(' decl ';' exp ';' exp ')' stmt
return_stmt -->
      'return' assign_exp ';'
    | 'return' ';'
exp_stmt -->
      exp ';'
    | ';'
exp -->
      exp_assign ',' exp
    | exp_assign
exp_assign -->
      exp_compare
    | exp_postfix operator_assign exp_assign
exp_compare -->
      exp_addsub
    | exp_addsub '<' exp_compare
    | exp_addsub '>' exp_compare
    | exp_addsub '<=' exp_compare
    | exp_addsub '>=' exp_compare^^I
exp_addsub -->
      exp_muldiv
    | exp_muldiv '+' exp_addsub
```

```
    | exp_muldiv '-' exp_addsub
exp_muldiv -->
      exp_postfix
    | exp_postfix '*' exp_muldiv
    | exp_postfix '/' exp_muldiv
    | exp_postfix '%' exp_muldiv
exp_postfix -->
      exp_unit exp_postfix1
exp_postfix1 -->
      e
    | '[' exp ']' exp_postfix1
    | '(' exp ')' exp_postfix1
    | '(' ')' exp_postfix1
    | '++' exp_postfix1
    | '--' exp_postfix1
exp_unit -->
      e
    | identifier
    | IntegerConstant
    | StringLiteral
    | '(' exp_assign ')'
cytpe1 -->
      void
    | int
    | char
operator_assign -->
      =
    | *=
    | /=
    | %=
    | +=
    | -=
```