语法分析实验

姓　　名：高　　月

学　　号：1120141944

班　　级：07121402

指导老师：计 卫 星

目录

一、实验目的

通过实践环节深入理解与编译实现有关的形式语言理论基本概念，掌握编译程序构造的一般原理、基本设计方法和主要实现技术，并通过运用自动机理论解决实际问题，从问题定义、分析、建立数学模型和编码的整个实践活动中逐步提高软件设计开发的能力。

二、实验内容

选择 C 语言的一个子集，基于 BIT-MiniCC 框架构建 C 语法子集的语法分析器，该语法分析器能够读入 XML 文件形式的属性字符流，进行语法分析并进行错误处理，如果输入正确时输出 XML 形式的语法树，输入不正确时报告语法错误。

三、实验环境

本次实验系统环境等信息如下表所示：

OS	IDE	Java SDK
macOS Sierra 10.12.5	IntelliJ IDEA 17.1.4	1.8 update 101

四、实验步骤

4.1 构造 C 语言文法

本次实验选择了 C 语言的一个子集，对实验要求中的文法进行扩展，构造的文法如下图所示，本次实验中使用递归下降分析方法。

```
CMPL_UNIT       : FUNC_LIST
FUNC_LIST       : FUNC_DEF FUNC_LIST | ε
FUNC_DEF        : TYPE_SPEC ID ( ARG_LIST ) CODE_BLOCK
TYPE_SPEC       : void | int | float
ARG_LIST        : ARGUMENT | ARGUMENT , ARG_LIST | ε
ARGUMENT        : TYPE_SPEC ID
CODE_BLOCK      : { STMT_LIST } | STMT
STMT_LIST       : STMT STMT_LIST | ε
STMT            : RTN_STMT | ASSIGN_STMT | DECDEF_STMT | BRH_STMT | LOOP_STMT | ;
RTN_STMT        : return EXPR ;
DECDEF_STMT     : TYPE_SPEC ID ; | TYPE_SPEC ASSIGN_STMT
ASSIGN_STMT     : ID = EXPR ;
BRANCH_STMT     : if CMP_CODE | else CODE_BLOCK
LOOP_STMT       : while CMP_CODE
CMP_CODE        : ( CMP_EXPR ) CODE_BLOCK
CMP_EXPR        : COMPARER == COMPARER | COMPARER < COMPARER | COMPARER > COMPARER
COMPARER        : ID | CONST
EXPR            : TERM EXPR2
EXPR2           : + TERM EXPR2 | - TERM EXPR2 | ε
TERM            : FACTOR TERM2
TERM2           : * FACTOR TERM2 | / FACTOR TERM2 | ε
FACTOR          : ID | CONST |  ( EXPR )
```

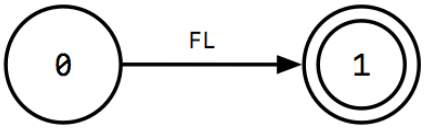
文法中符号含义如下：

CMPL_UNIT	编译单元
FUNC_LIST	函数列表
FUNC_DEF	函数定义
TYPE_SPEC	类型
ID	标识符
CONST	常量
ARG_LIST	参数列表
ARGUMENT	参数
CODE_BLOCK	代码块
STMT_LIST	语句列表
SMTM	语句
RTN_STMT	返回语句
ASSIGN_STMT	赋值语句
DECDEF_STMT	声明定义语句
BRH_STMT	分支语句
LOOP_STMT	循环语句
CMP_CODE	比较块和代码块
CMP_EXPR	比较表达式
COMPARER	比较元
EXPR	表达式
TERM	项
FACTOR	因子

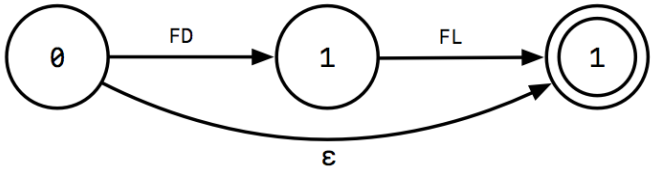
4.2 构造文法对应的状态转换图

为简化状态转化图，将文法中的符号简化标识：取每个符号的首字母和（如 CU 表示 CMPL_UNIT，F 表示 FACTOR）。

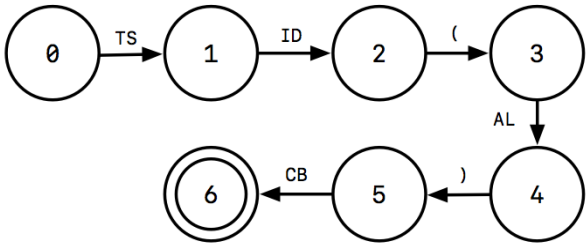
1. CMPL_UNIT -> FUNC_LIST 对应的状态转换图如下：



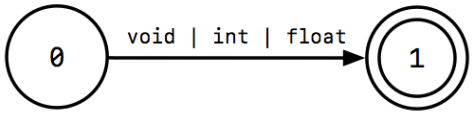
2. FUNC_LIST -> FUNC_DEF FUNC_LIST | ε 对应的状态转换图如下：



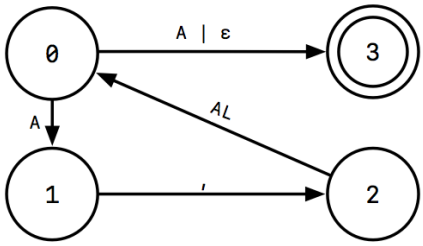
3. FUNC_DEF -> TYPE_SPEC ID (ARG_LIST) CODE_BLOCK 对应的状态转换图如下：



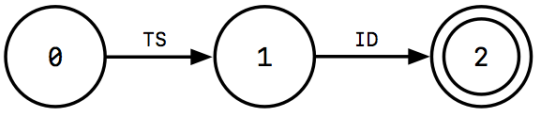
4. TYPE_SPEC -> void | int | float 对应的状态转换图如下：



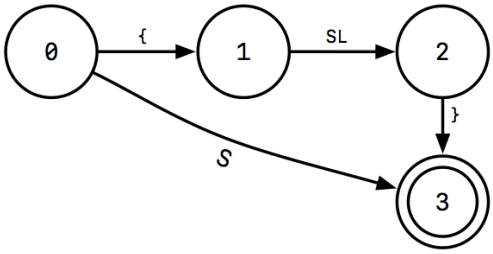
5. ARG_LIST -> ARGUMENT | ARGUMENT , ARG_LIST | ε 对应的状态转换图如下：



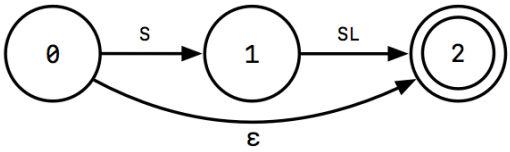
6. ARGUMENT -> TYPE_SPEC ID 对应的状态转换图如下：



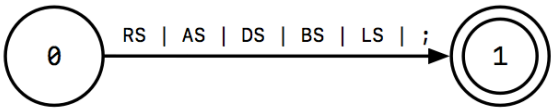
7. CODE_BLOCK -> { STMT_LIST } | STMT 对应的状态转换图如下：



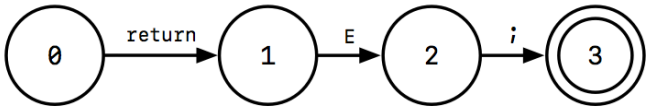
8. STMT_LIST -> STMT STMT_LIST | ε 对应的状态转换图如下：



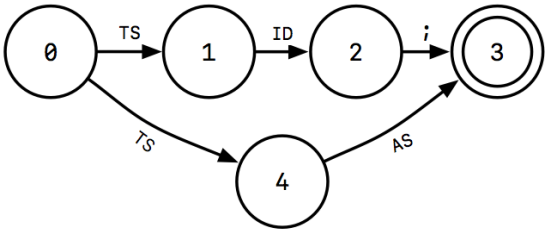
9. STMT -> RTN_STMT | ASSIGN_STMT | DECDEF_STMT | BRH_STMT | LOOP_STMT | ; 对应的状态转换图如下：



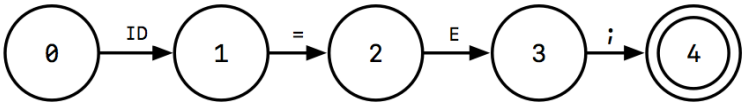
10. RTN_STMT -> return EXPR ; 对应的状态转换图如下：



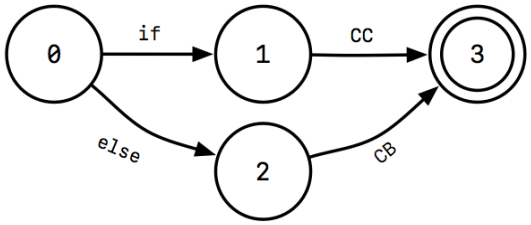
11. DECDEF_STMT -> TYPE_SPEC ID ; | TYPE_SPEC ASSIGN_STMT 对应的状态转换图如下：



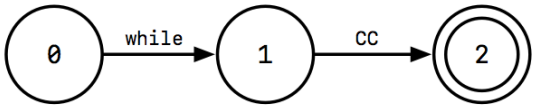
12. ASSIGN_STMT -> ID = EXPR ; 对应的状态转换图如下：



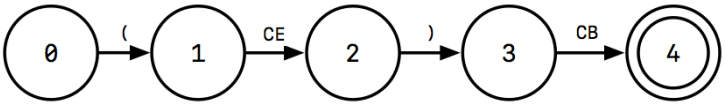
13. BRANCH_STMT -> if CMP_CODE | else CODE_BLOCK 对应的状态转换图如下：



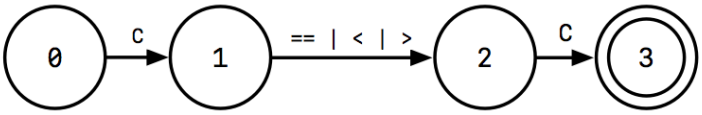
14. LOOP_STMT -> while CMP_CODE 对应的状态转换图如下：



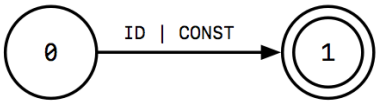
15. CMP_CODE -> (CMP_EXPR) CODE_BLOCK 对应的状态转换图如下：



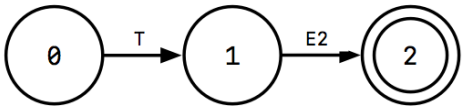
16. CMP_EXPR -> COMPARER == COMPARER | COMPARER < COMPARER | COMPARER > COMPARER 对应的状态转换图如下：



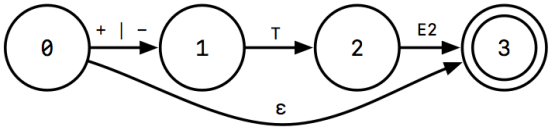
17. COMPARER -> ID | CONST 对应的状态转换图如下：



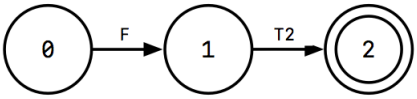
18. EXPR -> TERM EXPR2 对应的状态转换图如下：



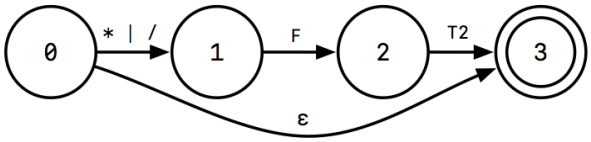
19. EXPR2 -> + TERM EXPR2 | - TERM EXPR2 | ε 对应的状态转换图如下：



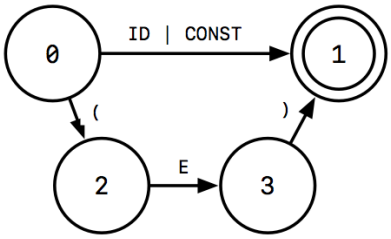
20. TERM -> FACTOR TERM2 对应的状态转换图如下：



21. TERM2 -> * FACTOR TERM2 | / FACTOR TERM2 | ε 对应的状态转换图如下：



22. FACTOR -> ID | CONST | (EXPR) 对应的状态转换图如下：



4.3 代码实现

此次试验使用递归下降的方法，根据文法定义各非终结符对应的函数，再根据文法进行递归的调用，递归下降分析器大体框架如下所示：

```
public void startSyntaxAnalysis(List<GWordInfo> wordLab, String oXMLFile) {...}
private Element cmplUnit() {...}
private Element funcList() {...}
private Element funcDef() {...}
private Element typeSpec() {...}
private Element argList() {...}
private Element argument() {...}
private Element codeBlock() {...}
private Element stmtList() {...}
private Element statement() {...}
private Element assignStmt() {...}
private Element decdefStmt() {...}
private Element returnStmt() {...}
private Element branchStmt() {...}
private Element loopStmt() {...}
private Element cmpCode() {...}
private Element compareExpr() {...}
private Element comparer() {...}
private Element expr() {...}
private Element expr2() {...}
private Element term() {...}
private Element term2() {...}
private Element factor() {...}
```

注：完整的代码包含 GWordInfo.java GParser.java GSyntaxAnlyzer.java 均在附件中

五、实验结果

5.1 实验准备

5.1.1 IDE 的配置

为便于运行测试，在 IntelliJ IDEA 中配置主函数参数即输出文件，主菜单 Run -->

Edit Configurations，配置如下：

5.1.2 修改 BIT-MiniCC 实验框架的配置文件

为完成此实验，将框架中的 config.xml 文件做如下修改：

```
<phase skip="true" type="java" path="" name="pp" />
<phase skip="false" type="java" path="bit.minisys.minicc.GScanner" name="scanning" />
<phase skip="false" type="java" path="bit.minisys.minicc.GParser" name="parsing" />
```

5.1.3 测试所用 C 语言代码

测试的 C 语言正确代码如下图所示：

```
int main ( ) {
    int a = 10 ;
    int b = 10 ;
    if ( a > b) a = c;
    else { c = a + b; }
    while ( a > b) { c = 100; a = 10; }
    return c ;
}

void pro ( int a, int b ) {
    float f ;
    f = 0.0 ;
    return f ;
}
```

测试的 C 语言错误代码如下图所示：(删除了主函数体的左大括号)

```
int main ( )
    int a = 10 ;
    int b = 10 ;
    if ( a > b) a = c;
    else { c = a + b; }
    while ( a > b) { c = 100; a = 10; }
    return c ;
}

void pro ( int a, int b ) {
    float f ;
    f = 0.0 ;
    return f ;
}
```

5.2 输出结果

5.2.1 测试输出的语法分析树 XML 文件

注：正确的、有误的测试程序对应的完整的分析树 XML 文件包含在附件中

5.2.2 错误处理

当递归下降分析其处理到错误时，立即终止当前线程，但其他线程不受影响，得益于 JavaFX Application 中 Platform 类的 `static void exit()` 函数。具体效果（控制台输出）如下：

```
Project name: test
Got ERROR when parse token #11
Got ERROR when parse token #12
Got ERROR when parse token #15
Finished parsing!
File saved to 'input/test.tree.xml'
Compiling completed!
```

第 11 个 token 就对应争取程序的主函数体的左大括号，由于使用了的 `exit()` 函数，其他线程仍然会继续往后处理直至其他线程均遇到错误，这样会相继引发更多的错误（如上图所示），最后的仍然会输出语法分析树。

六、实验心得

本次实验相比词法分析较为简单，难点在于 C 语言子集文法的构造，而文法到代码实现则容易的多。

通过本次实验实现了一个较为简单的递归下降语法分析器，在此过程中也遇到了一些问题：

1. C 语言子集的选取：为了理解语法分析的本质工作，本次实验选取了 C 语言中具有代表性的变量声明定义语句，赋值语句，返回语句，分支语句（`if-else`），循环语句（`while`）。
2. C 语言文法的构造：为了更大限度的减少代码实现中编写大量重复代码，将 `if` 和 `while` 后文法意义上相同的比较表达式和代码块组合成了一个非终结符。
3. 语法分析是的错误处理：本次实验中将错误处理放在了终结符的那一层函数，即在递归调用函数时，当处理到不满足文法定义的终结符时，立即报错，并终止此次分析。
4. 错误处理时退出函数的选择：本次实验选择了 `javafx.application.Platform.exit()` 而不是 `System.exit(status)`，来终止当前线程而不是整个程序，在实际中也有更好的应用。

总的来说，这次实验中收获到了很多，对编译器设计中语法分析阶段有了更深的认识，对 Java 编程也有了更深的理解，学习了从实际问题出发一步步解决问题的方法。