# 文法设计实验实验报告

陈照欣 1120191086

一、实验目的

本次实验的主要目的是了解程序设计语言的演化过程和相关标准的制定过程，深入理解与编译实现有关的形式语言理论，熟练掌握文法及其相关的概念，并能够使用文法对给定的语言进行描述，为后面的词法分析和语法分析做准备。

二、实验内容

本次实验需要依次完成以下三项内容：

（1）阅读附件提供的 C 语言和 Java 语言的规范草稿，了解语言规范化定义应包括的具体内容。

（2）选定 C 语言子集，并使用 BNF 表示方法文法进行描述，要求至少包括表达式、赋值语句、分支语句和循环语句；或者设计一个新的程序设计语言，并使用文法对该语言的词法规则和文法规则进行描述

（3）根据自己定义的文法子集，推导出“Hello Word”程序。 以上语言定义首先要给出所使用的字母表，在此基础上使用 2 型文法描述语法规则。

三、实验过程

1. 对于语言规范的理解

语法规范广义来讲可以指任意一种自然界中存在的语言有关发音、用词、语序和符号等方面的规则和规范。在编译原理课程中，我们主要讨论的是计算机使用的语言的相关规则。日常生活中，我们说话很有可能会产生二义性，人们能够根据语境对于语义进行理解和分析，但是仍然有可能出现理解上的偏差。计算机相比人类灵活性更低，因此我们在进行机器相关语言的设计时需要遵守更严格的语法

规范。 每一种高级语言都有其特殊的设计思路所在。例如 C 语言是一种面向过程的语言，Java 语言则是面向对象的语言，这样的语言特性实际上都需要用语法规范趋势线。语法规范需要简洁、清晰、没有二义性，并且复杂程度适中。语言由语法和语义两部分构成。例如 C 语言的规范中，语法大致可以分为表达式、声明、语句和定义这四种类型。

2. 文法设计

2.1 程序入口：

程序由声明的列表(或序列)组成，声明可以是函数或变量声明，顺序是任意的。至少必须有一个声明。所有的变量和函数在使用前必须声明。程序中最后的声明必须是一个函数声明，名字为 main。

```
compilationUnit
    :   translationUnit? EOF
    ;

translationUnit
    :   externalDeclaration translationUnit_
    ;

translationUnit_
    :   externalDeclaration translationUnit_
    |
    ;

externalDeclaration //外部声明
    :   functionDefinition
    |   declaration
    |   ';' // stray ;
    ;
```

2.2 语句定义

语句可分为：组合语句，表达式语句，迭代语句，跳转语句，选择语句等。具体定义如下：

```
statement
    :   compoundStatement
    |   expressionStatement
    |   iterationStatement
    |   jumpStatement
    |   selectionStatement
    ;

compoundStatement
    :   '{' blockItemList? '}'
    ;

expressionStatement
    :   expression ';'
    ;


iterationStatement
    :   'while' '(' expression ')' statement
#iterationStatement_while
    |   'do' statement 'while' '(' expression ')' ';'
#iterationStatement_dowhile
    |   'for' '(' expression? ';' expression? ';' expression? ')'
statement    #iterationStatement_for
    |   'for' '(' declaration expression? ';' expression? ')'
statement        #iterationStatement_forDeclared
    ;

jumpStatement
    :   'continue' ';'                #jumpStatement_continue
    |   'break' ';'                  #jumpStatement_break
    |   'return' expression? ';'        #jumpStatement_return
    ;

selectionStatement
    :   'if' '(' expression ')' statement ('else' statement)?
#selectionStatement_if
    |   'switch' '(' expression ')' statement
#selectionStatement_switch
    ;
```

2.3 声明语句

```
declaration
    :   typeSpecifiers initDeclaratorList? ';'
```

```
    ;

initDeclaratorList
    :   initDeclarator
    |   initDeclaratorList ',' initDeclarator
    ;

initDeclarator
    :   declarator
    |   declarator '=' assignmentExpression
    ;

declarator
    :   Identifier
    ;

typeSpecifiers   //数据类型
    :   typeSpecifier1
    |   typeSpecifier2 typeSpecifier2? typeSpecifier1?
    |   typeSpecifier3 (typeSpecifier2 typeSpecifier2?)?
typeSpecifier1
    ;

typeSpecifier1
    :   'void'
    |   'char'
    |   'int'
    |   'float'
    |   'double'
    ;

typeSpecifier2
    :   'long'
    |   'short'
    ;

typeSpecifier3
    :   'signed'
    |   'unsigned'
    ;
```

2.4 表达式

C 语言表达式的基本构成为：数据类型、常量与变量、数组、指针、字符串、文

件输入/输出等。

表达式的设计基本覆盖 C 语言中所有的操作符，由于时间关系对于部分不常用的语法内容进行了适当删减。

```
expressionStatement
    :   expression ';'
    ;

expression
    :   assignmentExpression    #expression_
    |   expression ',' assignmentExpression #expression_pass
    ;

unaryExpression
    :   postfixExpression       #unaryExpression_pass
    |   '++' unaryExpression    #unaryExpression_
    |   '--' unaryExpression    #unaryExpression_
    |   unaryOperator castExpression    #unaryExpression_
    |   '&&' Identifier    #unaryExpression_pass
    ;
postfixExpression
    :   primaryExpression
#postfixExpression_pass
    |   postfixExpression '[' expression ']'
#postfixExpression_arrayaccess
    |   postfixExpression '(' argumentExpressionList? ')'
#postfixExpression_funcall
    |   postfixExpression '.' Identifier
#postfixExpression_member
    |   postfixExpression '->' Identifier
#postfixExpression_point
    |   postfixExpression '++'
#postfixExpression_
    |   postfixExpression '--'
#postfixExpression_
    ;

assignmentExpression
    :   unaryExpression assignmentOperator assignmentExpression
    |   functionCall
    |   conditionalExpression
    ;
```

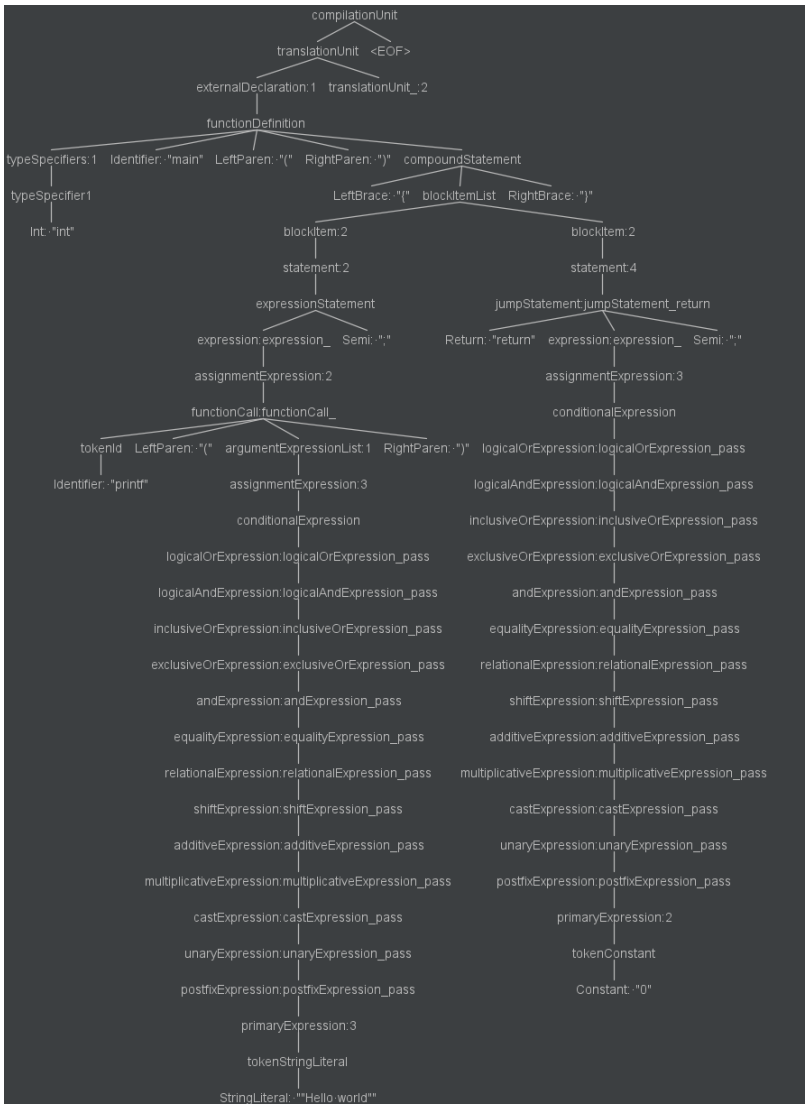
```
conditionalExpression
    :   logicalOrExpression ('?' expression ':'
conditionalExpression)?
    ;
logicalOrExpression
    :   logicalAndExpression    #logicalOrExpression_pass
    |   logicalOrExpression '||' logicalAndExpression
#logicalOrExpression_
    ;
logicalAndExpression
    :   inclusiveOrExpression   #logicalAndExpression_pass
    |   logicalAndExpression '&&' inclusiveOrExpression
#logicalAndExpression_
    ;
castExpression
    :   unaryExpression         #castExpression_pass
    ;

multiplicativeExpression
    :   castExpression          #multiplicativeExpression_pass
    |   multiplicativeExpression '*' castExpression
#multiplicativeExpression_
    |   multiplicativeExpression '/' castExpression
#multiplicativeExpression_
    |   multiplicativeExpression '%' castExpression
#multiplicativeExpression_
    ;

additiveExpression
    :   multiplicativeExpression    #additiveExpression_pass
    |   additiveExpression '+' multiplicativeExpression
#additiveExpression_
    |   additiveExpression '-' multiplicativeExpression
#additiveExpression_
    ;

shiftExpression
    :   additiveExpression  #shiftExpression_pass
    |   shiftExpression '<<' additiveExpression #shiftExpression_
    |   shiftExpression '>>' additiveExpression #shiftExpression_
    ;

relationalExpression
```

```
    :   shiftExpression     #relationalExpression_pass
    |   relationalExpression '<' shiftExpression
#relationalExpression_
    |   relationalExpression '>' shiftExpression
#relationalExpression_
    |   relationalExpression '<=' shiftExpression
#relationalExpression_
    |   relationalExpression '>=' shiftExpression
#relationalExpression_
    ;

equalityExpression
    :   relationalExpression    #equalityExpression_pass
    |   equalityExpression '==' relationalExpression
#equalityExpression_
    |   equalityExpression '!=' relationalExpression
#equalityExpression_
    ;

andExpression
    :   equalityExpression  #andExpression_pass
    |   andExpression '&' equalityExpression    #andExpression_
    ;

exclusiveOrExpression
    :   andExpression   #exclusiveOrExpression_pass
    |   exclusiveOrExpression '^' andExpression
#exclusiveOrExpression_
    ;

inclusiveOrExpression
    :   exclusiveOrExpression   #inclusiveOrExpression_pass
    |   inclusiveOrExpression '|' exclusiveOrExpression
#inclusiveOrExpression_
    ;
assignmentOperator
    :   '=' | '*=' | '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' | '&='
| '^=' | '|='
    ;


unaryOperator
    :   '&' | '*' | '+' | '-' | '~' | '!'
```

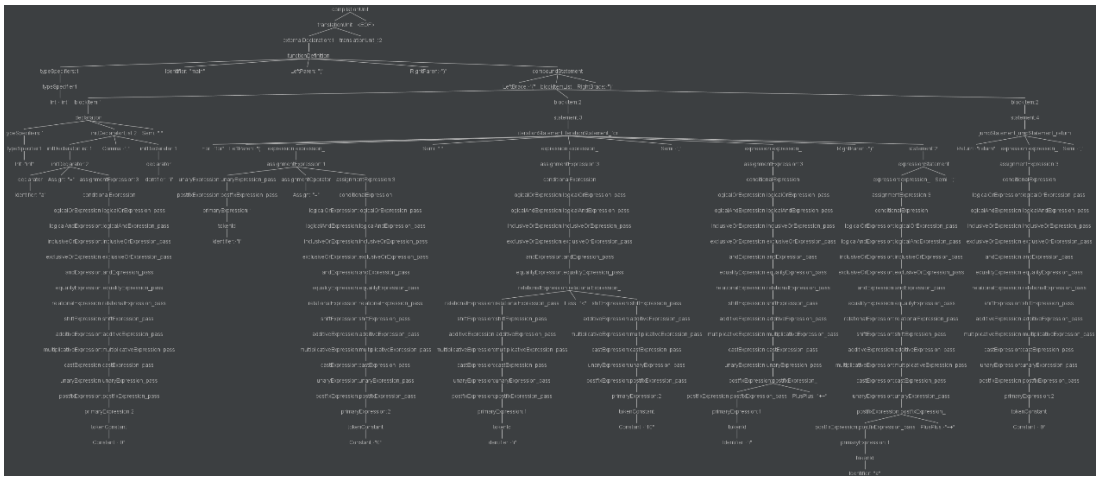
```
    ;


primaryExpression
    :   tokenId//Identifier
    |   tokenConstant//Constant
    |   tokenStringLiteral//StringLiteral+
    |   '(' expression ')'
    ;
tokenId
    : Identifier
    ;
tokenConstant
    : Constant
    ;
tokenStringLiteral
    : StringLiteral+
    ;
```

3. 语法分析验证:

3.1 简单的 hello world 程序




3.2 循环和赋值语句验证



四、实验心得

本次实验中，我仅仅是设计了一个 C 语言语法规范的较小子集，与完备的 C 语

言文法规定还有很远的距离。我学习并使用了 2 型文法描述去定义语法规范，利用 BNF 进行描述，同时在文法设计时也体会到了词法分析的作用和优势，即我们只需要对 token 流进行分析，而不需要对字符串进行操作，为文法设计提供了便利。