北京理工大學

语法分析实验报告

班　　级：07111504

姓　　名：徐恒达

学　　号：1120151811

目　录

1. 实验目的

语法分析（Syntax Analysis）是编译程序的核心部分。编译程序在完成词法分析之后，就进入语法分析阶段。语法分析的任务是：按照语言的语法规则，对单词串形式的源程序进行语法检查，并识别出相应的语法成分。本次实验的目的，是要对 C 语言进行语法分析，通过实验掌握语法分析的方法并理解语法分析在整个编译程序中的地位和重要性。

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 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。

3. 实验过程

本实验采用 LALR(1)分析方法。对于一个实际中常见的编程语言文法来说，文法符号大概有上百个，LR(1)分析得到的项目集规范族中的状态往往有上千个，占用的空间和计算需要的时间都非常多，LALR(1)对 LR(1)的项目集规范族中内核相同的项目集进行了合并，合并之后不会对分析器的识别能力有任何影响，但是使项目集的数量大大减少了。因此，LALR(1)是一种非常具有实践意义的分析方法。

整体算法

构建 LALR(1)项目集规范族有两种方法，一种是首先对文法构建 LR(1)项目集规范族，然后合并其中具有相同核心的项目集。这种方法虽然得到了精简的 LALR(1)项目集规范族，但在生成过程中还是构建了 LR(1)分析表，对内存和时间的消耗没有减少，因此，这只是理论上的一种思路，需要更简洁的方法直接得到 LALR(1)分析表，本次实验中使用一种从 LR(0)项目集规范族得到 LALR(1)项目集规范族的方法。

实验中构建 LALR(1)分析表的算法整体描述如下：

Algorithm 1 构建文法的 LALR(1)分析表

Input: 文法 G

Output： 文法 G 的 LALR(1)分析表

1) 创建文法 G 的 LR(0)项目集规范族 C0
2) 只保留每个项目集中的核心项
3) 计算各核心项之间搜索符的传播/生成关系
4) 从初始状态的项目集规范族开始，计算每个项目的搜索符
5) 对每个项目集求闭包即得到文法 G 的 LALR(1)项目集规范族 C1
6) 将 C1 转化为 LALR(1)分析表 T1

实际处理过程中，每个 LR(0)项目集中的所有项目并不是必须存储的，因为除了核心项目外的项目都是由核心项目生成的，我们可以只保留每个项目集中的核心项目，需要全部项目时，对项目集计算一下闭包即可得到。这是典型的一种用时间换空间的做法，同时也为后续计算搜索符之间的关系缩小了数据量。

算法中每一步的具体过程在下文中一一说明。

LR(0)项目集规范族

生成文法的 LR(0)项目集规范族的算法描述如下：

Algorithm 2 计算文法 G 的 LR(0)项目集规范族

Input: 文法 G

Output： 文法 G 的 LR(0)项目集规范族

将文法的开始符号记为S，添加产生式$S' \rightarrow S$使其成为一个增广文法

初始化项目集规范族collection $\leftarrow \Phi$

创建一个初始项目集$I_0 \leftarrow \Phi$，将文法项目$S' \rightarrow \cdot S$加入其中，然后计算闭包$I_0 = CLOSURE(I_0)$。

对于项目集中的每个项目$A \rightarrow \alpha \cdot \beta$，创建一条 goto 边到一个新的项目集 I，这个项目集中仅包含$A \rightarrow \alpha\beta$，计算闭包$I = CLOSURE(I)$，如果 I 不属于collection，就将 I 加入到collection中。

重复上一步操作，直到没有新的项目集加入为止。

返回文法 G 的 LR(0)项目集规范族collection。

其中计算一个 LR(0)项目集的闭包时算法中的关键操作，算法描述如下：

Algorithm 3 计算一个 LR(0)项目集的闭包

Input: LR(0)项目集 I

Output: I 的闭包 I'

初始化$I' \leftarrow \Phi$，然后将 I 中的所有项目加入到I'中。

对于I'中的每个形如$A \rightarrow \alpha \cdot B\beta$的产生式，若任何一个以 B 为左部的项目$B \rightarrow \cdot \alpha\beta$不在$I'$中，则将其加入$I'$。

重复上一步操作，直到没有新的项目加入为止。

项目集I'即为 I 的闭包

确定搜索符

对每个 LR(0)项目集去除非核心项目之后，就需要确定每个项目的搜索符，使其成为一个 LR(1)项目。我们需要确定每个 LR(0)项目集中自发生成的搜索符，同时也要确定搜索符从哪些项传播到了哪些项。这个检测实际上相当简单。令#为一个不在当前文法中的符号，令$A \rightarrow \alpha \cdot \beta$为项集I中的一个内核 LR(0)项。对每个 X 计算$J = GOTO(CLOSURE(\{[A \rightarrow \alpha \cdot \beta, \#]\}), X)$对于 J 中的每个内核项，我们检查它的搜索符号集合，如果#是它的搜索符号，那么搜索符就从$A \rightarrow \alpha \cdot \beta$传播到了这个项。所有其他的搜索符号都是自发生成的。这个思想在下面的算法中被精确地表达了出来。这个算法还用到了一个性质：J 中的所有内核项中点的左边都是 X，也就是说，它们必然是形如$B \rightarrow \gamma X \cdot \delta$的项。

Algorithm 4 确定搜索符号

Input: 一个 LR(0)项目集 I 的内核 K 以及一个文法符号 X。

Output: 由 I 中的项为GOTO(I, X)中内核项自发生成的搜索符号，以及 I 中将其搜索符号传播到GOTO(I, X)中内核项的项。

for (K 中的每个项$A \to \alpha \cdot \beta$)

$J := \text{CLOSURE}(\{[A \to \alpha \cdot \beta], \#\})$

if ($[B \to \gamma \cdot X\delta, a]$在J中，并且$a \neq \#$)

断定GOTO(I, X)中的项$B \to \gamma X \cdot \delta$的搜索符号a是自发生成的

if ($[B \to \gamma \cdot X\delta, \#]$在J中)

断定向前看符号从I中的项$A \to \alpha \cdot \beta$传播到了GOTO(I, X)中的项$B \to \gamma X \cdot \delta$之上

现在我们可以把搜索符号附加到 LR(0)项集的内核上，从而 LALR(1)项集。首先，我们直到$是初始 LR(0)项集中的$S' \to \cdot S$的向前看符号。算法 4 给出了所有自发生成的向前看符号。将所有这些向前看符号列出之后，我们必须让它们不断传播，直到不能继续传播为止。有很多方法可以实现这个传播过程。从某种意义上说，所有这些方法都跟踪已经传播到某个项但是尚未传播出去的 “新” 的向前看符号。下面的算法描述了一个将向前看符号传播到所有项中的技术。

Algorithm 5 LALR(1)项集族的内核的高效计算方法。

Input: 一个增广文法G′。

Output: 文法G′的 LALR(1)项集族的内核。

1) 构造 G 的 LR(0)项集族的内核。构造 LR(0)项集，然后再删除其中的非内核项。
2) 将算法 4 应用于每个 LR(0)项集的内核和每个文法符号 X，确定GOTO(I, X)中各内核项的哪些向前看符号是自发生成的，并确定向前看符号从 I 中的那个项被传播到GOTO(I, X)中的内核项上。
3) 初始化一个表格，表中给出了每个项集中的每个内核项相关的向前看符号。最初，每个项的向前看符号只包括那些被我们在步骤(2)中确定为自发生成的符号。
4) 不断扫描所有项集的内核项。当我们访问一个项i时，使用步骤(2)中得到的、用表格表示的信息，确定i将它的向前看符号传播到了哪些内核项中。项 i 的当前向前看符号集合被加到和这些被传播的内核项相关联的向前看符号集合中。我们继续在内核项上进行扫描，直到没有新的向前看符号被传播为止。

LR(1)闭包

我们有了 LALR(1)内核，就可以使用下面的算法对每个内核求闭包，然后再把这些 LALR(1)项集当作规范 LR(1)项集族，计算得到 LALR(1)语法分析表。

Algorithm 6 项集 I 的闭包

Input: 项目集 I。

Output: 项目集 I 的闭包。

repeat

for (I中的每个项$[A \to \alpha \cdot B\beta, a]$)

for (G′中的每个产生式$B \to \gamma$)

for (FIRST(βa)中的每个终结符 b)

将[B →· γ, b]加入到集合I中
until 不能向I中加入更多的项
return I

生成 LALR(1)分析表

使用如下的算法将 LALR(1)项目集规范族转化为 LALR(1)分析表。

Algorithm 7 构造 LALR(1)分析表

Input: 一个增广文法G′的 LALR(1)项集族

Output: 文法G′的 LALR(1)语法分析表的函数ACTION和GOTO。

1) 语法分析器的状态 i 根据I_i构造得到。状态 i 的于芬分析动作按照下面的规则确定：

① 如果$[A \to \alpha \cdot a\beta, b]$在$I_i$中，并且$GOTO(I_i, a) = I_j$，那么将ACTION[i, a]设置为“移入 j”。

这里 a 必须是一个终结符号。

② 如果$[A \to \alpha \cdot, a]$在I_i中且$A \neq S'$，那么将ACTION[i, a]设置为“归约$A \to \alpha$”。

③ 如果$S' \to S \cdot, \$$在I_i中，那么将ACTION[i, a]设置为“接受”。

如果根据上述规则会产生任何冲突动作，我就就说这个文法不是 LR(1)的。在这种情况下，这个算法无法为该文法生成的一个语法分析器。

2) 状态 i 相对于各个非终结符号A的 goto 转化按照下面的规则构造得到：如果$GOTO(I_i, A) = I_j$，那么$GOTO[i, A] = j$。

3) 所有没有按照规则(1)和(2)定义的分析表条目都设置为“报错”。

4) 语法分析器的初始状态是由包含$S' \to \cdot S, \$$的项集构造得到的状态。

使用算法 7 得到ACTION函数和GOTO函数，也就是得到了 LALR(1)语法分析表，按照通用的规则实现分析表的驱动程序后，就可以读入任意的文法，从而得到文法的语法分析器。

程序实现

程序采用 Java 语言实现，定义了如下几个类，分别对其说明如下。

1. Generator

Generator 类读入一个存储的文法的 txt 文件，对其进行 LALR(1)分析后，返回其对应的 LALR(1)分析表对象。

```
1   public class Generator {
2     private Alphabet alphabet;
3     private Grammar grammar;
4     Generator() {
5       alphabet = Alphabet.newInstance();
6       grammar = Grammar.newInstance();
7     }
8     LALRTable generate(String path) throws Exception {
9       FileReader fileReader = new FileReader(path);
10      BufferedReader reader = new BufferedReader(fileReader);
```

```
            readgrammar(reader); // 读入文法
            ItemSet itemSet = new ItemSet();
            Item item = new Item(grammar.get(0), 0, "$");
            itemSet.add(item); // 初始化项目集
            ItemSetCollection collection = new ItemSetCollection();
            collection.init(itemSet); // 初始化项目集规范族
            collection.build(); // 构建项目集规范族
            collection.extractKernel(); // 去除非内核项
            collection.generateSpreadRelation(); // 计算传播/生成关系
            collection.spread(); // 确定搜索符
            collection.buildLAClosure(); // 计算闭包
            return collection.buildLALRTable(); // 生成 LALR 分析表
        }
    }
}
```

2. LALR

LALR 类是一个驱动器，它内部包含一个 LALR(1)分析表，当对其输入属性字流时，驱动器就可以按照分析表中的规则对属性字流进行自底向上的分析，如果最终结果为接受，则驱动器返回表示语法树的 XML 文档对象，如果出错，则进行出错处理。

```
public class LALR {
    public Document parse(List<Token> list) {
        Document document = builder.newDocument();
        Stack<Integer> stack = new Stack<>();
        stack.push(0);
        snapshot(stack, list);
        int i = 0;
        String symbol;
        while (true) {
            int state = stack.peek();
            String type = table.getActionType(state, symbol);
            switch (type) {
                case "SHIFT":
                    state = table.getActionIndex(state, symbol);
                    stack.push(state); i++;
                    symbol = ttype;
                    break;
                case "REDUCTION":
                    int index = table.getActionIndex(state, symbol);
                    Production p = grammar.get(index);
                    int num = p.bodySize();
                    for (int j = 0; j < num; j++) stack.pop();
                    state = stack.peek();
                    state = table.getGoto(state, p.getHead());
                    stack.push(state);
                    break;
```

```
                default:
                    System.out.println(type);
                    break;
            }
        }
        return  document;
    }
}
```

此外，自下而上为不同的编译概念创建了数据结构，分别是存储所有文法符号的字母表类 Alphabet，产生式类 Production，所有产生设构成的文法类 Grammar，项目类 Item，项目集类 ItemSet，项目集规范族类 ItemSetCollection，分别描述如下。

3. 字母表 Alphabet

```
public class Alphabet {
    private static Alphabet alphabet = null;
    private HashMap<String, Boolean> map;
    private Alphabet() {
        map = new HashMap<>();
        startSymbol = null;
    }
    public static Alphabet newInstance() {
        if (alphabet == null) alphabet = new Alphabet();
        return alphabet;
    }
}
```

4. 产生式 Production

```
public class Production {
    private String head;
private ArrayList<String> body;

    Production(String head) {
        this.head = head;
        body = new ArrayList<>();
    }
}
```

5. 文法 Grammar

```
public class Grammar extends ArrayList<Production> {
    private static Grammar grammar = null;

    public static Grammar newInstance() {
        if (grammar == null)
            grammar = new Grammar();
```

```
            return grammar;
        }
    }
```

6. 项目 Item

```
public class Item {
    private Production production;
    private int position;
    private Set<String> LASymbols;
    private List<Item> spread;

    Item(Production production, int position) {
        this.production = production;
        this.position = position;
        LASymbols = new HashSet<>();
        spread = new ArrayList<>();
    }
}
```

7. 项目集 ItemSet

```
public class ItemSet extends ArrayList<Item> {
    private Map<String, ItemSet> map;
    ItemSet() {
        map = new HashMap<>();
    }
}
```

4. 实验结果

从 ISO C11 标准中选取一个文法子集，整理后如下。（只选取部分）

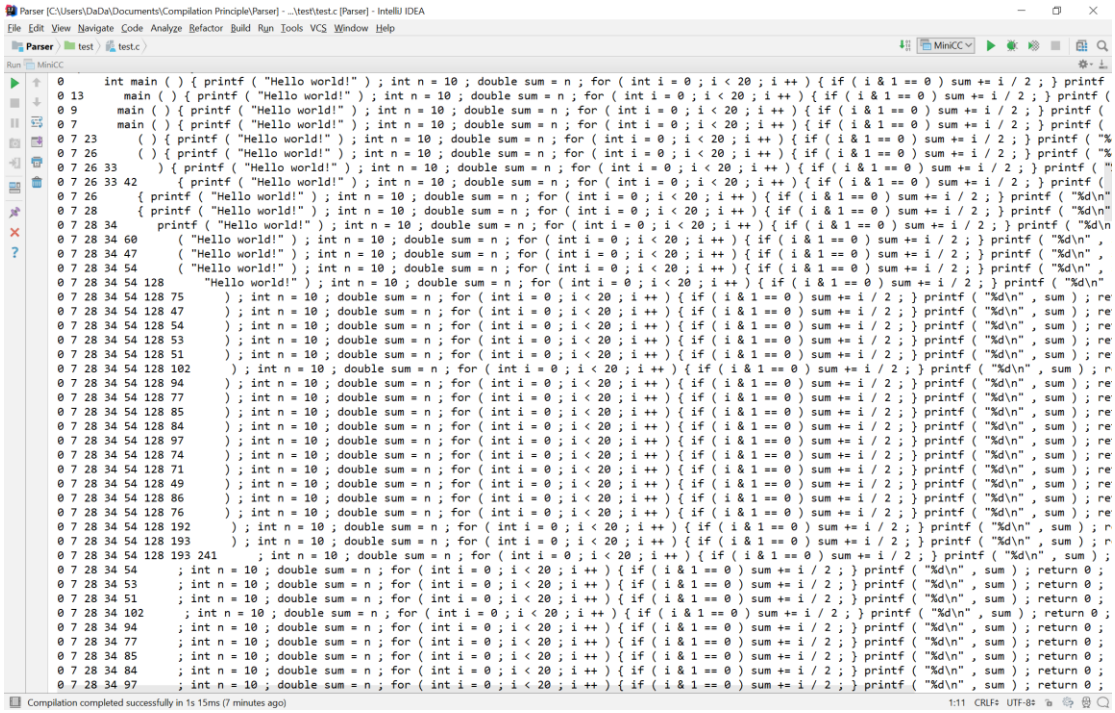
```
compilation-unit ::= translation-unit

translation-unit ::= external-declaration  | translation-unit external-declaration

external-declaration ::= function-definition  | declaration

function-definition ::= declaration-specifiers declarator compound-   statement

declaration-specifiers ::= type-specifier

  | type-qualifier type-specifier

declarator ::= direct-declarator
```

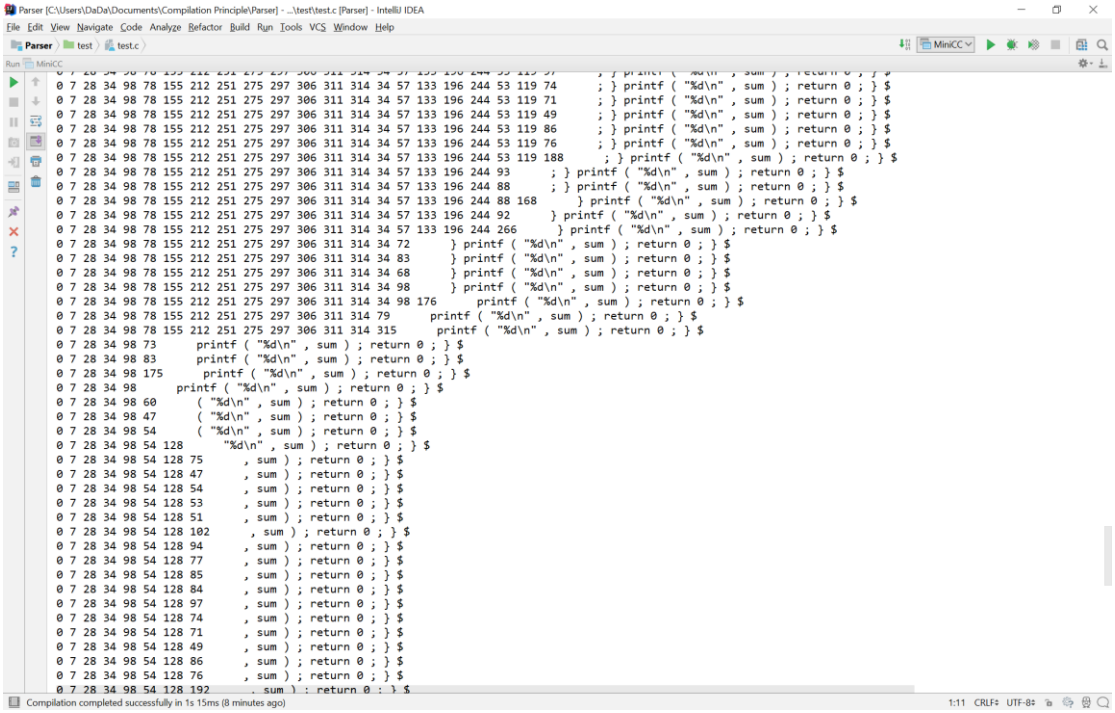
```
direct-declarator ::= identifier
   | direct-declarator ( )
| direct-declarator ( parameter-type-list )
type-specifier ::= void | char | short | int | long | float | double | signed | unsigned
type-qualifier ::= const | restrict | volatile | _Atomic
parameter-type-list ::= parameter-list | parameter-list , ...
parameter-list ::= parameter-declaration
   | parameter-list , parameter-declaration
parameter-declaration ::= declaration-specifiers declarator
   | declaration-specifiers abstract-declarator
......
```

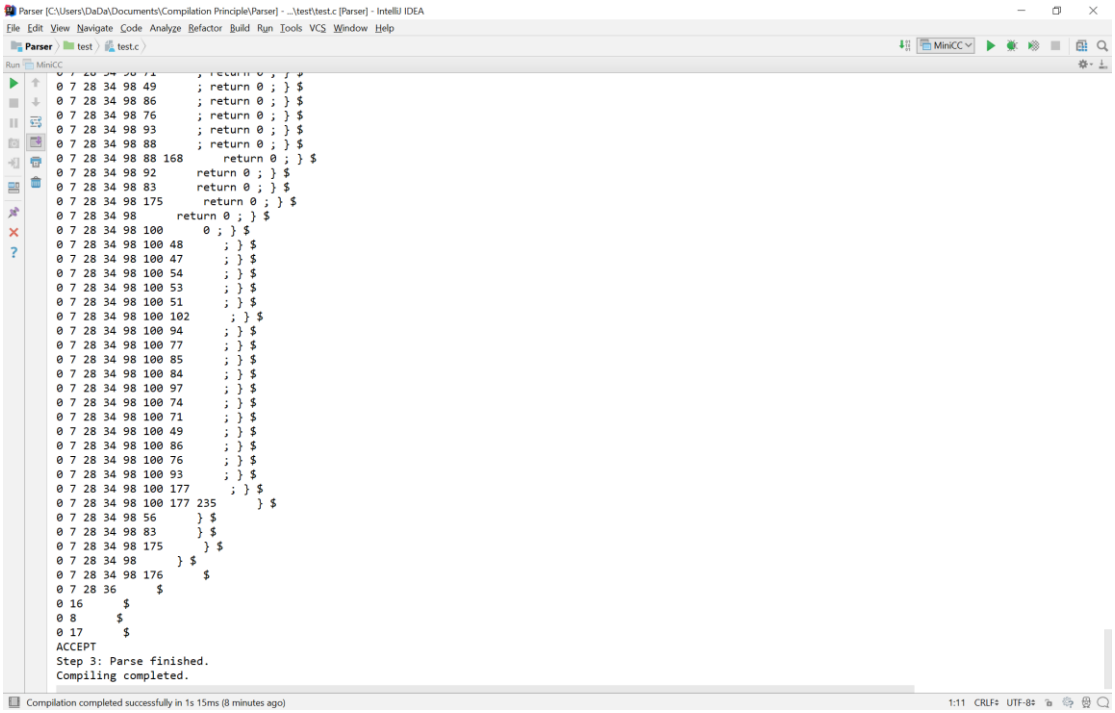
将其输入 Generator 对象后得到 LALR 分析表，准备一份测试用的 C 语言代码如下。其中包含了函数定义，变量声明，字符串，表达式语句，for 循环，if 语句等，能够比较全面地测试语法分析器的功能。

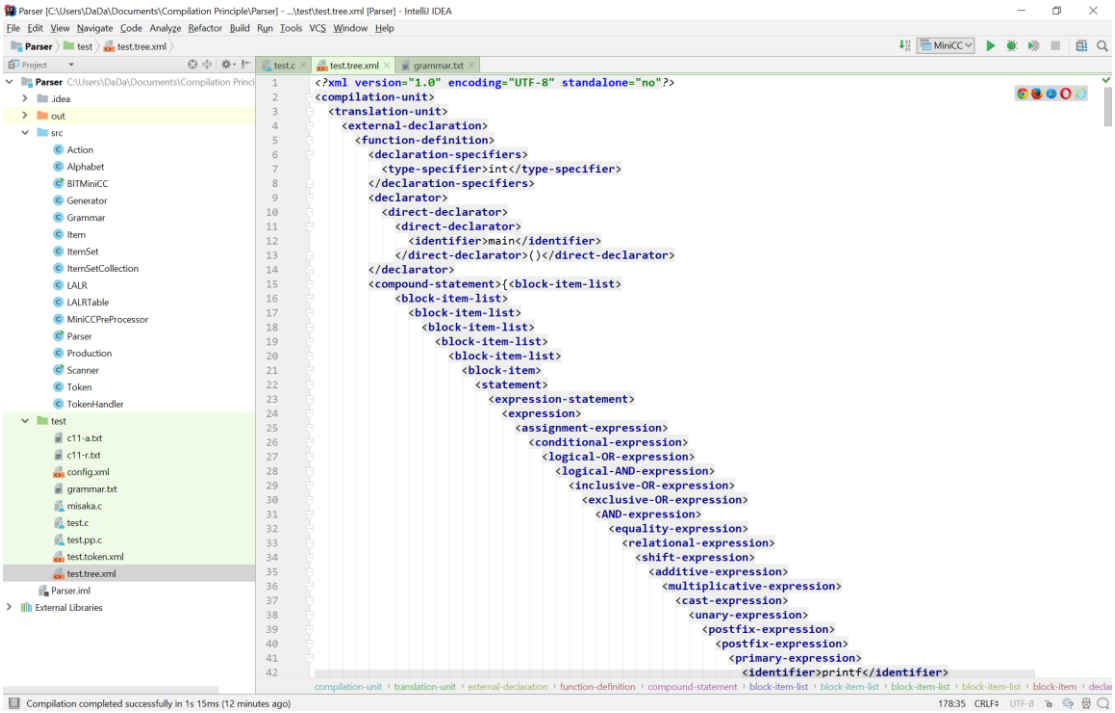
```c
int main()
{
    printf("Hello world!");
    int n = 10;
    double sum = n;
    for (int i = 0; i < 20; i++)
    {
        if (i & 1 == 0)
            sum += i / 2;
    }
    printf("%d\n", sum);
    return 0;
}
```

使用 BITMiniCC 框架，依次进行预处理、词法分析和语法分析后，语法分析程序显示 ACCEPT，语法分析成功，将分析过程中每一步堆栈中的状态和和最终的 XML 语法树展示如下。


Parser [C:\Users\DaDa\Documents\Compilation Principle\Parser] - ...\test\test.c [Parser] - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Parser test test.c
MiniCC
Compilation completed successfully in 1s 15ms (7 minutes ago)
1:11 CRLF UTF-8


Parser [C:\Users\DaDa\Documents\Compilation Principle\Parser] - ...\test\test.c [Parser] - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Parser test test.c
MiniCC
Compilation completed successfully in 1s 15ms (8 minutes ago)
1:11 CRLF UTF-8


Parser [C:\Users\DaDa\Documents\Compilation Principle\Parser] - ...\test\test.c [Parser] - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Parser test test.c
MiniCC
Run MiniCC
0 7 28 34 98 49 ; return 0 ; } $
0 7 28 34 98 86 ; return 0 ; } $
0 7 28 34 98 76 ; return 0 ; } $
0 7 28 34 98 93 ; return 0 ; } $
0 7 28 34 98 88 ; return 0 ; } $
0 7 28 34 98 88 168 return 0 ; } $
0 7 28 34 98 92 return 0 ; } $
0 7 28 34 98 83 return 0 ; } $
0 7 28 34 98 175 return 0 ; } $
0 7 28 34 98 return 0 ; } $
0 7 28 34 98 100 0 ; } $
0 7 28 34 98 100 48 ; } $
0 7 28 34 98 100 47 ; } $
0 7 28 34 98 100 54 ; } $
0 7 28 34 98 100 53 ; } $
0 7 28 34 98 100 51 ; } $
0 7 28 34 98 100 102 ; } $
0 7 28 34 98 100 94 ; } $
0 7 28 34 98 100 77 ; } $
0 7 28 34 98 100 85 ; } $
0 7 28 34 98 100 84 ; } $
0 7 28 34 98 100 97 ; } $
0 7 28 34 98 100 74 ; } $
0 7 28 34 98 100 71 ; } $
0 7 28 34 98 100 49 ; } $
0 7 28 34 98 100 86 ; } $
0 7 28 34 98 100 76 ; } $
0 7 28 34 98 100 93 ; } $
0 7 28 34 98 100 177 ; } $
0 7 28 34 98 100 177 235 } $
0 7 28 34 98 56 } $
0 7 28 34 98 83 } $
0 7 28 34 98 175 } $
0 7 28 34 98 } $
0 7 28 34 98 176 $
0 7 28 36 $
0 16 $
0 8 $
0 17 $
ACCEPT
Step 3: Parse finished.
Compiling completed.
Compilation completed successfully in 1s 15ms (8 minutes ago)
1:11 CRLF UTF-8


Parser [C:\Users\DaDa\Documents\Compilation Principle\Parser] - ...\test\test.tree.xml [Parser] - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Parser test test.tree.xml
MiniCC
Project
Parser C:\Users\DaDa\Documents\Compilation Princi
.idea
out
src
Action
Alphabet
BITMiniCC
Generator
Grammar
Item
ItemSet
ItemSetCollection
LALR
LALRTable
MiniCCPreProcessor
Parser
Production
Scanner
Token
TokenHandler
test
c11-a.txt
c11-r.txt
config.xml
grammar.txt
misaka.c
test.c
test.pp.c
test.token.xml
test.tree.xml
Parser.iml
External Libraries
test.c
test.tree.xml
grammar.txt
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<compilation-unit>
<translation-unit>
<external-declaration>
<function-definition>
<declaration-specifiers>
<type-specifier>int</type-specifier>
</declaration-specifiers>
<declarator>
<direct-declarator>
<direct-declarator>
<identifier>main</identifier>
</direct-declarator>()</direct-declarator>
</declarator>
<compound-statement>{<block-item-list>
<block-item-list>
<block-item-list>
<block-item-list>
<block-item-list>
<block-item-list>
<block-item>
<statement>
<expression-statement>
<expression>
<assignment-expression>
<conditional-expression>
<logical-OR-expression>
<logical-AND-expression>
<inclusive-OR-expression>
<exclusive-OR-expression>
<AND-expression>
<equality-expression>
<relational-expression>
<shift-expression>
<additive-expression>
<multiplicative-expression>
<cast-expression>
<unary-expression>
<postfix-expression>
<postfix-expression>
<primary-expression>
<identifier>printf</identifier>
compilation-unit › translation-unit › external-declaration › function-definition › compound-statement › block-item-list › block-item-list › block-item-list › block-item-list › block-item
Compilation completed successfully in 1s 15ms (12 minutes ago)
178:35 CRLF UTF-8

5. 心得体会

通过本次实验，基本掌握了语法分析部分中的自上而下分析方法的基本知识，在文法设计和简化上有了很多的练习，并且在代码实现的过程中，进一步加强对 Java 语言的理解和熟练度。

在本次实验中感触最深的就是，在设计 C 语言文法的时候，面对网上找到的英文的 C 语言完整文法理解起来非常吃力，但是后来在浏览了非常多的博客后，逐渐掌握了设计文法的窍门，并且凭借自己的理解将完整文法翻译出来，方便了我的程序设计和简化。另外由于之前设计大型程序的经验比较少，导致在遇到调用额外的 jar 包中函数的情况时，比如在生成语法树，要使用的 jdom.jar 中的很多类，也是在不断试验和资料检索后才逐渐学会掌握了使用 java 的“库函数”。

本次实验只是编译程序六步“词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成”中的第二步，距离一个完整可使用的编译器差距还很大。我以后一定会继续努力学习编译原理相关知识，早日完成自己的编译器！