哈尔滨工业大学（深圳）

《编译原理》
实验指导书

2024 秋

目录

前言

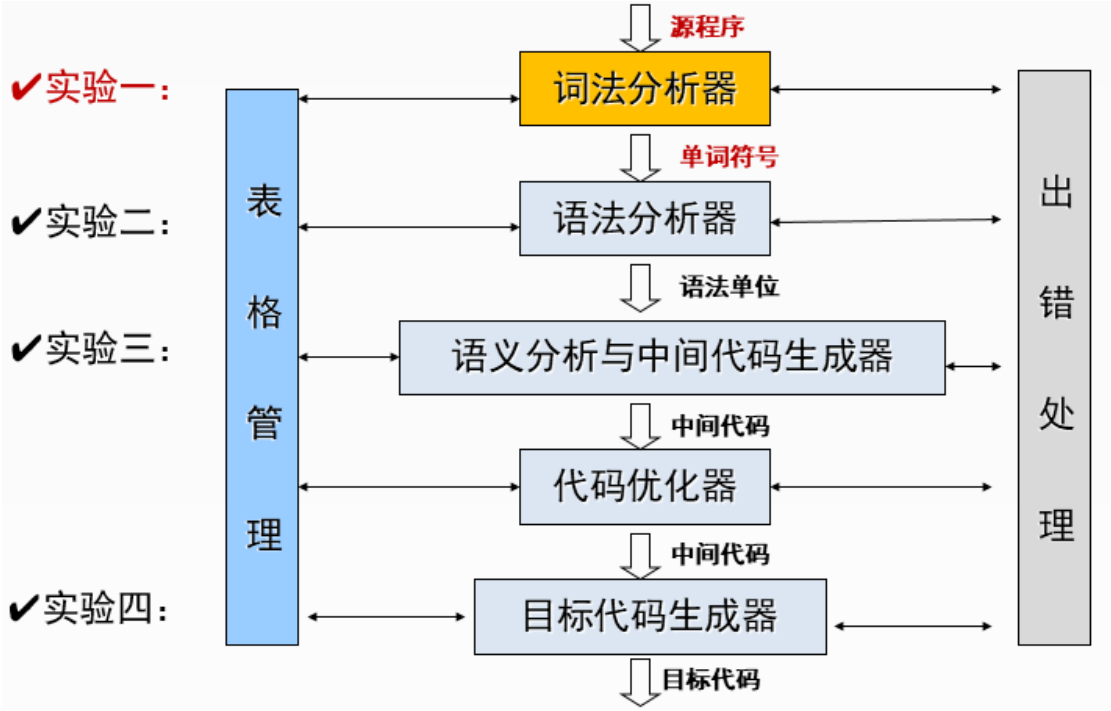
本学期的编译原理实验总共有 4 次实验，全部为设计型实验。

实验一：词法分析器的实现；

实验二：自底向上的语法分析—LR（1）；

实验三：典型语句的语义分析及中间代码生成；

实验四：目标代码生成。



第一部分

一 实验整体概述

一个 Java 实现的 TXTv2 语言编译器，目标平台为 RISC-V 32（指令集 RV32M）。

1.1 目录说明

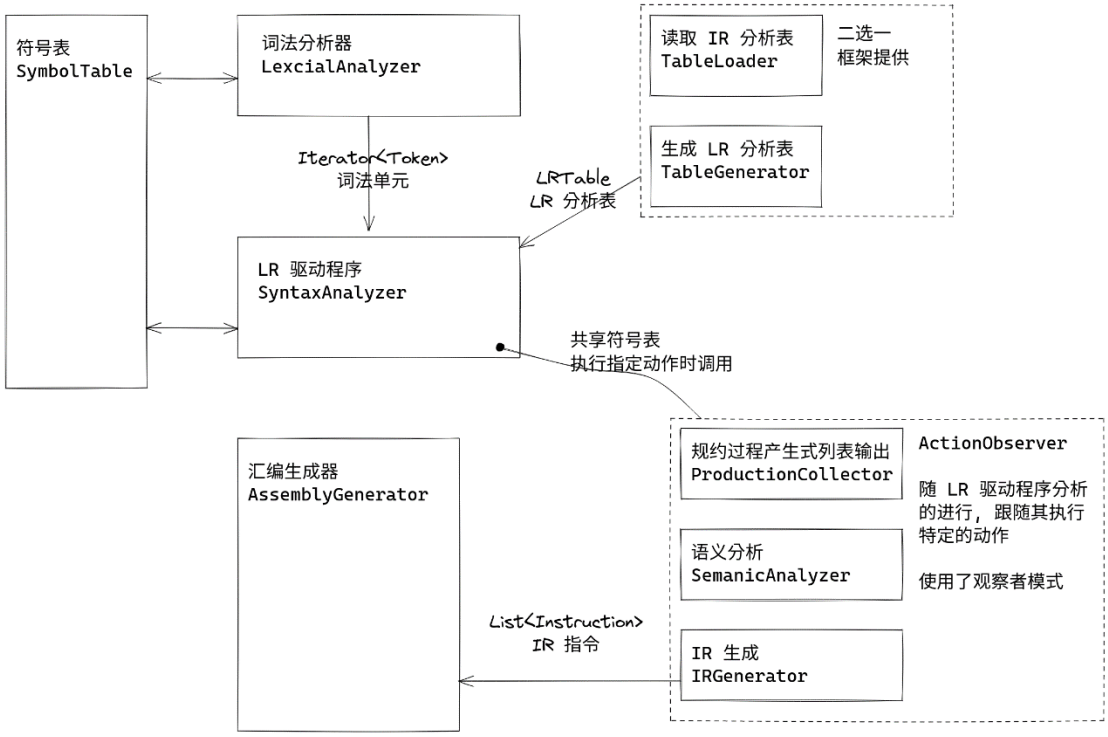
tree . -L 2 --dirsfirst --sort=name

```
.
├── data
│   ├── in                  # 提供给程序的输入数据
│   ├── out                 # 程序的输出数据
│   └── std                 # 用作参考的标准输出数据
├── scripts
│   ├── check-result.py     # 对输出进行 check 的脚本
│   ├── diff.py             # 忽略文件尾空行和行首位空白符的 diff 工具
│   └── make-template.py    # 从代码出框架的脚本，无需关注
└── src                     # 源码目录
```

tree src/cn/edu/hitsz/compiler -L 2 --dirsfirst --sort=name

```
src/cn/edu/hitsz/compiler
├── asm                                  # 实验四: 汇编生成
│   └── AssemblyGenerator.java           # 汇编生成器
├── ir                                   # 基础架构: IR
│   ├── BasicBlock.java                  # IR 中的基本块
│   ├── Instruction.java                 # IR 中的指令
│   ├── InstructionKind.java             # IR 中的指令类型
│   ├── IRImmediate.java                 # IR 中的立即数
│   ├── IRValue.java                     # IR 中的值
│   ├── IRVariable.java                  # IR 中的变量
│   └── Program.java                     # IR 中代表完整的一个程序的类
├── lexer                                # 实验一: 词法分析
│   ├── LexicalAnalyzer.java             # 词法分析器
│   ├── Token.java                       # 词法单元
│   └── TokenKind.java                   # 词法单元类别
├── parser                               # 实验二/三: 语法分析, 语义分析, IR 生成
│   ├── table                            # 读取/生成 LR 表的工具
│   ├── ActionObserver.java              # 观察者接口
│   ├── IRGenerator.java                 # IR 生成
│   ├── ProductionCollector.java         # 规约产生式记录
│   ├── SemanticAnalyzer.java            # 语义分析
│   └── SyntaxAnalyzer.java              # 语法分析
├── pass                                 # 可选部分: 优化 pass
│   ├── AdjustIRForRISCV.java            # 实验四 (可选部分): 体系结构相关优化与
IR 调整
│   ├── IRPass.java                      # 所有 IR 优化 pass 的基类
│   └── IRPassManager.java               # IR 优化 pass 管理器
├── symtab                               # 符号表
│   ├── SourceCodeType.java              # 源语言变量类型
│   ├── SymbolTableEntry.java            # 符号表条目
│   └── SymbolTable.java                 # 符号表
├── utils                                # 杂项/工具
│   ├── CasePaths.java                   # 记录程序中用到的相对于测试用例的各
类路径
│   ├── FileUtils.java                   # 文件读写工具
│   ├── IREmulator.java                  # 评测用 IR 解释器
│   └── ListForwarder.java               # 辅助类, 用于便捷地为某个类实现 List
接口
├── Main.java                            # 主函数
├── MainForAutoJudge.java                # 用于自动化评测的主函数
└── NotImplementedException.java         # 用于填充待实现部分的异常
```

1.2 代码架构示意图



1.3 源语言

由于实验种种限制，本次实验实现的简单编译器的源语言支持的语法包含变量的申明、赋值、简单算术表达式这些简单语法。

```
int result;
int a;
int b;
int c;
a = 8;
b = 5;
c = 3 - a;
result = a * b - ( 3 + b ) * ( c - a );
return result;
```

1.4 目标平台 && 目标语言

我们的目标是生成在支持 RV32M 指令集的 RISC-V 32 机器上可以成功运行的汇编代码。然而这份代码要成功运行还得通过汇编器和链接器生成可执行文件并链接到相应的库上。

然而在实际操作中我们并没有一台 RISC-V 32 的机器，就算通过 qemu 模拟, 我们仍不可避免地需要与 GNU 汇编伪指令操作打交道。为了避免这些麻烦, 我们使用 RARS(RISC-V Assembler and Runtime Simulator) 进行模拟。可以认为我们的目标机器为 RARS。

根据 RISC-V 的寄存器使用约定，我们使用 Caller 保存的 t0 到 t6，亦即 x5-7, x28-31 作为汇编代码中任意使用的寄存器, 使用 a0，亦即 x10 作为程序的返回值。并且我们使用 CompactDataAtZero 选项规定 data 的开始地址为 0x0。通过 RARS 我们可以这样进行汇编程序的执行：

```
$ java -jar rars.jar mc CompactDataAtZero a0 nc dec ae255 riscv1.asm
Program terminated by dropping off the bottom.
a0  10945
```

1.5 关于 NotImplementedException

这个 NonImplementedException 的异常在期待你实现但是你还没有实现的函数中被抛出。

由于我们一次性下发了整个项目的代码，所以在你完成整个实验之前，程序都会以抛出 NonImplementedException 异常然后结束。

注意：在完成实验一代码后，因为后续实验二、三、四在主函数中已调用但具体实现代码并未完成，所以依然会在后续代码中抛出 NonImplementedException 的异常，请你不必介意。如若不想让未完成的实验代码抛出异常，那你可以用以下方式修改主函数代码：

```
public class Main {
    public static void main(String[] args) {
        // 构建符号表以供各部分使用
        TokenKind.loadTokenKinds();
        final var symbolTable = new SymbolTable();

        // 词法分析
        final var lexer = new LexicalAnalyzer(symbolTable);
        lexer.loadFile(FilePathConfig.SRC_CODE_PATH);
        lexer.run();
        lexer.dumpTokens(FilePathConfig.TOKEN_PATH);
        final var tokens = lexer.getTokens();
        symbolTable.dumpTable(FilePathConfig.SYMBOL_TABLE_PATH);

        /**因为后续实验代码未完成，在主函数中调用会抛 NonImplementedException 异
        常，你可以选择先注释后续调用，解决抛异常的问题*/

        /**
        //实验二
        //实验三
        //实验四
        */
    }
```

如果你完成了每次实验的要求，那么这种结束方式是无关大雅的。倘若你担心遗漏了某次实验中待填充的函数，那么请你先清空 data/out 文件夹，随后运行程序，再检查一下程序运行之后在 data/out 中的输出情况。只要输出的文件数量符合要求且数量/内容是正确的，那么该实验就算是完成了。

二 实验环境及代码规范

2.1 实验环境

1. Java 17 及以上版本.
2. RARS.
3. 编译工作台 (可选)

2.2 Java 语言标准

使用 Java 最新的 LTS 版本，即 Java 17。Java 17 以上以及 Java 17 处于 preview 的特性都不建议使用，方便统一批改。

鼓励使用 Java 的新特性以及新的标准 API，如：

- ✧ 模式匹配 instanceof (Java 16)
- ✧ 增强的 switch 表达式 (Java 14)
- ✧ record 类(Java 16)
- ✧ Stream API 对元素流进行函数式操作 (Java 8)

2.3 Java 代码注意事项

Java 是一个很 OOP 的语言，在使用 Java 完成本实验时请注意之前所学的面向对象的六大原则。

1. 依赖倒转原则(DIP)

依赖倒转原则(DIP)是指一种特定的解耦(传统的依赖关系建立在高层次上,而具体的策略设置则应用在低层次的模块上)形式，使得高层次的模块不依赖于低层次的模块的实现细节，依赖关系被颠倒(反转)，从而使得低层次模块依赖于高层次模块的需求抽象。

高层次的模块不应该依赖于低层次的模块，两者都应该依赖于抽象接口。抽象接口不应该依赖于具体实现，而具体实现则应该依赖于抽象接口。

好的实现

```
List<Instruction> porcessInstruction(List<Instruction> originInstructions){
    // some code
    return res;
}
```

坏的实现

```
ArrayList<Instruction> porcessInstruction(ArrayList<Instruction> originInstructions){
    // some code
    return res;
}
```

2. 底米特原则

✧ 每个对象应该对其他对象尽可能最少的知道

✧ 每个对象应该仅和其朋友通信；不和陌生人通信

✧ 仅仅和直接朋友通信

好的实现

将各个模块统一交由顶层模块进行调用，各个模块通过顶层模块使用约定好的 API 接口进行传参。

坏的实现

各个模块相互调用，如链式调用：当 A、B、C 为同一层级的模块时，Main call A、A call B、B call C ...

2.3 Java 程序设计风格

Java 是 Java。换句话讲，Java 不是 C。

没必要什么数据结构都用数组(整数索引的表)来实现，特别是当这个"表" 实际上具有更深层的结构的时候。

深入地, 没必要什么东西都用一个整数来代表。如果一个对象具有任何与整数不同的性质, 那它就不是整数。尽量使用对象的引用来引用对象，而不是拿着一个 int 再去什么地方查询。

字符串是方便使用且默认不可变的，使用字符串与使用任何类型一样方便，字符串不再是危险且需要托管的。字符(Character)跟字节(Byte)是不同的，前者是任何在特定编码下代表某种自然语言中的最小单元的数据，后者是一个没有附加结构的 8bit 数据。不要假设所有字符都是 ASCII 编码下的合法字符。

通用数据结构(容器/集合类)是方便且随时可用的，应该尽可能根据职责拆分需要通用数据结构维护的数据，而不是集中写一个数据结构来维护所有信息。

对外隐藏是方便必要的，每个标识符的访问范围都应该尽可能地小。

特殊值是危险的，在任何需要暴露到包外的接口中，请使用 Optional 代替返回 null/-1，或者使用异常代替错误值。

对于本实验:

✧ 性能大部分时候是无关紧要的，请尽可能优雅地实现；

- ✧ 具体到本实验而言，输入保证所有字符都是合法的 ASCII 编码。你应尽可能使用 Character 中的方法，而非直接判断 ASCII 编码大小。

三 脚本工具

3.1 快速比对脚本

在完成实验时，我们可能会反复执行程序，输出结果，然后比对标准输出。然而人眼比对文本是一个较为繁琐的过程，我们在 scripts 文件夹下提供了两个快速比对的工具。

diff.py

Usage:

```
python -u diff.py /path/for/stdfile /path/for/srcfile
```

比对 /path/for/stdfile 与 /path/for/srcfile 的差异，忽略文件尾空行和行首尾空白符，若有差异则输出首行不匹配。

check-result.py

Usage:

```
python -u check-result.py n /path/for/stdfiles /path/for/outputfiles
```

其中 n 为实验 n 的序号，如 4 代表实验四：/path/for/stdfiles 与 /path/for/outputfiles 是标准文件的文件夹和输出文件的文件夹。

此脚本会比对本次实验与之前所有实验的输出文件和标准文件的差异. 特别地，对于实验四，我们直接输出了 rars 的输出，需要自行与程序的预期返回值比对。

脚本固定了 rars.jar 的路径，需要修改为你电脑上 rars.jar 的路径：

```
rars_path = "/home/test/rars.jar"
```

四 问题反馈

实验框架从一份已经过正确性检验的代码通过删改生成，框架的正确性已经过初步检验，注释部分可能存在部分错别字或措辞不当。

实验指导书可能存在部分错别字或措辞不准确，可能存在未说明清楚的部分，或某些部分如给出的示例实现部分未经过正确性验证，可能存在错误。若发现实验框架或实验指导书中存在任何错误，或你对其有任何疑问，欢迎 提交 issue 或直接向老师或助教反馈，我们将尽快解决。

第二部分

实验一　词法分析器的实现

1.1 实验题目

手工设计类 C 语言的词法分析器（可以是 c 语言的子集）。

1.2实验目的

1．加深对词法分析程序的功能及实现方法的理解；
2．对类 C 语言的文法描述有更深的认识，理解有穷自动机、编码表和符号表在编译的整个过程中的应用；
3．设计并编程实现一个词法分析程序，对类 C 语言源程序段进行词法分析，加深对高级语言的认识。
4. 实验学时数：2 学时。

1.3实验内容

编写一个词法分析程序，读取代码文件，对文件内自定义的类 C 语言程序段进行词法分析。 处理 C 语言源程序，过滤掉无用符号，分解出正确的单词，以

二元组形式存输出放在文件中。

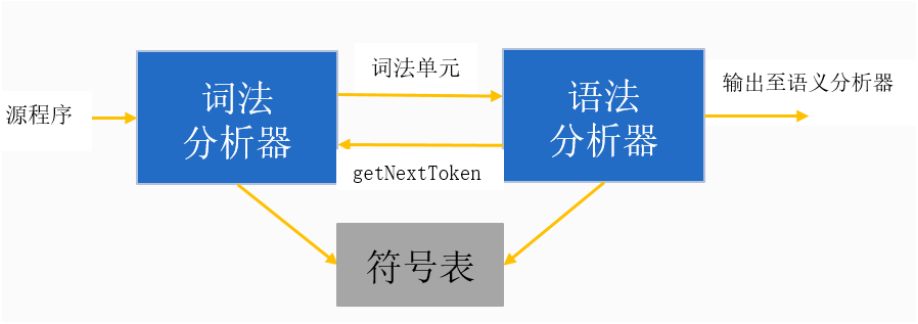
1．词法分析程序输入：以文件形式存放自定义的类 C 语言程序段;

2．词法分析程序输出：以文件形式存放的 Token 串和简单符号表；

3．词法分析程序输入单词类型要求：输入的 C 语言程序段包含常见的关键字，标识符，常数，运算符和分界符等。

1.4 分析与设计

要手工设计词法分析器，实现 C 语言子集的识别，就要明白什么是词法分析器，它的功能是什么。



词法分析是编译程序进行编译时第一个要进行的任务，主要是对源程序进行编译预处理（去除注释、无用的回车换行找到包含的文件等）之后，对整个源程序进行分解，分解成一个个单词。这些单词只有五类，分别是标识符、保留字、常数、运算符、界符。可以说词法分析面向的对象是单个字符，目的是把它们组成有效的单词（字符串），从而作为语法分析的输入来分析是否符合语法规则，并且进行语法制导的语义分析产生中间代码，进而优化并生成目标代码。

综上：词法分析器的输入是源代码字符流，输出是单词序列。

1) 什么是关键字（也是保留字）？

关键字是又程序语言定义的具有固定意义的标识符。例如 C 语言中的 if，else，for，while 都是保留字，这些字通常不用作一般标识符。

2) 什么是标识符？

标识符用来表示各种名字，如变量名，数组名，过程名等。

3) 什么是常数？

常数的类型一般是有整型，实型、布尔型、文字型等。

4) 什么是运算符？

运算符如+、-、*、/等等。

5) 什么是界符？

界符如逗号、分号、括号等等。

1.5 实验总体步骤

1.5.1 创建编码表

词法分析器的输出是单词序列，在 5 种单词种类中，关键字、运算符、分界符都是程序设计语言预先定义的，其数量是固定的。而标识符、常数则是由程序设计人员根据具体的需要按照程序设计语言的规定自行定义的，其数量可以是无穷多个。编译程序为了处理方便，通常需要按照一定的方式对单词进行分类和编码，在此基础上，将单词表示成二元组的形式（类别编码，单词值）。

单词名称	类别编码	单词值
int	1	–
return	2	–
=	3	–
,	4	–
Semicolon	5	–
+	6	–
–	7	–
*	8	–
/	9	–
(	10	–
)	11	–
id	51	内部字符串
IntConst	52	整数值
……	……	……
布尔常数	80	0 或 1
字符串常数	81	内部字符串

1.5.2 创建类 C 语言文法

多数程序语言单词的词法都能用正则文法来描述，基于单词的这种形式化描述会给词法分析器的设计与实现带来很大的方便，支持词法分析器的自动构造，比如 Flex、ANTLR 词法分析器生成工具。

1．正则文法表示

G=(V,T,P,S)，其中 V={S,A,B,C,digit,no_0_digit,letter,char},T={任意符号}，P 定义如下

约定：用 digit 表示数字：0,1,2,…,9; no_0_digit 表示数字：1,2,…,9;

用 letter 表示字母：A,B,…,Z,a,b,…,z,_

标识符：S→letter A　　A→letter A|digit A|ε

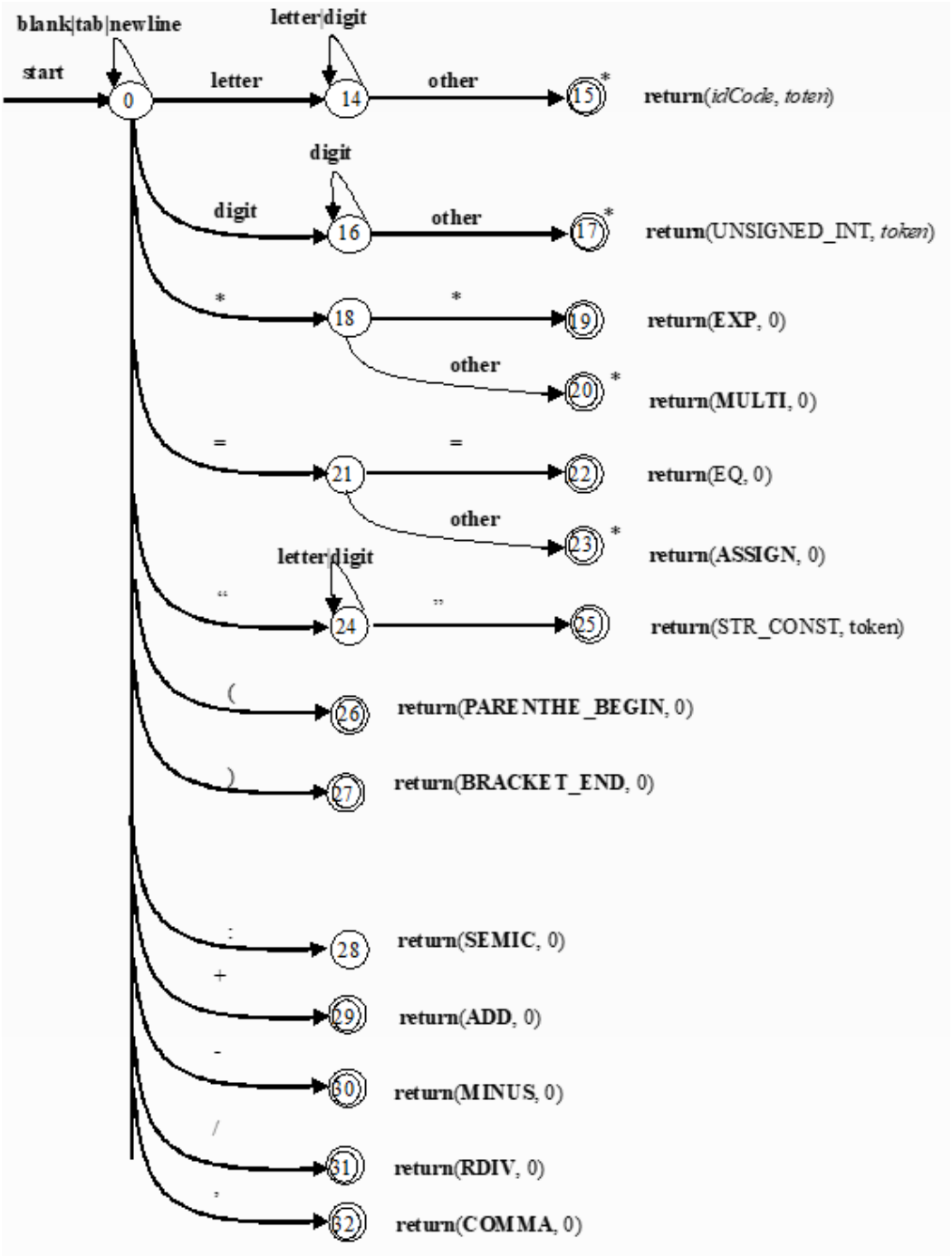
运算符、分隔符：S->B　B->=|*|+|-|/|(|);

整常数：S ->0|no_0_digitB　　B -> digitB | ε

1.5.3 有限自动机

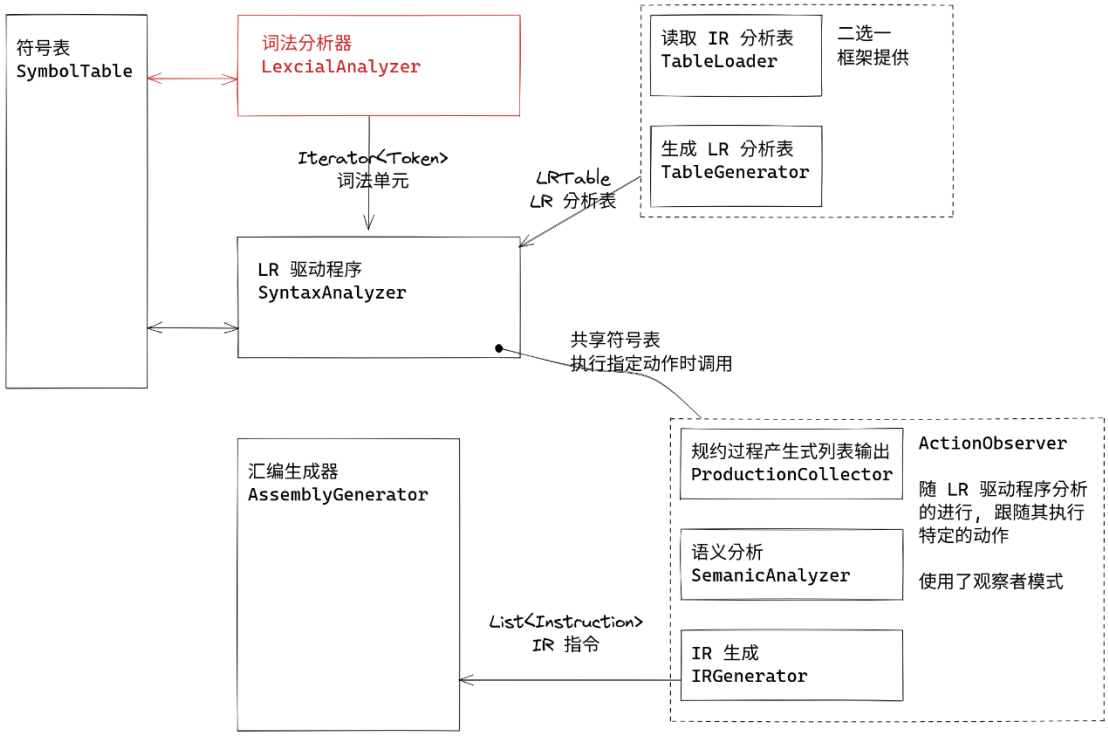
有限自动机识别的语言称为正则语言，有限自动机分为确定的有限自动机（DFA）和非确定的有限自动机（NFA）两种。DFA 和 NFA 都可以描述正则语言，DFA 规定只能有一个开始符号，且转移标记不能为空，代码实现较为方便，所以词法分析器使用 DFA 描述词法记号。

识别各类单词的状态转化图合并图：


blank|tab|newline
letter|digit
start
letter
other
return(idCode, token)
digit
digit
other
return(UNSIGNED_INT, token)
*
*
return(EXP, 0)
other
return(MULTI, 0)
=
=
return(EQ, 0)
other
return(ASSIGN, 0)
letter|digit
"
"
return(STR_CONST, token)
(
return(PARENTHE_BEGIN, 0)
)
return(BRACKET_END, 0)
;
return(SEMIC, 0)
+
return(ADD, 0)
-
return(MINUS, 0)
/
return(RDIV, 0)
,
return(COMMA, 0)

1.6 实验框架概述

1.6.1 编译程序整体框架图



实验一词法分析程序核心类是 LexcialAnalyzer 类，该类完成的主要功能包括源程序代码读入，词法分析获取 token 列表以及 token 列表写入文件。

在本次实验中，你将要使用自动机的写法实现一个简单的，特定的词法分析器。具体要实现的词法规则如下：

类别	正则表达式
int	int
return	return
=	=
,	,
Semicolon	;
+	+
-	-
*	*

/	/
(	(
)	)
Id	[a-zA-Z_][a-zA-Z]*
IntConst	[0-9]+

每个词法单元之间可能有任意的空白字符（\s，或 [\t\r\n]），这些字符应当忽略。

你需要根据读入的源语言的代码文本，生成词法单元迭代器并正确地将源语言中的每个标识符插入到符号表中，不要求记录每个词法单元的行号，起始列号，结束列号，不要求处理注释。

框架中已包含词法单元（Token）以及符号表（SymbolTable）的实现。从文件中读取内容、将生成的词法单元输出到文件中、将符号表输出到文件中的过程亦已实现。

1.6.2 词法分析程序的输入

1. 词法分析程序读入文件

data/in

├── coding_map.csv　　# 码点文件（编码表）

└── input_code.txt　　# 输入的代码

#coding_map.csv 码点文件

也就是编码表。

```
1 int
2 return
3 =
4 ,
5 Semicolon
6 +
7 -
8 *
9 /
10 (
11 )
51 id
52 IntConst
```

#关键字列表

创建关键字列表：关键字可以写入文件中，读文件读入到存储的数据结构；或者直接在程序写入数据结构存储。

注意：这个数据结构模板代码中没有，同学自己创建。

#input_code.txt 编译器要解析的源代码

把源程序代码读入到输入缓冲区，根据有限自动机的状态转移识别一个个单词 Token。

```
int result;
int a;
int b;
int c;
a = 8;
b = 5;
c = 3 - a;
result = a * b - ( 3 + b ) * ( c - a );
return result;
```

1.6.3 词法分析程序的输出

1. 词法分析程序输出结果文件

data/out

├── old_symbol_table.txt　　# 符号表

└—— token.txt　　# 词法单元列表

输出的文件内容具体是什么可以参考 data/std 目录下的同名文件。

old_symbol_table.txt 符号表

编译器用符号表来记录、收集和查找出现在源程序中的各种名字及其语义信息。每当源程序中出现一个新的名字时，则向符号表中填入一个新的表项来记录该名字；当在源程序中发现某个名字的新属性时，要找到该名字所对应的符号表表项，在表中记录其新发现的属性信息。因此，从数据结构的角度来看，符号表是一个以名字为关键字来记录其信息的数据结构，支持插入和查找两个基本操作。在词法分析阶段主要是往符号表里插入名字信息。

符号表的组织结构可以是线性表或者散列表。首先设计者可能会想到选择数组作为符号表的具体物理实现结构，在符号表的规模较小的情况下，线性结构可以满足。当设计的编译器规模较大时，可以考虑引入散列表来提高查找插入的效率。

```
(a, null)
(b, null)
(c, null)
(result, null)
```

#token.txt 词法单元列表

词法分析程序通过对输入源程序字符串分割识别，生成一系列独立的单词串，这些单词串包括标识符、关键字、常数、运算符、分界符。

其中关键字、运算符、分界符一符一码，标识符统一一个编码，常量一类一码。词法分析程序识别出单词串以二元组的形式存储，每个二元组由种别码和属性值组成，对于一符一码的单词我们仅记录其种别码，对于标识符、常量这类单词需要同时记录其属性值。

二元组：(种别码, 属性值)

每一个二元组在框架代码中是一个 Token 类实例，请参考框架代码。

以下是本实验中所给出的示例输入源程序经过词法分析程序处理后输出的二元组序列，注意($,)是在二元组序列末尾加上代表 EOF 的 token。

(int,) *(id,result)* *(Semicolon,)* *(int,)* *(id,a)* *(Semicolon,)* *(int,)* *(id,b)* *(Semicolon,)* *(int,)* *(id,c)* *(Semicolon,)* *(id,a)* *(=,)* *(IntConst,8)* *(Semicolon,)* *(id,b)* *(=,)* *(IntConst,5)* *(Semicolon,)* *(id,c)* *(=,)* *(IntConst,3)* *(-,)* *(id,a)* *(Semicolon,)*	*(id,result)* *(=,)* *(id,a)* *(*,)* *(id,b)* *(-,)* *((,)* *(IntConst,3)* *(+,)* *(id,b)* *(),)* *(*,)* *((,)* *(id,c)* *(-,)* *(id,a)* *(),)* *(Semicolon,)* *(return,)* *(id,result)* *(Semicolon,)* *($,)*

1.7 实验框架设计简介

1.7.1 数据结构说明

1. Term

该类为所有文法符号（终止符与非终止符）的基类。

2. Token

词法单元的实现，词法单元（Token）是词法分析的结果。

词法分析从源程序文件的文本流中识别出字符， 将一个或多个字符根据特定的规则识别出来单独的单词，由这些单词本身和单词的种别码组成词法单元。

一个词法单元逻辑上由两个部分组成：词法单元的类型（种别码），词法单元的单词文本。

```
class Token {
   private TokenKind kind;   //种别码
   private String text;    //单词值
}
```

然而，对于大部分简单的词法单元而言，并不需要保存词法单元代表的文本。在这种情况下，我们将此 Token 的 text 设置为""（空字符串），而不是 null。为了方便构造这两种 Token，我们提供了两个静态方法以供使用：

```
class Token {
   public static Token simple(TokenKind kind) {
      return new Token(kind, "");   //不需要保存单词值
   }

   public static Token normal(TokenKind kind, String text) {
      return new Token(kind, text);      //需要保存单词值
   }
}
```

在输出二元组序列末尾，添加代表 EOF 的二元组单元($,)：

```
    public static Token eof() {
          return new Token(TokenKind.eof(), "");
    }
```

3. TokenKind

词法单元类别（编码表）。

词法单元的类别一般实现为一枚举，将所有可能的词法单元类型硬编码到代码中。你也可以在运行时从一个文件中读入所有可能的词法单元类别。

coding_map.txt 文件：

```
1 int
2 return
3 =
4 ,
5 Semicolon
6 +
7 -
8 *
9 /
10 (
11 )
51 id
52 IntConst
```

考虑到代码可读性,我们最终选择使用某个词法单元类别的名字来唯一标识该类别，而将所谓"码点"作为词法单元类别的附加信息，仅供参考使用。于是我们的词法单元类别设计如下:

```
class TokenKind {
    private String id;
    private int code;
}
```

鉴于使用字符串作为唯一标识符有极大的因为拼写错误而出错的可能， 我们会在编译器运行时从 coding_map.txt 中读入所有可能的词法单元类别标识符,然后在每次构造词法单元类别时进行检查:

```
class TokenKind {
    public static TokenKind fromString(String id) {
        // 检查 id 是否被允许使用 (即是否在 coding_map.txt 中有定义)
        if (allowed == null || !allowed.containsKey(id)) {
            throw new RuntimeException("Illegal Identifier");
        }

        return allowed.get(id);
    }

    // 禁止外界构造新的 TokenKind
    private TokenKind(String id, int code) {
        // ...
    }
}
```

同时,为了进一步方便 Token 的构造，我们也为 simple 和 normal 提供了直接使用词法单元类别标识符的重载版本:

```
class Token {
    public static Token simple(String tokenKindId) {
        return simple(TokenKind.fromString(tokenKindId));
    }

    public static Token normal(String tokenKindId, String text) {
        return normal(TokenKind.fromString(tokenKindId), text);
    }
}
```

另外,TokenKind 作为 Term 的子类,代表文法中的非终结符。详见实验二指导书。

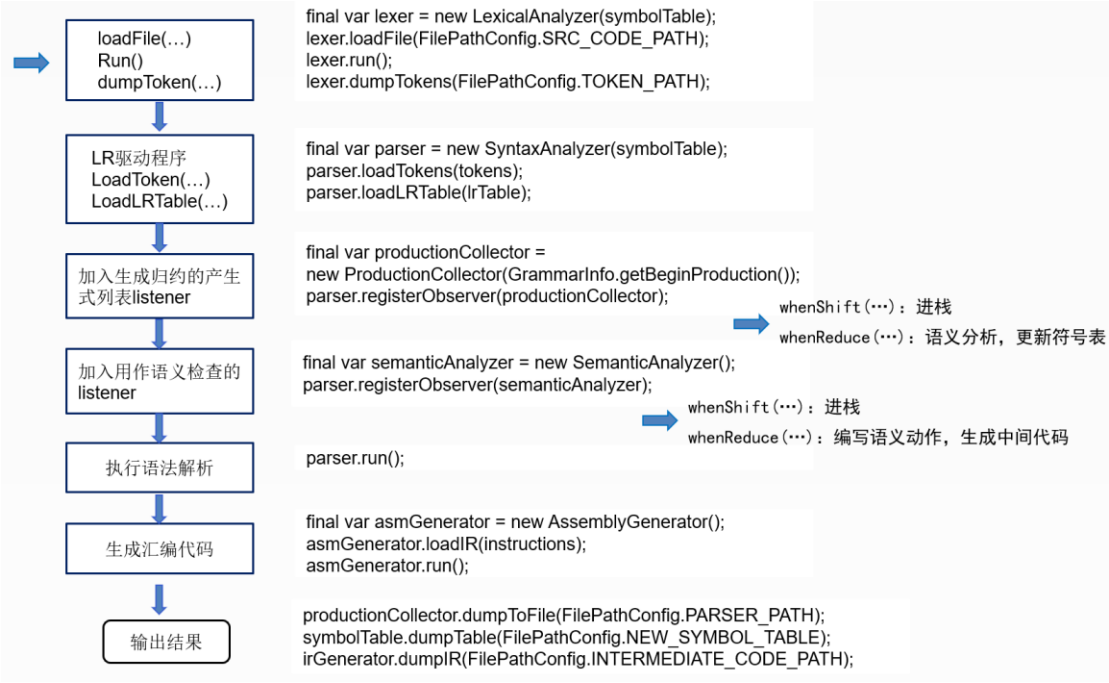
4. SymbolTable

符号表

在实验一中,你仅需要在读到一个标识符时(即生成类别为 id 的词法单元时)，检测符号表中是否已含有该标识符,若无则向符号表添加该标识符即可。注意：该函数在 SymbolTable 并未定义，需要自行编写。

```
// 在你的实现的某个地方...
if (!symbolTable.has(identifierText)) {
    symbolTable.add(identifierText);
}
```

1.7.2 主程序流程说明



以上是模版代码项目中 Main.java 文件中 main 函数代码流程，具体代码请大家参考. \template\src\cn\edu\hitsz\compiler\Main.java 文件。

main 函数流程说明如下：

①构建符号表，用来存储源语言中的标识符；

②开始词法分析，loadFile 方法用来 load 源语言源代码；接着执行 run 方法，也就是词法分析过程，dumpTokens 方法把词法分析的结果 tokens 输出到文件，dumpTable 方法把存储到符号表中的标识符输出到文件。

③读取 LR 分析表（LR 分析表可以用编译工作台工具生成），框架代码中也提供了 LR 分析表生成的代码，如需使用框架代码生成请自行启用相关代码。

④加载 LR 分析驱动程序；loadTokens 方法读入词法分析的结构 tokens，loadLRTable 读入 LR 分析表，这两个方法的定义在语法分析程序 SyntaxAnalyzer.java 中，需要开发者自己实现。

⑤框架使用了观察者设计模式，语法分析中需要注册 ProductionCollector 用于收集语法分析中归约产生式的观察者接口，执行 run 方法实现语法分析。注意：在完成实验二时，需要先把加入语义检查和 IR 生成的观察者相关代码注释起来，否则因为这其中有一些代码暂未实现，会导致报异常。

```
// 加入用作语义检查的 Observer
final var semanticAnalyzer = new SemanticAnalyzer();
parser.registerObserver(semanticAnalyzer);

// 加入用作 IR 生成的 Observer
final var irGenerator = new IRGenerator();
parser.registerObserver(irGenerator);
```

⑥框架使用了语法制导翻译的语义分析和中间代码生成，所以语义分析应该是和语法分析过程同步，语法分析每执行一次归约将该次归约的产生式完成语义分析。同样使用观察者模式，注册语义检查和 IR 生成的观察者。

⑦parser.run 方法执行，完成语法分析和语义分析，在对应在观察者中收集归约产生式和完成语义检查以及 IR 生成。

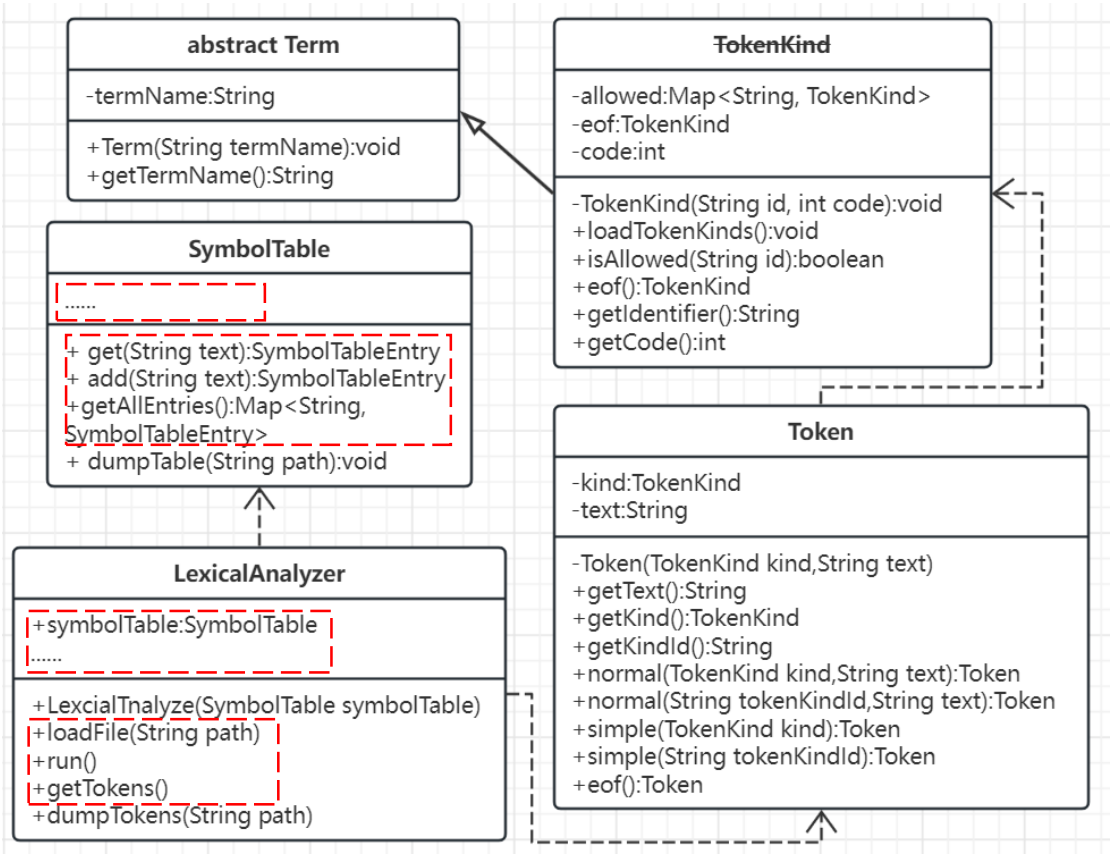
⑧在各观察者中输出结果；

```
//输出语法分析过程中归约的产生式列表
productionCollector.dumpToFile(FilePathConfig.PARSER_PATH);
//输出语义分析语义检查后更新的符号表
symbolTable.dumpTable(FilePathConfig.NEW_SYMBOL_TABLE);
//输出中间代码
irGenerator.dumpIR(FilePathConfig.INTERMEDIATE_CODE_PATH);
```

⑨loadIR 把生成的中间代码 load 到程序中，执行 run 方法，生成汇编代码。

1.8 实验框架代码说明

1.8.1 词法分析程序框架代码 UML 图



以上是词法分析程序相关类 UML 图，相关说明：

①LexicalAnalyzer 类：这个类是词法分析核心类，开发者需要完成的有 loadFile，run、getTokens 三个函数。loadFile 方法把文件中的源语言源代码 load 到程序中，可以直接采用完整读入的方法读入存储为一个字符串。run 方法执行此番分析，返回一个 token 序列，开发者需要自定义一个列表来存储 token 序列。getTokens 方法从词法分析过程中获取 Token 列表。

②SymbolTable 类：符号表类；词法分析中，当识别到一个标识符，需要把这个标识符名字添加到符号表中。在 SymbolTable 类中，开发者需要实现 has、add、get、getAllEntries 这四个方法，在符号表类中每一个标识符占一行（包括标识符名字和其他属性）也就是一个符号表条目。开发者需要自定义用来存储这些条目的数据结构，可以是一个集合，集合的数据类型应该包含标识符名字和其他属性，其中标识符属性可以使用 SymbolTableEntry 这个模版代码设计好的数据结构。

③Token 类：每一个 token 对象就是一个单词的包装，词法分析结果是一个 token 序列，所以在①中存储词法分析输出结果的数据结构可以使用 Token 类。每一个 token 对象包含单词值本身也就是 text 文本和种别码，getText 方法返回单词本身，getKind 返回 token 的类型，getKindId 返回 token 类型的文本表示；

1.8.2 主要类说明

1．LexicalAnalyzer 类

```
/**
    * 从给予的路径中读取并加载文件内容
    * @param path 路径
    */
public void loadFile(String path) {
      // TODO: 词法分析前的缓冲区实现
      //可自由实现各类缓冲区
      // 或直接采用完整读入方法
      throw new NotImplementedException();
}
```

```
/**
    * 执行词法分析，准备好用于返回的 token 列表 <br>
    * 需要维护实验一所需的符号表条目，而得在语法分析中才能确定的符号表条目的成员可以先设置为 null
   */
public void run() {
      // TODO: 自动机实现的词法分析过程
      throw new NotImplementedException();
}
```

```
/**
    * 获得词法分析的结果，保证在调用了 run 方法之后调用
    *
    * @return Token 列表
    */
   public Iterable<Token> getTokens() {
      // TODO: 从词法分析过程中获取 Token 列表
      // 词法分析过程可以使用 Stream 或 Iterator 实现按需分析
      // 亦可以直接分析完整个文件
      // 总之实现过程能转化为一列表即可
      throw new NotImplementedException();
   }
```

2. SymbolTable 类：

```
/**
     * 获取符号表中已有的条目
     *
     * @param text 符号的文本表示
     * @return 该符号在符号表中的条目
     * @throws RuntimeException 该符号在表中不存在
     */
    public SymbolTableEntry get(String text) {
        throw new NotImplementedException();
    }
```

```
/**
     * 在符号表中新增条目
     *
     * @param text 待加入符号表中的新符号的文本表示
     * @return 该符号在符号表中对应的新条目
     * @throws RuntimeException 该符号已在表中存在
     */
    public SymbolTableEntry add(String text) {
        throw new NotImplementedException();
    }
```

1.8.3 补充说明

框架代码下载地址

https://gitee.com/hitsz-cslab/Compiler/releases/tag/2022F.0.1

在线指导书

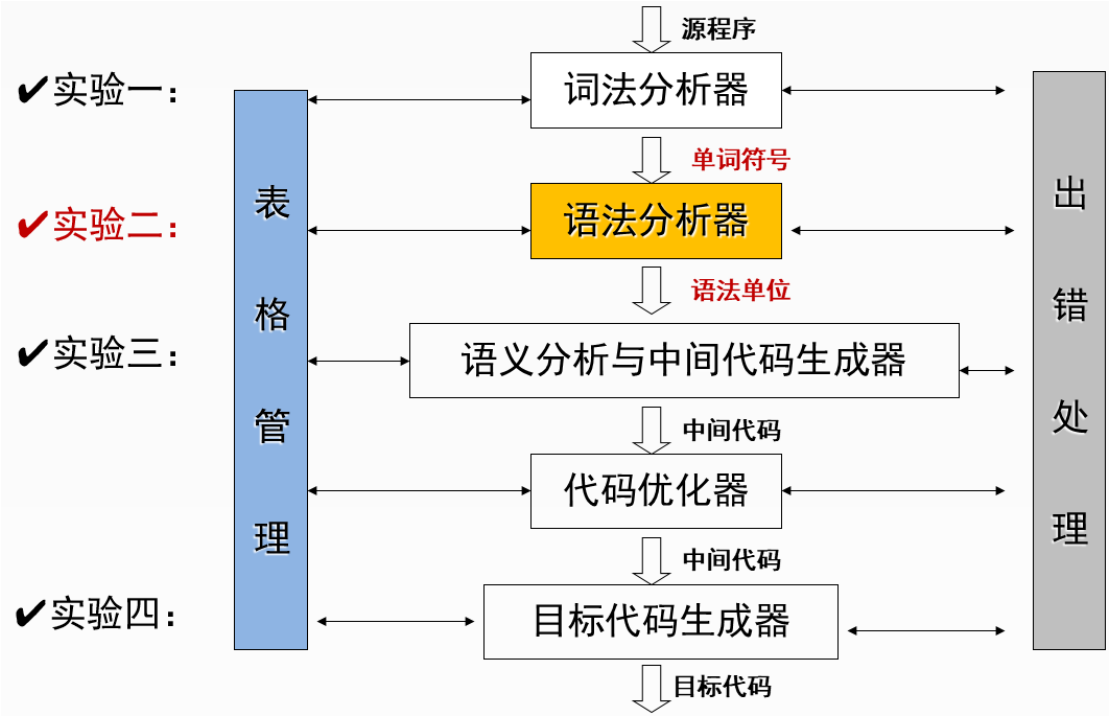
https://compiler-6bi.pages.dev/

说明：

1. 框架模板仅支持声明语句、赋值语句和简单算术表达式。同学想开发的编译器支持更多语法功能需自行修改以下文件：
   - ✧ 源代码文件：input_code.txt；
   - ✧ 编码表文件：coding_map.txt;
2. 注意事项：
   - ✧ 禁止使用 java.util 提供的正则表达式的文本处理工具来完成词法分析；

- ✧ 禁止直接查找空格来分割单词；

实验二 自底向上的语法分析 LR(1)



2.1 实验目的

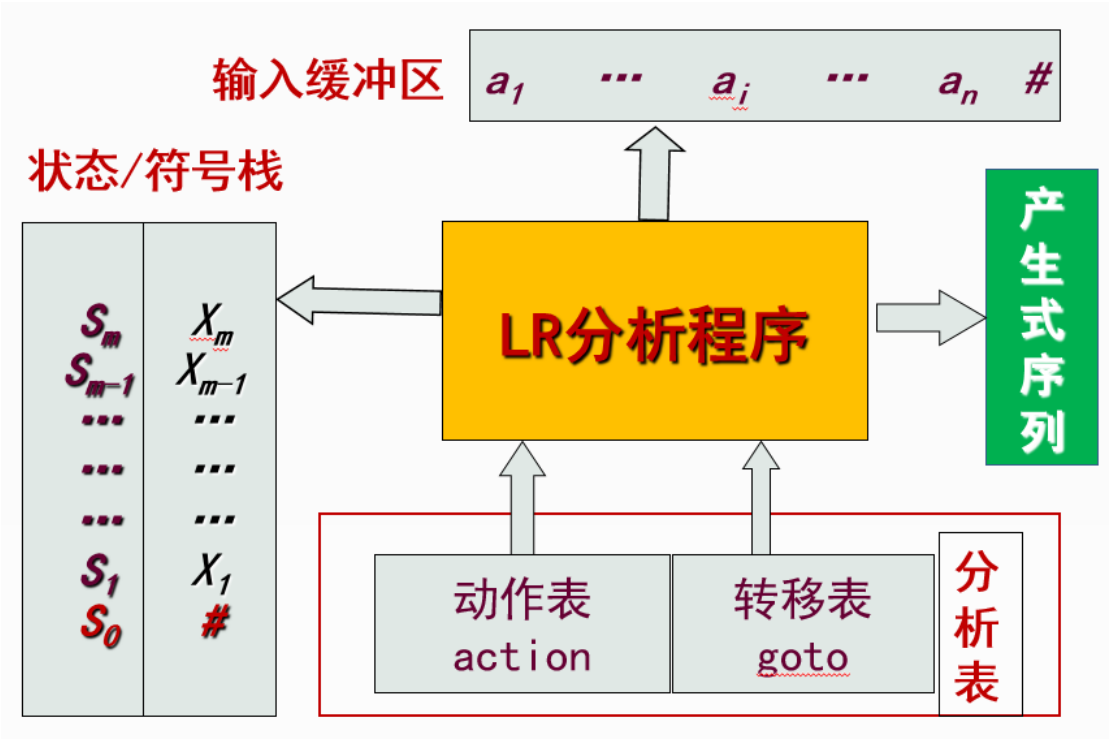
1. 深入了解语法分析程序实现原理及方法。
2. 理解 LR(1)分析法是严格的从左向右扫描和自底向上的语法分析方法。
3. 实验学时数：8 学时。

2.2 实验内容

1. 利用 LR(1)分析法，设计语法分析程序，对输入单词符号串进行语法分析；
2. 输出推导过程中所用产生式序列并保存在输出文件中；
3. 较低完成要求：实验模板代码中支持变量申明、变量赋值、基本算术运算的文法；
4. 较优完成要求：自行设计文法并完成实验。
5. 要求：实验一的输出作为实验二的输入。

说明：实验一测试用例较复杂的情况下，实验二也需要同学自行定义文法并生成 LR（1）分析表来完成实验；

2.3LR 语法分析器的总体结构



➢ 语法分析器的两个输入：

1. 实验一词法分析器输出的 Token 串；
2. LR 分析表（包括动作表（Action）和转移表（GOTO））；

➢ 语法分析器的两个栈：

1. 状态栈：记录语法分析过程中查 goto 表获得的状态；
2. 符号栈：记录语法分析过程中需要压栈的终结符和非终结符；

➢ 语法分析程序的四种动作

1. 移进：

   将下一输入符号移入栈

2. 归约：

   用产生式左侧的非终结符替换栈顶的句柄（产生式右部）

3. 接受：分析成功
4. 出错：出错处理

2.4 实验总体步骤

1. 定义描述程序设计语言语法的文法，并编写拓广文法；
2. 构造 LR(1)分析表（借助编译工作台完成）；

3. 设计数据结构读入文法、LR(1)分析表；
4. 编写 LR(1)驱动程序完成 LR 分析器移进、归约、出错、接受四个动作；
5. 输入：实验一输出的单词符号串、LR(1)分析表、文法；
6. 输出：产生式序列并保存在文件中。

2.4.1 编写拓广文法

（定义描述程序设计语言语法的文法，并编写拓广文法；以下文法是延续实验一源代码所定义的参考文法）

```
P -> S_list;
S_list -> S Semicolon S_list;
S_list -> S Semicolon;
S -> D id;
D -> int;
S -> id = E;
S -> return E;
E -> E + A;
E -> E - A;
E -> A;
A -> A * B;
A -> B;
B -> ( E );
B -> id;
B -> IntConst;
```

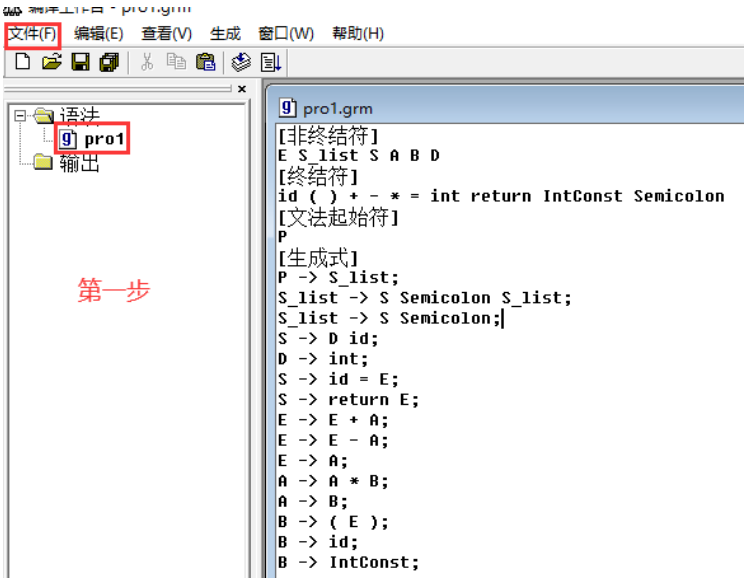
2.4.2 构造 LR(1)分析表

构造 LR(1)分析表可以借助编译工作台完成。

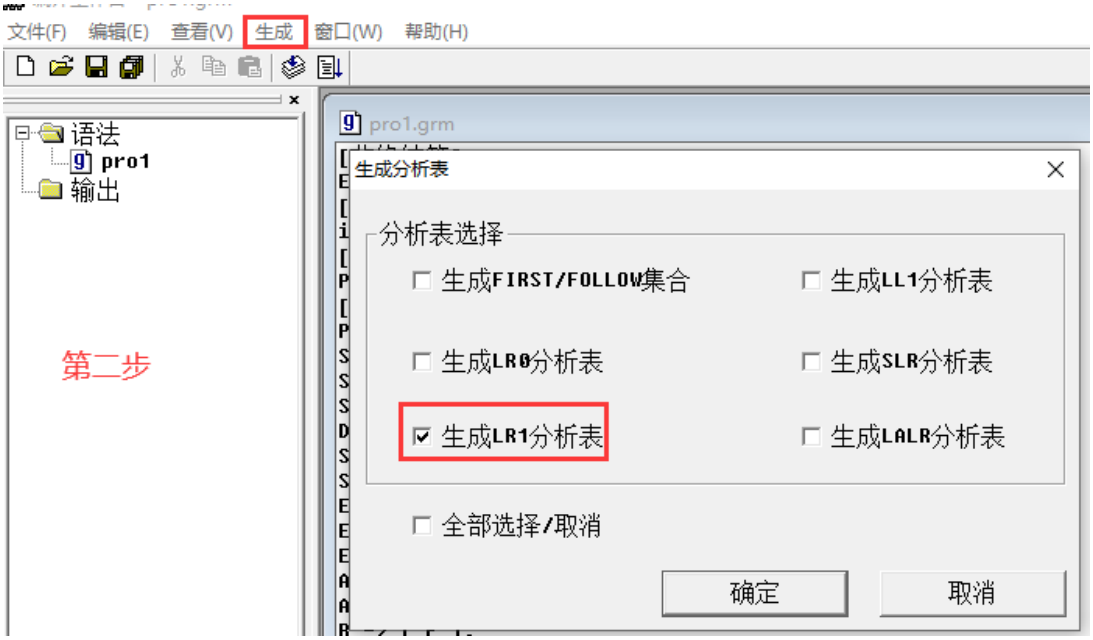
1) Windows 环境安装编译工作台

下载链接：https://gitee.com/hitsz-cslab/Compiler/releases/tag/2022F.0.1；

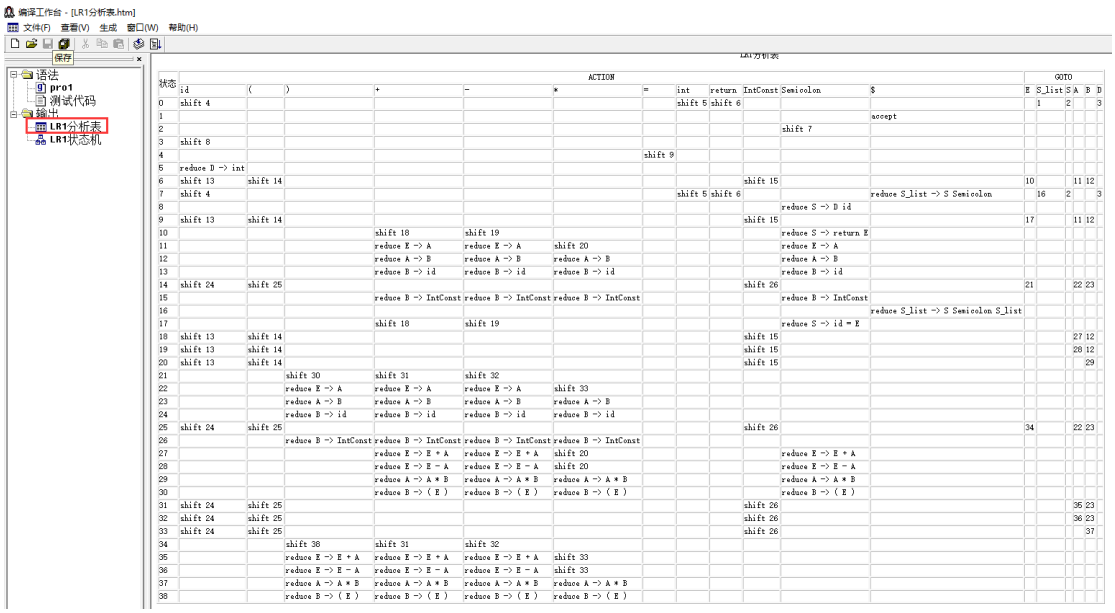
2) 创建一个语法文件，按照文件模板填写；



3）点击菜单“生成”——“生成分析表” ；

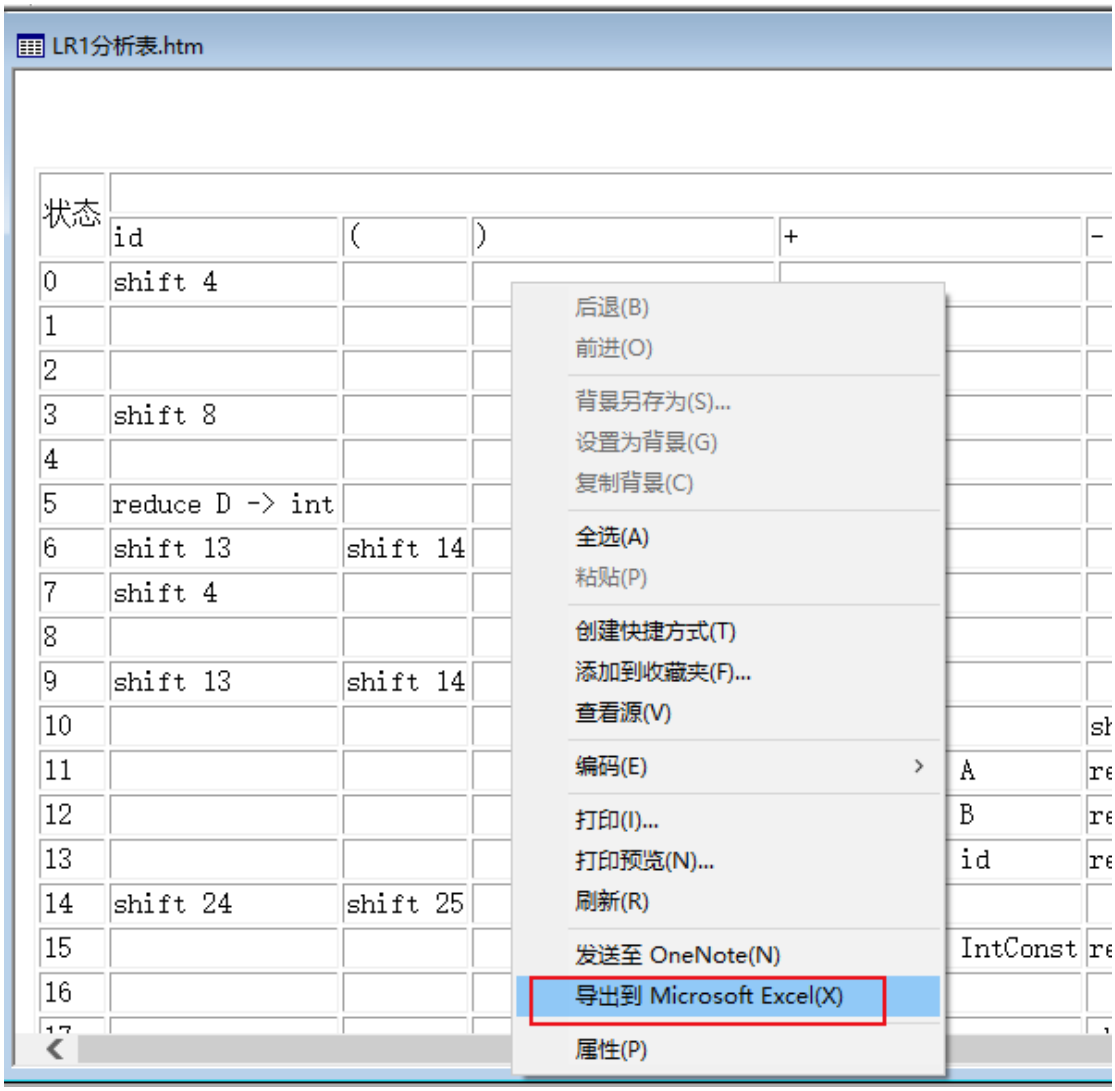


4）参考文法生成的 LR(1)分析表——包括 ACTION 和 GOTO：



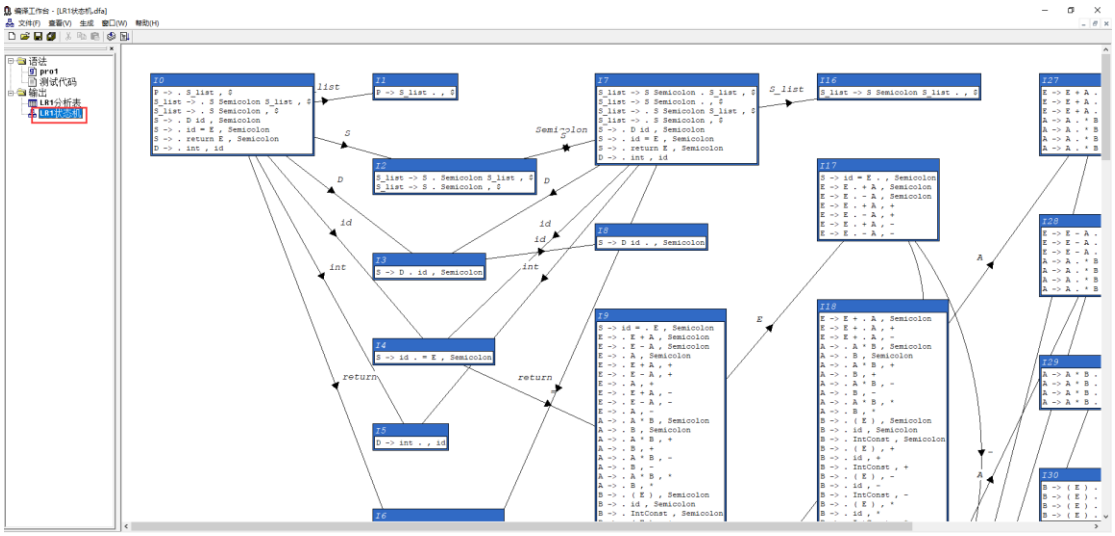
5）LR(1)分析表的导出

鼠标右键点击编译工作台生成的 LR(1)分析表，在弹框中选中导出到 Mircosoft Excel(X)，可以将编译工作台生成的分析表导出到 Excel 文件中。



状态	id	(	)	+	-
0	shift 4				
1					
2					
3	shift 8				
4					
5	reduce D -> int				
6	shift 13	shift 14			
7	shift 4				
8					
9	shift 13	shift 14			
10					sh
11				A	re
12				B	re
13				id	re
14	shift 24	shift 25			
15				IntConst	re
16					

6）编译工作台生成的 LR(1)状态机：



7）借助 LR 分析表动态分析句子

举例，分析如下源代码：

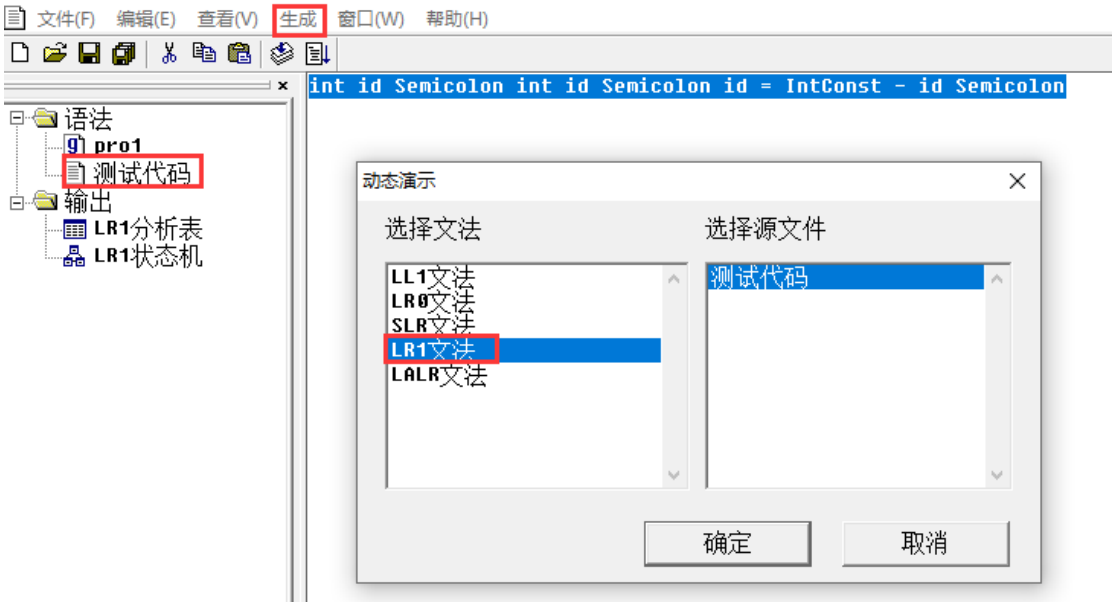
int a; int c; c = 3-a;

以上源代码经过实验一编码表内种别码替换后：

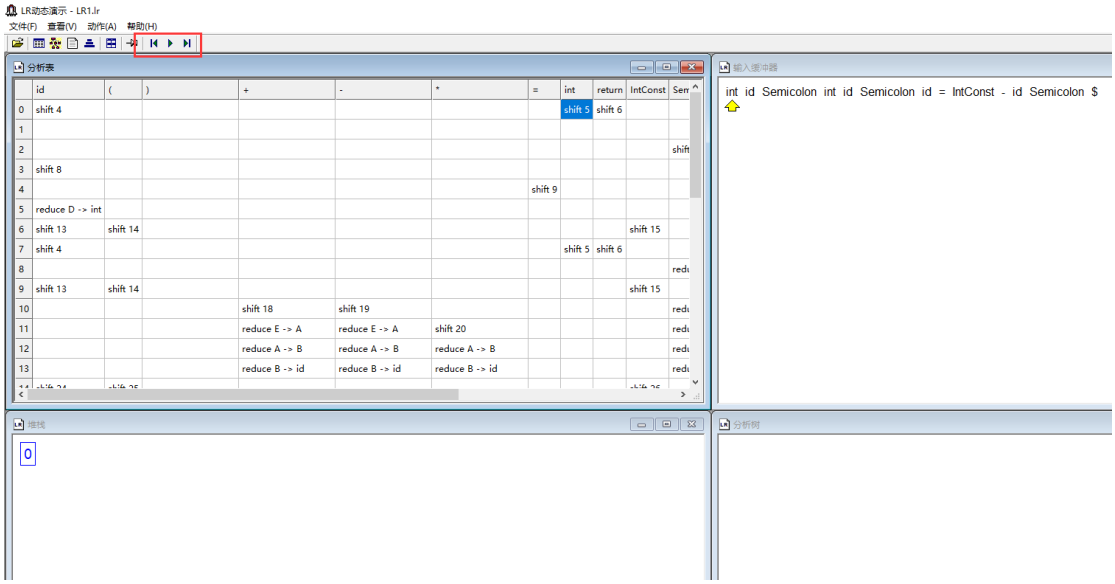
int id Semicolon int id Semicolon id = IntConst - id Semicolon

操作方法：

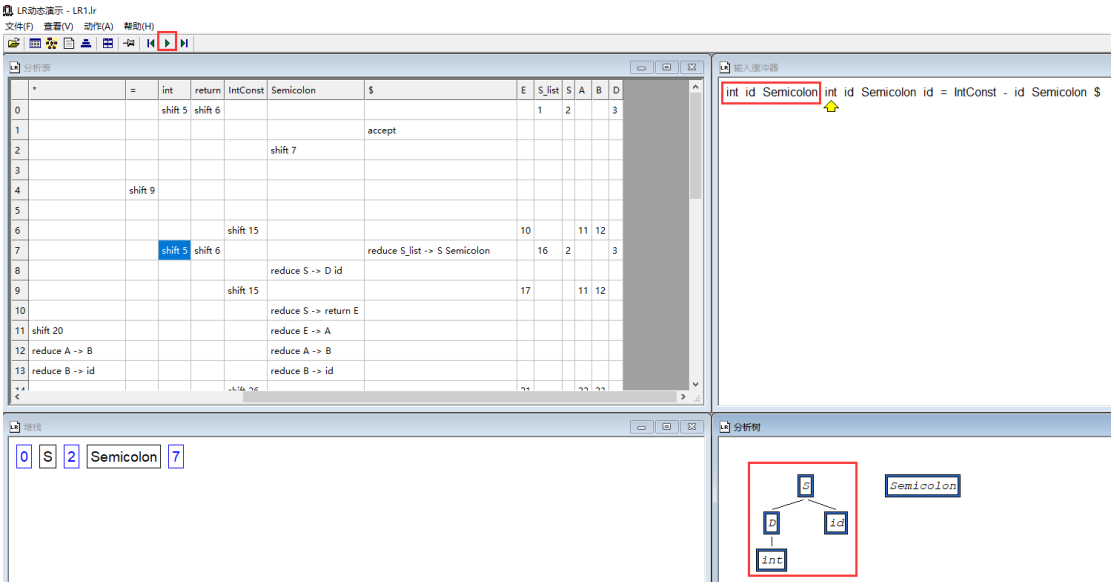
- 新建一个源文件，输入要分析的源代码；
- 点击生成——动态分析；



编译工作台启动动态分析句子：

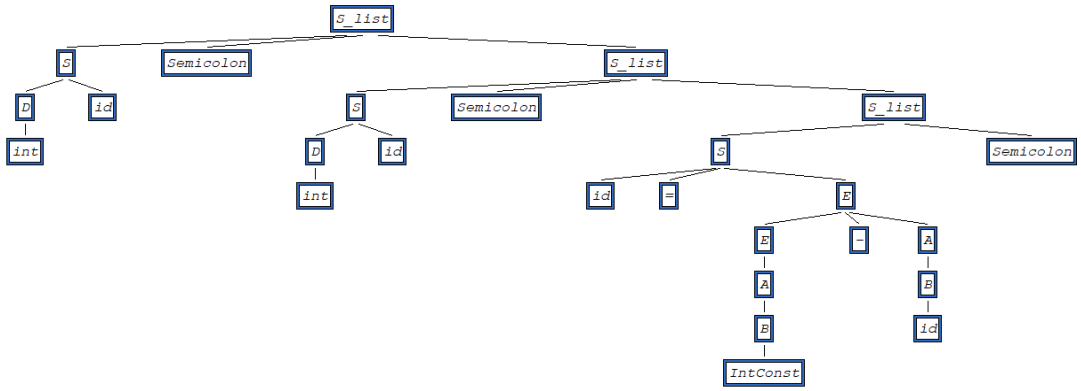


编译工作台分析句子中：



我们发现动态分析中符号和状态在同一个栈中，动态分析的过程可以动态的展示句子在语法分析中建立语法树的过程。

待分析完成，生成完整的语法树：



2.4.3 数据结构

1. LRTable

LR 分析表，实际上就是一张类似自动机的状态转移图，当在当前状态遇到某某符号时便执行动作并转移。归约时就根据“Action 表”里写的产生式生成非终结符，然后将生成的非终结符作为“输入符号”根据“Goto 表”来确定转移；移入时就直接根据“Action 表”转移到对应状态。

我们采用一种更接近邻接表而非邻接矩阵的方式来提供对 ACTION 与 GOTO 的访问；虽然，我们亦在 LRTable 中提供了“更像邻接矩阵”的访问接口：

```
class LRTable {
    private final List<Status> statusInIndexOrder;
    private final List<TokenKind> terminals;
    private final List<NonTerminal> nonTerminals;

    /**
     * @return 起始状态
     */
    public Status getInit() {
        // return initStatus;
        return statusInIndexOrder.get(0);
    }
    ……
}
```

理论上一个 LRTable 只需要保存起始状态(initStatus)即可。但是为了方便将整个表输出为 csv 查看，我们于 LRTable 中保存了一些多余的状态，而你的实现不应该使用这些信息。

提示：我们可以调用 LRTable 类的 getInit()方法返回 Status 来初始化状态栈（如果你的状态栈数据结构是 Status）。

2. Status

表示 LR 分析表中的一个状态，这是一个 record 类。

```
public record Status(int index, Map<TokenKind, Action> action, Map<NonTerminal, Status>
goto_)
```

```
public Action getAction(TokenKind terminal) {
        return action.getOrDefault(terminal, Action.error());
}

public Action getAction(Token token) {
        return getAction(token.getKind());
       }
}
```

我们可以根据当前状态栈栈顶状态和下一个待读入 token 来调用该类的 getAction(token)方法查询 LR(1)分析表，此方法返回一个 Action 对象也就是 LR 分析表的一个动作。使用 Action 对象的 getKind()方法返回 ActionKind 枚举类型，

判断要执行的动作是“移进”、“归约”、“接受”、“出错”。

```
public Status getGoto(NonTerminal nonTerminal) {
        return goto_.getOrDefault(nonTerminal, Status.error());
    }
```

当执行归约动作时，调用 Status 类的 getGoto(nonTerminal)方法返回一个 Status 对象，此函数调用传递参数 nonTerminal 是一个非终结符。(也就是在执行归约动作时符号栈弹出产生式右部长度相同个数的终结符或非终结符出栈，产生式左部的非终结符入栈；状态栈也弹出对应长度个数状态，此时符号栈和状态栈长度不一致，需要再根据当前符号栈（此时符号栈栈顶是一个非终结符）和状态栈栈顶再去查一次 goto 表，查到的状态压入状态栈)。

3. Action

代表 LR 分析表 action 表中的一个动作。

```
enum ActionKind {
    Shift,      // 移入
    Reduce,     // 规约
    Accept,     // 接受
    Error,      // 出错
}
```

LR 分析中的动作有四种，动作的类别使用枚举来表示。

```
class Action {
    private ActionKind kind;
    private Production production;  // 当且仅当动作为归约时它非空
    private Status status;         // 当且仅当动作为移入时它非空
}
```

你应当采用各个访问方法来访问这些信息， 这些访问性方法会检查动作类别并确定返回值非空。

```
class Action {
    public ActionKind getKind() {
        return kind;
    }
.....
}
```

使用 Action 对象的 getKind()方法返回 ActionKind 枚举类型，返回值可能是

“Shift”、“Reduce”、“Accept”、“Error”中任一个。

```
/**
 * @return 获得移入动作的状态
 * @throws RuntimeException 动作不是移入动作
 */
public Status getStatus() {
    if (kind != ActionKind.Shift) {
        throw new RuntimeException("Only shift action could hava a status");
    }
    assert status != null;
    return status;
}
```

语法分析器查询 LR(1)分析表的 Action 表时，查到动作包括“移进”指令和同时要移进状态栈的状态（比如 Shift 1：表示移进动作，同时要移进 token 到符号栈和状态 1 到状态栈),这是一个 Action 对象。调用 Action 类的 getStatus()函数返回一个 Status 对象，也就是获取到当前移进动作时的需要压入状态栈的状态。

```
/**
 * @return 获得归约动作的产生式
 * @throws RuntimeException 动作不是规约动作
 */
public Production getProduction() {
    if (kind != ActionKind.Reduce) {
        throw new RuntimeException("Only reduce action could have a production");
    }
    assert production != null;
    return production;
}
```

语法分析器在执行归约动作时，使用 Action 类的 getProduction()方法返回一个 Production 对象，获得执行归约动作时所用到的产生式。

4．Production

```
/**
 * 表示一条产生式，您不应该改动此文件
 * <br>
 * 产生式的等价性由其 index 唯一确定. 即，两条产生式 equals 当且仅当它们 index 相等.
 * @param index 该产生式的索引，为其在 grammar.txt 文件内的行号，从 1 开始
 * @param head 该产生式的头
 * @param body 该产生式的体
 */
public record Production(int index, NonTerminal head, List<Term> body)
```

Production 是一个 record 类，可以用 production.body().size()的方法来获取产生式右部长度，这个长度数据在语法分析器执行归约动作时需要从符号栈和状态栈栈顶出栈对应个数的符号和状态时使用。

2.5实验框架代码

2.5.1 LR(1)主程序流程

1. 利用编译工作台生成 LR(1)分析表；
2. 替换模板中的 LR1_table.csv（如果不修改 data->in->grammar.txt 文件也就是不修改拓广文法也可以不用替换 LR1_table.csv 文件）；
3. 调用 TableLoader 类的 load 方法读入编译工作台生成的 LR(1)表；

```
final var tableLoader = new TableLoader();
final var lrTable = tableLoader.load(FilePathConfig.LR1_TABLE_PATH);
```

4. 加载 LR 驱动程序，Load Token 和 Load LRTable;

```
final var parser = new SyntaxAnalyzer(symbolTable);
(lrTparser.loadTokens(tokens); //TODO
parser.loadLRTableable(lrTable); //TODO
```

loadTokens(tokens)函数把实验一生成的 token 串 load 到语法分析类中，loadLRTable(lrTable)函数把编译 LR(1)分析表 load 到语法分析类中。

注意：虽然创建 SyntaxAnalyzer 对象时传入了 symbolTable 参数，但实验二中无需操作符号表。

5. 加入生成归约产生式列表的 listener；

```
final var productionCollector = new ProductionCollector(GrammarInfo.getBeginProduction());
parser.registerObserver(productionCollector);
```

产生式数据结构类：Production（模板已完成），GrammarInfo 里完成了产生式加载；

6.执行语法解析

```
parser.run();      //TODO
```

在此方法中编写 LR 驱动程序，查询 LR(1)分析表，根据查询结果判断是 Shift、Reduce、Accept，并执行相应动作，同时调用 callWhenInShift、callWhenInReduce、callWhenInAccept。

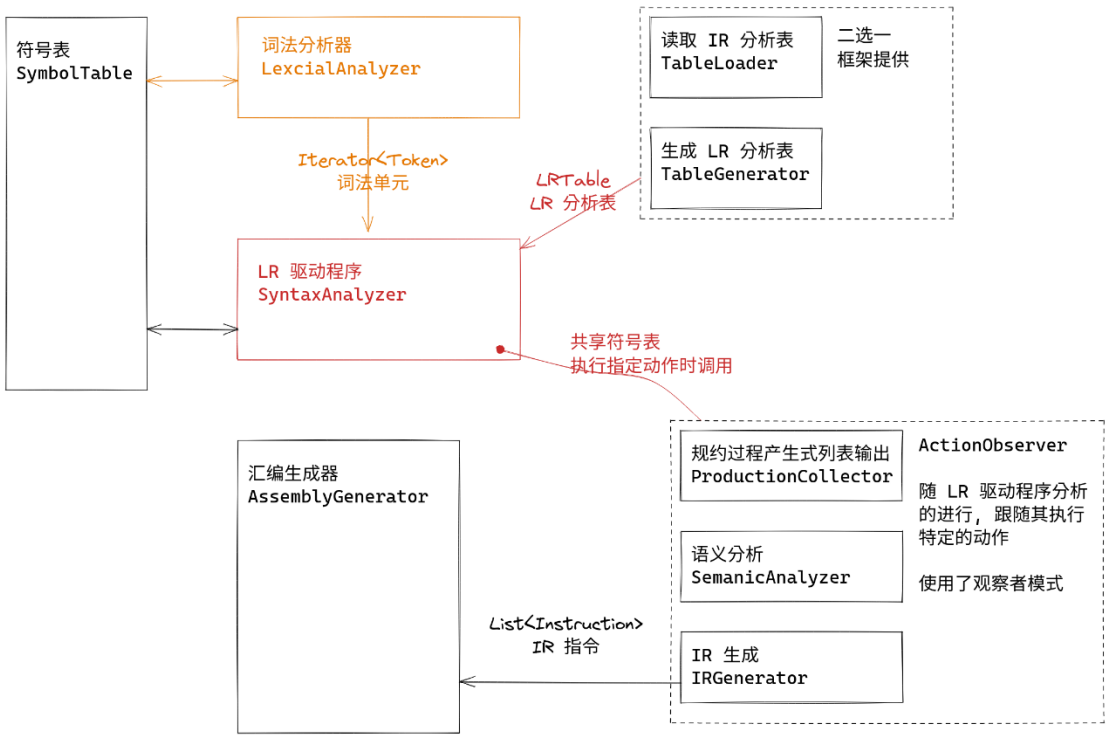
7. 输出结果

```
productionCollector.dumpToFile(FilePathConfig.PARSER_PATH);
```

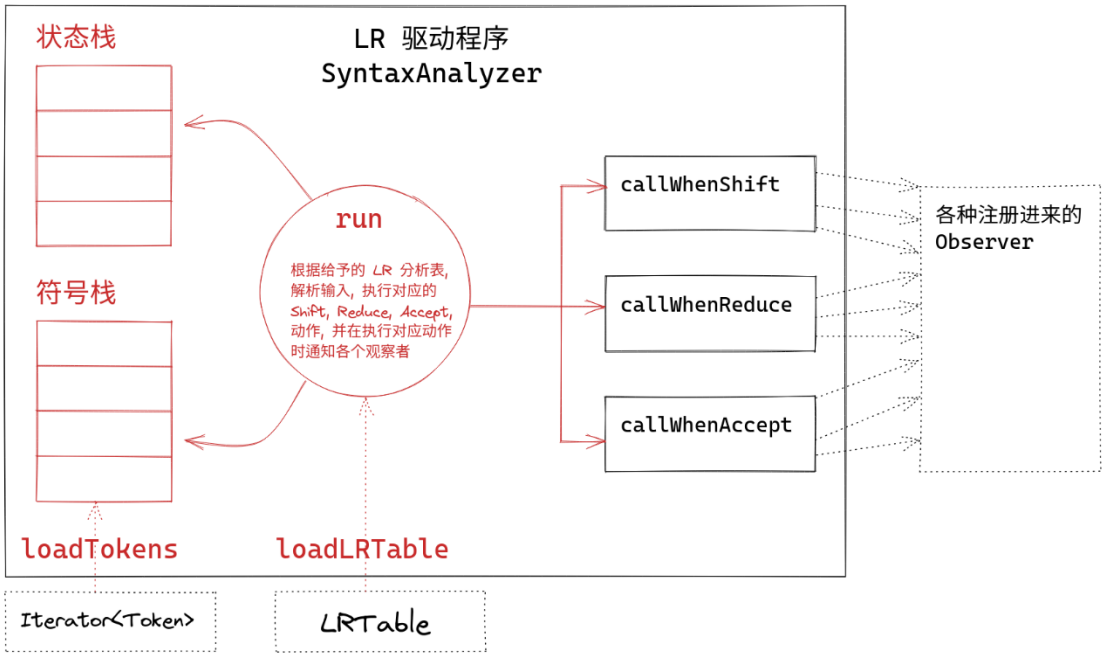
语法分析中使用的产生式列表输出到文件中 parser_list.txt 文件中。

语法分析主程序流程

2.5.2 LR(1)驱动程序框架图



在实验一中完成了词法分析器，词法分析器的输出 token.txt 作为实验二语法分析程序的输入。在本次实验中，你将要实现一个通用的 LR 语法分析驱动程序。它可以读入词法单元类别，任意的语法以及与之匹配的任意的 LR 分析表，随后读入词法单元流，依次根据分析表与语法执行移入、规约、接受、报错动作。



此驱动程序本身不会进行任何输出，它仅仅是作为一个被观察者，在动作发

生时通过调用观察者的方法以传递信息的方式通知各个观察者。为了确定你的代码正确，框架预先提供了一个在每次规约的时候记录规约到的产生式的观察者。若你正确实现了本实验，观察者将会按规约顺序输出所有记录到的产生式。

2.5.3 编写 LR(1)驱动程序

语法分析过程描述：

1．建立符号栈和状态栈，初始化栈；

2．根据状态栈栈顶元素和待读入的下一个 token 查询判断下一个待执行动作；

3．如果是 Shift，把 Action 的状态压入状态栈，对应的 token 压入符号栈；

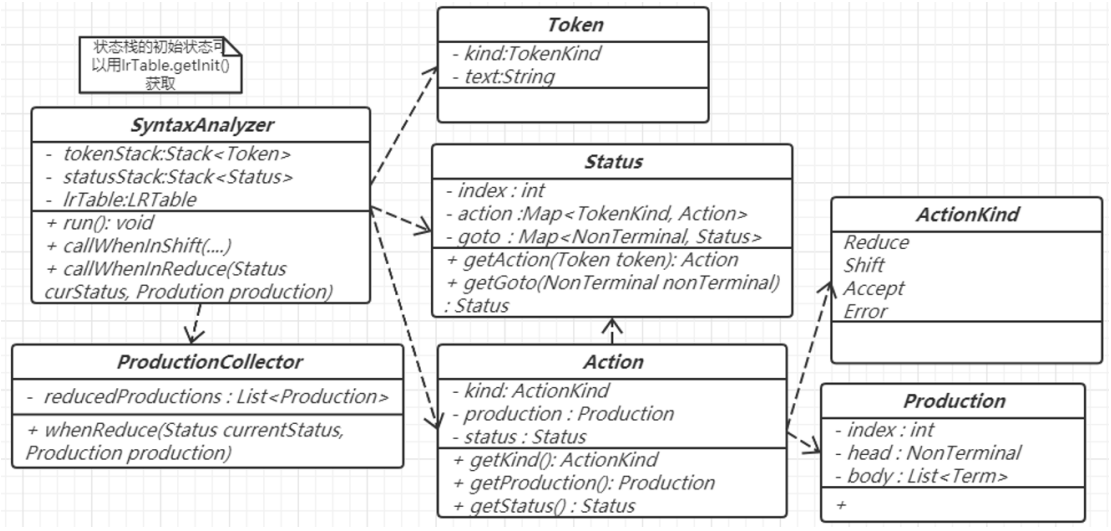
4．如果是 Reduce，根据产生式长度，符号栈和状态栈均弹出对应长度个符号和状态。

5. 把产生式左侧的非终结符压入符号栈；根据符号栈和状态栈栈顶状态获取 Goto 表的状态，压入状态栈，保持符号栈和状态栈栈顶高度一致；

6. 通知各个观察者；

7．如果是 Accept，语法分析执行结束；

8．ProductionCollector 观察者内部顺序记录归约所用到的产生式，语法分析结束输出到文件。



语法分析相关类类图

相关类说明：

1. ActionObserver

这是一个接口，接口里定义了 4 个接口方法；在实验二中 ProductionCollector 实现了这个接口，重写了这 4 个接口方法。

```
public interface ActionObserver {
    /**
     * 当驱动程序执行 Shift 动作时会调用此函数. Shift 会转移到的状态可以直接从参数中获取:
     * {@code currentStatus.getAction(currentToken).getStatus() }
     *
     * @param currentStatus 当前的状态
     * @param currentToken  当前的词法单元
     */
    void whenShift(Status currentStatus, Token currentToken);

    /**
     * 当驱动程序执行 Reduce 动作时会调用此函数. Goto 到的新状态可以直接从参数中获取
     * {@code currentStatus.getGoto(production.head()) }
     *
     * @param currentStatus 当前状态
     * @param production    待规约的产生式
     */
    void whenReduce(Status currentStatus, Production production);

    /**
     * 当驱动程序执行 Accept 动作时会调用此函数.
     *
     * @param currentStatus 当前状态
     */
    void whenAccept(Status currentStatus);

    /**
     * 当驱动程序接受符号表时会调用此函数, 实现此接口的类可以自行决定是否存储这个符号表
     *
     * @param table 符号表
     */
    void setSymbolTable(SymbolTable table);
}
```

2. ProductionCollector

```
public class ProductionCollector implements ActionObserver {

    private final Production beginProduction;
    private final List<Production> reducedProductions = new ArrayList<>();

    /**
     * 将结果输出到文件
     * @param path 文件路径
     */
    public void dumpToFile(String path) {
            FileUtils.writeLines(path,
            reducedProductions.stream().map(Production::toString).toList());
    }

    @Override
    public void whenReduce(Status currentStatus, Production production) {
        // 当规约时, 记录规约到的产生式
        reducedProductions.add(production);
    }

    @Override
    public void whenShift(Status currentStatus, Token currentToken) {
        // do nothing
    }

    @Override
    public void whenAccept(Status currentStatus) {
        // 当接受时, 记录下对起始产生式的规约
        reducedProductions.add(beginProduction);
    }

    @Override
    public void setSymbolTable(SymbolTable table) {
        // do nothing
    }
}
```

ProductionCollector 类实现了 ActionObserver 接口，重写这个接口方法。

在语法分析类 SyntaxAnalyzer 中发生归约动作时通过观察者对象调用 whenReduce(Status currentStatus, Production production)方法，传递当前归约所用产生式到 ProductionCollector 类中。

- 这个类在调用归约时自动收集归约的产生式列表，这个类已经完成，此实验

中无需修改。

- 该类将自己注册为 LR 驱动程序的动作观察者，在每次 reduce 执行时将归约的产生式保存起来，待到语法分析结束之后便能按归约顺序输出所有归约用到产生式。
- 该类的输出结果会被作为判断实验二代码正误的根据。
- 在语法分析实验中重点用到 whenReduce(Status currentStatus，Production production)方法，这个方法在归约时将归约的产生式加入到 reducedProductions 这个 List 数组中。

3. SyntaxAnalyzer

LR 语法分析驱动程序类。

```
private final SymbolTable symbolTable;
private final List<ActionObserver> observers = new ArrayList<>(); //观察者对象
…… //待扩展变量
```

```
    /**
     * 注册新的观察者
     * @param observer 观察者
     */
public void registerObserver(ActionObserver observer) {
        observers.add(observer);
        observer.setSymbolTable(symbolTable);
}
```

说明：注册观察者接口函数在 main 函数中调用，注册观察者，比如实验二生成归约列表的 listener，实验三语义检查的 listener 和 IR 生成的 listener。

```
/**
 * 在执行 shift 动作时通知各个观察者
 * @param currentStatus 当前状态
 * @param currentToken 当前词法单元
 */
public void callWhenInShift(Status currentStatus, Token currentToken) {
    for (final var listener : observers) {
        listener.whenShift(currentStatus, currentToken);
    }
}

/**
 * 在执行 reduce 动作时通知各个观察者
 * @param currentStatus 当前状态
 * @param production 待规约的产生式
 */
public void callWhenInReduce(Status currentStatus, Production production) {
    for (final var listener : observers) {
        listener.whenReduce(currentStatus, production);
    }
}

/**
 * 在执行 accept 动作时通知各个观察者
 * @param currentStatus 当前状态
 */
public void callWhenInAccept(Status currentStatus) {
    for (final var listener : observers) {
        listener.whenAccept(currentStatus);
    }
}
```

```
public void run() {
    // TODO: 实现驱动程序
    // 您需要根据上面的输入来实现 LR 语法分析的驱动程序
    // 请分别在遇到 Shift, Reduce, Accept 的时候调用上面的 callWhenInShift, callWhenInReduce, callWhenInAccept
    // 否则用于为实验二打分的产生式输出可能不会正常工作
    throw new NotImplementedException();
}
```

- 在语法分析中，依次读入实验一输出的 token，根据当前状态栈栈顶状态和待读入的下一个 token 查询 LR(1)分析表判断下一个待执行动作；

- 查到的状态是“Shift”，需要执行一次入栈操作，把 Action 中的状态压入状态栈，对应的 token 也压入符号栈；
- 查到的状态是“reduce”，根据产生式长度，符号栈和状态栈均弹出对应长度个非终结符或终结符和对应长度个状态；
- 再次根据符号栈和状态栈栈顶查询 LR(1)分析表的 GOTO 表，查询到的状态压入状态栈；
- 通知各观察者；
- 查询状态是 Accept，语法分析执行结束；
- ProductionCollector 观察者内部顺序记录归约所用到的产生式，语法分析结束后输出到文件。

2.5.4 输入输出说明

1. 本实验中的程序需要读入的文件：

tree data/in --sort=name

data/in

├── coding_map.csv # 码点文件（实验一的输入）

├── grammar.txt # 语法文件（实验二新增输入）

├── input_code.txt # 输入代码（实验一的输入）

└── LR1_table.csv # (可选) 第三方工具生成的 IR 分析表（实验二新增输入）

Grammar.txt 输入文件（产生式列表）：

```
P -> S_list;
S_list -> S Semicolon S_list;
S_list -> S Semicolon;
S -> D id;
D -> int;
S -> id = E;
S -> return E;
E -> E + A;
E -> E - A;
E -> A;
A -> A * B;
A -> B;
B -> ( E );
B -> id;
B -> IntConst;
```

LR1_table.csv 文件说明：

这是编译工作台生成的 LR（1）分析表，如果你在实验二中更换了 grammar.txt 的文法文件，那么需要根据新的产生式列表生成新的 LR(1)分析表文件，并替换掉模板中的 LR1_table.csv。

如果你不更新文法文件（grammar.txt）也就不需要更新 LR1_table.csv。

2.生成下面的文件：

```
tree data/out --sort=name
data/out
├── parser_list.txt              # 规约过程的产生式列表（实验二输出）
├── old_symbol_table.txt      # 语义分析前的符号表 （实验一输出）
└── token.txt                        # 词法单元流 （实验一输出）
```

parser_lsit.txt 文件存放的是语法分析中每次执行归约动作时的产生式列表。

parser_list.txt：

D -> int	*A -> B*
S -> D id	*E -> A*
D -> int	*B -> id*
S -> D id	*A -> B*
D -> int	*E -> E + A*
S -> D id	*B -> (E)*
D -> int	*A -> B*
S -> D id	*B -> id*
B -> IntConst	*A -> B*
A -> B	*E -> A*
E -> A	*B -> id*
S -> id = E	*A -> B*
B -> IntConst	*E -> E - A*
A -> B	*B -> (E)*
E -> A	*A -> A * B*
S -> id = E	*E -> E - A*
B -> IntConst	*S -> id = E*
A -> B	*B -> id*
E -> A	*A -> B*
B -> id	*E -> A*
A -> B	*S -> return E*
E -> E - A	*S_list -> S Semicolon*
S -> id = E	*S_list -> S Semicolon S_list*
B -> id	*S_list -> S Semicolon S_list*
A -> B	*S_list -> S Semicolon S_list*
B -> id	*S_list -> S Semicolon S_list*
*A -> A * B*	*S_list -> S Semicolon S_list*
E -> A	*S_list -> S Semicolon S_list*
B -> IntConst	*S_list -> S Semicolon S_list*
	S_list -> S Semicolon S_list
	P -> S_list

2.5.5 补充说明

在处理符号栈时，我们可能希望将 Token 和 NonTerminal 同时装在栈中。但 Token 和 NonTerminal 并没有共同祖先类(除了 Object)。我们当然可以使用 Stack<Object>，但其在使用中有诸多不便。于是我们期望有一个类似 Union<Token, NonTerminal> 的结构，就可以将栈定义为：Stack<Union<Token, NonTerminal>>。因为我们只将 Union 在这里使用一次，我们可以简单定义一个 Symbol 来实现 Union<Token, NonTerminal> 的功能。

下面给出一个 Symbol 的简单实现示例：

```
class Symbol{
    Token token;
    NonTerminal nonTerminal;

    private Symbol(Token token, NonTerminal nonTerminal){
        this.token = token;
        this.nonTerminal = nonTerminal;
    }

    public Symbol(Token token){
        this(token, null);
    }

    public Symbol(NonTerminal nonTerminal){
        this(null, nonTerminal);
    }

    public isToken(){
        return this.token != null;
    }

    public isNonterminal(){
        return this.nonTerminal != null;
    }
}
```

说明：由于实验属于设计型任务，同学在完成实验中可以自己设计，实验指导书仅供参考。