|  |
| --- |
| 哈尔滨工业大学(深圳) |
| **《编译原理》实验报告** |
|  |
| 学 院: 计算机科学与技术   |  |  | | --- | --- | | 姓 名: | 孙铎 | | 学 号: | 200110503 | | 专 业: | 计算机科学与技术 | | 日 期: | 2022-xx-xx | |

1. **实验目的与方法**

**实验目的为本次实验的实验目的，方法为所使用的语言，软件环境等。**

* 1. **词法分析器**
     1. **实验目的：**
* 加深对词法分析程序的功能及实现方法的理解。
* 对类C语言单词符号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。
* 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。
  + 1. **实验方法：**
* 语言：Java语言
* 软件环境：JDK17
* IDE：IntelliJ IDEA
* 框架：哈工大(深圳)2022春编译原理实验代码框架
  1. **语法分析**
     1. **实验目的：**
* 深入了解语法分析程序实现原理及方法。
* 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。
* 加深对LR分析表的认识，理解其在LR(1)分析中的重要作用。
* 通过实现具体的语法分析程序，加深对移入、归约等重要概念的理解。
  + 1. **实验方法：**
* 语言：Java语言
* 软件环境：JDK17
* IDE：IntelliJ IDEA
* 框架：哈工大(深圳)2022春编译原理实验代码框架
  1. **典型语句的语义分析及中间代码生成**
     1. **实验目的：**
* 巩固对语义分析的基本功能和原理的认识。
* 加深对自底下上语法制导翻译技术的理解掌握声明语句、赋值语句和算术运算语句的翻译方法。
* 理解中间代码的表示形式，掌握三地址码的实现方式。
* 理解SDD中语义属性的含义与作用。
* 理解SDT中语义动作的作用与执行方式。
* 理解符号表在语法制导翻译中的作用。
  + 1. **实验方法：**
* 语言：Java语言
* 软件环境：JDK17
* IDE：IntelliJ IDEA
* 框架：哈工大(深圳)2022春编译原理实验代码框架
  1. **目标代码生成**
     1. **实验目的**
* xxx
  + 1. **实验方法**
* 语言：Java语言
* 软件环境：JDK17
* IDE：IntelliJ IDEA
* 框架：哈工大(深圳)2022春编译原理实验代码框架

1. **实验内容及要求**

**每次实验室的实验内容和要求描述清楚。**

* 1. **词法分析器**
* 确定词法规则对应的DFA，确定自动机状态, 转移方式, 以及在特定转移步骤时要采取的动作
* 编写一个词法分析程序，读取代码文件，对文件内的类C语言程序段进行词法分析。
* 处理类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。
* 完善符号表SymbolTalbe类的方法，使其能在本实验框架下的编译过程中发挥应有的作用。
* 词法分析程序输入：以文件形式存放的类C语言程序段。
* 词法分析程序输出：以文件形式存放的Token串和简单符号表。
  1. **语法分析**
* 利用LR(1)分析法，设计语法分析程序，结合文法对输入单词符号串（实验一的输出）进行语法分析。
* 设计LR(1)分析法需要的符号栈和状态栈数据结构。
* 合理运用框架提供的LR分析表，正确匹配ACTION和GOTO。
* 正确解析ACTION类型，根据LR(1)的规则执行对应的栈操作，并通知观察者。
* 输出推导过程中所用产生式序列并保存在输出文件中。
  1. **典型语句的语义分析及中间代码生成**
* 采用实验二中的文法，为语法正确的单词串设计翻译方案，完成语法制导翻译。
* 利用该翻译方案，对所给程序段进行分析，输出生成的中间代码序列和更新后的符号表，并保存在相应文件中中间代码使用三地址码 的四元式表示。
* 注：更新后的符号表只需要保存id和type两个属性。
* 实现声明语句、简单赋值语句和算术表达式的语义分析与中间代码生成。
* 使用框架中的模拟器 IREmulator 验证生成的中间代码的正确性。
  1. **目标代码生成**
* **xxx**
* **xxx**

1. **实验总体流程与函数功能描述**
   1. **词法分析**
      1. **编码表**

|  |  |  |
| --- | --- | --- |
| 单词名称 | 类别编码 | 单词值 |
| int | 1 | - |
| return | 2 | - |
| = | 3 | - |
| , | 4 | - |
| Semiclon | 5 | - |
| + | 6 | - |
| - | 7 |  |
| \* | 8 | - |
| / | 9 | - |
| ( | 10 | - |
| ) | 11 | - |
| id | 51 | - |
| IntConst | 52 | - |

* + 1. **正则文法**

**正则定义：**

* 字母：letter -> A|B|...|Z|a|b|...|z
* 数字：

digit -> 0|1|2|...|9

no\_0\_digit -> 1|2|...|9

**正则文法：**

* 标识符：

S -> letter A

A -> letter A|digit A|**ε**

注：关键字先按照标识符识别，再判断是否在关键字集合中

* 整常数：

S -> no\_0\_digit B

B -> digit B|**ε**

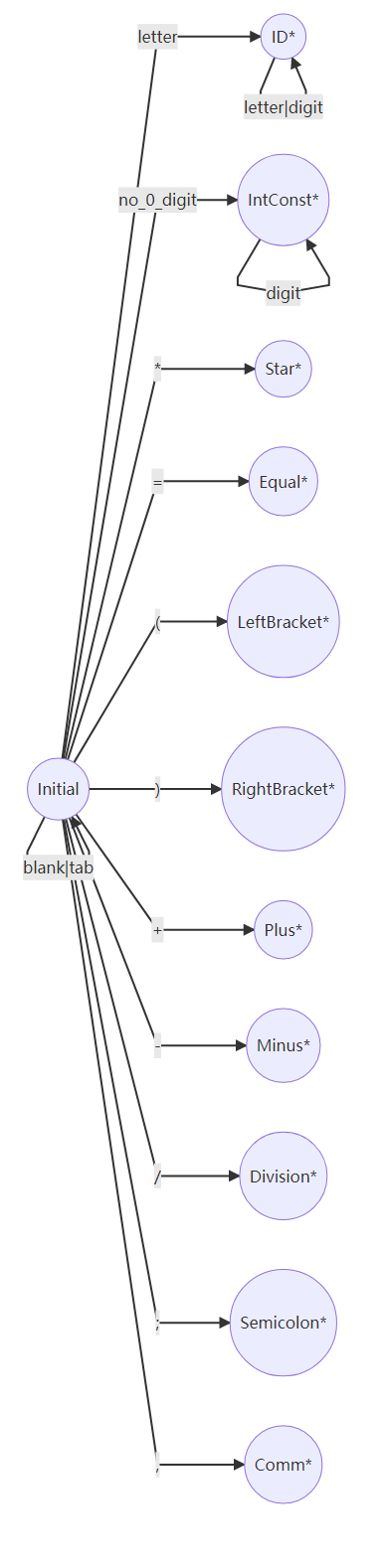
* 运算符、分隔符：

S -> C

C -> = | \* | + | - | / | ( | ) | , | ;

注：空格、换行符、制表符属于需要过滤掉的符号，不在文法中体现

* + 1. **状态转换图**



注：

* 标记了\*的状态代表可接收状态，该自动机只有初始状态为不可接收状态，其他状态均为可接收状态。
* 在可接收状态停机说明读取到了完整的单词，在不可接收状态停机说明读取到了非法符号。
* 自动机从开始状态到停机的状态转移过程中，会读取字符并记录字符，供停机后处理使用。
* 当自动机处于某状态接收到某符号，致使自动机没有可跳转到的状态时，自动机会停机并分析停机前接收到的所有字符。停机之前接收到的最后一个字符会被“退回”，重回开始状态后，继续读取该符号。（如果是非法符号则直接停机并抛出异常）。
* 换行符已经在读取源文件时自动去除，故这里只体现空格和制表符。
  + 1. **主要函数流程**

注：下述函数流程主要按照main函数的执行顺序来说明，且主要说明了框架中TODO的部分，省略其他已经实现好部分。

* LexicalAnalyzer(SymbolTable)

这是LexicalAnalyzer的构造函数。

先将传入的符号表保存到成员变量symbolTalbe中，再初始化下列成员变量：

* codeWords：用于记录读入的源程序文本，初始化为空的StringBuilder。
* keyWords：关键字集合，只包含int和return。
* tokens：用于记录解析源程序时生成的token，初始化为空的ArrayList。
* LexicalAnalyzer.loadFile(String)

通过参数path，创建字符缓冲流BufferedReader读取源代码文本，逐行读取并拼接到字符缓冲区codeWords中，拼接时忽略换行符（默认源代码每行以分号结尾，不考虑不以分号结尾的情况），并在读取完毕后再拼接一个$符号，代表源代码文本结束，方便后续分析。

* LexicalAnalyzer.run()

run方法代表了词法分析的主要过程。

首先定义好一个枚举类，代表自动机的各个状态。

run方法主要执行流程如下：

1. 初始化当前状态为初始状态，初始化一个StringBuilder用于记录每次读入的字符。
2. for循环遍历每个字符，使用switch-case结构，根据当前状态和遍历到的字符，做出相应动作，跳转到对应的状态。

* 在非初始状态下，如果遍历到的字符在自动机中没有对应的下一个状态，则将停机标志置为true，并分析StringBuilder中已有的字符（不包括本次遍历到的字符），生成token，加入到成员变量tokens中，如果是标识符还需要同时加入符号表symbolTable中，然后跳转到初始状态；如果有下一个状态，则不需要更改停机标志，直接跳转到对应状态即可。
* 在初始状态下，在非初始状态下，如果遍历到的字符在自动机中没有对应的下一个状态，则抛出异常（因为初始状态是不可接收状态，不能在此停机）；如果有下一个状态，则不需要更改停机标志，直接跳转到对应状态即可。

1. 每完成一次遍历字符和状态跳转后，需要在for循环内判断停机标志，若为false，则将遍历到的字符放入StringBuilder中用于之后的分析，并继续遍历；若为true，则判断当前遍历到的字符是否为结束符$，若是结束符，则将Token.eof()加入成员变量tokens中，若不是结束符，则清空StringBuilder，回退遍历索引，让停机前遍历到的最后一个字符可供下次使用。

* LexicalAnalyzer.getTokens()

由于在run方法中已经在识别单词后生成了token并加入到成员变量tokens中，该方法不需要特殊处理，直接返回成员变量tokens即可。

* SymbolTable.entries

这是List<SymbolTableEntry>类型的成员变量，用于表示符号表中的记录。

* Symboltable.has(String)

遍历entries成员的每条记录，判断是否有某个entry的text字段与传入的参数相同，若有，则返回true，若没有，则返回false。

* SymbolTable.add(String)

先通过has方法判断符号表中是否已经存在名为参数text的记录，若有则抛出异常，若没有，则根据参数text构造一个entry并加入成员变量entries中。

* SymbolTable.get(String)

先通过has方法判断符号表中是否存在名为参数text的记录，若没有则抛出异常，若有，则遍历成员变量entries，寻找到text字段与参数相等的entry并返回。

* SymbolTable.getAllEntries()

遍历成员变量entries，同时生成一个以text字段为key，以整条entry为value的哈希表，并返回。

* 1. **语法分析**
     1. **拓展文法**

通过加入 P -> Slist，让P成为开始符号，且只有该产生式含有开始符号P，这样在归约过程中能分清是否已归约到文法的最初开始符。

1: P -> Slist

2: S\_list -> S Semicolon S\_list

3: S\_list -> S semicolon

4: S -> D id

5: D -> int

6: S -> id = E

7: S -> return E

8: E1 -> E2 + A

9: E1 -> E2 – A

10: E -> A

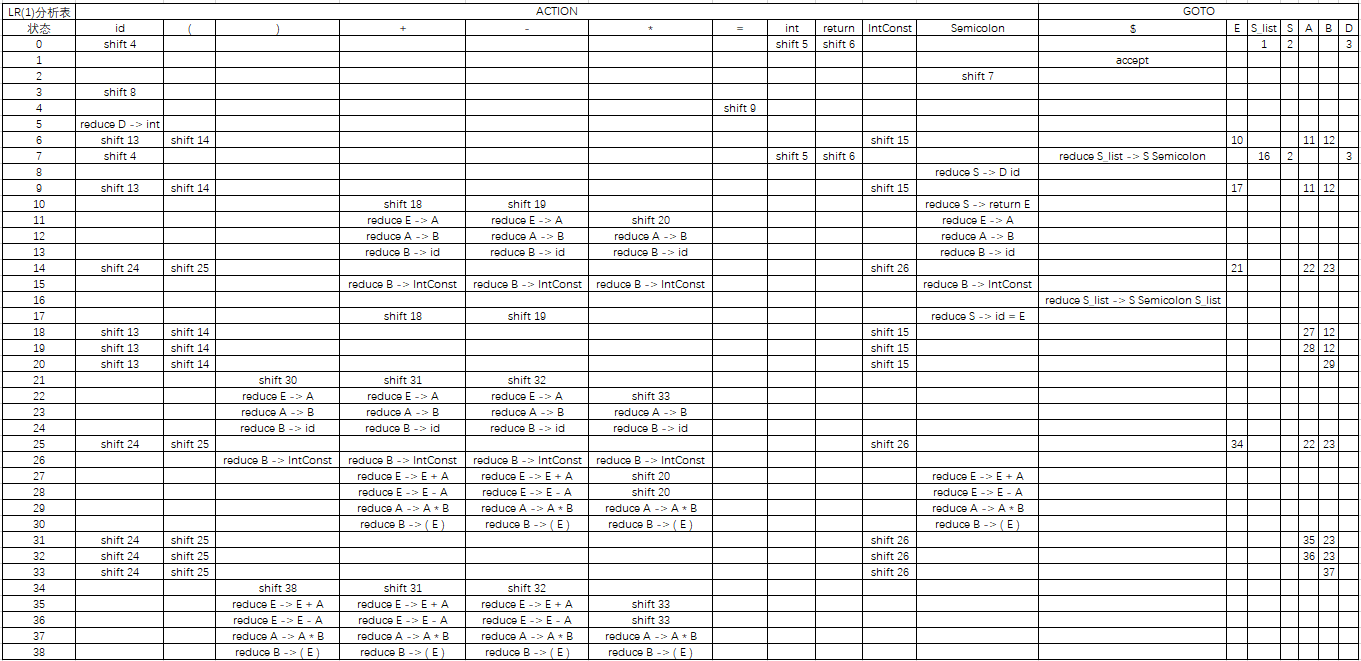
11: A1 -> A2 \* B

12: A -> B

13: B -> ( E )

14: B -> id

15: B -> IntConst

* + 1. **LR1分析表**
    2. **状态栈和符号栈的数据结构**
* 状态栈：Stack<Status> statusStack

Status即为框架提供的类，Stack是JDK中的栈数据结构，通过push和pop方法即可完成入栈和出栈。

* 符号栈：Stack<Symbol> symbolStack

Symbol是自定义类，可以代表非终结符或终结符（token），包含两个成员：

* Token token
* Nonterminal nonTerminal

Token和Nonterminal均为框架提供的类。

Symbol类中提供了判断文法符号是非终结符还是终结符和获取文法符号的方法。

* + 1. **LR驱动程序流程描述**

注：下述函数流程主要按照main函数的执行顺序来说明，且主要说明了框架中TODO的部分，省略其他已经实现好部分。

1. 创建语法分析类SyntaxAnalyzer创建对象parse，传入符号表作为其成员变量。parse是负责完成LR驱动程序的主要对象。
2. 调用parse.loadTokens方法，传入tokens（词法分析生成的token序列），加载词法单元。在SyntaxAnalyzer中提前设置了Deque<Token> tokenDeque成员，用于存放token序列，使用JDK提供的双端队列Deque，有peek方法只查看队列头的token而不消耗，符合本场景的需求。该方法中，遍历传入的token序列，将token依次入队，并同时向符号栈symbolStack中压入文法结束符Token.eof()，代表符号栈已经准备好LR驱动。
3. 调用parse.loadLRTable方法，传入借助第三方程序构造的LRTable，保存该table于SyntaxAnalyzer的成员变量lrTable中，同时并同时向状态栈statusStack中压入初始状态lrTable.getInit()，代表状态栈已经准备好LR驱动。
4. 给parser注册生成规约列表、语义检查、生成中间表示形式的三个观察者。
5. 调用parser.run()方法正式开始LR驱动语法分析：
   1. 通过状态栈statusStack获取当前状态，通过tokenDeque以栈的方式查看栈顶的输入符号，根据当前状态和栈顶输入符号，借助lrTable，得到应当执行的动作action。
   2. 根据action的类型，执行不同的程序段

* 若为Shift类型，代表执行移入动作，调用callWhenInShift方法通知观察者，向状态栈中压入移入动作后的状态action.getStatus()，同时将一个输入token出队，代表消耗一个输入符号，并将出队的token压入符号栈。
* 若为Reduce类型，代表执行归约动作，调用callWhenInReduce方法通知观察者，获取用于归约的产生式，先从状态栈和符号栈中同步弹出与产生式右部符号数相等个数的元素，然后将产生式左部非终结符压入符号栈，再根据当前状态栈栈顶的状态和符号栈栈顶的符号（即产生式左部非终结符），借助lrTable，执行相应的GOTO动作，得到相应状态，并将该状态加入状态栈中。
* 若为Accept类型，代表LR(1)分析成功结束，调用callWhenInAccept方法通知观察者。
* 若为Error类型，则分析出错，抛出异常。
  1. **语义分析和中间代码生成**
     1. **翻译方案**

1: P -> Slist

2: S\_list -> S Semicolon S\_list

3: S\_list -> S semicolon

4: S -> D id {p = lookup(id.name); if p == nil then error; enter(id.name, D.type);}

5: D -> int {D.type = int;}

6: S -> id = E {p = lookup(id.name); if p == nil then error; gencode(id.val = E.val);}

7: S -> return E {gencode(return E.val);}

8: E1 -> E2 + A {E1.val = newtemp(); gencode(E1.val = E2.val + A.val);}

9: E1 -> E2 – A {E1.val = newtemp(); gencode(E1.val = E2.val - A.val);}

10: E -> A {E.val = A.val;}

11: A1 -> A2 \* B {A1.val = newtemp(); gencode(A1.val = A2.val \* B.val);}

12: A -> B {A.val = B.val;}

13: B -> ( E ) {B.val = E.val;}

14: B -> id {B.val = lookup(id.name); if B.val == nil then error;}

15: B -> IntConst {B.val = IntConst.lexval;}

* + 1. **语义分析和中间代码生成使用的数据结构**
* 记录类型信息（type）和标识符名（lexval）的综合属性栈：

SemanticAnalyzer.java: Deque<LexVal> synStk;

这里使用LexVal作为泛型，LexVal是新建的内部类，有SourceCodeType和String两个类型的成员变量，这样做是因为，在计算完标识符的type综合属性后，要更新符号表，这就需要标识符名来寻找到符号表中相应的entry，所以将标识符名也作为一个综合属性记录于栈中，故使用同时封装了两种类型的内部类，并提供相应的getter和类型检查，满足该场景的需求。

* 记录值信息（val）的综合属性栈：

IRGenerator.java: Deque<IRValue> synStk;

直接存放IRImmemiate实例或IRVariable实例，这样规约时就可以直接出栈，加以利用，生成三地址码。

* + 1. **主要流程描述（两个观察者的实现）**
* SemanticAnalyzer:

- setSymbolTable:

在该类中提前设置SymbolTable类型成员，在该方法中，直接将参数table赋值给该成员即可。

- whenShift:

从参数中得到此移入动作对应的token，进而获取token类型，如果是关键字类型，则需要将其类型信息（type）压入综合属性栈；如果是标识符类型，则需要根据标识符名字（name）在符号表中查询是否存在该标识符，若不存在则抛出异常，若存在则将名字信息（name）压入综合属性栈；如果是其他类型，则向综合属性栈中压入空记录占位（空记录也是LexVal类型，两个成员变量均为null）。

- whenReduce：

使用switch-case判断归约时使用的产生式，SemanticAnalyzer只需要关注涉及到标识符类型和符号表的语义动作。如果是 S -> D id 则弹栈并从栈中获取标识符id的名字和非终结符D的类型，然后更新符号表，再向栈中压入空记录；如果是 D -> int ，理论上需要弹栈获取类型信息，再将类型信息入栈，总得来说属性栈不发生变化，所以这种情况可以不执行代码；如果其他产生式，在本类中不需要关注其语义动作，直接根据产生式右部和左部的符号数量进行对应次数的出栈和入栈，入栈时使用空记录即可。

- whenAccept：

执行该方法代表已经使用开始符号对应的产生式归约了，LR(1)分析成功结束，而开始符号对应的产生式在本场景下不需要语义动作，分析结束了也不必再更新属性栈，所以可以不执行代码。

* IRGenerator:

- setSymbolTable:

在该类中提前设置SymbolTable类型成员，在该方法中，直接将参数table赋值给该成员即可。

- whenShift:

从参数中得到此移入动作对应的token，进而获取token类型，如果是整数常量类型，则根据其值信息创建一个IRImmediate实例，并压入综合属性栈；如果是标识符类型，则根据标识符名字在符号表中查询是否存在该标识符，若不存在则抛出异常，若存在则根据名字创建一个IRVariable实例，并压入综合属性栈；如果是其他类型，则向综合属性栈中压入空记录占位（空记录是提前写好的一种类型IRNullValue，实现了IRValue）。

- whenReduce

使用switch-case判断归约时使用的产生式，IRGenerator只需要关注涉及到变量值和生成三地址码的语义动作。

如果是 S -> id = E，则弹栈并获得两个IRValue信息，据此生成Mov三地址指令，再压入空记录；

如果是 S -> return E，则弹栈并获取E对应的IRValue，据此生成Ret三地址指令，再压入空记录；

如果是E1 -> E2 + A、E1 -> E2 – A、A1 -> A2 \* B，弹栈获得两个IRValue信息，据此生成对应运算符的Add、Sub、Mul三地址指令，再创建代表临时变量的IRVariable，这是左部符号的综合属性，直接入栈即可。

如果是 E -> A、A -> B、B -> id、B -> IntConst，理论上需要将右部符号的IRValue赋值给左部符号，赋值前后综合属性栈不发生变化，所以可以不执行代码。

如果是 B -> (E)，则三次分别弹出(、E、)对应的记录，在第二次弹出E时，保存下IRValue信息，最后再创建代表临时变量的IRVariable，这是左部符号的综合属性，直接入栈即可。

如果其他产生式，在本类中不需要关注其语义动作，直接根据产生式右部和左部的符号数量进行对应次数的出栈和入栈，入栈时使用空记录即可。

- whenAccept

执行该方法代表已经使用开始符号对应的产生式归约了，LR(1)分析成功结束，而开始符号对应的产生式在本场景下不需要语义动作，分析结束了也不必再更新属性栈，所以不需要执行代码。

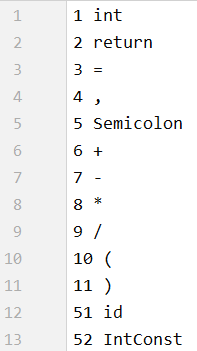
* 1. **目标代码生成**
     1. **主要流程描述**

1. **实验结果与分析**

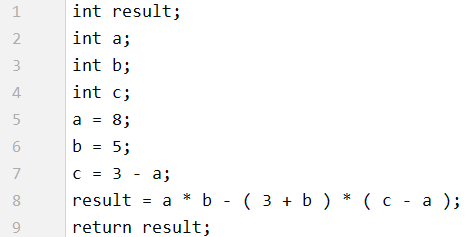
**对实验的输入输出结果进行展示与分析。注意：要求给出编译器各阶段（词法分析、语法分析、中间代码生成、目标代码生成）的输入输出并进行分析说明。**

* 1. **词法分析**
* 输入：

- 码点文件：coding\_map.csv



- 输入的代码：input\_code.txt



* 输出：

- 符号表：old\_symbol\_table.txt

表格

描述已自动生成

- 词法单元列表：token.txt

图片包含 文本

描述已自动生成文本

描述已自动生成表格

中度可信度描述已自动生成

* 分析说明：

输入的码点文件coding\_map.csv对应了编码表，也就是token中的种别码，源代码文件input\_code.csv就是要解析并生成token的字符序列。

输出文件old\_symbol\_table.txt是符号表，此时的符号表中只登记了标识符名，并不包含类型信息，可以看出该符号表中有四条记录，已经包含了源代码中声明的四个变量。token.txt就是解析源代码后生成的token序列，用一个二元组（一对括号）代表一个token，每个token都正确标识了单词的种别与属性值（没有属性值则留空），与源代码一一对应。

* 1. **语法分析**
* 输入：

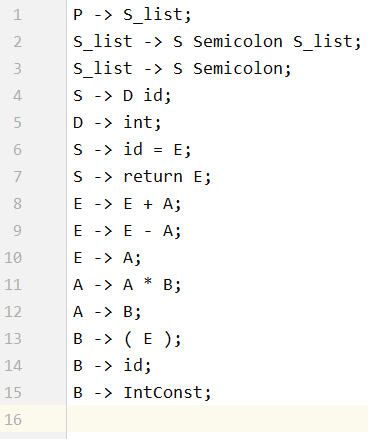
- 码点文件：coding\_map.csv

与词法分析中的一致

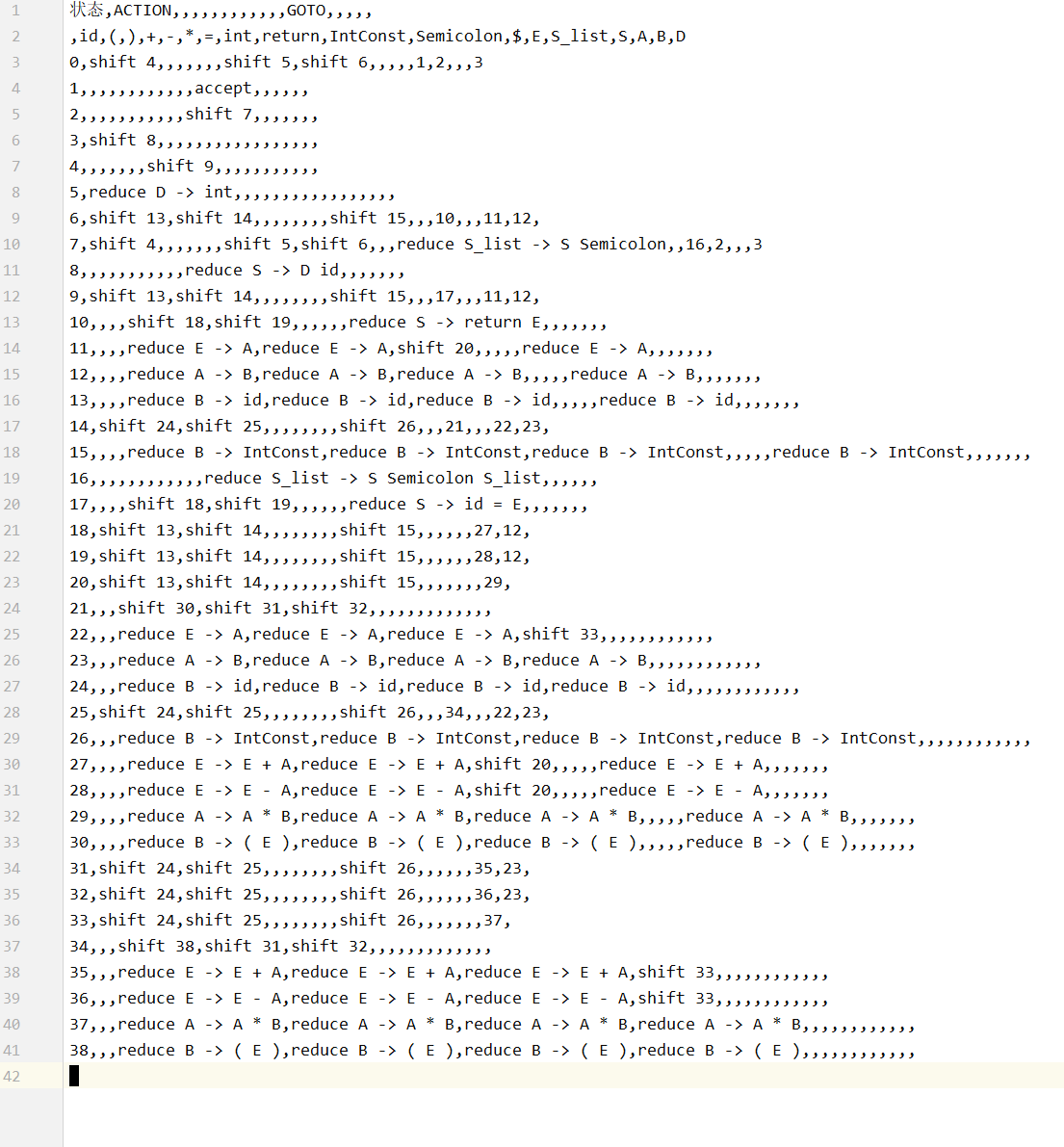
- 输入的代码：input\_code.txt

与词法分析中的一致

- 语法文件：grammar.txt



- 第三方工具生成的LR分析表：LR1\_table.csv



* 输出：

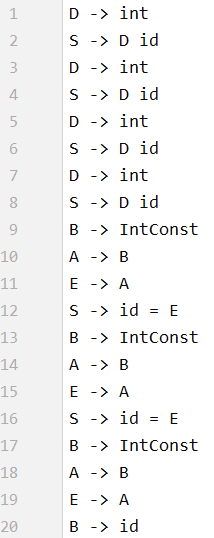
- 语义分析前的符号表：old\_symbol\_table.txt

与词法分析中的一致

- 词法单元列表：token.txt

与词法分析中的一致

- 规约过程的产生式列表：parser\_list.txt

 文本

中度可信度描述已自动生成 表格

中度可信度描述已自动生成

* 分析说明：

与词法分析中一致的部分不再赘述。

语法文件grammar.txt记录了一个LR文法的所有产生式，每行对应一条产生式，这是语法分析、语义分析、中间代码生成等编译过程的重要依据。LR分析表文件LR1\_table.csv由第三方工具根据语法文件grammar.txt生成，对应了LR分析中需要使用到的LR分析表，可以由程序读取并使用，是LR分析中执行移入、归约等动作的重要依据。

输出的规约过程产生式列表文件parser\_list.txt每行都是grammar.txt中的某一条产生式，依次记录了对token序列进行LR分析的过程中执行归约动作时用到的产生式，代表了对token序列的规约过程。

* 1. **中间代码生成**
* 输入：

- 码点文件：coding\_map.csv

与词法分析中的一致

- 输入的代码：input\_code.txt

与词法分析中的一致

- 语法文件：grammar.txt

与语法分析中的一致

- 第三方工具生成的LR分析表：LR1\_table.csv

与语法分析中的一致

* 输出：

- 语义分析前的符号表：old\_symbol\_table.txt

与词法分析中的一致

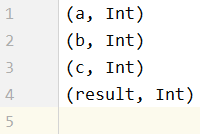
- 词法单元列表：token.txt

与词法分析中的一致

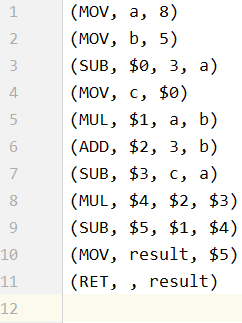
- 规约过程的产生式列表：parser\_list.txt

与语法分析中的一致

- 语义分析后的符号表：new\_symbol\_table.txt



- 中间表示：intermediate\_code.txt



- 中间表示的模拟执行的结果：ir\_emulate\_result.txt

图形用户界面, 应用程序

中度可信度描述已自动生成

* 分析说明：

与词法分析和语法分析中一致的部分不再赘述。

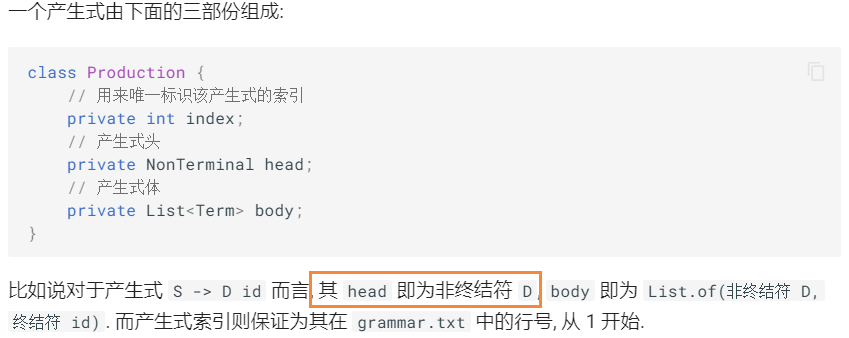
语义分析后的符号表相比于语义分析之前，补充了每条记录的类型信息。intermediate\_code.txt文件记录了中间表示形式，每一行代表一条三地址码指令，所有三地址指令依次执行即能反映源代码的逻辑，是input\_code.txt中源代码在编译过程中的中间表示形式。ir\_emulate\_result.txt文件中记录了一个数值，也就是依次执行intermediate\_code.txt中三地址指令后得到的返回值，该数值应当与运行input\_code.txt中源代码得到的返回值一致。

* 1. **目标代码生成**

1. **实验中遇到的困难与解决办法**

**描述实验中遇到的困难与解决办法，对实验的意见与建议或收获。**

* 1. **实验一**
     1. **遇到的困难与解决办法：**
* 在关键字的识别中，如果让每个关键字都在自动机中有所体现，则需要设计较多的状态以区分关键字与标识符，这样的设计十分繁琐，不利用自动机的实现。所以在设计自动机时，考虑到关键字与标识符的相似性，将关键字暂时与标识符统一起来识别，待识别结束后，再通过程序中设置的关键字集合对关键字和标识符加以区分，这样简化了自动机的实现，易于理解和编程。
* 编码表中有相对较多的运算符和分隔符，对应了自动机中较多的状态，在使用switch-case结构实现自动机时，如果每个运算符和标识符都对应一个case，则程序会显得十分冗长，且各运算符和标识符的处理逻辑类似，所以可以将它们都放到一个case语句块中，统一处理，方便编程。
* 在自动机识别出一个完整的单词后，需要能到达某个“状态”，分析已经读取的输入，形成一个token。考虑到这样的需求，在程序中引入了“停机”这个概念，当已经识别出一个完整的单词，再遇到其他类型单词的输入时，就会让自动机停机，并根据停机时结束在的状态识别单词，生成token。
  + 1. **收获与建议：**
* 主要的收获是完成了对自动机的具体实现，让自己对自动机的理解不只停留在抽象层面。
* 虽然在本框架中，符号表没有起到核心作用，但符号管理依然是编译中的重要模块。建议进一步突出符号表中的作用，至少要在指导书上给予更详细的介绍，说明需要同学们实现的部分。
  1. **实验二**
     1. **遇到的困难与解决办法：**
* 语法分析时，符号栈中需要同时存放终结符和非终结符，对于本实验，要同时存放Token和NonTerminal类型，这两种类型在本框架中并没有继承关系。所以新建了一个静态内部类Symbol，包含Token和NonTerminal两种类型的成员变量，可以用作Token和NonTerminal，分别提供对应的getter并进行类型检查，即可符合框架要求。
  + 1. **收获与建议：**
* 框架对于Action、Status、Production类的封装很值得学习，巧妙结合了面向对象的思想，将几者的关系联系了起来，在语法分析应用时十分方便。但同时，框架自带的方便设计也降低了实验的难度，可以考虑让同学们自己实验这些部分，在指导书上给予必要的指导，这样也可以加深同学们对于状态和动作等概念的理解。
* 本次实验的重点在于对LR分析表的使用，根据LR分析表进行语法分析的各种动作，但几乎不涉及到LR自动机的构造与分析表的形成（大部分同学使用的都是第三方工具，而非框架自带的TableGenerator类）。可以考虑要求同学们自己编写一部分TableGenerator类，加深同学们对FIRST集、FOLLOW集、LR自动机等概念的理解。
  1. **实验三**
     1. **遇到的困难与解决办法：**
* SemanticAnalyzer负责标识符类型信息的登记，在相应产生式归纳时，该观察者需要标识符名和标识符类型两个信息，才能完成对符号表的更新，但框架使用的观察者模式并没有在归约时将token传给观察者，这就需要观察者自己在移进时注意保存标识符名的信息。因而，综合属性栈中需要存放标识符类型（SourceCodeType）和标识符名（String）两种类型的属性，仿照实验二将Token和NonTerminal封装的思想，在这里也新建一个内部类LexVal，代表两种类型之一，提供对应的getter与类型检查，方便移进和归约时的统一使用。
  + 1. **收获与建议：**
* 框架使用观察者类负责语法动作的执行，且不同类型的语法动作也可由不同观察者执行，这样将语法动作执行解耦合给观察者的思想十分值得学习。
* 指导书中有一处可能误导同学们的错误（不知道现在是否已经改正）：



一条产生式的头部也称作这条产生式的左部，也就是箭头左边的符号，而不是指导书上写的右部符号串的第一个符号。

* 1. **实验四**
     1. **遇到的困难与解决办法：**
     2. **收获与建议：**