

# 编译原理课程实验报告

## 实验 2：语法分析

姓名	王少博	院系	计算机科学与技术学院	学号	181110315
任课教师	韩希先	指导教师	韩希先		
实验地点	研究院中 507	实验时间	2020.11.5		
实验课表现	出勤、表现得分		实验报告得分		实验总分
	操作结果得分				

### 一、实验目的

基本目的：

#### 1. 巩固对语法分析的基本功能和原理的认识。

自底向上的语法分析的基本功能，是从给定的输入符号串出发，试图自底向上地为其建立一棵语法分析树。自底向上的语法分析的原理是从输入串出发，反复利用产生式进行归约，如果最后能得到文法的开始符号，则输入串是相应文法的一个句子，否则输入串有语法错误。在分析过程中，寻找当前句型最左的和某个产生式的右部相匹配的子串，用该产生式的左部符号代替该子串。如果每步都能正确选择子串，就可以得到输入串的最左归约过程，即规范归约过程。

#### 2. 通过对语法分析表的自动生成加深语法分析表的认识。

通过对闭包的求解、*first* 集的求解、*follow* 集的求解、*action* 表的求解、*goto* 表的求解，从而自动生成语法分析表。首先求 *I* 的闭包  $CLOSURE(I)$ ，然后求 *I* 在遇到某个文法符号 *X* 之后的后继项目集  $GO(I, X)$ ，不断地循环求出  $G'$  的项目集规范族 *C*。接着只要求出  $FIRST(X)$  进而求出  $FOLLOW(A)$  以后，再加上项目集规范族 *C* 就能得到  $SLR(1)$  分析表了。

#### 3. 理解并处理语法分析中的异常和错误。

语法分析中的异常和错误主要是移进归约冲突和归约归约冲突，通过建立包含出错处理的优先矩阵来进行处理。LR 分析过程中，如果遇到异常和错误，则会中止分析，并且可以根据分析过程的输出来查看错误发生的位置、种类以及原因。

三次实验所有代码已部署于 github, <https://github.com/HITWH18SE/Compiler-Principles>

### 二、实验内容

#### (1) 文法描述

##### 函数定义部分：

- 程序 → 函数定义
- 函数定义 → 函数定义 函数定义
- 函数定义 → 变量类型 *id* () { 函数块 }
- 函数定义 → 变量类型 *id* ( 传入参数 ) { 函数块 }
- 传入参数 → 变量类型 *id*
- 传入参数 → 变量类型 *id*, 传入参数
- 函数块 → 函数块 函数块
- 函数块 → 变量类型 *id* ;

##### 赋值部分

- 函数块 → *id* = 算术表达式 ;

### 循环部分

- 函数块 → while ( 布尔表达式 ) { 函数块 }

### 分支部分

- 函数块 → if ( 布尔表达式 ) { 函数块 }
- 函数块 → if ( 布尔表达式 ) { 函数块 } else { 函数块 }

### 变量说明部分

- 变量类型 → int
- 变量类型 → float
- 算术表达式 → 算术表达式 算术运算符 算术表达式
- 算术表达式 → - 算术表达式
- 算术表达式 → ( 算术表达式 )
- 算术表达式 → id
- 算术表达式 → digit

### 布尔表达式部分

- 布尔表达式 → 算术表达式 比较运算符 算术表达式
- 布尔表达式 → 布尔表达式 && 布尔表达式
- 布尔表达式 → 布尔表达式 || 布尔表达式
- 布尔表达式 → ! 布尔表达式
- 布尔表达式 → ( 布尔表达式 )
- 布尔表达式 → true | false

### 其他部分

- 比较运算符 → < | > | <= | >= | == | !=
- 算术运算符 → + | - | \* | /

变量说明、赋值、表达式、分支、循环都在文法中有所体现，产生式左部为了方便都用“函数块”来表示。

## (2) 语法分析程序的总体结构及物理实现

### 总体结构:

语法分析程序由以下 8 个模块组成:

- 1) 求文法 $G$ : 识别以**文本形式保存**的文法 $G$ , 并获得非终结符 $V$ , 终结符 $T$ , 表达式 $P$ 以及开始符号 $S$ 。
- 2) 求项目集闭包: 求解文法 $G$ 的拓广文法 $G'$ 的某个 LR(0)项目集 $I$ 的闭包。
- 3) 求后继项目集: 求解 $I$ 在遇到某个文法符号 $X$ 之后的后继项目集。
- 4) 求项目集规范族: 求解文法 $G$ 的拓广文法 $G'$ 的项目集规范族。
- 5) 求 FIRST 集: 求解文法 $G$ 中每个终结符或非终结符的 FIRST 集。
- 6) 求 FOLLOW 集: 求解文法 $G$ 中每个终结符或非终结符的 FOLLOW 集。
- 7) 求 SLR(1)分析表: 根据文法  $G$  的项目集规范族以及 FOLLOW 集, 构建 SLR(1)分析表
- 8) 进行 LR 分析: 根据词法分析得到的 token 序列以及 SLR(1)分析表, 完成 LR 分析。

### 物理实现:

首先执行函数 7), 在其内部依次执行 1)到 6)这 6 个功能函数, 最后执行函数 8)。具体地,

首先，加载语言，求解源程序的 first 集和 follow 集，初始化 SLR 分析表，初始化 action 表和 goto 表，然后初始化语法分析程序的状态栈 stateStack 和数值栈 valueStack，然后读入经过词法分析的源程序，然后开始循环对源程序进行语法分析，并且打印 action 表，goto 表，状态栈 stateStack。

### (3) 语法分析表及其数据结构和查找算法

#### 数据结构：

SLR(1)语法分析表由 action 表与 goto 表两部分组成，数据结构都为 python 的列表 list，其内部嵌套字典 dict。在外部用数字下标作为索引，代表着不同的状态，而内部用终结符或非终结符的名称作为索引，其存储的字符串即代表着该状态下遇到该字符需要执行的动作。

#### 查找算法：

首先把状态 0 和 # 压入栈中，将 token 序列装入输入缓冲区，然后根据栈顶的状态以及输入缓冲区当前第一个字符查询 SLR(1)分析表的 action 表，如果查询到的字符串是空的，说明出错，如果字符串是 acc，说明分析成功，如果字符串第一个字母是 s，后面跟着数字，说明要将缓冲区当前字符以及数字对应状态压入栈中，如果字符串第一个字母是 r，后面跟着数字，说明要根据数字对应的表达式归约，将表达式里的终结符、非终结符以及状态都从栈中弹出，然后再根据栈顶的状态以及输入缓冲区当前第一个字符查询 SLR(1)分析表的 goto 表，如果字符串为空，说明出错，否则将查询到的状态压入栈中。

### (4) 语法分析表的生成算法

算法的大致步骤如下：

**输入：**文法  $G=(V, T, P, S)$  的拓广文法  $G'$ ；

**输出：** $G'$  的 LR(0)分析表，即 action 表和 goto 表；

**步骤：**

1. 令  $I_0 = \text{CLOSURE}(\{S' \rightarrow S\})$ ，构造  $G'$  的 LR(0)项目集规范族  $C = \{I_0, I_1, \dots, I_n\}$
2. 让  $I_i$  对应状态  $i$ ， $I_0$  对应状态 0，0 为初始状态。

3. **For**  $k=0$  **To**  $n$  **Do**

(1) **If**  $A \rightarrow \alpha.a\beta \in I_k$  **And**  $a \in T$  &  $\text{GO}(I_k, a) = I_j$  **then**  $\text{action}[k, a] = S_j$ ;

(2) **If**  $A \rightarrow \alpha.B\beta \in I_k$  **And**  $B \in V$  &  $\text{GO}(I_k, B) = I_j$  **then**  $\text{goto}[k, B] = j$ ;

(3) **If**  $A \rightarrow \alpha. \in I_k$  **And**  $A \rightarrow \alpha$  为  $G$  的第  $j$  个产生式 **then**

**For**  $\forall a \in \text{FOLLOW}(A)$  **Do**

$\text{action}[k, a] = r_j$ ;

**End**

(4) **If**  $S' \rightarrow S. \in I_k$  **then**  $\text{action}[k, \#] = \text{acc}$

**End**

4. 上述(1)到(4)步未填入信息的表项均置为 error。

本次实验要构建的语法分析表为 SLR(1)分析表，而 SLR(1)分析表是建立在 LR(0)分析表基础上的，因此首先需要完成 LR(0)分析表的前置条件，再加上 FOLLOW 集即可。

根据构建算法，对于文法  $G$  的拓广文法  $G'$  的一个 LR(0)项目集  $I$ ，程序首先要求  $I$  的闭

包  $CLOSURE(I)$ ，然后要求  $I$  在遇到某个文法符号  $X$  之后的后继项目集  $GO(I, X)$ 。在不断地循环求出所有的项目集  $I$  以后，这些项目集  $I$  就是  $G'$  的项目集规范族  $C$ 。接着只要根据项目规范族  $C$  就能构建出对应的  $LR(0)$  分析表。

在  $LR(0)$  分析表的构造函数中修改其中一部分规则，就能得到  $SLR(1)$  分析表，但这部分规则需要某个非终结符  $A$  的  $FOLLOW$  集，而求  $FOLLOW(A)$  又需要每个终结符或非终结符  $X$  的  $FIRST(X)$ ，即  $FIRST(X)$ 。所以在求出  $FIRST(X)$  进而求出  $FOLLOW(A)$  以后，再加上项目集规范族  $C$  就能得到  $SLR(1)$  分析表了。

### (5) 错误处理

生成文法  $G$  对应的  $SLR(1)$  语法分析表时，可以通过查看项目集规范族  $C$  的构建过程以及  $FOLLOW$  集的求解过程来判断哪里出现了问题，如果文法  $G$  本身没有问题，则在  $LR$  分析以前不会出现错误。如果  $SLR1$  分析表中产生移进规约冲突，就报  $SLR1$  Conflict error 错误，然后程序退出。如果在进行  $LR$  分析时出错，说明是词法分析得到的 token 序列不符合语法，在读取到某个字符时，程序在  $SLR(1)$  分析表的对应位置查询到的动作是空白，说明无法继续归约下去，程序也会就此中止。因此只要查看程序输出的  $LR$  分析过程，就能查找到错误发生的位置以及类型。

## 三、实验结果

三次实验所有代码已部署于 github, <https://github.com/HITWH18SE/Compiler-Principles>

### (1) 测试样例 1:

```
1 ▼ int main(int a, int b){
2     int d;
3     a = 1;
4     if (a <= b) a = b;
5 ▼     else{
6         while (a > b){
7             a = a - b;
8         }
9     }
10 }
11 }
```

图 1 测试样例 1

如图 1 所示为一个没有语法错误的正确测试样例，预计的语法分析输出结果为分析成功。

语法分析结果：（由于篇幅有限，输出结果超过 1000 行，只显示部分结果）

```
1 状态栈: [0]
2 符号栈: ['#']
3 输入缓冲区: int id ( int id , int id , int id )
{ int id ; id = digit ; if ( id <= id ) { id =
id ; } else { while ( id > id ) { id = - ( id +
digit ) * id ; } } float id ( float id ) { id
= id * ( id * ( id + digit ) ) ; if ( ( id +
digit < digit ) || ( id - id > digit ) ) { id =
id ; } } #
4 分析表内容: S4
5 当前动作: 移进状态4, 输入符号int
6
7 状态栈: [0, 4]
8 符号栈: ['#', 'int']
9 输入缓冲区: id ( int id , int id , int id ) {
int id ; id = digit ; if ( id <= id ) { id = id
; } else { while ( id > id ) { id = - ( id +
digit ) * id ; } } float id ( float id ) { id
= id * ( id * ( id + digit ) ) ; if ( ( id +
digit < digit ) || ( id - id > digit ) ) { id =
id ; } } #
10 分析表内容: r13
11 当前动作: 按第13个产生式归约: ['int'] ->
变量类型, 将状态3压入栈中
```

```

580 分析表内容: r14
581 当前动作: 按第14个产生式归约: ['float'] ->
      变量类型, 将状态10压入栈中
582
583 状态栈: [0, 2, 3, 7, 8, 10]
584 符号栈: ['#', '函数定义', '变量类型', 'id', '(',
      '变量类型']
585 输入缓冲区: id ) { id = id * ( id * ( id +
      digit ) ) ; if ( ( id + digit < digit ) || ( id
      - id > digit ) ) { id = id ; } } #
586 分析表内容: S13
587 当前动作: 移进状态13, 输入符号id
588
589 状态栈: [0, 2, 3, 7, 8, 10, 13]
590 符号栈: ['#', '函数定义', '变量类型', 'id', '(',
      '变量类型', 'id']
591 输入缓冲区: ) { id = id * ( id * ( id + digit )
      ) ; if ( ( id + digit < digit ) || ( id - id >
      digit ) ) { id = id ; } } #
592 分析表内容: r5
593 当前动作: 按第5个产生式归约: ['变量类型', 'id']
      -> 传入参数, 将状态9压入栈中

1045 状态栈: [0, 2, 3, 7, 8, 9, 12, 15, 22, 30]
1046 符号栈: ['#', '函数定义', '变量类型', 'id', '(',
      '传入参数', ')', '{', '函数块', '}']
1047 输入缓冲区: #
1048 分析表内容: r4
1049 当前动作: 按第4个产生式归约: ['变量类型', 'id',
      '(', '传入参数', ')', '{', '函数块', '}'] ->
      函数定义, 将状态6压入栈中

1050
1051 状态栈: [0, 2, 6]
1052 符号栈: ['#', '函数定义', '函数定义']
1053 输入缓冲区: #
1054 分析表内容: r2
1055 当前动作: 按第2个产生式归约: ['函数定义',
      '函数定义'] -> 函数定义, 将状态2压入栈中

1056
1057 状态栈: [0, 2]
1058 符号栈: ['#', '函数定义']
1059 输入缓冲区: #
1060 分析表内容: r1
1061 当前动作: 按第1个产生式归约: ['函数定义'] ->
      程序, 将状态1压入栈中

1062
1063 状态栈: [0, 1]
1064 符号栈: ['#', '程序']
1065 输入缓冲区: #
1066 分析表内容: acc
1067 当前动作: 分析成功
1068

```

图 2 测试样例 1 结果

如上图所示为部分 LR 分析过程及其结果，可以看出输入缓冲区的字符按照 SLR(1)分析表进行压栈和归约，最终分析成功，与预期输出结果一致。

(2) 测试样例 2:

```

1  int hi(int d){
2      d / d + d;
3  }

```

图 3 测试样例 2

如图所示为一个有语法错误的测试样例，预计的输出结果为分析错误，并且显示出错时的字符位置。

语法分析结果:

```

2  状态栈: [0]
3  符号栈: ['#']
4  输入缓冲区: int id ( int id ) { id / id + id ; }
   #
5  分析表内容: S4
6  当前动作: 移进状态4, 输入符号int
7
8  状态栈: [0, 4]
9  符号栈: ['#', 'int']
10 输入缓冲区: id ( int id ) { id / id + id ; } #
11 分析表内容: r13
12 当前动作: 按第13个产生式归约: ['int'] ->
   变量类型, 将状态3压入栈中
13
14 状态栈: [0, 3]
15 符号栈: ['#', '变量类型']
16 输入缓冲区: id ( int id ) { id / id + id ; } #
17 分析表内容: S7
18 当前动作: 移进状态7, 输入符号id
19
20 状态栈: [0, 3, 7]
21 符号栈: ['#', '变量类型', 'id']
22 输入缓冲区: ( int id ) { id / id + id ; } #
23 分析表内容: S8
24 当前动作: 移进状态8, 输入符号(
25
26 状态栈: [0, 3, 7, 8]
27 符号栈: ['#', '变量类型', 'id', '(']
28 输入缓冲区: int id ) { id / id + id ; } #
29 分析表内容: S4
30 当前动作: 移进状态4, 输入符号int
31
32 状态栈: [0, 3, 7, 8, 4]
33 符号栈: ['#', '变量类型', 'id', '(', 'int']
34 输入缓冲区: id ) { id / id + id ; } #
35 分析表内容: r13
36 当前动作: 按第13个产生式归约: ['int'] ->
   变量类型, 将状态10压入栈中
37
38 状态栈: [0, 3, 7, 8, 10]
39 符号栈: ['#', '变量类型', 'id', '(', '变量类型']
40 输入缓冲区: id ) { id / id + id ; } #
41 分析表内容: S13
42 当前动作: 移进状态13, 输入符号id
43
44 状态栈: [0, 3, 7, 8, 10, 13]
45 符号栈: ['#', '变量类型', 'id', '(', '变量类型',
   'id']
46 输入缓冲区: ) { id / id + id ; } #
47 分析表内容: r5
48 当前动作: 按第5个产生式归约: ['变量类型', 'id'] ->
   传入参数, 将状态9压入栈中
49
50 状态栈: [0, 3, 7, 8, 9]
51 符号栈: ['#', '变量类型', 'id', '(', '传入参数']
52 输入缓冲区: ) { id / id + id ; } #
53 分析表内容: S12
54 当前动作: 移进状态12, 输入符号)
55
56 状态栈: [0, 3, 7, 8, 9, 12]
57 符号栈: ['#', '变量类型', 'id', '(', '传入参数',
   ')']
58 输入缓冲区: { id / id + id ; } #
59 分析表内容: S15

```

```
60 当前动作：移进状态15，输入符号{
61
62 状态栈：[0, 3, 7, 8, 9, 12, 15]
63 符号栈：['#', '变量类型', 'id', '(', '传入参数',
    ')', '{']
64 输入缓冲区：id / id + id ; } #
65 分析表内容：S19
66 当前动作：移进状态19，输入符号id
67
68 状态栈：[0, 3, 7, 8, 9, 12, 15, 19]
69 符号栈：['#', '变量类型', 'id', '(', '传入参数',
    ')', '{', 'id']
70 输入缓冲区：/ id + id ; } #
71 分析表内容：
72 当前动作：分析出错
73
```

图 4 测试样例 2 结果

输入缓冲区的字符按照 SLR(1)分析表进行压栈和归约，最终在 id / id + id 之间缺失的=符号处弹出了分析出错的提示，与预期输出结果一致。

#### 四、实验中遇到的问题总结

##### （二）思考题的思考与分析

思考题 1：给出在生成语法分析表时所遇到的困难，以及是如何处理的？

思考题 2：思考还可以什么形式来给出语法分析的结果？

思考题 3：如果在语法分析中遇到了语法错误，是应该中断语法分析呢，还是应该进行适当处理后继续语法分析，你是怎么处理的？

##### （一）实验过程中遇到的问题以及解决方法

##### 问题 1：FIRST 集与 FOLLOW 集的代码实现

在按照伪代码编写 FIRST 集与 FOLLOW 集的实际代码时，“集合不再变化时停止循环”这个描述是比较含糊的，这个时候我们需要更加形式化的判断方法来解决这个问题：增加一个标记变量，标记一下集合中的元素个数是否发生变化。

##### 问题 2：文法 G 的定义

类 C 语言文法大多是适用于 LL(1)分析法的文法，即消除了左递归的文法，但它并不适用于自底向上分析的 SLR(1)分析法。因为 SLR(1)分析法不存在“试探”的过程，那些消除了左递归的表达式虽然能推出 $\epsilon$ ，但输入缓冲区对分析表不会进行这种试探，所以必须要另一种格式的文法才行。在便捷性和适用性的权衡中，我们把大多产生式左部设置为函数块。

##### （二）思考题的思考与分析

##### 思考题 1：给出在生成语法分析表时所遇到的困难，以及是如何处理的？

1. 搞清楚了哪些算法需要 G，哪些需要 G'：求 C 与 FOLLOW 集的算法有的需要文法 G，有的又需要它的拓广文法 G'，而生成 SLR(1)的函数一开始的输入文法又是 G'，很容易在调用函数时将这两个文法搞混，导致结果出错。有时需把生成 SLR(1)的函数的输入文法改成了 G，在函数内部再进行拓广。
2. 对 token 序列进行预处理，建立一个输入缓冲区是比较必要的：对 token 序列进行 LR 分析时，我发现不能直接采用 token 序列，因为文法 G 的终结符有时是大类，比如标识符 id，而这是 token 序列里的种类，有时又是具体的属性值，比如具体的保留字、关键字。

以语法分析树的形式给出语法分析的结果，每次按照 SLR(1)表指示归约时，将表达式左部的字符作为父结点，表达式右部的字符作为子节点，这样自底向上地构造出一棵树，如果分析成功，则树的根节点就是文法  $G$  的开始字符  $S$ 。还可以通过绘制识别拓展文法全部活前缀的 DFA，从图中看出各个状态的转换，然后构造语法分析表，通过绘制有限确定状态机来显示语法分析的结果。

[illegible]

图 5 语法分析树结果

1. 如果是产生了移进规约冲突，或者是规约规约冲突，我们不必中断语法分析，可以对程序采取紧急方式恢复策略，发现错误时跳过一些输入符号，直到出现下一个语法成分包含的第一个符号为止，在实现的时候，通常用某个期望的同步记号作为相应的标记，这种错误处理方法的效果依赖于属于 **follow** 集合的同步记号的选择。
2. 如果是其他未知的语法错误，事先没有准备的，就可以中断语法分析，因为通常在这种情况下是很难处理的，而且这也是很少出现的。在语法分析过程中遇到了这类错误应该中断语法分析，也就意味着当前栈顶状态与输入缓冲区最前面的字符在 **SLR(1)** 分析表中对应的动作字符串是空白，接下来无论如何不可能继续归约下去。

## 五、实验体会



通过语法分析的实验，对 LR(0)分析法和 SLR(1)分析法都有了更加深刻的了解。LR(0)分析法不需要向前查看输入符号，只需要根据当前的栈顶状态就可以确定下一步所应采取的动作。但是，SLR(1)分析法让分析器向前查看一个输入符号，以便确定面对当前输入符号时是否进行规约，以及按照哪个产生式进行规约。只需要在 LR(0)分析法的算法中更改一小部分代码即可。理解构造 SLR(1)分析表的过程以后，语法分析也没有太多的难度，这些函数只要在合适的地方调用，就可以很顺利地完成整个流程。模块化编程的作用确实很大，划分功能子模块之后编程如鱼得水。

实现算法的顺序和书本的教学顺序是一样的，先实现闭包 CLOSURE，然后实现后继 GO，然后实现项目集规范族 C，然后实现 LR(0)分析表的构造，然后实现求 FIRST 集与 FOLLOW 集，最后修改 LR(0)分析表的代码，实现 SLR(1)分析表的构造。

在实现这些函数的过程中，难度最高的两个算法还是求 FIRST 集与 FOLLOW 集，书上的伪代码算法描述比较抽象，好在 python 语言实现比较简洁，最后还是解决了。同时，文法 G 的定义是关键，不能只考虑 LR(0)分析法的情况，还需考虑 SLR(1)分析法。

总之，在此次实验中，获益匪浅，独立完成实验的体验成就感很大。

指导教师评语：

日期：