

# 《编译原理》书面作业

截止日期：2025 年 10 月 28 日

若发现问题，或有任何想法（改进题目、调整任务量等等），请及时联系助教

**Q1.** 针对以下文法  $G[S]$  是否可以设计一个确定的自顶向下预测分析过程？如果可以，需要向前查看多少个输入符号？

1.

$$S \rightarrow A \mid B$$

$$A \rightarrow aAb \mid c$$

$$B \rightarrow aBbb \mid d$$

2.

$$S \rightarrow abA \mid \epsilon$$

$$A \rightarrow Saa \mid b$$

**参考答案：**

1. 不可以。
2. 可以，向前查看 2 个输入符号。

**Q2.** 考虑下列文法  $G[S]$ ：

$$S \rightarrow T < U \mid b > U$$

$$T \rightarrow aS < S \mid cU \mid > b$$

$$U \rightarrow > Ta \mid < Sb$$

1. 计算每个非终结符的 First 集合、Follow 集合。
2. 计算每个产生式的预测集合（PS）。该文法是否是 LL(1) 文法？说明原因。
3. 画出该文法的 LL(1) 分析预测表。
4. 现有输入符号串  $> b < > > ba$ ，运行表驱动 LL(1) 分析程序，请画出该过程的下推栈。
5. 设计该文法的递归下降分析程序，下面给出了相关全局变量和函数的声明，其中 `matchToken(char)` 的定义与课件一致。请写出 `parseS()` 的定义。

```

1 static char lookahead;
2 void parseS();
3 void parseT();
4 void parseU();
5 void matchToken(char expected);

```

参考答案:

1.

$$First(S) = \{a, b, c, >\}$$

$$First(T) = \{a, c, >\}$$

$$First(U) = \{<, >\}$$

$$Follow(S) = \{a, b, <, \#\}$$

$$Follow(T) = \{a, <\}$$

$$Follow(U) = \{a, b, <, \#\}$$

2.

$$PS(S \rightarrow T < U) = \{a, c, >\}$$

$$PS(S \rightarrow b > U) = \{b\}$$

$$PS(T \rightarrow aS < S) = \{a\}$$

$$PS(T \rightarrow cU) = \{c\}$$

$$PS(T \rightarrow > b) = \{>\}$$

$$PS(U \rightarrow > Ta) = \{>\}$$

$$PS(U \rightarrow < Sb) = \{<\}$$

该文法是 LL(1) 文法, 因为: (注意  $T$  的三条产生式的  $PS$  是两两相交为空, 不是三个相交为空)

$$PS(S \rightarrow T < U) \cap PS(S \rightarrow b > U) = \emptyset$$

$$PS(T \rightarrow aS < S) \cap PS(T \rightarrow cU) = \emptyset$$

$$PS(T \rightarrow aS < S) \cap PS(T \rightarrow > b) = \emptyset$$

$$PS(T \rightarrow cU) \cap PS(T \rightarrow > b) = \emptyset$$

$$PS(U \rightarrow > Ta) \cap PS(U \rightarrow < Sb) = \emptyset$$

3. 如下表所示。

|     | $a$      | $b$     | $c$     | $<$    | $>$     | $\#$ |
|-----|----------|---------|---------|--------|---------|------|
| $S$ | $T < U$  | $b > U$ | $T < U$ |        | $T < U$ |      |
| $T$ | $aS < S$ |         | $cU$    |        | $> b$   |      |
| $U$ |          |         |         | $< Sb$ | $> Ta$  |      |

4. 如下表所示。

| 下推栈      | 余留符号串       | 下一步动作                       |
|----------|-------------|-----------------------------|
| #S       | > b <>> ba# | 应用产生式 $S \rightarrow T < U$ |
| #U < T   | > b <>> ba# | 应用产生式 $T \rightarrow > b$   |
| #U < b > | > b <>> ba# | 匹配栈顶和当前输入符号                 |
| #U < b   | b <>> ba#   | 匹配栈顶和当前输入符号                 |
| #U <     | <>> ba#     | 匹配栈顶和当前输入符号                 |
| #U       | >> ba#      | 应用产生式 $U \rightarrow > Ta$  |
| #aT >    | >> ba#      | 匹配栈顶和当前输入符号                 |
| #aT      | > ba#       | 应用产生式 $T \rightarrow > b$   |
| #ab >    | > ba#       | 匹配栈顶和当前输入符号                 |
| #ab      | ba#         | 匹配栈顶和当前输入符号                 |
| #a       | a#          | 匹配栈顶和当前输入符号                 |
| #        | #           | 结束                          |

5. 如下所示。(不必拘泥于语法细节)

```

1 void parseS() {
2     switch lookahead {
3         case 'a':
4         case 'c':
5         case '>':
6             parseT();
7             matchToken('<');
8             parseU();
9             break;
10        case 'b':
11            matchToken('b');
12            matchToken('>');
13            parseU();
14            break;
15        default:
16            printf("syntax error\n");
17            exit(1);
18    }
19 }
```

**Q3.** 考虑下列 LL(1) 文法 G[S]:

$$S \rightarrow P$$

$$P \rightarrow \wedge PP \mid \vee PP \mid \neg P \mid \underline{id}$$

其中,  $\wedge, \vee, \neg$  分别代表命题逻辑与、或、非等运算符单词,  $\underline{id}$  代表标识符单词。现有输入符号串:  $\vee \vee a \wedge bc \vee \neg a \wedge cb$ 。

1. 在针对该符号串的表驱动 LL(1) 分析过程中, 分析栈中最多会出现几个  $S$ 、几个  $P$  ?
2. 若因误操作使输入串多了一个符号, 变为  $\vee \vee a \wedge bc \vee \neg a \wedge cb$ , 当分析过程中发生错误时, 关于报错信息, 你认为最不可能的选择是 (4 选 1):  
(1) 缺运算数      (2) 多运算数      (3) 缺运算符      (4) 多运算符
3. 如果想要从该出错位置恢复分析, 可以进行什么操作?

**参考答案:**

1. 最多会出现 1 个  $S$ , 3 个  $P$ 。
2. 缺运算数。说明: 读入第二个  $c$  时出错, 可以直接报“多运算数”(多第二个  $c$ ); 若不遇到第二个  $c$ , 而是遇到一个运算符, 则不会出错, 所以可以报“缺运算符”; 另外, 若此时输入已结束, 则不会出错, 但下一个输入符号是运算符 ( $\vee$ ), 所以也可以报“多运算符”。
3. 可以删掉当前输入符号 ( $c$ ), 并将出错时使用的产生式左边的非终结符 ( $P$ ) 退回到分析栈顶, 然后就可以恢复分析了。可能还有其他恢复手段, 只要合理都是可以的。

**Q4. 按要求进行文法变换:**

1. 提取左公因子:

$$S \rightarrow T \mid T + T \mid T * T$$

$$T \rightarrow Ta \mid Tb \mid cU$$

$$U \rightarrow U0 \mid U1 \mid \epsilon$$

2. 按照  $S, U, T$  的顺序消除左递归:

$$S \rightarrow S + S \mid (S) \mid T$$

$$T \rightarrow UU b \mid Ta$$

$$U \rightarrow TT c \mid c$$

**参考答案:**

- 1.

$$S \rightarrow TS'$$

$$S' \rightarrow +T \mid *T \mid \epsilon$$

$$T \rightarrow TT' \mid cU$$

$$T' \rightarrow a \mid b$$

$$U \rightarrow UU' \mid \epsilon$$

$$U \rightarrow 0 \mid 1$$

2. 对于  $S$ , 消除直接左递归:

$$S \rightarrow (S)S' \mid TS'$$

$$S' \rightarrow +SS' \mid \epsilon$$

$$T \rightarrow UU b \mid Ta$$

$$U \rightarrow TT c \mid c$$

对于  $U$ ，暂时无需消除；

对于  $T$ ，首先用  $T \rightarrow TTcUb \mid cUb$  替代  $T \rightarrow UUb$ ，再消除直接左递归：

$$S \rightarrow (S)S' \mid TS'$$

$$S' \rightarrow +SS' \mid \epsilon$$

$$T \rightarrow cUbT'$$

$$T' \rightarrow TcUbT' \mid aT' \mid \epsilon$$

$$U \rightarrow TTc \mid c$$

# Principles and Practice of Compiler Construction HW

Due: 10/28/2025

**If you have any suggestions, feel free to contact the TAs.**

**Q1.** Can we design a definite top-down parsing process for the CFG  $G[S]$  as follows? If possible, how many look-ahead tokens are needed?

1.

$$S \rightarrow A \mid B$$

$$A \rightarrow aAb \mid c$$

$$B \rightarrow aBbb \mid d$$

2.

$$S \rightarrow abA \mid \epsilon$$

$$A \rightarrow Saa \mid b$$

**Solution:**

1. No.

2. Yes, with 2 look-ahead tokens.

**Q2.** Consider the following CFG  $G[S]$ :

$$S \rightarrow T < U \mid b > U$$

$$T \rightarrow aS < S \mid cU \mid > b$$

$$U \rightarrow > Ta \mid < Sb$$

1. Calculate the FIRST and FOLLOW sets of each of the nonterminals.
2. Calculate the prediction sets (PS) of each production. Is the grammar LL(1)? Why or why not?
3. Draw the LL(1) parsing table of the grammar.
4. Consider input string  $> b < > > ba$ , run the LL(1) parsing algorithm using the table, draw the stack during the process.
5. Design a recursive descent program for the grammar. Declaration of global variables and functions are given as follows, where the definition of `matchToken(char)` is the same as

in the slide. Give the definition of function `parseS()`

```

1 static char lookahead;
2 void parseS();
3 void parseT();
4 void parseU();
5 void matchToken(char expected);

```

**Solution:**

1.

$$First(S) = \{a, b, c, >\}$$

$$First(T) = \{a, c, >\}$$

$$First(U) = \{<, >\}$$

$$Follow(S) = \{a, b, <, \#\}$$

$$Follow(T) = \{a, <\}$$

$$Follow(U) = \{a, b, <, \#\}$$

2.

$$PS(S \rightarrow T < U) = \{a, c, >\}$$

$$PS(S \rightarrow b > U) = \{b\}$$

$$PS(T \rightarrow aS < S) = \{a\}$$

$$PS(T \rightarrow cU) = \{c\}$$

$$PS(T \rightarrow > b) = \{>\}$$

$$PS(U \rightarrow > Ta) = \{>\}$$

$$PS(U \rightarrow < Sb) = \{<\}$$

The grammar is LL(1), because:

*Pay attention to: we need the intersection of any two of  $T$ 's production's  $PS$ s, rather than the intersection of all three, to be empty.*

$$PS(S \rightarrow T < U) \cap PS(S \rightarrow b > U) = \emptyset$$

$$PS(T \rightarrow aS < S) \cap PS(T \rightarrow cU) = \emptyset$$

$$PS(T \rightarrow aS < S) \cap PS(T \rightarrow > b) = \emptyset$$

$$PS(T \rightarrow cU) \cap PS(T \rightarrow > b) = \emptyset$$

$$PS(U \rightarrow > Ta) \cap PS(U \rightarrow < Sb) = \emptyset$$

3. As follows.

|     | $a$      | $b$     | $c$     | $<$    | $>$     | $\#$ |
|-----|----------|---------|---------|--------|---------|------|
| $S$ | $T < U$  | $b > U$ | $T < U$ |        | $T < U$ |      |
| $T$ | $aS < S$ |         | $cU$    |        | $> b$   |      |
| $U$ |          |         |         | $< Sb$ | $> Ta$  |      |

4. As follows.

| Stack    | Remaining Symbols | Next Action                          |
|----------|-------------------|--------------------------------------|
| #S       | > b <>> ba#       | Use production $S \rightarrow T < U$ |
| #U < T   | > b <>> ba#       | Use production $T \rightarrow > b$   |
| #U < b > | > b <>> ba#       | Match token                          |
| #U < b   | b <>> ba#         | Match token                          |
| #U <     | <>> ba#           | Match token                          |
| #U       | >> ba#            | Use production $U \rightarrow > Ta$  |
| #aT >    | >> ba#            | Match token                          |
| #aT      | > ba#             | Use production $T \rightarrow > b$   |
| #ab >    | > ba#             | Match token                          |
| #ab      | ba#               | Match token                          |
| #a       | a#                | Match token                          |
| #        | #                 | End                                  |

5. As follows.

*No need to concern the grammatical detail.*

```

1 void parseS() {
2     switch lookahead {
3         case 'a':
4         case 'c':
5         case '>':
6             parseT();
7             matchToken('<');
8             parseU();
9             break;
10        case 'b':
11            matchToken('b');
12            matchToken('>');
13            parseU();
14            break;
15        default:
16            printf("syntax error\n");
17            exit(1);
18    }
19 }
```

**Q3.** Consider the following LL(1) grammar G[S]:

$$S \rightarrow P$$

$$P \rightarrow \wedge PP \mid \vee PP \mid \neg P \mid \underline{id}$$

Where  $\wedge, \vee, \neg$  stand for operator tokens logic and, logic or, logic not, respectively,  $\underline{id}$  stands for identifier tokens. Consider the input string:  $\vee \vee a \wedge bc \vee \neg a \wedge cb$ .

1. During the table-driven LL(1) parsing process for this input string, what is the maximum number of  $S$  and  $P$  that can appear in the parsing stack?
2. If an extra symbol appears in the input string due to some mistake, the new string being  $\vee \vee a \wedge bcc \vee \neg a \wedge cb$ . When the parsing process comes to an error, which is the least possible error info?  
 (1) missing operand      (2) extra operand      (3) missing operator      (4) extra operator
3. If we want to resume the parsing process from where the error happens, what can we do?

**Solution:**

1. 1  $S$ , 3  $P$ .
2. Missing operand. Explanation:  
 Error occurs when reading the second  $c$ , we can just throw "extra operand" (the second  $c$  is extra);  
 If we get an operator rather than the second  $c$ , no error will occur, so we can throw "missing operator";  
 Besides, if the input ends after the second  $c$ , no error will occur, but the next input symbol is an operator  $\vee$ , so we can also throw "extra operator".
3. We can delete the current symbol  $c$ , and push the nonterminal  $P$  from the left-hand side of the production rule used at the error point back onto the top of the parsing stack, and we can continue analysing.  
*Maybe there are other manners, rational is ok.*

**Q4.** Do grammar transformations as required:

1. Left factor:

$$S \rightarrow T \mid T + T \mid T * T$$

$$T \rightarrow Ta \mid Tb \mid cU$$

$$U \rightarrow U0 \mid U1 \mid \epsilon$$

2. Eliminate left recursion in the order of  $S, U, T$ :

$$S \rightarrow S + S \mid (S) \mid T$$

$$T \rightarrow UU b \mid Ta$$

$$U \rightarrow TTc \mid c$$

**Solution:**

1.

$$\begin{aligned}
S &\rightarrow TS' \\
S' &\rightarrow +T \mid *T \mid \epsilon \\
T &\rightarrow TT' \mid cU \\
T' &\rightarrow a \mid b \\
U &\rightarrow UU' \mid \epsilon \\
U &\rightarrow 0 \mid 1
\end{aligned}$$

2. For  $S$ , eliminate immediate left recursion:

$$\begin{aligned}
S &\rightarrow (S)S' \mid TS' \\
S' &\rightarrow +SS' \mid \epsilon \\
T &\rightarrow UUb \mid Ta \\
U &\rightarrow TTc \mid c
\end{aligned}$$

For  $U$ , no need to eliminate:

For  $T$ , first use  $T \rightarrow TTcUb \mid cUb$  to substitute  $T \rightarrow UUb$ , and then eliminate immediate left recursion:

$$\begin{aligned}
S &\rightarrow (S)S' \mid TS' \\
S' &\rightarrow +SS' \mid \epsilon \\
T &\rightarrow cUbT' \\
T' &\rightarrow TcUbT' \mid aT' \mid \epsilon \\
U &\rightarrow TTc \mid c
\end{aligned}$$