4 自顶向下解析

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆ 掌握 LL(1) 文法及其解析算法
- ⋆⋆⋆ 掌握 PEG 文法的 Packrat 解析算法
- 了解 Earley 解析算法

4.1 自顶向下解析

给定一套语法规则 G 和句子 s，寻找由 G 推导出 s 的过程称为解析。本章将介绍一种自顶向下的解析方法，即从 G 的起始符号出发，逐步展开非终结符，直到生成的终结符序列与目标句子完全匹配。具体而言，本章采用最左推导策略，即在每一步中选择当前句型中最左侧的非终结符进行展开。对于无二义性的语法 G，若 $s \in L(G)$，则存在且仅存在一种最左推导；若 $s \notin L(G)$，则不存在任何有效的解析过程。

语法解析的难点在于：在每一步推导中，一个非终结符可能对应多条产生式，如何选择合适的产生式是关键问题；此外，某些文法中可能存在递归结构，从而导致解析过程无法终止。一种基本思路是对上下文无关文法施加一定限制，以避免复杂情况。接下来，将介绍几种在不同限制条件下的解析方法：

- LL(1) 文法：不允许左递归，且不允许回溯
- PEG 文法：不允许左递归
- Earley 算法：对上下文无关文法不作限制

4.2 LL(1) 文法和解析

4.2.1 LL(1) 文法

为了降低解析算法的复杂度，可以对上下文无关文法施加一定的限制。LL(1) 文法（Left-to-right, Leftmost, 前瞻 1 个字符）有两个基本要求：一是不含左递归，二是在解析过程中不需要回溯。接下来将分别讨论这两个性质。

左递归与消除

对于一条产生式，若其右侧推导出的句型的最左符号可以为该产生式左侧的非终结符，则称该文法存在左递归。例如：`E` $\mapsto$ `E OP1 E1`。左递归会导致自顶向下解析过程中出现无限递归，从而使解析无法终止。通常可以通过对文法进行等价变换来消除左递归。

一般地，对于形如：

$$X \mapsto X\alpha_1 \mid X\alpha_2 \mid \cdots \mid \beta_1 \mid \beta_2$$

的产生式（其中 β_i 不以 X 开头），可以改写为：

$$\begin{aligned} X &\mapsto \beta_1 Y \mid \beta_2 Y \\ Y &\mapsto \alpha_1 Y \mid \alpha_2 Y \mid \epsilon \end{aligned}$$

上一章定义的计算器语法中，第 [1] 条和第 [3] 条产生式存在左递归。通过应用上述方法，可以消除这些左递归。修改后的语法规则如式 4.1 所示。

```
[1] E  ↦  E1 E'
[2] E'  ↦  OP1 E1 E'
[3] E'  ↦  ϵ
[4] E1  ↦  E2 E1'
[5] E1'  ↦  OP2 E2 E1'
[6] E1'  ↦  ϵ
[7] E2  ↦  E3 OP3 E2
[8] E2  ↦  E3
[9] E3  ↦  NUM
[10] E3  ↦  '('  E  ')'
[11] NUM  ↦  <UNUM>
[12] NUM  ↦  '-'  <UNUM>
[13] OP1  ↦  '+'
[14] OP1  ↦  '-'
[15] OP2  ↦  '*'
[16] OP2  ↦  '/'
[17] OP3  ↦  '^'
```
(4.1)

无回溯语法

对于同一非终结符的任意两条产生式，若它们可能推导出的串的首个终结符不同，则可以通过前瞻一个终结符来唯一确定所使用的产生式。当产生式右侧的首符号为非终结符时，需要递归考察该非终结符所能推导出的首终结符。

以规则 4.2 为例，若当前待展开的非终结符为 X，则根据下一个输入符号为 'a'、'b' 或 'c'，总可以选择出唯一的产生式，从而无需回溯。

```
[i] X ↦ 'a'...
[j] X ↦ 'b'...
[k] X ↦ Y...
[p] Y ↦ 'c'...
...
```
(4.2)

当语法存在回溯时，可以通过提取左公因子对文法进行等价变换，从而消除回溯。例如：

```
X ↦ 'a' A | 'a' B | 'b'
        ⇓
X ↦ 'a' Y | 'b'
Y ↦ A | B
```
(4.3)

语法规则 4.1 中，第 [7] 条和第 [8] 条产生式同属于非终结符 E2，且右侧的首个符号都是 E3，因此在某些情况下可能导致回溯问题。通过应用上述左公因子提取方法，可以对该部分语法进行改写，从而消除回溯。修改后的结果见语法规则 4.4 。

```
[1] E ↦ E1 E'
[2] E' ↦ OP1 E1 E'
[3] E' ↦ ϵ
[4] E1 ↦ E2 E1'
[5] E1' ↦ OP2 E2 E1'
[6] E1' ↦ ϵ
[7] E2 ↦ E3 E2'
[8] E2' ↦ OP3 E2
[9] E2 ↦ ϵ
[10] E3 ↦ NUM
[11] E3 ↦ '(' E ')'
[12] NUM ↦ <UNUM>
[13] NUM ↦ '-' <UNUM>
[14] OP1 ↦ '+'
[15] OP1 ↦ '-'
[16] OP2 ↦ '*'
[17] OP2 ↦ '/'
[18] OP3 ↦ '^'
```
(4.4)

4.2.2 LL(1) 文法解析

接下来，我们利用 LL(1) 文法的无回溯特性，构建预测分析表，用于记录每个非终结符可产生的首个终结符及其对应产生式。

我们定义

$$First\big(X \xrightarrow{[i]} \beta_1\beta_2\dots\beta_n\big)$$

表示应用 X 的第 i 条产生式所能产生的首终结符集合。计算规则如下：

- 如果 $\epsilon \notin First(\beta_1)$，则
$$First\big(X \xrightarrow{[i]} \beta_1\beta_2\dots\beta_n\big) = First(\beta_1)$$
- 如果 $\beta_1,\dots,\beta_k$ 都可能产生 ϵ，则
$$First\big(X \xrightarrow{[i]} \beta_1\beta_2\dots\beta_n\big) = \bigcup_{j=1}^{k+1} First(\beta_j)$$

  其中 k 是使 β_{k+1} 首次不能产生 ϵ 的索引；如果所有 β_j 都可能产生 ϵ，则 $First$ 集还需包含 ϵ。

表 4.1 展示了语法 4.4 中每条产生式对应的 $First$ 集。

表 4.1: 语法 4.4 的 $First$ 集合

	<UNUM>	'+'	'-'	'*'	'/'	'^'	'('	')'	ϵ
E	[1]		[1]				[1]		
E'		[2]	[2]						[3]
E1	[4]		[4]				[4]		
E1'				[5]	[5]				[6]
E2	[7]		[7]				[7]		
E2'						[8]			[9]
E3	[10]		[10]				[11]		
NUM	[12]		[13]						
OP1		[14]	[15]						
OP2				[16]	[17]				
OP3						[18]			

说明：行表示非终结符，列表示终结符，单元格中的值表示对应的规则编号。

由于句子词元序列中不包含 ϵ，$\epsilon \in First(X \to \beta)$ 对规则选择的帮助有限。因此需要对表 4.1 中的 ϵ 一列进行特殊处理。主要思路是：当 $\epsilon \in First(X \to \beta)$ 时，进一步考虑 X 之后可能出现的字符 $Follow(X \xrightarrow{[i]} ...)$，并据此决定是否采用规则 $X \to \epsilon$。为此，我们使用 $First^+(X \xrightarrow{[i]} \beta)$ 表示应用 X 的第 i 条规则所能产生的首个终结符集合（不含 ϵ）。

$$First^+(X \mapsto \beta) = \begin{cases} First(\beta), & \text{if } \epsilon \notin First(\beta) \\ First(\beta) \cup Follow(X), & \text{otherwise} \end{cases}$$

基于 $First^+$ 集的定义，可以严格表述 LL(1) 文法的无回溯性质：

$$\forall i \neq j, \quad First^+(X \to \beta_i) \cap First^+(X \to \beta_j) = \emptyset.$$

根据 $First^+$ 集合的计算方法，我们更新表 4.1 并消除其中的 ϵ 列，最终得到 LL(1) 解析表。结果如表 4.2 所示，可以看出该表的所有单元格至多包含一条规则，符合无回溯文法的特性。

表 4.2: LL(1) 解析表：记录每条规则的 $First^+$ 集合

	<UNUM>	'+'	'-'	'*'	'/'	'^'	'('	')'
E	[1]		[1]				[1]	
E'		[2]	[2]					[3]
E1	[4]		[4]				[4]	
E1'		[6]	[6]	[5]	[5]			[6]
E2	[7]		[7]				[7]	
E2'		[9]	[9]	[9]	[9]	[8]		[9]
E3	[10]		[10]				[11]	
NUM	[12]		[13]					
OP1		[14]	[15]					
OP2				[16]	[17]			
OP3						[18]		

通过查询 LL(1) 解析表，可以实现精准快速的解析。图 4.1 展示了使用表 4.2 解析算式 <UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)> 的过程及最终结果。图中每个节点的属性 m:[n] 表示在第 m 步应用规则 n 进行展开。

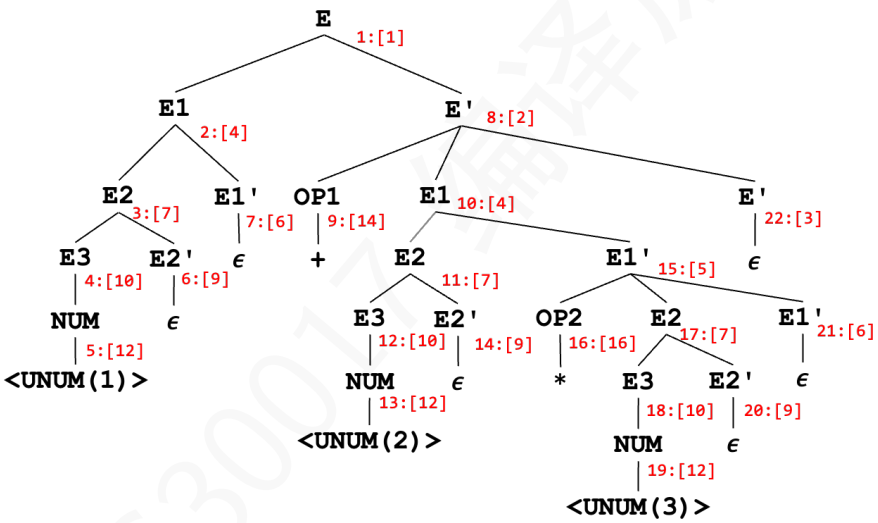


图 4.1: 应用表 4.2 解析 <UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)

4.3 PEG 文法与解析

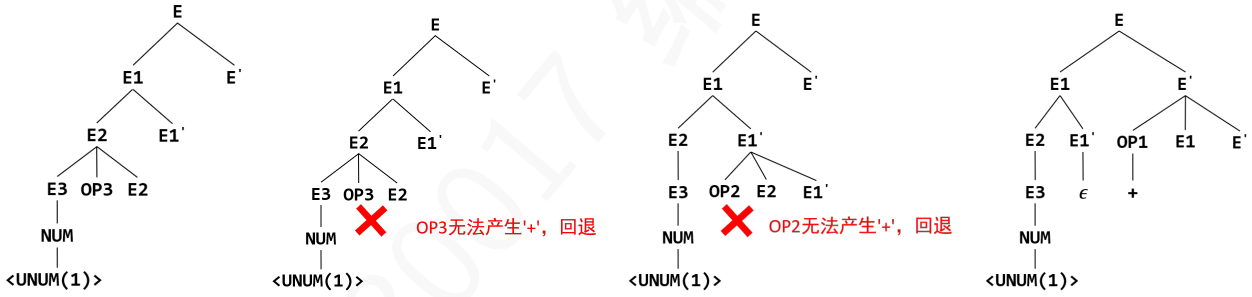
由于 LL(1) 文法在实际应用中受限，改写语法规则往往冗长。相比之下，PEG（Parsing Expression Grammar, PEG）[1] 允许有限回溯，能够更灵活地定义语言规则，因此应用场景更广泛。需要注意的是，PEG 同样不允许左递归，并通过有序选择（运算符：`/`）避免解析时的二义性。

对消除左递归后的文法 4.1进行适当改写，并引入条件选择，即可得到如下 PEG 文法：

```
E  ↦  E1 E'
E'  ↦  OP1 E1 E' / ϵ
E1  ↦  E2 E1'
E1'  ↦  OP2 E2 E1' / ϵ
E2  ↦  E3 OP3 E2 / E3
E3  ↦  NUM / '(' E ')'
NUM  ↦  <UNUM> / '-'  <UNUM>
OP1  ↦  '+' / '-'
OP2  ↦  '*' / '/'
OP3  ↦  '^'
```

(4.5)

由于 PEG 的递归下降解析可能导致指数级回溯，Packrat 解析方法 [2] 通过在每个输入位置缓存每条规则的解析结果，实现记忆化。这样既保持了 PEG 的完整回溯能力，又将解析时间复杂度降低到线性。



(a) `1 ∘ '+' 2 '*' 3`　(b) `1 ∘ '+' 2 '*' 3`　(c) `1 ∘ '+' 2 '*' 3`　(d) `1 '+' ∘ 2 '*' 3`

图 4.2: 使用 PEG 文法解析算式 `1 '+' 2 '*' 3` 的步骤示意图。游标 ∘ 左侧表示当前已处理字符。

Packrat 解析算法如代码 1 所示，其中每条规则的解析结果都会存入记忆表，从而避免重复解析并保证解析效率。图 4.2以算式 `1 '+' 2 '*' 3` 为例展示了 PEG 文法的解析过程。其中，在图 4.2b 中，当解析无法继续进行时，对应的记忆表状态如表 **??** 所示，此时触发回溯。具体而言，解析器由产生式 `E2 ↦ E3 OP3 E2` 回退并切换至 `E2 ↦ E3` 。由于此前在相同输入位置已经成功解析过 `E3`，因此可以直接复用其解析结果，而无需重新计算。

表 4.3: PEG 文法解析过程中记忆表的变化：左表为图 4.2b回溯发生时的中间状态，右图为最终解析完成后的状态。表中列 1 ∼ 5 表示输入串 `1 '+' 2 '*' 3` 的位置，行表示各非终结符。单元格中的数字表示在对应输入位置从非终结符解析成功的字符数。

(a) 回溯时的记忆表状态

	1	2	3	4	5
E					
E'					
E1					
E1'					
E2					
E3	1（复用）				
NUM	1				
OP1					
OP2					
OP3					
	1	+	2	*	3

(b) 最终解析完成后的记忆表

	1	2	3	4	5
E	5				
E'		4			
E1	1		3		
E1'				2	
E2	1		1		1
E3	1		1		1
NUM	1		1		1
OP1		1			
OP2				1	
OP3					
	1	+	2	*	3

算法 1 Packrat 解析算法

input: G: PEG grammar with a start symbol P; ts: token stream;
output: a parse result (success / failure) with parse tree;

procedure PackratParse(ts, G)
  初始化记忆表 M // $M[X, i]$ 记录非终结符 X 在位置 i 的解析结果
  return Parse(P, 0)
end procedure
procedure Parse(X, i)
  if $M[X, i]$ 已存在 **then**
    return $M[X, i]$ // 直接返回缓存结果（避免重复解析）
  end if
  for each $X \leftarrow \alpha \in G$ **do** // 按顺序尝试 PEG 产生式
    $(success, j) \leftarrow$ ParseSeq(α, i)
    if $success$ **then**
      $M[X, i] \leftarrow (true, j)$
      return $(true, j)$ // PEG 选择第一个成功的分支
    end if
  end for
  $M[X, i] \leftarrow (false, i)$
  return $(false, i)$
end procedure
procedure ParseSeq(α, i)
  $j \leftarrow i$
  for each 符号 s in α **do** // 依次匹配序列中的符号
    if s 是终结符 **then**
      if $ts[j] == s$ **then**
        $j \leftarrow j + 1$
      else
        return $(false, i)$
      end if
    else// s 是非终结符
      $(success, j') \leftarrow$ Parse(s, j)
      if not $success$ **then**
        return $(false, i)$
      end if
      $j \leftarrow j'$
    end if
  end for
  return $(true, j)$
end procedure

4.4 Earley 解析算法

Earley 解析算法 [3] 是一种通用的 CFG 解析算法，不对具体的语法规则做任何限制。Earley 算法包括以下三种基本操作：

- **预测**：对于规范项 $X \to \alpha \circ Y\beta$（符号 $\circ$ 表示当前解析位置），根据语法规则推导 $Y \to \circ\gamma$；
- **扫描**：如果下一个待解析的终结符是 a，且存在状态 $X \to \alpha \circ a\beta$，则移进终结符 a，并更新规范项为 $X \to \alpha a \circ \beta$；
- **完成**：如果规范项为 $Y \to \gamma \circ$，即完成了对非终结符 Y 的展开，则将所有关联状态 $X \to \alpha \circ Y\beta$ 更新为 $X \to \alpha Y \circ \beta$。

算法 2 详细描述了 Earley 算法的解析过程。该算法在符号展开时能够有效避免无限递归和路径爆炸问题，提高解析效率。

算法 2 Earley 解析算法

```
input: G: context-free grammar with a start symbol P; ts: token stream;
output: a parse tree;
procedure EarleyParse(ts, G)
    for each P → γ ∈ G do // 初始化，选取 G 中每一条以 P 开头的规则
        S[0].add((P → ∘γ, 0)) // 添加到 Earley 解析状态 S[0] 中，第二个参数 0 表示规则起源于 Earley 解析状态 S[0]
    end for
    for each i in 0..ts.len() do // 遍历每一个终结符
        for each item in S[i] do // 遍历 S[i] 中的每一条规则状态
            match NextSymbol(item) :
                case END ⇒ Complete(item, i) // 当前规则右侧字符串已经全部匹配，则执行完成操作
                case ts[i] ⇒ Scan(item, i, ts) // 当前规则状态的下一个字符为终结符，且恰好是目标终结符，执行扫描操作
                case NON-TERMINAL ⇒
                    Predict(item, i, G) // 当前规则状态的下一个字符为非终结符，执行预测操作
            end match
        end for
    end for
end procedure
procedure Complete((A → β∘, j), i) // 完成操作：j 表示此条规则的起始 Earley 解析状态，i 是当前 Earley 解析状态
    for each (B → α∘Aδ, k)∈ S[j] do // 完成操作只会更新 S[j] 中的相关的规则状态
        S[i].add((B → αA∘δ, k)) // 移进完成的非终结符 A
        if δ == ϵ then // 规则 B → αA∘ 已经扫描完成
            Complete((B → αA∘, k), i) // 继续对 S[k] 中相关的规则执行完成操作
        end if
    end for
end procedure
procedure Predict((A → α∘Bβ, j), i) // 预测操作
    for each B → γ in G do
        S[i].add((B → γ, i))
    end for
end procedure
procedure Scan((A → α∘aβ, j), i) // 扫描操作
    if a == ts[i] then
        S[i + 1].add((A → αa∘β, j)) // 移进终结符，并将新的规则状态添加到下一个 Earley 解析状态 S[i+1] 中
    end if
end procedure
```

下面以词元序列 `<UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)>` 为例[1]，演示 Earley 算法的解析过程，其具体步骤如表 4.4–4.9 所示。

表 4.4: 状态 $S[0]$：∘ `<UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)>`

序号	操作	条目	
		规范项	起源
1	初始化	E → ∘ E OP1 E1	S[0]
2	初始化	E → ∘ E1	S[0]
3	预测 2	E1 → ∘ E1 OP2 E2	S[0]
4	预测 2	E1 → ∘ E2	S[0]
5	预测 4	E2 → ∘ E3 OP3 E2	S[0]
6	预测 5	E2 → ∘ E3	S[0]
7	预测 5	E3 → ∘ NUM	S[0]
8	预测 5	E3 → ∘ '(' E ')'	S[0]
9	预测 7	NUM → ∘ <UNUM>	S[0]
10	预测 7	NUM → ∘ '-' <UNUM>	S[0]
11	扫描 9	-	-

表 4.5: 状态 $S[1]$：`<UNUM(1)>` ∘ `'+' <UNUM(2)> '*' <UNUM(3)>`

序号	操作	条目	
		规范项	起源
1	扫描 s[0]-9	NUM → <UNUM> ∘	S[0]
2	完成：基于 1 更新 s[0]-7	E3 → NUM ∘	S[0]
3	完成：基于 2 更新 s[0]-5	E2 → E3 ∘ OP3 E2	S[0]
4	完成：基于 2 更新 s[0]-6	E2 → E3 ∘	S[0]
5	完成：基于 4 更新 s[0]-4	E1 → E2 ∘	S[0]
6	完成：基于 5 更新 s[0]-2	E → E1 ∘	S[0]
7	完成：基于 5 更新 s[0]-3	E1 → E1 ∘ OP2 E2	S[0]
8	完成：基于 6 更新 s[0]-1	E → E ∘ OP1 E1	S[0]
9	预测 3	OP3 → ∘ '^'	S[1]
10	预测 7	OP2 → ∘ '*'	S[1]
11	预测 7	OP2 → ∘ '/'	S[1]
12	预测 8	OP1 → ∘ '+'	S[1]
13	预测 8	OP1 → ∘ '-'	S[1]
14	扫描 12	-	-

[1]符号说明：`'+'` 表示词元 `<ADD>`，`'*'` 表示词元 `<MUL>`。

表 4.6: 状态 $S[2]$：<UNUM(1)> '+' ∘ <UNUM(2)> '*' <UNUM(3)>

序号	操作	条目	
		规范项	起源
1	扫描 s[1]-12	OP1 → '+' ∘	S[1]
2	完成：基于 1 更新 s[1]-8	E → E OP1 ∘ E1	S[0]
3	预测 2	E1 → ∘ E1 OP2 E2	S[2]
4	预测 2	E1 → ∘ E2	S[2]
5	预测 4	E2 → ∘ E3 OP3 E2	S[2]
6	预测 5	E2 → ∘ E3	S[2]
7	预测 5	E3 → ∘ NUM	S[2]
8	预测 5	E3 → ∘ '(' E ')'	S[2]
9	预测 7	NUM → ∘ <UNUM>	S[2]
10	预测 7	NUM → ∘ '-' <UNUM>	S[2]
11	扫描 9	-	-

表 4.7: 状态 $S[3]$：<UNUM(1)> '+' <UNUM(2)> ∘ '*' <UNUM(3)>

序号	操作	条目	
		规范项	起源
1	扫描 s[2]-9	NUM → <UNUM> ∘	S[2]
2	完成：基于 1 更新 s[2]-7	E3 → NUM ∘	S[2]
3	完成：基于 2 更新 s[2]-5	E2 → E3 ∘ OP3 E2	S[2]
4	完成：基于 2 更新 s[2]-6	E2 → E3 ∘	S[2]
5	完成：基于 4 更新 s[2]-4	E1 → E2 ∘	S[2]
6	完成：基于 5 更新 s[2]-2	E → E OP1 E1 ∘	S[0]
7	完成：基于 5 更新 s[2]-3	E1 → E1 ∘ OP2 E2	S[2]
8	预测 3	OP3 → ∘ '^'	S[3]
9	预测 7	OP2 → ∘ '*'	S[3]
10	预测 7	OP2 → ∘ '/'	S[3]
11	扫描 9	-	-

说明：S[3]-6 还会导致完成和更新 S[0]-1 的操作，如果下一个词元是'+' 或'-' 时有用；为节约空间，此处未列出相关规范项。

表 4.8: 状态 $S[4]$：<UNUM(1)> '+' <UNUM(2)> '*' ∘ <UNUM(3)>

序号	操作	条目	
		规范项	起源
1	扫描 s[3]-9	OP2 → '*' ∘	S[3]
2	完成：基于 1 更新 s[3]-7	E1 → E1 OP2 ∘ E2	S[2]
3	预测 2	E2 → ∘ E3 OP3 E2	S[4]
4	预测 2	E2 → ∘ E3	S[4]
5	预测 3	E3 → ∘ NUM	S[4]
6	预测 3	E3 → ∘ '(' E ')'	S[4]
7	预测 5	NUM → ∘ <UNUM>	S[4]
8	预测 5	NUM → ∘ '-' <UNUM>	S[4]
11	扫描 7	-	-

表 4.9: 状态 $S[5]$：`<UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)>` ∘

序号	操作	条目	
		规范项	起源
1	扫描 s[4]-7	`NUM` → `<UNUM>` ∘	S[4]
2	完成：基于 1 更新 s[4]-5	`E3` → `NUM` ∘	S[4]
3	完成：基于 2 更新 s[4]-3	`E2` → `E3` ∘ `OP3 E2`	S[4]
4	完成：基于 2 更新 s[4]-4	`E2` → `E3` ∘	S[4]
5	完成：基于 4 更新 s[4]-2	`E1` → `E1 OP2 E2` ∘	S[2]
6	完成：基于 5 更新 s[2]-2	`E` → `E OP1 E1` ∘	S[0]

Earley 本质上是一种动态规划方法，可以有效支持左递归。具体而言，在处理左递归产生式（如 `E` ↦ `E OP1 E1`）时，预测步骤会将该产生式加入当前状态集，但由于每个项目由产生式、游标位置和起始位置唯一确定，已经出现过的项目不会被重复加入，从而避免了无限递归。

参考文献

[1] Bryan Ford “Parsing expression grammars: a recognition-based syntactic foundation.” In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2004.

[2] Bryan Ford. “Packrat parsing: simple, powerful, lazy, linear time, functional pearl.” ACM SIGPLAN Notices 37, no. 9 (2002): 36-47.

[3] Jay Earley. “An efficient context-free parsing algorithm.” *Communications of the ACM*, 1970.

练习

1) 已知如下正则表达式的语法规则：

```
[1] Regex  ↦  Regex '|' Concat
[2] Regex  ↦  Concat
[3] Concat  ↦  Concat Closure
[4] Concat  ↦  Closure
[5] Closure  ↦  Closure '*'
[6] Closure  ↦  Item
[7] Item  ↦  '(' Regex ')'
[8] Item  ↦  <Char>
```

(4.6)

a) 判断上述语法是否为 LL(1) 文法；若不是，请将其改写为 LL(1) 文法，并构造对应的预测分析表。

b) 判断上述语法是否符合 PEG 文法的要求；若不符合，请将其改写为等价的 PEG 文法，并给出对正则表达式 `ab*|c` 的解析过程。

c) 使用 Earley 算法对正则表达式 `ab*|c` 进行解析。

d) 分析 LL(1)、PEG（Packrat）和 Earley 三种解析方法在该问题上的时间复杂度及其差异。