4 自顶向下解析

徐辉，xuh@fudan.edu.cn

本章学习目标：

- 了解 Earley 解析算法；
- 掌握 LL(1) 文法；
- 掌握 LL(1) 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，找到由 G 推导出 s 的过程称为解析。本章介绍一种自顶向下的解析思路，即从 G 的初始符号开始，根据语法规则递归展开每个非终结符，直至最终生成的终结符序列与目标句子完全一致。本章采用最左推导方法，即每次选择当前状态最左侧的非终结符展开。对于无二义性的语法 G，如果 $s \in L(G)$，则有且只有一种解析方式；如果 $s \notin L(G)$，则不存在任何解析方式。

该语法解析问题的难点是每一步推导过程的非终结符可能存在多条展开规则，应如何选取唯一合适的规则？一个基本思路是根据当前非终结符和目标终结符选取产生式。下面分别介绍两种解析方法，Earley 算法和 LL(1) 文法。

4.2 Earley 解析算法

Earley 解析算法 [1] 是一种通用的 CFG 解析算法，即不对具体的 CFG 语法规则做任何限定。Earley 算法涉及以下三种基本操作（非正式定义）：

- **预测**：对于解析过程中产生的规范项 $X \to \alpha \circ Y\beta$（符号 $\circ$ 表示当前解析位置），根据语法规则展开 $Y \to \circ\gamma$；
- **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，并将规范项更新为 $X \to \alpha a \circ \beta$；
- **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，将所有关联状态 $X \to \alpha \circ Y\beta$ 更新为 $X \to \alpha Y \circ \beta$。

Earley 算法的具体解析过程详见算法 1。该算法可有效避免和应对符号展开时的无限递归和路径爆炸问题。

算法 1 Earley 解析算法

Input: G: context-free grammar with a start symbol P; ts: token stream;
Output: a parse tree;

```
1: procedure EarleyParse(ts, G)
2:     for each P → γ ∈ G do // 初始化，选取 G 中每一条以 P 开头的规则
3:         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]
4:     end for
5:     for each i in 0..ts.len() do // 遍历每一个终结符
6:         for each item in S[i] do // 遍历 S[i] 中的每一条规则状态
7:             match NextSymbol(item) :
8:                 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
9:                 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
10:                case NON-TERMINAL ⇒
11:                    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
12:            end match
13:        end for
14:    end for
15: end procedure
16: 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
17:    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
18:        S[i].add((B → αA∘δ, k)) // 移进完成的非终结符 A
19:        if δ == ϵ then // 规则 B → αA∘ 已经扫描完成
20:            Complete((B → αA∘, k), i) // 继续对 S[k] 中相关的规则执行完成操作
21:        end if
22:    end for
23: end procedure
24: procedure Predict((A → α∘Bβ, j), i) // 预测操作
25:    for each B → γ in G do
26:        S[i].add((B → γ, j))
27:    end for
28: end procedure
29: procedure Scan((A → α∘aβ, j), i) // 扫描操作
30:    if a == ts[i] then
31:        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
32:    end if
33: end procedure
```

下面以标签序列 `<UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3)>` 为例展示 Earley 算法的解析步骤，参见表 4.1-4.6。

表 4.1: 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 4.2: 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	-	-

表 4.3: 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 4.4: 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	-	-

表 4.5: 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 4.6: 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]

4.3 LL(1) 文法和解析

4.3.1 LL(1) 文法

为了降低解析算法的复杂度，我们可以强制要求 CFG 文法具备某些特性，如 LL(1)（Left-to-right, Leftmost, 前瞻 1 个字符）有两个基本要求：一是不含左递归，二是无回溯。下面对这两个特性进行探讨。

4.3.1.1 左递归和消除

对一条语法规则来说，如果其右侧推导出的第一个符号与左侧相同，则存在左递归问题，如 `E` ↦ `E OP1 E1`。左递归可能会导致解析时的规则搜索过程无限递归，无法终止。一般可以通过修改语法规则消除左递归，如采用下列方式。

$$\begin{aligned} &\texttt{X} \mapsto \texttt{X 'a'} \mid \texttt{X 'b'} \mid \texttt{'c'} \mid \texttt{'d'} \\ &\qquad\Downarrow \\ &\texttt{X} \mapsto \texttt{'c' Y} \mid \texttt{'d' Y} \\ &\texttt{Y} \mapsto \texttt{'a' Y} \mid \texttt{'b' Y} \mid \epsilon \end{aligned} \tag{4.1}$$

上一章定义的计算器语法规则中第 [1] 和 [3] 条存在左递归问题。应用上述方法可消除其中的左递归，结果如语法规则 4.2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  ↦  '^'
```
(4.2)

4.3.1.2 无回溯语法

对于每个非终结符的任意两条规则，如果其产生的首个终结符均不同，则前瞻一个终结符总能够选择一条正确的规则。当规则的首个字符是非终结符时，应对该非终结符递归展开直至遇到终结符为止。如规则 4.3中，如果当前规范项待展开的非终结符是 X，很容易根据下一个终结符是 'a'、'b'、或 'c' 选择出一条仅有的正确规则。

```
[i] X  ↦  'a'...
[j] X  ↦  'b'...
[k] X  ↦  Y...
[p] Y  ↦  'c'...
...
```
(4.3)

当语法规则存在回溯问题时，可以通过提取左公因子消除回溯。

```
X  ↦  'a' A | 'a' B | 'b'
          ⇓
X  ↦  'a' Y | 'b'
Y  ↦  A | B
```
(4.4)

语法规则 4.2中的第 [7] 和 [8] 两条规则右侧的首个符号都是 E3，存在回溯问题。应用上述方法对这两条规则进行改写可消除回溯问题。结果如语法规则 4.5所示。

$$
\begin{aligned}
&\texttt{[1] E} \mapsto \texttt{E1 E'}\\
&\texttt{[2] E'} \mapsto \texttt{OP1 E1 E'}\\
&\texttt{[3] E'} \mapsto \epsilon\\
&\texttt{[4] E1} \mapsto \texttt{E2 E1'}\\
&\texttt{[5] E1'} \mapsto \texttt{OP2 E2 E1'}\\
&\texttt{[6] E1'} \mapsto \epsilon\\
&\texttt{[7] E2} \mapsto \texttt{E3 E2'}\\
&\texttt{[8] E2'} \mapsto \texttt{OP3 E2}\\
&\texttt{[9] E2} \mapsto \epsilon\\
&\texttt{[10] E3} \mapsto \texttt{NUM}\\
&\texttt{[11] E3} \mapsto \texttt{'(' E ')'}\\
&\texttt{[12] NUM} \mapsto \texttt{<UNUM>}\\
&\texttt{[13] NUM} \mapsto \texttt{'-' <UNUM>}\\
&\texttt{[14] OP1} \mapsto \texttt{'+'}\\
&\texttt{[15] OP1} \mapsto \texttt{'-'}\\
&\texttt{[16] OP2} \mapsto \texttt{'*'}\\
&\texttt{[17] OP2} \mapsto \texttt{'/'}\\
&\texttt{[18] OP3} \mapsto \texttt{'\^{}'}
\end{aligned}
\tag{4.5}
$$

4.3.2 LL(1) 文法解析

我们接下来利用 LL(1) 文法无回溯的特性构建一张解析表，记录每个非终结符可以产生的首个终结符和相应的规则。我们定义 $First(X \xrightarrow{[i]} \beta_1\beta_2...\beta_n)$ 表示应用 X 的第 i 条规则可产生的首字符集合。如果 $\epsilon \notin \beta_1$，则 $First(X \xrightarrow{[i]} \beta_1\beta_2...\beta_n) = First(\beta_1)$；如果 $\epsilon \in \beta_1 \& ... \& \epsilon \in \beta_i$，则 $First(X \xrightarrow{[i]} \beta_1\beta_2...\beta_n) = First(\beta_1) \cup ... \cup First(\beta_{i+1})$。表 4.7展示了语法 4.5中每条规则对应的 $First$ 集合。

表 4.7: 语法 4.5的 $First$ 集合。行：非终结符；列：终结符；单元格：规则编号。

	<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]			

由于句子标签序列中不包含 ϵ，$\epsilon \in First(X \to \beta)$ 对于规则选取的帮助不大，因此还需对表 4.7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 $X \to \epsilon$。因此我们使用 $First^+(X \xrightarrow{[i]} \beta)$ 表示应用 X 的第 i 条规则可产生的首个终结符集合（不含 ϵ）。

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

基于上述定义，我们可以准确定义无回溯语法的必要性质：

$$\forall 1 \le i, j \le n, First^+(X \to \beta_i) \cap \text{ First }^+(X \to \beta_j) = \emptyset$$

基于 $First^+$ 集合的计算方法，我们更新表 4.7并消除其中的 ϵ 列，得到最终的 LL(1) 解析表。结果如表 4.8所示，可以看出该表的所有单元格至多存在一条规则，满足无回溯文法的特性。

表 4.8: 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]		

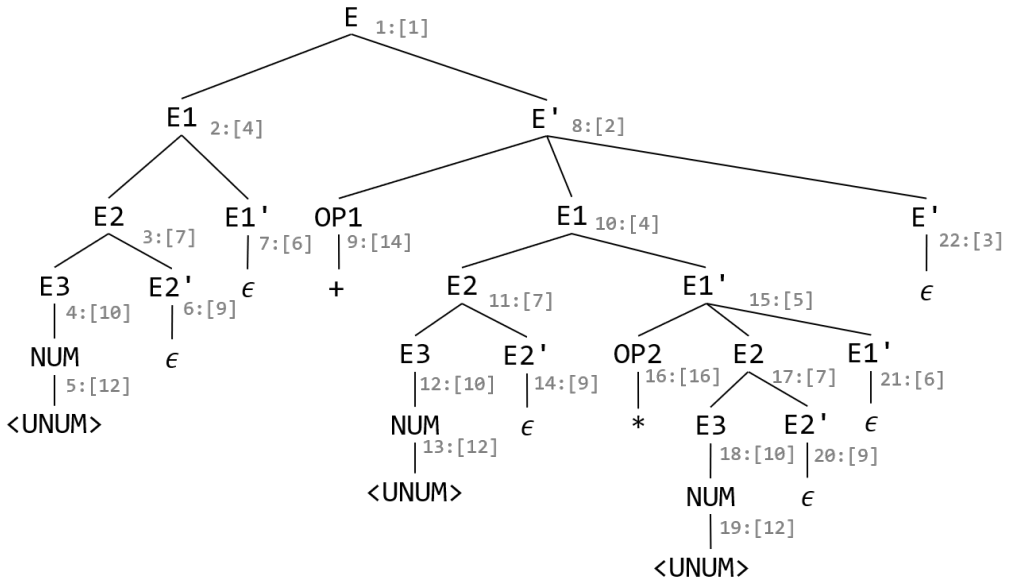

图 4.1: 应用表 4.8解析 <UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3) 的过程和语法解析树。

通过查询 LL(1) 解析表可以实现精准快速解析。图 4.1展示了使用表 4.8解析算式 <UNUM(1)> '+' <UNUM(2)> '*' <UNUM(3) 的过程和最终结果。图中每个节点的属性 m:[n] 表示在第 m 步对该节点应用规则 n 展开。

练习

1. 已知下列正则表达式 CFG 语法规则，应用 Earley 算法解析正则表达式 `ab*|c`。

$$
\begin{aligned}
&\texttt{[1] Regex} \mapsto \texttt{Regex '|' Concat}\\
&\texttt{[2] Regex} \mapsto \texttt{Concat}\\
&\texttt{[3] Concat} \mapsto \texttt{Concat Closure}\\
&\texttt{[4] Concat} \mapsto \texttt{Closure}\\
&\texttt{[5] Closure} \mapsto \texttt{Closure '*'}\\
&\texttt{[6] Closure} \mapsto \texttt{Item}\\
&\texttt{[7] Item} \mapsto \texttt{'(' Regex ')'}\\
&\texttt{[8] Item} \mapsto \texttt{<Char>}
\end{aligned}
\tag{4.6}
$$

2. 上述文法是否是 LL(1)，如果不是将其改为 LL(1) 并构造解析表。

3. 分析比较 Earley 算法和 LL(1) 算法的复杂度。

Bibliography

[1] Jay Earley. “An efficient context-free parsing algorithm.” Communications of the ACM 13, no. 2 (1970): 94-102.