**编译原理 (南京大学软件学院)** **Spring 2024**

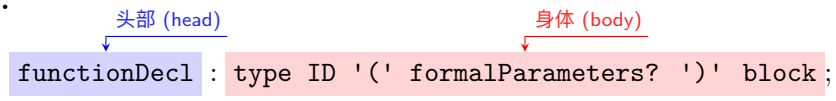
# 第 5~6 节: 上下文无关文法, $LL(1)$ 语法分析算法

*Lecturer:* 魏恒峰 *Scribes:* 张枕玮

## §1 上下文无关文法

**1.1. 回顾. 上下文无关文法语法.** 我们上节课学习了如何在 ANTLR 中书写语法, 我们借助下面的例子为每一个部分起一个名字.

**例子 1.1.**




每一行称为产生式. 意思是把左边的头部替换为右边的身体. 每一个的头部都是一个非终结符, 因为它还可以被替换为右边的内容. 右边的身体有可能是非终结符, 也可能是终结符 (如 ID, '(' 等), 也有可能是空串 $\epsilon$. 这样的文法我们叫做上下文无关文法. 我们给出其定义:

**定义 1.1 (上下文无关文法 (context-free grammar, CFG)).** 上下文无关文法 $G$ 是一个四元组 $G = (T, N, S, P)$ :

- $T$ 是**终结符号 (Terminal)** 集合, 对应于词法分析器产生的词法单元
- $N$ 是**非终结符号 (Non-terminal)** 集合
- $S$ 是**开始 (Start)** 符号 ($S \in N$ 且唯一)
- $P$ 是**产生式 (Production)** 集合

$$A \in N \longrightarrow \alpha \in (T \cup N)^*$$

- **头部/左部 (Head)** $A$ : 单个非终结符;
- **体部/右部 (Body)** $\alpha$ : 终结符与非终结符构成的串, 也可以是空串 $\epsilon$

$S$ 是开始符号, 所有的程序都从开始符号展开得到.

**例子 1.2.** 考虑

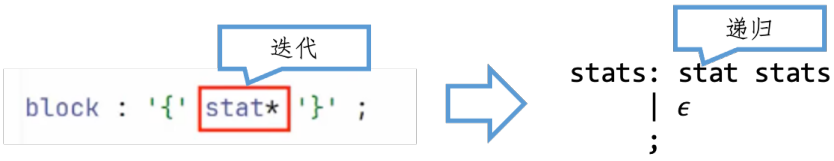

实际上他们是下面的简写 (用右递归的形式写出):

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' \mid \epsilon \\ T &\to FT' \\ T' &\to *FT' \mid \epsilon \\ F &\to (E) \mid \text{ id} \end{aligned}$$

另外一个上下文无关文法是

$$S \to aSa$$
$$S \to bSb$$
$$S \to a$$
$$S \to b$$
$$S \to \epsilon$$

但是在 ANTLR 中可以用类似正则表达式的方法 (“?”, “+”) 来实现. 实际上这也是可以的, 因为我们有扩展的 BNF 范式 (EBNF), 可以证明, 这两种情况是等价的. 这种形式只是一种简写. 比如, 带有 “?” 的可以拆解为两条规则, 同样可以按照如下的方法去除 “*”.




上下文无关文法强调左端一定是一个终结符. 否则就变成了上下文相关文法. 这就意味着我们还要根据之前的内容进行决策可以产生哪一条. 如下产生式,

$$S \to aBC$$
$$S \to aSBC$$
$$CB \to CZ$$
$$CZ \to WZ$$
$$WZ \to WC$$
$$WC \to BC$$
$$aB \to ab$$
$$bB \to bb$$
$$bC \to bc$$
$$cC \to cc$$

$aB$ 和 $CB$ 两种情况对于 $B$ 的处置是不一样的. 这就使得一个非终结符展开成什么样子取决于它的上下文.

**1.2. 上下文无关文法的语义.** 类似于正则表达式, 上下文无关文法 $G$ 定义了一个语言 $L(G)$. 这个语言的 “串” 是按照规则把左边替换做右边 (推导) 实现的. 每一步推导需要选择替换哪个非终结符号, 以及使用哪个产生式. 例如上面的表达式可以经过如下推导为 $-(id + id)$.

$$E \Longrightarrow -E \Longrightarrow -(E) \Longrightarrow -(E + E) \Longrightarrow -(\mathbf{id} + E) \Longrightarrow -(\mathbf{id} + \mathbf{id})$$

为了方便起见, 引入一些记号:

- $E \Longrightarrow -E$：经过一步推导得出
- $E \overset{+}{\Rightarrow} (\mathbf{id} + E)$：经过一步或多步推导得出
- $E \overset{*}{\Rightarrow} -(\mathbf{id} + E)$：经过零步或多步推导得出

同样可以从最右边的符号开始推导, 如下.

$E \Longrightarrow -E \Longrightarrow -(E) \Longrightarrow |-(E+E)| \Longrightarrow -(E+\mathbf{id}) \Longrightarrow -(\mathbf{id}+\mathbf{id})$

如果我们在推导的每一步中都选择最左边的非终结符, 称为**最左推导 (Leftmost derivation)**, 如果在推导的每一步中都选择最右边边的非终结符, **最右推导 (Rightmost derivation)**.

我们的推导只有在推导到全是终结符的时候才认为结束. 如果可以由开始符号推导为字符串 $\alpha$(可能含有非终结符), 我们就说 $\alpha$ 是文法 $G$ 的一个句型.

**定义 1.2** (**句型 (Sentential Form)**)**.** 如果 $S \overset{*}{\Rightarrow} \alpha$, 且 $\alpha \in (T \cup N)^*$, 则称 $\alpha$ 是文法 $G$ 的一个句型

如果我们每个都推到头了, 我们称为句子.

**定义 1.3** (**句子 (Sentence)**)**.** 如果 $S \overset{*}{\Rightarrow} w$, 且 $w \in T^*$, 则称 $w$ 是文法 $G$ 的一个句子.

有了句子, 就可以定义文法的语言. 即所有句子的集合.

**定义 1.4** (文法 $G$ 生成的语言 $L(G)$)**.** 文法 $G$ 的语言 $L(G)$ 是它能推导出的所有句子构成的集合

$$L(G) = \{w \mid S \overset{*}{\Rightarrow} w\}$$

关于文法 $G$ 的两个基本问题:

1. Membership 问题: 给定字符串 $x \in T^*, x \in L(G)$ ?
2. $L(G)$ 究竟是什么?

对于 Membership 问题, 就是用来构建语法分析器的: 为输人的词法单元流寻找推导、构建语法分析树, 或者报错. 对于 $L(G)$ 是什么, 这是程序设计语言设计者需要考虑的问题.

**例子 1.3.**

$$S \to SS$$
$$S \to (S)$$
$$S \to \epsilon$$

表达的语言 $L(G) = \{$所有匹配的括号$\}$.

$$S \to aSb$$
$$S \to \epsilon$$

表达的语言 $L(G) = \{aaaa \ldots abbb \ldots b\}$, 并且 $a$ 的数量和 $b$ 的数量相等.

**例子 1.4.** 1) 希望产生字母表 $\Sigma = \{a, b\}$ 上的所有回文串 (Palindrome) 构成的语言. 我们如何设计语言?

考虑递归地求解问题. 考虑基础情况和推导的情况.

$$S \to aSa$$
$$S \to bSb$$
$$S \to a$$
$$S \to b$$
$$S \to \epsilon$$

2) 我们希望产生形如 $\{b^n a^m b^{2n} \mid n \geq 0, m \geq 0\}$ 的语言, 应该如何构造?

实际上就是上面的翻版. $b$ 的关系只要产生形如 $S \to bSbb$; 而 $a$ 的关系可以这样处理:

$$S \to bSbb \mid A$$
$$A \to aA \mid \epsilon$$

3) 产生形如 $\{x \in \{a,b\}^* \mid x$ 中 $a,b$ 个数相同 $\}$ 的语言.

尝试 1. $V \to aVb|bVa|\epsilon$, 错误: 只生成前面一半 $a,b$ 相同, 后面一半 $a,b$ 相同的. $aabbba$ 并不能被生成.

尝试 2. $V \to aVb|bVa|VV|\epsilon$.

尝试 3. $V \to aVbV|bVaV|\epsilon$

正确性说明: 每一次将串分为 4 部分, 第四部分肯定能找出一个字符串分割出来的部分和 2, 3 部分的 $a,b$ 个数相同. (可以用 $a = +1, b = -1$ 的折线图解释).

4) 产生形如 $\{x \in \{a,b\}^* \mid x$ 中 $a,b$ 个数不同 $\}$ 的语言. 答案为

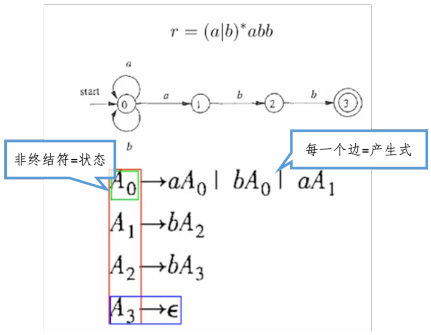
$$S \to T \mid U$$
$$T \to VaT \mid VaV$$
$$U \to VbU \mid VbV$$
$$V \to aVbV|bVaV|\epsilon$$

为什么不使用优雅、强大的正则表达式描述程序设计语言的语法? 因为正则表达式的表达能力严格弱于上下文无关文法.

实际上, 上面的上下文有关文法描述的语言是 $L = \{a^n b^n c^n\}$, 而这是难以被上下文无关文法表达的.

**1.3. 语言的表达能力简介.** 我们来证明正则表达式的表达能力严格弱于上下文无关文法. 首先, 任意一个 RE, 都可以被 CFG 表示; 并且存在一个 CFG 的文法, 它无法被 RE 表示.

首先, 每个正则表达式 $r$ 对应的语言 $L(r)$ 都可以使用上下文无关文法来描述. 对于一个 RE, 可以写出对应的 NFA, 这样就可以写出它的 CFG.




此外, 若 $\delta(A_i, \epsilon) = A_j$, 则添加 $A_i \to A_j$. 这说明 RE 不强于 CFG.

另外, 我们发现语言

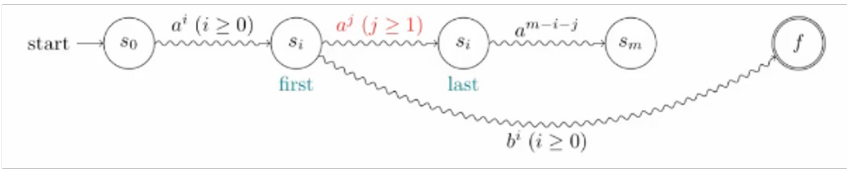
$$L = \{a^n b^n \mid n \geq 0\}$$

无法使用正则表达式来描述.

**定理 1.1.** $L = \{a^n b^n \mid n \geq 0\}$ 无法使用正则表达式描述.

*Proof.* 考虑反证法: 假设存在一个正则表达式 $r : L(r) = L = \{a^n b^n \mid n \geq 0\}$, 则存在有限状态自动机 $D(r) : L(D(r)) = L$; 设其状态数为 $k \geq 1$. 在假想的自动机上面输入 $a^m (m \geq k)$, 这个自动机经历了 $m+1$ 个状态. 这表明经过的状态中有两个是同一个状态.

取这个相同状态 $s_i$ 经过的最早一次和最后一次, 有下图上侧的数量关系.




现在输入 $a^i b^i$, 那么一定到达了一个接收状态. 如上图下侧所示. 如果输入了 $a^{i+j} b^i$, 这个自动机也会接受这个状态, 矛盾! □

这表明 DFA 会被大输入 "撑爆". 即, 经过足够大的输入, DFA 总是会回到某个先前经历过的状态, 这就意味着我无论在这样的状态上面绕多少圈, 再让它抵达终点也是可行的. 这就是著名的定理: Pumping Lemma. Pumping 的意思是 Repeat.

**定理 1.2.** If $L$ is a regular language, then there exists a number $p \geq 1$ (pumping length) such that any string $s$ in $L$ of length $\geq p$ can be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. $|y| \geq 1$
2. $|xy| \leq p$
3. $\forall i \geq 0 : xy^i z \in L$

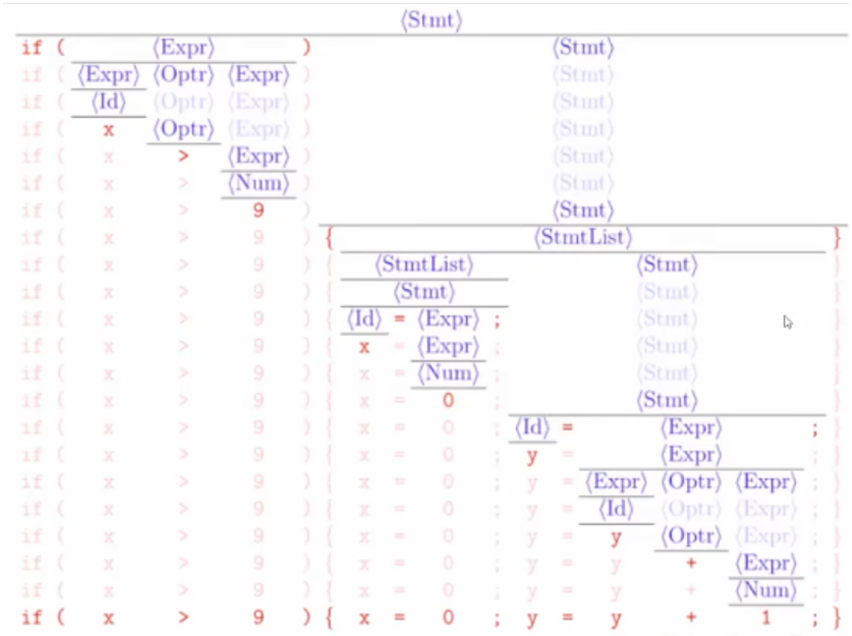
**例子 1.5.** $D = \left\{1^{n^2} \mid n \geq 0\right\}$ is not regular.

考虑 $s = 1^{p^2} \in D$, ( $p$ is the pumping length), 满足 $|s| \geq p$. 于是 $s = xyz \in D$. 但是, $xy^2z \notin D$. 因为 $p^2 < |xy^2z| = |xyz| + |y| \leq p^2 + p < p^2 + 2p + 1 = (p+1)^2$. 因此 $xy^2z$ 介于完全平方数之间, 故一定不属于 $D$ 中.

那么, 上下文无关语言的表达能力在哪里? 实际上, 只要我们的字符串足够的长, 前后一定能够经过两个一样的非终结符. 这就是上下文无关语言的 Pumping Lemma. 有了这两个相同的非终结符, 就可以进行构造了.

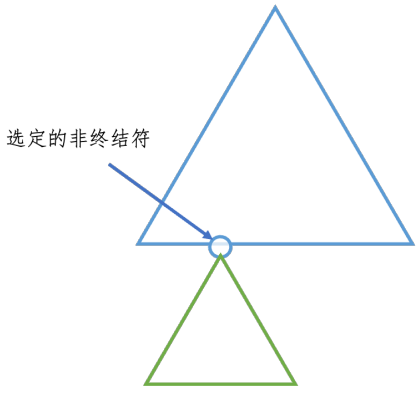
## §2 递归下降的 $LL(1)$ 语法分析器

**2.1. 简介.** 我们接下来试图构造语法分析树. 例如下图所示. 如果构建成功, 在给出树的时候顺便给出了推导; 如果不成功, 那么就报错.




有两种方法: 自顶向下 vs. 自底向上. 现在只考虑无二义性的文法. 这意味着, 每个句子对应唯一的一棵语法分析树. 接下来我们要做的事情是构造一个**自顶向下的, 递归下降的, 基于预测分析表的, 使用与 $LL(1)$ 文法的** $LL(1)$ 语法分析器.

**a) 自顶向下构建语法分析树** 根节点是文法的起始符号 $S$, 叶节点是词法单元流 $w\$$, 仅包含终结符号与特殊的文件结束符 \$ (EOF).

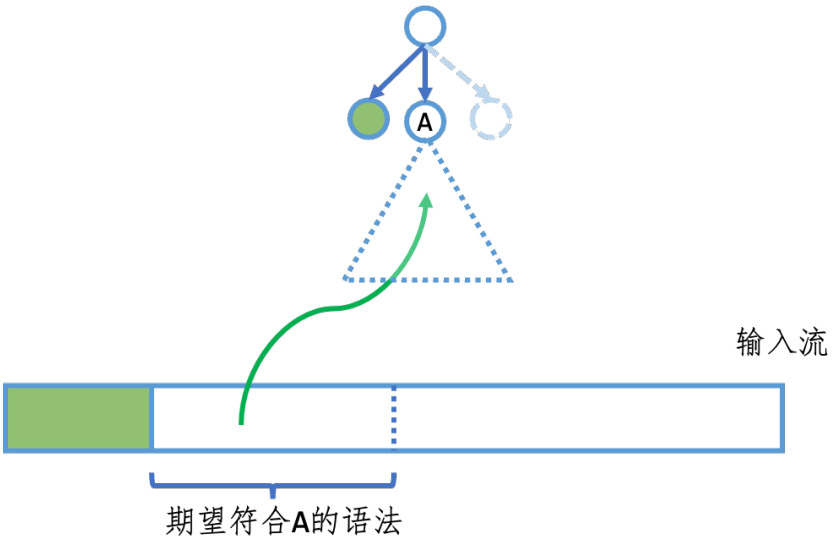



既然要 Fiona 每个中间节点表示对某个非终结符应用某个产生式进行推导, 重点在于选择哪个非终结符, 以及选择哪个产生式.

对于 $LL(1)$ 算法, 在推导的每一步, 总是选择最左边的非终结符进行展开(也就是最左推导, 第二个 $L$ 的意思). 第一个 $L$ 的意思是从左向右读入词法单元.

**b) 递归下降的** 为每个非终结符写一个递归函数, 内部按需调用其它非终结符对应的递归函数, 下降一层.

```
void A(){
  for (i = 1 to k){    先不必纠结如何选取, 这是 LL(0) 的核心.
  选择一个 A 的产生式, A → X1X2···Xk.
    if Xi 是一个非终结符号    递归下降调用其他非终结符对应的函数, 为输入流递归地匹配.
      调用过程 Xi();成功匹配了一个词法单元
    else if Xi 等于当前输入的符号 a
      读入下一个输入符号;
    else /* 出现了不期望的词法单元, 错误. */
    }
}
```

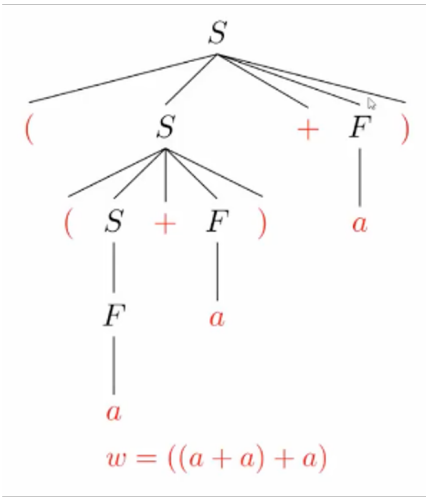



**例子 2.1.** 考虑如下表达式

$$S \to F$$
$$S \to (S + F)$$
$$F \to a$$

以及 $w = ((a + a) + a)$.

匹配过程为




**2.2. 基于预测分析表的.** 对于上面的例子, 同样是展开非终结符 $S$, 为什么前两次选择了 $S \to (S + F)$, 而第三次选择了 $S \to F$ ? 这就是预测分析表要做的事情.

如上例, 第一个选择第二条规则, 是因为只有这一个会得到和输入匹配的字符串. 和上面的不同, 这时候我们可以选择 $S$, 将来有机会变为 $a$. 根本原因是: 因为它们面对的当前词法单元不同.

假设我们有一张表, 指明了每个非终结符在面对不同的润法单元或文件结束符时, 该选择哪个产生式 (按编号进行索引) 或者报错 (空单元格). 如下表:

| | ( | ) | $a$ | + | \$ |
|---|---|---|---|---|---|
| $S$ | 2 | | 1 | | |
| $F$ | | | 3 | | |

这张表叫做**预测分析表 (prediction table)**.

**定义 2.1** ($LL(1)$ 文法)**.** 如果文法 $G$ 的预测分析表是无冲突的 (每个单元格里只有一个产生式 (编号)), 则 $G$ 是 $LL(1)$ 文法.

对于这样的内容, 我们可以对于当前选择的非终结符, 仅根据输人中当前的词法单元 ($LL(1)$) 即可确定需要使用哪条产生式.

**例子 2.2.** 对于下面的文法, 其不是 $LL(1)$ 文法

$$S \to abC \mid abC$$
$$C \to c \mid d$$

因为只看当前的词法单元, 都不行. 至少要往后看 2 个, 因此是 $LL(3)$ 文法.

有一些文法无论 $k$ 有多大, 它都不可能是 $LL(k)$ 文法. 比如我们以前见到的`'if' expr 'then' stat` 语句. 要分析他们, 需要往前面看任意多个词法单元. 下次会介绍如何处理这个.

那么这样一来就可以写写递归下降的算法了. 如上面的例子为例, 就有这样的代码:

```
1: procedure MATCH(t)                                  ▷ t 是预期的词法单元
2:     if token = t then                               ▷ token 是输入中当前的词法单元
3:         token ← NEXT-TOKEN()
4:     else
5:         ERROR(token, t)
```

```
1: procedure F()
2:     if token = 'a' then
3:         MATCH('a')
4:     else
5:         ERROR(token, {'a'})
```

以及

```
1: procedure S()
2:     if token = '(' then
3:         MATCH('(')
4:         S()
5:         MATCH('+')
6:         F()
7:         MATCH(')')
8:     else if token = 'a' then
9:         F()
10:    else
11:        ERROR(token, {'(', 'a'})                    ▷ 好处是可以把期望的内容报错出来
```

下面介绍如何构造这个预测分析表. 先看下面的例子.

```
prog : func_call | decl EOF;

func_call : ID '(' arg ')' ;
decl : 'int' ID optional_init ';' ;

arg : 'int' ID optional_init ;
optional_init
    : '=' ID  # Init
    |         # NoInit
    ;
```

```
   int x = y;     int x;
f(int x = y)     f(int x)
```

区分 `int x = y;` 和 `f(int x)` 在读到 `int` 和读到 `f` 的时候选择不同的东西展开的区别在于它可否通过后续的展开出现对应的期望的词法单元. 也就是在他们所有可能的展开中最左端的终结符中(这样的集合叫做 First 集合)有没有我们想要的单元.

继续往下读, 读到 `decl` 中的 `optional_init` 规则的时候, `int x=y` 选择了`'=' ID` 这条规则, 而 `f(int x)`中选择了 $\epsilon$ 规则. 这是这个算法的核心. 当然是因为当前我们遇到的字符 (红色标记) 不同. 后者看到的是右括号, 如果我选择空的话, 后面的右括号就是我要匹配的那一个–这和当前输入匹配. 这就说明, 我们要考察 `arg` 后面有可能出现那些终结符. 这个可能出现的集合会被称为 Follow 集合.

总结一下, First($\alpha$) 是可从 $\alpha$ 推导得到的句型的首终结符号的集合,

**定义 2.2** (First 集合)**.** 对于任意的 (产生式的右部) $\alpha \in (N \cup T)^*$ :

$$\text{First}(\alpha) = \{t \in T \cup \{\epsilon\} \mid \alpha \overset{*}{\Rightarrow} t\beta \vee \alpha \overset{*}{\Rightarrow} \epsilon\}.$$

考虑非终结符 $A$ 的所有产生式 $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_m$, 如果它们对应的 First $(\alpha_i)$ 集合互不相交, 则只需查看当前输入词法单元, 即可确定选择哪个产生式 (或报错).

Follow $(A)$ 是可能在某些句型中紧跟在 $A$ 右边的终结符的集合.

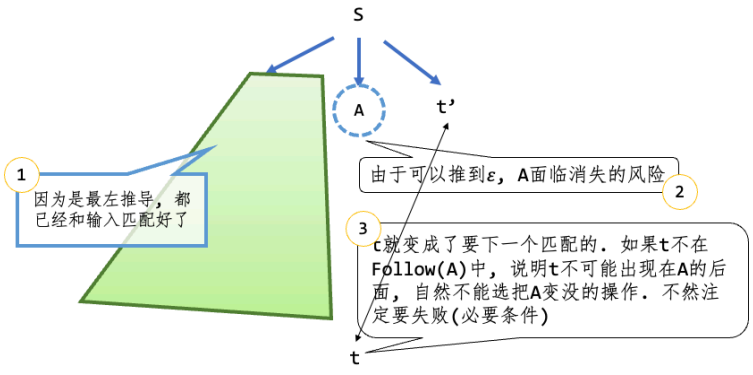
**定义 2.3** (Follow($A$) 集合)**.** 对于任意的 (产生式的左部) 非终结符 $A \in N$ :

要收集的紧跟在 $A$ 后面的终结符

$$\text{Follow}(A) = \{t \in T \cup \{S\} \mid \exists s.S \overset{*}{\Rightarrow} s \triangleq \beta A\, t\, \gamma\}.$$

Follow 的含义是: 考虑产生式 $A \to \alpha$, 如果从 $\alpha$ 可能推导出空串 ($\alpha \overset{*}{\Rightarrow} c$), 则只有当当前词法单元 $t \in \text{Follow}(A)$, 才可以选择该产生式.

另一个直观解释, 就是当前假设有一个预测分析表, 我们期望往这个非终结符 $A$ 所在的那一行进行规则填写. 也就是碰到了某一个终结符之后就应该选某条规则. 假设现在我们有 3 个终结符 $t_1, t_2, t_3$. 那么对于 $A \to \alpha \overset{*}{\Rightarrow} \epsilon$ 这一个特殊的产生式, 问题在于 $t_1, t_2, t_3$ 这里面满足哪一个特殊的条件才可以让我选这条产生式? 这个条件就是 $t \in \text{Follow}(A)$. 如下图所示.




接下来就比较简单了. 先算每个符号 $X$ 的 First($X$) 集合:

```
1: procedure FIRST(X)
2:     if X ∈ T then                                  ▷ 规则 1: X 是终结符
3:         First(X) = X
4:     for X → Y_1 Y_2 ... Y_k do                     ▷ 规则 2: X 是非终结符
5:         First(X) ← First(X) ∪ {First(Y_1) \ {ϵ}}
6:         for i ← 2 to k do
7:             if ϵ ∈ L(Y_1 ... Y_{i-1}) then
8:                 First(X) ← First(X) ∪ {First(Y_i) \ {ϵ}}
9:         if ϵ ∈ L(Y_1 ... Y_k) then                 ▷ 规则 3: X 可推导出空串
10:            First(X) ← First(X) ∪ {ϵ}
```

不断应用上面的规则, 直到每个 First$(X)$ 都不再变化. 再计算每个符号串 $\alpha$ 的 FIRST$(\alpha)$ 集合. 对于 $\alpha = X\beta$

$$\text{FIRST}(\alpha) = \begin{cases} \text{FIRST}(X) & \epsilon \notin L(X) \\ (\text{First}(X) \backslash \{\epsilon\}) \cup \text{FIRSt}(\beta) & \epsilon \in L(X) \end{cases}$$

最后注意: 如果 $\epsilon \in L(\alpha)$, 则 $\epsilon \in \text{FIRST}(\alpha)$

**例子 2.3.** 求如下的文法的 First 集合和 Follow 集合.

$$\begin{aligned} X &\to Y \\ X &\to a \\ Y &\to \epsilon \\ Y &\to c \\ Z &\to d \\ Z &\to XYZ \end{aligned}$$

首先看 $\text{First}(Y) = \{c, \epsilon\}$, (能写 $\epsilon$ 是因为没有其他的了). 于是对于 $\text{First}(X) = \{a, c, \epsilon\}$. $F(Z) = \{d, a, c, ??\}$, $\text{First}(XYZ) = \{a, c, d, ??\}$ 再次应用, 发现他们都可以得到一个不动点. 问题在于, $\epsilon$ 在不在里面. 由于 $Z \to \epsilon\epsilon Z, Z \to d$, 因此 $Z$ 不能变为 $\epsilon$. 我们的迭代就结束了.

接下来看 Follow 集合.

```
1: procedure FOLLOW(X)
2:     for X 是开始符号 do                          ▷ 规则 1: X 是开始符号
3:         Follow(X) ← Follow(X) ∪ {$}
4:     for A → αX do                                ▷ 规则 2: X 是某产生式右部的最后一个符号
5:         Follow(X) ← Follow(X) ∪ Follow(A)
6:     for A → αXβ do                               ▷ 规则 3: X 是某产生式右部中间的一个符号
7:         Follow(X) ← Follow(X) ∪ (First(β) \ {ϵ})
8:         if ϵ ∈ First(β) then
9:             Follow(X) ← Follow(X) ∪ Follow(A)
```

不断应用上面的规则, 直到每个 FOLLOW$(X)$ 都不再变化.

**例子 2.4.** 接上例. 首先知道 Follow$(X) = \{\$, ...\}$, 由于 Follow$(Y)\backslash\{\epsilon\} \subset$ Follow$(X)$, Follow$(X) = \{\$, c, ...\}$. 由于 First$(YZ) = \{a, c, d\}$, 因此 Follow$(X) = \{\$, c, a, d\}$.

然后 Follow$(Y) = \{\$, c, a, d, ...\}$, 由于 First$(Z) = \{a, c, d\}$, 因为没有 $\epsilon$, 所以结束了.Follow$(Y) = \{\$, c, a, d\}$.

最后计算 Follow$(Z) \subset$ Follow$(Z)$, 我们要不断迭代直到它不再变化为止. 如果我们从空集开始构造, 迭代一次就不变化了, 说明是空集. 这表明从起始符号展开根本不会到达 $Z$.

然后基于 FIRST 与 FOLLOW 集合计算给定文法 $G$ 的预测分析表. 对应每条产生式 $A \to \alpha$ 与终结符 $t$, 如果

$$t \in \text{FIRST}(\alpha)$$
$$\alpha \overset{*}{\Rightarrow} \epsilon \wedge t \in \text{FOLLOW}(A)$$

则在表格 $[A, t]$ 中填入 $A \to \alpha$ (编号)

上述的分析我们都是用充分性来做的. 换句话说就是当下的选择未必正确, 但此刻“你别无选择”.

**例子 2.5.** 接上例, 填表的结果是

| | $a$ | $c$ | $d$ | \$ |
|---|---|---|---|---|
| $X$ | 1,2 | 1 | 1 | 1 |
| $Y$ | 3 | 3,4 | 3 | 3 |
| $Z$ | 6 | 6 | 5,6 | |

总结一下, $LL(1)$ 指的是:

- $L$ : 从左向右 (left-to-right) 扫描输入
- $L$ : 构建最左 (leftmost) 推导
- 1: 只需向前看一个输人符号便可确定使用哪条产生式