§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.

$$\overset{\text{头部 (head)}}{\boxed{\texttt{functionDecl}}} : \overset{\text{身体 (body)}}{\boxed{\texttt{type ID '(' formalParameters? ')' block}}} ;$$

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 ϵ. 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)** α : 终结符与非终结符构成的串, 也可以是空串 ϵ

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

$$E \to \overset{\text{相加}}{\boxed{E+E}} \mid \overset{\text{相乘}}{\boxed{E*E}} \mid \overset{\text{加括号}}{\boxed{(E)}} \mid \overset{\text{变为负号}}{\boxed{-E}} \mid \overset{\text{最小单位}}{\boxed{\text{id}}}$$

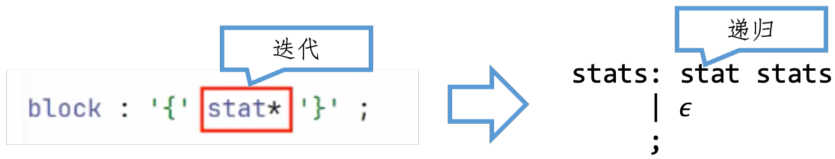
实际上他们是下面的简写 (用右递归的形式写出):

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

另外一个上下文无关文法是

$$\begin{aligned}
&S \to aSa \\
&S \to bSb \\
&S \to a \\
&S \to b \\
&S \to \epsilon
\end{aligned}$$

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,

$$
\begin{aligned}
S &\to aBC \\
S &\to aSBC \\
CB &\to CZ \\
CZ &\to WZ \\
WZ &\to WC \\
WC &\to BC \\
aB &\to ab \\
bB &\to bb \\
bC &\to bc \\
cC &\to cc
\end{aligned}
$$

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 $-(id+id)$.

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

为了方便起见, 引入一些记号:

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

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

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 α(可能含有非终结符), 我们就说 α 是文法 G 的一个句型.

定义 1.2 (**句型 (Sentential Form)**)**.** 如果 $S \stackrel{*}{\Rightarrow} \alpha$, 且 $\alpha \in (T \cup N)^*$, 则称 α 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 $S \stackrel{*}{\Rightarrow} w$, 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

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

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 $L(G) = \{aaaa \dots abbb \dots b\}$, 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

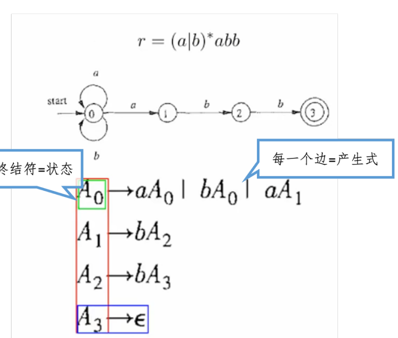
$$S \to T \mid U$$
$$T \to VaT \mid VaV$$
$$U \to VbU \mid VbV$$
$$V \to aVbV|bVaV|\epsilon$$

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

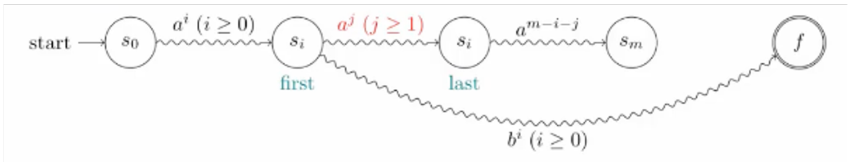
$$L = \{a^n b^n \mid n \geq 0\}$$

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. $\forall i \geq 0 : xy^i z \in L$

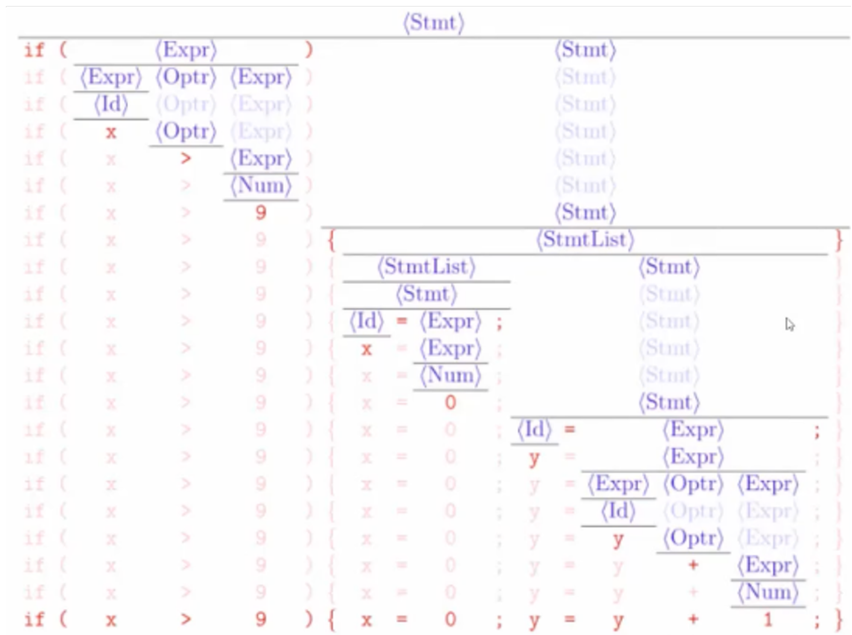
例子 1.5. $D = \left\{1^{n^2} \mid n \geq 0\right\}$ is not regular.

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, $xy^2 z \notin D$. 因为 $p^2 < |xy^2 z| = |xyz| + |y| \leq p^2 + p < p^2 + 2p + 1 = (p+1)^2$. 因此 $xy^2 z$ 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.

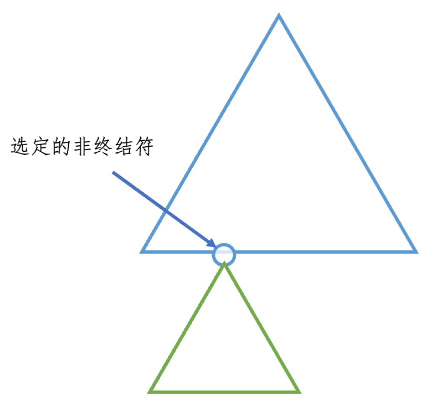
§2 递归下降的 $LL(1)$ 语法分析器

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, $A \to X_1X_2\cdots X_k$.

    if X_i 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 $X_i()$;   成功匹配了一个词法单元

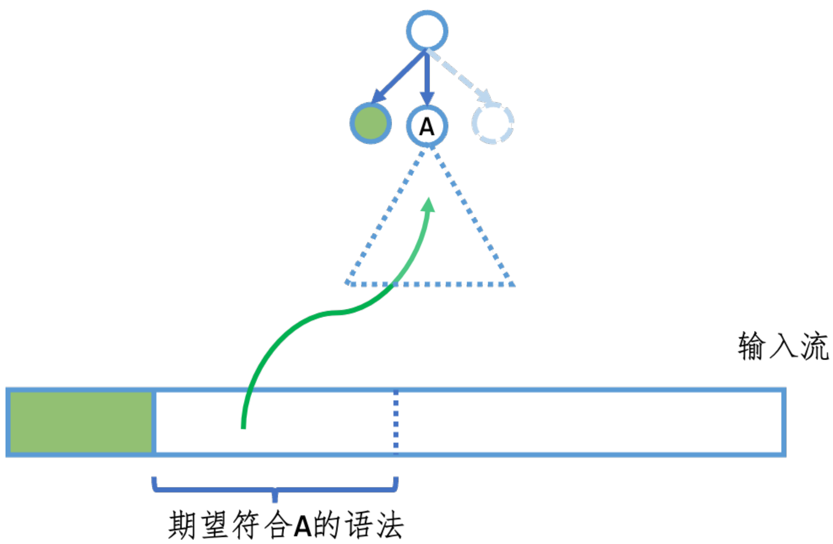
    else if X_i 等于当前输入的符号 a

      读入下一个输入符号;

    else /* 出现了不期望的词法单元, 错误. */

    $\}$

$\}$

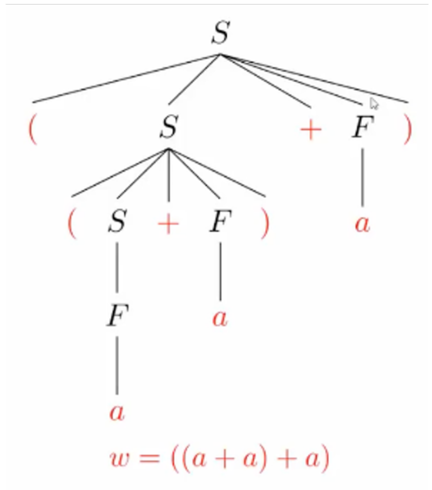


例子 2.1. 考虑如下表达式

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

以及 $w=((a+a)+a)$.

匹配过程为


S
(
S
+
F
)
(
S
+
F
)
a
F
a
a
w = ((a + a) + a)