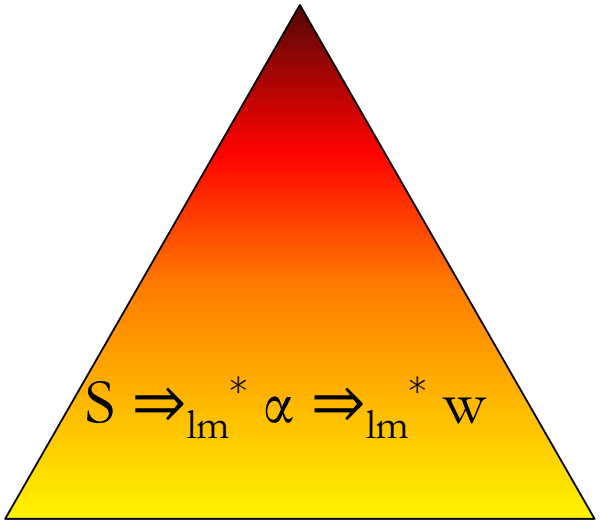
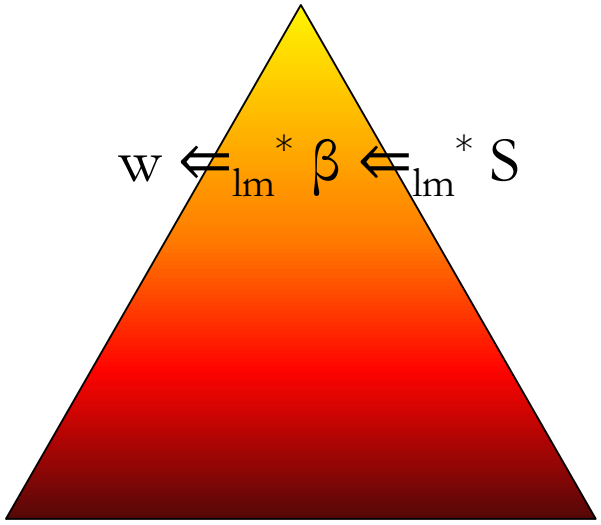
第六章 自上而下语法分析

— 2024

语法分析概貌

- 语法分析本质上是按照文法识别输入符号串w是否为一个有确定结构的句子，即是否是一个合法的语法单位的过程：
  - 寻找到一个最左推导$S \Rightarrow_{lm}^{*} \alpha \Rightarrow_{lm}^{*} w$；
  - 从S为根开始自上而下地扩展出一颗产物为w的语法树；
  - 寻找到一个最左归约$w \Leftarrow_{lm}^{*} \beta \Leftarrow_{lm}^{*} S$；
  - 以w为产物，自下而上地构建出一颗根为S的语法树。





自上而下的语法分析

- 概述：输入串w是整个源程序的记号串，它来自于词法分析的输出。
  - 寻找到一个最左推导$S \Rightarrow_{lm}^{*} \alpha \Rightarrow_{lm}^{*} w$；或者，
  - 以S为根，从根开始自上而下地为w扩展出一颗语法树。
- 语法分析中逐次读入输入符号（可以是调用词法分析器返回单词），然后选择合适的产生式或者用于形成一次直接推导，或者用于扩展当前语法树的一个叶子结点。
- 这个过程一直进行到输入串w被读完（消耗完），同时w的推导完成成功，或者w的语法树扩展成功，则说w是合法句子，语法分析结束。
- 否则报出语法错误，而不是简单的以拒绝二字罢休。

- 思考：如何寻找到一个最左推导$S \Rightarrow_{lm}^{*} \alpha \Rightarrow_{lm}^{*} w$？

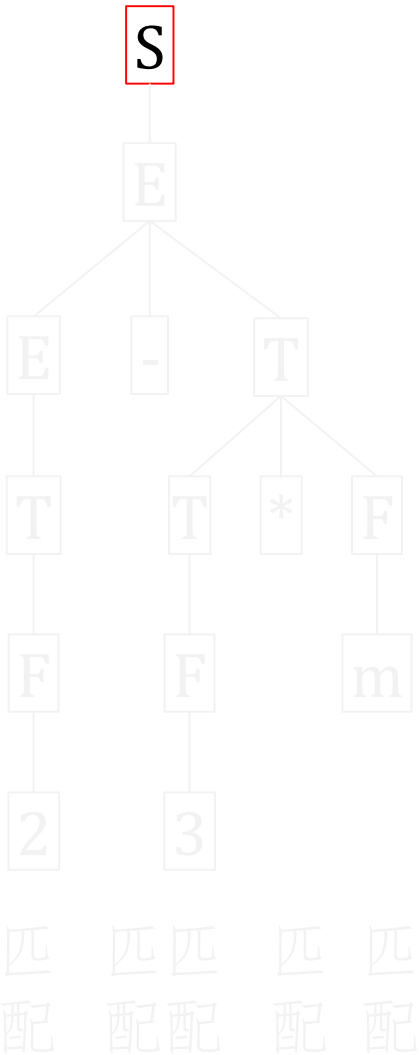
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

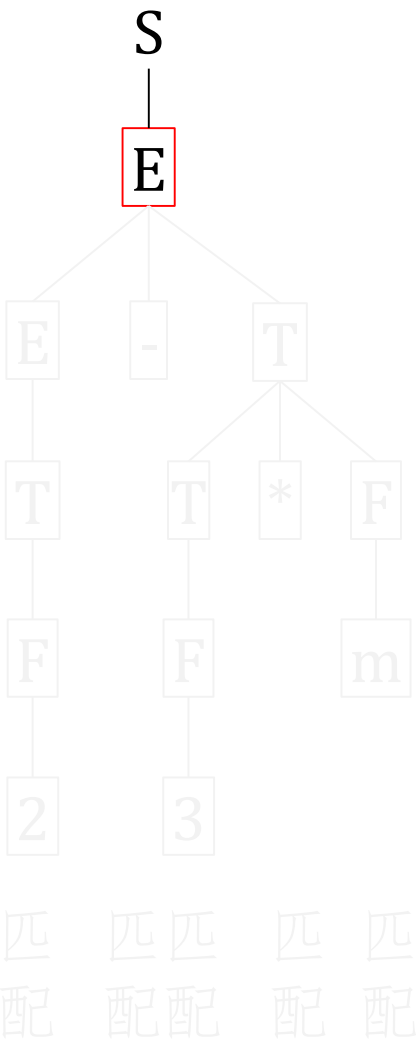
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

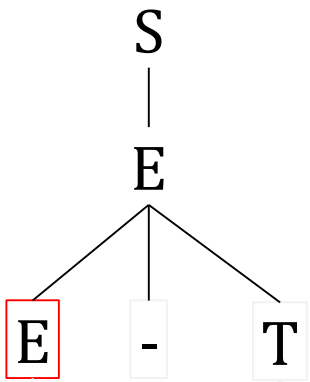
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

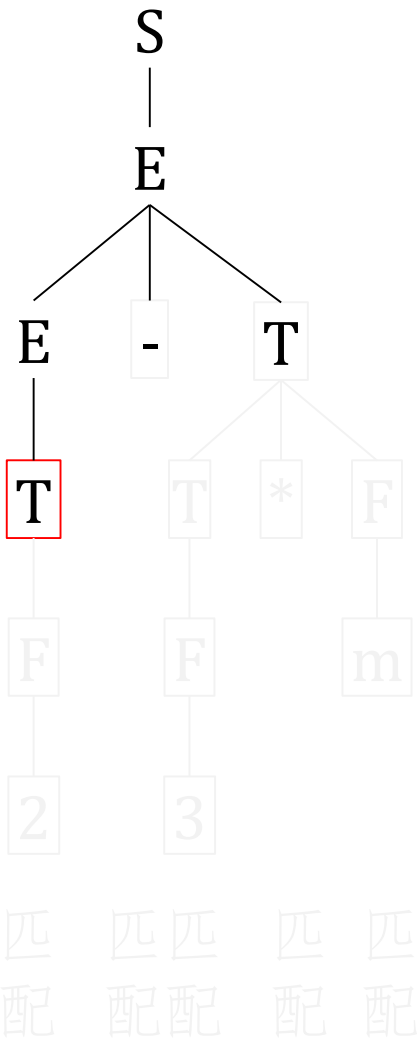
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

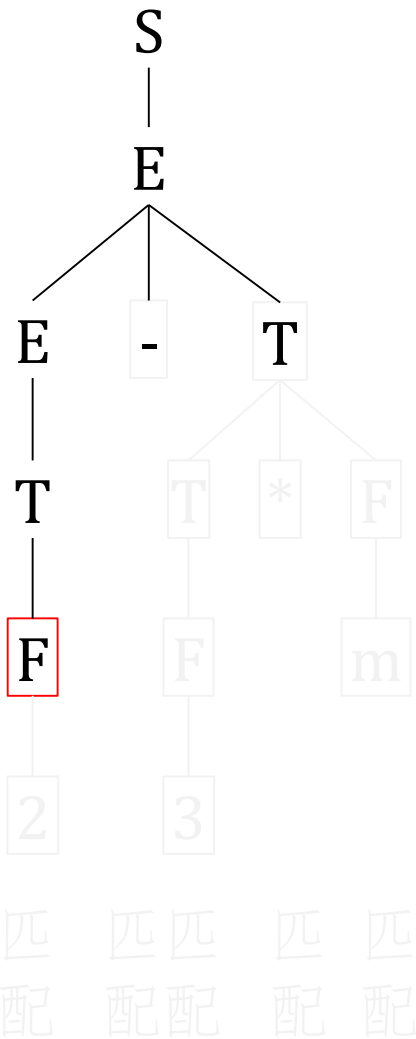
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

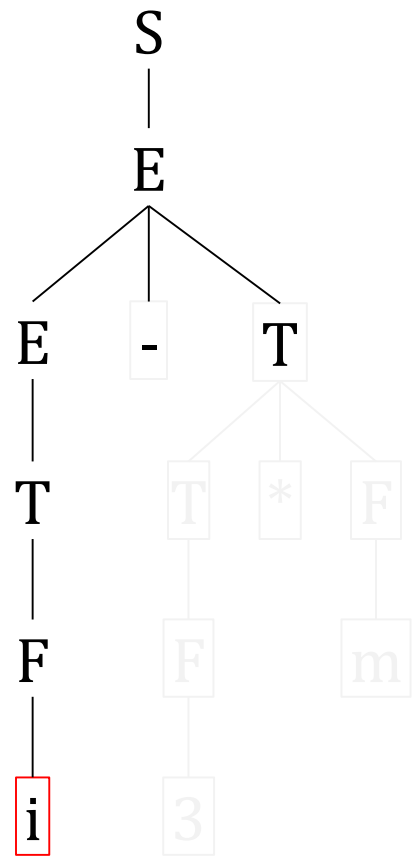
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

匹配 匹配 匹配 匹配 匹配

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

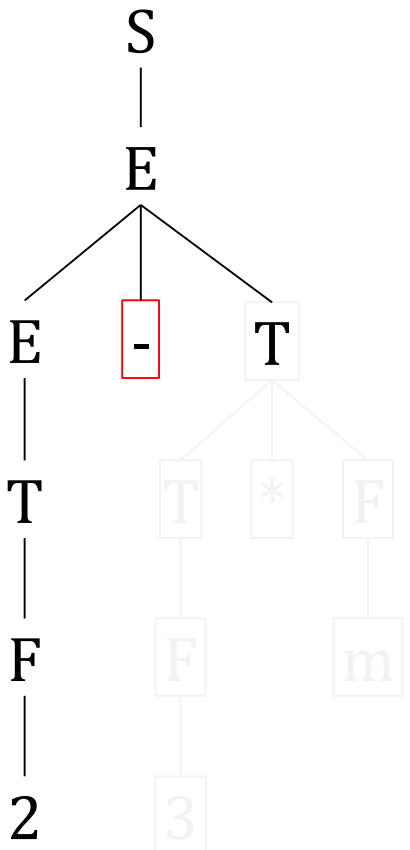
The Parsing Process

剩余输入串：

-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

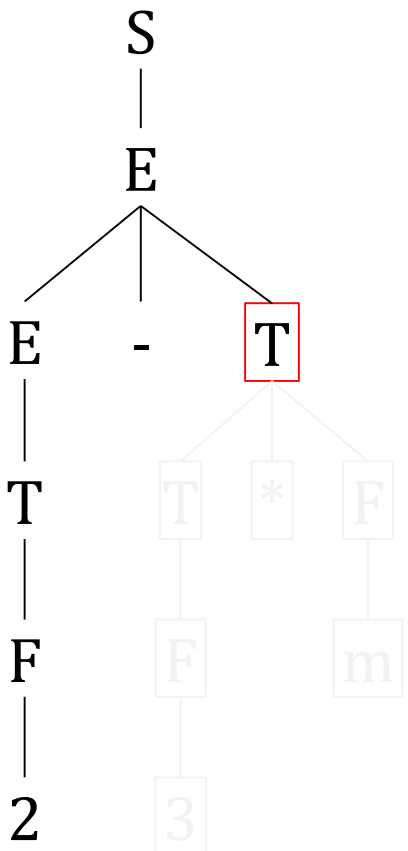
The Parsing Process

剩余输入串：

3*m

^

当前输入符号



匹配

$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

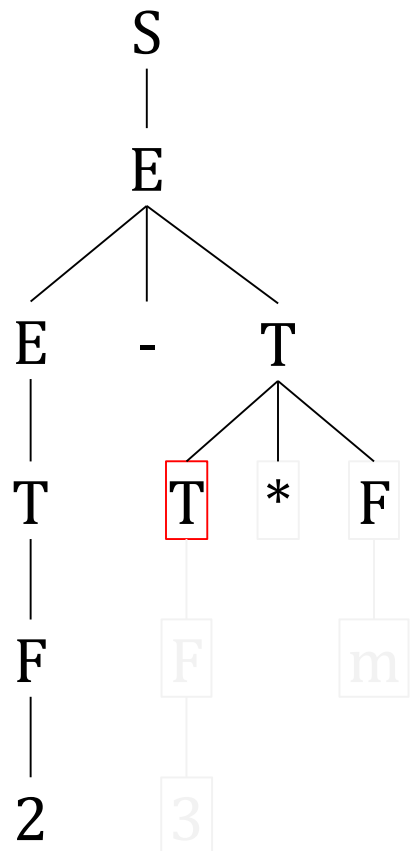
The Parsing Process

剩余输入串：

3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

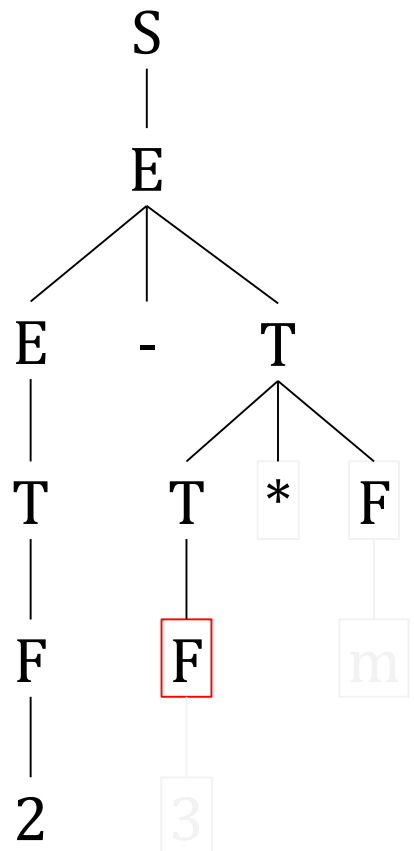
The Parsing Process

剩余输入串：

3*m

^

当前输入符号



匹配 匹配 匹配

$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

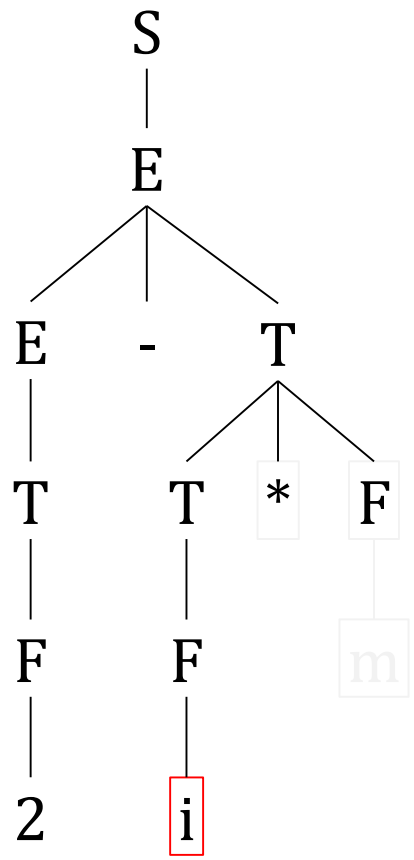
The Parsing Process

剩余输入串：

3*m

^

当前输入符号



匹配 匹配 匹配 匹配

$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

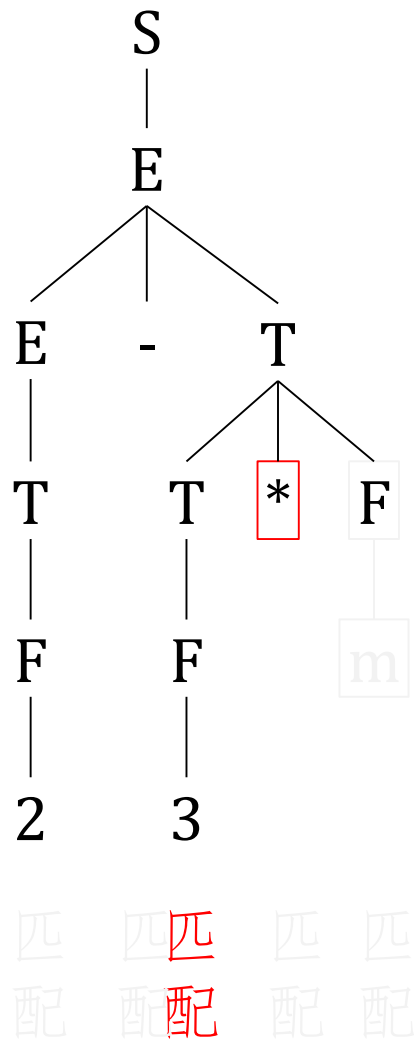
The Parsing Process

剩余输入串：

*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

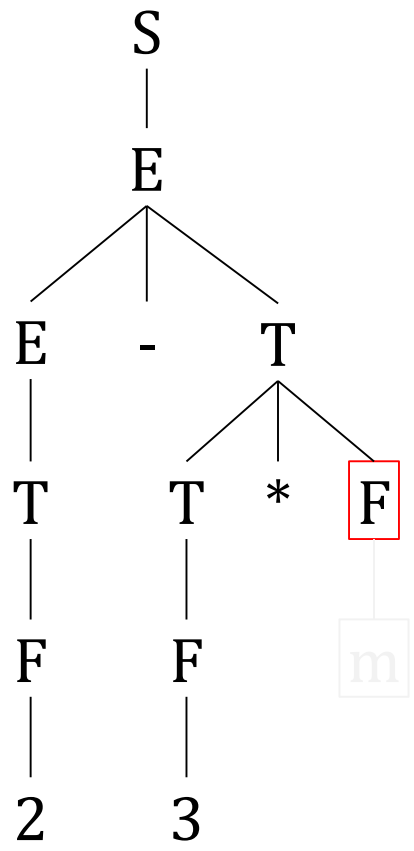
The Parsing Process

剩余输入串：

m

^

当前输入符号



匹配

匹配

$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

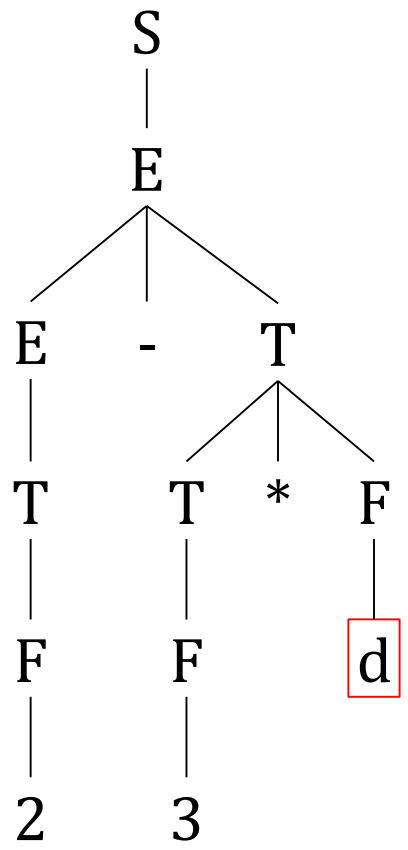
The Parsing Process

剩余输入串：

m

^

当前输入符号



匹配 匹配 匹配 匹配

$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

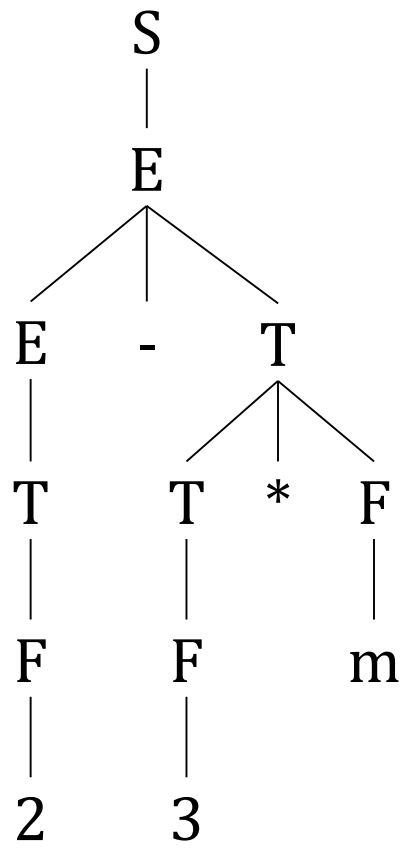
F →d

The Parsing Process

剩余输入串：

^

当前输入符号



匹 匹匹 匹 匹

配 配配 配 配

$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

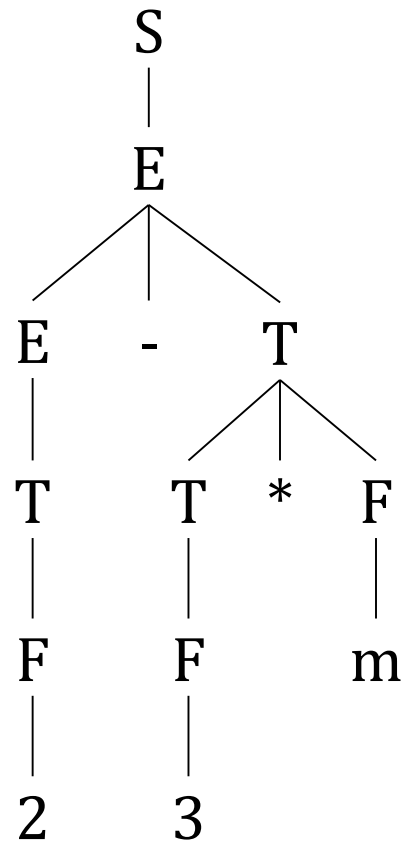
The Parsing Process

剩余输入串：

^

当前输入符号

成功！



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

The Parsing Process

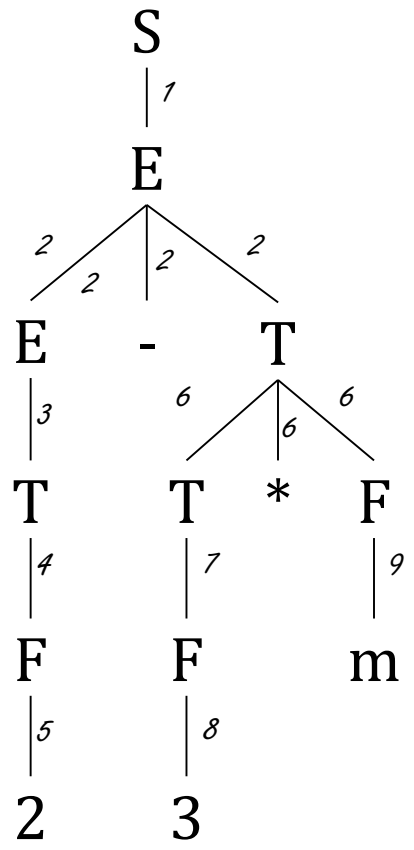
剩余输入串：

^

当前输入符号

成功！

1



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

观察

- 分析过程对应于最左推导。
- 分析过程对应于扩展语法树，并始终知道当前唯一待扩展结点。
- 分析过程有三类基本步骤（或称动作或行为）：
  - 当待扩展结点为变元时，应用产生式，扩展出该结节点的孩子结点，依次为它的候选式中文法符号所示；
  - 当待扩展结点为终结符时，匹配掉当前输入符号，除非匹配失败；匹配失败时分析过程由错误处理接管从而表明分析失败；
  - 当剩余串为空时分析成功。
- 事后看到，分析过程是先根遍历有序树（语法树）的过程。

6.1 自上而下分析面临的问题

- 在最左推导策略中，仍然存在不确定性，即多个候选式的任意选择其一。

- 候选式的选择的不确定性导致分析过程可能出现两个问题。
  - 分析过程中的“死循环”问题；
  - 分析过程中的回溯问题。

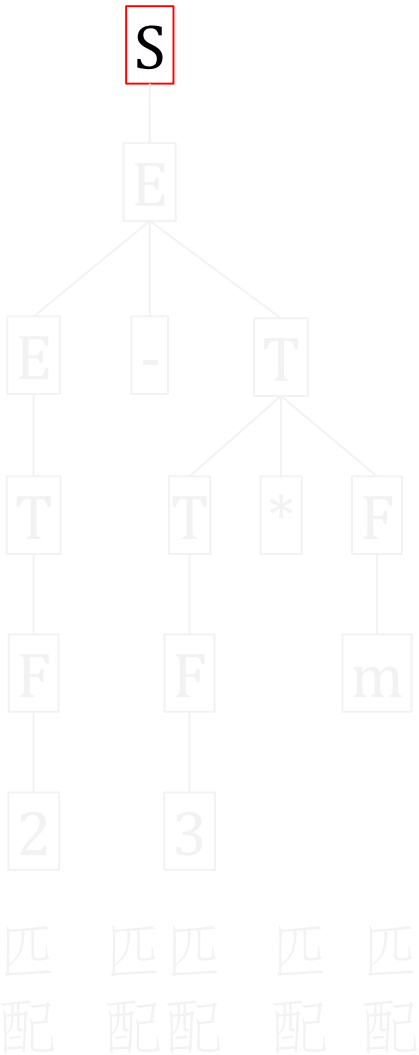
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

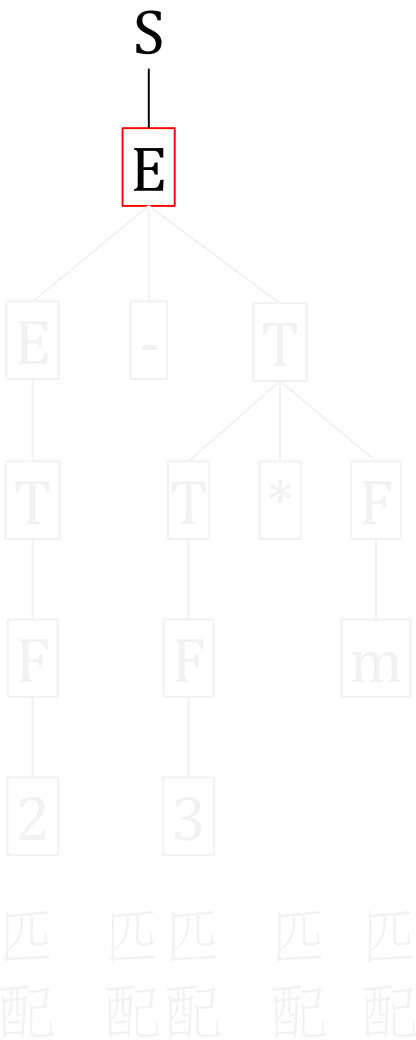
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

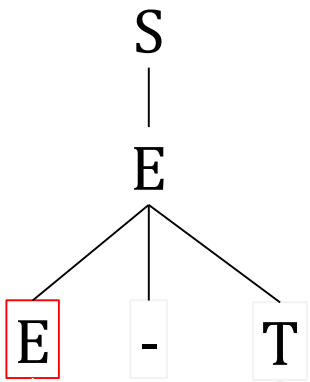
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

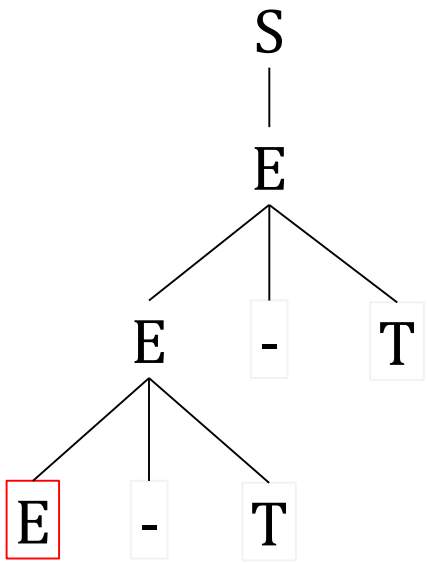
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

E-T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

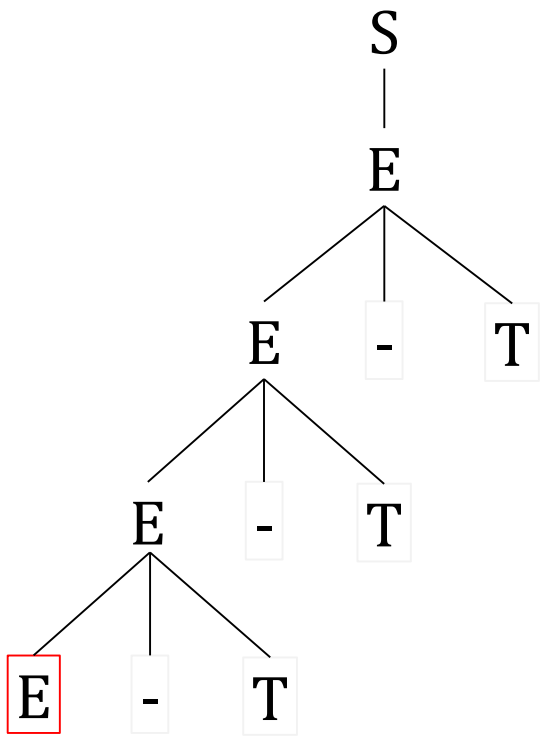
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

E-T-T

E-T-T-T

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

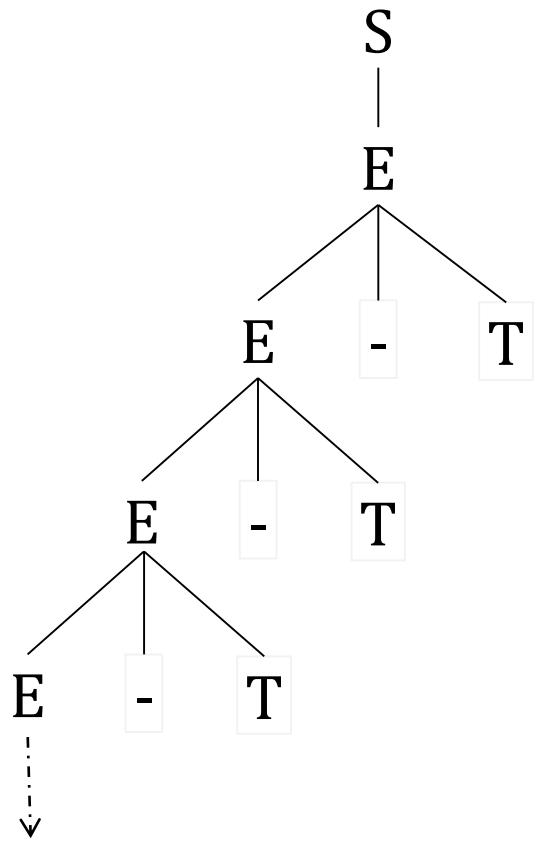
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



“死循环”了！

$\Rightarrow_{lm}^{*}$：

S

E

E-T

E-T-T

E-T-T-T

…

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

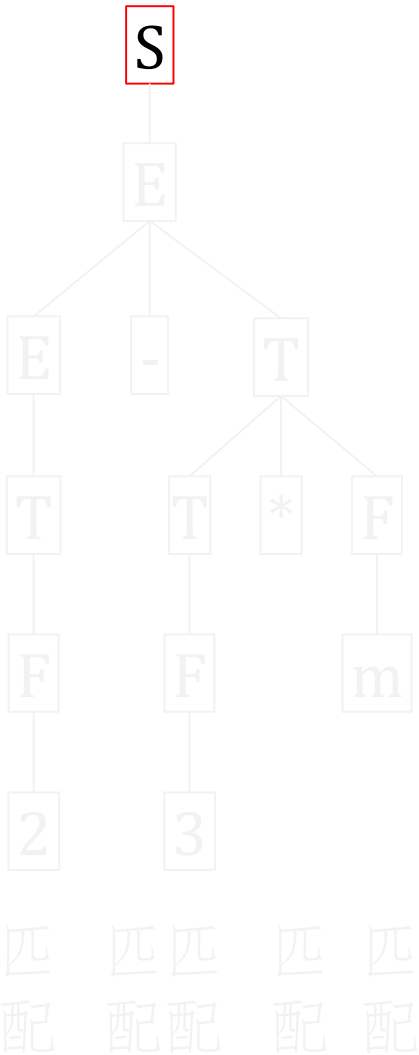
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

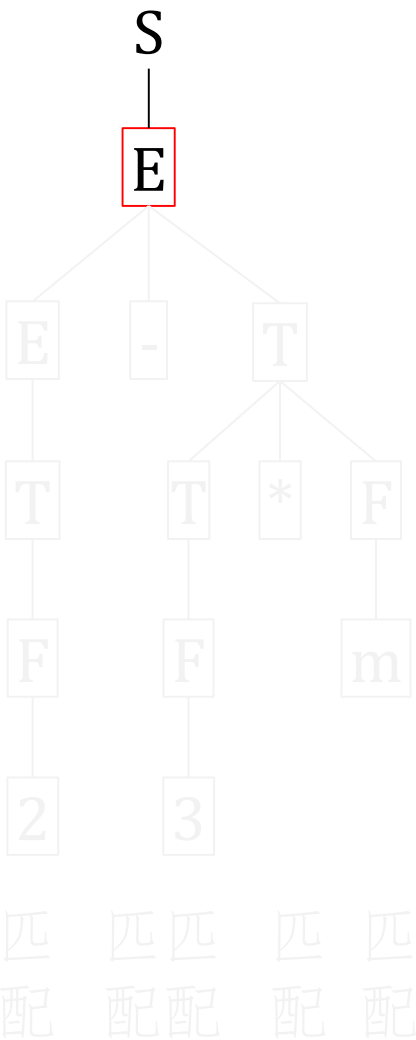
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

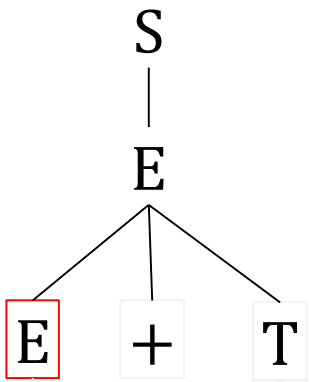
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E+T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

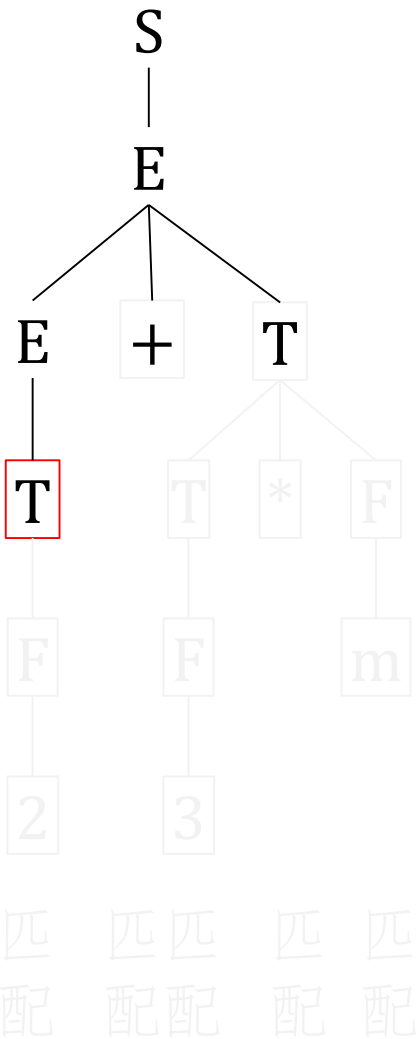
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E+T

T+T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

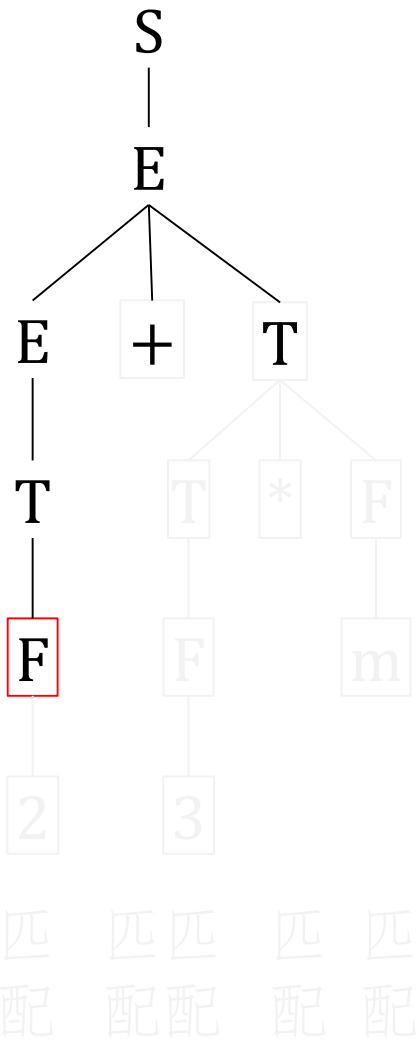
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E+T

T+T

F+T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

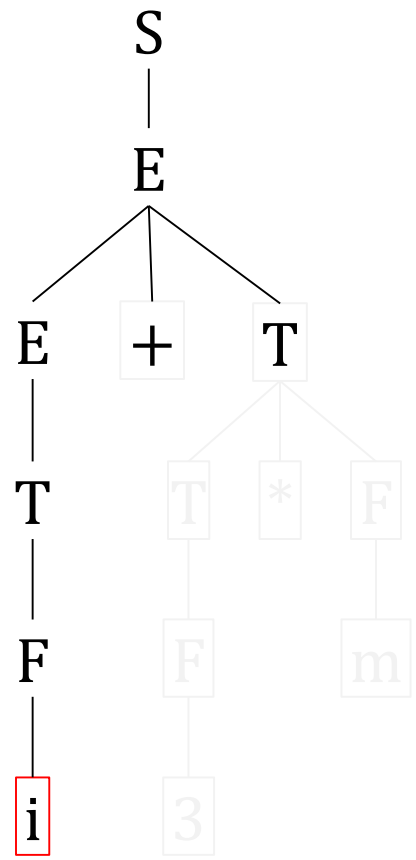
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E+T

T+T

F+T

2+T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

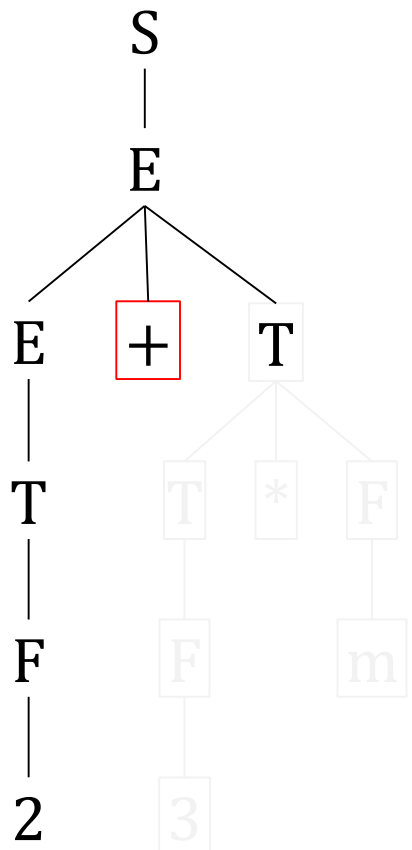
The Parsing Process

剩余输入串：

-3*m

^

当前输入符号



匹配

$\Rightarrow_{lm}^{*}$：

S

E

E+T

T+T

F+T

2+T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

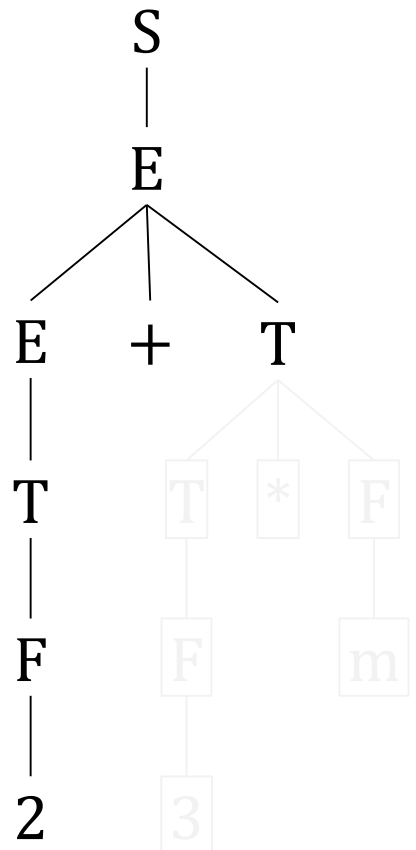
The Parsing Process

剩余输入串：

-3*m

^

当前输入符号



匹配失败！

$\Rightarrow_{lm}^{*}$：

S

E

E+T

T+T

F+T

2+T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

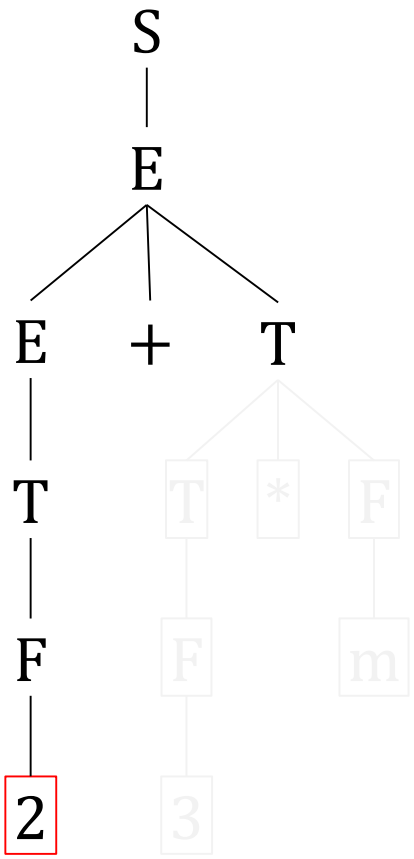
The Parsing Process

剩余输入串：

-3*m

^

当前输入符号



匹配 匹配 匹配 匹配

回溯！

$\Rightarrow_{lm}^{*}$：

S

E

E+T

T+T

F+T

2+T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

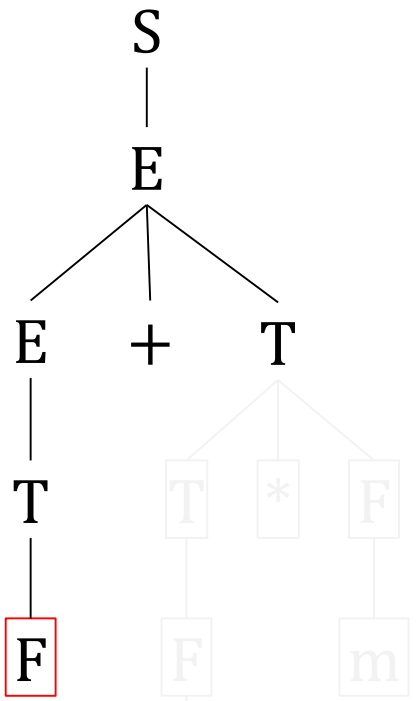
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



匹配 匹配 匹配 匹配

回溯！

$\Rightarrow_{lm}^{*}$：

S

E

E+T

T+T

F+T

2+T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

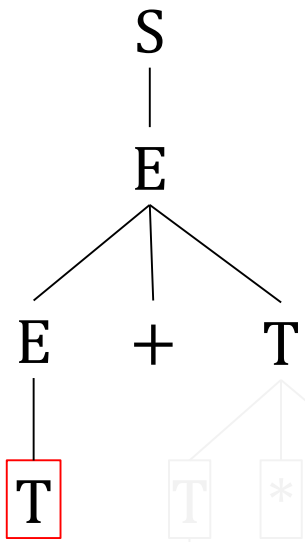
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



回溯！

$\Rightarrow_{lm}^{*}$：

S

E

E+T

T+T

F+T

2+T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

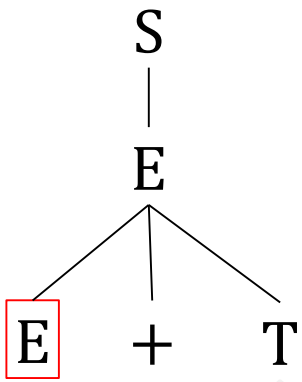
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



匹配 匹配 匹配 匹配

回溯！

$\Rightarrow_{lm}^{*}$：

S

E

E+T

T+T

F+T

2+T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

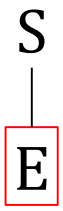
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



回溯！

$\Rightarrow_{lm}^{*}$：

S

E

E+T

T+T

F+T

2+T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

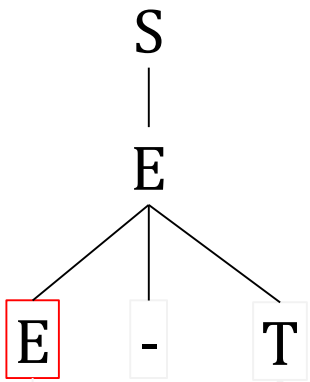
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E - T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

继续扩展下去

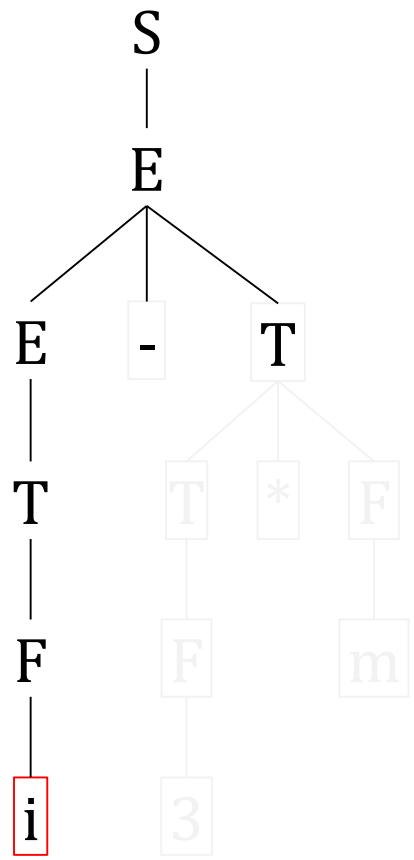
The Parsing Process

剩余输入串：

2-3*m

^

当前输入符号



继续扩展

$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

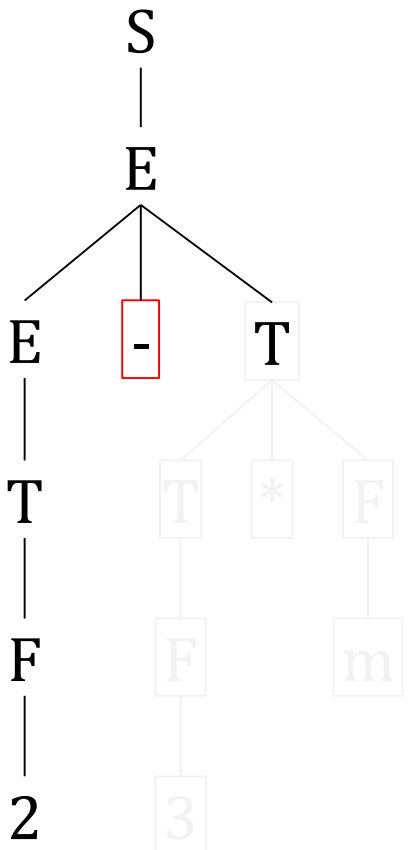
The Parsing Process

剩余输入串：

-3*m

^

当前输入符号



$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

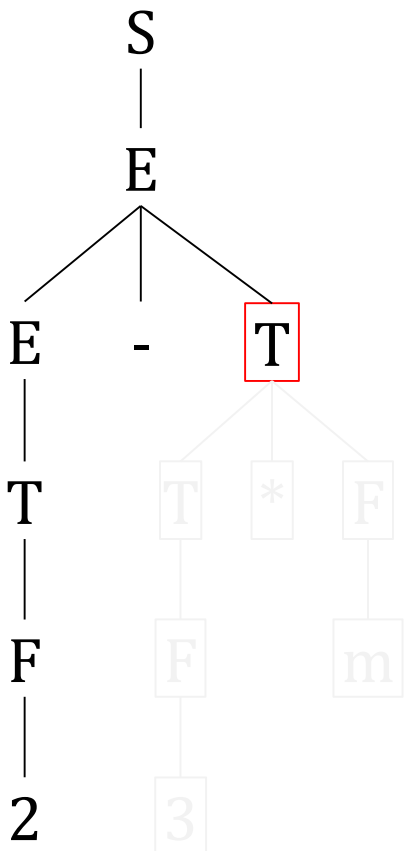
The Parsing Process

剩余输入串：

3*m

^

当前输入符号



匹配

$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

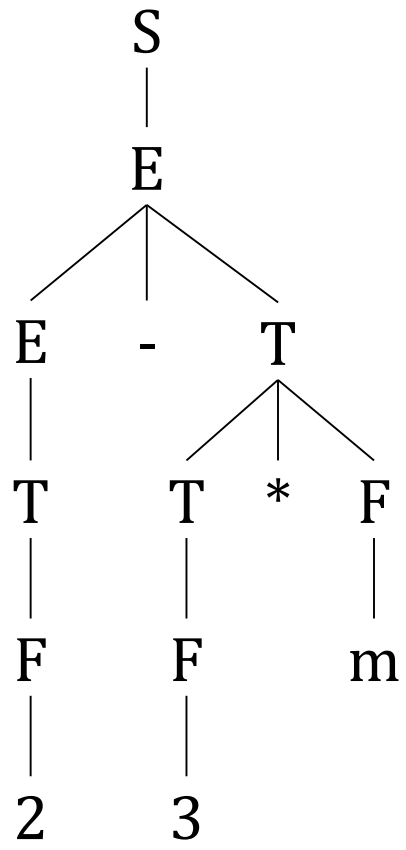
The Parsing Process

剩余输入串：

^

当前输入符号

成功！



最终到达

$\Rightarrow_{lm}^{*}$：

S

E

E-T

T-T

F-T

2-T

2-T*F

2-F*F

2-3*F

2-3*m

文法：

S →E

E →E + T

E →E – T

E →T

T →T * F

T →T / F

T →F

F →(E)

F →i

F →d

归纳“死循环”与回溯问题

- 分析过程中出现“死循环”的原因：
  - 左递归文法，即导致如$P \Rightarrow^{+} P\alpha$这样的文法。
  - 扩展P结点或替换P时可能发生不消耗输入而反复扩展P结点的现象。
- 分析过程中的回溯问题
  - 把分析过程看作一个搜索过程，在每个选择点处试探下个选择，如果已经明确当前的试探失败，则退回到前一个选择点处并试探不同的选项。
  - 产生回溯的原因是在扩展变元结点时存在多个可用候选式。
- 对这个问题的解决对分析过程而言是必要的。
  - 分析过程显然不允许出现“死循环”；
  - 回溯问题让分析过程变得复杂（错误处理、效率），事实上可以简化。

解决思路

- 对于分析过程面临的“死循环”和回溯问题，

通过修剪文法G得到解决。

消除左递归G_t

消除回溯

$G \twoheadrightarrow G_a \twoheadrightarrow G_r \rightarrowtail G_g \Rightarrow G_u$

LL(1)文法

本章内容

- 6.2 LL(1)分析方法
  - 修剪文法以消除左递归和引起回溯的产生式规则得到LL(1)文法。
  - C6.2.1 消除文法中的左递归
  - C6.2.2 消除回溯：计算FIRST集和FOLLOW集
  - C6.2.3 无回溯的分析步骤
  - C6.2.4 LL(1)分析：条件与框架
- 6.3 分析程序的构造
  - 6.3.1 递归下降分析程序
  - 6.3.2 预测分析程序与预测分析表

6.2.1 消除文法左递归

- 直接左递归（消除）
  - $P \rightarrow P\alpha \mid \beta$，其中P不是β的前缀。
- 间接左递归（消除）
  - $N \rightarrow A\alpha$
  - $A \rightarrow N\beta \mid \gamma$
- 间接左递归（回避）（通过限制文法为G_a可以做到）
  - $N \rightarrow \alpha N$
  - $\alpha \Rightarrow^{*} \varepsilon$

消除直接左递归

- 直接左递归
  - $P \rightarrow P\alpha \mid \beta$，其中P不是β的前缀。
  - $L(P)=L(\beta\alpha^*)$
- 修剪为：
  - $P \rightarrow \beta P'$
  - $P' \rightarrow \alpha P' \mid \varepsilon$
- 解释为：
  - $P \Rightarrow^* \beta\alpha^*$的另一种产生式表示
  - 不允许文法中有循环的情况$P \rightarrow P$和$P \Rightarrow^+ P$

推广到一般情形

- 一般地，文法$G_u \equiv G_a \equiv G_r$中以P为左部的产生式形如：
  - $P \rightarrow P\alpha_1 \mid \ldots \mid P\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$，
    其中每个α都不等于ε，而且每个β都不以P开头。

- 通过消除直接左递归得到：
  - $P \rightarrow \beta_1 P' \mid \ldots \mid \beta_n P'$
  - $P' \rightarrow \alpha_1 P' \mid \ldots \mid \alpha_m P' \mid \varepsilon$

- 例：$T \rightarrow T*F \mid T/F \mid F$
- 结果文法：
  - $T \rightarrow FT'$
  - $T' \rightarrow *FT' \mid /FT' \mid \varepsilon$

消除间接左递归

- 间接左递归
  - $N \rightarrow A\alpha$
  - $A \rightarrow N\beta \mid \gamma$
- 应用代入法形成一个产生式之后，即为直接左递归消除之。

- 代入情形把A的代入N的：
  - $N \rightarrow N\beta\alpha \mid \gamma\alpha$
  - $A \rightarrow N\beta \mid \gamma$

结果文法$\equiv G_u$：

$N \rightarrow \gamma\alpha N'$

$N' \rightarrow \beta\alpha N' \mid \varepsilon$

- 代入情形把N的代如A的：
  - $N \rightarrow A\alpha$
  - $A \rightarrow A\alpha\beta \mid \gamma$

结果文法$\equiv G_u$：

$N \rightarrow A\alpha$

$A \rightarrow \gamma A'$

$A' \rightarrow \alpha\beta A' \mid \varepsilon$

消除文法左递归的通用方法

- 对代入加以约束如下：
  - 对文法G_a的所有非终结符进行排序，按次序代入。
- 性质：不同排序得到的结果文法可能不同，但都是等价的。
- 算法：

1. 输入$G_r=G_a=(V,T,P,S)$；输出G'不含左递归产生式。
2. 对P的元素建立索引$1\le i\le |V|$（也就是排序）
3. for(i=1; i<|V|; i++)
4. for(j=1; j< i-1; j++){
5. 将P_j代入P_i的每个候选中（若可代入的话）；
6. 消除关于P_i的直接左递归； }
7. 化简（去除无用的文法符号）即得到G'。

- 算法有效的条件：输入文法不含单位产生式和ε产生式。
- 注意：结果文法中可能含有ε产生式。

举例：消除下述文法的左递归

例1：

S →Qc | c

Q →Rb | b

R →Sa | a

排序为：R, Q, S

S →Sabc | abc | bc | c

Q →Sab | ab | b

R →Sa | a

结果文法：

S →abcS' | bcS' | cS'

S →abcS' | ε

排序为：S, Q, R

S →Qc | c

Q →Rb | b

R →Qca | ca | a

结果文法：

S →Qc | c

Q →Rb | b

R →bcaR' | caR' | aR'

R' →bcaR' | ε

例2：

A →Ac|Bd|ε

B →Ad|Ba

消除ε产生式：

A →c|Ac|Bb

B →d|Ad|Ba

例2续：

A→Bb|Ac|c

B→Ba|Ad|d

i=1,

消除A直接左递归

A→BbA'|cA'

A'→cA'|ε

i=2, j=1

A代入B中

B→Ba|BbA'd|cA'd|d

i=2, j=1

消除B 直接左递归

B→cA'dB'|dB'

B'→aB'|bA'dB'|ε

排序A，B

结果文法

A→BbA'|cA'

A'→cA'|ε

B→cA'dB'|dB'

B'→aB'|bA'dB'|ε

```
for(i=1; i<|V|; i++)
  for( j=1;  j< i-1;  j++){
  将Pj代入Pi的每个候选中；
  消除关于Pi的直接左递归}
```

6.2.2 消除回溯

- 产生回溯的原因分析：
  - 尽管遵守最左推导策略，但多候选式变元在被用于形成直接推导时存在不确定性，从而可能引起分析过程发生回溯。
- 产生回溯的两个原因：
  - 当一个变元的候选式们有公共前缀时，无法根据当前输入符号决定用哪个候选式扩展这个变元。
  - 当一个变元的候选式们推导出的串们有一些有公共终结符前缀时，无法根据当前输入符号决定用哪个候选式扩展这个变元。
  - 对于当前输入符号，有ε候选式的待扩展变元让分析过程不确定是应用该ε产生式呢还是用其它可应用候选式。
- 考虑消除回溯的方法：
  - 分别针对这三个原因构建消除回溯的方法。

原因1：多个候选式有公共前缀

- 例如文法：
  - N → if then
  - N → if then else
- 假定 *N* 是待扩展变元，if 是当前输入符号，两个候选式都可用。

- 解决办法：让公共前缀只出现在单个产生式中
  - N → if then N'
  - N' → else | ε
- 这时，当前输入符号为if时，分析过程只有唯一选择。
- 结论：同一个变元的有公共前缀的候选式们合并为一个最大公共前缀与一个新的变元连接，并将余下的部分都作为新变元的候选式。

问题1的解决

- 若有A→xβ|xγ|xδ|η|...|μ，其中x≠ε， η,...,μ都没有前缀x，
- 那么将其修剪为：
  - A→xA' |η|...|μ
  - A' → β|γ|δ
- 如法继续迭代处理余下的候选式xA',η,...,μ，直到A的候选式彼此都没有公共前缀为止。
- 同时还要如法处理新引入的变元A'等。

原因2：候选式推导出的串有终结符公共前缀

- 例：候选式第一个符号为非终结符时，如：

  $N \rightarrow N_1\ \alpha_1$

  $N \rightarrow N_2\ \alpha_2$

- 选择哪一个取决于N_1和N_2推导出来的串的第一个终结符跟当前输入符号的匹配情况。
- 若N_1或N_2能推导出ε，还需要考虑到α_1和α_2进行判断
- 所以统一观察候选式推导终结符前缀的情况，再进行处理，但仍未考虑到候选式推导不出来终结符前缀的情况（这属于原因3）。

问题2的解决

- 问题描述：
  - 当前输入符号为a，待扩展变元为A时，
  - 若有$A\to\alpha|...|\beta|\eta|...|\mu$，$\alpha\Rightarrow^{*}a\gamma$, ..., $\beta\Rightarrow^{*}a\delta\eta$，其中$a\in T$，且$\eta,...,\mu$都不能推导出前缀为a的串，
  - 那么，选择哪个候选式进行扩展？
- 解决办法：
  - 定义α的首符集，FIRST(α)为α能推导出来的所有串的终结符前缀的集合。若该串为ε那么首符集中包含ε。
  - 修剪文法让每一个变元都满足它的候选式的首符集都两两不相交。

问题3：ε候选式带来的不确定性

- 对于当前输入符号，有ε候选式的待扩展变元让分析过程应用该ε产生式还是其它可应用候选式变得不确定。
- 推广到一般：
  - 对于 $A\rightarrow\alpha_1|...|\alpha_n$，其中$n>0$，存在$\varepsilon\in FIRST(\alpha_k), 0\leq k\leq n$。
  - A已满足它的任意两个候选式的首符集都不相交。

- 讨论：$n=1$；$n>1$; k唯一；k不唯一。
- 问题出在当前输入符号不在任意候选式的首符集里时，是否可以使用α_k扩展A？

问题3的解决

- 若A的候选式的首符集里找不到当前输入符号a时，再看A的FOLLOW集中有没有，若有就用α_k扩展A，否则按出错对待。
- FOLLOW(A)为所有句型中能紧跟在A后面的那些终结符组成的集合。
- 修剪文法对于每个变元，它的候选式首符集两两不相交，同时，若有首符集含ε的话还要求所有候选式的首符集都与左部变元的FOLLOW集不相交。
- 分析过程示范：当a和A分别为当前输入符号和当前待扩展变元时：①若a属于A的候选式α的首符集FIRST(α)，那么应用α扩展A；②若A的候选式α的首符集FIRST(α)包含ε，并且a属于FOLLOW(A)，那么用α扩展A；③其它情况为语法错误。

小结：回溯问题的解决

文法G的产生式形如： $A \rightarrow \alpha_1 | ... | \alpha_n$

则可以计算出每个候选式的首符集，如果任意两个候选式的首符集都两两不相交的话，则意味着由A的每个候选式所能够推导出来的符号串的首符号均不相同。

于是，对语法树中当前结点A，并且当前输入符号为a时只能够唯一地选择A的候选α_k，即$a \in FIRST(\alpha_k)$。

如果还不能选出候选式，则看A的首符集有没有ε，若有的话，再看a是否属于A的FOLLOW集（这里要求A的FOLLOW集与A的首符集不相交），若是，则对A结点唯一地选择A的可空候选式。

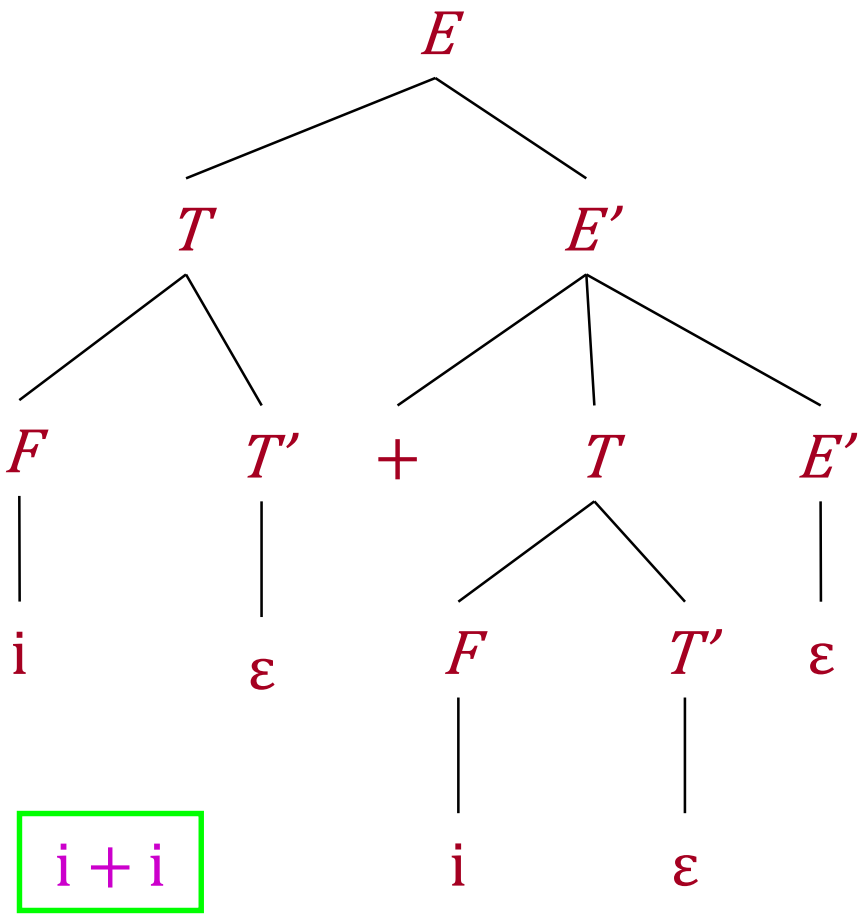
其它情况属于语法错误。

把确定性分析提炼为

- 设当前输入符号为a，语法树当前待扩展结点为A，且有规则$A\rightarrow\alpha_1|\alpha_2|...|\alpha_n$，
  - 如果 a 在$FIRST(\alpha_k)$中则采用规则$A\rightarrow\alpha_k$扩展；
  - 如果 a 不在FIRST(A)中，但是有某个k有$\varepsilon\in FIRST(\alpha_k)$且$a\in FOLLOW(A)$，则采用规则$A\rightarrow\alpha_k$扩展 。
- 若不满足上述情况，则a的出现是一种语法错误。

例：无回溯分析

E ⇒ TE’ ⇒ FT’E’ ⇒ iT’E’



i + i

E→TE’

E’→+TE’|-TE’|ε

T →FT’

T’→*FT’|/FT’|ε

F→(E)| i

i ∈FIRST(TE’)

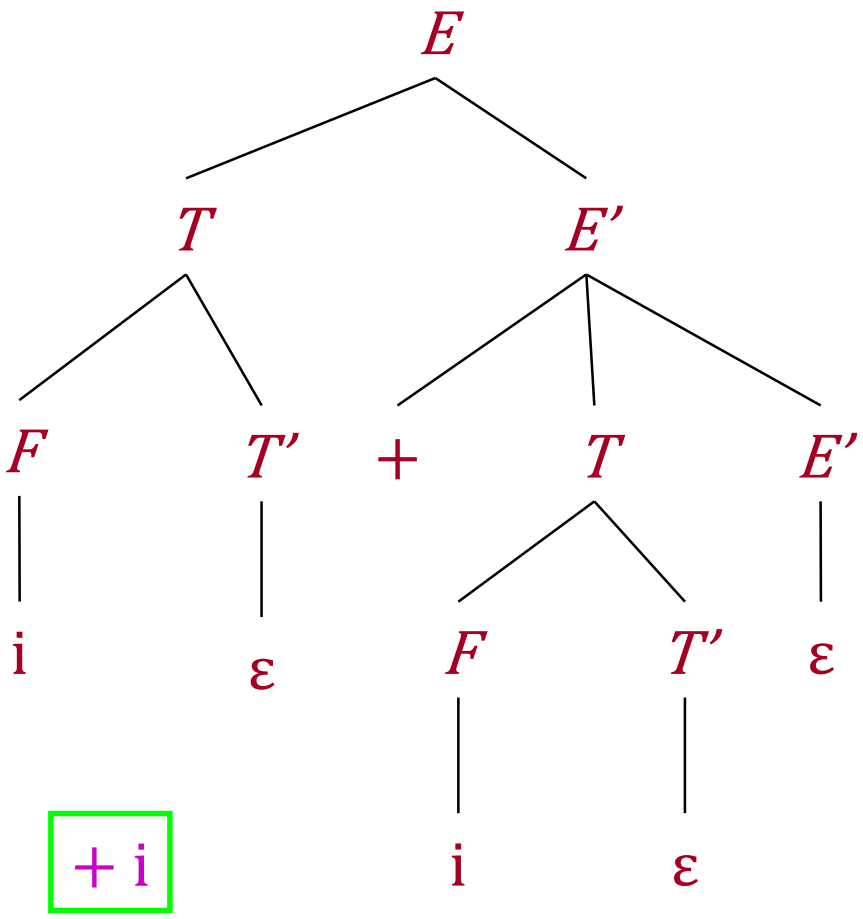
i ∈FIRST(FT’)

i ∈FIRST(i)

例：无回溯分析

E ⇒ TE’ ⇒ FT’E’ ⇒ iT’E’

⇒iE’⇒i+TE’



+ i

E→TE’

E’→+TE’|-TE’|ε

T →FT’

T’→*FT’|/FT’|ε

F→(E)| i

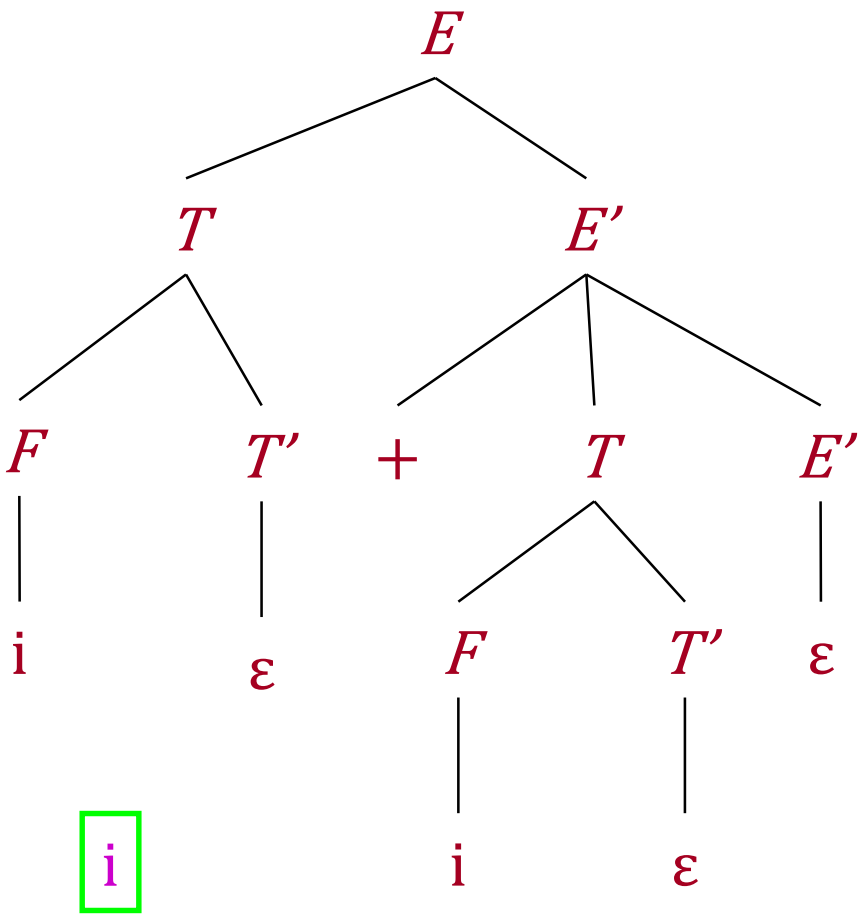
ε ∈FIRST(T’)； +∈FOLLOW(T’)

+∈FIRST(+TE’)

例：无回溯分析

$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow iT'E'$

$\Rightarrow iE' \Rightarrow i+TE' \Rightarrow i+FT'E' \Rightarrow i+iT'E'$



$E \rightarrow TE'$

$E' \rightarrow +TE' | -TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | /FT' | \varepsilon$

$F \rightarrow (E) | i$

$i \in FIRST(FT')$

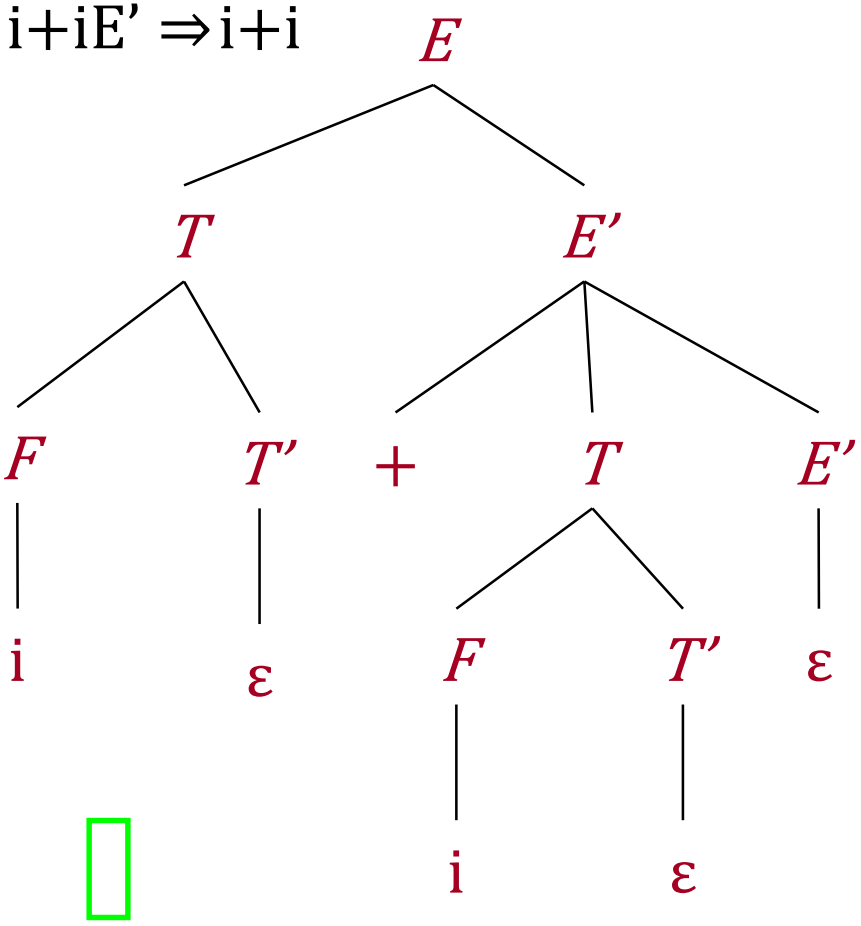
$i \in FIRST(i)$

例：无回溯分析

E ⇒ TE' ⇒ FT'E' ⇒ iT'E'

⇒iE'⇒i+TE' ⇒i+FT'E' ⇒i+iT'E'

⇒i+iE' ⇒i+i



E→TE'

E'→+TE'|-TE'|ε

T →FT'

T'→*FT'|/FT'|ε

F→(E)| i

ε ∈FIRST(T')； #∈FOLLOW(T')

ε ∈FIRST(E')； #∈FOLLOW(E')

6.2.3 无回溯的分析步骤

- 计算首符集和FOLLOW集；
- 修剪文法为LL(1)文法；
- 构建LL(1)分析框架。

计算首符集和FOLLOW集

- FIRST(α)为α能推导出来的所有串的终结符前缀的集合。若该串为ε那么首符集中包含ε。
  - FIRST(α)={a|α ⇒* aβ，a∈T, β∈(V∪T)*}；
  - 若α ⇒*ε，则ε∈FIRST(α)。
- FOLLOW(A)为所有句型中能紧跟在A后面的那些终结符组成的集合。
  - FOLLOW(B) = {a∈T∪{#} | S⇒*αBγ, a∈FIRST(γ#)}
  - 为了一致期间，给输入串加一个后缀#，该符号被解释为终结符，但不属于T。
  - 初始符号的FOLLOW集中必然有#。
- 计算方法的显著特点是反复扫描产生式规则最终得到结果。

计算文法符号的首符集

输入：CFG(V,T,P,S)；

输出：文法符号的首符集

for all a∈T do FIRST(a):={a}；

for all A∈V do FIRST(A):={}；

While there are changes to any FIRST(A) do

    for each production A →α do

    add FIRST(α) to FIRST(A)；

计算符号串的首符集

输入：CFG(V,T,P,S)，$\alpha=X_1X_2...X_n$；$\alpha\in(V\cup T)^*$，$n\geq 0$

输出：$FIRST(X_1X_2...X_n)$

k := 1; Continue := true;

while (Continue = true && k ≤ n) {

add $FIRST(X_k) \setminus \{\varepsilon\}$ to $FIRST(\alpha)$;

if (ε is not in $FIRST(X_k)$) Continue := false;

k := k + 1;}

if (Continue = true) add {ε} to FIRST(α);

对于候选式α：
α=ε，ε加入FIRST(α)
α=a，FIRST(α)={a}
α=a...，a加入FIRST(α)

找出α的最大前缀ρ且$\rho\Rightarrow^*\varepsilon$，
让ρ的每个元素的首符集\{ε}
都加入FIRST(α)
如果ρ=α那么ε加入FIRST(α)

例：计算FIRST集

- P →Ď Š
- Ď →ε | D; Ď
- Š →S | Š; S
- D →T d
- T →int
- S →d(e)

A→α， FIRST(α)加入FIRST(A)

找出α的最大前缀ρ且ρ⇒*ε，
让ρ的每个元素的首符集\{ε}
都加入FIRST(α)
如果ρ=α那么ε加入FIRST(α)

;	{;}			
d	{d}			
int	{int}			
(	{(}			
e	{e}			
)	{)}			
P	{}	{}		{int,d}
Ď	{ε}		{ε,int}	
Š	{}	{d}		
D	{}	{int}		
T	{int}			
S	{d}			

若a∈T，则FIRST(a)={a}
若α=ε，ε加入FIRST(α)
若α=a...，a加入FIRST(α)

例：计算FIRST集

- E→TE'
- E'→+TE'|-TE'|ε
- T →FT'
- T'→*FT'|/FT'|ε
- F→(E)| i

- \+ {+}
- \- {-}
- * {*}
- / {/}
- ({(}
-) {)}
- i {i}
- E {} {(,i}
- E' {+,-,ε}
- T {} {(,i}
- T' {*,/,ε}
- F {(,i}

计算FOLLOW集的约束规则

- 对于任一非终结符A，FOLLOW(A)是一个由终结符组成的集合，可能含#，

  - 若A为文法初始符号，则#属于FOLLOW(A)　　（规则1）

  - 若有产生式 $B\rightarrow\alpha A\beta$

    则　　$FIRST(\beta)\setminus\{\varepsilon\}\subseteq FOLLOW(A)$　　（规则2）

  - 若有产生式 $B\rightarrow\alpha A\beta$ 且$\varepsilon\in FIRST(\beta)$

    则　　$FOLLOW(B)\subseteq FOLLOW(A)$　　（规则3）

计算FOLLOW集的算法

输入：CFG(V,T,P,S)；输出：每个非终结符的FOLLOW集

FOLLOW(S) := {#}; //规则1

for all A∈V do if A≠S then FOLLOW (A):={}; //规则1

while there are changes to any FOLLOW(A) do

for each production A →$X_1X_2...X_n$ do

for i := 1 to n do

if $X_i \in V$ then

add FIRST($X_{i+1}X_{i+2}...X_n$) \ {ε} to FOLLOW(X_i)//规则2

if ε ∈FIRST($X_{i+1}X_{i+2}...X_n$) then

add FOLLOW(A) to FOLLOW(X_i) //规则3

注：当i=n时$X_{i+1}X_{i+2}...X_n$=ε

例：计算FOLLOW集

- ; {;}
- d {d}
- int {int}
- ({(}
- e {e}
-) {)}
- P {int,d}
- Ď {ε,int}
- Š {d}
- D {int}
- T {int}
- S {d}

- 规则1：FOLLOW(S)={#}，其它空；
- 规则2：若A→αBβ是一个产生式，则将FIRST(β)\{ε}加入FOLLOW(B)中；
- 规则3：若A→αB是一个产生式，或者A→αBβ是产生式且ε∈FIRST(β)则将FOLLOW(A)加入FOLLOW(B)中。

- P →Ď Š
- Ď →ε | D; Ď
- Š →S | Š; S
- D →T d
- T →int
- S →d(e)

P	{#}		
Ď	{}	{d}	
Š	{}	{;}	{#,;}
D	{}	{;}	
T	{}	{d}	
S	{}		{#,;}

例：计算FOLLOW集

- \+ {+}
- \- {-}
- * {*}
- / {/}
- ({(}
-) {)}
- i {i}
- E {(,i}
- E' {+,-,ε}
- T {(,i}
- T' {*,/,ε}
- F {(,i}

- E→TE'
- E'→+TE'|-TE'|ε
- T →FT'
- T'→*FT'|/FT'|ε
- F→(E)| i

E	{#}	{#,)}	
E'	{}	{}	{#,)}
T	{}	{+,-}	{+,-,#,)}
T'	{}	{}	{+,-,#,)}
F	{}	{*,/}	{*,/,+,-,#,)}

- 规则1：FOLLOW(S)={#}，其它空；
- 规则2：若A→αBβ是一个产生式，则将FIRST(β) \ {ε}加入FOLLOW(B)中；
- 规则3：若A→αB是一个产生式，或者A→αBβ是产生式且ε∈FIRST(β)则将FOLLOW(A)加入FOLLOW(B)中。

LL(1)文法定义

- 文法$G=(\mathbb{V}, T, \boldsymbol{\mathcal{P}}, S)$被称为LL(1)文法，当且仅当：

① $\forall(A,\alpha)\in\boldsymbol{\mathcal{P}}\cdot\alpha\neq A\beta, \beta\in(\mathbb{V}\cup T)^*$
（文法不含左递归产生式）；并且，

② 对G的任意产生式$A\to\alpha_1|\alpha_2|\ldots|\alpha_n, n\in N^+$，都满足，
$\forall 1\leq i,j\leq n\cdot i\neq j\to FIRST(\alpha_i)\cap FIRST(\alpha_j)=\varphi$
（变元的各个候选式的首符集两两不相交）；并且，

③ $\forall(A,\alpha)\in\boldsymbol{\mathcal{P}}\cdot\varepsilon\in FIRST(\alpha)\to FIRST(A)\cap FOLLOW(A)=\varphi$
（若变元的首符集含ε那么与FOLLOW集不相交）
（$\forall A\in\mathbb{V}\cdot\varepsilon\in FIRST(A)\to FIRST(A)\cap FOLLOW(A)=\varphi$）。

- 讨论LL(1)文法G：
  - $G\equiv G_u$?　　$G\equiv G_g$?　　$G\equiv G_a$?　　$G\equiv G_r$?
  - $\forall(A,\alpha)\in\boldsymbol{\mathcal{P}}\cdot\varepsilon\in FIRST(\alpha)\to FIRST(\alpha)\cap FOLLOW(A)=\varphi$?

归纳：修剪为LL(1)文法的算法

- 已知文法$G=(\mathbb{V}, T, \boldsymbol{\mathcal{P}}, S)$修剪为LL(1)文法的算法：
- 输入：G；输出：G_d

① 把G修剪为G_a；　（去除ε产生式）

② 把G_a修剪为G_r；　（去除单位产生式）

③ 把G_r修剪为G_g再修剪为G_u；　（去除无用符号）

④ 把G_u修剪为无左递归产生式的文法G_t（exhaustible）；

⑤ 把G_t修剪为LL(1)文法G_d（determind）。

- 讨论G修剪为G_d的过程：
  - $G_t \equiv G_u$?　　$G_t \equiv G_g$?$G_t \equiv G_a$?$G_t \equiv G_r$?
  - $G_d \equiv G_u$?　　$G_d \equiv G_g$?　　$G_d \equiv G_a$?　　$G_d \equiv G_r$?

LL(1)分析框架

- 初始化：文法初始符号为当前待扩展结点；输入串第一个符号设为当前输入符号。
- 重复以下步骤直至无可扩展变元或出错：
  - 设当前输入符号为a，语法树当前待扩展结点为A，且有产生式规则$A\rightarrow\alpha_1|\alpha_2|...|\alpha_n$，
  - 如果 a 在$FIRST(\alpha_k)$中则采用规则$A\rightarrow\alpha_k$扩展；
  - 如果 a 不在FIRST(A)中，但是有某个k有$\varepsilon\in FIRST(\alpha_k)$且$a\in FOLLOW(A)$，则采用规则$A\rightarrow\alpha_k$扩展 。
  - 若不满足上述情况，则a的出现是语法错误。

C4.4 递归下降分析程序

- LL(1)文法$(\mathbb{V}, \mathrm{T}, \boldsymbol{\mathcal{P}}, \mathrm{S})$的分析程序是对LL(1)分析框架的一种程序实现，实现成每个变元对应一个递归子程序的形式。

- 下面通过一个例子展示如何进行程序实现。

例：文法和程序

- 已知文法（CFG）：
- P →Ď Š
- Ď →ε | D ; Ď
- D →int d
- Š →S | Š ; S
- S →d = E | print d
- E →i | d | E + i | E * d

```
int x;
int y;
x=1;
y=2*x+1;
print x;
print y
```

例：修剪为LL(1)文法

- 去除单位产生式：
- Š →d = E | print d | Š ; d = E | Š ; print d
- 消除直接左递归：
- Š →d = E Š' | print d Š'
- Š' →; d = E Š' | ; print d Š' | ε
- 消除回溯：
- Š' →; S | ε
- S →d = E Š' | print d Š'
- 消除左递归：
- E →i E' | d E'
- E' →+ i E' | * d E' | ε
- 消除冗余：（2和5候选式相同，放弃S，修改4为8）
- Š' →; Š | ε

已知文法：

- P →Ď Š
- Ď →ε | D ; Ď
- D →int d
- Š →S | Š ; S
- S →d = E | print d
- E →i | d | E + i | E * d

检查结果文法是否为LL(1)文法

- 结果文法：
- P →Ď Š
- Ď →ε | D ; Ď
- D →int d
- Š →d = E Š' | print d Š'
- Š' →; Š | ε
- E →i E' | d E'
- E' →+ i E' | * d E' | ε

- 首符集

P	{}	{int}	{int,d,print}
Ď	{ε}	{ε,int}	
D	{int}		
Š	{d,print}		
Š'	{;,ε}		
E	{i,d}		
E'	{+,*,ε}		

结果文法是LL(1)文法

- LL(1)文法：
- P →Ď Š
- Ď →ε | D ; Ď
- D →int d
- Š →d = E Š' | print d Š'
- Š' →; Š | ε
- E →i E' | d E'
- E' →+ i E' | * d E' | ε

- 首符集

P	{int,d,print}
Ď	{ε,int}
D	{int}
Š	{d,print}
Š'	{;,ε}
E	{i,d}
E'	{+,*,ε}

- FOLLOW集

P	{#}	
Ď	{d,print}	
D	{;}	
Š	{}	{#}
Š'	{}	{#}
E	{;}	{;,#}
E'	{}	{;,#}

讨论：在解决具体问题时存在灵活性

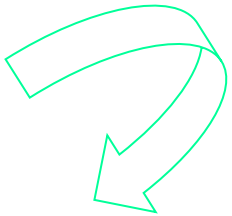
- G_a （考虑产生式分组独立处理，如ε产生式与左递归产生式不在一个组的情形）
- G_r （单位产生式不会引起$P \Rightarrow {}^{+}P$）
- G_g （）
- G_u （考虑到消除左递归后出现不可达符号的情况）
- G_t （考虑到再消除ε产生式又导致左递归的情况，所以消除左递归给文法带来新的ε产生式）
- G_d （对每个变元，通过提取候选式公共前缀并引入新变元的办法合并它们，这些候选式出去公共前缀余下的部分都作为新变元的候选式。这次修剪是在G_t上进行，直到达到LL(1)文法条件。注意到一些新变元是琐碎的。）
- 文法修剪过程中也可能出现重复的候选式，可以去除重复。

- 总之，建议以求出LL(1)文法为目标灵活运用。

①G->Ga->Gr

P →Ď Š
Ď →ε | D ; Ď
D →int d
Š →S | Š ; S
S →d = E | print d
E →i | d | E + i | E * d

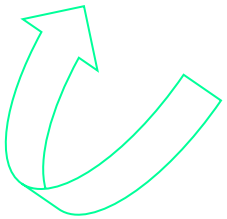
P →Ď Š | Š
Ď →D ; Ď | D ;
D →int d
Š →S | Š ; S
S →d = E | print d
E →i | d | E + i | E * d



P →Ď Š | d = E | print d | Š ; S
Ď →D ; Ď | D ;
D →int d
Š →d = E | print d | Š ; S
S →d = E | print d
E →i | d | E + i | E * d

①消除直接左递归

P →Ď Š | d = E | print d | Š ; S
Ď →D ; Ď | D ;
D →int d
Š →d = E Š' | print d Š'
Š' →; S Š' | ε
S →d = E | print d
E →i E' | d E'
E' →+ i E' | * d E' | ε



P →Ď Š | d = E | print d | Š ; S
Ď →D ; Ď | D ;
D →int d
Š →d = E | print d | Š ; S
S →d = E | print d
E →i | d | E + i | E * d

①消除回溯?

P →Ď Š | d = E | print d | Š ; S	{d,print,int} {#}
Ď →D ; Ď | D ;	{int} {d,print}
D →int d	{int} {;}
Š →d = E Š' | print d Š'	{d,print} {;,#}
Š' →; S Š' | ε	{;,ε} {;,#}
S →d = E | print d	{d,print} {;,#}
E →i E' | d E'	{i,d} {;,#}
E' →+ i E' | * d E' | ε	{+,*,ε} {;,#}

②G->Ga

P →Ď Š

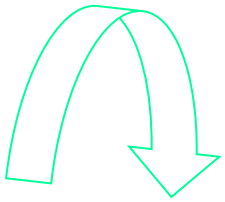
Ď →ε | D ; Ď

D →int d

Š →S | Š ; S

S →d = E | print d

E →i | d | E + i | E * d



P →Ď Š | Š

Ď →D ; Ď | D ;

D →int d

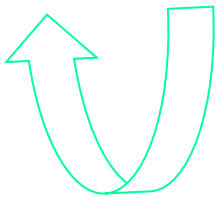
Š →S | Š ; S

S →d = E | print d

E →i | d | E + i | E * d

②消除左递归

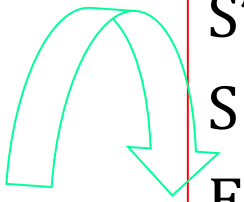
P →Ď Š | Š
Ď →D ; Ď | D ;
D →int d
Š →S Š'
Š' →; S Š' | ε
S →d = E | print d
E →i E' | d E'
E' →+ i E' | * d E' | ε



P →Ď Š | Š
Ď →D ; Ď | D ;
D →int d
Š →S | Š ; S
S →d = E | print d
E →i | d | E + i | E * d

②消除回溯

P →Ď Š | Š
Ď →D ; Ď | D ;
D →int d
Š →S Š'
Š' →; S Š' | ε
S →d = E | print d
E →i E' | d E'
E' →+ i E' | * d E' | ε



P →Ď Š | Š
Ď →D ; D'
D' →Ď | ε
D →int d
Š →S Š'
Š' →; S Š' | ε
S →d = E | print d
E →i E' | d E'
E' →+ i E' | * d E' | ε

②LL(1)文法

P →Ď Š | Š	{int,d,print}	{#}
Ď →D ; D'	{int}	{d,print}
D' →Ď | ε	{int,ε}	{d,print}
D →int d	{int}	{;}
Š →S Š'	{d,print}	{#}
Š' →; S Š' | ε	{;,ε}	{#}
S →d = E | print d	{d,print}	{;,#}
E →i E' | d E'	{i,d}	{;,#}
E' →+ i E' | * d E' | ε	{+,*,ε}	{;,#}

②LL(1)文法

P →Ď Š | Š
Ď →D ; D’
D’ →Ď | ε
D →int d
Š →S Š’
Š’ →; S Š’ | ε
S →d = E | print d
E →i E’ | d E’
E’ →+ i E’ | * d E’ | ε

P	{int,d,print}	{#}
Ď	{int}	{d,print}
D’	{int,ε}	{d,print}
D	{int}	{;}
Š	{d,print}	{#}
Š’	{;,ε}	{#}
S	{d,print}	{;,#}
E	{i,d}	{;,#}
E’	{+,*,ε}	{;,#}

③消除左递归

P →Ď Š

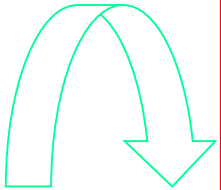
Ď →ε | D ; Ď

D →int d

Š →S | Š ; S

S →d = E | print d

E →i | d | E + i | E * d



P →Ď Š

Ď →ε | D ; Ď

D →int d

Š →S Š'

Š' →; S Š' | ε

S →d = E | print d

E →i E' | d E'

E' →+ i E' | * d E' | ε

③LL(1)文法

P →Ď Š
Ď →ε | D ; Ď
D →int d
Š →S Š'
Š' →; S Š' | ε
S →d = E | print d
E →i E' | d E'
E' →+ i E' | * d E' | ε

P	{int,d,print}	{#}
Ď	{int,ε}	{d,print}
D	{int}	{;}
Š	{d,print}	{#}
Š'	{;,ε}	{#}
S	{d,print}	{;,#}
E	{i,d}	{;,#}
E'	{+,*,ε}	{;,#}

例：递归下降分析程序构造

- scan()返回词法单位类别：
- NUM、ID、SCO、MUL、ADD、ASG、INT、PRINT
- int tok;　　//当前token

```
int scan(){
    tok=scanner();
}

void main(){
    scan();
    Prog();
}
```

LL(1)文法：

P →Ď Š

Ď →ε | D ; Ď

D →int d

Š →d = E Š' | print d Š'

Š' →; Š | ε

E →i E' | d E'

E' →+ i E' | * d E' | ε

例：递归下降分析程序构造

- P →Ď Š　　{int,d,print}　{#}

```
int P() {if(tok∈[INT ID PRINT]){Dlist();Slist();}else error();}
int Prog(){
        if(tok==INT||tok==ID||tok==PRINT){Dlist();Slist();}
        else error();}
```

- Ď →ε | D ; Ď　　{ε,int}　{d,print}

```
int Dlist()     {
    if(tok∈[INT]){D();match(SCO);Dlist();}
        else if(tok∉[ID PRINT])error();}
int Dlist(){
        if(tok==INT){
                Decl();if(tok==SCO){scan();Dlist();}else error()
        }else if(tok!=ID&&tok!=PRINT)error();}
```

例：递归下降分析程序构造

- D →int d {int} {;}

```
int D() {if(tok ∈[INT]){match(INT);match(ID);}else error();}
int Decl(){if(tok==INT){scan();
        if(tok==ID)scan()else error();}else error();}
```

- Š →d = E Š' | print d Š' {d,print} {#}

```
int Slist()     {
        if(tok∈[ID]){match(ID);match(ASG);E();Slp();}
        else if(tok∈[PRINT]){match(PRINT);match(ID);Slp();}
        else error();}
int Slist(){
        if(tok==ID){scan();if(tok==ASG){scan();Exp();Slp();}}
        else if(tok==PRINT){scan();if(tok==ID){scan();Slp();}}
        else error();}
```

例：递归下降分析程序构造

- Š' →; Š | ε　　{;,ε}　　{#}

```
int Slp()       {
        if(tok∈[SCO]){match(SCO);Slist();}
        else if(tok ∉[EOF])error();}
int Slp(){if(tok==SCO){scan();Slist();}
                else if(tok!=EOF)error();}}
```

- E →i E' | d E'　　{i,d}　　{;,#}

```
int E()         {
        if(tok∈[NUM]){match(NUM);Ep();}
        else if(tok∈[ID]){match(ID);Ep();}else  error();}
int Exp(){if(tok==NUM){scan();Ep();}
        else if(tok==ID){scan();Ep();}else error();}
```

例：递归下降分析程序构造

- E' →+ i E' | * d E' | ε　　{+,*,ε}　　{;,#}

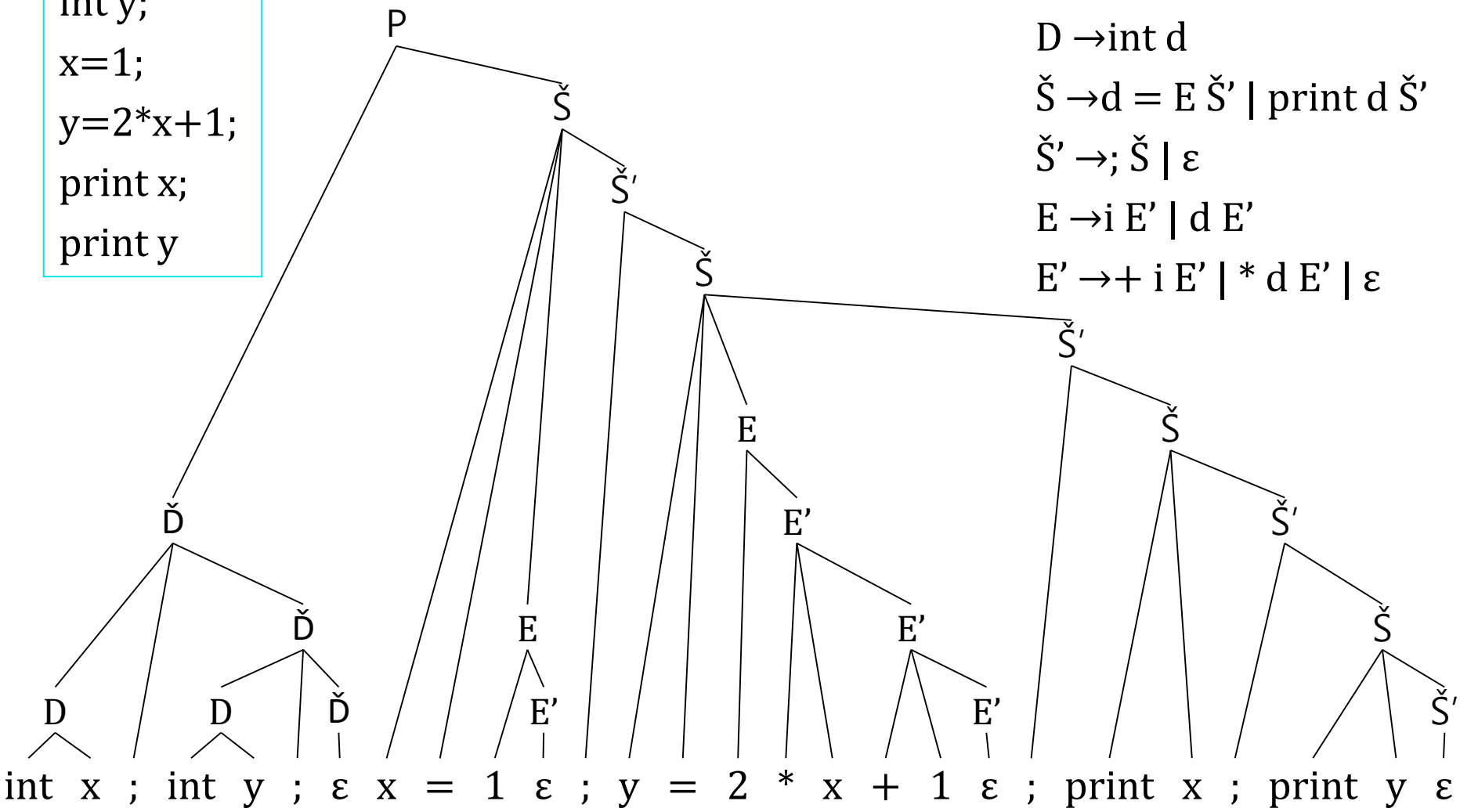
```
int Ep()        {
        if(tok ∈[ADD]){match(ADD);match[NUM];Ep();}
        else if(tok ∈[MUL]){match(MUL);match(ID);Ep();}
        else if(tok∉[SCO EOF])error();}
```

```
int Ep(){
        if(tok==ADD){scan();
                if(tok==NUM){scan();Ep();}else error();}
        else if(tok==MUL){scan();
                if(tok==ID){scan();Ep();}else error();}
        else if(tok!=SCO&&tok!=EOF)error();}
```

下一步：输出给定程序的语法树

```
int x;
int y;
x=1;
y=2*x+1;
print x;
print y
```

P →Ď Š
Ď →ε | D ; Ď
D →int d
Š →d = E Š' | print d Š'
Š' →; Š | ε
E →i E' | d E'
E' →+ i E' | * d E' | ε



例：构建语法树

- struct node {int name; struct node *laoda; struct node *laoer; struct node *laosan; struct node *laosi; }
- #define P 1001; #define DLP 1010; #define DL 1002; #define D 1003; #define SL 1004; #define SLP 1005; #define E 1006; #define EP 1007;
- int tok; int scan(){tok=scanner();}
- node* tree;
- node *mknode(name){
  t=new(node); t.name=name; t.laoda=NULL; return t;}
- void main(){
- scan();
- tree=mknode(P);
- Prog(tree);
- output(tree);}　　//输出形式建议采用嵌套表示形式

例：生成语法树

- P →Ď Š　　{int,d,print}　{#}

```
int Prog(tree){
   if(tok==INT||tok==ID||tok==PRINT){
         t=mknode(DL);
         tree.laoda=t;
         Dlist(t);
         t=mknode(SL);
         tree.laoer=t;
         Slist(t);
   }else error();
}
```

例：生成语法树

- Ď →ε | D ; Ď　　{ε,int}　{d,print}

```
int Dlist(tree){
  if(tok==INT){
      t=mknode(D); tree.laoda=t; Decl(t);
      if(tok==SCO){
          tree.laoer=mknode(tok); scan();
          t=mknode(DL); tree.laosan=t; Dlist(t);
      }else if (tok ==ID||tok==PRINT){
          tree.laoda=mknode(EPSILON);
      }else error();
  }else error();
}
```

例：生成语法树

- D →int d　　{int}　{;}

```
int Decl(tree){
   if(token==INT){
        tree.laoda=mknode(INT);
        scan();
        if(token==ID){
             tree.laoer=mknode(ID);
             scan();
        }else error();
   }else if(tok!=SCO)error();
}
```

例：生成语法树

- Š' →; Š | ε　　{;,ε}　　{#}

```
int Slp(tree){if(token==SCO){tree.laoda=mknode(SCO);
        scan();t=mknode(SL); tree.laoer=t; Slist(t);
        }else if(token==EOF){tree.laoda=mknode(EPSILON);
        }else error(); }
```

例：生成语法树

- Š →d = E Š' | print d Š'　　{d,print}　　{#}

```
int Slist(tree){
 if(tok==ID){tree.laoda=mknode(tok); scan();
        if(tok==ASG){tree.laoer=mknode(tok); scan();
        t=maknode(E);tree.laosan=t;Exp(t); t=mknode(SLP);
        tree.laosi=t; Slp(t);
        }else error();
 }else if(tok==PRINT){tree.laoda=mknode(tok); scan();
        if(tok==ID){tree.laoer=mknode(tok); scan();
                t=maknode(SLP);tree.laosan=t;Slp(t);
        }else error();
 }else if(tok!=EOF) error();
}
```

例：生成语法树

- Š' →; Š | ε　　{;,ε}　　{#}

```
int Slp(tree){
   if(token==SCO){
        tree.laoda=mknode(SCO);
        scan();
        t=mknode(SL);
        tree.laoer=t; Slist(t);
   }else if(token==EOF){
        tree.laoda=mknode(EPSILON);
   }else error();
}
```

例：生成语法树

- Š →d = E Š' | print d Š'　　{i,d}　{;,#}

```
int Slist(tree){
  if(tok==ID){
        tree.laoda=mknode(tok); scan();
        if(tok==ASG){
          tree.laoer=mknode(tok); scan(); t=maknode(E);
          tree.laosan=t; Exp(t); t=mknode(SLP);tree.laosi=t; Slp(t);
        }else error();
  }else if(tok==PRINT){
        tree.laoda=mknode(tok); scan();
        if(tok==ID){
          tree.laoer=mknode(tok); scan();
          t=maknode(SLP);tree.laosan=t;Slp(t);
        }else error();
  }else error();
}
```

例：生成语法树

- E →i E' | d E'　　{i,d}　{;,#}

```
int Exp(tree){if(tok==NUM||tok==ID){
 tree.laoda=mknode(tok); scan(); t=mknode(EP);
 tree.laoer=t; Ep(t);}else error();}
```

- E' →+ i E' | * d E' | ε　　{+,*,ε}　　{;,#}

```
int Ep(tree){if(tok==ADD){tree.laoda=mknode(tok);scan();
if(tok==NUM){tree.laoer=mknode(tok);scan(); t=mknode(EP);
tree.laosan=t; Ep(t);}else error();}else
if(tok==MUL){tree.laoda=mknode(tok);scan();
if(tok==ID){tree.laoer=mknode(tok);scan();t=mknode(EP);
tree.laosan=t; Ep(t);}else if(tok==SCO||tok==EOF){
tree.laoda=mknode(EPSILON);}else error();}
```

小结：递归下降分析程序

- 调用一次词法分析器返回一个记号。
- 每一个变元（非终结符）对应一个递归过程；
  - 在递归过程中，根据当前符号和变元的首符集、FOLLOW集决定所用候选式
  - 若当前符号属于所用候选式的首符集，那么对于候选式逐符号依次进行：遇到终结符则匹配掉（失败即出错）；遇变元则调用对应递归过程；处理完即返回。
  - 若首符集有ε则当前符号为FOLLOW集元素时应用首符集含ε的候选式，随后同上一步处理。
  - 对于其它情况为出错。
- 出错时就退出分析程序进入错误处理。
- 如果没有错误就严格按照LL(1)分析过程进行完。
- 直到所有递归过程都返回就表示分析成功。

C4.5 预测分析程序

- 是对LL(1)分析框架的另一种实现。
- 将LL(1)分析的条件表示为预测分析表
- 采用预测分析表和栈实现自上而下的语法分析程序
- 这种方法实现的语法分析程序又叫预测分析程序

预测分析表M[A, a]

- 当前待扩展变元为A，当前输入符号为a时，采用M[A, a]元素作为候选式进行扩展。因为M[A,a]满足：
  - 如果 a 在FIRST(α_k)中则M[A,a]=α_k；
  - 如果 a 不在FIRST(A)中，但是有某个k有$\varepsilon \in$FIRST(α_k)且a$\in$FOLLOW(A)，则M[A,a]=α_k。
  - 其它情况为语法错误。

- 在最左推导或最左扩展语法树过程中，根据当前待扩展变元A和当前输入符号a，查M[A,a]得出可应用的候选式。如此重复进行直到给定句子的推导或语法树形成为止，出错除外。

例：预测分析表构建

文法：

E→TE’

E’→+TE’|-TE’|ε

T →FT’

T’→*FT’|/FT’|ε

F→(E)| i

首符集：

E {(,i}

E’ {+,-,ε}

T {(,i}

T’ {*,/,ε}

F {(,i}

FOLLOW集：

E {#,)}

E’ {#,)}

T {+,-,#,)}

T’ {+,-,#,)}

F {*,/,+,-,#,)}

	+	−	*	/	(	)	i	#
E					TE’		TE’	
E’	+TE’	−TE’				ε		ε
T					FT’		FT’	
T’	ε	ε	*FT’	/FT’		ε		ε
F					(E)		i	

例：

栈	输入串	动作
#E	2-2*2#	E→TE’
#E’T	2-2*2#	T→FT’
#E’T’F	2-2*2#	F→i
#E’T’i	2-2*2#	match
#E’T’	-2*2#	T’→ε
#E’	-2*2#	E’→-TE’
#E’T-	-2*2#	match
#E’T	2*2#	T→FT’
#E’T’F	2*2#	F→i
#E’T’i	2*2#	match
#E’T’	*2#	T’→*FT’
#E’T’F*	*2#	match
#E’T’F	2#	F→i
#E’T’i	2#	match
#E’T’	#	T’→ε
#E’	#	E’→ε
#	#	accept

```
push(#);          //使用一个全局栈
push(E);
scan();
while(1){
  X=pop();
  if(X==#)if(X==tok)break;
                  else error();
  if(X∈T){if(X==tok)scan();
                  else error();
  }else{
    α=M[X,tok];
    if((X,α)∈P)          //是产生式
      pushlist(α);       //反序进栈
    else error();
  }else error();}        //end while
//分析成功
```

栈	输入串	动作
#E	2-2*2#	E→TE’
#E’T	2-2*2#	T→FT’
#E’T’F	2-2*2#	F→i
#E’T’i	2-2*2#	match
#E’T’	-2*2#	T’→ε
#E’	-2*2#	E’→-TE’
#E’T-	-2*2#	match
#E’T	2*2#	T→FT’
#E’T’F	2*2#	F→i
#E’T’i	2*2#	match
#E’T’	*2#	T’→*FT’
#E’T’F*	*2#	match
#E’T’F	2#	F→i
#E’T’i	2#	match
#E’T’	#	T’→ε
#E’	#	E’→ε
#	#	accept

	+	-	*	/	(	)	i	#
E					TE’		TE’	
E’	+TE’	-TE’				ε		ε
T					FT’		FT’	
T’	ε	ε	*FT’	/FT’		ε		ε
F					(E)		i	

```
push(#);
push(E); scan();
while(1){  X=pop();
 if(X==#)if(X==tok)break;
                    else error();
 if(X∈T){if(X==tok)scan();
                    else error();
 }else{
   α=M[X,tok];
   if((X,α)∈P)
     pushlist(α);
   else error();
 }else error();}   //end while
```

例：简单语言的LL(1)文法

P→ĎŠ	{int,d,print}	{#}
Ď→ε|D;Ď	{int,ε}	{d,print}
D→int d	{int}	{;}
Š→d=EŠ'|print dŠ'	{d,print}	{#}
Š'→;Š|ε	{;,ε}	{#}
E→iE'|dE'	{i,d}	{;,#}
E'→+iE'|*dE'|ε	{+,*,ε}	{;,#}

	+	*	i	d	;	int	print	=	#
P				ĎŠ		ĎŠ	ĎŠ		
Ď				ε		D;Ď	ε		
D						int d			
Š				d=EŠ'			print dŠ'		
Š'					;Š				ε
E			iE'	dE'					
E'	+iE'	*dE'			ε				ε

例：语法分析过程

栈	输入串	动作
#P	int x; x=2*x+1; print x#	P→ĎŠ
#ŠĎ	int x; x=2*x+1; print x#	Ď→D;Ď
#ŠĎ;D	int x; x=2*x+1; print x#	D→int d
#ŠĎ;d int	int x; x=2*x+1; print x#	匹配
#ŠĎ;d	x; x=2*x+1; print x#	匹配
#ŠĎ;	; x=2*x+1; print x#	匹配
#ŠĎ	x=2*x+1; print x#	Ď→ε
#Š	x=2*x+1; print x#	Š→d=EŠ’
#Š’E=d	x=2*x+1; print x#	匹配

	+	*	i	d	;	int	print	=	#
P				ĎŠ		ĎŠ	ĎŠ		
Ď				ε		D;Ď	ε		
D						int d			
Š				d=EŠ’			print dŠ’		
Š’					;Š				ε
E			iE’	dE’					
E’	+iE’	*dE’			ε				ε

分析过程（续）

栈	输入串	动作
#P	int x; x=2*x+1; print x#	P→ĎŠ
#ŠĎ	int x; x=2*x+1; print x#	Ď→D;Ď
#ŠĎ;D	int x; x=2*x+1; print x#	D→int d
#ŠĎ;d int	int x; x=2*x+1; print x#	匹配
#ŠĎ;d	x; x=2*x+1; print x#	匹配
#ŠĎ;	; x=2*x+1; print x#	匹配
#ŠĎ	x=2*x+1; print x#	Ď→ε
#Š	x=2*x+1; print x#	Š→d=EŠ’
#Š’E=d	x=2*x+1; print x#	匹配
#Š’E=	=2*x+1; print x#	匹配
#Š’E	2*x+1; print x#	E→iE’
#Š’E’i	2*x+1; print x#	匹配
#Š’E’	*x+1; print x#	E’→*dE’
#Š’E’d*	*x+1; print x#	匹配
#Š’E’d	x+1; print x#	匹配
#Š’E’	+1; print x#	E’→+iE’
#Š’E’i+	+1; print x#	匹配
#Š’E’i	1; print x#	匹配
#Š’E’	; print x#	E’→ε
#Š’	; print x#	Š’→;Š

分析过程（续）

栈	输入串	动作
#Š'E=	=2*x+1; print x#	匹配
#Š'E	2*x+1; print x#	E→iE'
#Š'E'i	2*x+1; print x#	匹配
#Š'E'	*x+1; print x#	E' →*dE'
#Š'E'd*	*x+1; print x#	匹配
#Š'E'd	x+1; print x#	匹配
#Š'E'	+1; print x#	E' →+iE'
#Š'E'i+	+1; print x#	匹配
#Š'E'i	1; print x#	匹配
#Š'E'	; print x#	E' →ε

	+	*	i	d	;	int	print	=	#
P				ĎŠ		ĎŠ	ĎŠ		
Ď				ε		D;Ď	ε		
D						int d			
Š				d=EŠ'			print dŠ'		
Š'					;Š				ε
E			iE'	dE'					
E'	+iE'	*dE'			ε				ε

分析过程（续）

栈	输入串	动作
#Š’	; print x#	Š’→;Š
#Š;	; print x#	匹配
#Š	print x#	Š→print d Š’
#Š’d print	print x#	匹配
#Š’d	x#	匹配
#Š’	#	Š’→ε
#	#	接受

	+	*	i	d	;	int	print	=	#
P				ĎŠ		ĎŠ	ĎŠ		
Ď				ε		D;Ď	ε		
D						int d			
Š				d=EŠ’			print dŠ’		
Š’					;Š				ε
E			iE’	dE’					
E’	+iE’	*dE’			ε				ε

例：

S→I|other	{if,other}	{#,else}
I→if(E)SL	{if}	{#,else}
L→else S|ε	{else,ε}	{#,else}
E→0|1	{0,1}	{)}

	if	other	else	0	1	#
S	S→I	S→other				
I	I→if(E)SL					
L			L→else S L→ε			L→ε
E				E→0	E→1	

Parsing Stack	Input	Action
#S	if(0)if(1)other else other#	S→I
#I	if(0)if(1)other else other#	I→if(E)SL
#LS)E(if	if(0)if(1)other else other#	match
# LS)E(	(0)if(1)other else other#	match
# LS)E	0)if(1)other else other#	E→0
# LS)0	0)if(1)other else other#	match
# LS)	)if(1)other else other#	match
# LS	if(1)other else other#	S→I
# LI	if(1)other else other#	I→if(E)SL
#LLS)E(if	if(1)other else other#	match
#LLS)E(	(1)other else other#	match
#LLS)E	1)other else other#	E→1
#LLS)1	1)other else other#	match
#LLS)	)other else other#	match
#LLS	other else other#	S→other
#LLother	other else other#	match
#LL	else other#	L→else S
#LSelse	else other#	match
#LS	other#	S→other
#Lother	other#	match
#L	#	L→ε
#	#	accept

小结：预测分析程序

- 预测分析程序对应于最右推导，栈保存着句型的一个不少变元的前缀，对于最后一个变元出现，它之后的终结符随后与输入符号逐一匹配掉，到这个变元升到栈顶的时候，就要归约了。当然，递归子程序对应最左推导，也有用武之地。
- 初始时栈底是#，栈顶是初始符号；当前输入符号为a；
- 分析表：二维数组M[A, a],a∈T,A∈V，元素为可用候选式。
- 重复进行如下动作：
  - 栈顶为变元A：弹出并压入M[A, a]诸符号；
  - 栈顶为终结符：弹出并与a进行匹配；
  - 栈顶为#且a=#，则分析过程终止，分析成功。
- 出错的情形
  - 终结符不匹配；
  - 表项M[A, a]是产生式以外情况。

自上而下分析中的错误处理

- 尽快找出错误；
- 一旦发现错误后，还能够继续分析下去；
- 尽量减少一个错误所导致的错误（error cascade problem）；
- 避免在错误上死循环。

- 出现错误的情形：
- 栈顶终结符与当前输入Token不匹配；
- 栈顶为变元A，当前输入Token为a，但M[A,a]为空。

错误处理的思想

- 栈顶为终结符
  - 弹出
- 栈顶为变元
  - 跳过输入Token流直至遇到同步符号集元素出现
- 同步符号集
  - FOLLOW(A)
  - FIRST(A)

小结

- 掌握消除左递归、消除回溯、LL(1)文法
- 掌握计算首符集、FOLLOW集
- 掌握LL(1)分析框架
- 掌握构造预测分析表
- 自上而下分析程序

- 作业：P141习题6.2-6.4
- 选作题：实现自上而下分析程序能够输出抽象语法树。