第四章 自上而下的分析

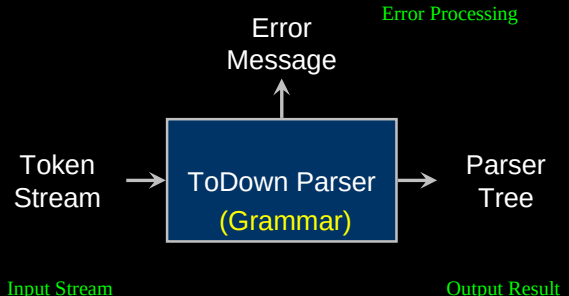
- 自上而下分析过程概述
  - 给定句子从文法开始符号经最左推导构造语法树
  - 文法为无二义性的上下文无关文法
- 无回溯的自上而下分析方法(LL(1)分析法)
  - 左递归导致推导过程死循环－消除文法左递归
  - 修改文法以避免回溯－修剪文法
    - FIRST(β), FOLLOW(A)
- 自上而下分析程序的构造
  - 递归下降分析程序
  - 预测分析程序

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树: *Start*

剩余串:
2-2*2
句型:
Start
所用产生式:

4.1 自上而下的分析过程概述



Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:
Start
Expr

剩余串:
2-2*2
句型:
Expr
所用产生式:
Start → *Expr*

程序的结构

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

S→E
E→E+T|E–T|T
T→T*i|T/i|i

G=(V_T,V_N,**P**,S)
Int, i ∈ V_T
Start,*Expr*,*Term*,S,E,T∈V_N
Start=S

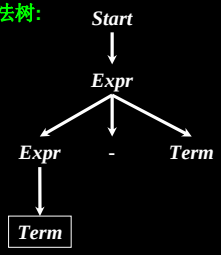
Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:
Start
Expr
Expr - *Term*

剩余串:
2-2*2
句型:
Expr- Term
所用产生式:
Expr → *Expr - Term*

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

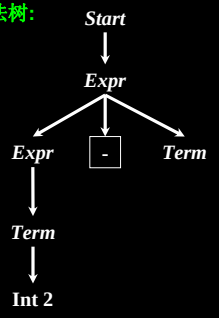
语法树:



剩余串:
2-2*2
句型:
Term - *Term*
所用产生式:
Expr → *Term*

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:

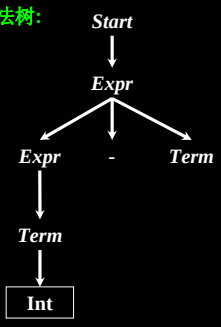


匹配
Token

剩余串:
-2*2
句型:
2 - *Term*

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

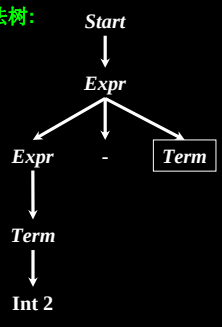
语法树:



剩余串:
2-2*2
句型:
Int - *Term*
所用产生式:
Term → Int

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

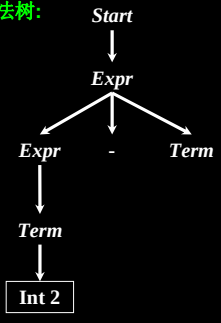
语法树:



剩余串:
2*2
句型:
2 - *Term*

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:

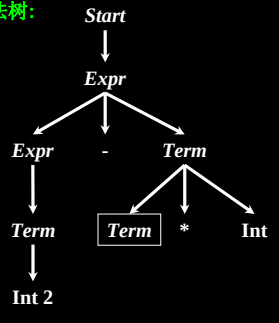


匹配
Token

剩余串:
2-2*2
句型:
2 - *Term*

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

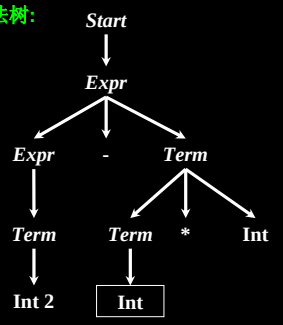
语法树:



剩余串:
2*2
句型:
2 – *Term* * Int
所用产生式:
Term → *Term* * Int

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

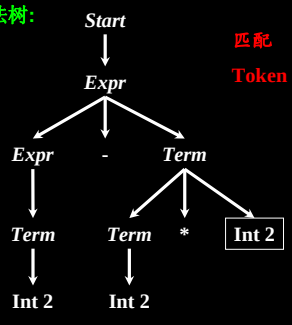
语法树:



剩余串:
2*2
句型:
2 – Int * Int
所用产生式:
Term → Int

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:

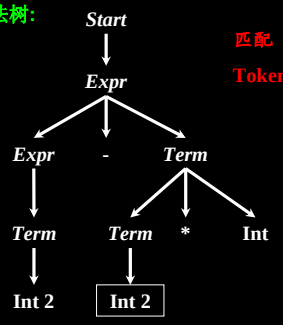


匹配
Token

剩余串:
2
句型:
2 – 2 * 2

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:

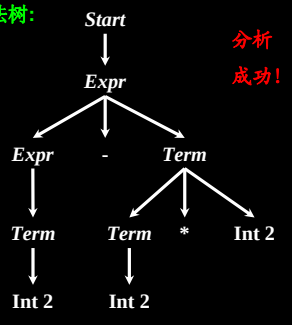


匹配
Token

剩余串:
2*2
句型:
2 – Int * Int

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:

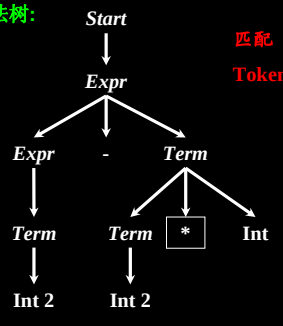


分析
成功!

剩余串:
句型:
2 – 2 * 2

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:



匹配
Token

剩余串:
*2
句型:
2 – 2 * Int

观察

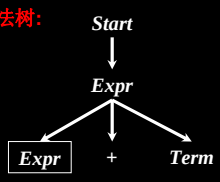
- 三个动作:
  - 应用产生式扩展语法树中的当前非终结符
  - 匹配当前的终结符（消耗掉输入流中一个Token）
  - 正确接受语法树（输出结果）或失败
- 语法分析过程表现为:
  - 先根遍历语法树

存在的问题

- 扩展当前结点时，可用的产生式规则有多个
- 分析过程中的回溯问题
  - 可看作一个搜索问题
  - 在每个选择的点处试探下个选择
  - 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
- 分析过程中的死循环问题

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:



剩余串:

2-2*2

句型:

Expr + *Term*

所用产生式:

Expr → *Expr* + *Term*

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:

剩余串:

2-2*2

句型:

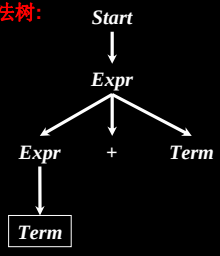
Start

所用产生式:

有回溯的例子...

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:



剩余串:

2-2*2

句型:

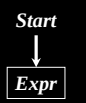
Term + *Term*

所用产生式:

Expr → *Term*

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:



剩余串:

2-2*2

句型:

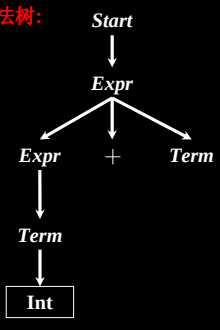
Expr

所用产生式:

Start → *Expr*

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:



匹配

Token

剩余串:

2-2*2

句型:

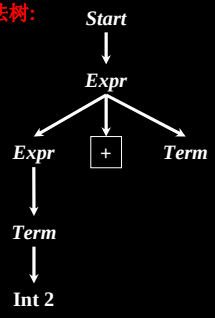
Int + *Term*

所用产生式:

Term → Int

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:

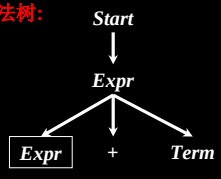


不匹配
Token

剩余串:
-2*2
句型:
2 + *Term*

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:

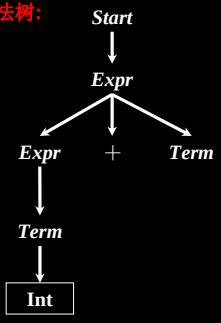


回溯

剩余串:
2-2*2
句型:
Expr + *Term*
所用产生式:
Expr → *Expr* + *Term*

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:

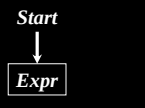


回溯

剩余串:
2-2*2
句型:
Int + *Term*
所用产生式:
Term → Int

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:

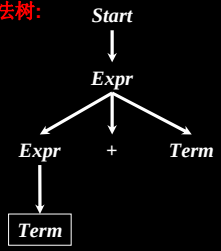


回溯

剩余串:
2-2*2
句型:
Expr
所用产生式:
Start → *Expr*

Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

语法树:

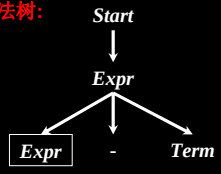


回溯

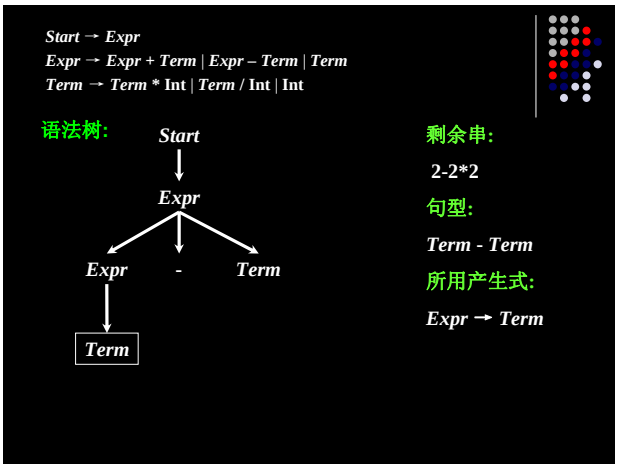
剩余串:
2-2*2
句型:
Term + *Term*
所用产生式:
Expr → *Term*

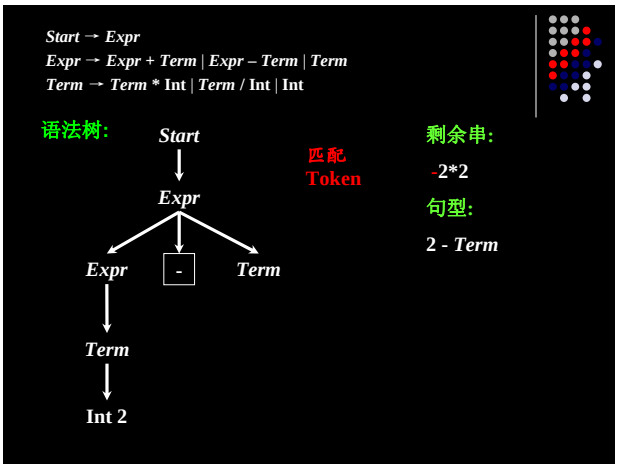
Start → *Expr*
Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*
Term → *Term* * Int | *Term* / Int | Int

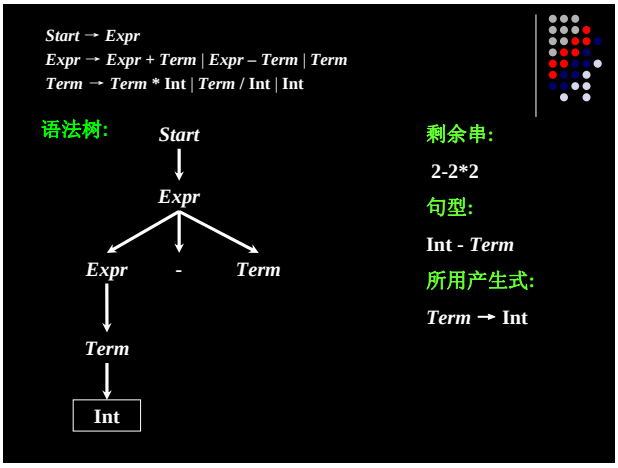
语法树:

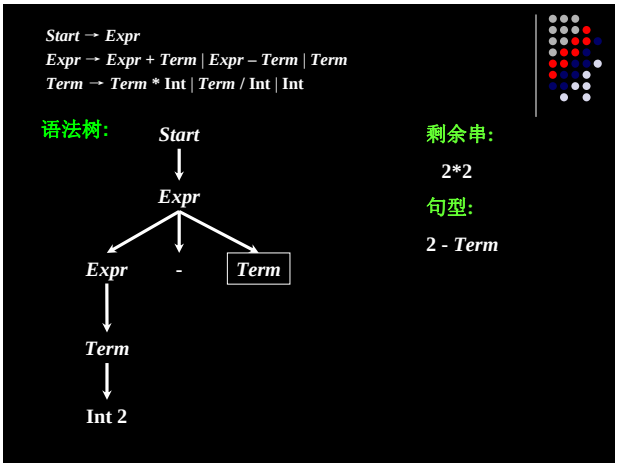


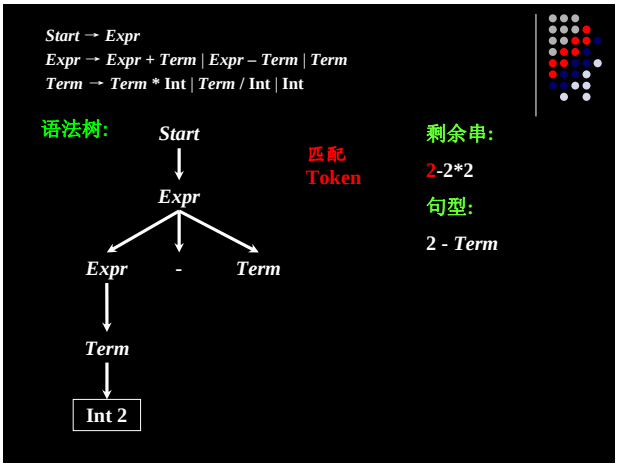
剩余串:
2-2*2
句型:
Expr - *Term*
所用产生式:
Expr → *Expr* - *Term*

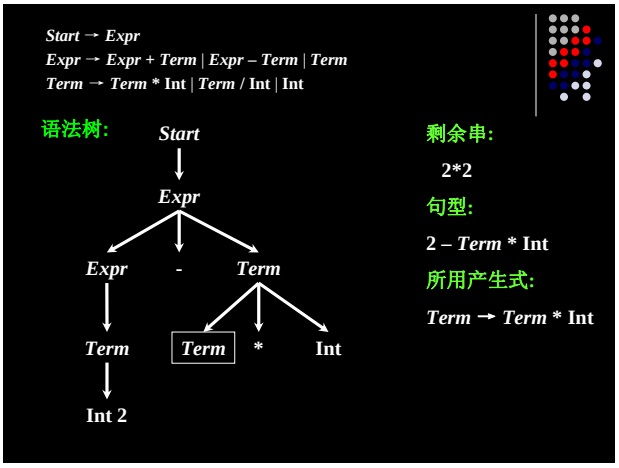

Start → Expr
Expr → Expr + Term | Expr – Term | Term
Term → Term * Int | Term / Int | Int
语法树:
Start
Expr
Expr
-
Term
Term
剩余串:
2-2*2
句型:
Term - Term
所用产生式:
Expr → Term

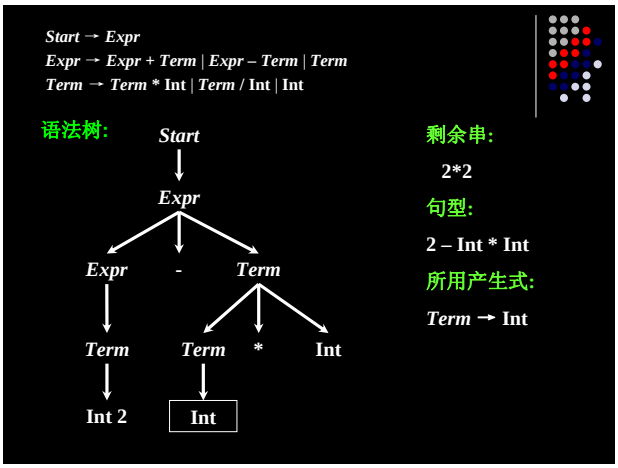

Start → Expr
Expr → Expr + Term | Expr – Term | Term
Term → Term * Int | Term / Int | Int
语法树:
Start
Expr
Expr
-
Term
Term
Int 2
匹配
Token
剩余串:
-2*2
句型:
2 - Term

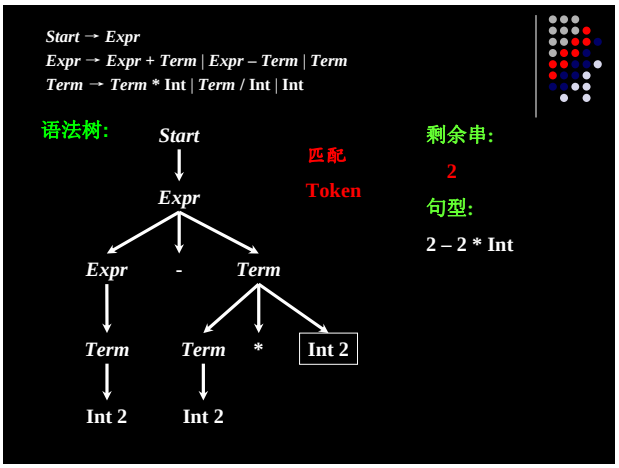

Start → Expr
Expr → Expr + Term | Expr – Term | Term
Term → Term * Int | Term / Int | Int
语法树:
Start
Expr
Expr
-
Term
Term
Int
剩余串:
2-2*2
句型:
Int - Term
所用产生式:
Term → Int

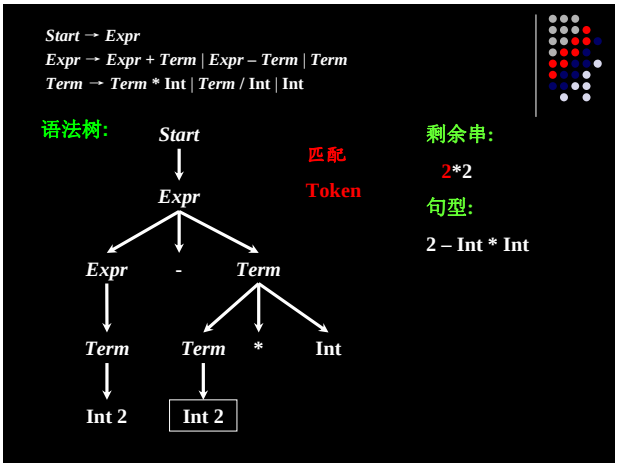

Start → Expr
Expr → Expr + Term | Expr – Term | Term
Term → Term * Int | Term / Int | Int
语法树:
Start
Expr
Expr
-
Term
Term
Int 2
剩余串:
2*2
句型:
2 - Term

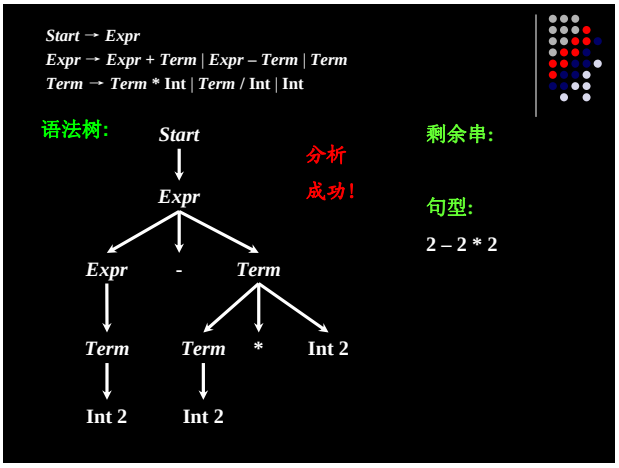

Start → Expr
Expr → Expr + Term | Expr – Term | Term
Term → Term * Int | Term / Int | Int
语法树:
Start
Expr
Expr
-
Term
Term
Int 2
匹配
Token
剩余串:
2-2*2
句型:
2 - Term

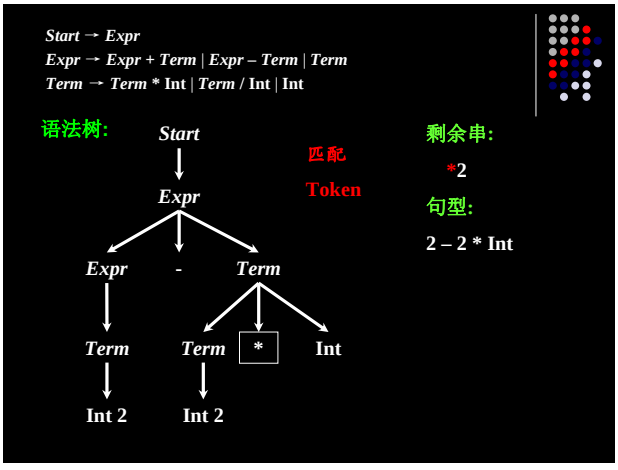

Start → Expr
Expr → Expr + Term | Expr – Term | Term
Term → Term * Int | Term / Int | Int
语法树:
Start
Expr
Expr
-
Term
Term
*
Int
Term
Int 2
剩余串:
2*2
句型:
2 – Term * Int
所用产生式:
Term → Term * Int

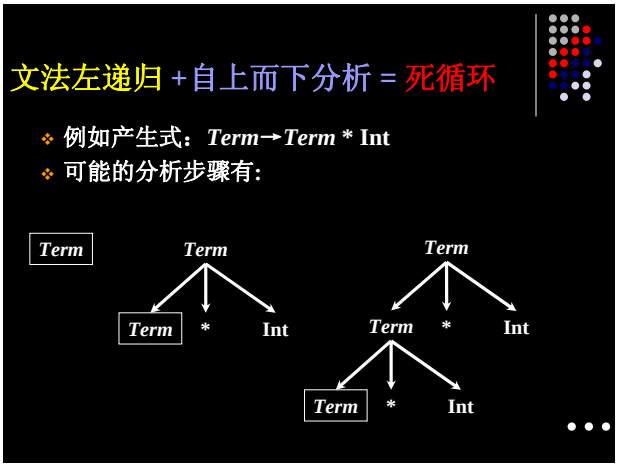

Start → Expr
Expr → Expr + Term | Expr – Term | Term
Term → Term * Int | Term / Int | Int
语法树:
Start
Expr
Expr - Term
Term Term * Int
Int 2 Int
剩余串:
2*2
句型:
2 – Int * Int
所用产生式:
Term → Int


Start → Expr
Expr → Expr + Term | Expr – Term | Term
Term → Term * Int | Term / Int | Int
语法树:
匹配
Token
Start
Expr
Expr - Term
Term Term * Int 2
Int 2 Int 2
剩余串:
2
句型:
2 – 2 * Int


Start → Expr
Expr → Expr + Term | Expr – Term | Term
Term → Term * Int | Term / Int | Int
语法树:
匹配
Token
Start
Expr
Expr - Term
Term Term * Int
Int 2 Int 2
剩余串:
2*2
句型:
2 – Int * Int


Start → Expr
Expr → Expr + Term | Expr – Term | Term
Term → Term * Int | Term / Int | Int
语法树:
分析
成功!
Start
Expr
Expr - Term
Term Term * Int 2
Int 2 Int 2
剩余串:
句型:
2 – 2 * 2


Start → Expr
Expr → Expr + Term | Expr – Term | Term
Term → Term * Int | Term / Int | Int
语法树:
匹配
Token
Start
Expr
Expr - Term
Term Term * Int
Int 2 Int 2
剩余串:
*2
句型:
2 – 2 * Int

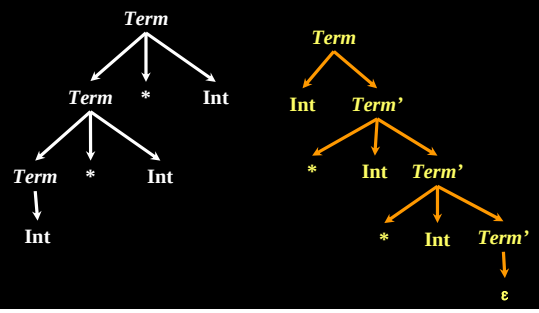

文法左递归 +自上而下分析 = 死循环
例如产生式：Term→Term * Int
可能的分析步骤有:
Term
Term
Term * Int
Term
Term * Int
Term * Int

4.2 无回溯的自上而下分析方法/ LL(1)分析法

- 对于语法分析而言，修剪文法以删除左递归

- *Term* → *Term* * Int
- *Term* → *Term* / Int
- *Term* → Int

- *Term* → Int *Term'*
- *Term'* → * Int *Term'*
- *Term'* → / Int *Term'*
- *Term'* → ε



4.2.1 消除文法的左递归性

- 直接左递归
  - *Term* → *Term* * Int
  - *Term* → Int
- 修剪为：
  - *Term* → Int *Term'*
  - *Term'* → * Int *Term'*
  - *Term'* → ε

- 直接左递归
  - $P \to P\alpha \mid \beta$
  - β 不以P开头
- 修剪为：
  - $P \to \beta P'$
  - $P' \to \alpha P' \mid \varepsilon$

❖不考虑无意义的规则： $P \to P$

②间接左递归及消除方法

-含间接左递归的文法

$$N \to A\alpha$$
$$A \to N\beta \mid \gamma$$
$$N \to N\beta\alpha \mid \gamma\alpha$$

代入法：将N的候选A α 中的 A 用A的两个候选式分别替换得到N的新候选式作为修剪文法。其中以A为左部的产生式规则不再有用，故不在修剪后的文法中出现。

-潜在左递归

$$N \to \alpha N$$

而 $\alpha \overset{*}{\Rightarrow} \varepsilon$

-故不考虑原文法中含 ε 产生式的情况。

①消除直接左递归

Original Grammar Fragment

- *Term* → *Term* * Int
- *Term* → *Term* / Int
- *Term* → Int

New Grammar Fragment

- *Term* → Int *Term'*
- *Term'* → * Int *Term'*
- *Term'* → / Int *Term'*
- *Term'* → ε

举例：代入法

- 例4.3
  - S->pNq
  - N->a|Bc| ε

- 将S的候选式pNq中的N用N的候选式代入后得：
  - S-> paq|pBcq|pq
  - N->a|Bc| ε

③消除文法左递归的算法

- 对文法G的所有非终结符进行排序
- 按上述顺序对每一个非终结符P_i依次执行:
  - for(j=1； j< i-1； j++)
    - 将P_j代入P_i的每个候选中（若可代入的话）；
  - 消除关于P_i的直接左递归；
- 化简上述所得文法(即删除无用非终结符)。

算法有效的条件：文法G不含P->P产生式和 ε 产生式

4.2.2 消除回溯

举例：消除下述文法的左递归

- S -> Sa | Ab | b | c
- A -> Bc | a
- B -> Sb | b

- S -> Aa | b
- A -> Ac | Sd | ε

对文法G的所有非终结符进行排序
按上述顺序对每一个非终结符P_i依次执行:
  for(j=1； j< i-1； j++)
    将P_j代入P_i的每个候选中；
  消除关于P_i的直接左递归；
化简上述所得文法。

4.2.2.1 产生回溯的原因分析

例：给定文法

- *Start* → *Expr*
- *Expr* → *Term Expr'*
- *Expr'* → + *Expr'*
- *Expr'* → -*Expr'*
- *Expr'* → ε
- *Term* → Int *Term'*
- *Term'* → * Int *Term'*
- *Term'* → / Int *Term'*
- *Term'* → ε

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

- A -> Ba | Aa | c
- B -> Bb | Ab | d

P_1=A
P_2=B

- A->BaA'|cA'
- A'->aA'| ε
- B->cA'bB'|dB'
- B'->bB'|aA'bB' | ε

- i=1,
- 消除A直接左递归
- A->BaA'|cA'
- A'->aA'| ε

- i=2, j=1
- A代入B中
- B->Bb|BaA'b|cA'b|d

- i=2, j=1
- 消除B 直接左递归
- B->cA'bB'|dB'
- B'->bB'|aA'bB' | ε

对文法G的所有非终结符进行排序
按上述顺序对每一个非终结符P_i依次执行:
  for(j=1； j< i-1； j++)
    将P_j代入P_i的每个候选中；
  消除关于P_i的直接左递归；
化简上述所得文法。

Choice Points

- 假设 *Term'* 是语法树中的当前结点
- 现有三个产生式可用
  - *Term'* → * Int *Term'*
  - *Term'* → / Int *Term'*
  - *Term'* → ε
- 根据下一个Token来决定
  - 若下一Token是 *, 用 *Term'* → * Int *Term'*
  - 若下一Token是 /, 用 *Term'* → / Int *Term'*
  - 其它情况？ 用 *Term'* → ε

问题：两个候选式有相同的前缀

- 例如文法

NT → if then
NT → if then else

- 假定 *NT* 是语法树中当前结点，if 是下一个Token
- 则不能够确定用哪个产生式

首符集（FIRST集）概念

- T∈ FIRST(α) if T can appear as the first symbol in a derivation starting from α
- FIRST(α)={a| α =>* a…， a∈V_T }
  - 特别地，若 α =>* ε ，则 ε ∈FIRST(α)

- Start with concept of *NT* deriving ε
- Two rules
  - *NT* → ε implies *NT* derives ε
  - *NT* → NT_1 |...| NT_n and for all $1 \leqslant i \leqslant n$ NT_i derives ε implies *NT* derives ε

解决办法: 提左公因子

- 让公共前缀只出现在单个产生式中

NT → if then *NT'*
NT' → else
NT' → ε

- 遇到下一个Token是if时，只有唯一的选择!

推导 ε 的不动点算法

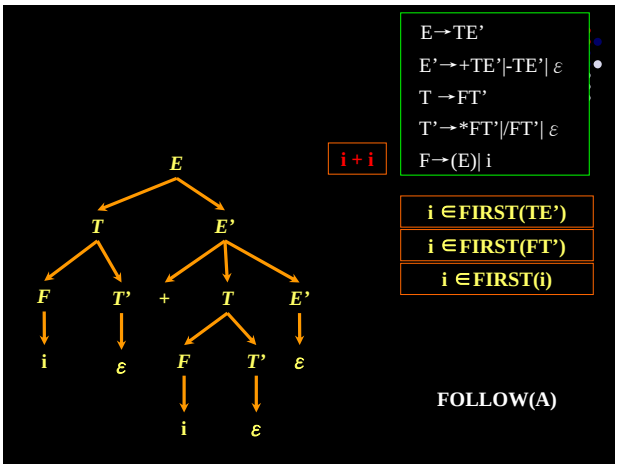
for all nonterminals *NT*
    set *NT* derives ε to be false
for all productions of the form *NT* → ε
    set *NT* derives ε to be true
while (some *NT* derives ε changed in last iteration)
  for all productions of the form *NT* → NT_1 |...| NT_n
    if (for all $1 \leqslant i \leqslant n$ NT_i derives ε)
      set *NT* derives ε to be true

另一种情况

- 候选式第一个符号为非终结符时，如：
  NT → NT_1 α_1
  NT → NT_2 α_2
- 选择哪一个取决于NT_1 和 NT_2 推导出来的串的第一个Token 跟输入流中的下一个Token的匹配情况
- 特别地，当 NT_1或NT_2 能推导出 ε 时
  - 这时必须基于 α_1 和 α_2进行判断

解决回溯问题的结论

文法G的产生式形如： A→ α_1 |…| α_m
则可以计算出：FIRST(α_i)，i=1,…,m
如果对于任意1<=i, j<=m都有
FIRST (α_i) ∩ FIRST(α_j) = φ
则意味着由A的每个候选式所能够推导出来的符号串的开头符号（属于V_T）均不相同
于是，对语法树中当前结点A，并且当前输入符号（Token）为a时只能够唯一地选择A的候选α_i，即a∈FIRST(α_i)



(1) FIRST(β)和FOLLOW(A)计算

给定文法G，$V= V_N \cup V_T$，$\alpha \in V^*$，$S, A \in V_N$

$FIRST(\alpha)=\{a \mid \alpha =>^* a\beta, a \in V_T, \beta \in V^*\}$

特别地，若 $\alpha =>^* \varepsilon$ ，则 $\varepsilon \in FIRST(\alpha)$

$FOLLOW(A)=\{a \mid S =>^* \alpha Aa\beta, a \in V_T, \alpha, \beta \in V^*\}$

4.2.2.2 消除回溯的方法

- 必要条件

文法任一个非终结符的首符集两两不相交。

- 方法：提取公共左因子

若 $A \to \delta\beta_1 | \cdots | \delta\beta_n | \gamma_1 | \cdots | \gamma_m$， $m, n \geq 1$

其中每个γ不以δ开头

那么可以等价地改写成：

$A \to \delta A' | \gamma_1 | \cdots | \gamma_m$

$A' \to \beta_1 | \cdots | \beta_n$

首符集的性质

- $T \in$ FIRST(T)
- FIRST (S) $\subseteq$ FIRST ($S\beta$)
- 若NT =>* ε 则 FIRST (β) $\subseteq$ FIRST ($NT\beta$)
- 产生式$NT \to S\beta$ 隐含着 FIRST ($S\beta$) $\subseteq$ FIRST (NT)

- 说明：T 是终结符, NT 是非终结符, S 或是终结符或是非终结符，β 是由终结符和非终结符构成的符号串

4.2.3 无回溯的分析步骤

- 当前Token为a，当前非终结符为A，且有规则 $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$
- 若a∈FIRST(α_i),采用规则$A \to \alpha_i$
- 若a不属于A的任何一个候选首符集，则：
  - 若 ε 属于某个FIRST(α_i)且a∈FOLLOW(A),则采用规则$A \to \varepsilon$
  - FOLLOW(A)=$\{a \mid S =>^* \alpha Aa\beta, a \in V_T, \alpha, \beta \in V^*\}$
  - 否则，选择别的非终结符规则，若仍不满足上述情况，则a的出现是一种语法错误。

推理法

Term' → * Int *Term'*
Term' → / Int *Term'*
Term' → ε

性质
1) $T \in$ First(T)
2) First(S) $\subseteq$ First($S\beta$)
3) NT =>* ε 隐含 First(β) $\subseteq$ First($NT\beta$)
4) $NT \to S\beta$ 隐含 First($S\beta$) $\subseteq$ First(NT)

Request: What is FIRST(*Term'*)?

约束条件
First(* *Num Term'*) $\subseteq$ First(*Term'*)
First(/ *Num Term'*) $\subseteq$ First(*Term'*)
First(*) $\subseteq$ First(* *Num Term'*)
First(/) $\subseteq$ First(/ *Num Term'*)
* ∈First(*)
/ ∈First(/)

约束条件传播算法

约束条件

First(* *Num Term'*) ⊆ First(*Term'*)
First(/ *Num Term'*) ⊆ First(*Term'*)
First(*) ⊆ First(* *Num Term'*)
First(/) ⊆ First(/ *Num Term'*)
* ∈First(*)
/ ∈First(/)

解
First(*Term'*)= {}
First(* *Num Term'*) = {}
First(/ *Num Term'*) = {}
First(*) = {}
First(/) = {}

初始化各个首符集为 {}
传播约束条件直到不动点（不再变化）

约束条件传播算法

约束条件

First(* *Num Term'*) ⊆ First(*Term'*)
First(/ *Num Term'*) ⊆ First(*Term'*)
First(*) ⊆ First(* *Num Term'*)
First(/) ⊆ First(/ *Num Term'*)
* ∈First(*)
/ ∈First(/)

解
First(*Term'*)= {*,/}
First(* *Num Term'*) = {*}
First(/ *Num Term'*) = {/}
First(*) = {*}
First(/) = {/}

初始化各个首符集为 {}
传播约束条件直到不动点（不再变化）

约束条件传播算法

约束条件

First(* *Num Term'*) ⊆ First(*Term'*)
First(/ *Num Term'*) ⊆ First(*Term'*)
First(*) ⊆ First(* *Num Term'*)
First(/) ⊆ First(/ *Num Term'*)
* ∈First(*)
/ ∈First(/)

解
First(*Term'*)= {}
First(* *Num Term'*) = {}
First(/ *Num Term'*) = {}
First(*) = {*}
First(/) = {/}

初始化各个首符集为 {}
传播约束条件直到不动点（不再变化）

约束条件传播算法

约束条件

First(* *Num Term'*) ⊆ First(*Term'*)
First(/ *Num Term'*) ⊆ First(*Term'*)
First(*) ⊆ First(* *Num Term'*)
First(/) ⊆ First(/ *Num Term'*)
* ∈First(*)
/ ∈First(/)

解
First(*Term'*)= { ε ,*,/}
First(* *Num Term'*) = {*}
First(/ *Num Term'*) = {/}
First(*) = {*}
First(/) = {/}

初始化各个首符集为 {}
传播约束条件直到不动点（不再变化）

最后再加一条：若 α =>* ε ，则 ε ∈FIRST(α)

约束条件传播算法

约束条件

First(* *Num Term'*) ⊆ First(*Term'*)
First(/ *Num Term'*) ⊆ First(*Term'*)
First(*) ⊆ First(* *Num Term'*)
First(/) ⊆ First(/ *Num Term'*)
* ∈First(*)
/ ∈First(/)

解
First(*Term'*)= {}
First(* *Num Term'*) = {*}
First(/ *Num Term'*) = {/}
First(*) = {*}
First(/) = {/}

初始化各个首符集为 {}
传播约束条件直到不动点（不再变化）

求所有非终结符FISRT集的迭代算法 ①构造FIRST(X),X∈$V_T \cup V_N$

- **连续使用以下规则，直至再无终结符，或 ε 加入任一FIRST集为止**
  - 若X∈V_T，则X属于FIRST(X)
  - 若X∈V_N，且有X->a α ，则a属于FIRST(X)
  - 若X∈V_N，若有 X-> ε ，则 ε 属于FIRST(X)
  - 若X∈V_N，有X->Y…，且Y∈V_N，则 FIRST(Y) \ { ε } 包含于FIRST(X)

②构造FIRST(α)，$\alpha = X_1X_2\ldots X_n$

- FIRST(X_1)中非 ε 元素加入FIRST(α)
- 若 ε ∈FIRST(X_1)，
  则FIRST(X_2)的所有非 ε 元素加入FIRST(α)
- 若 ε ∈FIRST(X_1)，ε ∈FIRST(X_2)，
  则FIRST(X_3)的所有非 ε 元素加入FIRST(α)
- 一般地，若 ε ∈ FIRST(X_i), i=1..m，m<n,
  FIRST(X_i)的所有非 ε 元素加入FIRST(α)，当m=n时
  则将 ε 加入FIRST(α)

文法中不含 ε 产生式时，构造所有非终结符A的FIRST集的简化算法

```
for all nonterminals A do First(A):={};
While there are changes to any First(A) do
  for each production A→X1X2…Xn do
    add First(X1) to First(A);
```

构造所有非终结符A的FIRST集的算法

```
for all nonterminals A do First(A):={};
While there are changes to any First(A) do
  for each production A → X1X2…Xn do
  k := 1; Continue := true;
  While Continue = true and k <= n do
    add First(Xk) \ { ε } to First(A)
    if ε is not in First(Xk) then Continue := false;
    k := k + 1;
  if Continue = true then add { ε } to First(A)
```

FOLLOW(A)定义

- 给定一个非终结符A，FOLLOW(A)由终结符组成，可能含#，
  - 若A为文法开始符号，则#属于FOLLOW(A)
  - 若有产生式 B→αAβ
    则 FIRST(β) \ { ε } ⊆FOLLOW(A)
  - 若有产生式 B→αAβ 且 ε ∈FIRST(β)
    则 FOLLOW(B) ⊆FOLLOW(A)

例：

S → *I* | other
I → if (*E*) *S L*
L → else *S* | ε
E → 0 | 1

Grammar rule	Pass I	Pass 2
S → *I*		First(S)={if,other}
S → other	First(S)={other}	
I → if (*E*) *S L*	First(I)={if}	
L → else *S*	First(L)={else}	
L → ε	First(L)={else, ε }	
E → 0	First(E)={0}	
E → 1	First(E)={1}	

构造FOLLOW集的算法

```
Follow(start-symbol) := {#};
for all nonterminals A ≠start-symbol do Follow(A):={};
While there are changes to any Follow sets do
  for each production A → X1X2…Xn do
    for each Xi that is a nonterminal do
      add First(Xi+1Xi+2…Xn) \ { ε } to Follow(Xi)
      (* Note: if i=n, then Xi+1Xi+2…Xn = ε *)
      if ε is in First(Xi+1Xi+2…Xn) then
        add Follow(A) to Follow(Xi)
```

4）例，已知文法G：

E → TE’　　T’→ *FT’|/FT’| ε

E’ → +TE’|-TE’| ε　　F →(E)| I

T → FT’

求G的每个非终结符A的 FOLLOW(A)集

FIRST集的构造：

FIRST(E) = FIRST(T) = FIRST(F) = { (, i }

FIRST(E’) = { + , ε }

FIRST(T’) = { * , ε }

Grammar rule	Pass I	Pass 2	Pass 3
E → TE’			First(E)={(,i}
T → FT’		First(T)={(,i}	
F →(E)	First(F)={(}		
F → i	First(F)={(,i}		
T’→ *FT’	First(T’)={*}		
T’→ /FT’	First(T’)={*,/}		
T’→ ε	First(T’)={*,/, ε }		
E’ → +TE’	First(E’)={+}		
E’ → -TE’	First(E’)={+,-}		
E’ → ε	First(E’)={+,-, ε }		

FOLLOW集的构造

① #属于FOLLOW(S)

②若存在B→ αAβ，则FIRST(β) \ { ε } 包含于FOLLOW(A)

由①得： FOLLOW(E) = { # }

由②得：

由E→TE’有 FIRST(E’) \ { ε } ⊆ FOLLOW(T)

则 FOLLOW(T) = { + }

由T→FT’有 FIRST(T’) \ { ε } ⊆ FOLLOW(F)

则 FOLLOW(F) = { * }

由F→(E)有 FIRST( )) ⊆ FOLLOW(E)

则 FOLLOW(E) = { # ,) }

FIRST(E’) = { + , ε }　　FIRST(T’) = { * , ε }

Grammar rule	Pass I	Pass 2	Pass 3
E → TE’	Follow(E)={#}	Follow(T)={),#}	
T → FT’		Follow(F)=?	
F →(E)	Follow(F)={),#}		
F → i			
T’→ *FT’		Follow(F)=?	
T’→ /FT’		Follow(F)=?	
T’→ ε			
E’ → +TE’		Follow(T)= {),#,+,-}	
E’ → -TE’		Follow(T)= {),#,+,-}	
E’ → ε			

FOLLOW集续

③若有B-> αAβ 且 ε ∈FIRST(β)

则 FOLLOW(B) ⊆ FOLLOW(A)

E→TE’有 FOLLOW(E) ⊆ FOLLOW(E’)

则FOLLOW(E’)={#,) }

E→TE’且 E’→ ε 有 FOLLOW(E) ⊆ FOLLOW(T)

则FOLLOW(T)={ +, #,) }

T→FT’有FOLLOW(T) ⊆ FOLLOW(T’)

则FOLLOW(T’)= { +, #,) }

T→FT’且T’→ ε 有FOLLOW(T)⊆FOLLOW(F)

则FOLLOW(F)= { *, +, #,) }

FOLLOW(T) = { + }
FOLLOW(F) = { * }
FOLLOW(E) = { # ,) }

最终构造结果

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

FIRST(E) = FIRST(T) = FIRST(F) = { (, i }
FIRST(E') = { + , ε }
FIRST(T') = { * , ε }
FOLLOW(E) = FOLLOW(E') = {) , # }
FOLLOW(T) = FOLLOW(T') = { + ,) , # }
FOLLOW(F) = { * , + ,) , # }

例4.4 文法 G：

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

每个非终结符所对应的递归子程序如下所示：

4.3递归下降分析程序

- **对文法的要求：**
  - 上下文无关文法、无二义性；
  - 不含左递归；
  - 每一个非终结符的任意两个候选式的首符集均不相交；
  - 对文法中的每一个非终结符A,若它存在某个候选首符集含 ε ,则有 FIRST(A)∩FOLLOW(A)= φ
- **LL(1)文法**
  - L：从左到右扫描输入流；
  - L：最左推导；
  - 1：向前查看一个Token

```
PROCEDURE E;
BEGIN
  T;E'
END;
```

E → TE'

```
PROCEDURE E';
BEGIN
IF SYM='+'THEN
 BEGIN
   ADVANCE;
   T;E'
 END
END;
```

E' → +TE' | ε

```
PROCEDURE T;
BEGIN
  F;T'
END;
```

T → FT'

Predictive Parsing + Hand Coding = Recursive Descent Parser

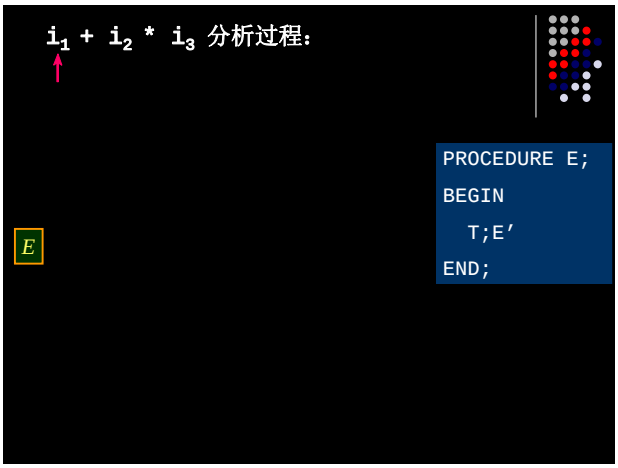
- One procedure per nonterminal
- That procedure examines the current input symbol
- Recursively calls procedures for RHS of chosen production
- Procedures return true if parse succeeded, false otherwise

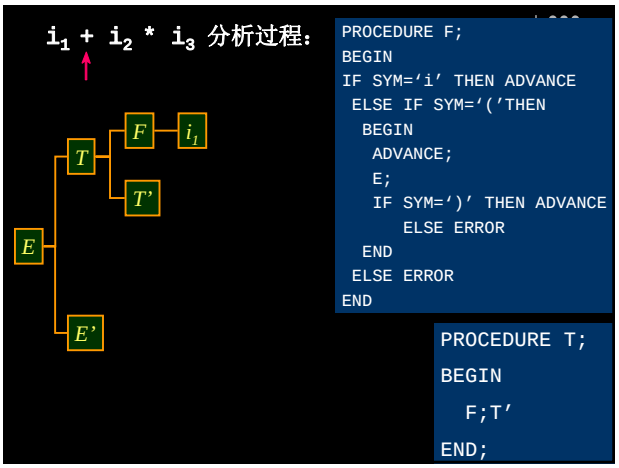
```
PROCEDURE T';
BEGIN
 IF SYM='*'THEN
 BEGIN
  ADVANCE;
  F;T'
 END
END
```

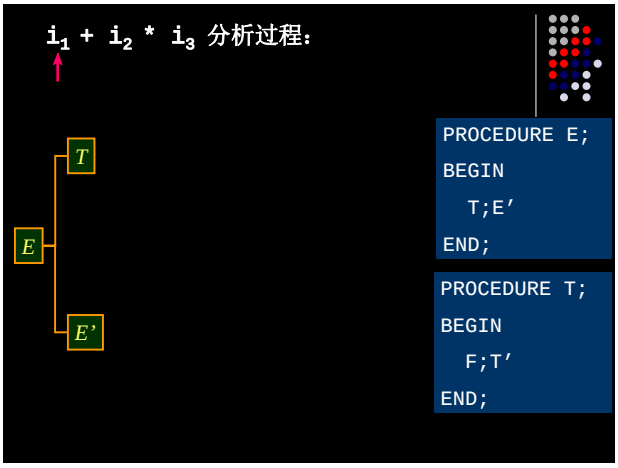
T' → *FT' | ε

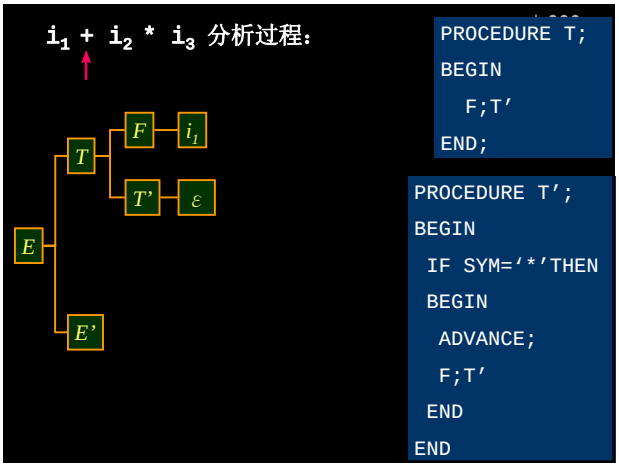
```
PROCEDURE F;
BEGIN
IF SYM='i' THEN ADVANCE
 ELSE IF SYM='('THEN
  BEGIN
   ADVANCE;
   E;
   IF SYM=')' THEN ADVANCE
      ELSE ERROR
  END
 ELSE ERROR
END
```

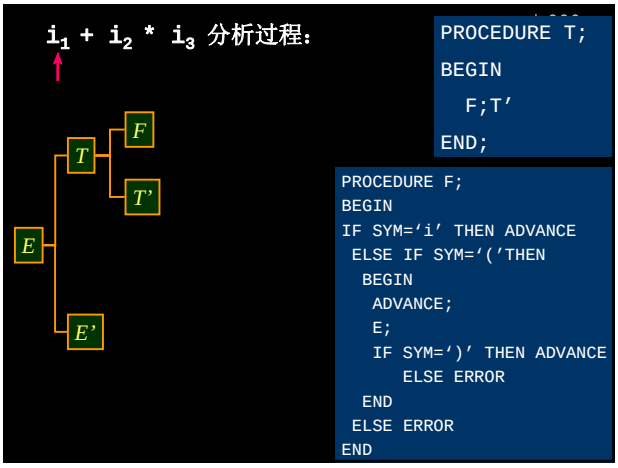
F → (E) | i

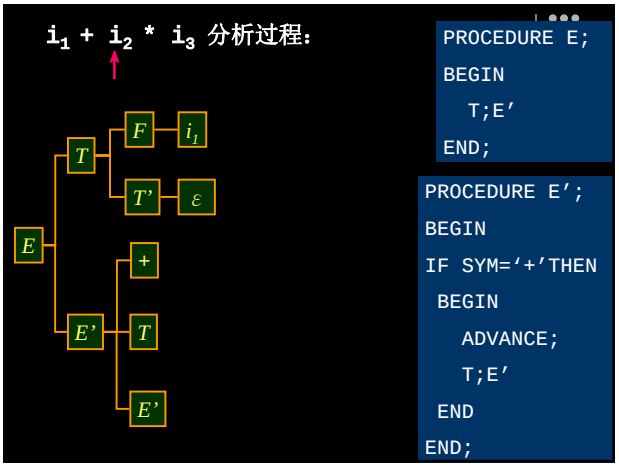

i1 + i2 * i3 分析过程：
E
PROCEDURE E;
BEGIN
T;E’
END;


i1 + i2 * i3 分析过程：
E
T
F
i1
T’
E’
PROCEDURE F;
BEGIN
IF SYM=‘i’ THEN ADVANCE
ELSE IF SYM=‘(’THEN
BEGIN
ADVANCE;
E;
IF SYM=‘)’ THEN ADVANCE
ELSE ERROR
END
ELSE ERROR
END
PROCEDURE T;
BEGIN
F;T’
END;

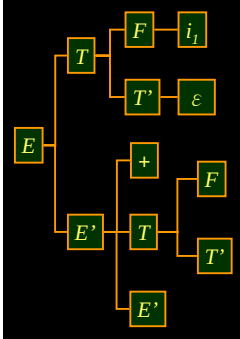

i1 + i2 * i3 分析过程：
T
E
E’
PROCEDURE E;
BEGIN
T;E’
END;
PROCEDURE T;
BEGIN
F;T’
END;


i1 + i2 * i3 分析过程：
E
T
F
i1
T’
ε
E’
PROCEDURE T;
BEGIN
F;T’
END;
PROCEDURE T’;
BEGIN
IF SYM=‘*’THEN
BEGIN
ADVANCE;
F;T’
END
END


i1 + i2 * i3 分析过程：
E
T
F
T’
E’
PROCEDURE T;
BEGIN
F;T’
END;
PROCEDURE F;
BEGIN
IF SYM=‘i’ THEN ADVANCE
ELSE IF SYM=‘(’THEN
BEGIN
ADVANCE;
E;
IF SYM=‘)’ THEN ADVANCE
ELSE ERROR
END
ELSE ERROR
END


i1 + i2 * i3 分析过程：
E
T
F
i1
T’
ε
+
E’
PROCEDURE E;
BEGIN
T;E’
END;
PROCEDURE E’;
BEGIN
IF SYM=‘+’THEN
BEGIN
ADVANCE;
T;E’
END
END;

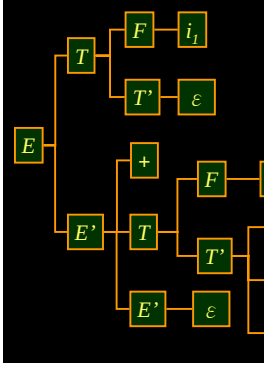
$i_1 + i_2 * i_3$ 分析过程：




```
PROCEDURE E';
BEGIN
IF SYM='+'THEN
 BEGIN
   ADVANCE;
   T;E'
 END
END;
```

```
PROCEDURE T;
BEGIN
  F;T'
END;
```

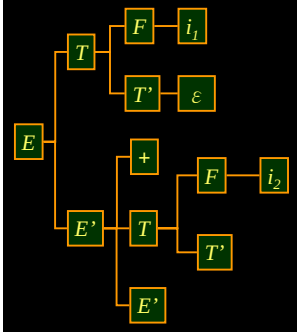
$i_1 + i_2 * i_3$ 分析过程：




```
PROCEDURE T;
BEGIN
  F;T'
END;
```

```
PROCEDURE T';
BEGIN
 IF SYM='*'THEN
 BEGIN
  ADVANCE;
  F;T'
 END
END
```

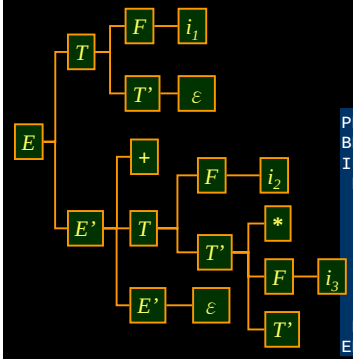
$i_1 + i_2 * i_3$ 分析过程：




```
PROCEDURE T;
BEGIN
  F;T'
END;
```

```
PROCEDURE F;
BEGIN
IF SYM='i' THEN ADVANCE
 ELSE IF SYM='('THEN
  BEGIN
   ADVANCE;
   E;
   IF SYM=')' THEN ADVANCE
      ELSE ERROR
  END
 ELSE ERROR
END
```

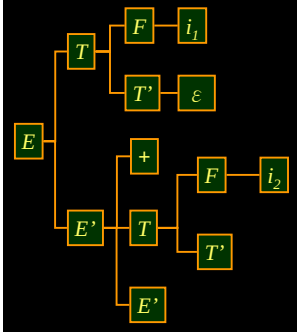
$i_1 + i_2 * i_3$ 分析过程：




```
PROCEDURE T';
BEGIN
 IF SYM='*'THEN
 BEGIN
  ADVANCE;
  F;T'
 END
END
```

```
PROCEDURE F;
BEGIN
IF SYM='i' THEN ADVANCE
 ELSE IF SYM='('THEN
  BEGIN
   ADVANCE;
   E;
   IF SYM=')' THEN ADVANCE
      ELSE ERROR
  END
 ELSE ERROR
END
```

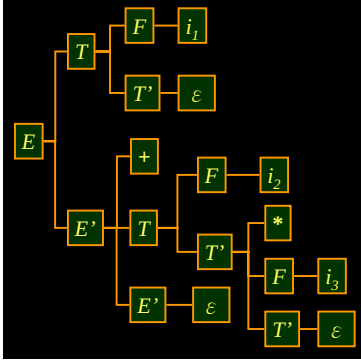
$i_1 + i_2 * i_3$ 分析过程：




```
PROCEDURE F;
BEGIN
IF SYM='i' THEN ADVANCE
 ELSE IF SYM='('THEN
  BEGIN
   ADVANCE;
   E;
   IF SYM=')' THEN ADVANCE
      ELSE ERROR
  END
 ELSE ERROR
END
```

```
PROCEDURE T;
BEGIN
  F;T'
END;
```

$i_1 + i_2 * i_3$ 分析过程：




```
PROCEDURE T';
BEGIN
 IF SYM='*'THEN
 BEGIN
  ADVANCE;
  F;T'
 END
END
```

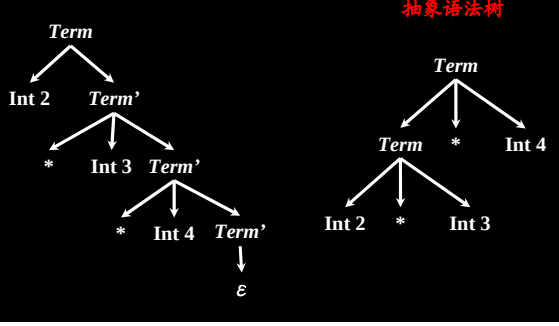
递归下降分析程序的实现

- 每一非终结符对应一个递归子程序；
- 递归子程序：是一个布尔过程，一旦发现它的某个候选式与输入串匹配，它就按此式扩充语法树，并返回true，指针移过已匹配子串。否则，返回false，保持原来的语法树和指针不变。

2*3*4的语法树

Term → Int *Term'*
Term' → * Int *Term'*
Term' → / Int *Term'*
Term' → ε

抽象语法树



例

Term → Int *Term'*
Term' → * Int *Term'*
Term' → / Int *Term'*
Term' → ε

- **Term()**

```
if(token=Int n){
  token = NextToken();
  return(TermPrime())
}else return(false)
```

- **TermPrime()**

```
if(token=*)||(token=/){
  token = NextToken();
  if (token=Int n) {
    token=NextToken();
    return(TermPrime())
  } else return(false)
}else return(true)
```

Building Parse Tree (Concrete Tree)

- **Term()**

```
if(token=Int n){
   oldToken=token; token=NextToken();
   node=TermPrime();
   if(node=NULL)return oldToken;
   else return(new TermNode(oldToken,node));
}else throw SyntaxError
```

- **TermPrime()**

```
if(token=*)||(token=/){
   first=token; next=NextToken();
   if(next=Int n){
        token=NextToken();
        return(new TermPrimeNode(first,next,TermPrime())
   }else throw SyntaxError
}else return(NULL)
```

Building A Parse Tree

- Have each procedure return the section of the parse tree for the part of the string it parsed
- Use exceptions to make code structure clean
- In general, can adjust parse algorithm to satisfy different goals – typically, produce abstract parse tree instead of concrete parse tree

Building Parse Tree (Concrete Tree)

- **Term()**

```
if(token=Int n){
   oldToken=token; token=NextToken();
   node=TermPrime();
   if(node=NULL)return oldToken;
   else return(new TermNode(oldToken,node));
}else throw SyntaxError
```

Int 2

- **TermPrime()**

```
if(token=*)||(token=/){
   first=token; next=NextToken();
   if(next=Int n){
        token=NextToken();
        return(new TermPrimeNode(first,next,TermPrime())
   }else throw SyntaxError
}else return(NULL)
```

2*3*4

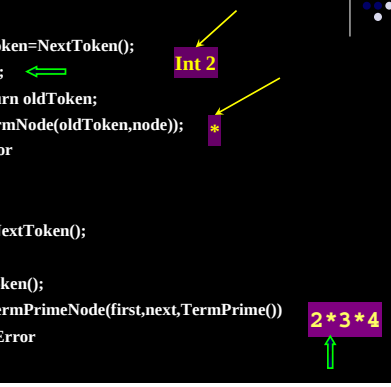
Building Parse Tree (Concrete Tree)

- Term()

```
if(token=Int n){
    oldToken=token; token=NextToken();
    node=TermPrime();
    if(node=NULL)return oldToken;
    else return(new TermNode(oldToken,node));
}else throw SyntaxError
```

- TermPrime()

```
if(token=*)||(token=/){
    first=token; next=NextToken();
    if(next=Int n){
        token=NextToken();
        return(new TermPrimeNode(first,next,TermPrime())
    }else throw SyntaxError
}else return(NULL)
```



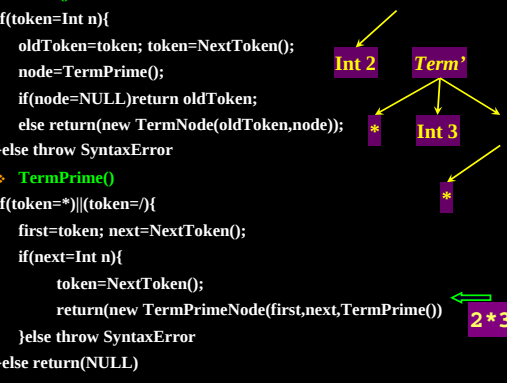
Building Parse Tree (Concrete Tree)

- Term()

```
if(token=Int n){
    oldToken=token; token=NextToken();
    node=TermPrime();
    if(node=NULL)return oldToken;
    else return(new TermNode(oldToken,node));
}else throw SyntaxError
```

- TermPrime()

```
if(token=*)||(token=/){
    first=token; next=NextToken();
    if(next=Int n){
        token=NextToken();
        return(new TermPrimeNode(first,next,TermPrime())
    }else throw SyntaxError
}else return(NULL)
```



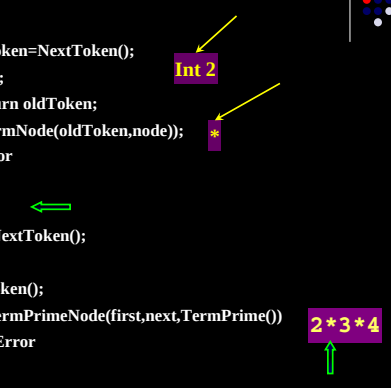
Building Parse Tree (Concrete Tree)

- Term()

```
if(token=Int n){
    oldToken=token; token=NextToken();
    node=TermPrime();
    if(node=NULL)return oldToken;
    else return(new TermNode(oldToken,node));
}else throw SyntaxError
```

- TermPrime()

```
if(token=*)||(token=/){
    first=token; next=NextToken();
    if(next=Int n){
        token=NextToken();
        return(new TermPrimeNode(first,next,TermPrime())
    }else throw SyntaxError
}else return(NULL)
```



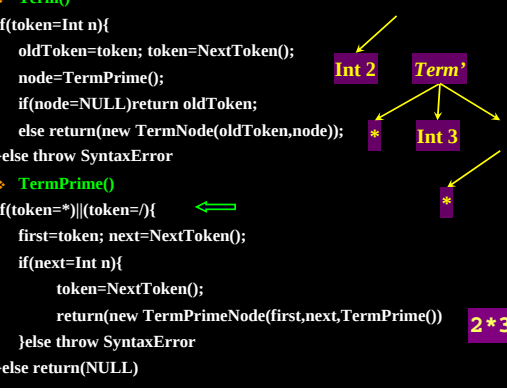
Building Parse Tree (Concrete Tree)

- Term()

```
if(token=Int n){
    oldToken=token; token=NextToken();
    node=TermPrime();
    if(node=NULL)return oldToken;
    else return(new TermNode(oldToken,node));
}else throw SyntaxError
```

- TermPrime()

```
if(token=*)||(token=/){
    first=token; next=NextToken();
    if(next=Int n){
        token=NextToken();
        return(new TermPrimeNode(first,next,TermPrime())
    }else throw SyntaxError
}else return(NULL)
```



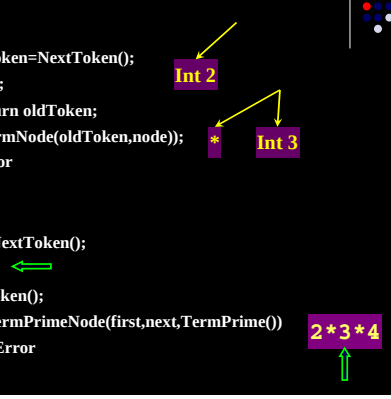
Building Parse Tree (Concrete Tree)

- Term()

```
if(token=Int n){
    oldToken=token; token=NextToken();
    node=TermPrime();
    if(node=NULL)return oldToken;
    else return(new TermNode(oldToken,node));
}else throw SyntaxError
```

- TermPrime()

```
if(token=*)||(token=/){
    first=token; next=NextToken();
    if(next=Int n){
        token=NextToken();
        return(new TermPrimeNode(first,next,TermPrime())
    }else throw SyntaxError
}else return(NULL)
```



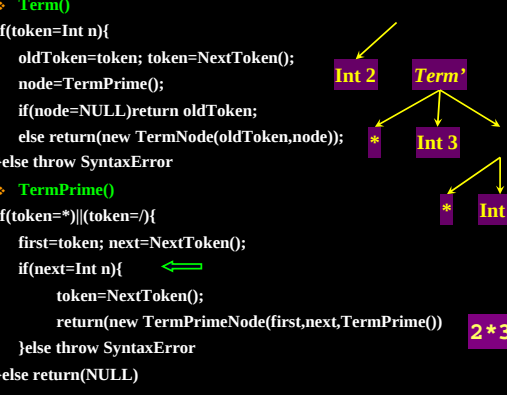
Building Parse Tree (Concrete Tree)

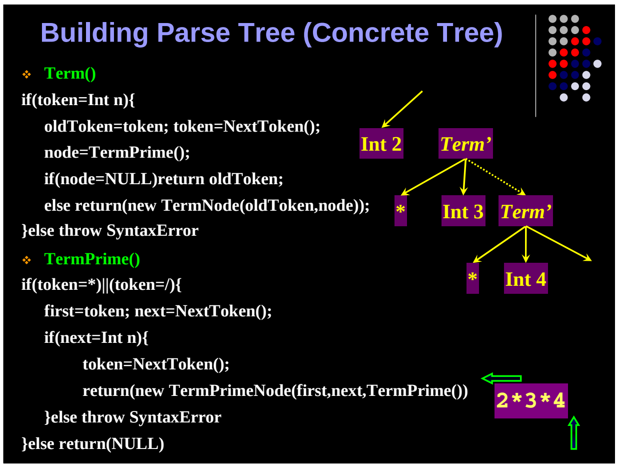
- Term()

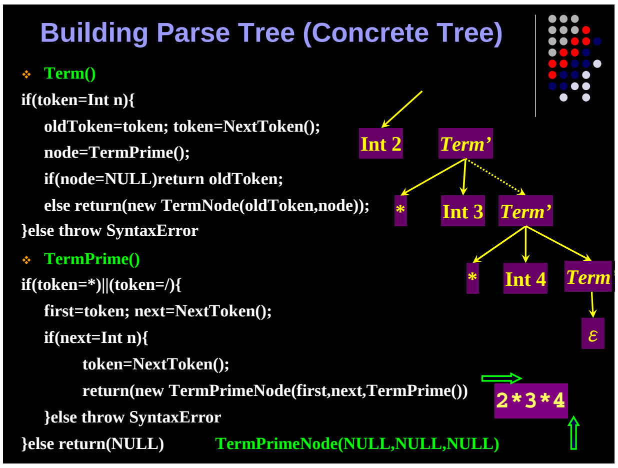
```
if(token=Int n){
    oldToken=token; token=NextToken();
    node=TermPrime();
    if(node=NULL)return oldToken;
    else return(new TermNode(oldToken,node));
}else throw SyntaxError
```

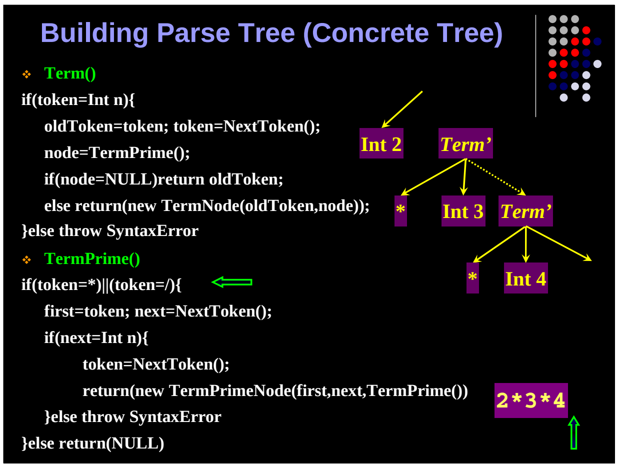
- TermPrime()

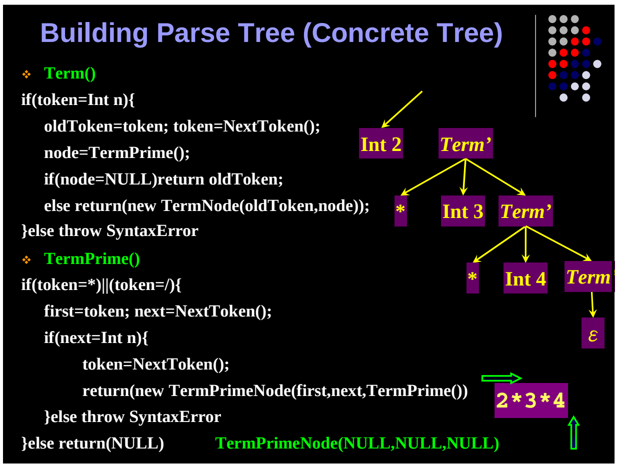
```
if(token=*)||(token=/){
    first=token; next=NextToken();
    if(next=Int n){
        token=NextToken();
        return(new TermPrimeNode(first,next,TermPrime())
    }else throw SyntaxError
}else return(NULL)
```

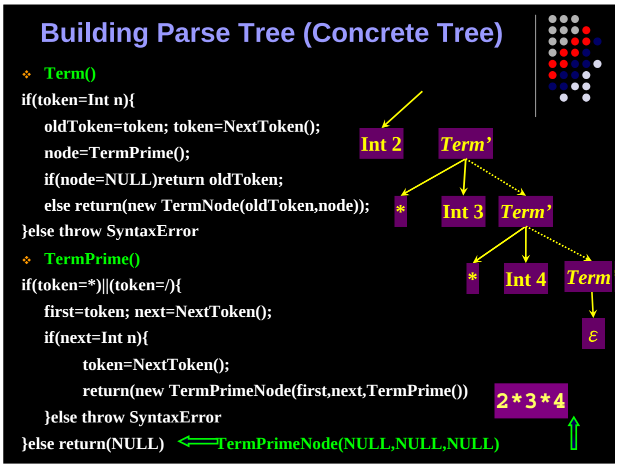


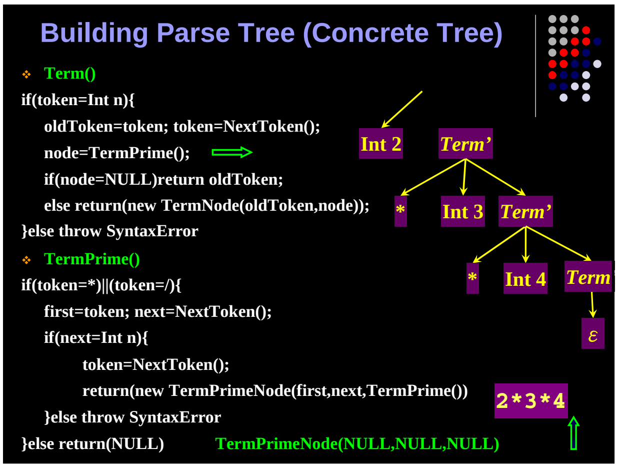

Building Parse Tree (Concrete Tree)
Term()
if(token=Int n){
oldToken=token; token=NextToken();
node=TermPrime();
if(node=NULL)return oldToken;
else return(new TermNode(oldToken,node));
}else throw SyntaxError
TermPrime()
if(token=*)||(token=/){
first=token; next=NextToken();
if(next=Int n){
token=NextToken();
return(new TermPrimeNode(first,next,TermPrime())
}else throw SyntaxError
}else return(NULL)
Int 2
Term'
*
Int 3
Term'
*
Int 4
2*3*4


Building Parse Tree (Concrete Tree)
Term()
if(token=Int n){
oldToken=token; token=NextToken();
node=TermPrime();
if(node=NULL)return oldToken;
else return(new TermNode(oldToken,node));
}else throw SyntaxError
TermPrime()
if(token=*)||(token=/){
first=token; next=NextToken();
if(next=Int n){
token=NextToken();
return(new TermPrimeNode(first,next,TermPrime())
}else throw SyntaxError
}else return(NULL)
TermPrimeNode(NULL,NULL,NULL)
Int 2
Term'
*
Int 3
Term'
*
Int 4
Term
ε
2*3*4


Building Parse Tree (Concrete Tree)
Term()
if(token=Int n){
oldToken=token; token=NextToken();
node=TermPrime();
if(node=NULL)return oldToken;
else return(new TermNode(oldToken,node));
}else throw SyntaxError
TermPrime()
if(token=*)||(token=/){
first=token; next=NextToken();
if(next=Int n){
token=NextToken();
return(new TermPrimeNode(first,next,TermPrime())
}else throw SyntaxError
}else return(NULL)
Int 2
Term'
*
Int 3
Term'
*
Int 4
2*3*4


Building Parse Tree (Concrete Tree)
Term()
if(token=Int n){
oldToken=token; token=NextToken();
node=TermPrime();
if(node=NULL)return oldToken;
else return(new TermNode(oldToken,node));
}else throw SyntaxError
TermPrime()
if(token=*)||(token=/){
first=token; next=NextToken();
if(next=Int n){
token=NextToken();
return(new TermPrimeNode(first,next,TermPrime())
}else throw SyntaxError
}else return(NULL)
TermPrimeNode(NULL,NULL,NULL)
Int 2
Term'
*
Int 3
Term'
*
Int 4
Term
ε
2*3*4


Building Parse Tree (Concrete Tree)
Term()
if(token=Int n){
oldToken=token; token=NextToken();
node=TermPrime();
if(node=NULL)return oldToken;
else return(new TermNode(oldToken,node));
}else throw SyntaxError
TermPrime()
if(token=*)||(token=/){
first=token; next=NextToken();
if(next=Int n){
token=NextToken();
return(new TermPrimeNode(first,next,TermPrime())
}else throw SyntaxError
}else return(NULL)
TermPrimeNode(NULL,NULL,NULL)
Int 2
Term'
*
Int 3
Term'
*
Int 4
Term
ε
2*3*4


Building Parse Tree (Concrete Tree)
Term()
if(token=Int n){
oldToken=token; token=NextToken();
node=TermPrime();
if(node=NULL)return oldToken;
else return(new TermNode(oldToken,node));
}else throw SyntaxError
TermPrime()
if(token=*)||(token=/){
first=token; next=NextToken();
if(next=Int n){
token=NextToken();
return(new TermPrimeNode(first,next,TermPrime())
}else throw SyntaxError
}else return(NULL)
TermPrimeNode(NULL,NULL,NULL)
Int 2
Term'
*
Int 3
Term'
*
Int 4
Term
ε
2*3*4

Building Parse Tree (Concrete Tree)

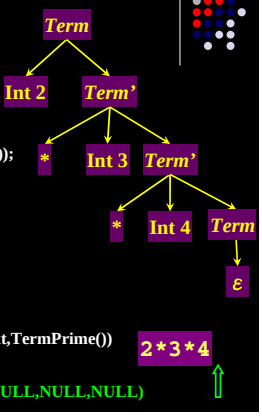
- Term()

```
if(token=Int n){
    oldToken=token; token=NextToken();
    node=TermPrime();
    if(node=NULL)return oldToken;
    else return(new TermNode(oldToken,node));
}else throw SyntaxError
```

- TermPrime()

```
if(token=*)||(token=/){
    first=token; next=NextToken();
    if(next=Int n){
        token=NextToken();
        return(new TermPrimeNode(first,next,TermPrime())
    }else throw SyntaxError
}else return(NULL)
```

TermPrimeNode(NULL,NULL,NULL)



Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
   token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

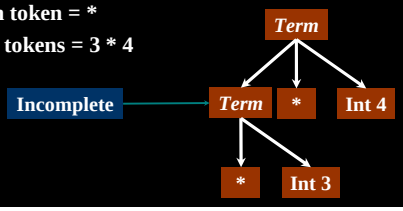
- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
        node.right=token; token=NextToken();
        if(token=*)||(token=/){ oldNode = node;
            node=new TermNode(oldNode,NULL,NULL);
            TermPrime(node); }
        return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```

直接产生抽象语法树

- TermPrime构造不完全树
  - 缺少最左孩子
  - 返回根和一个不完全的结点
- (root, incomplete) = TermPrime()
  - Called with token = *
  - Remaining tokens = 3 * 4

Term → Int *Term'*
Term' → * Int *Term'*
Term' → / Int *Term'*
Term' → ε



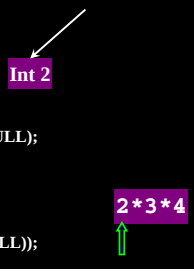
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
   token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
        node.right=token; token=NextToken();
        if(token=*)||(token=/){ oldNode = node;
            node=new TermNode(oldNode,NULL,NULL);
            TermPrime(node); }
        return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



Building Parse Tree (Concrete Tree)

- Term()

```
if(token=Int n){
    oldToken=token; token=NextToken();
    node=TermPrime();
    if(node=NULL)return oldToken;
    else return(new TermNode(oldToken,node));
}else throw SyntaxError
```

- TermPrime()

```
if(token=*)||(token=/){
    first=token; next=NextToken();
    if(next=Int n){
        token=NextToken();
        return(new TermPrimeNode(first,next,TermPrime())
    }else throw SyntaxError
}else return(NULL)
```

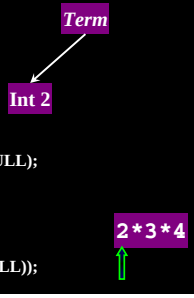
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
   token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
        node.right=token; token=NextToken();
        if(token=*)||(token=/){ oldNode = node;
            node=new TermNode(oldNode,NULL,NULL);
            TermPrime(node); }
        return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



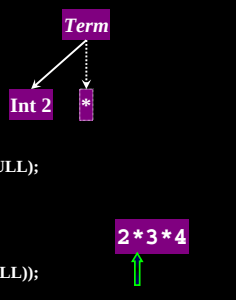
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
  token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
       node.right=token; token=NextToken();
      if(token=*)||(token=/){ oldNode = node;
          node=new TermNode(oldNode,NULL,NULL);
          TermPrime(node); }
      return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



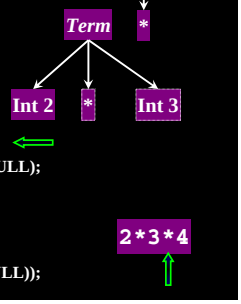
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
  token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
       node.right=token; token=NextToken();
      if(token=*)||(token=/){ oldNode = node;
          node=new TermNode(oldNode,NULL,NULL);
          TermPrime(node); }
      return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



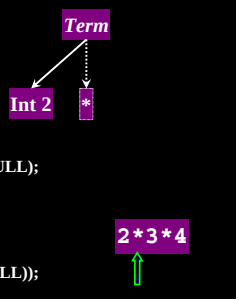
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
  token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
       node.right=token; token=NextToken();
      if(token=*)||(token=/){ oldNode = node;
          node=new TermNode(oldNode,NULL,NULL);
          TermPrime(node); }
      return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



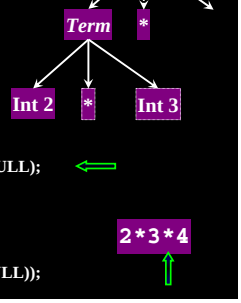
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
  token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
       node.right=token; token=NextToken();
      if(token=*)||(token=/){ oldNode = node;
          node=new TermNode(oldNode,NULL,NULL);
          TermPrime(node); }
      return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



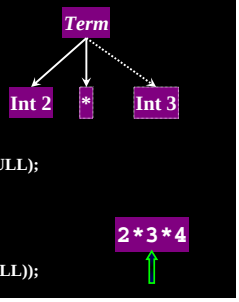
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
  token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
       node.right=token; token=NextToken();
      if(token=*)||(token=/){ oldNode = node;
          node=new TermNode(oldNode,NULL,NULL);
          TermPrime(node); }
      return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



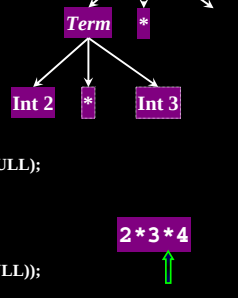
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
  token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
       node.right=token; token=NextToken();
      if(token=*)||(token=/){ oldNode = node;
          node=new TermNode(oldNode,NULL,NULL);
          TermPrime(node); }
      return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



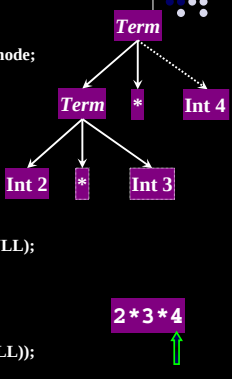
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
  token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
       node.right=token; token=NextToken();
      if(token=*)||(token=/){ oldNode = node;
          node=new TermNode(oldNode,NULL,NULL);
          TermPrime(node); }
      return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



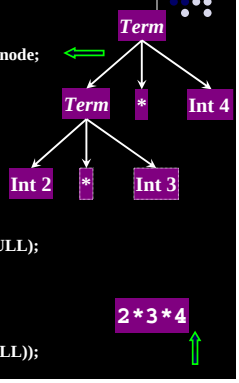
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
  token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
       node.right=token; token=NextToken();
      if(token=*)||(token=/){ oldNode = node;
          node=new TermNode(oldNode,NULL,NULL);
          TermPrime(node); }
      return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



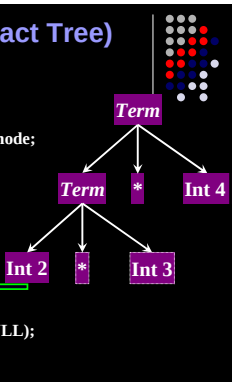
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
  token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
       node.right=token; token=NextToken();
      if(token=*)||(token=/){ oldNode = node;
          node=new TermNode(oldNode,NULL,NULL);
          TermPrime(node); }
      return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



程序实现小结

- 有 ε，自动匹配，认为不会失败
- 出错处理
- 过程Term()和TermPrime()
- NextToken()
- TermNode()和TermPrimeNode()

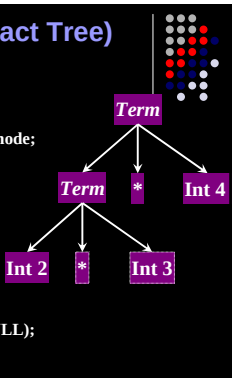
Building Parse Tree (Abstract Tree)

- Term()

```
if(token=Int n){
    node=new TermNode(token,NULL,NULL);
  token=NextToken(); TermPrime(node); return node;
}else throw SyntaxError
```

- TermPrime(node)

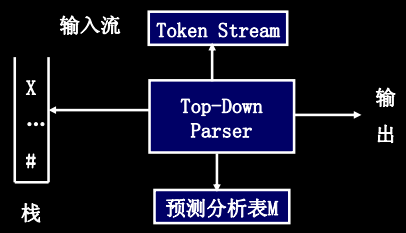
```
if(token=*)||(token=/){
    node.op=token; token=NextToken();
    if(token=Int n){
       node.right=token; token=NextToken();
      if(token=*)||(token=/){ oldNode = node;
          node=new TermNode(oldNode,NULL,NULL);
          TermPrime(node); }
      return node;
    }else throw SyntaxError
}else return(new TermNode(oldToken,NULL,NULL));
```



4.4 预测分析程序

- 构造预测分析表
- 采用预测分析表和栈实现递归下降分析程序
- 这种方法实现的语法分析程序又叫预测分析程序

4.5.1 预测分析程序工作过程



分析表格式

E→TE’
E’→+TE’| ε
T→FT’
T’→*FT’| ε
F→(E)| i

	id	+	*	(	)	#
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→ ε	E’→ ε
T	T→FT’			T→FT’		
T’		T’→ ε	T’→*FT’		T’→ ε	T’→ ε
F	F→id			F→(E)		

程序构成

- 输入流：含有要分析的Token串，右端有#
- 栈：栈底是#，栈内是一系列文法符号。
  开始时，#和S先进栈
- 分析表：二维数组M[A，a]，其中a∈V_T；A∈V_N
- 动作：根据分析表内元素做规定的推导（也可包括语义动作）

id+id*id分析过程示例

	id	+	*	(	)	#
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→ ε	E’→ ε
T	T→FT’			T→FT’		
T’		T’→ ε	T’→*FT’		T’→ ε	T’→ ε
F	F→id			F→(E)		

输出：

开始：

E
#

id + id * id#

预测分析程序的动作

- 设X为当前栈顶符号， a为当前Token，则
- 若X=a=#，分析停止，宣告成功地完成分析；
- 若X=a≠#，则X弹出栈，输入指针移动一步；
- 若X∈V_N，则去查分析表M的元素M[X，a]，该元素或为以X为左部的产生式，或为出错信息。
  - 如：M[X, a]为产生式X→UVW, 则X出栈，且W、V、U依次进栈；动作包括扩展语法树或输出产生式等。
  - 若M[X, a]为错误信息，则调用错误处理程序。

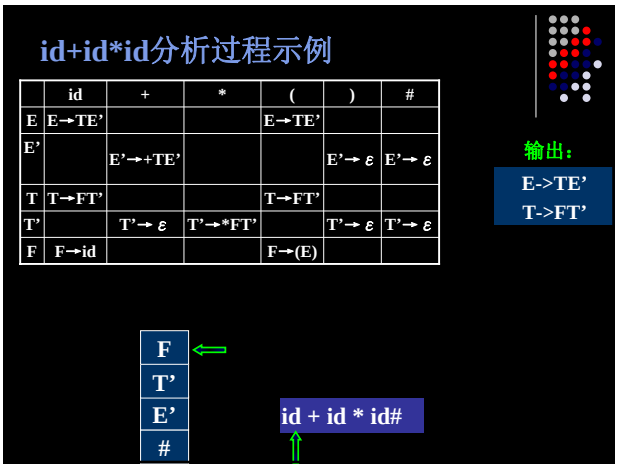
id+id*id分析过程示例

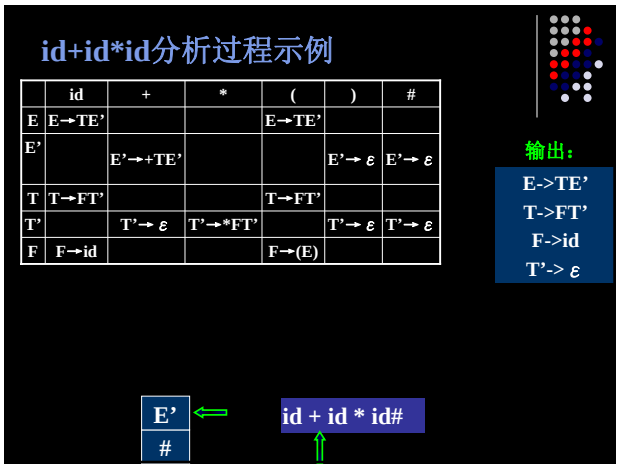
	id	+	*	(	)	#
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→ ε	E’→ ε
T	T→FT’			T→FT’		
T’		T’→ ε	T’→*FT’		T’→ ε	T’→ ε
F	F→id			F→(E)		

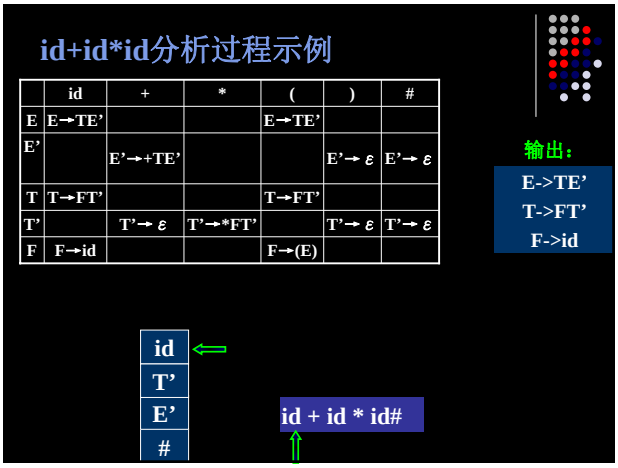
输出：
E->TE’

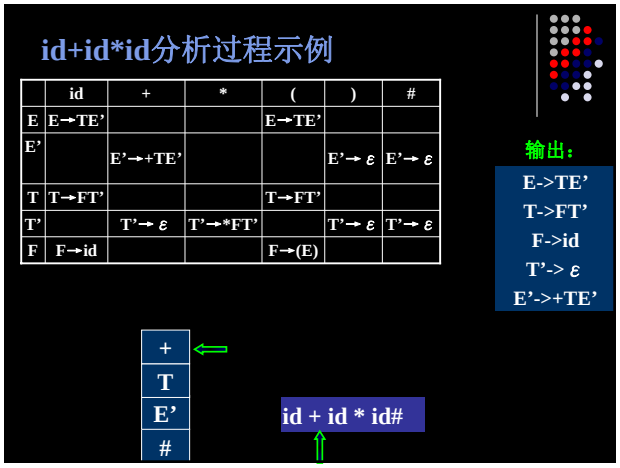
T
E’
#

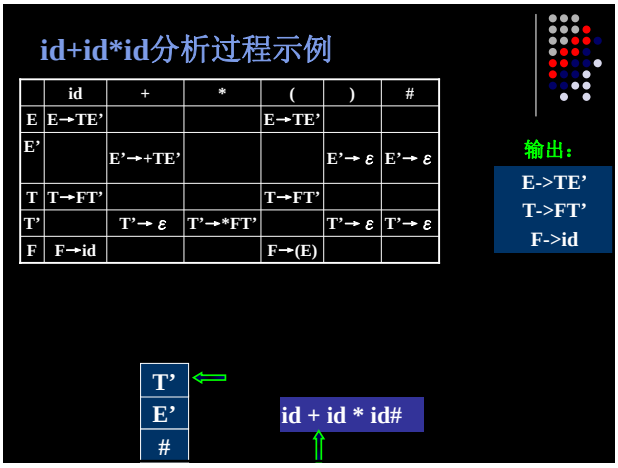
id + id * id#

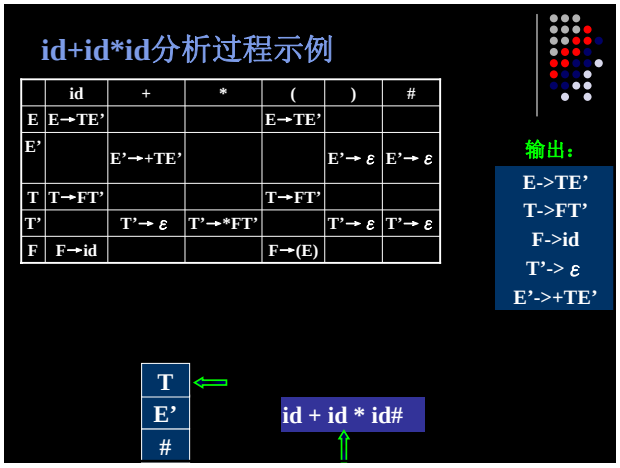

id+id*id分析过程示例
id + * ( ) #
E E→TE' E→TE'
E' E'→+TE' E'→ε E'→ε
T T→FT' T→FT'
T' T'→ε T'→*FT' T'→ε T'→ε
F F→id F→(E)
输出：
E->TE'
T->FT'
F
T'
E'
#
id + id * id#

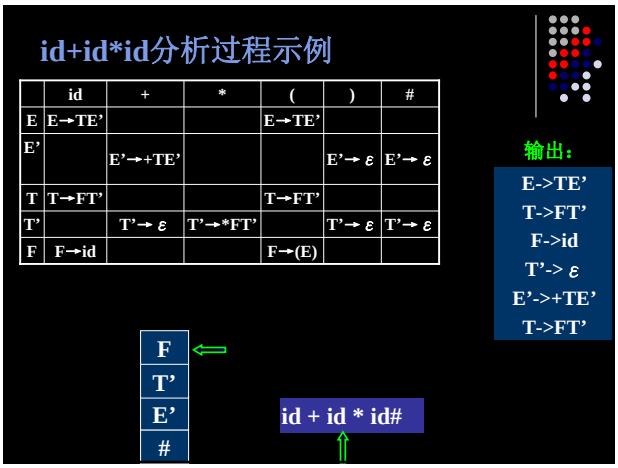

id+id*id分析过程示例
id + * ( ) #
E E→TE' E→TE'
E' E'→+TE' E'→ε E'→ε
T T→FT' T→FT'
T' T'→ε T'→*FT' T'→ε T'→ε
F F→id F→(E)
输出：
E->TE'
T->FT'
F->id
T'-> ε
E'
#
id + id * id#


id+id*id分析过程示例
id + * ( ) #
E E→TE' E→TE'
E' E'→+TE' E'→ε E'→ε
T T→FT' T→FT'
T' T'→ε T'→*FT' T'→ε T'→ε
F F→id F→(E)
输出：
E->TE'
T->FT'
F->id
id
T'
E'
#
id + id * id#


id+id*id分析过程示例
id + * ( ) #
E E→TE' E→TE'
E' E'→+TE' E'→ε E'→ε
T T→FT' T→FT'
T' T'→ε T'→*FT' T'→ε T'→ε
F F→id F→(E)
输出：
E->TE'
T->FT'
F->id
T'-> ε
E'->+TE'
+
T
E'
#
id + id * id#


id+id*id分析过程示例
id + * ( ) #
E E→TE' E→TE'
E' E'→+TE' E'→ε E'→ε
T T→FT' T→FT'
T' T'→ε T'→*FT' T'→ε T'→ε
F F→id F→(E)
输出：
E->TE'
T->FT'
F->id
T'
E'
#
id + id * id#


id+id*id分析过程示例
id + * ( ) #
E E→TE' E→TE'
E' E'→+TE' E'→ε E'→ε
T T→FT' T→FT'
T' T'→ε T'→*FT' T'→ε T'→ε
F F→id F→(E)
输出：
E->TE'
T->FT'
F->id
T'-> ε
E'->+TE'
T
E'
#
id + id * id#


id+id*id分析过程示例
id + id * id#
输出：
E->TE’
T->FT’
F->id
T’-> ε
E’->+TE’
T->FT’
F->id
T’->*FT’

id+id*id分析过程示例

	id	+	*	(	)	#
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→id			F→(E)		

输出：

E->TE'
T->FT'
F->id
T'-> ε
E'->+TE'
T->FT'
F->id
T'->*FT'
F->id

T'
E'
#

id + id * id#

id+id*id分析过程示例

	id	+	*	(	)	#
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→id			F→(E)		

成功！

输出：

E->TE'
T->FT'
F->id
T'-> ε
E'->+TE'
T->FT'
F->id
T'->*FT'
F->id
T'-> ε
E'-> ε

id + id * id#

#

id+id*id分析过程示例

	id	+	*	(	)	#
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→id			F→(E)		

输出：

E->TE'
T->FT'
F->id
T'-> ε
E'->+TE'
T->FT'
F->id
T'->*FT'
F->id
T'-> ε

E'
#

id + id * id#

观察

- 分析表项若为产生式，则为一个产生式，且含一个候选；
- 栈顶元素为终结符时弹出并与当前输入Token匹配；
- 栈顶元素为非终结符时，分析过程表现为用栈顶元素和当前输入Token查分析表项，并用查到的产生式右部进栈替代当前栈顶（可能有多次进栈）；
- 出错的情况：
  - **终结符不匹配**
  - **表项是产生式以外情况**

id+id*id分析过程示例

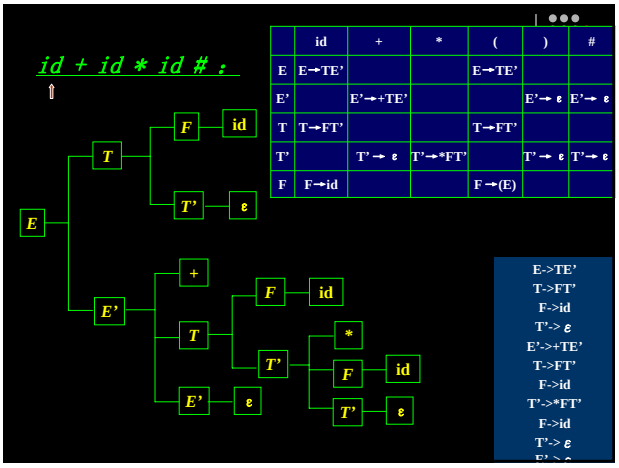
	id	+	*	(	)	#
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→id			F→(E)		

输出：

E->TE'
T->FT'
F->id
T'-> ε
E'->+TE'
T->FT'
F->id
T'->*FT'
F->id
T'-> ε
E'-> ε

#

id + id * id#



4.5.2 构造预测分析表

- 给定文法G构造预测分析表M
- 对文法的每一个产生式A->α ,进行以下两步：
  - 对于任a∈FIRST(α), 把A->α加进M[A, a]
  - 若 ε ∈FIRST(α)，则把A->α加进 M[A, b], b∈FOLLOW(A)或者b=#；

Grammar rule	Pass 1	Pass 2	Pass 3	Pass 4
E → TE’	Follow(E)={#}	Follow(T)={+,-}	Follow(T)={+,-,),#}	Follow(E’)={),#}
T → FT’		Follow(F)={*,/}	Follow(F)={*,/,+,-),#}	Follow(T’)={+,-,),#}
F →(E)	Follow(E)={),#}			
F → i				
T’→ *FT’		Follow(F)=	Follow(F)=	
T’→ /FT’		Follow(F)=	Follow(F)=	
T’→ ε				
E’ → +TE’		Follow(T)=		
E’ → -TE’		Follow(T)=		
E’ → ε				

例1

- 已知文法G：

  E → TE’　　T’→ *FT’|/FT’| ε

  E’ → +TE’|-TE’| ε　　F →(E)| i

  T → FT’
  - 求G的每个候选式α的FIRST集；
  - 求G的每个非终结符A的FOLLOW集；
  - 构造文法G的预测分析表。

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

FIRST(E) = FIRST(T) = FIRST(F) = { (, i }

FIRST(E’) = { +, -, ε }

FIRST(T’) = { *, /, ε }

Follow(E)={),#}

Follow(T)={+,-,),#}

Follow(F)={*,/,+,-),#}

Follow(E’)={),#}

Follow(T’)={+,-,),#}

Grammar rule	Pass 1	Pass 2	Pass 3
E → TE’			First(E)={(,i}
T → FT’		First(T)={(,i}	
F →(E)	First(F)={(}		
F → i	First(F)={(,i}		
T’→ *FT’	First(T’)={*}		
T’→ /FT’	First(T’)={*,/}		
T’→ ε	First(T’)={*,/, ε }		
E’ → +TE’	First(E’)={+}		
E’ → -TE’	First(E’)={+,-}		
E’ → ε	First(E’)={+,-, ε }		

例2

Q→SQ’

Q’→;Q

Q’→ ε

S→s

	s	;	#
Q	Q→SQ’		
S	S→s		
Q’		Q’→;Q	Q’→ ε

FIRST(Q) = { s }

FIRST(S) = { s }

FIRST(Q’) = { ;, ε }

Follow(Q)={#}

Follow(S)={;, #}

Follow(Q’)={#}

例3：

$S \rightarrow I \mid other$

$I \rightarrow if\ (\ E\ )\ S\ L$

$L \rightarrow else\ S \mid \varepsilon$

$E \rightarrow 0 \mid 1$

FIRST(S) = { if, other }
FIRST(I) = { if }
FIRST(L) = { else, ε }
FIRST(E) = {0,1}

FOLLOW(S) = {#, else} FOLLOW(I) = {#, else}
FOLLOW(L) = {#, else} FOLLOW(E) = {)}

	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	

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；**
  - **栈顶为非终结符A，当前输入Token为a，但M[A,a]为空。**

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

错误处理的思想

- 栈顶为终结符
  - **弹出**
- 栈顶为非终结符
  - **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)**

4.6 LL(1)分析法小结

- **LL（1）文法**
  - 分析表M不含多重定义入口
- **LL（1）文法的条件**
  - 文法G是LL(1)的，则对于G的每一个非终结符A的任何两个不同产生式 A -> α | β,有：

（1）FIRST(α) ∩FIRST(β) = φ

（2）若 β=>* ε，则 FIRST(α) ∩ FOLLOW(A) = φ

作业

- P81-82第四章习题1至5

- 上机题（与下一章的上机题任选一个完成）：
  针对if语句的文法编写一个递归下降分析程序，输出结果为抽象语法树。注意，if语句文法中的表达式E采用四则运算表达式的文法；抽象语法树的格式自行设计，如果需要降低难度的话，也可用具体语法树而不用抽象语法树作为输出。