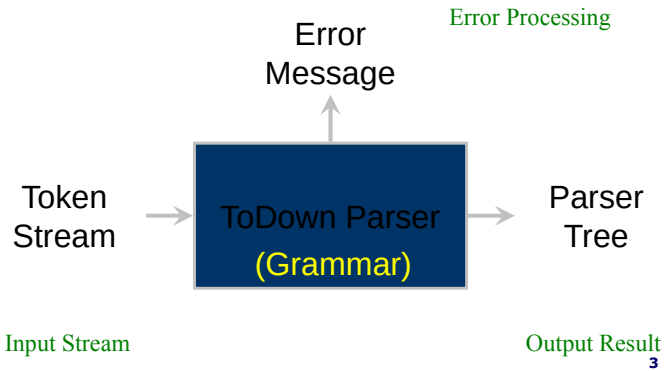
第四章 自上而下的分析

本章内容

- 自上而下分析过程概述
  - 无二义性的上下文无关文法
  - 最左推导和语法树
- LL(1)分析法（无回溯的自上而下分析方法）
  - 消除文法左递归（避免推导过程出现死循环）
  - 修剪文法（避免回溯）
    - 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

2

4.1 自上而下的分析过程概述



3

Token串(程序)的结构

$G=(V_T, V_N, S, \wp)$

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

Int ∈ V_T
Start,*Expr*,*Term* ∈ V_N
Start Symbol: *Start*

S→E
E→E+T|E–T|T
T→T*i|T/i|i

i ∈ V_T
S, E, T ∈ V_N
Start Symbol: S

4

上下文无关文法回顾

$E \to (E) | a$

$E \Rightarrow (E) \Rightarrow ((E)) \Rightarrow ((a))$

$L(G)=\{a,(a),((a)),(((a))),\ldots\}$

$E \to (E)$

$E \Rightarrow (E) \Rightarrow ((E)) \Rightarrow \ldots$

$L(G)=\{\}$

$E \to E+a | a$

$E \Rightarrow E+a \Rightarrow E+a+a \Rightarrow \ldots$

$L(G)=\{a,a+a,a+a+a,\ldots\}$

5

上下文无关文法回顾

$A \to Aa | a$

$A \to aA | a$

$L(G)=\{a^n | n$为整数，$n \geq 1\}$

$A \to A\alpha | \beta$

$L(G)=\{\beta\alpha^n | n$为整数，$n \geq 0\}$

$A \to \alpha A | \beta$

$L(G)=\{\alpha^n\beta | n$为整数，$n \geq 0\}$

$A \to \varepsilon$

$L(G)=\{\varepsilon\}$

$A \to Aa | \varepsilon$

$A \to aA | \varepsilon$

$L(G)=\{a^n | n$为整数，$n \geq 0\}$

6

上下文无关文法回顾

$A \to (A)A | \varepsilon$

任意配对的括号

stmt-sequence→*stmt* ; *stmt-sequence* | *stmt*
stmt →s

L(G)={s, s;s, s;s;s, …}

stmt-sequence→*stmt* ; *stmt-sequence* | *ε*
stmt →s

L(G)={ε, s;, s;s;, s;s;s;, …}

7

上下文无关文法回顾

stmt-sequence→*nonempty-stmt-sequence* | ε
nonempty-stmt-sequence →
stmt ; *nonempty-stmt-sequence* | *stmt*
stmt →s

L(G)={ε, s, s;s, s;s;s, …}

8

上下文无关文法回顾

statement→*if-stmt*|other
if-stmt → if (*exp*) *statement*
| if (*exp*) *statement* else *statement*
exp →0|1

L(G)={other,
if (0) other,
if (1) other,
if (0) other else other,
if (1) other else other,
if (0) if (0) other,
if (0) if (1) other else other,
if (1) other else if (0) other else other,
…}

9

上下文无关文法回顾

statement→*if-stmt*|other
if-stmt → if (*exp*) *statement*
| if (*exp*) *statement* else *statement*
exp →0|1

⇩ 修剪

statement→*if-stmt*|other
if-stmt → if (*exp*) *statement* *else-part*
else-part → else *statement* | ε
exp →0|1

10

Parse Trees and Abstract Syntax Trees

- 推导：从开始符号起构造特定句子的一个方法
- 推导：不能够唯一地表示所推导出的句子的结构

11

推导举例

exp→*exp op exp*|(*exp*) | number
op →+|-|*

(number – number) * number

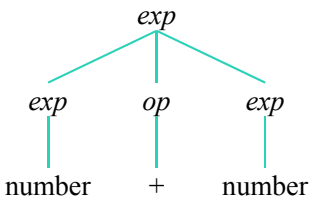
exp⇒*exp op exp*
⇒(*exp*) *op exp*
⇒(*exp op exp*) *op exp*
⇒(number *op exp*) *op exp*
⇒(number - *exp*) *op exp*
⇒(number - number) *op exp*
⇒(number - number) * *exp*
⇒(number - number) * number

12

Parse Tree

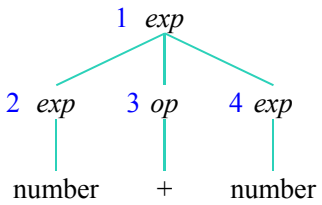
- 语法树（**Parse Tree**）对应于推导
  - 内结点用非终结符标记；
  - 叶子结点用终结符标记；
  - 任一个内结点的孩子表示某个直接推导中该结点被用它的孩子替换。

(1) *exp*⇒*exp op exp*
(2) ⇒number *op exp*
(3) ⇒number + *exp*
(4) ⇒number + number

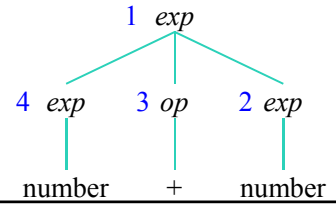


13

Parse Tree的结点命名



(1) *exp*⇒*exp op exp*
(2) ⇒number *op exp*
(3) ⇒number + *exp*
(4) ⇒number + number



exp⇒*exp op exp*
⇒*exp op* number
⇒ *exp* + number
⇒number + number

14

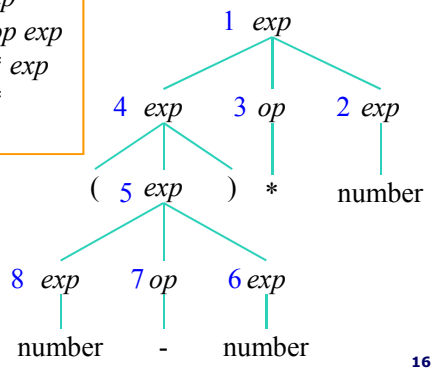
Parse Tree

- 一个**Parse Tree**一般对应多个推导，表示句子的基本结构是相同的。

15

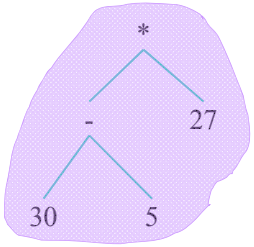
exp⇒*exp op exp*
⇒(*exp*) *op exp*
⇒(*exp op exp*) *op exp*
⇒(number *op exp*) *op exp*
⇒(number - *exp*) *op exp*
⇒(number - number) *op exp*
⇒(number - number) * *exp*
⇒(number - number) * number

(30 – 5) * 27

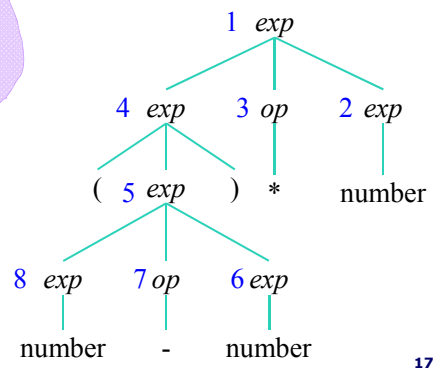


16

从Parse Tree到抽象语法树



(30 – 5) * 27



抽象语法树

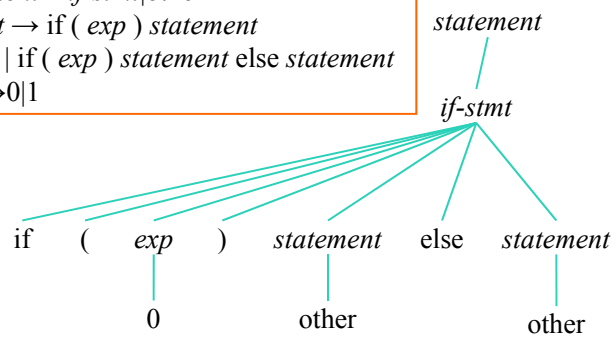
- **Abstract Syntax Tree （Syntax Tree)**
- 一种有用的对**Token**串的结构进行表示的手段
- 例

```
typedef enum {Plus, Minus, Times} OpKind;
typedef enum {OpK, ConstK} ExpKind;
typedef struct streenode
        { ExpKind kind;
          OpKind op;
          struct streenode *lchild, *rchild;
          int val;
        } StreeNode;
typedef StreeNode *SyntaxTree;
```


从Parse Tree到抽象语法树

if (0) other else other

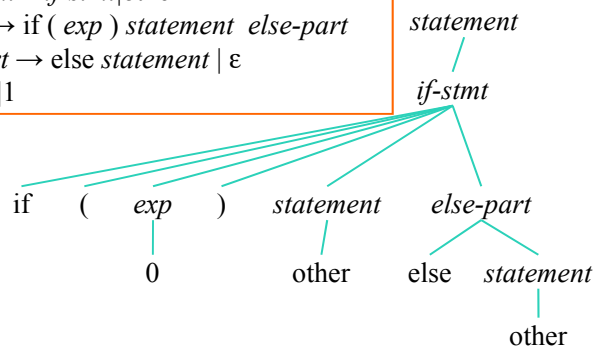
statement→*if-stmt*|other
if-stmt → if (*exp*) *statement*
| if (*exp*) *statement* else *statement*
exp →0|1



从Parse Tree到抽象语法树

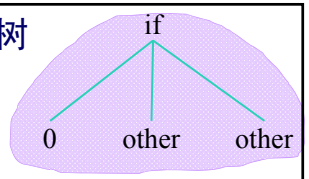
if (0) other else other

statement→*if-stmt*|other
if-stmt → if (*exp*) *statement* *else-part*
else-part → else *statement* | ε
exp →0|1



抽象语法树

if (0) other else other



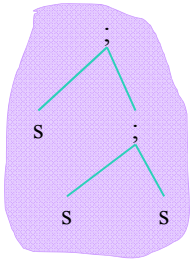

```
typedef enum {ExpK, StmtK} NodeKind;
typedef enum {Zero, One} ExpKind;
typedef enum {IfK, OtherK} StmtKind;
typedef struct streenode
        { NodeKind kind;
          ExpKind ekind;
          StmtKind skind;
          struct streenode *test, *thenpart , *elsepart;
        } StreeNode;
typedef StreeNode *SyntaxTree;
```

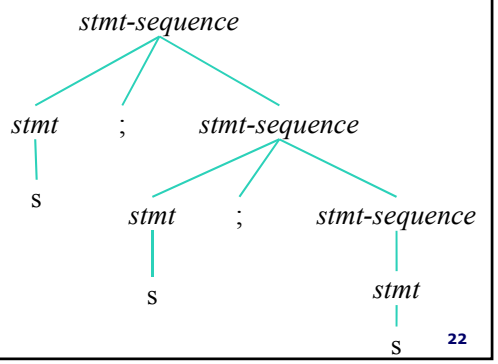
21

从Parse Tree到抽象语法树

s;s;s

stmt-sequence→*stmt* ; *stmt-sequence* | *stmt*
stmt →s





22

二义性文法回顾

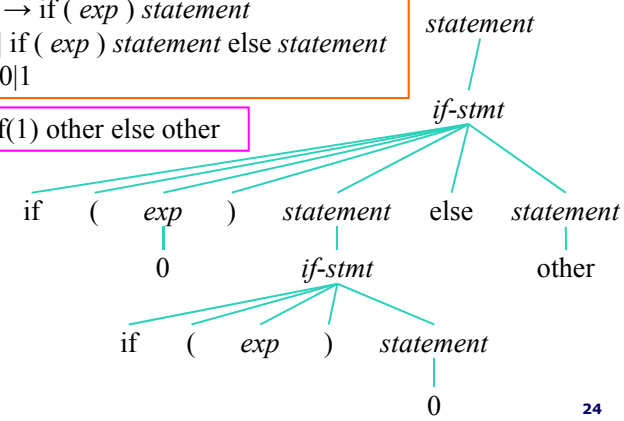
- **Parse Tree** 和抽象语法树都能表达语法结构
- 如果语言中的任意串（**Token**串）：
  - 唯一地对应一颗**Parse Tree**？
  - 可能会对应多个**Parse Tree**？

23

悬挂Else问题

statement→*if-stmt*|other
if-stmt → if (*exp*) *statement*
| if (*exp*) *statement* else *statement*
exp →0|1

if (0) if(1) other else other

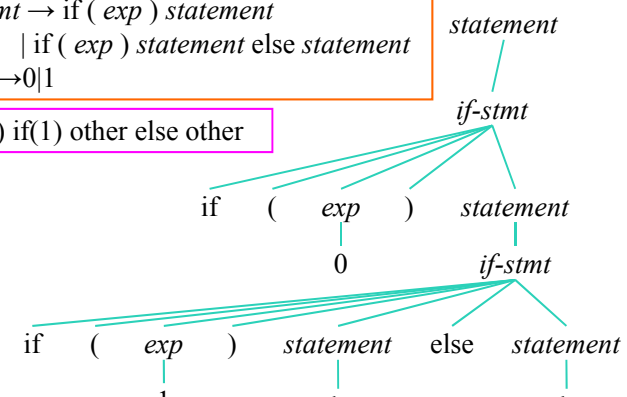


24

悬挂Else问题

statement→*if-stmt*|other
if-stmt → if (*exp*) *statement*
| if (*exp*) *statement* else *statement*
exp →0|1

if (0) if(1) other else other



25

悬挂Else问题的解决

```
if (x != 0)
        if (y == 1/x) ok=TRUE;
        else z = 1/x;
```

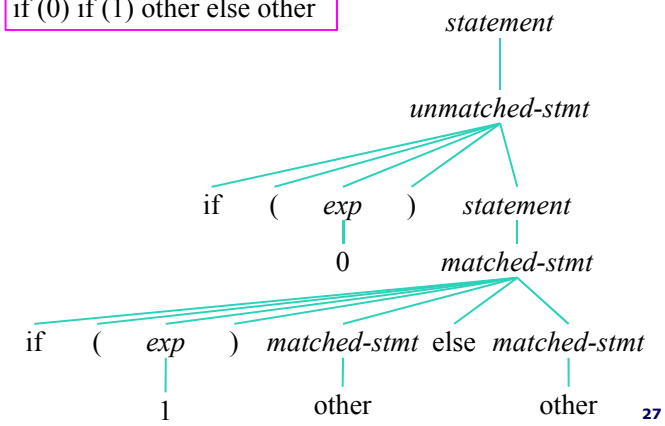
- 最近嵌套规则：
  - **else部分归属于最近的没有else匹配的if部分**

statement → *matched-stmt* | *unmatched-stmt*
matched-stmt → if (*exp*) *matched-stmt else matched-stmt*
| other
unmatched-stmt → if (*exp*) *statement*
| if (exp) *matched-stmt* else *unmatched-stmt*
exp → 0 | 1

26

悬挂Else问题解决举例

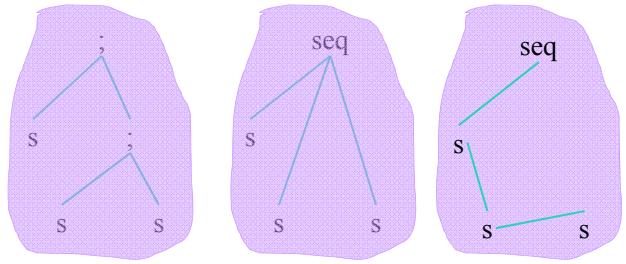
if (0) if (1) other else other



27

无关紧要的二义性

stmt-sequence→*stmt* ; *stmt-sequence* | *stmt*
stmt →s



s;s;s　　s;s;s　　s;s;s

28

4.1 自上而下分析过程概述

- **Parse Tree** 语法树
- **Abstract Syntax Tree** 抽象语法树

- 输入串（**Token**串）
- 当前输入符号（一个**Token**）
- 语法树（当前语法树）
- 当前扩展结点
- 分析成功、或失败
- 回溯

29

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

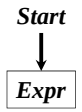
语法树: *Start*

剩余串:
2-2*2
句型:
Start
所用产生式:

30

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

语法树:

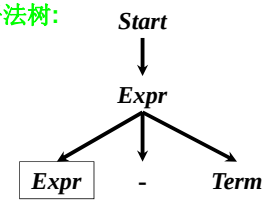


剩余串:
2-2*2
句型:
Expr
所用产生式:
Start → *Expr*

31

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

语法树:

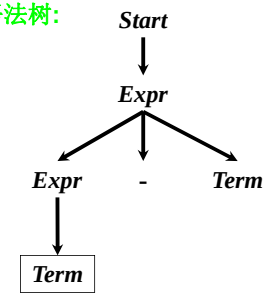


剩余串:
2-2*2
句型:
Expr- Term
所用产生式:
Expr → *Expr* - *Term*

32

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*

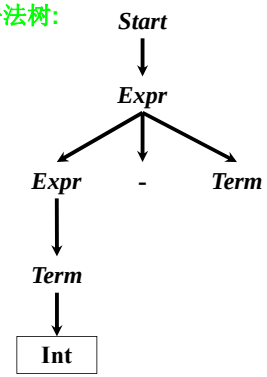
所用产生式:

Expr → *Term*

33

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*

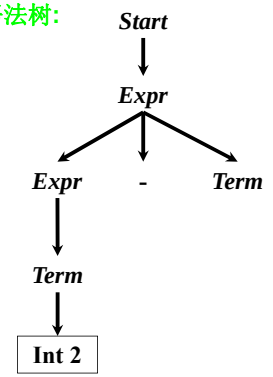
所用产生式:

Term → Int

34

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

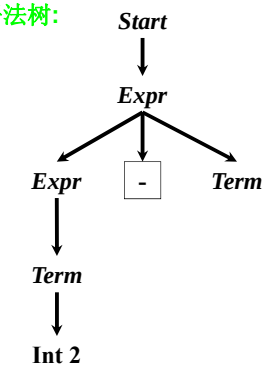
句型:

2 - *Term*

35

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

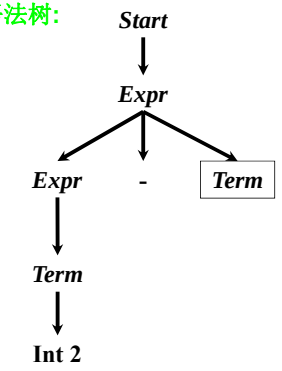
句型:

2 - *Term*

36

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

语法树:



剩余串:

2*2

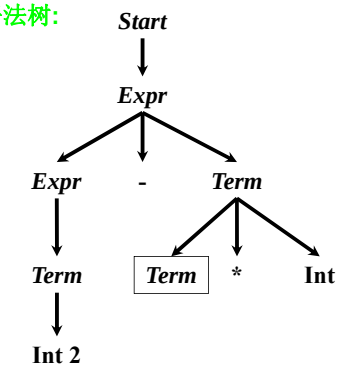
句型:

2 - *Term*

37

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

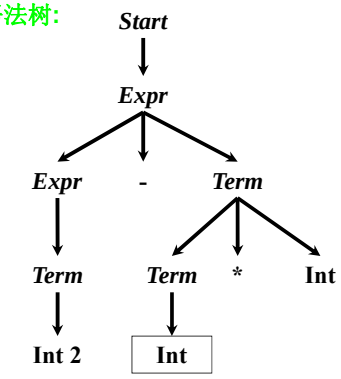
所用产生式:

Term → *Term* * Int

38

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

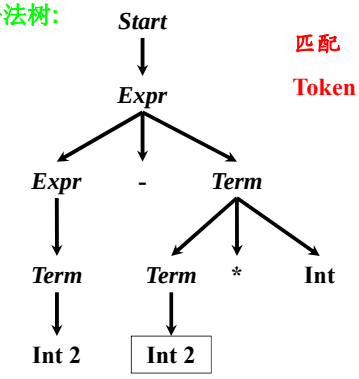
所用产生式:

Term → Int

39

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

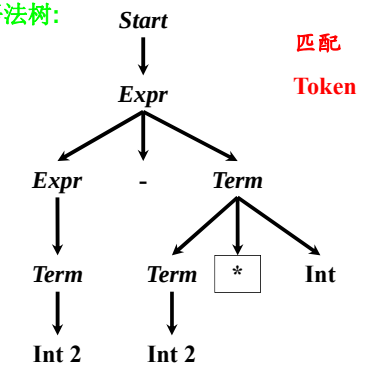
句型:

2 – Int * Int

40

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

语法树:



匹配

Token

剩余串:

*2

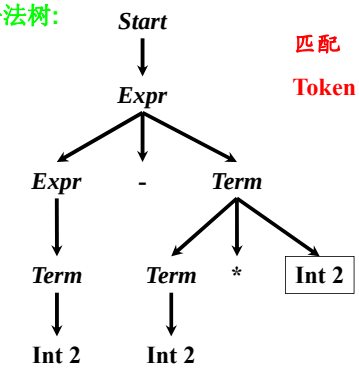
句型:

2 – 2 * Int

41

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

语法树:



匹配

Token

剩余串:

2

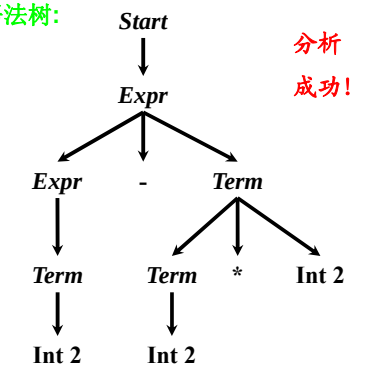
句型:

2 – 2 * 2

42

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

43

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

44

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

45

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

语法树: *Start*

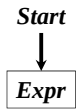
剩余串:
2-2*2
句型:
Start
所用产生式:

有回溯的例子…

46

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

语法树:

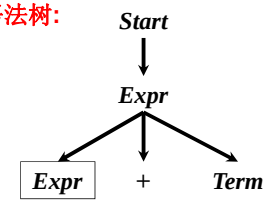


剩余串:
2-2*2
句型:
Expr
所用产生式:
Start → *Expr*

47

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

语法树:

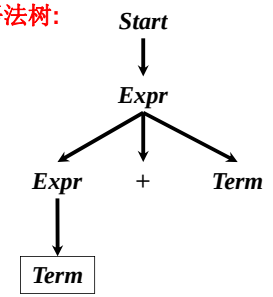


剩余串:
2-2*2
句型:
Expr + *Term*
所用产生式:
Expr → *Expr* + *Term*

48

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*

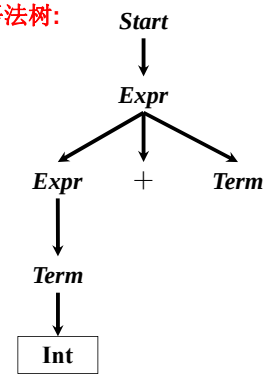
所用产生式:

Expr → *Term*

49

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*

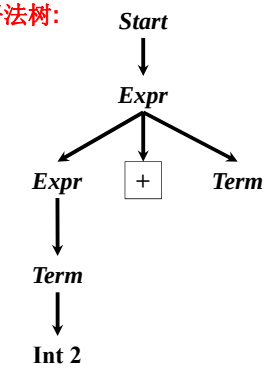
所用产生式:

Term → Int

50

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

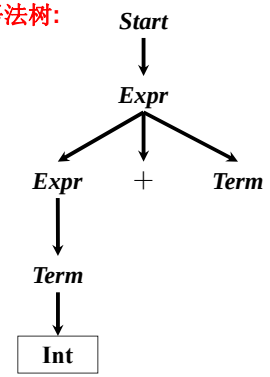
句型:

2 + *Term*

51

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*

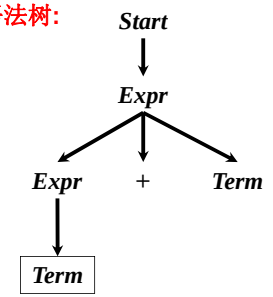
所用产生式:

Term → Int

52

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*

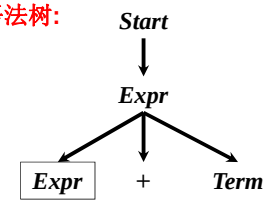
所用产生式:

Expr → *Term*

53

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*

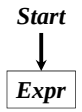
所用产生式:

Expr → *Expr* + *Term*

54

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

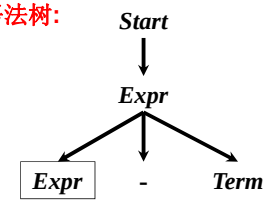
所用产生式:

Start → *Expr*

55

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*

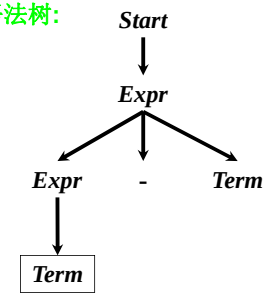
所用产生式:

Expr → *Expr* - *Term*

56

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*

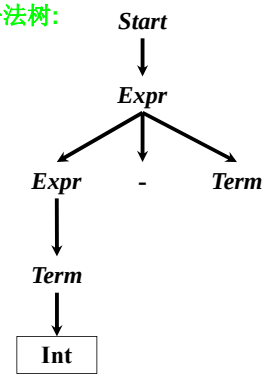
所用产生式:

Expr → *Term*

57

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*

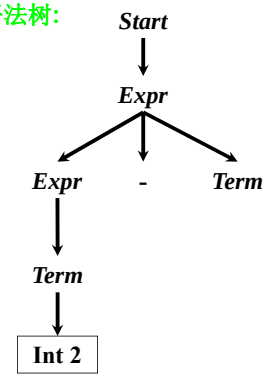
所用产生式:

Term → Int

58

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

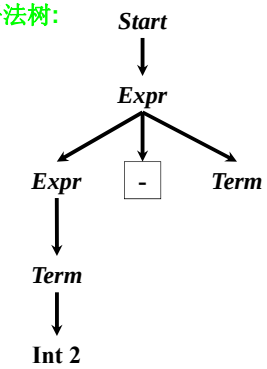
句型:

2 - *Term*

59

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

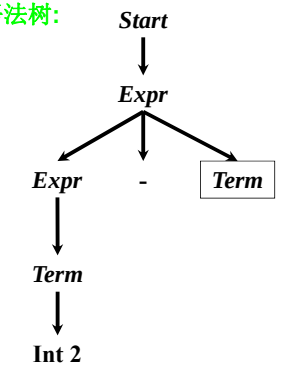
句型:

2 - *Term*

60

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

语法树:



剩余串:

2*2

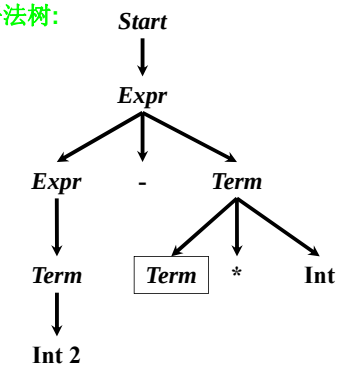
句型:

2 - *Term*

61

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

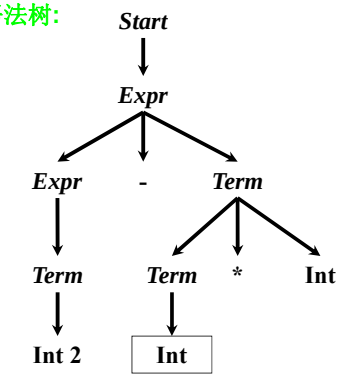
所用产生式:

Term → *Term* * Int

62

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

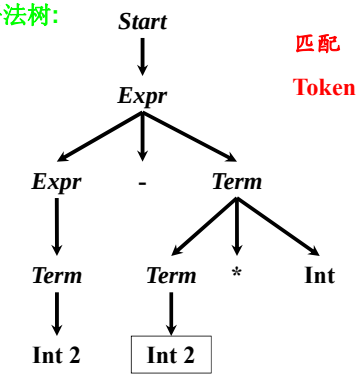
所用产生式:

Term → Int

63

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

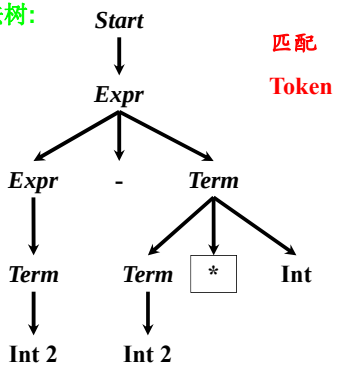
句型:

2 – Int * Int

64

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

语法树:



匹配
Token

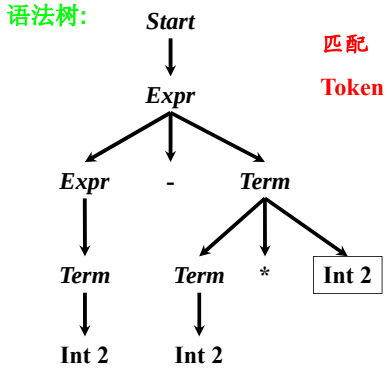
剩余串:
*2

句型:
2 – 2 * Int

65

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

语法树:



匹配
Token

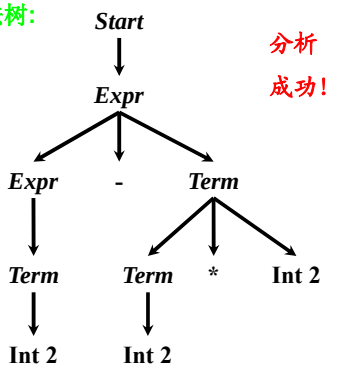
剩余串:
2

句型:
2 – 2 * Int

66

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

语法树:



分析
成功!

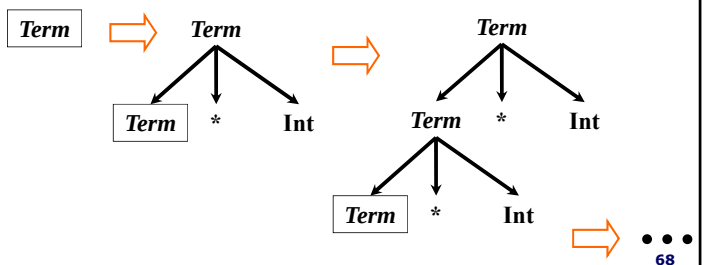
剩余串:

句型:
2 – 2 * 2

67

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:



68

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

69

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 → Pα | β
  - β不以P开头

⇩

- 修剪为：
  - P →βP'
  - P' →αP' | ε

❖不考虑无意义的规则： P → P 和 P =>+ P

70

①消除直接左递归

Original Grammar Fragment

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

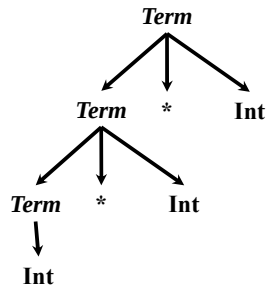
New Grammar Fragment

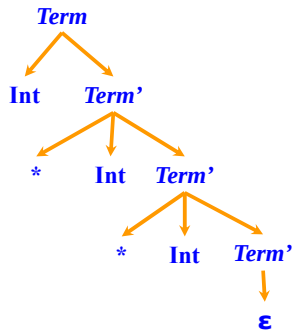
- *Term* → Int *Term'*
- *Term'* → * Int *Term'*
- *Term'* → / Int *Term'*
- *Term'* →ε

71

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

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





72

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Aα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。

73

例：代入

- 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

74

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

75

举例：消除下述文法的左递归

- 例1：
  - **S → Sa | Ab | b | c**
  - **A → Bc | a**
  - **B → Sb | b**
- 例2：
  - **S → Aa | b**
  - **A → Ac | Sd | ε**

for i:=1 to m do /*V_N*/
for j:= 1 to i-1 do
replace each grammar rule choice of the form $P_i \to P_j\beta$ by the rule $P_i \to \alpha_1\beta \mid \alpha_2\beta \mid \ldots \mid \alpha_k\beta$, where $P_j \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$ is the current rule for P_j
消除P_j的直接左递归

76

- 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

for i:=1 to m do /*V_N*/
for j:= 1 to i-1 do
replace each grammar rule choice of the form $P_i \rightarrow P_j\beta$ by the rule $P_i \rightarrow \alpha_1\beta|\alpha_2\beta|\ldots|\alpha_k\beta$, where $P_j \rightarrow \alpha_1|\alpha_2|\ldots|\alpha_k$ is the current rule for P_j
消除P_j的直接左递归

77

4.2.2 消除回溯

- 产生回溯的原因分析
- 消除回溯的方法

78

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

79

Choice Points

- 假设 *Term'* 是语法树中的当前(待扩展)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

80

问题：两个候选式有相同的前缀

- 例如文法

  $N \rightarrow$ if then
  $N \rightarrow$ if then else

- 假定 N 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

81

解决办法: 提左公因子

- 让公共前缀只出现在单个产生式中

  $N \rightarrow$ if then N'
  $N' \rightarrow$ else
  $N' \rightarrow \varepsilon$

- 遇到下一个Token是if时，只有唯一的选择!

82

另一种情况

- 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 跟输入流中的下一个符号（**Token**）的匹配情况
- 特别地，当 N_1或N_2 能推导出ε时
  - 这时必须基于α_1 和α_2进行判断

83

首符集（FIRST集）概念

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

- **Start with concept of N deriving ε**
- **Two rules**
  - **$N \rightarrow \varepsilon$ implies N derives ε**
  - **$N \rightarrow N_1 \mid \ldots \mid N_n$ and for all $1 \le i \le n$, N_i derives ε implies N derives ε**

84

推导ε的不动点算法

for all nonterminals N
    set N derives ε to be false
for all productions of the form $N \rightarrow \varepsilon$
    set N derives ε to be true
while (some N derives ε changed in last iteration)
  for all productions of the form $N \rightarrow N_1 \mid \ldots \mid N_n$
    if (for all $1 \le i \le n$, N_i derives ε)
      set N derives ε to be true

85

回溯问题的解决

文法G的产生式形如： $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_m$
则可以计算出：FIRST(α_i)，i=1,…,m
如果对于任意1<=i, j<=m都有

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$$

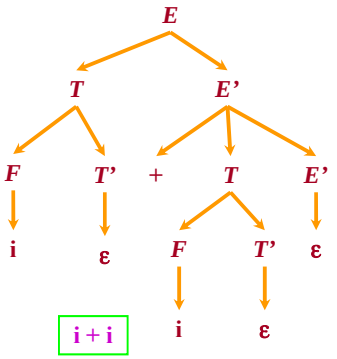
则意味着由A的每个候选式所能够推导出来的符号串的开头符号（属于V_T）均不相同

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 $a \in FIRST(\alpha_i)$

86

例：消除回溯

E ⇒ TE' ⇒ FT'E' ⇒ iT'E'



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

i∈FIRST(TE')
i∈FIRST(FT')
i∈FIRST(i)

FOLLOW(A)
= {a|S =>*αAaβ，
a∈V_T，α,β∈V* }

87

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所有候选首符集两两不相交。

- 文法修剪：提取公共左因子

若$A \rightarrow \delta\beta_1 \mid \cdots \mid \delta\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m$， $m,n \ge 1$
其中每个γ不以δ开头
那么可以等价地改写成：

$$A \rightarrow \delta A' \mid \gamma_1 \mid \cdots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \cdots \mid \beta_n$$

88

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$A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$
  - $\exists i$ ($a \in FIRST(\alpha_i)$,则采用规则$A \to \alpha_i$)
  - $\forall i$ ($a \notin FIRST(\alpha_i)$), 且$\exists i(\varepsilon \in FIRST(\alpha_i))$, 且 $a \in FOLLOW(A)$则采用规则$A \to \alpha_i$)
- 若不满足上述情况，则a的出现是一种语法错误。

(1) FIRST(β)和FOLLOW(A)计算

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

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

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

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

FIRST集的性质

- $t \in FIRST(t)$
- $FIRST(S) \subseteq FIRST(S\beta)$
- 若$N \Rightarrow^* \varepsilon$则 $FIRST(\beta) \subseteq FIRST(N\beta)$
- 产生式$N \to S\beta$隐含着 $FIRST(S\beta) \subseteq FIRST(N)$

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

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) $N \Rightarrow^* \varepsilon$ 隐含 $First(\beta) \subseteq First(N\beta)$
4) $N \to S\beta$ 隐含 $First(S\beta) \subseteq First(N)$

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

约束条件
First(* *Num Term'*) ⊆ First(*Term'*)
First(/ *Num Term'*) ⊆ First(*Term'*)
First(*) ⊆ First(* *Num Term'*)
First(/) ⊆ 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）

93

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）

94

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）

95

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）

96

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)

97

求所有非终结符FISRT集的迭代算法

①构造FIRST(X), $X \in V_T \cup V_N$

- 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$X \in V_T$，则X属于FIRST(X)
  - 若$X \in V_N$，且有$X \rightarrow a\alpha$，则a属于FIRST(X)
  - 若$X \in V_N$，若有 $X \rightarrow \varepsilon$，则ε属于FIRST(X)
  - 若$X \in V_N$，有$X \rightarrow Y\ldots$, 且$Y \in V_N$，则 FIRST(Y) \ {ε} 包含于FIRST(X)

98

求FISRT集的迭代算法 (续)

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$\varepsilon \in FIRST(X_1)$，则$FIRST(X_2)$的所有非ε元素加入FIRST(α)
- 若$\varepsilon \in FIRST(X_1)$, $\varepsilon \in FIRST(X_2)$，则$FIRST(X_3)$的所有非ε元素加入FIRST(α)
- 一般地，若$\varepsilon \in FIRST(X_i)$, i=1..m，m<n, $FIRST(X_{i+1})$的所有非ε元素加入FIRST(α)，当m=n时则将ε加入FIRST(α)

99

求FIRST集的迭代算法的描述

for *all nonterminals A* do *First(A)*:={};
While *there are changes to any First(A)* do
  for *each production* $A \rightarrow X_1X_2\ldots X_n$ do
    k := 1; *Continue* := true;
    While *Continue* = true and k <= n do
      add $First(X_k) \setminus \{\varepsilon\}$ *to First(A)*
      if *ε is not in* $First(X_k)$ then *Continue* := false;
      $k := k + 1$;
    if *Continue* = true then *add* {ε} *to First(A)*

100

例：

$S \rightarrow I \mid \text{other}$
$I \rightarrow \text{if}\ (\ E\ )\ S\ L$
$L \rightarrow \text{else}\ S \mid \varepsilon$
$E \rightarrow 0 \mid 1$

Grammar rule	Pass I	Pass 2
$S \rightarrow I$		First(S)={other,if}
$S \rightarrow$ other	First(S)={other}	
$I \rightarrow$ if (E) $S\ L$	First(I)={if}	
$L \rightarrow$ else S	First(L)={else}	
$L \rightarrow \varepsilon$	First(L)={else,ε}	
$E \rightarrow 0$	First(E)={0}	
$E \rightarrow 1$	First(E)={0, 1}	

101

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* $A \rightarrow X_1X_2\ldots X_n$ do
    add First(X_1) *to First(A);*

102

FOLLOW集

- 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为文法开始符号，则#属于FOLLOW(A)
  - 若有产生式 $B \rightarrow \alpha A \beta$
    则 FIRST(β) \ { ε } ⊆ FOLLOW(A)
  - 若有产生式 $B \rightarrow \alpha A \beta$ 且 $\varepsilon \in$ FIRST(β)
    则 FOLLOW(B) ⊆ FOLLOW(A)

103

构造FOLLOW集的算法

Follow(start-symbol) := {#};
for *all nonterminals* $A \neq$ *start-symbol* do *Follow(A)*:={};
While *there are changes to any Follow sets* do
  for *each production* $A \rightarrow X_1X_2\ldots X_n$ do
    for each X_i *that is a nonterminal* do
      add First($X_{i+1}X_{i+2}\ldots X_n$) \ {ε} *to Follow*(X_i)
      /* *Note: if i=n, then* $X_{i+1}X_{i+2}\ldots X_n = \varepsilon$ */
      if ε *is in First*($X_{i+1}X_{i+2}\ldots X_n$) then
        add *Follow(A) to Follow*(X_i)

104

例

4）例，已知文法G, 求G的每个非终结符A的FOLLOW(A)集

E → TE'　　T'→ *FT'|/FT'|ε

E' → +TE'|-TE'|ε　　F →(E)| i

T → FT'

FIRST集的构造：

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

FIRST(E') = { + , ε}

FIRST(T') = { * , ε}

105

First集

①构造FIRST(X), $X \in V_T \cup V_N$

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

Grammar rule	Pass I	Pass II	Pass III
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}		

106

Follow集

① #属于FOLLOW(S)

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

③若有B→Aβ 且 ε∈FIRST(β)则 FOLLOW(B) ⊆ FOLLOW(A)

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

107

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}**

108

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) = { # ,) }

109

最终构造结果

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

FIRST(E') = { + , ε}

FIRST(T') = { * , ε}

FOLLOW(E) = FOLLOW(E') = {) , # }

FOLLOW(T) = FOLLOW(T') = { + ,) , # }

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

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

110

4.3递归下降分析程序

- LL(1)文法
  - 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, 若它的某个候选首符集含ε, 则有 FIRST(A)∩FOLLOW(A)=ϕ
- LL(1)分析
  - L：从左到右扫描输入流；
  - L：最左推导；
  - 1：向前查看一个Token

111

Recursive Descent Parser

- **Predictive Parsing + Hand Coding**

- 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

112

例：文法非终结符对应递归子程序

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

113

```
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'

114

```
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

115

$i_1 + i_2 * i_3$ 分析过程：

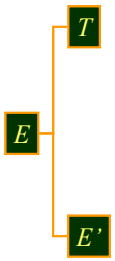
↑

E

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

116

$i_1 + i_2 * i_3$ 分析过程：

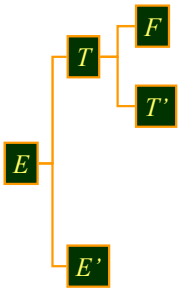



```
PROCEDURE E;
BEGIN
  T;E'
END;
```

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

117

$i_1 + i_2 * i_3$ 分析过程：

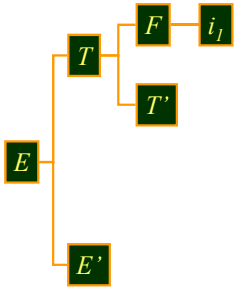



```
PROCEDURE T;
BEGIN
  F;T'
END;
```

```
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
```

118

$i_1 + i_2 * i_3$ 分析过程：

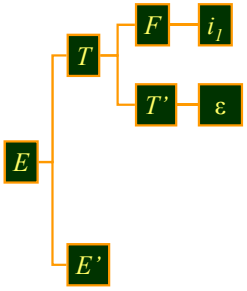



```
PROCEDURE F;
BEGIN
IF SYM='i' THEN ADVANCE
 ELSE IF SYM='('THEN
  BEGIN
   ADVANCE;
   E;
   IF SYM=')' THEN ADVANCE
      ELSE ERROR
  END
 ELSE ERROR
END
```

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

119

$i_1 + i_2 * i_3$ 分析过程：

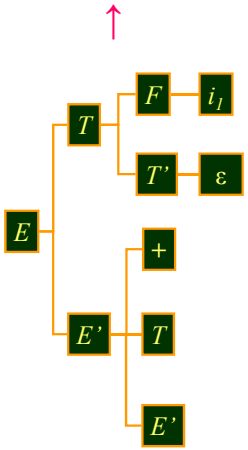



```
PROCEDURE T;
BEGIN
  F;T'
END;
```

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

120

$i_1 + i_2 * i_3$ 分析过程：

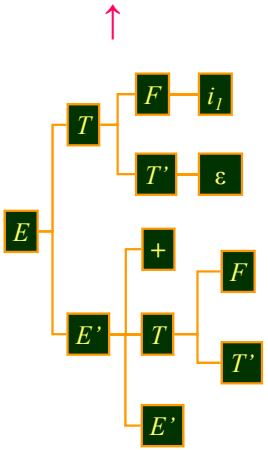



```
PROCEDURE E;
BEGIN
  T;E'
END;
```

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

121

$i_1 + i_2 * i_3$ 分析过程：

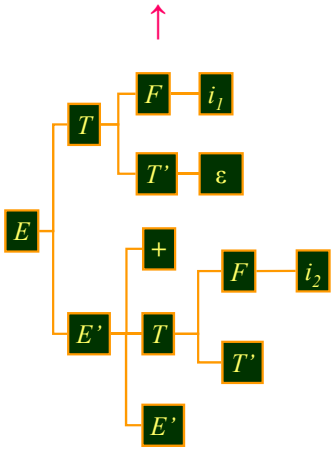



```
PROCEDURE E';
BEGIN
IF SYM='+'THEN
 BEGIN
   ADVANCE;
   T;E'
 END
END;
```

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

122

$i_1 + i_2 * i_3$ 分析过程：

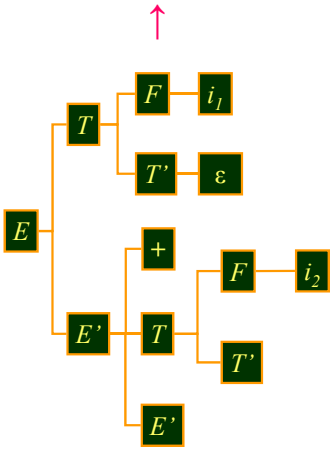



```
PROCEDURE T;
BEGIN
  F;T'
END;
```

```
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
```

123

$i_1 + i_2 * i_3$ 分析过程：

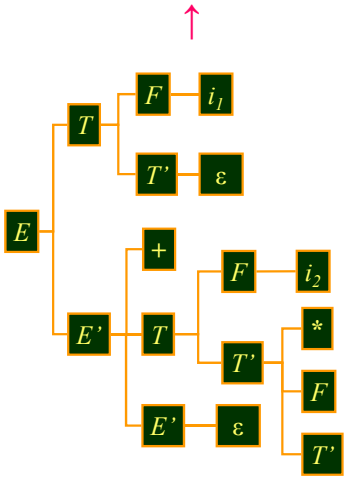



```
PROCEDURE F;
BEGIN
IF SYM='i' THEN ADVANCE
 ELSE IF SYM='('THEN
  BEGIN
   ADVANCE;
   E;
   IF SYM=')' THEN ADVANCE
      ELSE ERROR
  END
 ELSE ERROR
END
```

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

124

$i_1 + i_2 * i_3$ 分析过程：

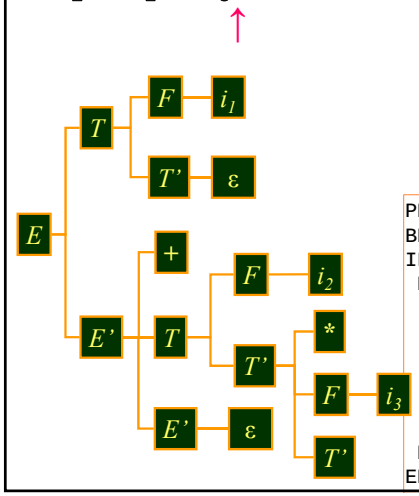



```
PROCEDURE T;
BEGIN
  F;T'
END;
```

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

125

$i_1 + i_2 * i_3$ 分析过程：

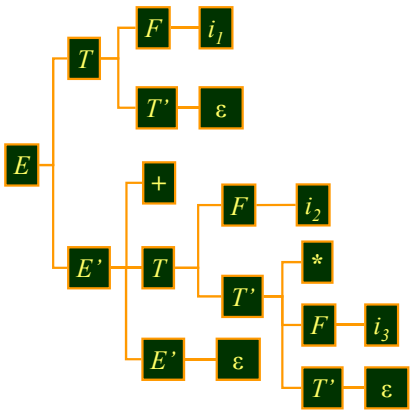



```
PROCEDURE T';
BEGIN
 IF SYM='*'THEN
 BEGIN
  ADVANCE;
  F;T'
 END
END
```

```
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
```

126

$i_1 + i_2 * i_3$ 分析过程：




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

127

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，它就按此式扩展语法树，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。

128

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)
```

129

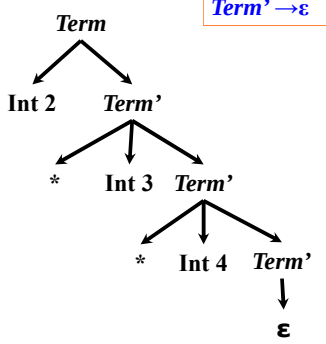
Building A Parse Tree

- **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**

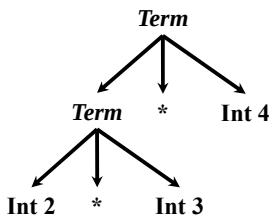
130

2*3*4的语法树

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



抽象语法树



131

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)
```

132

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

2*3*4

133

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

*

2*3*4

134

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

*

2*3*4

135

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

*

Int 3

2*3*4

136

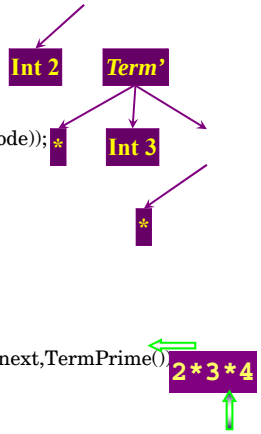
Building Parse Tree (Concrete Tree)

- 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)
```



137

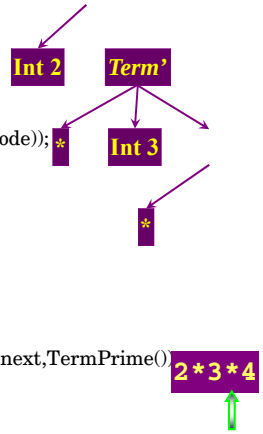
Building Parse Tree (Concrete Tree)

- 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)
```



138

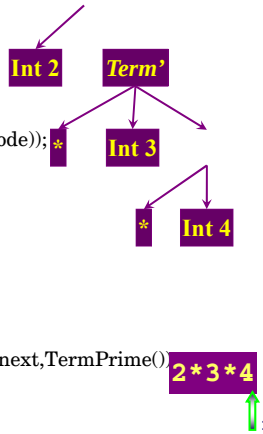
Building Parse Tree (Concrete Tree)

- 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)
```



139

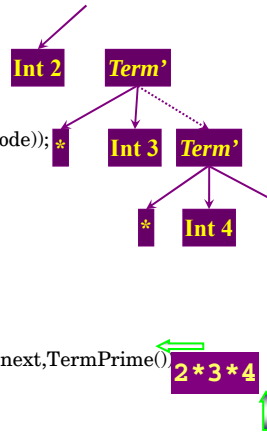
Building Parse Tree (Concrete Tree)

- 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)
```



140

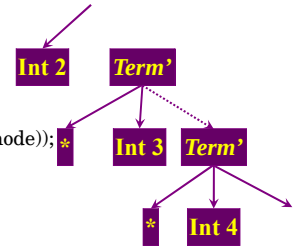
Building Parse Tree (Concrete Tree)

- 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)
```



2*3*4

41

Building Parse Tree (Concrete Tree)

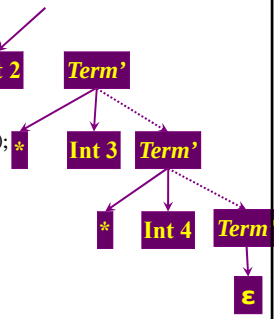
- Term()

```
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)



2*3*4

42

Building Parse Tree (Concrete Tree)

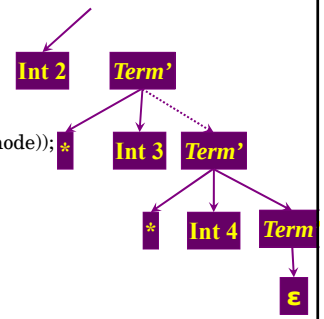
- Term()

```
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)



2*3*4

43

Building Parse Tree (Concrete Tree)

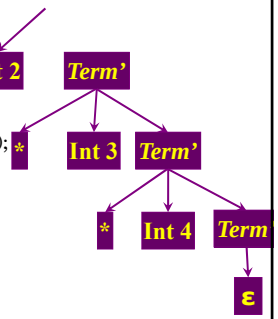
- Term()

```
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)



2*3*4

44

Building Parse Tree (Concrete Tree)

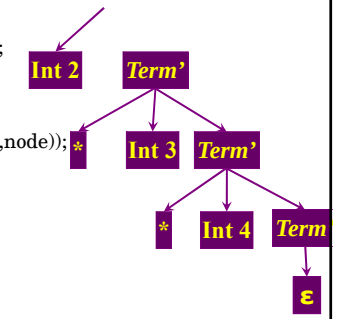
- Term()

```
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)



45

Building Parse Tree (Concrete Tree)

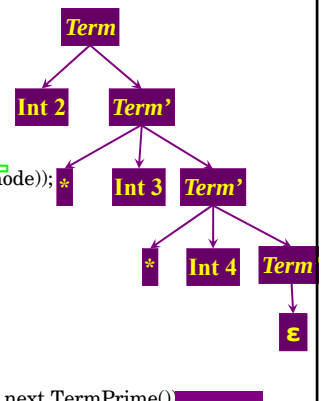
- Term()

```
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)



46

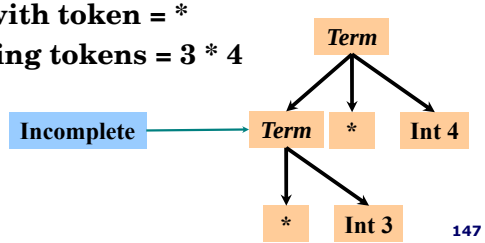
直接产生抽象语法树

- **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

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

- **(root, incomplete) = TermPrime()**
  - **Called with token = ***
  - **Remaining tokens = 3 * 4**

2*3*4



147

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)
```

148

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 node
```

149

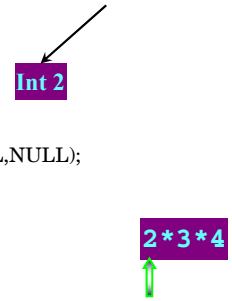
Building Parse Tree (Abstract Tree)

- **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 node
```



150

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 node
```


151

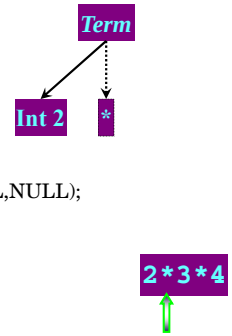
Building Parse Tree (Abstract Tree)

- 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 node
```



152

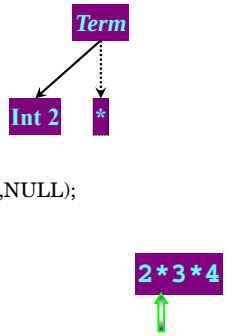
Building Parse Tree (Abstract Tree)

- 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 node
```



153

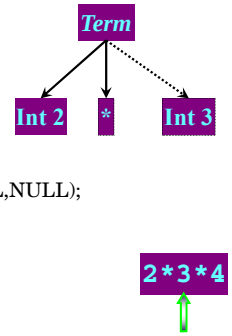
Building Parse Tree (Abstract Tree)

- 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 node
```



154

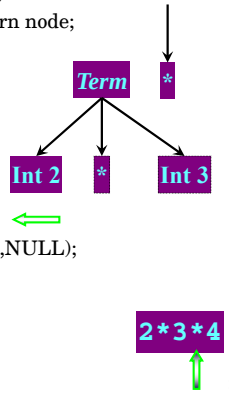
Building Parse Tree (Abstract Tree)

- 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 node
```



155

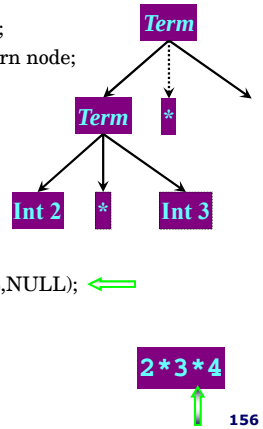
Building Parse Tree (Abstract Tree)

- 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 node
```



156

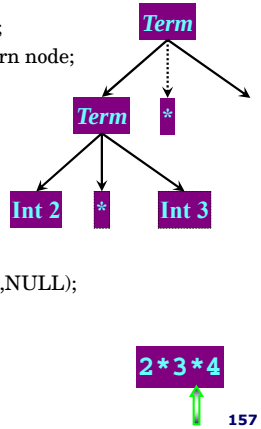
Building Parse Tree (Abstract Tree)

- 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 node
```



157

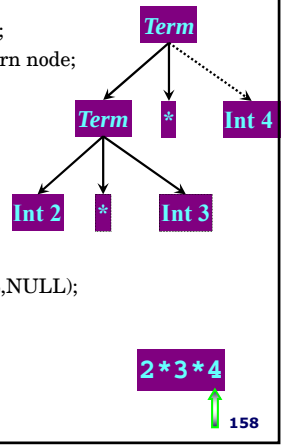
Building Parse Tree (Abstract Tree)

- 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 node
```



158

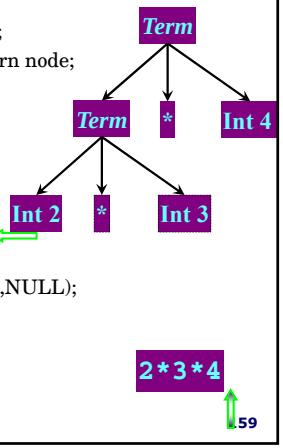
Building Parse Tree (Abstract Tree)

- 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 node
```



159

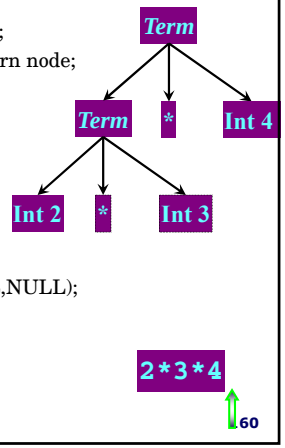
Building Parse Tree (Abstract Tree)

- 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 node
```



160

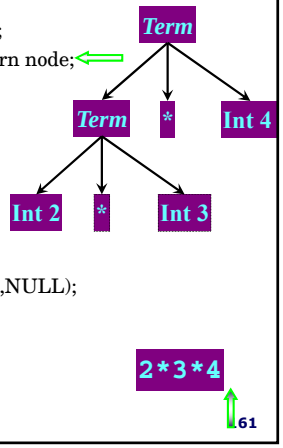
Building Parse Tree (Abstract Tree)

- 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 node
```



161

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()

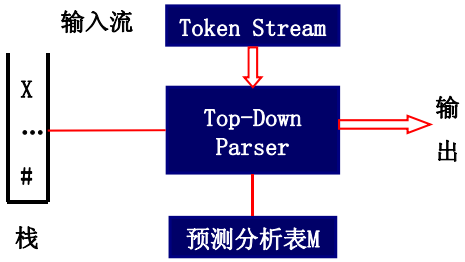
162

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

163

4.5.1 预测分析程序工作过程



164

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$a \in V_T$；$A \in V_N$
- 动作：根据分析表内元素做规定的推导

165

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， a置为下一个Token；
- 若$X \in V_N$，则查分析表元素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。

166

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)		

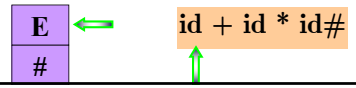
167

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：



168

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#

169

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#

170

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#

171

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#

172

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#

173

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#

174

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#

175

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
T'
E'
#

id + id * id#

176

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

id
T'
E'
#

id + id * id#

177

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'
E'
#

id + id * id#

178

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
T'
E'
#

id + id * id#

179

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
T'
E'
#

id + id * id#

180

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

id
T'
E'
#

id + id * id#

181

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#

182

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#

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#

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

成功！

id + id * id#

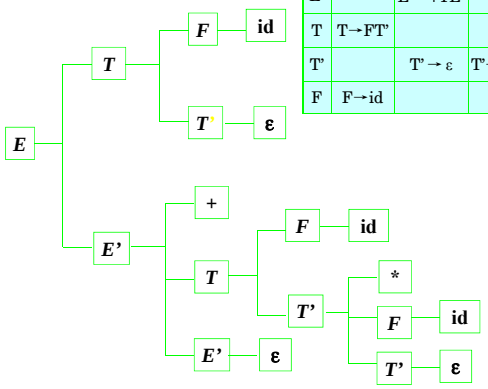
#

观察

- 分析表项若为产生式，则只含一个候选式；
- 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

186

*id + id * id # ;*



	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)		

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

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=#；

188

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。

189

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}		

190

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				

191

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

FIRST(E') = { +, -,ε}

FIRST(T') = { *, /,ε}

Follow(E)={),#}

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

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

Follow(E')={),#}

Follow(T')={+,-,),#}

	(	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						

对于A→α,

∀a∈FIRST(α) (M[A, a]:= 'A→α')

ε∈FIRST(α) → ∀b∈FOLLOW(A)∪{#}(M[A, b]:= 'A→α')

192

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')={#}

193

例3:

$S \to I \mid other$
$I \to if\ (\ E\ )\ S\ L$
$L \to else\ S \mid \varepsilon$
$E \to 0 \mid 1$

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	

194

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

195

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(A) ∩ FOLLOW(A) = φ

196

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。

197

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)**

198

本章练习

- **P81-82**第四章习题**1,2,4**

- 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。

199