

第三章 语言和文法的形式定义

1.语言是什么?

2.如何描述语言?

3.语言中的句子如何识别?

如何来描述一种语言？

- 当一个语言仅含有有限个句子时，可采用枚举来表示此种语言
- 即把该语言中的全部句子一一列举出来即可。
- 例如，若一个语言L只含如下的两个句子，则可将它表示为：

$L = \{ \text{I am teacher, You are students} \}$

缺点：语言一般是无限的，而非有限的。

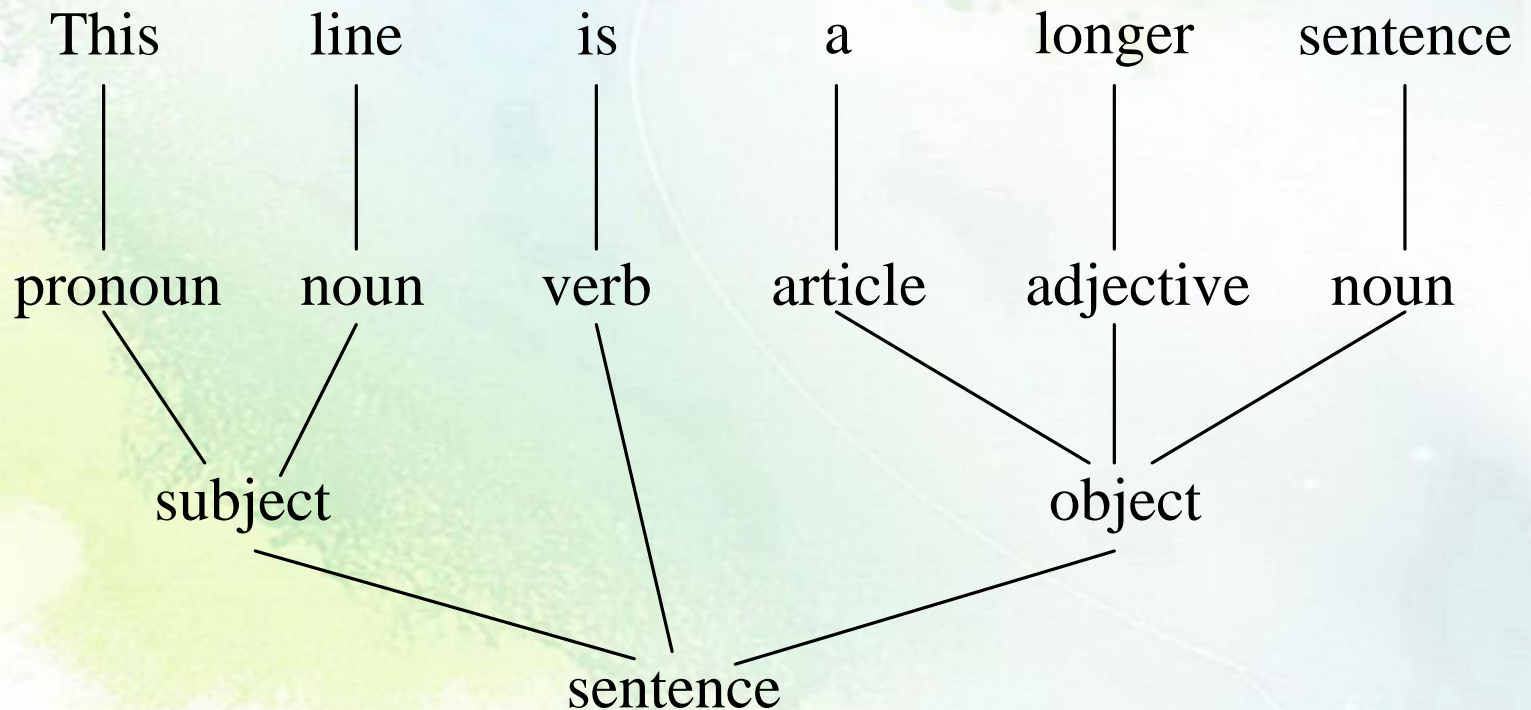
如何来描述一种语言？

如果语言是无穷的，找出语言的有穷表示。

语言的有穷表示有两个途经：

- 生成方式（文法）：语言中的每个句子可以用严格定义的规则来构造。
- 识别方式（自动机）：用一个过程，当输入的一任意串属于语言时，该过程经有限次计算后就会停止并回答“是”，若不属于，要么能停止并回答“不是”，要么永远继续下去。

This line is a longer sentence



英文句子规则的定义

<句子>::=<主语><复合谓语>

<主语>::=<名词>

<名词>::=GZ | SH | BJ | city

<复合谓语>::=<系动词><表语>

<系动词>::=is | was

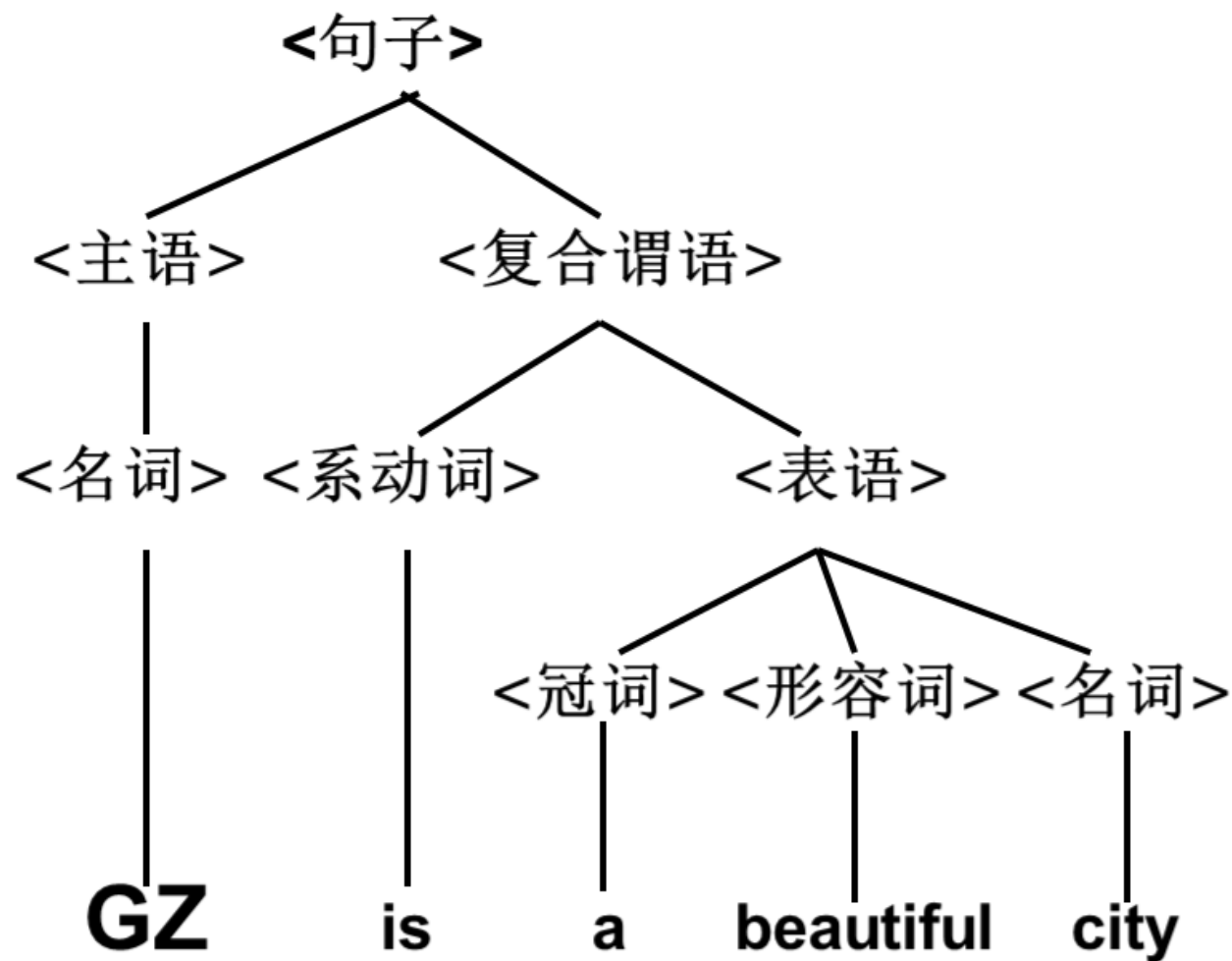
<表语>::=<冠词><形容词><名词>

<冠词>::=a | an | the

<形容词>::=beautiful | great | wonderful

说明：我们通常用符号→来替代上述的符号 ::=

如： <冠词>→a | an | the



上下文无关语言的描述

上下文无关文法 (*context-free grammars*)

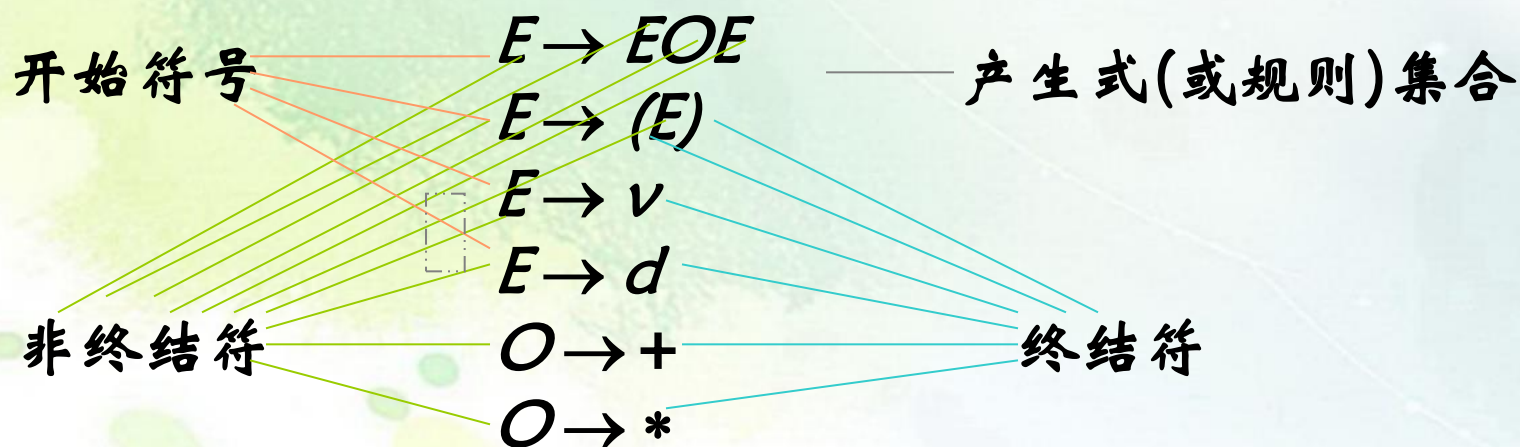
例子

$$E \rightarrow EOE$$
$$E \rightarrow (E)$$
$$E \rightarrow v$$
$$E \rightarrow d$$
$$O \rightarrow +$$
$$O \rightarrow *$$

上下文无关语言的描述

上下文无关文法的四个基本要素:

1. 终结符 (*terminals*) 的集合 有限符号集, 相当于字母表
2. 非终结符 (*nonterminals*) 的集合 有限变量符号的集合
3. 开始符号 (*start symbol*) 一个特殊的非终结符
4. 规则或产生式 (*productions*) 的集合 形如: $\langle \text{head} \rangle \rightarrow \langle \text{body} \rangle$



上下文无关语言的描述

上下文无关文法的形式定义:

一个上下文无关文法 *CFG* (context-free grammars) 是一个四元组 $G = (V_N, V_T, P, S)$.

非终结符的集合

终结符的集合

产生式的集合

开始符号

满足

$$V_N \cap V_T = \Phi$$

$$S \in V_N$$

产生式形如 $A \rightarrow \alpha$, 其中 $A \in V_N, \alpha \in (V_N \cup V_T)^*$

- **产生式(规则)的作用:**

- ——用于定义或描述语言中的语法成分,
- ——常用来产生(即推导)出语言中的句子, 故一般也将规则称为产生式。

注意:

- (1)规则右部的运算符有:选择($|$)、并置、括号($()$),
- (2)无重复($*$)运算符, 用递归实现重复。
- (3)规则右部可使用元符号 ε :为了能产生空串。

在表示文法时，通常有以下的习惯用法：

- 大写字母A~Z表示非终结符，或者用尖括号把非终结符括起来；
- 前面的小写字母a、b、c...表示单个终结符号；
- 后面的小写字母u、v、w、x、y、z以及 α 、 β 等符号表示($V=V_N \cup V_T$)上的符号串。

文法与正则表达式的异同点

- 文法——反映句子的组成
正则表达式——反映单词的组成
- 采用类似的命名惯例和运算

上下文无关语言的描述

上下文无关文法举例

CFG $G_{\text{exp}} = (\{E, O\}, \{ (,), +, *, v, d \}, P, E)$.

其中产式集合 P 为

$$E \rightarrow EOE$$
$$E \rightarrow (E)$$
$$E \rightarrow v$$
$$E \rightarrow d$$
$$O \rightarrow +$$
$$O \rightarrow *$$

上下文无关语言的描述

产生式(规则)集合的缩写记法:

形如 $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_n$ 的产生式集合可简缩记为 $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$, 如

$E \rightarrow EOE$

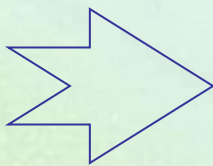
$E \rightarrow (E)$

$E \rightarrow v$

$E \rightarrow d$

$O \rightarrow +$

$O \rightarrow *$



$G[E]:$

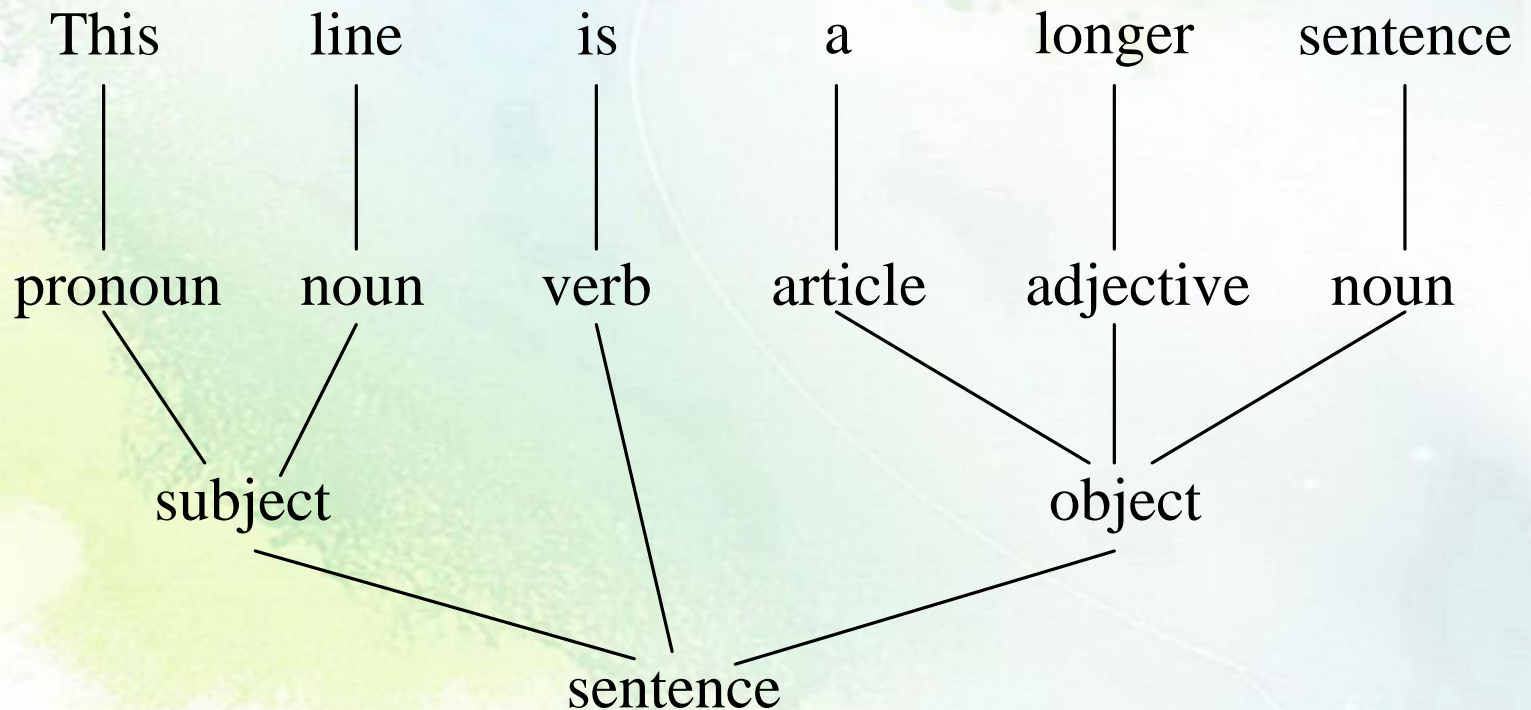
$E \rightarrow EOE | (E) | v | d$

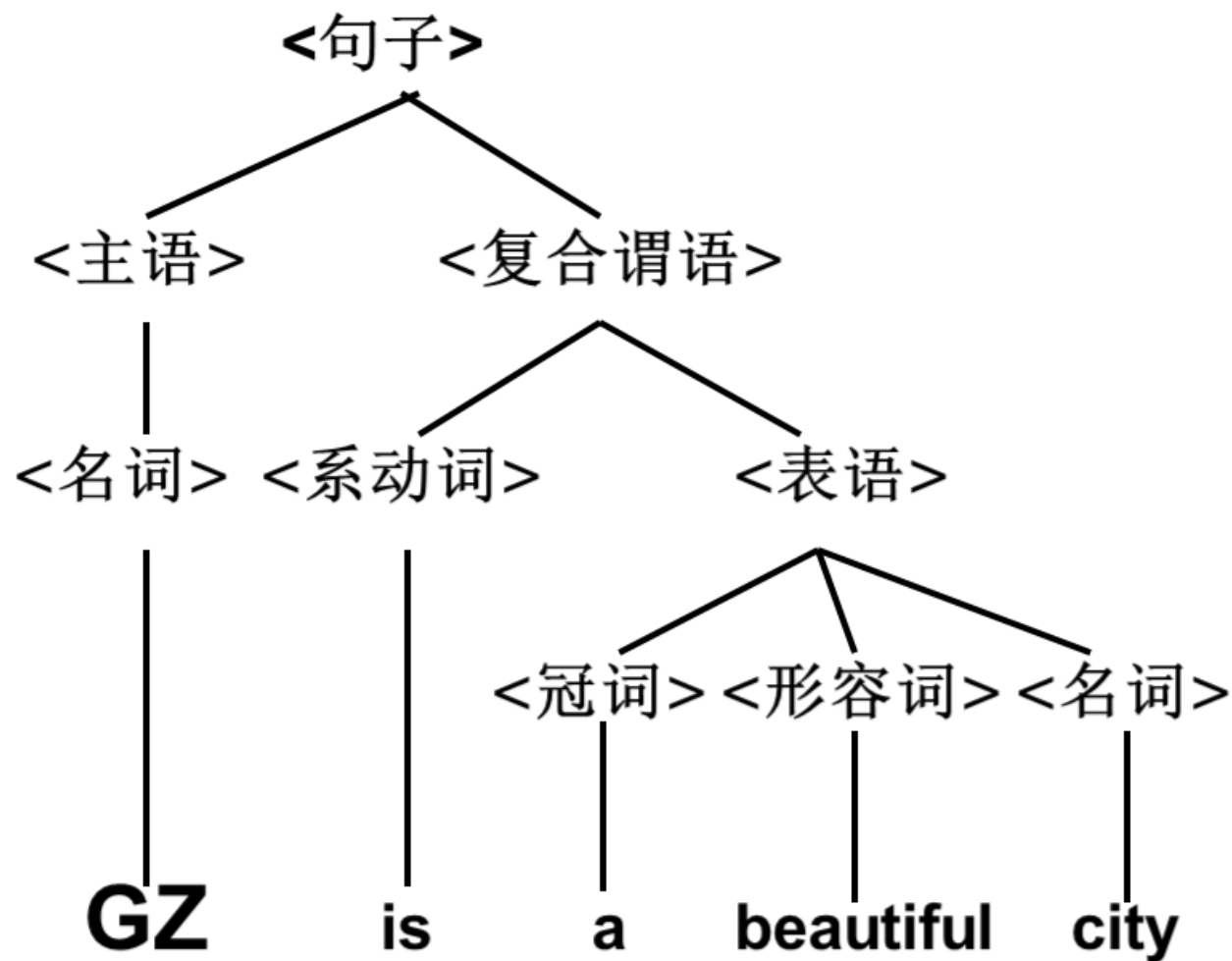
$O \rightarrow + | *$

语言的分析方法

- 自顶向下分析 (top-down parsing)
- 由底向上分析 (bottom-up parsing)

This line is a longer sentence





归约与推导

- **归约** 将产生式的右部 (body) 替换为产生式的左部 (head) .
- **推导** 将产生式的左部 (head) 替换为产生式的右部 (body) .

归约过程举例

对于CFG $G_{\text{exp}} = (\{E, O\}, \{ (,), +, *, v, d \}, P, E)$, P 为

(1) $E \rightarrow EOE$

(2) $E \rightarrow (E)$

(3) $E \rightarrow v$

(4) $E \rightarrow d$

(5) $O \rightarrow +$

(6) $O \rightarrow *$

递归推理出字符串 $v*(v+d)$ 的一个归约过程为

$$\begin{aligned} & v*(v+d) \xrightarrow{(4)} v*(v+E) \xrightarrow{(6)} vO(v+E) \xrightarrow{(3)} vO(E+E) \\ & \xrightarrow{(5)} vO(EOE) \xrightarrow{(1)} vO(E) \xrightarrow{(2)} vOE \xrightarrow{(3)} EOE \xrightarrow{(1)} E \end{aligned}$$

推导过程举例

对于CFG $G_{\text{exp}} = (\{E, O\}, \{ (,), +, *, v, d \}, P, E)$, P 为

$$(1) E \rightarrow EOE$$

$$(2) E \rightarrow (E)$$

$$(3) E \rightarrow v$$

$$(4) E \rightarrow d$$

$$(5) O \rightarrow +$$

$$(6) O \rightarrow *$$

递归推理出字符串 $v*(v+d)$ 的一个推导过程为

$$\begin{aligned} & E \xrightarrow{(1)} EOE \xrightarrow{(6)} E * E \xrightarrow{(2)} E * (E) \xrightarrow{(3)} v * (E) \\ & \xrightarrow{(1)} v * (EOE) \xrightarrow{(5)} v * (E + E) \xrightarrow{(3)} v * (v + E) \xrightarrow{(4)} v * (v + d) \end{aligned}$$

推导关系

对于 CFG $G = (V_N, V_T, P, S)$, 上述推导过程可用关系 \Rightarrow_G 描述. 设 $\alpha, \beta \in (V_N \cup V_T)^*$, $A \rightarrow \gamma$ 是一个产生式, 则定义

$$\alpha A \beta \Rightarrow_G \alpha \gamma \beta.$$

若 G 在上下文中是明确的, 则简记为 $\alpha A \beta \Rightarrow \alpha \gamma \beta$.

扩展推导关系到自反传递闭包

定义上述关系的自反传递闭包, 记为 $\xRightarrow{*}_G$, 可归纳定义如下:

基础 对任何 $\alpha \in (V_N \cup V_T)^*$, 满足 $\alpha \xRightarrow{*}_G \alpha$.

归纳 设 $\alpha, \beta, \gamma \in (V_N \cup V_T)^*$, 若 $\alpha \xRightarrow{*}_G \beta$, $\beta \Rightarrow_G \gamma$ 成立, 则

$$\alpha \xRightarrow{*}_G \gamma.$$

最左推导 (leftmost derivations)

若推导过程的每一步总是替换出现在最左边的非终结符，则这样的推导称为**最左推导**。

最左推导关系用 \Rightarrow_{lm} 表示，

自反传递闭包用 $\xRightarrow{*}_{lm}$ 表示。

如对于文法 G_{exp} ，下面是关于 $v*(v+d)$ 的一个最左推导：

$$\begin{aligned} E &\xRightarrow{*}_{lm} EOE \xRightarrow{*}_{lm} vOE \xRightarrow{*}_{lm} v * E \\ &\xRightarrow{*}_{lm} v * (E) \xRightarrow{*}_{lm} v * (EOE) \xRightarrow{*}_{lm} v * (vOE) \\ &\xRightarrow{*}_{lm} v * (v + E) \xRightarrow{*}_{lm} v * (v + d) \end{aligned}$$

$$E \rightarrow EOE$$

$$E \rightarrow (E)$$

$$E \rightarrow v$$

$$E \rightarrow d$$

$$O \rightarrow +$$

$$O \rightarrow *$$

最右推导 (rightmost derivations)

若推导过程的每一步总是替换出现在最右边的非终结符，
则这样的推导称为**最右推导**。

最右推导关系用 \Rightarrow_{rm} 表示，

其自反传递闭包用 $\xRightarrow{*}_{rm}$ 表示。

如对于文法 G_{exp} ，下面是关于 $v*(v+d)$ 的一个最右推导：

$$\begin{array}{lcl} E & \xRightarrow{*}_{rm} & EOE \xRightarrow{*}_{rm} EO(E) \xRightarrow{*}_{rm} EO(EOE) \\ & & \xRightarrow{*}_{rm} EO(EOd) \xRightarrow{*}_{rm} EO(E+d) \xRightarrow{*}_{rm} EO(v+d) \\ & & \xRightarrow{*}_{rm} E*(v+d) \xRightarrow{*}_{rm} v*(v+d) \end{array} \quad \begin{array}{l} E \rightarrow EOE \\ E \rightarrow (E) \\ E \rightarrow v \\ E \rightarrow d \\ O \rightarrow + \\ O \rightarrow * \end{array}$$

句型 (*sentential forms*)

设 CFG $G = (V_N, V_T, P, S)$, 称 $\alpha \in (V_N \cup V_T)^*$

为 G 的一个句型, 当且仅当 $S \xRightarrow{*} \alpha$.

若 $S \xRightarrow{*}_{\overline{lm}} \alpha$, 则 α 是一个左句型;

若 $S \xRightarrow{*}_{\overline{rm}} \alpha$, 则 α 是一个右句型.

若句型 $\alpha \in V_T^*$, 则称 α 为一个句子 (*sentence*).

上下文无关语言的定义

上下文无关文法的语言

设 CFG $G = (V_N, V_T, P, S)$, 定义 G 的语言为

$$L(G) = \{w \mid w \in V_T^* \wedge S \xRightarrow[G]{*} w\}$$

上下文无关语言 (context-free languages)

如果一个语言 L 是某个 CFG G 的语言, 即

$$L(G) = L,$$

则 L 是上下文无关语言.

问题：属于 VT^+ 的符号串 x ,是否一定属于 $L(G)$?

例2 设有文法 $G1=(VN, VT, P, A)$

其中, $VN=\{A\}$ $VT=\{a\}$

$P=\{A \rightarrow a\}$

试问：此文法描述的语言 $L(G1)=?$

解：由于从开始符号 A 出发, 只能推导出一个句子 a ,
所以 $L(G1)=\{a\}$, 因此文法 $G1$ 所定义的语言 $L(G1)$ 是
有穷语言。

由于 $VT^+ = \{a, aa, aaa, \dots\}$, 因此 $L(G1)$ 只是 VT^+
的一个真子集。

例3 设有文法 $G_2 = (VN, VT, P, A)$

其中, $VN = \{A\}$ $VT = \{a\}$

$P = \{A \rightarrow Aa, A \rightarrow \varepsilon\}$

试问: 此文法描述的语言 $L(G_2) = ?$

解: 由于从开始符号 A 出发, 可以有

$A \Rightarrow Aa \Rightarrow \varepsilon a = a$

$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow \varepsilon aa = aa$

$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow Aaaa = \varepsilon aaa = aaa$

即 $L(G_2) = \{a, aa, aaa, aaaa, \dots\} = \{a^n, n \geq 1\}$ 。

因此文法 G_1 所定义的语言 $L(G_2)$ 是无穷语言。

结论

最左推导、最右推导、最左归约、最右归约

不难发现：

最左推导 = 最右归约

最右推导 = 最左归约

规范推导 = 最右推导，最左归约 = 规范归约

可以说，凡是句子，必定存在规范归约是自底向上分析技术可行的依据。

结论

- 语言是一切句子的集合；
- 程序设计语言是一切程序的集合；
- 程序是(程序设计) 语言的句子

几个基本概念

- **直接递归**：设 G 是一文法， $U \rightarrow u$ 是 G 的一个规则，如果 u 具有 $\alpha U \beta$ 的形式，其中 α 、 β 不同时为 ε ，则称产生式 $U \rightarrow u$ 是**直接递归**的；

- **递归**：若存在推导

$$U \rightarrow u \Rightarrow^* \alpha U \beta$$

则称规则 $U \rightarrow u$ 是**递归**的。

- **直接左递归和左递归的规则**：当 $\alpha = \varepsilon$ 而 $\beta \neq \varepsilon$ 时，则将规则 $U \rightarrow u$ 分别称直接左递归和左递归规则。
- **递归文法**：文法中至少含有一个递归的非终结符。

语言分析在计算机的实现

- 1.文法的存储结构
- 2.推导过程的存储结构

1.文法的存储结构

G[E]:

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow i$

$\langle \text{句子} \rangle ::= \langle \text{主语} \rangle \langle \text{复合谓语} \rangle$

$\langle \text{主语} \rangle ::= \langle \text{名词} \rangle$

$\langle \text{复合谓语} \rangle ::= \langle \text{系动词} \rangle \langle \text{表语} \rangle$

一种是数组表示。

一种是链表表示。

数组表示:

左部	右部符号串	右部长度
----	-------	------

相应的C语言数据结构定义:

```
typedef struct
```

```
{ 符号 左部符号;
```

```
    符号 右部符号串[MaxRightPartLength+1];
```

```
    int 右部长度;
```

```
} 规则;
```

```
规则 文法[MaxRuleNum+1];
```

其中, typedef char 符号[MaxLength+1];

符号 非终结符号集[MaxVnNum+1];

符号 终结符号集[MaxVtNum+1];

为便于处理，以符号的序号代替符号本身：

```
typedef struct
```

```
{ int 左部符号序号;
```

```
    int 右部符号串[MaxRightPartLength+1];
```

```
    int 右部长度;
```

```
} 规则;
```

```
规则 文法[MaxRuleNum+1];
```

文法G在计算机内的存储表示:

101	101, 1, 102	3
101	102	1
102	102, 2, 103	3
102	103	1
103	3, 101, 4	3
103	5	1

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow i$

规则 $E \rightarrow E+T$:在计算机内的存储表示:

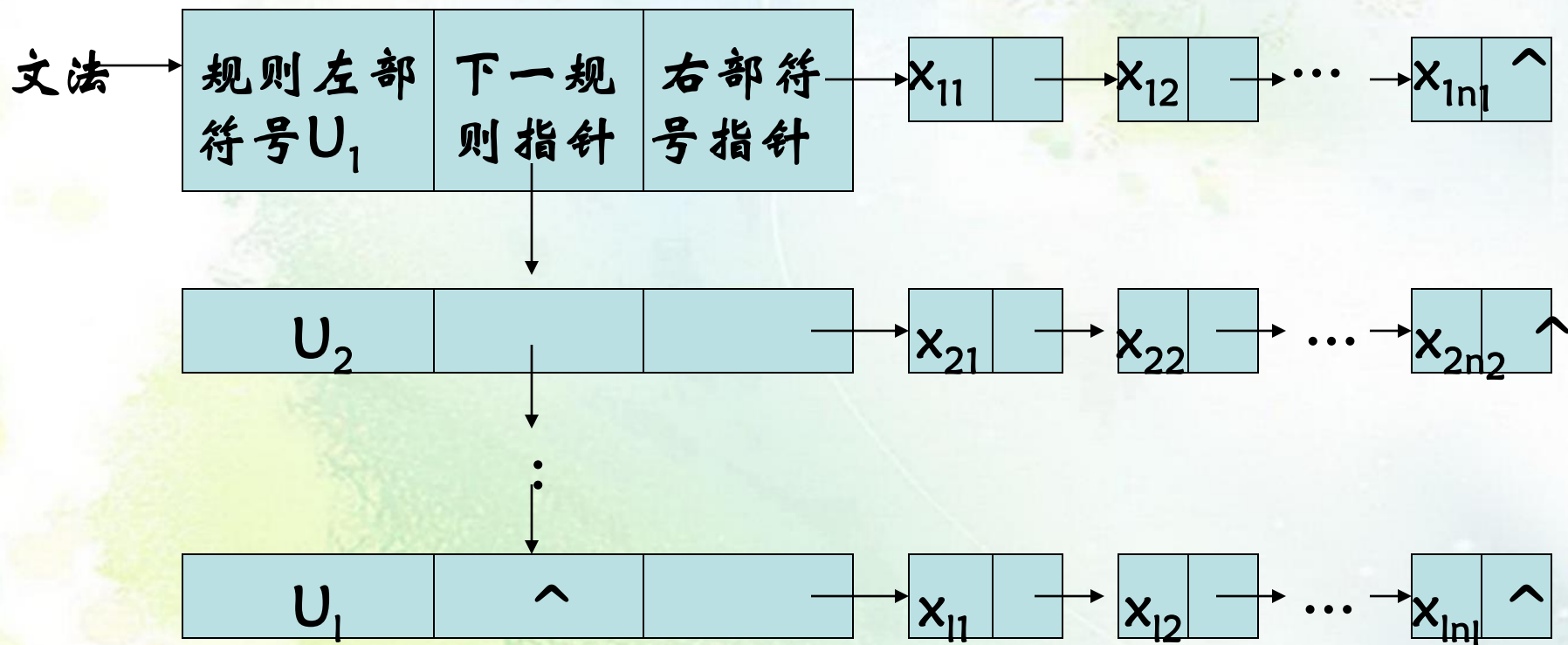
文法[1]: { 101, { 101, 1, 102 }, 3 }

规则 $E \rightarrow T$ 在计算机内的存储表示:

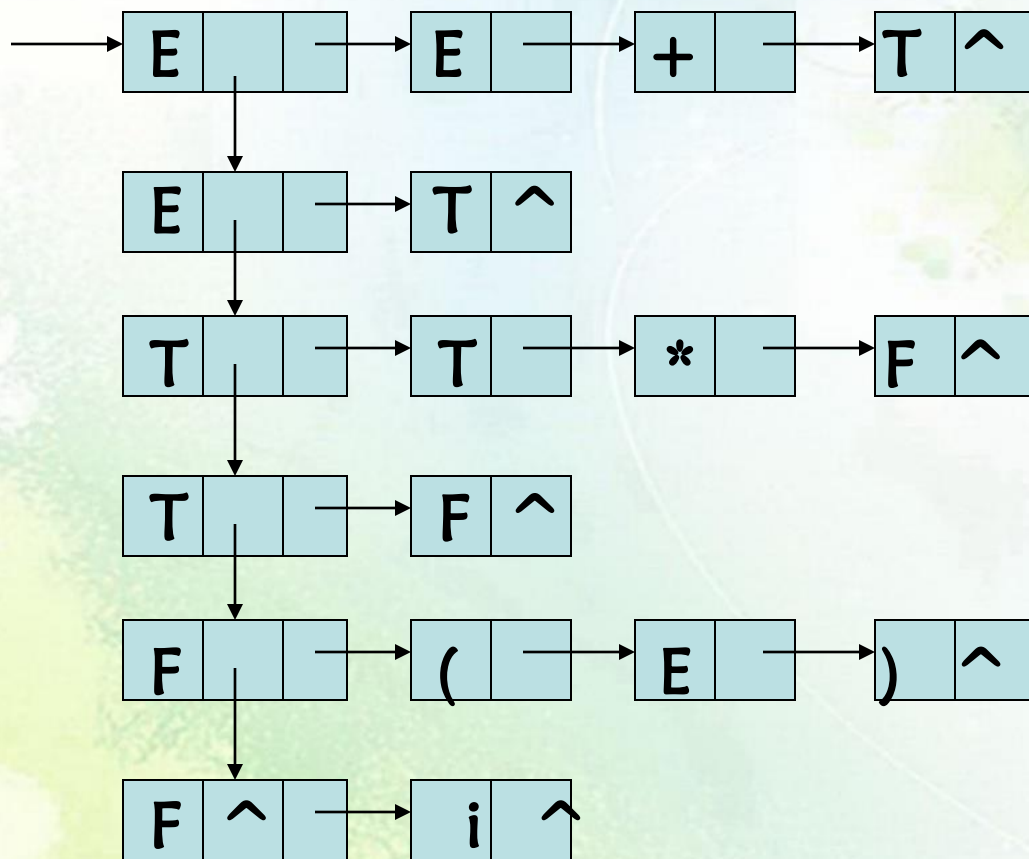
文法[2]: { 101, { 102 }, 1 }

$T \in V_T$: T在 V_T 中的序号, $U \in V_N$: U在 V_N 中的序号+100

文法的链式表示:



文法G



2. 推导过程的存储结构

推导过程的构造: 句子生成的过程

重点: 推导过程在计算机内的存储表示。

$$\begin{aligned} G[E]: \quad & E \rightarrow E+T \mid T \\ & T \rightarrow T * F \mid F \\ & F \rightarrow (E) \mid i \end{aligned}$$

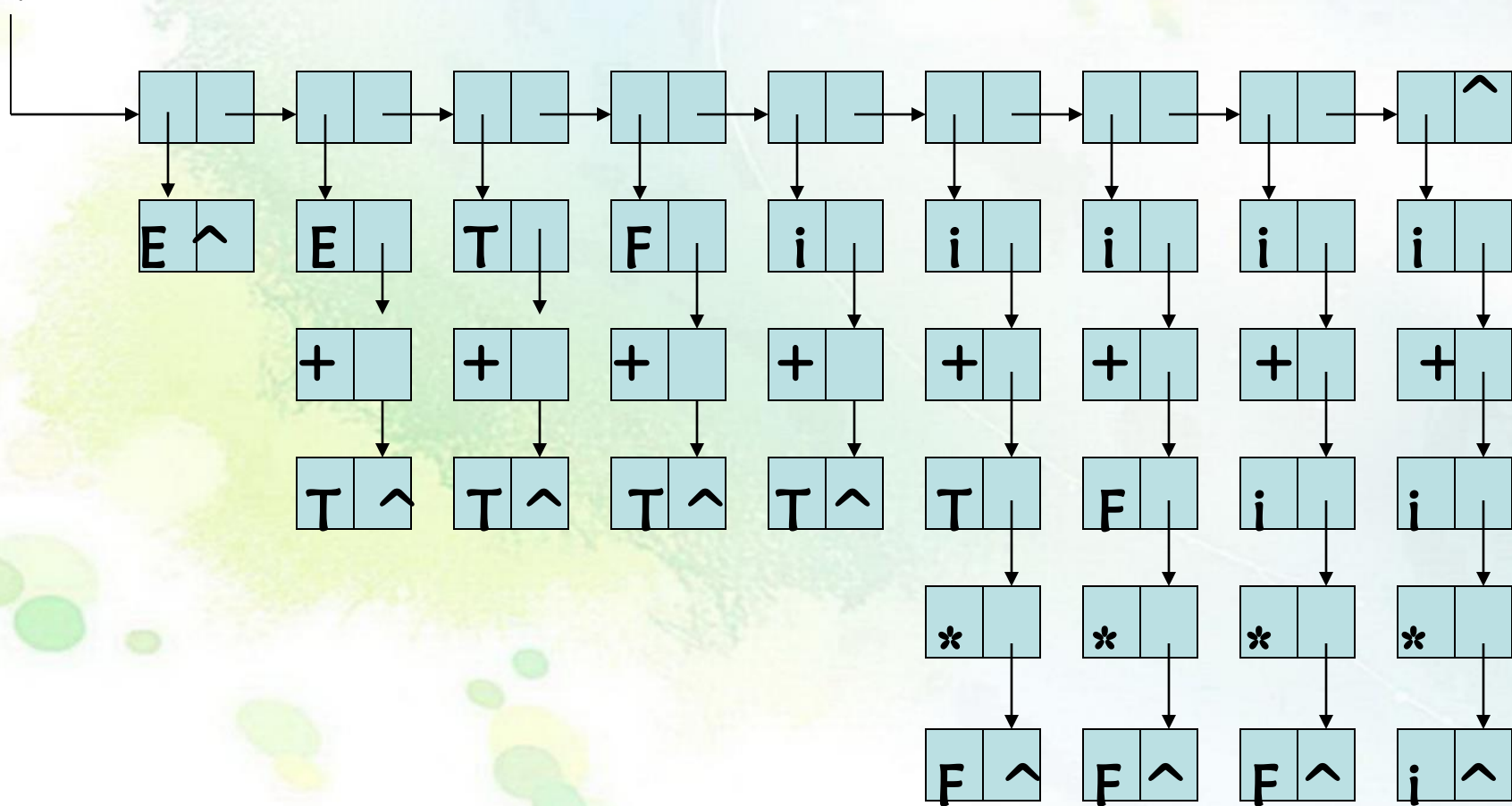
$$\begin{aligned} E &\Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow i+T \Rightarrow i+T * F \\ &\Rightarrow i+F * F \Rightarrow i+i * F \Rightarrow i+i * i \end{aligned}$$

2. 推导过程的存储结构

推导过程的构造: 句子生成的过程

重点: 推导过程在计算机内的存储表示。

推导



其中包含两类结点，一类链接结点结构形如：

句型首符号结点指针	下一句型链头指针
-----------	----------

另一类符号结点结构形如：

文法符号序号	后继符号结点指针
--------	----------

链接结点可采用结构类型定义如下：

```
typedef struct 链接结点
{ 符号结点类型 *句型首符号结点指针;
  struct 链接结点 *下一句型链头指针;
} 链接结点类型;
```

符号结点可采用结构类型定义如下：

```
typedef struct 符号结点
{ int 文法符号序号;
  struct 符号结点 *后继符号结点指针;
} 符号结点类型;
```

其中，为简单起见，文法符号用文法符号的序号代替。

推导过程分析

G[E]:
 $E \rightarrow E+T \mid T$
 $T \rightarrow T*F \mid F$
 $F \rightarrow (E) \mid i$

$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow i+T \Rightarrow i+T*F$
 $\Rightarrow i+F*F \Rightarrow i+i*F \Rightarrow i+i*i$

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*i \Rightarrow E+F*i \Rightarrow E+i*i$
 $\Rightarrow T+i*i \Rightarrow F+i*i \Rightarrow i+i*i$

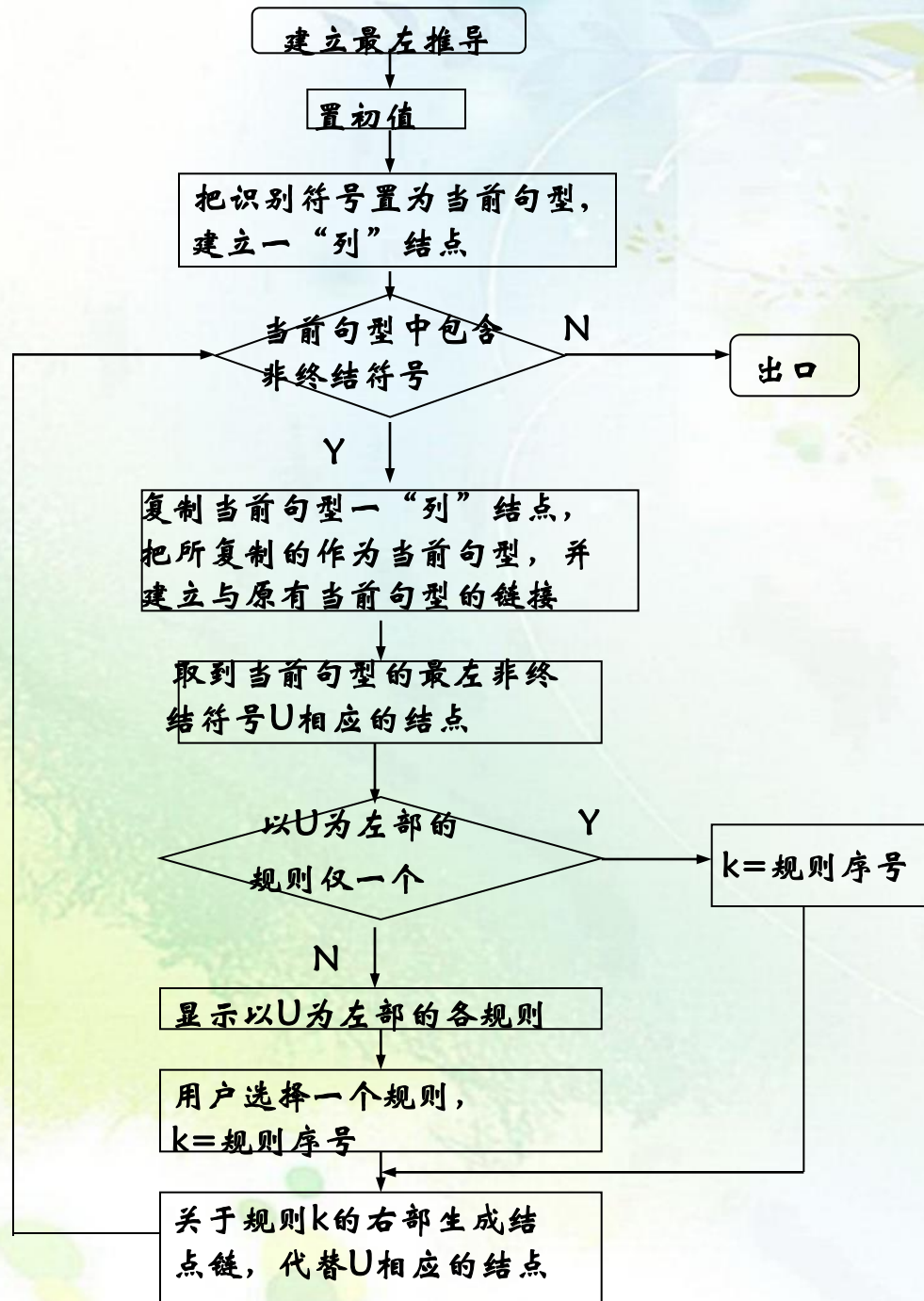
$E \Rightarrow E+T \Rightarrow T+T \Rightarrow T+T*F \Rightarrow F+T*F \Rightarrow F+F*F$
 $\Rightarrow i+F*F \Rightarrow i+F*i \Rightarrow i+i*i$

显然每一“列”（垂直链）中各结点相应的符号组成一个句型。

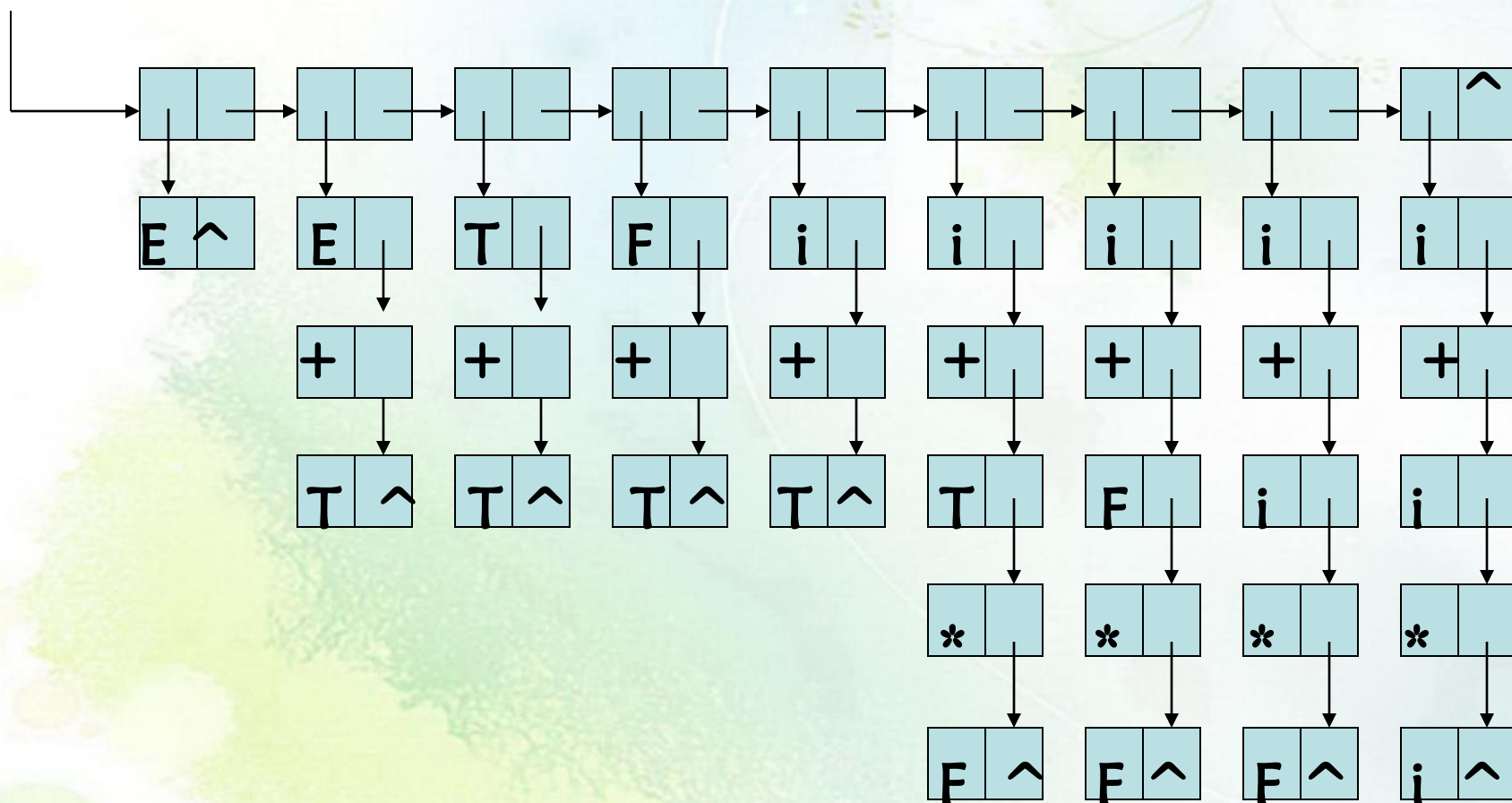
一个推导，其中的每一步直接推导，总是把相应句型中最左的非终结符号，替换为其规则右部符号串，称为**最左推导**。

一个推导，其中的每一步直接推导，总是把相应句型中最右的非终结符号，替换为其规则右部符号串，称为**最右推导**。

这易于用C语言结构类型实现。以最左推导的建立为例，程序控制流程示意图如下。



推导



文法与语言的 Chomsky 分类方法

— 文法 (grammar) 是一个四元组

$$G = (V_N, V_T, P, S),$$

V_N 、 V_T 、 P 及 S 的含义如前. Chomsky 通过对产生式施加不同的限制, 把文法及其对应的语言分成四种类型, 即 0 型、1 型、2 型和 3 型.

文法与语言的 Chomsky 分类方法

— 0 型文法

0 型文法 $G = (V_N, V_T, P, S)$ 的产生式形如

$$\alpha \rightarrow \beta,$$

其中 $\alpha, \beta \in (V_N \cup V_T)^*$, 但 α 中至少包含一个非终结符.

能够用 0 型文法定义的语言称为 0 型语言或递归可枚举语言.

文法与语言的 Chomsky 分类方法

– 1 型文法

1 型文法 $G = (V_N, V_T, P, S)$ 的产生式形如

$$\alpha \rightarrow \beta, \text{ 且 } |\alpha| \leq |\beta|,$$

仅 $S \rightarrow \varepsilon$ 例外, 且要求 S 不得出现在任何产生式的右部。

1 型文法也称为 **上下文有关文法** (context-sensitive grammars)。

能够用 1 型文法定义的语言称为 1 型语言或上下文有关语言。

1型文法

例：1型（上下文有关）文法

文法 $G[S]$:

$S \rightarrow CD$

$aAb \rightarrow bCd$

$C \rightarrow aCA$

$bBa \rightarrow aCd$

$C \rightarrow bCB$

$aBb \rightarrow bDd$

$C \rightarrow a$

$D \rightarrow b$

$bAa \rightarrow bDd$

文法与语言的 Chomsky 分类方法

– 2 型文法

2 型文法 $G = (V_N, V_T, P, S)$ 的产生式形如

$$A \rightarrow \beta,$$

其中 $A \in V_N$, $\beta \in (V_N \cup V_T)^*$.

2 型文法也称为上下文无关文法.

能够用 2 型文法定义的语言称为 2 型语言, 或上下文无关语言.

2型文法

例：2型（上下文无关）文法

文法 $G[S]$:

$$S \rightarrow AB$$
$$A \rightarrow BS \mid 0$$
$$B \rightarrow SA \mid 1$$

文法与语言的 Chomsky 分类方法

— 3 型文法

3 型文法 $G = (V_N, V_T, P, S)$ 的产生式形如

$A \rightarrow aB$ 或 $A \rightarrow a$, 【右线性规则】

其中 $A, B \in V_N$, $a \in V_T \cup \{\varepsilon\}$.

能够用 3 型文法定义的语言称为 3 型语言, 或正规语言.

3 型文法也称为正则文法 (或正规文法).

产生式也可以形如

$A \rightarrow Ba$ 或 $A \rightarrow a$, 【左线性规则】

3型文法

$G[S]:$

$S \rightarrow A0 \mid B1 \mid 0$

$A \rightarrow A0 \mid B1 \mid S0$

$B \rightarrow B1 \mid 1 \mid 0$

$G[A]:$

$A \rightarrow aT$

$A \rightarrow a$

$T \rightarrow aT$

$T \rightarrow dT$

$T \rightarrow a$

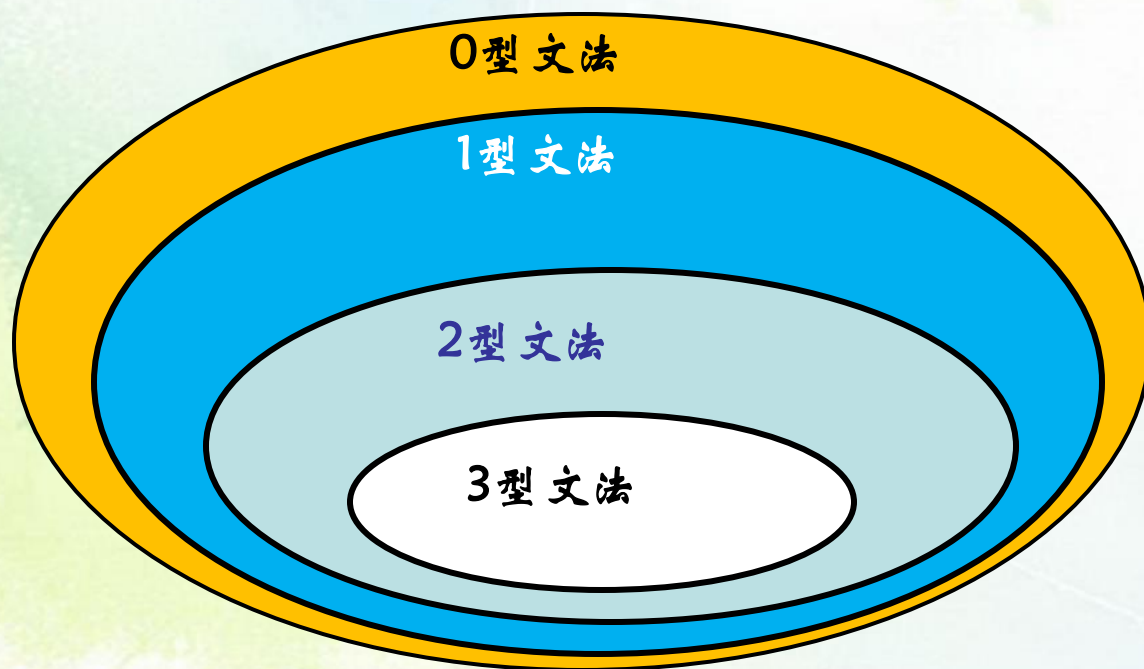
$T \rightarrow d$

文法与语言的 Chomsky 分类方法

分类	语言名称	语法	识别器	产生式形式
3	正规语言 或正则语言	正则文法 或正规文法 Regular	有穷状态自动机 Finite-State Automaton	$A \rightarrow aB$ 或 $A \rightarrow a$ 其中: $A, B \in N, a \in \Sigma$ 也可具有形式: $A \rightarrow Ba$ 或 $A \rightarrow a$ 但两种形式不能同时兼存在。
2	上下文 无关语言	上下文无关文法 Context-Free	下推自动机 Push-Down Automaton	$A \rightarrow \alpha$ 其中: $A \in N, \alpha \in (N \cup \Sigma)^*$
1	上下文 有关语言	上下文有关文法 Context-Sensitive	线性限界自动机 Linear-Bounded Automaton	$\alpha \rightarrow \beta$ 其中: $\alpha, \beta \in (N \cup \Sigma)^*$, 且 $ \alpha \leq \beta $
0	递归可枚举 语言	0 型文法 Unrestricted	图灵机 Turing Machine	$\alpha \rightarrow \beta$ 其中: $\alpha, \beta \in (N \cup \Sigma)^*$

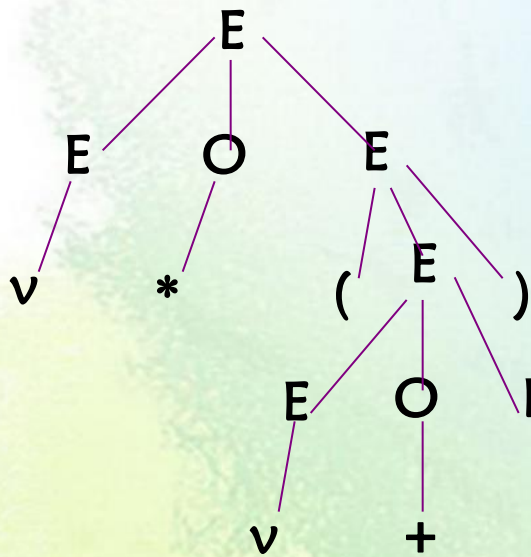
文法与语言的 Chomsky 分类方法

四类文法之间的逐级“包含”关系



分析树

— 归约过程自底而上构造了一棵树 如对于文法 G_{exp} ，关于 $v*(v+d)$ 的一个归约过程可以认为是构造了如下一棵树：



(1) $E \rightarrow EOE$

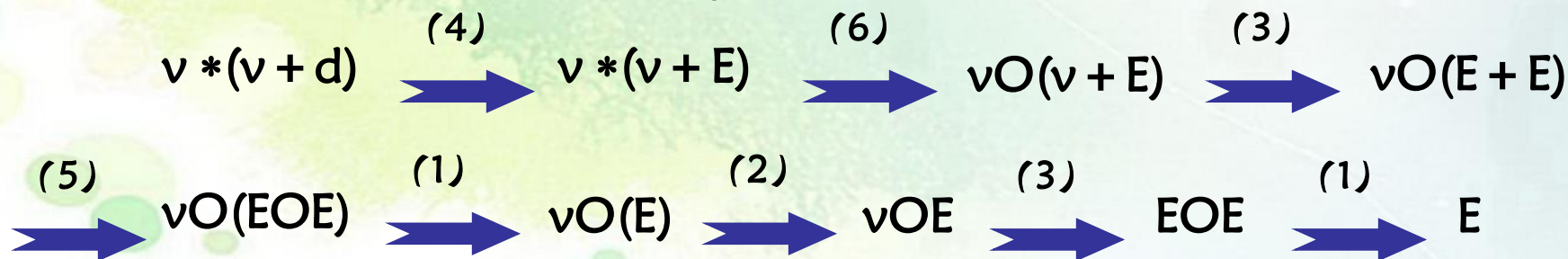
(2) $E \rightarrow (E)$

(3) $E \rightarrow v$

(4) $E \rightarrow d$

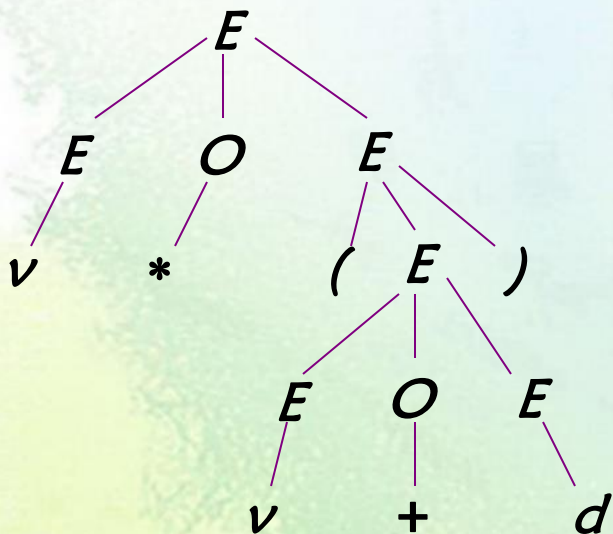
(5) $O \rightarrow +$

(6) $O \rightarrow *$



分析树

- 推导过程自顶而下构造了一棵树 如对于文法 G_{exp} , 关于 $v*(v+d)$ 的一个推导过程可以认为是构造了如下一棵树:



$$(1) E \rightarrow EOE$$

$$(2) E \rightarrow (E)$$

$$(3) E \rightarrow v$$

$$(4) E \rightarrow d$$

$$(5) O \rightarrow +$$

$$(6) O \rightarrow *$$

$$\begin{aligned}
 & E \xrightarrow{(1)} EOE \xrightarrow{(6)} E * E \xrightarrow{(2)} E * (E) \xrightarrow{(3)} v * (E) \\
 & \xrightarrow{(1)} v * (EOE) \xrightarrow{(5)} v * (E + E) \xrightarrow{(3)} v * (v + E) \xrightarrow{(4)} v * (v + d)
 \end{aligned}$$

分析树

对于 CFG $G = (V_N, V_T, P, S)$, 语法分析树是满足下列条件的树:

(1) 每个内部结点由一个非终结符标记。

(2) 每个叶结点或由一个非终结符, 或由一个终结符, 或由 ε 来标记。但标记为 ε 时, 它必是其父结点唯一的子。

(3) 如果一个内部结点标记为 A , 而其孩子从左至右分别标记为 X_1, X_2, \dots, X_k , 则 $A \rightarrow X_1 X_2 \dots X_k$ 是 P 中的一个产生式。

(4) 如果树的所有端末节点 (叶子) 上的标记从左向右排列为字符串 w , 则 w 是 G 的 **句型**, 若 w 中仅含终结符, 则它为 G 所产生的 **句子**。

注意: 只有 $k=1$ 时上述 X_i 才有可能为 ε , 此时结点 A 只有唯一的子, 且 $A \rightarrow \varepsilon$ 是 P 中的一个产生式。

问题：对于一个给定的文法和一个句型，如何构造相应的分析树？

例 $G[S]$:

$$S \rightarrow V = E$$

$$V \rightarrow i$$

$$E \rightarrow F + F \mid F$$

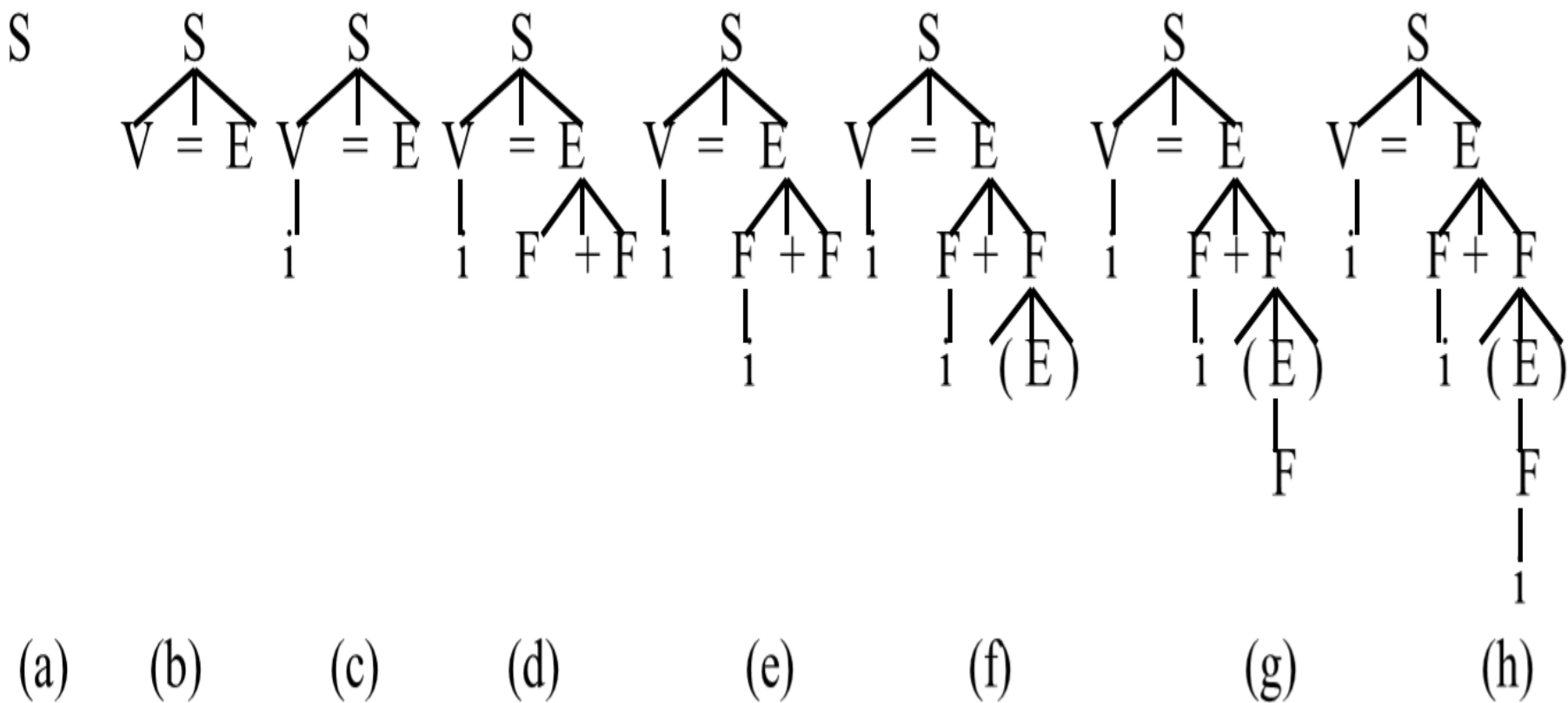
$$F \rightarrow (E) \mid i$$

分析所输入的符号串 $i = i + (i)$ 是否为其句子，并画出相应的分析树。

1. 自顶向下
2. 自底向上

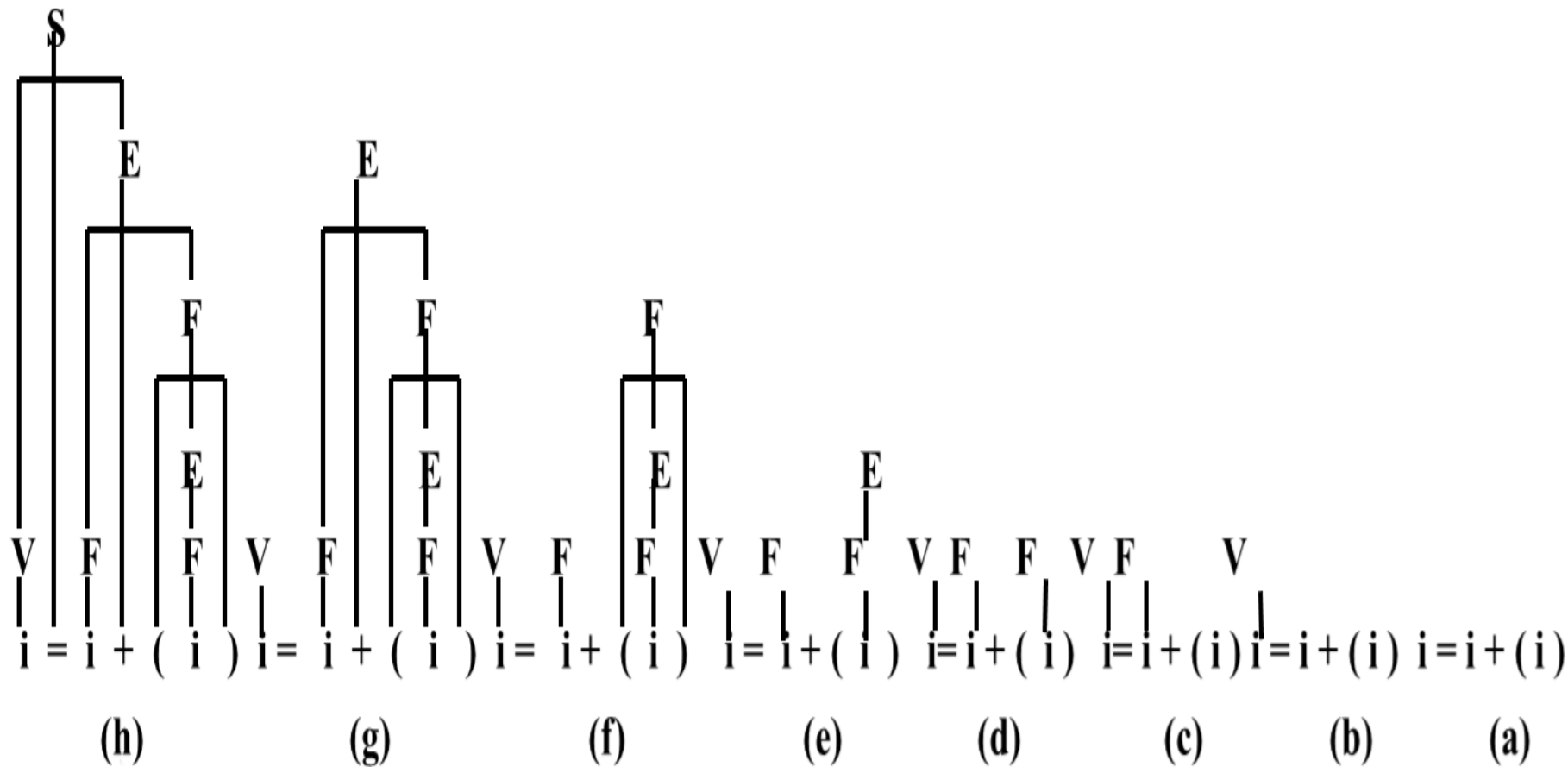
自顶向下分析：不断进行直接推导的过程。

构造方向 →



自底向上分析：不断进行直接归约的过程。

构造方向 ←



分析树的存储结构

首先结合例子考察语法分析树应有怎样的数据结构，然后再考虑怎样构造。

例 设文法 $G[E]$:

$E ::= E + T \mid E - T \mid T$

$T ::= T * F \mid T / F \mid F$

$F ::= (E) \mid i$

输入符号串是 $i-i*i$,

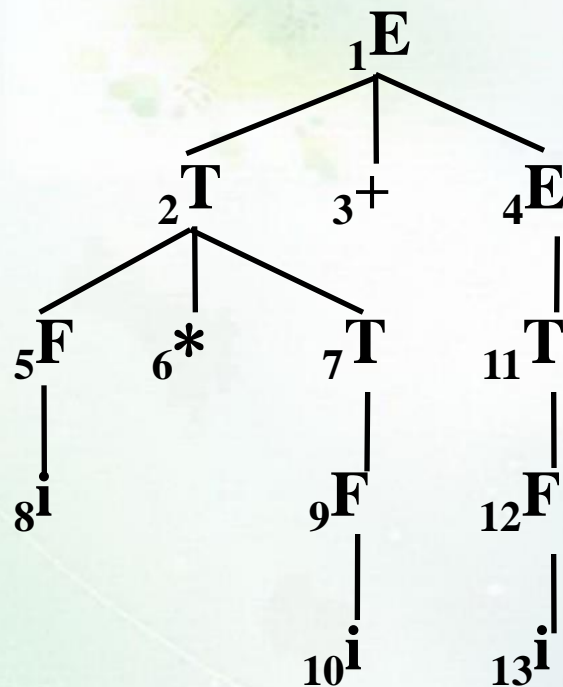
推导为:

$E \Rightarrow T + E \Rightarrow F * T + E \Rightarrow i * T + E$

$\Rightarrow i * F + E \Rightarrow i * i + E \Rightarrow i * i + T$

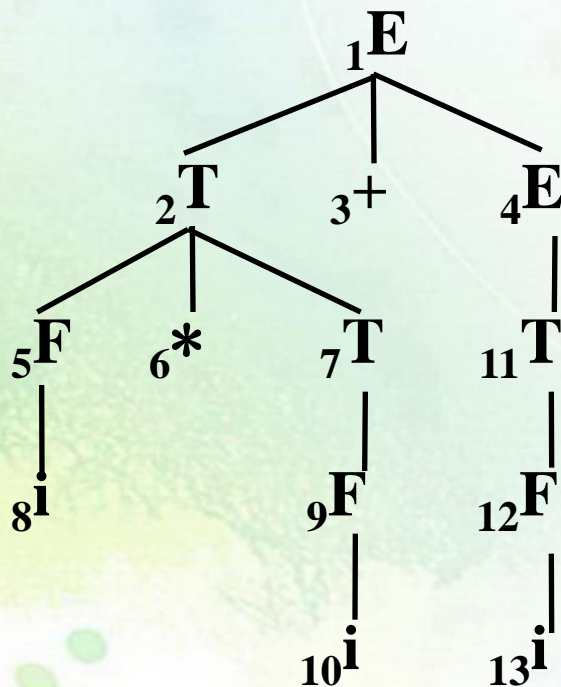
$\Rightarrow i * i + F \Rightarrow i * i + i$

可构造语法分析树如图:

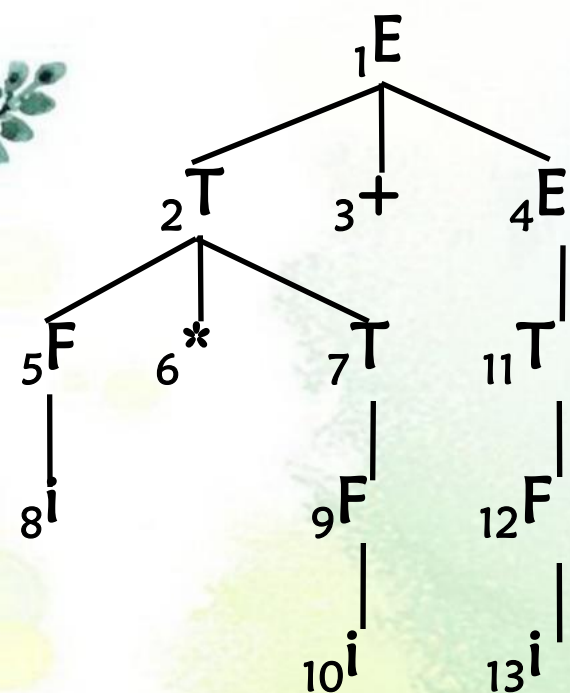


确定每个结点在语法分析树中的位置，
包括各个结点之间的相互关系，便确定了语法分析树，
因此为语法分析树结点可设计如下的数据结构：

结点序号	文法符号序号	父结点序号	左兄结点序号	右子结点序号
------	--------	-------	--------	--------



按推导生成的分析树及其存储表示



结点序号	目标	父结点	左兄结点	右子结点
1	E	0	0	4
2	T	1	0	7
3	+	1	2	0
4	E	1	3	11
5	F	2	0	8
6	*	2	5	0
7	T	2	6	9
8	i	5	0	0
9	F	7	0	10
10	i	9	0	0
11	T	4	0	12
12	F	11	0	13
13	i	12	0	0

(a) 分析树

(b) 分析树的存储表示

```
typedef struct  
{ int 结点序号;  
  int 文法符号序号;  
  int 父结点序号;  
  int 左兄结点序号;  
  int 右子结点序号;  
} 结点类型;
```

结点类型 语法分析树[MaxNodeNum];

构造分析树的步骤如下:

步骤1 以识别符号 Z 建立根结点, 序号为1, 且以这仅包含一个非终结符号的句型 Z 作为当前句型。

步骤2 从当前句型中找出最左的非终结符号 U , 显示以 U 为左部的一切规则, 根据所给的输入符号串, 选择其中的一个规则 $U ::= X_1X_2\cdots X_m$, 以 U 为分支名字结点, 以 $X_1X_2\cdots X_m$ 作为分支结点符号串, 构造分支。建立父子兄弟结点关系。

步骤3 重复步骤2, 直到当前句型中不再包含非终结符号, 分析树构造结束, 最终的分析树为所求。

总结

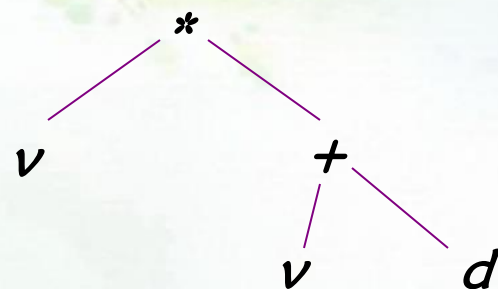
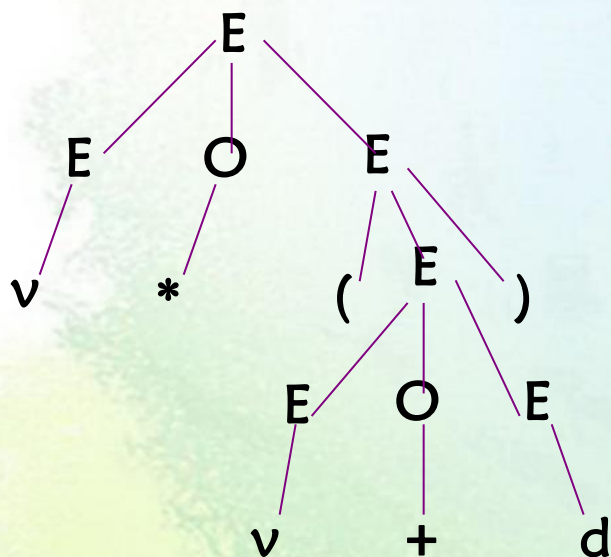
(1)优点:分析树可以反映推导的全过程, 信息齐全。

(2)缺点: 过于复杂,耗费空间。

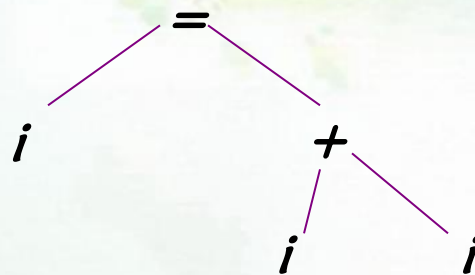
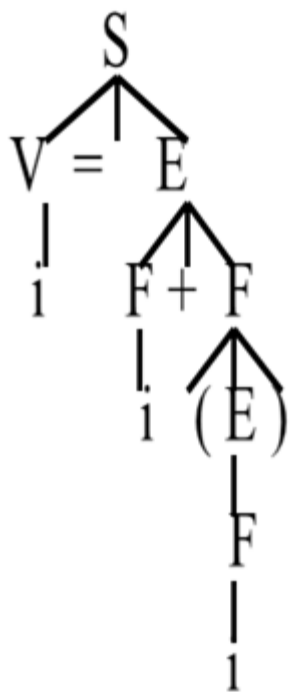
解决方法

- 压缩——只存储有用信息——对于后续阶段来说
- 语法树

分析树与语法树



分析树与语法树



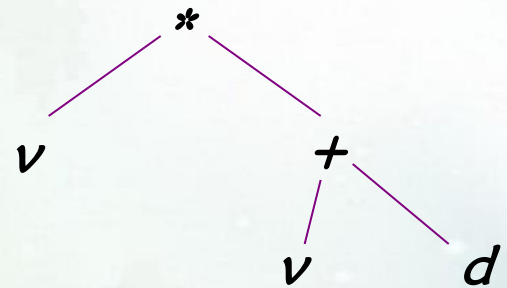
语法树

特点:

- 分析树表示所有推导步骤;
- 语法树只保留对后续分析有用的信息,因此无法还原记号序列, 比分析树效率更高。

- 算术表达式对应语法树的C语言描述:

```
typedef enum { Plus, Minus, Times, Division } OpKind;
typedef enum { OpKind, ConstKind, VarKind } ExpKind;
typedef struct streenode {
    ExpKind kind;
    OpKind op;
    struct streenode *lchild, *rchild;
    int val;
    char varname[20];
} STreeNode;
typedef STreeNode *SyntaxTree;
```



条件判断语句的语法树

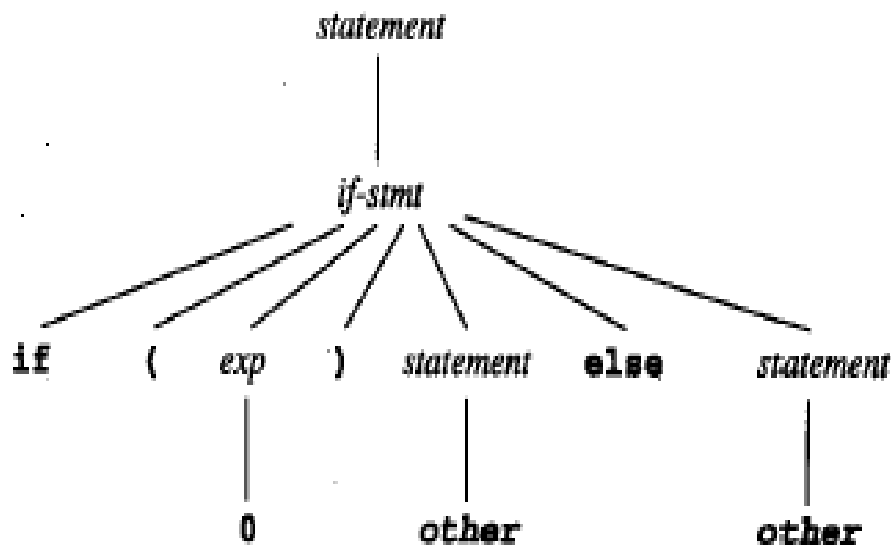
例. if语句的文法:

$statement \rightarrow if-stmt \mid other$

$if-stmt \rightarrow if(exp) statement \mid if(exp) statement else statement$

$exp \rightarrow 0 \mid 1$

串: if (0) other else other



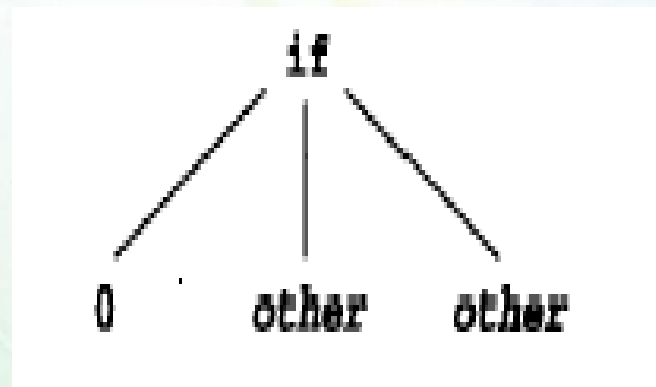
问题:分析该树中对后续分析阶段有用的信息有哪些?

if语句的语法树结构

(1)测试表达式

(2)then 部分

(3)else部分(如果出现)



if语句的语法树结构

```
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;
```


TINY语言的各语句的语法树结构

如：if, repeat, read, write 等语句的语法树结构

详细见参考书：P98的C语言表示

语法树示例见参考书 P100的图3-6和程序清单3-3。

文法的二义性

- 对于一个文法的同一个句子而言，若存在两个不同的语法树与之对应，我们就称该文法是二义性的。

例. $G3 = (\{\text{if, then, else, } e, a\}, \{S\}, P, S)$

$G3[S]$:

$S \rightarrow \text{if } e \text{ then } S$

| $\text{if } e \text{ then } S \text{ else } S$

| a

句子 $\text{if } e \text{ then if } e \text{ then } a \text{ else } a$

$S \Rightarrow \text{if } e \text{ then } S$

$\Rightarrow \text{if } e \text{ then if } e \text{ then } S \text{ else } S$

$\Rightarrow \text{if } e \text{ then if } e \text{ then } a \text{ else } S$

$\Rightarrow \text{if } e \text{ then if } e \text{ then } a \text{ else } a$

最左推导1

$S \Rightarrow \text{if } e \text{ then } S \text{ else } S$

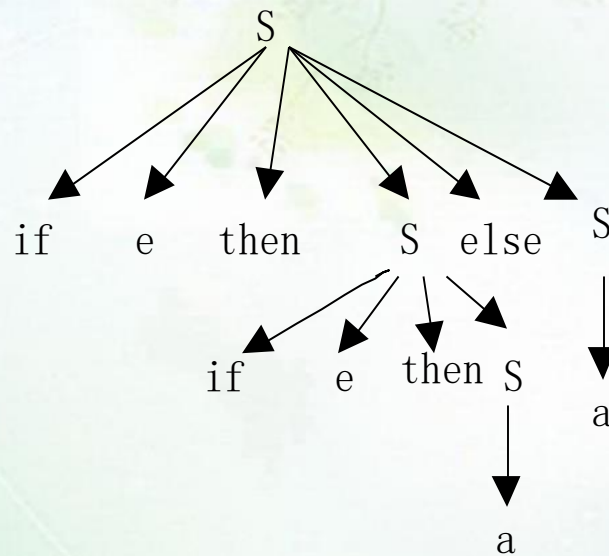
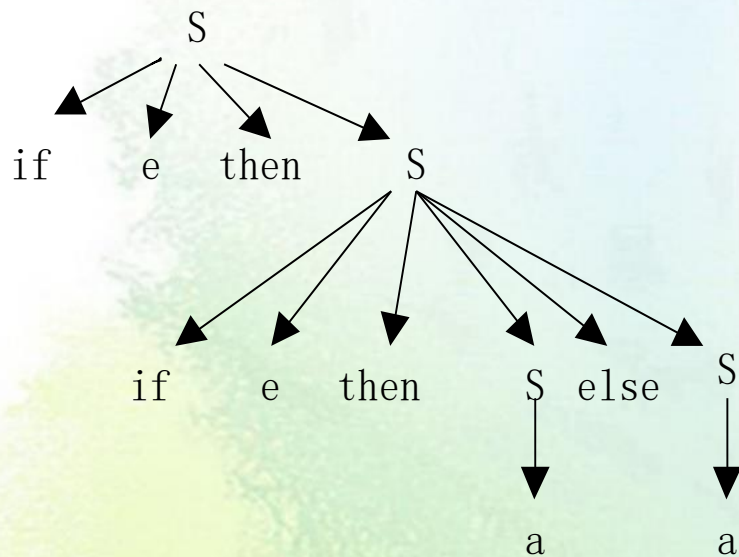
$\Rightarrow \text{if } e \text{ then if } e \text{ then } S \text{ else } S$

$\Rightarrow \text{if } e \text{ then if } e \text{ then } a \text{ else } S$

$\Rightarrow \text{if } e \text{ then if } e \text{ then } a \text{ else } a$

最左推导2

• 上例的两棵分析树



文法的二义性

- **二义性的判定** 一个 CFG 是否为二义的问题是不可判定的，即不存在解决该问题的算法。
- **消除二义性** 没有通用的办法可以消除文法的二义性。但在实践中，对于常用的文法，可以找到特定的消除歧义性的办法。

注意事项

对于程序设计语言来说，重要的是：**描述它的文法应是无二义性的。**

一个语言，可以为它设计二义性的文法，也可以为它设计无二义性的文法，因此一般说，讨论语言的二义性是无意义的。重要的是对同一个语言，为它构造无二义性文法，如果构造的是二义性文法，**应设法在不改变语言的前提下，把二义性文法等价变换成无二义性文法。**

鉴于二义性不可判定，所能做的是寻找一组**充分条件**，使得满足这些条件的文法必定是无二义性的。注意，这些条件只是充分条件，未必是无二义性的必要条件。

悬挂else问题的解决方法

例. $G3 = (\{ \text{if, then, else, } e, a \}, \{ S \}, P, S)$

$G3[S]$:

$S \rightarrow \text{if } e \text{ then } S$

| $\text{if } e \text{ then } S \text{ else } S$

| a

悬挂else问题的解决方法

- **方法1:** 设置一个限制规则, 在分析程序中实现

即: else 要与最近的上一个未被匹配的if匹配.

- 方法2:改造文法:

$S \rightarrow \text{matched-stmt} \mid \text{unmatched-stmt}$

$\text{matched-stmt} \rightarrow \text{if } e \text{ then matched-stmt else}$
 $\text{matched-stmt} \mid a$

$\text{unmatched-stmt} \rightarrow \text{if } e \text{ then } S \mid \text{if } e \text{ then}$
 $\text{matched-stmt} \text{ else unmatched-stmt}$

- **方法3:重新设计书写语法**

- **具体做法1:** else部分一定要出现

- **注意:**

该办法已在LISP和其他**函数**语言中用到了(但须返回一个值)。

- **具体做法2：**使用一个if匹配的關鍵字来作为语句的结束。

如： if $x \neq 0$ then

 if $y = 1/x$ then $ok := \text{true}$;

 else $z := 1/x$;

 end if;

end if;

注意： Algol 68、Ada、visual basic、visual foxpro均使用类似的做法。

悬挂else问题的解决方法

- 上述做法对应文法的描述为:

if-stmt → if e then *state-seq* end if |

if e then *state-seq* else *state-seq* end if

如何为语言构造文法

例: $L1 = \{ a^i b^j c^k \mid i, j, k \geq 1 \}$

$S \rightarrow A$

$A \rightarrow aA \mid aB'$

$B \rightarrow bB \mid bC$

$C \rightarrow cC$

例：构造语言 $L_2 = \{ ab^i \mid i \geq 0 \}$ 的文法

例. $V_T = \{a, b\}$ 的句子 S 的集合是由一个 b 及在其前后有相同数目的 a 组成:

$$S = \{ b, aba, aabaa, aaabaaa, \dots \} = \{ a^n b a^n \mid n \geq 0 \}$$

语言的等价

设 G_1 、 G_2 为两个文法,若它们所产生的语言相等,即 $L(G_1) = L(G_2)$,则称 G_1 和 G_2 等价。

一个问题

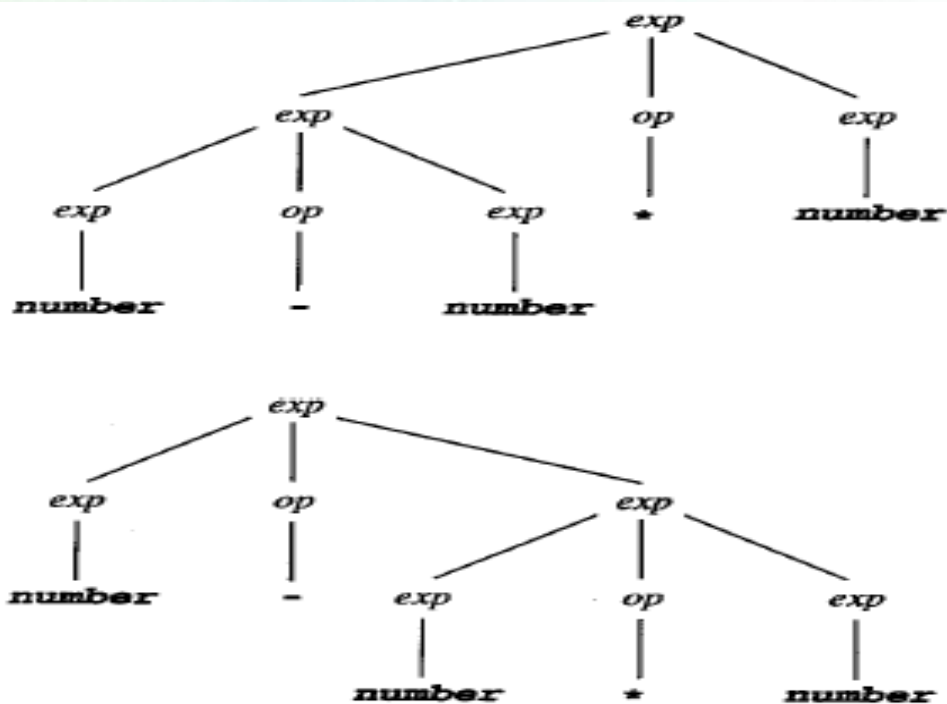
- 如何用文法规则描述算术表达式?

问题分析

$exp \rightarrow exp\ op\ exp \mid (exp) \mid number$

$op \rightarrow + \mid - \mid * \mid /$

对于串34-3*42，则分析树有：



文法改造过程

$exp \rightarrow exp \ op \ exp \mid (exp) \mid number$

$op \rightarrow + \mid - \mid * \mid /$

- **步骤(1)**:构造能反映运算符间优先关系的规则

即+ - 后于* / 再后于()

- 由于在树型中离根越近,则优先级越低,
- 而推导过程均由文法开始符号进行分析,
- 因此,在规则中最接近文法开始符号的运算符号应该是+ -,而* /则远一点,而()则更远.

$exp \rightarrow exp \text{ addop } exp \mid term$

$addop \rightarrow + \mid -$

$term \rightarrow term \text{ mulop } term \mid factor$

$mulop \rightarrow * \mid /$

$factor \rightarrow (exp) \mid number$

对于该文法,串 $34-3*42$ 就只有唯一的语法树。

• 文法改造过程

- 同级运算符的优先关系：左结合
- 根据语法树中谁深谁优先的原理可得：
- 左方的同级运算符只能在左边产生，而不能在右边产生。

$exp \rightarrow \textcolor{red}{exp} \text{ addop } \textcolor{red}{exp} \mid term$ 左右均可产生

$term \rightarrow \textcolor{red}{term} \text{ mulop } \textcolor{red}{term} \mid factor$ 左右均可产生

结论：删除右边产生的递归即可

即将规则

$$exp \rightarrow exp \text{ addop } exp \mid term$$

改造为

$$exp \rightarrow exp \text{ addop } term \mid term$$

结论:

左递归规则可实现左结合，而右递归规则可实现右结合。

- 改造结果:

$exp \rightarrow exp \text{ addop } term \mid term$

$addop \rightarrow + \mid -$

$term \rightarrow term \text{ mulop } factor \mid factor$

$mulop \rightarrow * \mid /$

$factor \rightarrow (exp) \mid number$

表达式 $34-3*42$ 的分析树为:

表达式 $34-3-42$ 的分析树为:

扩展的表示法：EBNF

原因：

(1) 表达能力弱、符号不丰富

(2) 程序设计语言的控制结构有：顺序、重复和选择结构

- EBNF：扩充BNF，在规则中可表示重复和可选。

重复操作

$A \rightarrow Aa \mid b$ (左递归)

或

$A \rightarrow aA \mid b$ (右递归)

- 在EBNF中用花括号 $\{...\}$ 来表示重复
- 因此上述规则可用以下规则写出:

$A \rightarrow b\{a\}$

和

$A \rightarrow \{a\}b$

可选操作

- EBNF 中用方括号 **[...]** 来表示可选

statement \rightarrow if-stmt | other

if-stmt \rightarrow if exp then statement

 | if exp then statement else statement

exp \rightarrow 0 | 1

可表示:

statement \rightarrow if-stmt | other

if-stmt \rightarrow if exp then statement **[else statement]**

exp \rightarrow 0 | 1

TINY语言的语法

TINY语言的上下文无关文法：

详细见参考书：P97的BNF表示。

TINY 语言的语法规则列表

program \rightarrow stmt-sequence

stmt-sequence \rightarrow stmt-sequence ; statement | statement

statement \rightarrow if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt

if-stmt \rightarrow if exp then stmt-sequence end

| if exp then stmt-sequence else stmt-sequence end

repeat-stmt \rightarrow repeat stmt-sequence until exp

assign-stmt \rightarrow identifier := exp

read-stmt \rightarrow read identifier

write-stmt \rightarrow write exp

exp \rightarrow simple-exp comparison-op simple-exp | simple-exp

comparison-op \rightarrow < | =

simple-exp \rightarrow simple-exp addop term | term

addop \rightarrow + | -

term \rightarrow term mulop factor | factor

mulop \rightarrow * | /

factor \rightarrow (exp) | number | identifier

文法中规则有效性的分析

有害规则：形如 $U \rightarrow U$ 的产生式，会导致文法出现二义性

多余规则：指文法中任何句子的推导都不会用到的规则

文法中不含有不可到达和不可终止的非终结符

1) 文法中某些非终结符不在任何规则的右部出现，

该非终结符称为**不可到达**。

2) 文法中某些非终结符，由它不能推出终结符号串，

该非终结符称为**不可终止**。

化简文法——删除无效规则

目标：文法中不含有有害规则和多余规则

文法中规则有效性的条件

对于文法 $G[S]$ ，为了保证任一非终结符 A 在句子推导中出现，必须满足如下两个条件：

1. A 必须在某句型中出现

即有 $S \Rightarrow^* \alpha A \beta$ ，其中 α, β 属于 V^*

2. 必须能够从 A 推出终结符号串 t

即 $A \Rightarrow^* t$ ，其中 $t \in V_T^*$

化简文法

例, $G[S]$:

1) $S \rightarrow Be$

2) $B \rightarrow Ce$

D 为不可到达

3) $B \rightarrow Af$

C 为不可终止

4) $A \rightarrow Ae$

5) $A \rightarrow e$

6) $C \rightarrow Cf$

7) $D \rightarrow f$

产生式 2), 6), 7) 为多余规则应去掉。

文法中的 ε 规则的问题

文法中某些规则可具有形式 $A \rightarrow \varepsilon$ ，称这种规则为 ε 规则

处理方法：一般可以不做任何处理

例，

$$A \rightarrow dB \quad B \rightarrow aB \mid \varepsilon$$

自顶而下语法分析的问题

一种约定：最左推导

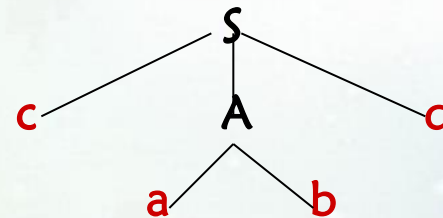
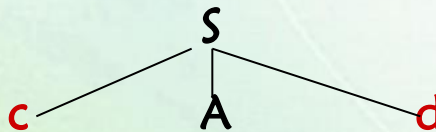
从左到右的分析算法，即总是从左到右地分析输入符号串，首先分析符号串中的最左符号，进而依次分析右边的一个符号，直到分析结束。

自顶而下语法分析的问题

例, 文法 $G[S]$:

$$S \rightarrow cAd$$
$$A \rightarrow ab$$
$$A \rightarrow b$$

分析输入串 $w = cabd$ 是否为该文法的句子



推导过程: $S \Rightarrow cAd$
 $\Rightarrow \underline{cabd}$

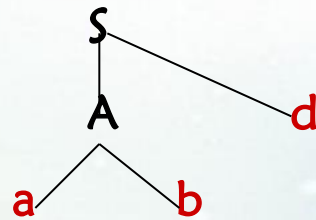
自顶而下语法分析的问题

例，文法 $G[S]$: $S \rightarrow Ad \mid Bc$

$A \rightarrow ab$

$B \rightarrow ba$

分析输入串 $w = abd$ 是否为该文法的句子



推导过程: $S \Rightarrow Ad$

$\Rightarrow \underline{abd}$

自顶而下语法分析的问题

问题1: 在自顶而下的分析方法中**如何选择**使用**哪条规则**进行推导?

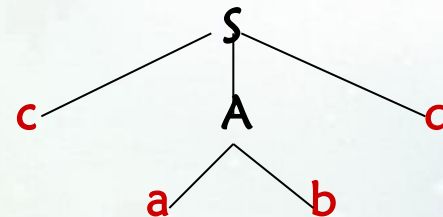
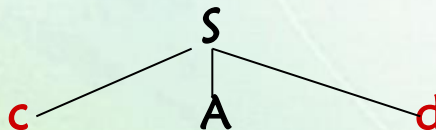
假定要被代换的最左非终结符号是A, 且有n条规则: $A \rightarrow B_1 | B_2 | \dots | B_n$, 那么如何确定用哪个右部去替代A?

自顶而下语法分析的问题

例，文法 $G[S]$:

$$S \rightarrow cAd$$
$$A \rightarrow ab$$
$$A \rightarrow a$$

分析输入串 $w = cabd$ 是否为该文法的句子



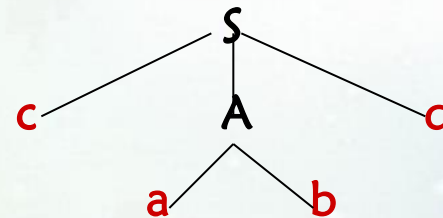
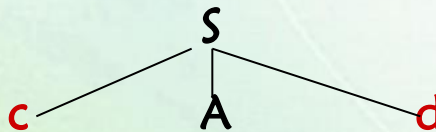
推导过程: $S \Rightarrow cAd$
 $\Rightarrow \underline{cabd}$

自顶而下语法分析的问题

例, 文法 $G[S]$:

$$S \rightarrow cAd$$
$$A \rightarrow ab$$
$$A \rightarrow a$$

分析输入串 $w = cad$ 是否为该文法的句子



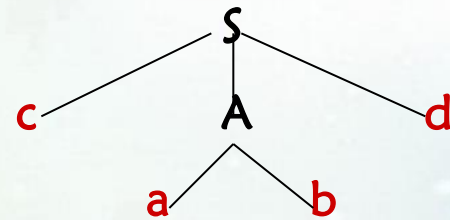
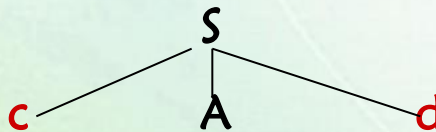
推导过程: $S \Rightarrow cAd$
 $\Rightarrow \underline{cabd}$

自顶而下语法分析的问题

例，文法 $G[S]$:

$$S \rightarrow cAd$$
$$A \rightarrow ab$$
$$A \rightarrow a$$

分析输入串 $w = cab$ 是否为该文法的句子



推导过程: $S \Rightarrow cAd$
 $\Rightarrow cabd$

自顶而下语法分析的问题

例, 文法 $G[S]$: $S \rightarrow cAd$

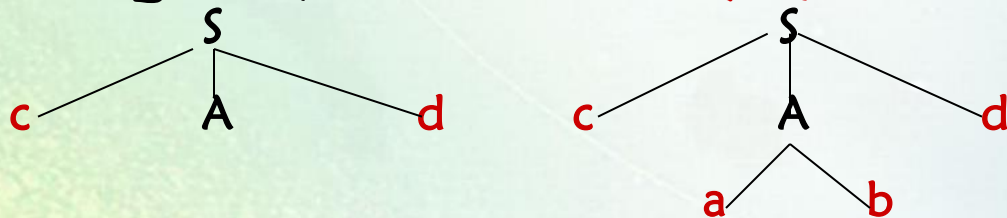
$S \rightarrow cB$

$A \rightarrow ab$

$A \rightarrow a$

$B \rightarrow aa$

分析输入串 $w = caa$ 是否为该文法的句子



推导过程: $S \Rightarrow cAd$
 $\Rightarrow \underline{cabd}$

问题原因的分析1

效率问题 回溯、选择规则效率

问题2：同一非终结符规则右部存在多条规则且最左的符号相同

$G[S]:$

$$S \rightarrow cAd \mid cB$$
$$A \rightarrow ab \mid a$$
$$B \rightarrow aa$$

问题原因的分析2

效率问题 回溯、选择规则效率

问题3：同一非终结符规则右部存在多条规则且经过多步推导后，最左的符号相同

G[S]:
 $S \rightarrow Ad \mid Bc$
 $A \rightarrow ab$
 $B \rightarrow aa$

问题2的解决

效率问题 回溯、选择规则效率

问题2：同一非终结符规则右部存在多条规则且最左的符号相同

$G[S]:$

$$S \rightarrow cAd \mid cB$$
$$A \rightarrow ab \mid a$$
$$B \rightarrow aa$$

问题归纳

左公共因子

当两个或更多文法规则共享一个通用前缀串。

如： $A \rightarrow \alpha\beta \mid \alpha\gamma$

或 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n$

解决方法：

(1) 手工调整

(2) 文法改写，用EBNF改写

——左公共因子的提取

$$A \rightarrow \alpha\beta \mid \alpha\gamma$$

解决方法：将左边的 α 提取出来，

即

$$A \rightarrow \alpha(\beta \mid \gamma)$$

或：

该规则分解为两个规则：

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta \mid \gamma$$

规则： $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n$

提取左公因子后变为：

$$A \rightarrow \alpha(\beta_1 | \beta_2 | \dots | \beta_n)$$

或：

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

TINY 语言的语法规则列表

program \rightarrow stmt-sequence

stmt-sequence \rightarrow stmt-sequence ; statement | statement

statement \rightarrow if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt

if-stmt \rightarrow if exp then stmt-sequence end

| if exp then stmt-sequence else stmt-sequence end

repeat-stmt \rightarrow repeat stmt-sequence until exp

assign-stmt \rightarrow identifier := exp

read-stmt \rightarrow read identifier

write-stmt \rightarrow write exp

exp \rightarrow simple-exp comparison-op simple-exp | simple-exp

comparison-op \rightarrow < | =

simple-exp \rightarrow simple-exp addop term | term

addop \rightarrow + | -

term \rightarrow term mulop factor | factor

mulop \rightarrow * | /

factor \rightarrow (exp) | number | identifier

EBNF中TINY语言的文法

program \rightarrow *stmt-sequence*

stmt-sequence \rightarrow *statement* { **;** *statement* }

statement \rightarrow *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*

if-stmt \rightarrow **if** *exp* **then** *stmt-sequence* [**else** *stmt-sequence*] **end**

repeat-stmt \rightarrow **repeat** *stmt-sequence* **until** *exp*

assign-stmt \rightarrow **identifier** **:=** *exp*

read-stmt \rightarrow **read** **identifier**

write-stmt \rightarrow **write** *exp*

exp \rightarrow *simple-exp* [*comparison-op* *simple-exp*]

comparison-op \rightarrow **<** | **=**

simple-exp \rightarrow *term* { *addop* *term* }

addop \rightarrow **+** | **-**

term \rightarrow *factor* { *mulop* *factor* }

mulop \rightarrow ***** | **/**

factor \rightarrow (*exp*) | **number** | **identifier**

问题1的解决

问题1: 在自顶而下的分析方法中**如何选择**使用哪**条规则**进行推导?

假定要被代换的最左非终结符号是A, 且有n条规则: $A \rightarrow B_1 | B_2 | \dots | B_n$, 那么如何确定用哪个右部去替代A?

例, 文法G[S]:
 $S \rightarrow Ad | Bc$
 $A \rightarrow ab$
 $B \rightarrow ba$

问题3的解决

效率问题 回溯、选择规则效率

同一非终结符规则右部存在多条规则且经过多步推导后，最左的符号相同

$$\begin{aligned} G[S]: \quad & S \rightarrow Ad \mid Bc \\ & A \rightarrow ab \\ & B \rightarrow aa \end{aligned}$$

消除回溯性:

使得对于任何 $U \in V_N$, $U ::= x_1 \mid x_2 \mid \cdots \mid x_n$, 如果

$$x_i \xRightarrow{*} T_i v \quad \text{和} \quad x_j \xRightarrow{*} T_j w,$$

且 $T_i, T_j \in V_T$, 则 $T_i \neq T_j, i \neq j, (1 \leq i, j \leq n)$ 。

结论

为了

- (1) 提高分析效率,避免回溯
- (2) 解决诸如 $S \rightarrow AB|CD$ 的规则选择问题;
- (3) 解决是否存在公共因子的问题

有必要求出**每一条规则**的**开头非终结符号**的所有**打头终结符号**

• ———**First集合**

怎样求First集合?

归纳!

first集合计算方法

考虑情况1:

$G[S]=\{$

$S \rightarrow AB$

$A \rightarrow Ba$

$B \rightarrow Cb$

$C \rightarrow ef$

$\}$

(1) 求出 $\text{first}(C)=?$

(2) 求出 $\text{first}(A)=?$

first集合计算方法

考虑情况2:

$G[S]=\{$

$S \rightarrow AB \mid CD$

$A \rightarrow aB \mid dD$

$B \rightarrow cC \mid bD$

$C \rightarrow ef \mid gh$

$D \rightarrow i \mid j$

$\}$

求出 $\text{first}(A)=?$

求出 $\text{first}(S)=?$

first集合计算方法

考虑情况3:

$G[S]=\{$

$S \rightarrow ABC \mid D$

$A \rightarrow aB \mid \varepsilon$

$B \rightarrow cC \mid \varepsilon$

$C \rightarrow eC \mid \varepsilon$

$D \rightarrow i \mid j$

$\}$

求出 $\text{first}(D)=?$

求出 $\text{first}(A)=?$

求出 $\text{first}(S)=?$

First集合计算方法之归纳1

$$\text{First}(x) = \begin{cases} x & x \text{ 为终结符号} \\ x & x = \varepsilon \\ \text{First}(x_1) - \{\varepsilon\} & X \rightarrow X_1 X_2 \dots X_n \\ \cup \text{First}(x_2) - \{\varepsilon\} & X \rightarrow X_2 \dots X_n \quad \varepsilon \in \text{first}(x_1) \\ \cup \dots\dots\dots \\ \cup \text{First}(x_n) - \{\varepsilon\} & X \rightarrow X_n \quad \varepsilon \in \text{first}(x_i) \quad 1 \leq i < n \\ \cup \varepsilon & \varepsilon \in \text{first}(x_i) \quad 1 \leq i \leq n \end{cases}$$

对于规则 $X \rightarrow x_1x_2...x_n$, $first(x)$ 的计算算法如下:

$First(x) = \{ \};$

$K=1;$

While ($k \leq n$)

{ if (x_k 为终结符号或 ε) $first(x_k) = x_k;$

$first(x) = first(x) \cup first(x_k) - \{\varepsilon\}$

If ($\varepsilon \notin first(x_k)$) break;

$k++;$

}

If ($k == n+1$) $first(x) = first(x) \cup \varepsilon$

First集合计算方法之归纳2

$$\text{First}(x) = \begin{cases} x & x \text{ 为终结符号} \\ x & x = \varepsilon \\ \text{First}(Y) & X \rightarrow YA \\ \cup \text{First}(A) & A \rightarrow YA \mid \varepsilon \quad \varepsilon \in \text{first}(Y) \end{cases}$$

递归表达

请自行完成算法!

EBNF中TINY语言的文法

program \rightarrow *stmt-sequence*

stmt-sequence \rightarrow *statement* { **;** *statement* }

statement \rightarrow *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*

if-stmt \rightarrow **if** *exp* **then** *stmt-sequence* [**else** *stmt-sequence*] **end**

repeat-stmt \rightarrow **repeat** *stmt-sequence* **until** *exp*

assign-stmt \rightarrow **identifier** **:=** *exp*

read-stmt \rightarrow **read** **identifier**

write-stmt \rightarrow **write** *exp*

exp \rightarrow *simple-exp* [*comparison-op* *simple-exp*]

comparison-op \rightarrow **<** | **=**

simple-exp \rightarrow *term* { *addop* *term* }

addop \rightarrow **+** | **-**

term \rightarrow *factor* { *mulop* *factor* }

mulop \rightarrow ***** | **/**

factor \rightarrow (*exp*) | **number** | **identifier**

自顶向下分析法的问题

文法 $G[S]$:

$$S \rightarrow aA \mid d$$

$$A \rightarrow bAS \mid \varepsilon$$

输入串 $W=abd$ 。

自顶向下的推导过程为:

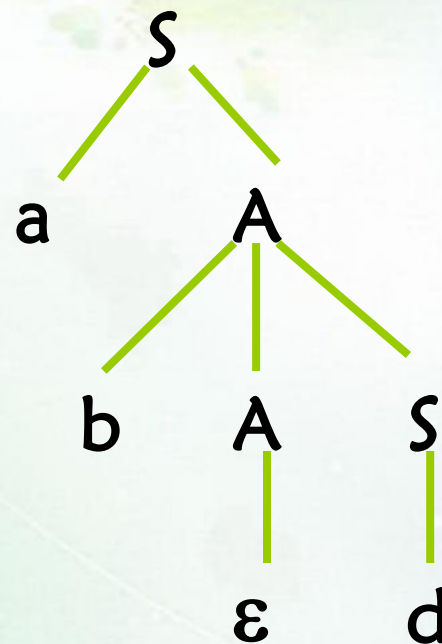
$$S \Rightarrow aA$$

$$\Rightarrow abAS$$

$$\Rightarrow abS$$

$$\Rightarrow abd$$

相应的分析树为:



问题: 当文法中存在规则, 如: $A \rightarrow \varepsilon$ 时, 什么时候选择它的问题。

问题分析

- 为了提高分析效率，我们有必要知道紧跟着A后面出现的终结符号是谁？以便决定是否选择规则 $A \rightarrow \varepsilon$ 。
- ——follow集合。



怎样求Follow集合？

Follow 集合的计算方法分析

- 考虑以下的几种情况:
- 1. 若 A 是开始符号, 那么 $\text{Follow}(A) = ?$ {
- 2. 若存在规则 $B \rightarrow \alpha A \gamma$, 则 $\text{Follow}(A) = ?$ }
- 3. 若存在规则 $B \rightarrow \alpha A \gamma$, 且 ϵ 在 $\text{First}(\gamma)$ 中, 则 $\text{Follow}(A) = ?$

Follow 集合的定义

- **定义**：给出一个非终结符 A ，那么集合 $\text{Follow}(A)$ 则是由终结符或**结束符号** $\$$ 组成。

集合 $\text{Follow}(A)$ 的定义如下：

- 1. 若 A 是开始符号，则 $\$$ 就在 $\text{Follow}(A)$ 中。
- 2. 若存在规则 $B \rightarrow \alpha A \gamma$ ，则 $\text{First}(\gamma) - \{\epsilon\}$ 在 $\text{Follow}(A)$ 中。
- 3. 若存在规则 $B \rightarrow \alpha A \gamma$ ，且 ϵ 在 $\text{First}(\gamma)$ 中，则 $\text{Follow}(A)$ 包括 $\text{Follow}(B)$ 。

Follow 计算的算法设计

$G[S]=\{$

$S \rightarrow ABC$

$A \rightarrow aB \mid bB$

$B \rightarrow cC \mid \varepsilon$

$C \rightarrow ef \mid gh$

$\}$

$A \rightarrow X_1 X_2 \dots X_i X_{i+1} \dots X_n$

计算Follow集合的算法

1.初始化:

1.1 $\text{Follow}(\text{开始符号}) = \{ \$ \}$

1.2 其他任何一个非终结符号A, 则执行 $\text{Follow}(A) = \{ \}$

2.循环: 反复执行

2.1 循环: 对于文法中的每条规则 $A \rightarrow X_1 X_2 \dots X_n$ 都执行

2.1.1 对于该规则中的每个属于非终结符号的 X_i , 都执行

2.1.1.1 把 $\text{First}(X_{i+1} X_{i+2} \dots X_n) - \{ \epsilon \}$ 添加到 $\text{Follow}(X_i)$

2.1.1.2 if ϵ in $\text{First}(X_{i+1} X_{i+2} \dots X_n)$, 则把 $\text{Follow}(A)$ 添加到 $\text{Follow}(X_i)$

直到任何一个Follow集合的值都没有发生变化为止。

$$A \rightarrow X_1 X_2 \dots X_i X_{i+1} \dots X_n$$

自顶而下语法分析的问题

— 考虑下列文法分析 ba^n 的分析过程

文法 $G[S]$:

(1) $S \rightarrow Sa$

(2) $S \rightarrow b$

$S \Rightarrow Sa$ (1)

$\Rightarrow Saa$ (1)

$\Rightarrow Saaa$ (1)

.....

$\Rightarrow Sa^n$ (2)

$\Rightarrow ba^n$

问题：左递归

直接左递归的消除方法1

$A \rightarrow Aa \mid b$ (左递归)

- 在EBNF中用花括号 $\{...\}$ 来表示重复
- 因此上述规则可用以下规则写出:

$A \rightarrow b\{a\}$

直接左递归的消除方法2

$A \rightarrow Aa \mid b$ (左递归)

- 改写为右递归

$A \rightarrow bA'$

$A' \rightarrow aA' \mid \varepsilon$

间接左递归

研究下面文法：

$G[A]:$

$A \rightarrow Aa \mid Bb \mid Cc$

$B \rightarrow Ab \mid Bb$

$C \rightarrow Ac \mid Cb$

解决思路

- (1) 逐个逐个非终结符进行解决；
- (2) 将干净非终结符代入未解决的非终结符中，并将其消除干净；
- (3) 反复实施。

实例分析

例 消除下面文法的左递归

$$A \rightarrow Ba \mid Aa \mid c$$

$$B \rightarrow Bb \mid Ab \mid d$$

解：

(1) 逐个逐个进行解决，先对A进行处理，这样就得到文法：

$$A \rightarrow BaA' \mid cA'$$

$$A' \rightarrow aA' \mid \varepsilon$$

$$B \rightarrow Bb \mid Ab \mid d$$

第二步：代入再消除

即将非终结符号B中出现A的，均被代入，
因此就得到文法：

$$A \rightarrow BaA' \mid cA'$$

$$A' \rightarrow aA' \mid \varepsilon$$

$$B \rightarrow Bb \mid BaA'b \mid cA'b \mid d$$

第三步：消除 B 的直接左递归以得到：

$$A \rightarrow BaA' \mid cA'$$

$$A' \rightarrow aA' \mid \varepsilon$$

$$B \rightarrow cA'bB' \mid dB'$$

$$B' \rightarrow bB' \mid aA'bB' \mid \varepsilon$$

这个文法没有左递归。

消除算法

(1) 将文法 G 的所有非终结符号按任一种顺序排列为

A_1, \dots, A_m ;

(2) 执行循环语句:

for($i=1; i \leq m; i++$)

{

for ($j=1; j \leq i-1; j++$)

将规则 $A_i \rightarrow A_j \alpha$ 改写;

// 改写方法如下: 如果 $A_j \rightarrow \beta_1 | \beta_2 | \dots | \beta_k$

// $A_i \rightarrow \beta_1 \alpha | \beta_2 \alpha | \dots | \beta_k \alpha$

消除 A_i 规则中的直接左递归;

}

(3) 化简由 (2) 所得的文法, 即消去多余规则

例.G[Z]:

$$Z \rightarrow Za \mid Sbc \mid dS$$

$$S \rightarrow Zef \mid gSh$$

试消除左递归:

G'[Z]:

$$Z \rightarrow (Sbc \mid dS)Z'$$

$$S \rightarrow (dSZ'ef \mid gSh)S'$$

$$Z' \rightarrow aZ' \mid \varepsilon$$

$$S' \rightarrow bcZ'efS' \mid \varepsilon$$

例. 文法 $G[Z]$:

$$Z \rightarrow Sa \mid Tb \mid cZ$$

$$S \rightarrow Tde \mid Zf \mid Sg$$

$$T \rightarrow Sh \mid jTk$$

试消除左递归

$G'[Z]$:

$$Z \rightarrow Sa \mid Tb \mid cZ$$

$$S \rightarrow (T(bf \mid de) \mid cZf) S' \quad S' \rightarrow (af \mid g) S' \mid \varepsilon$$

$$T \rightarrow (cZfS'h \mid jTk) T' \quad T' \rightarrow (bf \mid de) S'hT' \mid \varepsilon$$

问题归纳

1. 左递归

(1) 简单的左递归

$$A \rightarrow A\alpha \mid \beta$$

(2) 复杂一点的左递归

$$A \rightarrow A\alpha \mid A\gamma \mid \beta$$

(3) 更复杂的左递归——间接左递归

$$A \rightarrow B\alpha \mid \dots$$

$$B \rightarrow A\beta \mid \dots$$

• 情况1: 简单直接左递归的消除

格式:

$$A \rightarrow A\alpha \mid \beta$$

α 和 β 是终结符和非终结符组成的串,且 β 不以 A 开头

消除方法1: 采用EBNF改写文法规则 $A \rightarrow \beta \{ \alpha \}$

消除方法2: 采用右递归来取代

$$A \rightarrow \beta A'$$

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

- 情况2: 普遍的直接左递归

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \beta_1 | \beta_2 | \dots | \beta_m$$

其中 β_1, \dots, β_m 均不以 A 开头。

可先改为:

$$A \rightarrow A(\alpha_1 | \alpha_2 | \dots | \alpha_n) | (\beta_1 | \beta_2 | \dots | \beta_m)$$

接着使用情况1的方法进行改造, 如改为:

$$A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_m A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A' | \varepsilon$$

• 情况3: 更复杂的左递归——间接左递归

$$A \rightarrow Bb \mid \dots$$

$$B \rightarrow Aa \mid \dots$$

- (1) 逐个逐个非终结符进行解决;
- (2) 将干净非终结符代入未解决的非终结符中, 并将其消除干净;
- (3) 反复实施.

TINY 语言的语法规则列表

program \rightarrow stmt-sequence

stmt-sequence \rightarrow stmt-sequence ; statement | statement

statement \rightarrow if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt

if-stmt \rightarrow if exp then stmt-sequence end

| if exp then stmt-sequence else stmt-sequence end

repeat-stmt \rightarrow repeat stmt-sequence until exp

assign-stmt \rightarrow identifier := exp

read-stmt \rightarrow read identifier

write-stmt \rightarrow write exp

exp \rightarrow simple-exp comparison-op simple-exp | simple-exp

comparison-op \rightarrow < | =

simple-exp \rightarrow simple-exp addop term | term

addop \rightarrow + | -

term \rightarrow term mulop factor | factor

mulop \rightarrow * | /

factor \rightarrow (exp) | number | identifier

EBNF中TINY语言的文法

program \rightarrow *stmt-sequence*

stmt-sequence \rightarrow *statement* { **;** *statement* }

statement \rightarrow *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*

if-stmt \rightarrow **if** *exp* **then** *stmt-sequence* [**else** *stmt-sequence*] **end**

repeat-stmt \rightarrow **repeat** *stmt-sequence* **until** *exp*

assign-stmt \rightarrow **identifier** **:=** *exp*

read-stmt \rightarrow **read** **identifier**

write-stmt \rightarrow **write** *exp*

exp \rightarrow *simple-exp* [*comparison-op* *simple-exp*]

comparison-op \rightarrow **<** | **=**

simple-exp \rightarrow *term* { *addop* *term* }

addop \rightarrow **+** | **-**

term \rightarrow *factor* { *mulop* *factor* }

mulop \rightarrow ***** | **/**

factor \rightarrow (*exp*) | **number** | **identifier**
