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={ I am teacher , You are students }

缺点：语言一般是无限的，而非有限的。

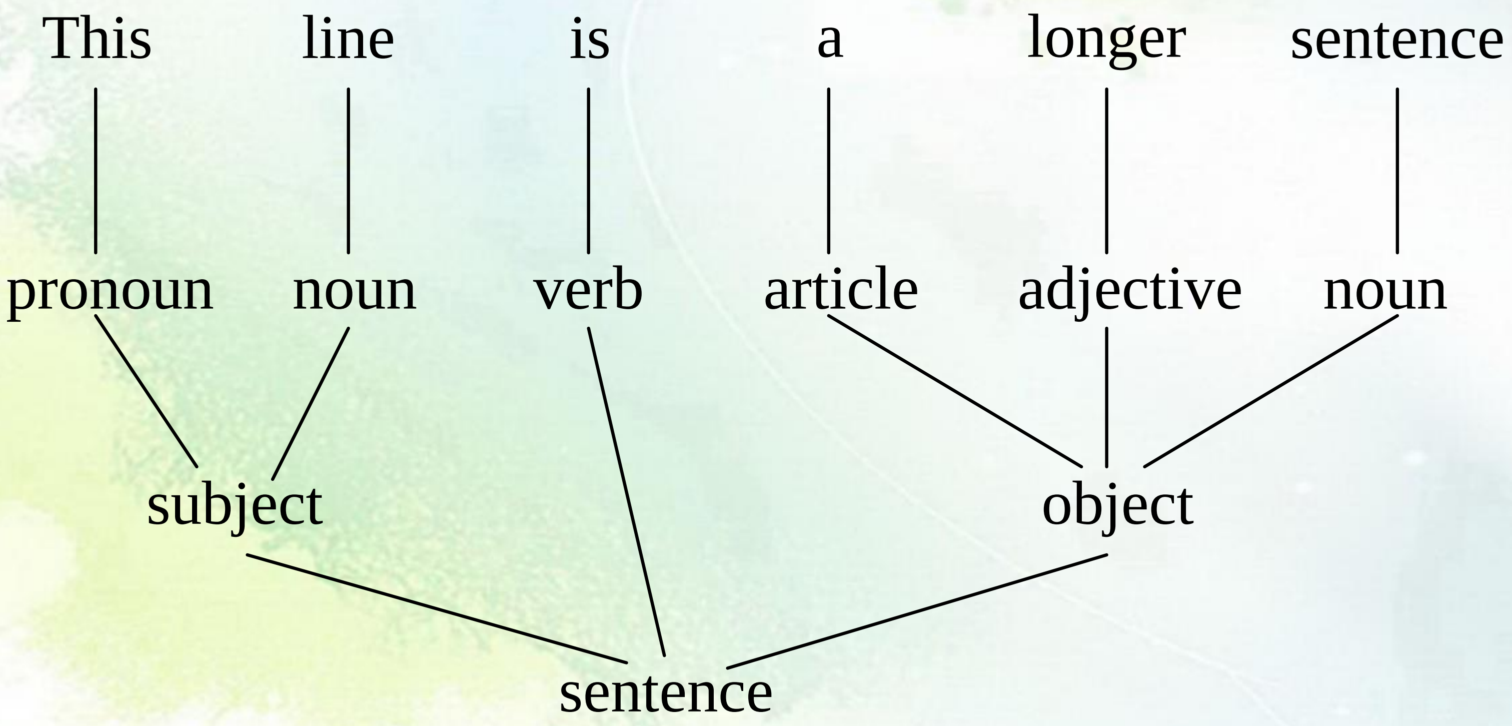
如何来描述一种语言？

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



如何表示规则(产生式)?

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

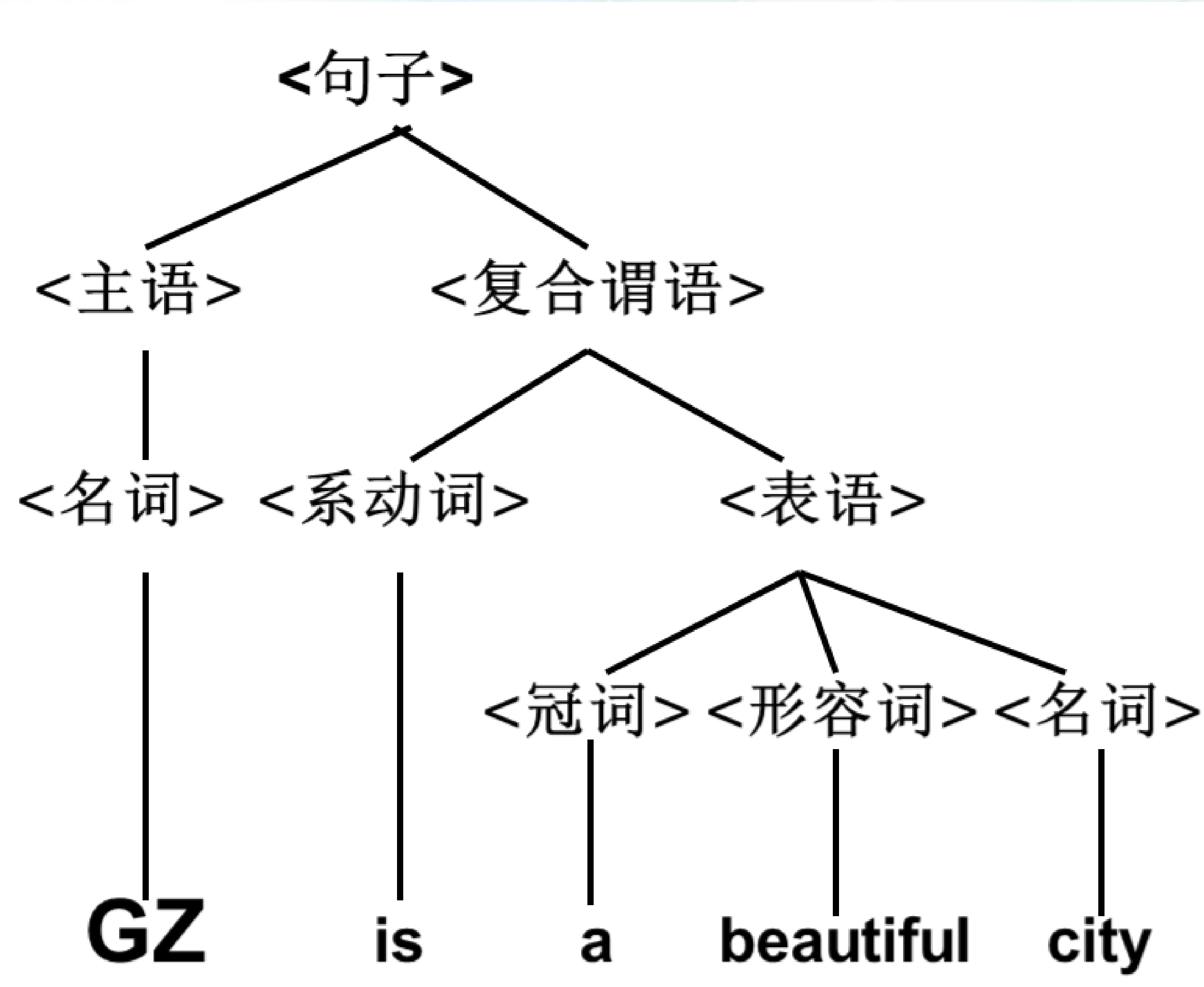
<表语>::=<冠词><形容词><名词>

<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


<句子>
<主语>
<复合谓语>
<名词>
<系动词>
<表语>
<冠词>
<形容词>
<名词>
GZ
is
a
beautiful
city

上下文无关语言的描述

上下文无关文法(*context-free grammars*)

例子

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

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：<head>→<body>

开始符号

$E \rightarrow EOE$

$E \rightarrow (E)$

$E \rightarrow v$

$E \rightarrow d$

$O \rightarrow +$

$O \rightarrow *$

产生式(或规则)集合

非终结符

终结符

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 → V_N

终结符的集合 → V_T

产生式的集合 → P

开始符号 → S

满足

$V_N \cap V_T = \Phi$

$S \in V_N$

产生式形如 $A \to \alpha$, 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=VN∪VT)上的符号串。

文法与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 $G_{exp} = (\{E,O\}, \{ (,), +, *, \nu, d \}, P, E)$.

其中产式集合P为

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

上下文无关语言的描述

产生式(规则)集合的缩写记法:

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

$$
\begin{aligned}
&E \to EOE \\
&E \to (E) \\
&E \to v \\
&E \to d \\
&O \to + \\
&O \to *
\end{aligned}
$$

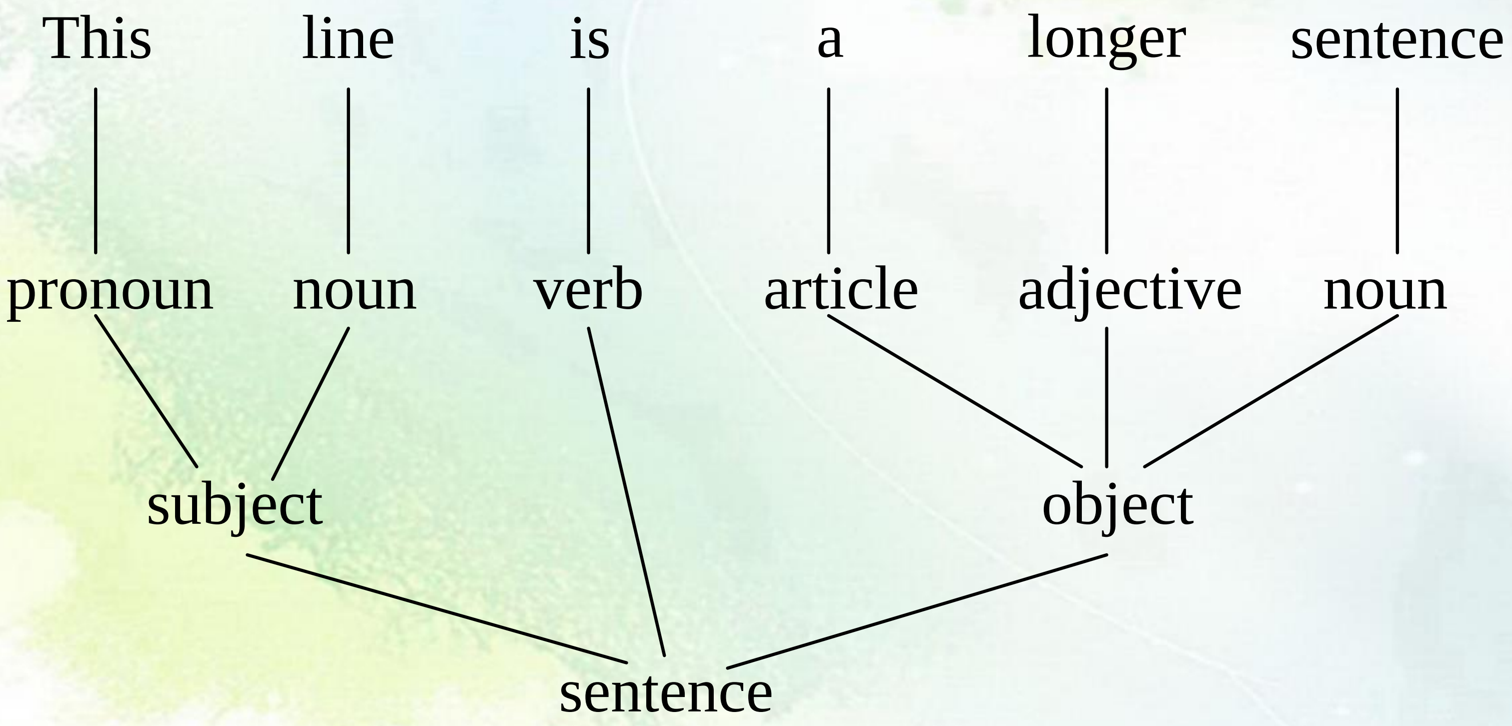
G[E]:

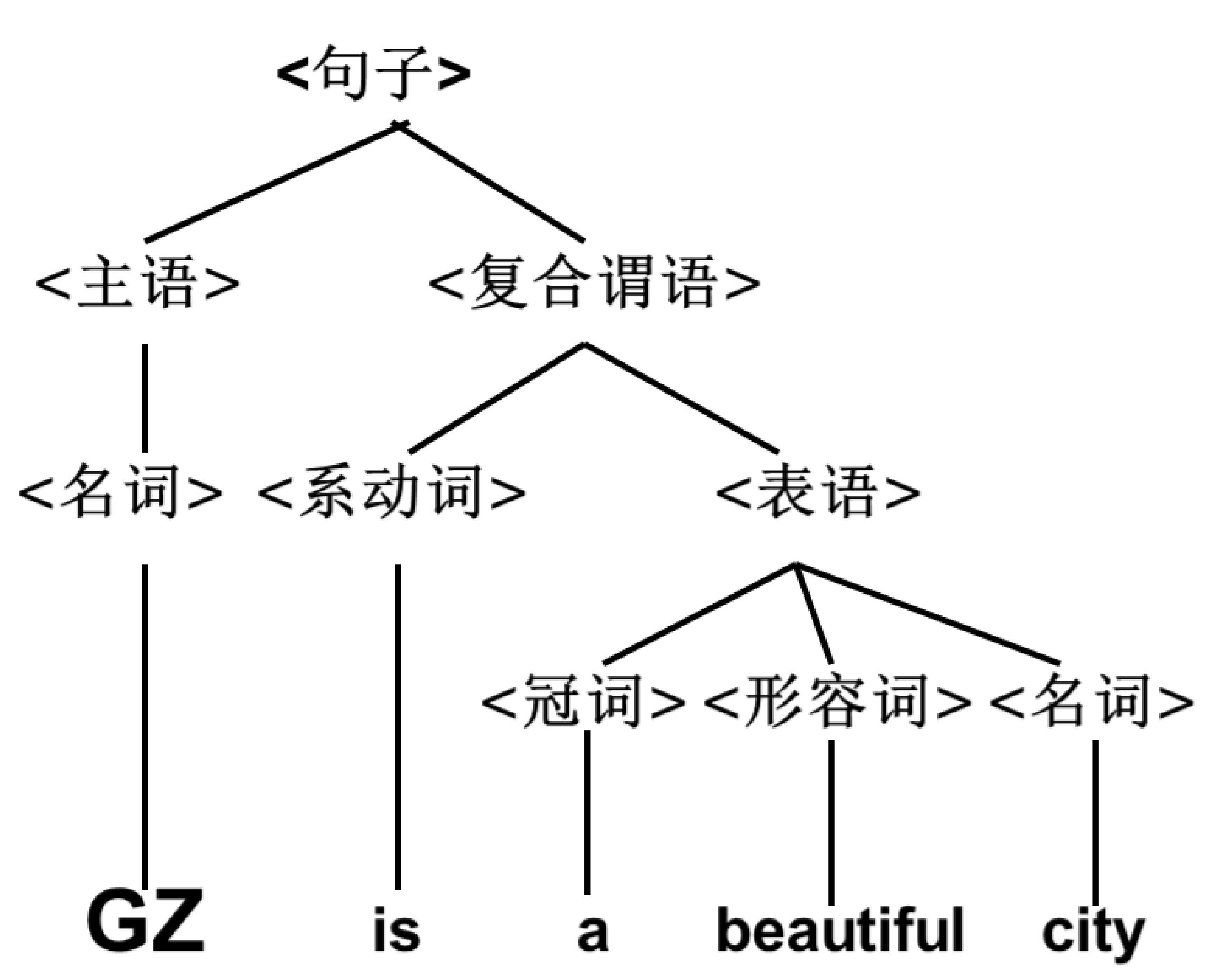
$$
\begin{aligned}
&E \to EOE | (E) | v | d \\
&O \to + | *
\end{aligned}
$$

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




<句子>
<主语>
<复合谓语>
<名词>
<系动词>
<表语>
<冠词>
<形容词>
<名词>
GZ
is
a
beautiful
city

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

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

推导过程举例

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

$$E \xrightarrow{(1)} EOE \xrightarrow{(6)} E*E \xrightarrow{(2)} E*(E) \xrightarrow{(3)} v*(E)$$

$$\xrightarrow{(1)} v*(EOE) \xrightarrow{(5)} v*(E+E) \xrightarrow{(3)} v*(v+E) \xrightarrow{(4)} v*(v+d)$$

推导关系

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

$$\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta .$$

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 $\overset{*}{\underset{G}{\Rightarrow}}$ ，可归纳定义如下：

基础　对任何 $\alpha \in (V_N \cup V_T)^*$，满足 $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$.

归纳　设 $\alpha,\beta,\gamma \in (V_N \cup V_T)^*$，若 $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ ，$\beta \underset{G}{\Rightarrow} \gamma$ 成立，则

$$\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma .$$

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 $\underset{lm}{\Rightarrow}$ 表示，

自反传递闭包用 $\overset{*}{\underset{lm}{\Rightarrow}}$ 表示.

如对于文法G_{exp}，下面是关于 $v*(v+d)$ 的一个最左推导：

$$E \overset{*}{\underset{lm}{\Rightarrow}} EOE \overset{*}{\underset{lm}{\Rightarrow}} vOE \overset{*}{\underset{lm}{\Rightarrow}} v*E$$

$$\overset{*}{\underset{lm}{\Rightarrow}} v*(E) \overset{*}{\underset{lm}{\Rightarrow}} v*(EOE) \overset{*}{\underset{lm}{\Rightarrow}} v*(vOE)$$

$$\overset{*}{\underset{lm}{\Rightarrow}} v*(v+E) \overset{*}{\underset{lm}{\Rightarrow}} v*(v+d)$$

$E \to EOE$

$E \to (E)$

$E \to v$

$E \to d$

$O \to +$

$O \to *$

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 $\underset{rm}{\Rightarrow}$ 表示，

其自反传递闭包用 $\overset{*}{\underset{rm}{\Rightarrow}}$ 表示.

如对于文法G_{exp}，下面是关于 $v*(v+d)$ 的一个最右推导：

$$E \overset{*}{\underset{rm}{\Rightarrow}} EOE \overset{*}{\underset{rm}{\Rightarrow}} EO(E) \overset{*}{\underset{rm}{\Rightarrow}} EO(EOE)$$

$$\overset{*}{\underset{rm}{\Rightarrow}} EO(EOd) \overset{*}{\underset{rm}{\Rightarrow}} EO(E+d) \overset{*}{\underset{rm}{\Rightarrow}} EO(v+d)$$

$$\overset{*}{\underset{rm}{\Rightarrow}} E*(v+d) \overset{*}{\underset{rm}{\Rightarrow}} v*(v+d)$$

$E \to EOE$

$E \to (E)$

$E \to v$

$E \to d$

$O \to +$

$O \to *$

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 $S \overset{*}{\Rightarrow} \alpha$.

若 $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$，则 α 是一个左句型;

若 $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha$，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

$$L(G) = \{ w \mid w \in V_T^* \wedge S \overset{*}{\underset{G}{\Rightarrow}} w \}$$

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→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，… }，因此L(G1)只是VT^+的一个真子集。

例3 设有文法 G2=(VN，VT，P，A)

其中， VN={A} VT={a}

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

试问：此文法描述的语言L(G2)=?

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(G2)={a，aa，aaa，aaaa，…}={a^n，n>=1}。

因此文法G1所定义的语言L(G2)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；
  把程序看做程序设计语言的句子。

- 程序是(程序设计）语言的句子

- 如何系统地构造程序？或者，一般地，
  如何为一个(程序设计)语言生成句子？

几个基本概念

- **直接递归**：设G是一文法，$U \to u$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$U \to u$ 是**直接递归**的；

- **递归**：若存在推导

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

则称规则$U \to u$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$U \to u$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→E+T

E→T

T→T*F

T→F

F→(E)

F→i

＜句子＞::=＜主语＞＜复合谓语＞

＜主语＞::=＜名词＞

＜复合谓语＞::=＜系动词＞＜表语＞

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-->E+T:在计算机内的存储表示：

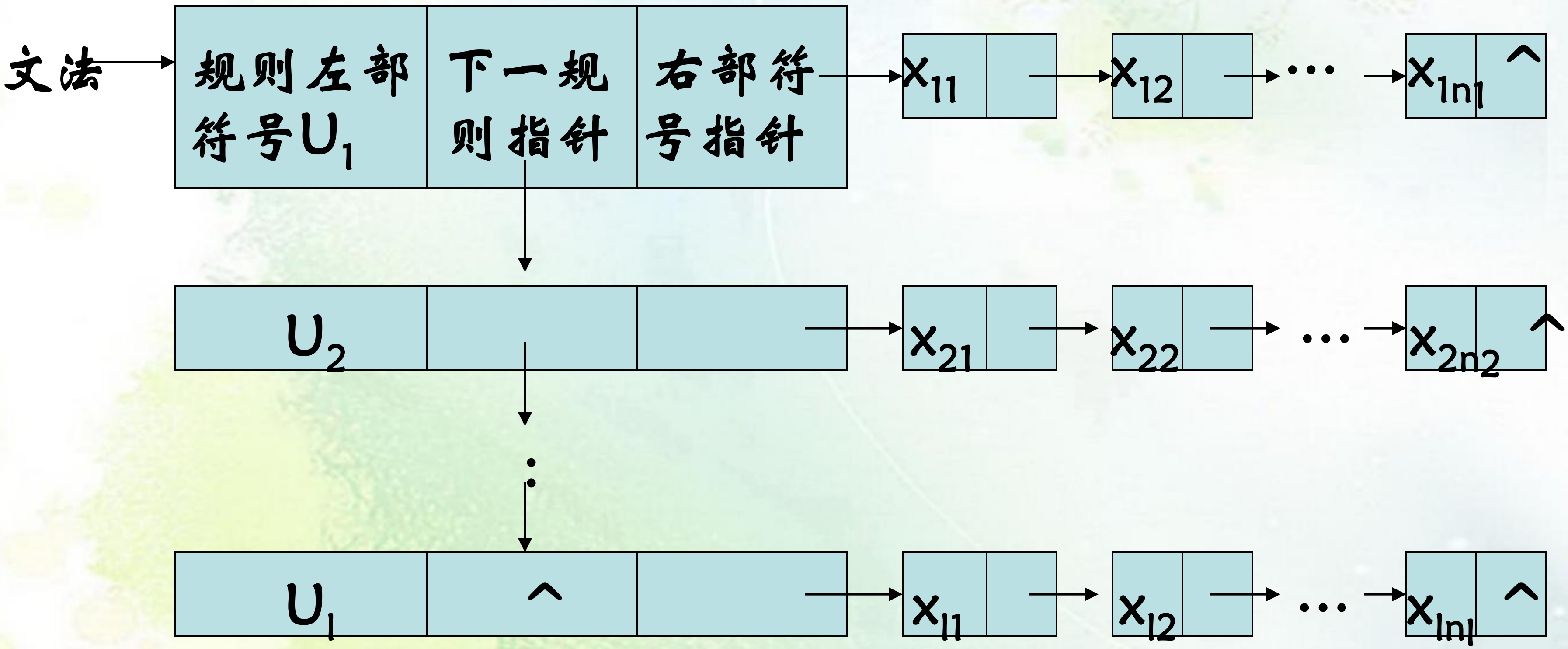
文法[1]：{101，{101，1，102}，3}

规则E-->T在计算机内的存储表示：

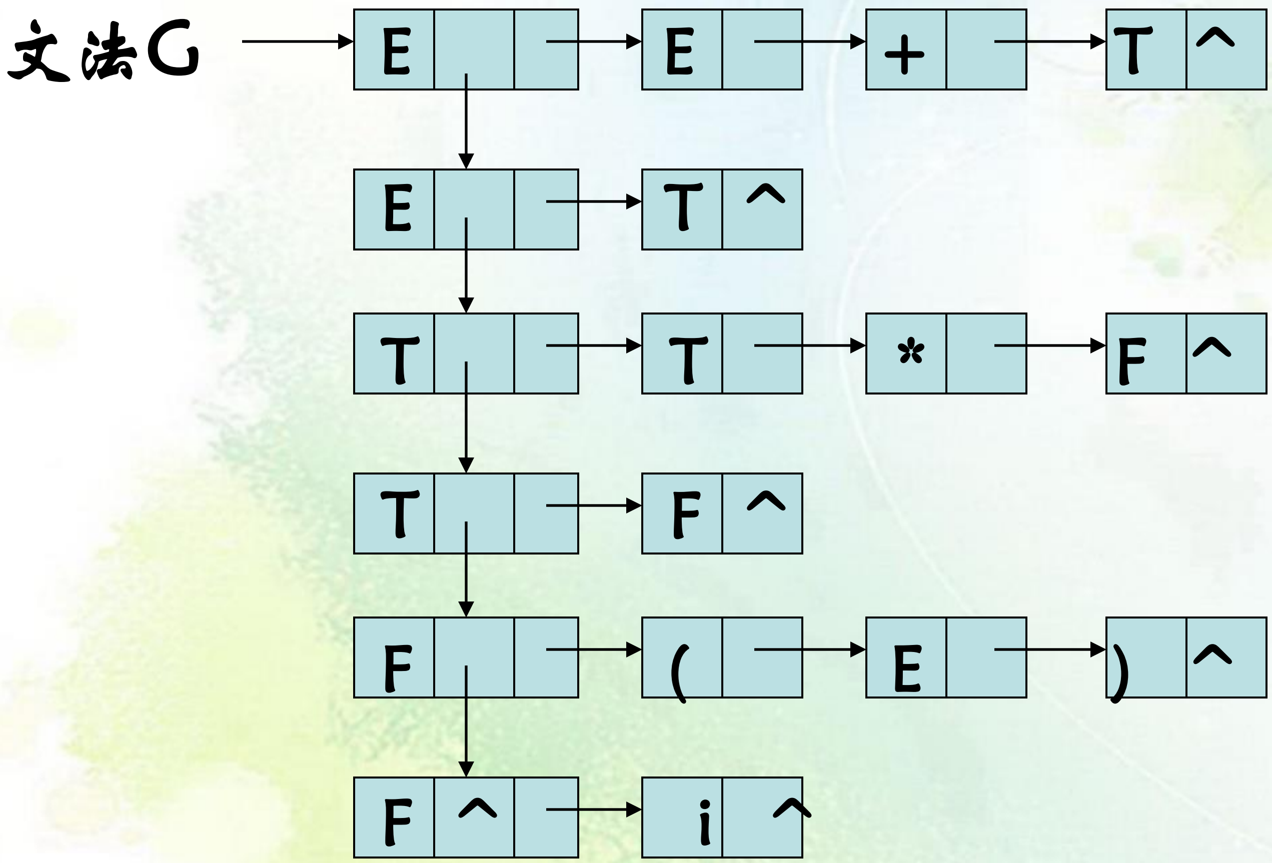
文法[2]：{101，{102}，1}

$T \in V_T$：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
规则左部符号U_1
下一规则指针
右部符号指针
x_{11}
x_{12}
x_{1n_1}
U_2
x_{21}
x_{22}
x_{2n_2}
U_l
x_{l1}
x_{l2}
x_{ln_l}


文法G
E E + T ^
E T ^
T T * F ^
T F ^
F (E) ^
F ^ i ^

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。

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

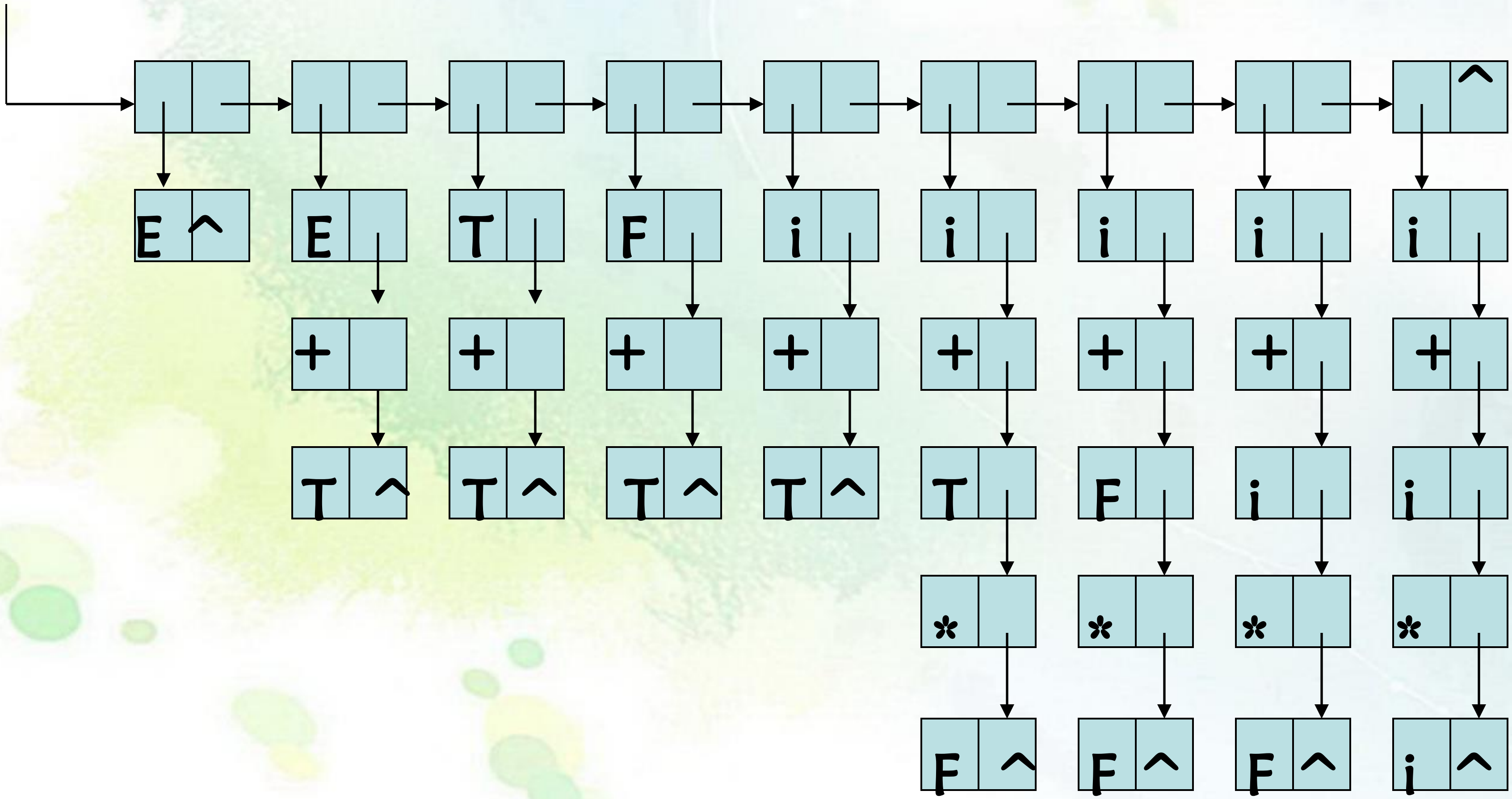
$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow i+T \Rightarrow i+T*F$$
$$\Rightarrow i+F*F \Rightarrow i+i*F \Rightarrow i+i*i$$

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

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

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

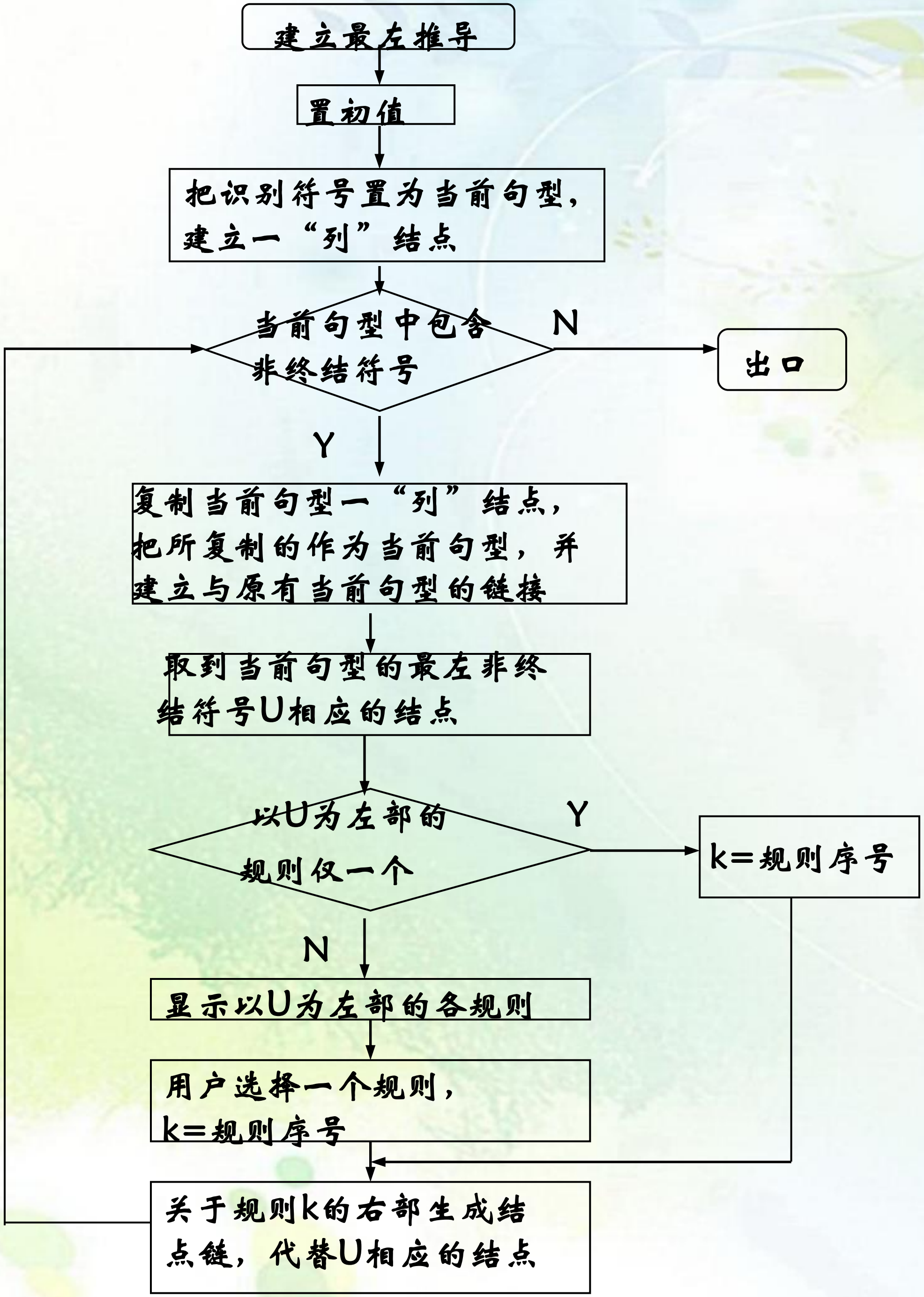
$E \Rightarrow E+T \Rightarrow T+T \Rightarrow T+T*F \Rightarrow F+T*F \Rightarrow F+F*F$
$\Rightarrow i+F*F \Rightarrow i+F*i \Rightarrow i+i*i$

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。


建立最左推导
置初值
把识别符号置为当前句型，
建立一“列”结点
当前句型中包含
非终结符号
N
出口
Y
复制当前句型一“列”结点，
把所复制的作为当前句型，并
建立与原有当前句型的链接
取到当前句型的最左非终
结符号U相应的结点
以U为左部的
规则仅一个
Y
k=规则序号
N
显示以U为左部的各规则
用户选择一个规则，
k=规则序号
关于规则k的右部生成结
点链，代替U相应的结点

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

$$\alpha \to \beta,$$

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

$\alpha \to \beta$，且 $|\alpha| \le |\beta|$，

仅 $S \to \varepsilon$ 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]：

$S \to CD$

$C \to aCA$

$C \to bCB$

$C \to a$

$D \to b$

$bAa \to bDd$

$aAb \to bCd$

$bBa \to aCd$

$aBb \to bDd$

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

$$A \to \beta,$$

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]：　$S \rightarrow AB$

$A \rightarrow BS|0$

$B \rightarrow SA|1$

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]：

$S \to A0|B1|0$

$A \to A0|B1|S0$

$B \to B1|1|0$

G[A]：

$A \to aT$

$A \to a$

$T \to aT$

$T \to dT$

$T \to a$

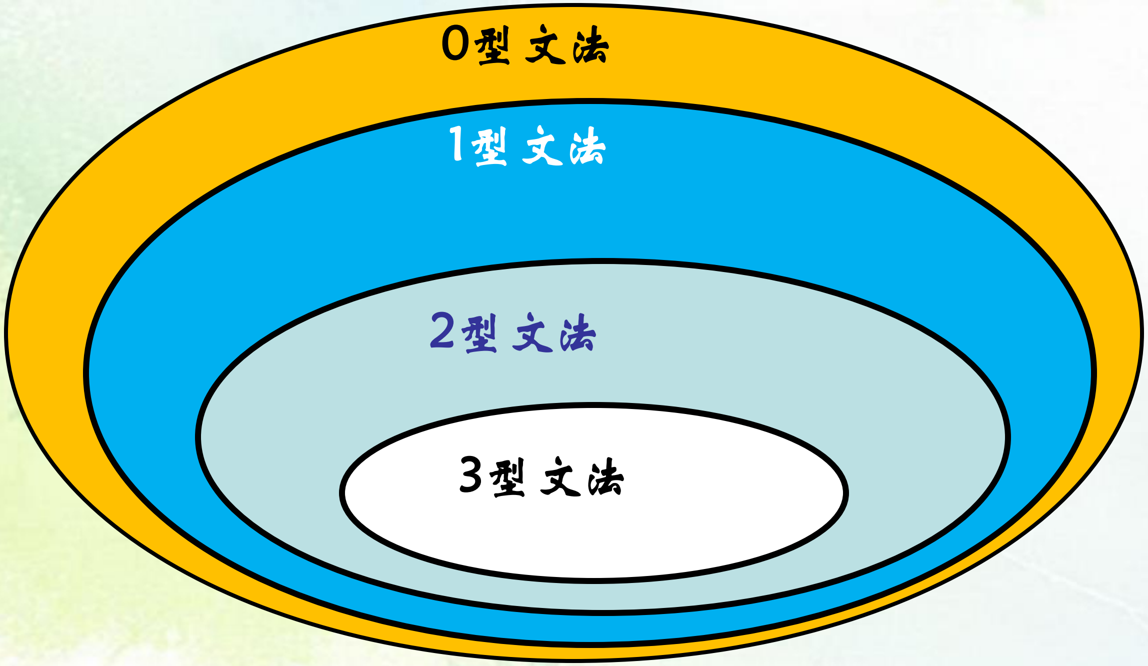
$T \to d$

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**	$A \to aB$ 或 $A \to a$ 其中:$A, B \in N$, $a \in \Sigma$ 也可具有形式: $A \to Ba$ 或 $A \to a$ 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**	$A \to \alpha$ 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**	$\alpha \to \beta$ 其中:$\alpha, \beta \in (N \cup \Sigma)^*$, 且$\lvert\alpha\rvert \leqslant \lvert\beta\rvert$
0	递归可枚举 语言	0 型文法 **Unrestricted**	图灵机 **Turing Machine**	$\alpha \to \beta$ 其中:$\alpha, \beta \in (N \cup \Sigma)^*$

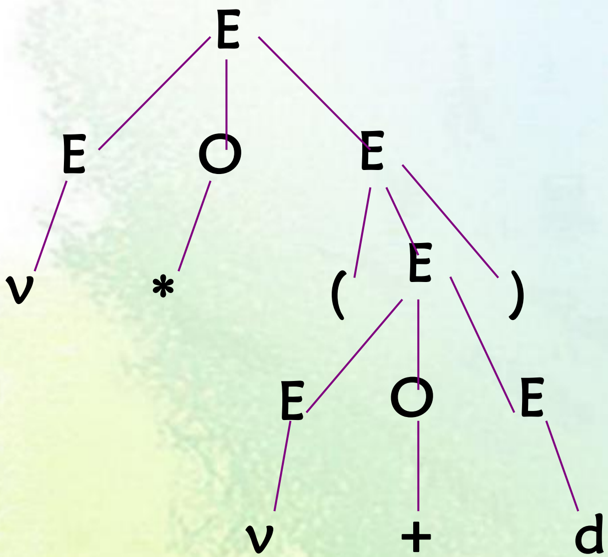
文法与语言的Chomsky分类方法

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) $E \to EOE$

(2) $E \to (E)$

(3) $E \to v$

(4) $E \to d$

(5) $O \to +$

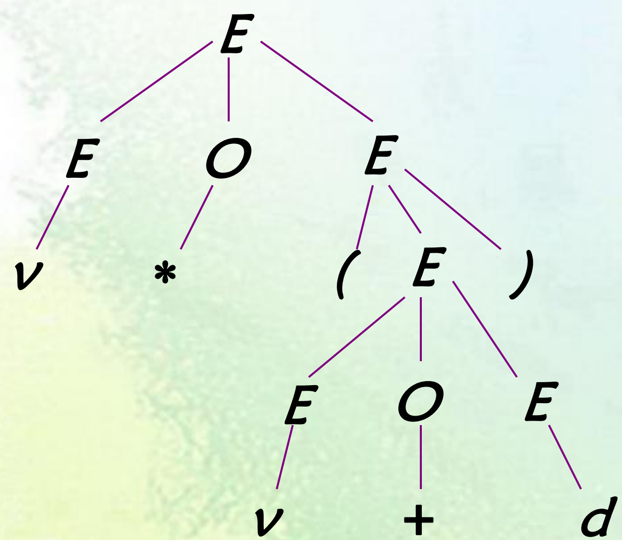
(6) $O \to *$

$$v*(v+d) \xrightarrow{(4)} v*(v+E) \xrightarrow{(6)} vO(v+E) \xrightarrow{(3)} vO(E+E)$$

$$\xrightarrow{(5)} vO(EOE) \xrightarrow{(1)} vO(E) \xrightarrow{(2)} vOE \xrightarrow{(3)} EOE \xrightarrow{(1)} E$$

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) $O \rightarrow *$

$$E \xrightarrow{(1)} EOE \xrightarrow{(6)} E*E \xrightarrow{(2)} E*(E) \xrightarrow{(3)} v*(E)$$

$$\xrightarrow{(1)} v*(EOE) \xrightarrow{(5)} v*(E+E) \xrightarrow{(3)} v*(v+E) \xrightarrow{(4)} v*(v+d)$$

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孩子.

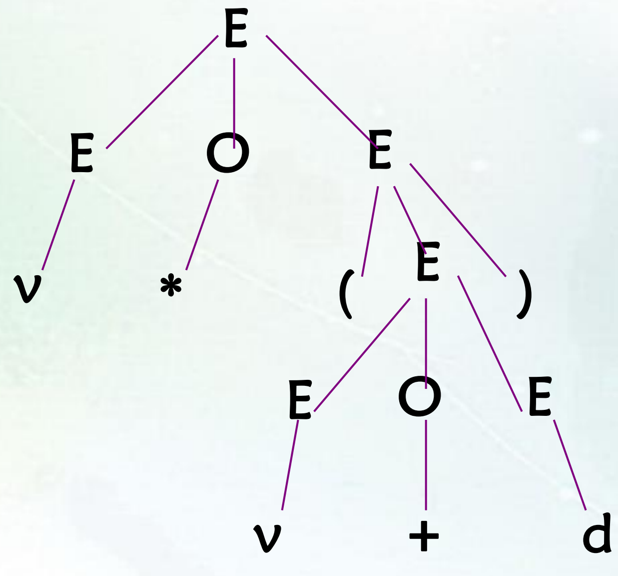
(3) 如果一个内部结点标记为 A，而其孩子从左至右分别标记为$X_1,X_2, \cdots,X_k$，则 $A\rightarrow X_1X_2\cdots X_k$ 是 P 中的一个产生式.

（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 只有唯一的孩子，且 $A\rightarrow\varepsilon$ 是P 中的一个产生式.

几个术语

- **句型**:从上图的分析树可知：符号串i+i*i的句型有…。
- **短语**:子树的末端结点（叶子）形成的符号串(相对于子树的根)。
- **简单短语**：只有父子两代的子树的末端结点（叶子）形成的符号串。
- **句柄**：最左的简单短语。



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→V=E

V→i

E → F+F | F

F →(E) | i

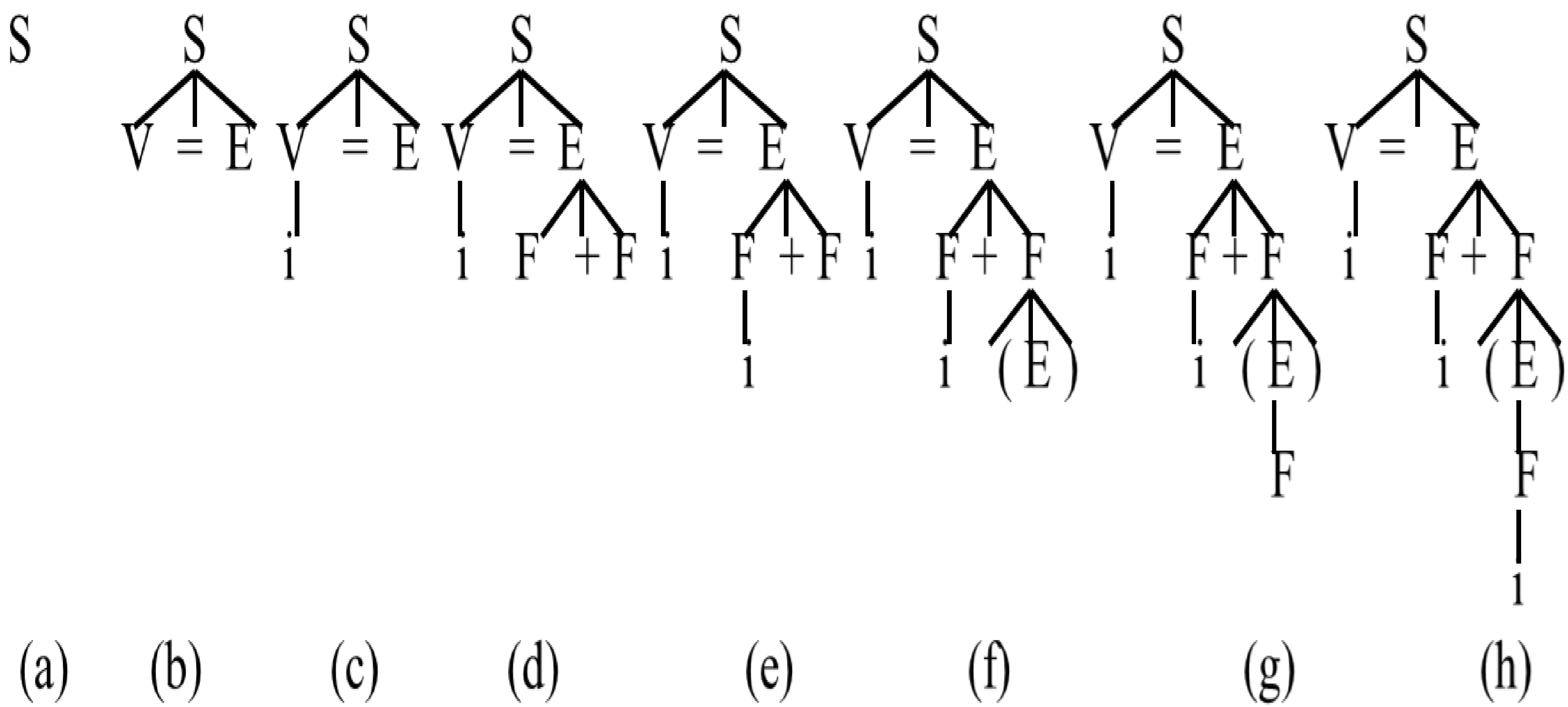
分析所输入的符号串i=i+(i)是否为其句子，并画出相应的分析树。

1. 自顶向下

2. 自底向上

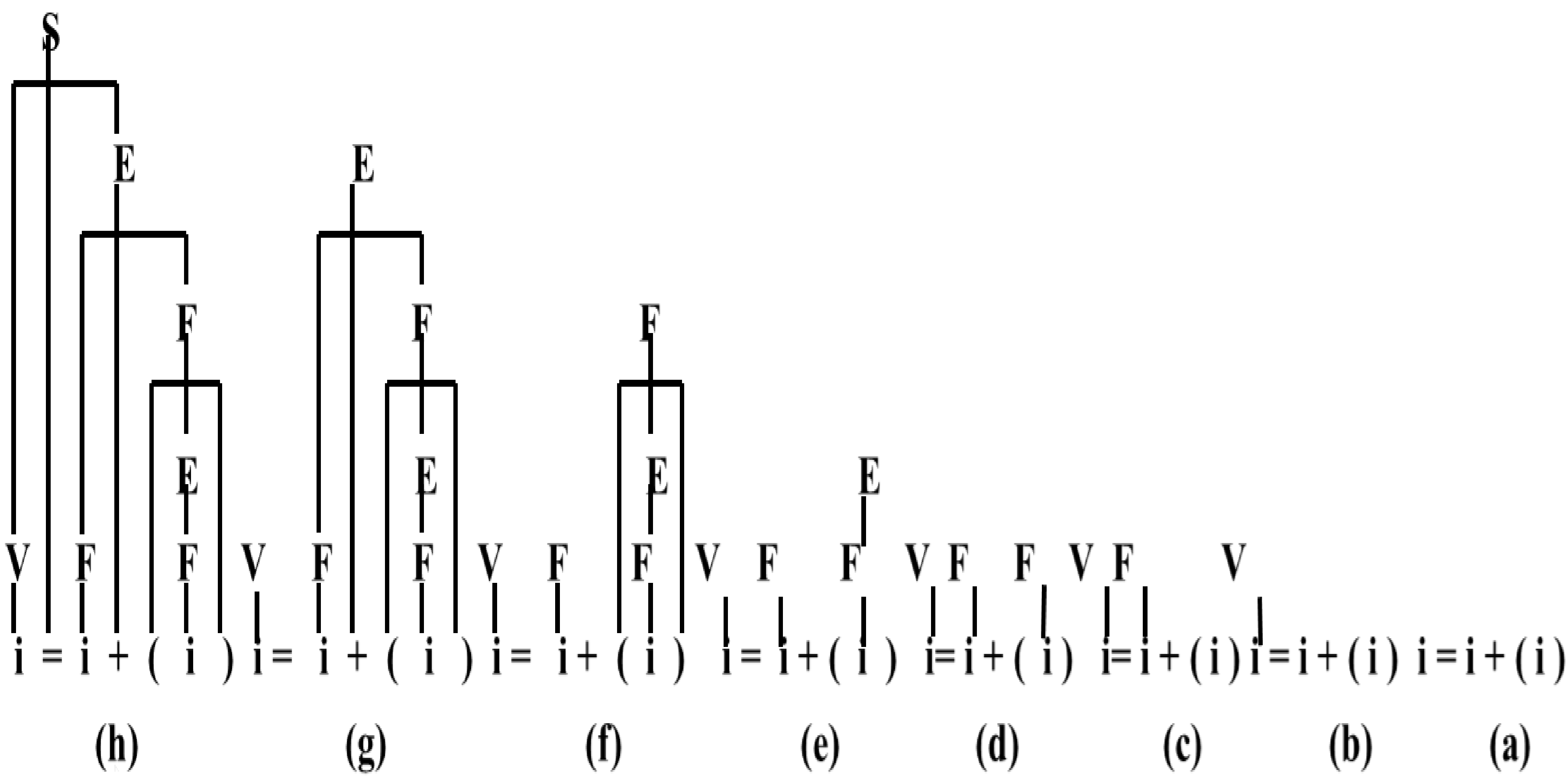
自顶向下分析：不断进行直接推导的过程。

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|E-T|T

T::=T*F|T/F|F

F::=(E)|i

输入符号串是i-i*i,

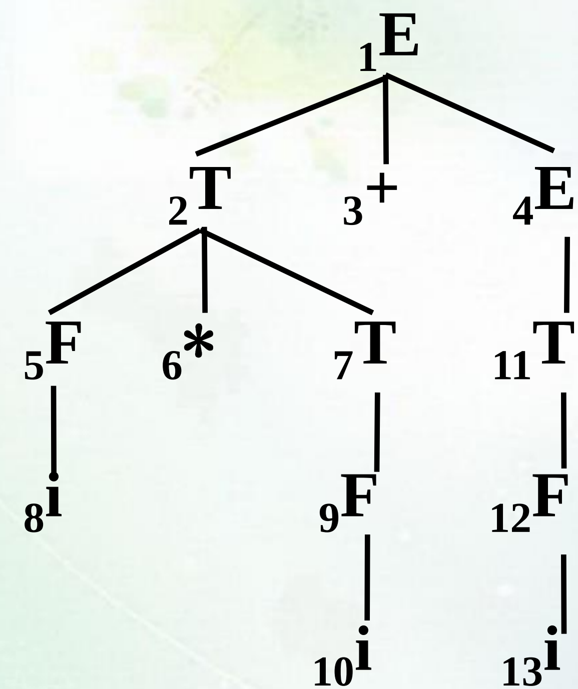
推导为：

E=>T+E=>F*T+E=>i*T+E

=>i*F+E=> i*i+E=> i*i+T

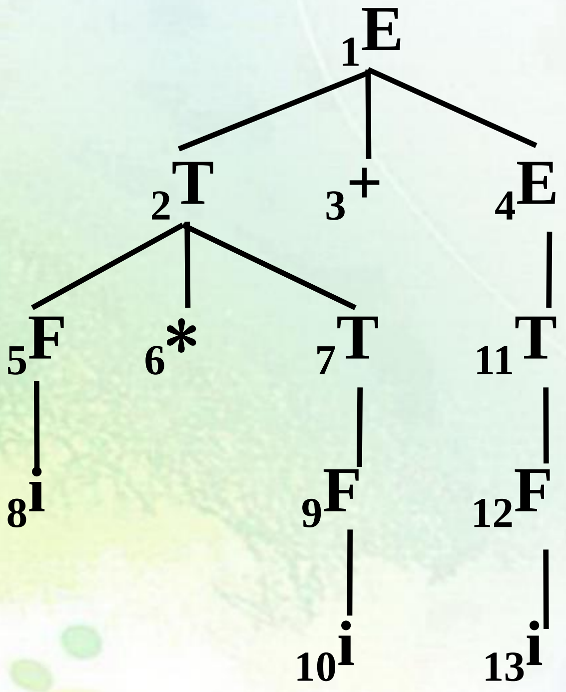
=> i*i+F=> i*i+i

可构造语法分析树如图：

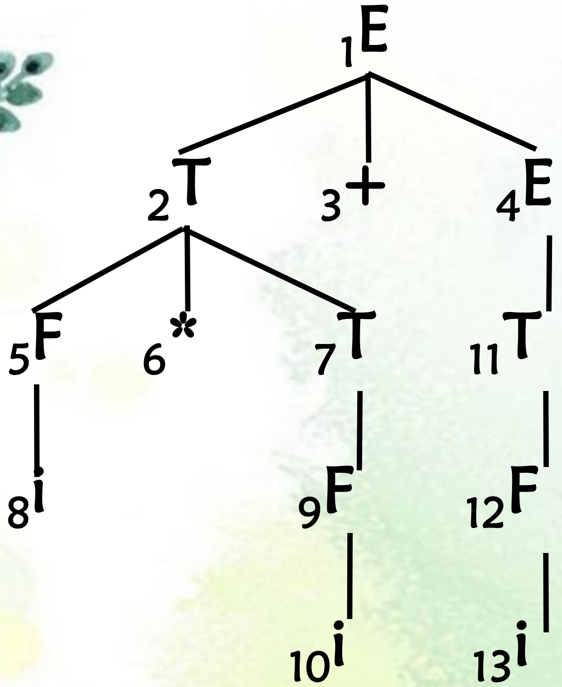


确定每个结点在语法分析树中的位置，
包括各个结点之间的相互关系，便确定了语法分析树，
因此为语法分析树结点可设计如下的数据结构：

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

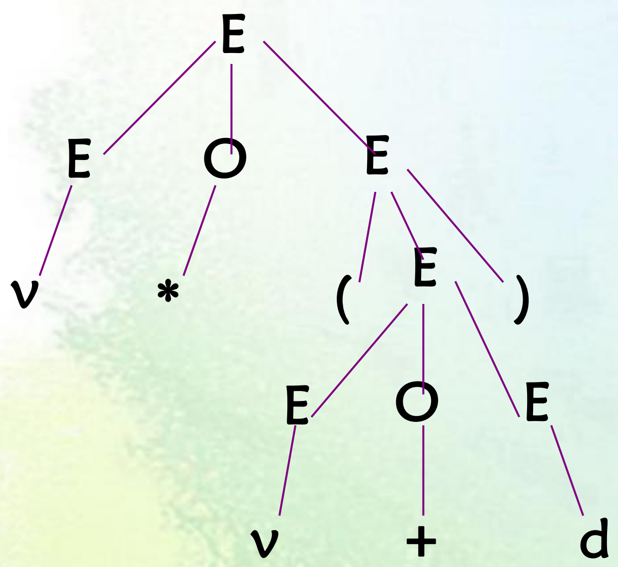
(1)优点:分析树可以反映推导的全过程，信息齐全。

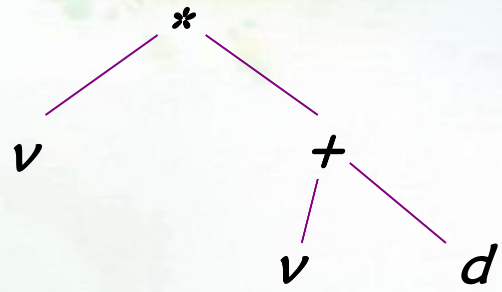
(2)缺点：过于复杂,耗费空间.

解决方法

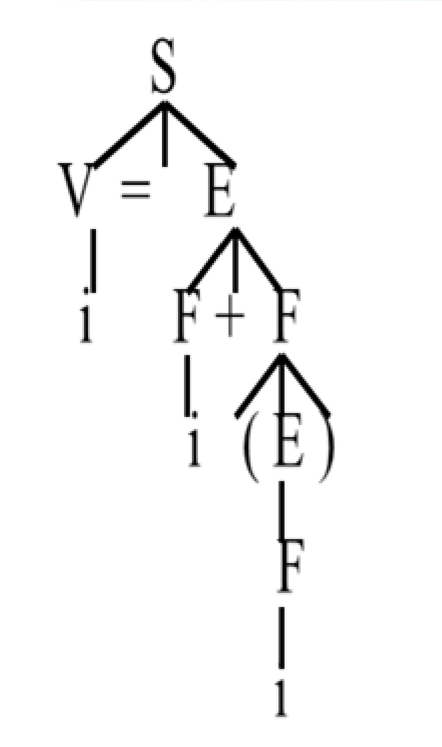
- 压缩——只存储有用信息——对于后续阶段来说
- 语法树

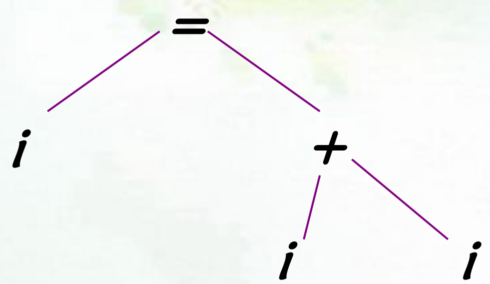
分析树与语法树





分析树与语法树





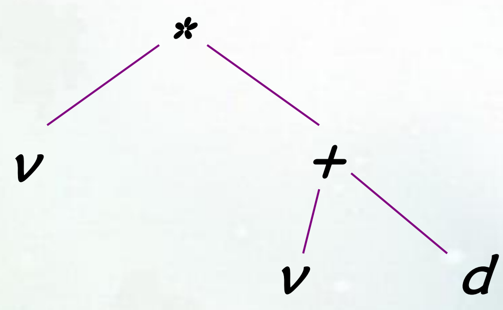
语法树

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

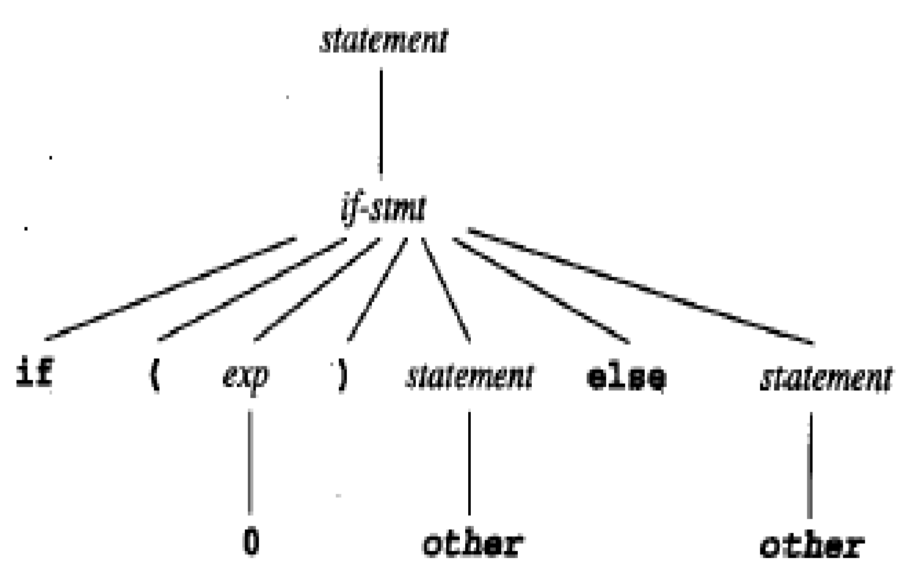
例. if语句的文法：

statement → *if-stmt* | *other*

if-stmt→if(*exp*)*statement*|if(*exp*) *statement* else *statement*

exp → 0 | 1

串： if (0) other else other



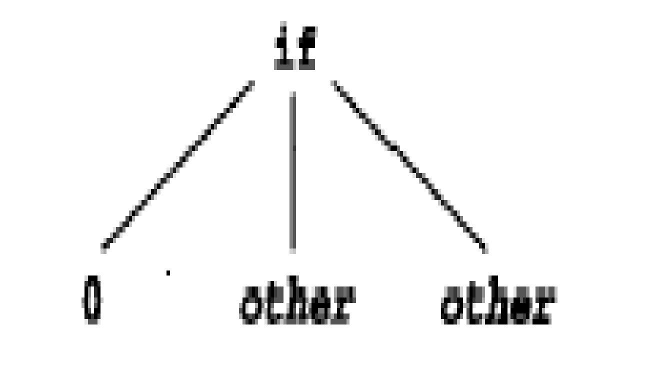
问题:分析该树中对后续分析阶段有用的信息有哪些?

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=({if,then,else, e , a},{S},P,S)

G3[S]:

S→if e then S

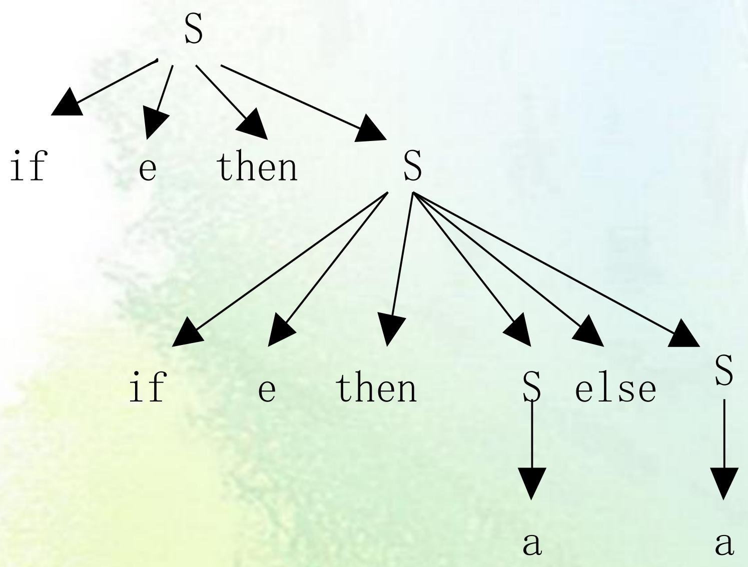
| if e then S else S

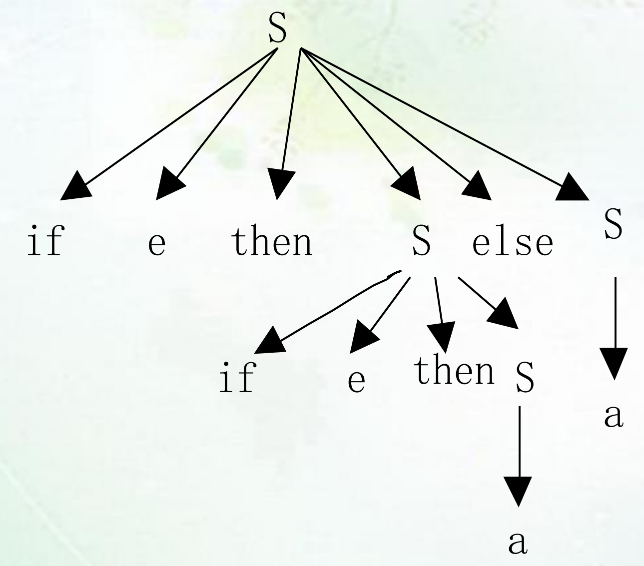
| a

句子if e then if e then a else a

最左推导1	最左推导2
S⇒if e then S ⇒if e then if e then S else S ⇒if e then if e then a else S ⇒if e then if e then a else a	S⇒if e then S else S ⇒if e then if e then S else S ⇒if e then if e then a else S ⇒if e then if e then a else a

• 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＝({if,then,else, e , a},{S},P,S)

G3[S]：

$S \rightarrow$ if e then S

| if e then S 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 → matched-stmt | unmatched-stmt

matched-stmt → if e then matched-stmt else matched-stmt | a

unmatched-stmt → if e then S | if e then matched-stmt 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匹配的关键字来作为语句的结束。

如： if x!=0 then

```
if x!=0 then
        if y=1/x then ok:= 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：$L1=\{ a^i b^j c^k \mid i, j, k \geqslant 1 \}$

如何为语言构造文法

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

G1 [S]:

S→ABC

A → Aa|a

B→ Bb|b

C → Cc|c

G1’[S]:

S → Sc|Bc

B → Bb|Ab

A → Aa|a

类似地，可为语言L2={ ab^i| i≥0 }构造两个文法如下：

G[A]: A→ a | Ab

与

G′[A]: A → a|aB　　B → Bb | b

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:

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

例： $G1[S]=\{ S \rightarrow aSa,\ S \rightarrow b \}$

$G2[S]=\{ S \rightarrow aA,\ S \rightarrow b,\ A \rightarrow Sa \}$

语言的等价

设G1、G2 为两个文法,若它们所产生的语言相等,即L(G1)= L(G2)，则称G1和G2等价。

一个问题

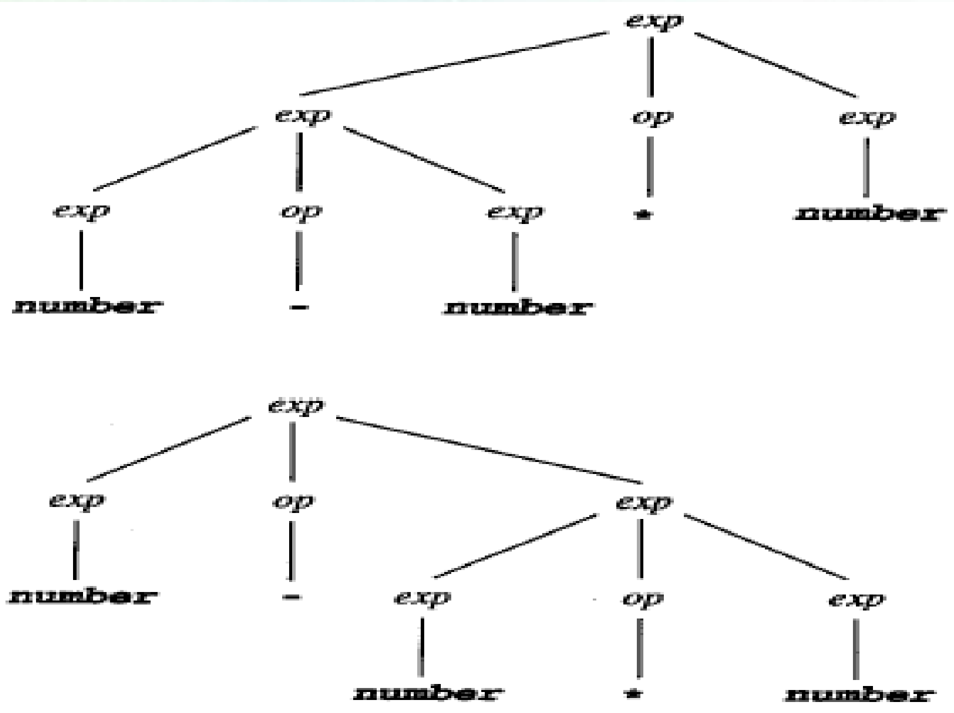
- 如何用文法规则描述算术表达式?

问题分析

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

op → + | - | * |/

对于串34-3*42，则分析树有：



文法改造过程

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

op → + | - | * |/

- 步骤(1) :构造能反映运算符号间优先关系的规则

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→*exp addop exp* | *term*

addop→+|-

term→*term mulop term* | *factor*

mulop→ *|/

factor→(*exp*) | *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

- 同级运算符号的优先关系：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→exp addop exp | term　　左右均可产生

term→term mulop term | factor　左右均可产生

结论:删除右边产生的递归即可

即将规则

$exp \rightarrow exp\ addop\ exp \mid term$

改造为

$exp \rightarrow exp\ addop\ term \mid term$

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 → *exp addop term* | *term*
addop → + | -
term → *term mulop factor* | *factor*
mulop → * |/
factor → (*exp*) | *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

$$A \to Aa \mid b \text{ (左递归)}$$

或

$$A \to aA \mid b \text{ (右递归)}$$

- 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:

$$A \to b\{a\}$$

和

$$A \to \{a\}b$$

可选操作

- EBNF中用方括号[. . .]来表示可选

statement → if-stmt | other

if-stmt → if exp then statement

        | if exp then statement else statement

exp → 0 | 1

可表示:

statement → if-stmt | other

if-stmt → if exp then statement [else statement]

exp → 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 语言的文法规则列表

program→stmt-sequence

stmt-sequence→ stmt-sequence ; statement |statement

statement→ if-stmt |repeat-stmt |assign-stmt | read-stmt|write-stmt

if-stmt→if exp then stmt-sequence end

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

repeat-stmt→repeat stmt-sequence until exp

assign-stmt→identifier := exp

read-stmt→read identifier

write-stmt→write exp

exp→simple-exp comparison-op simple-exp |simple-exp

comparison-op→ < | =

simple-exp→simple-exp addop term | term

addop→ + |-

term→term mulop factor | factor

mulop→ * | /

factor→(exp) | number | identifier

文法中规则有效性的分析

有害规则：形如U→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$S \stackrel{*}{\Rightarrow} \alpha A \beta$，其中$\alpha$，$\beta$属于$V^*$

2. 必须能够从A推出终结符号串t

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

化简文法

例，G[S]：

1) S→Be
2) B→Ce
3) B→Af
4) A→Ae
5) A→e
6) C→Cf
7) D→f

D为不可到达

C为不可终止

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→ ε，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→dB　B→aB| ε

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。

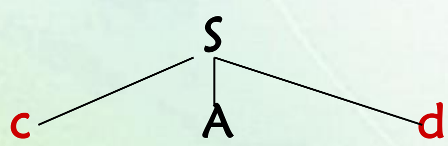
自顶而下语法分析的问题

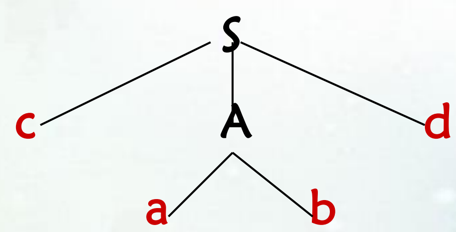
例，文法G[S]： $S \rightarrow cAd$

$A \rightarrow ab$

$A \rightarrow b$

分析输入串w=cabd是否为该文法的句子





推导过程：$S \Rightarrow cAd$

$\Rightarrow cabd$

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：$S \Rightarrow Ad$

$\Rightarrow abd$

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：$A \rightarrow B1|B2|\cdots|Bn$，那么如何确定用哪个右部去替代A？

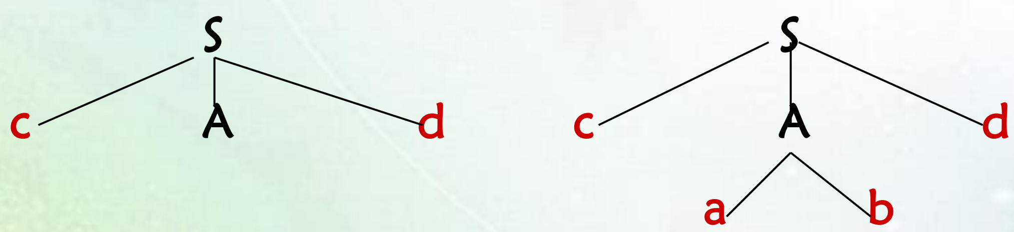
自顶而下语法分析的问题

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：$S \Rightarrow cAd$

$\Rightarrow cabd$

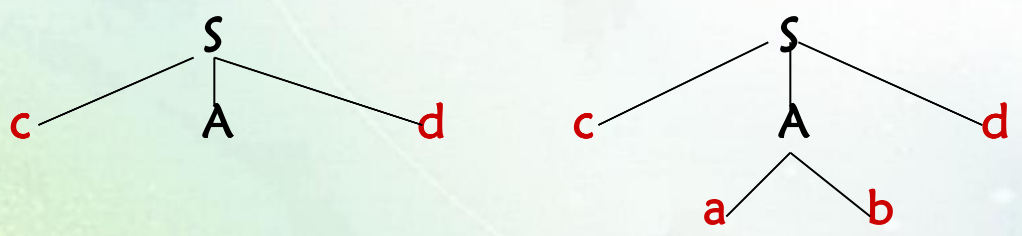
自顶而下语法分析的问题

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：$S \Rightarrow cAd$

$\Rightarrow cabd$

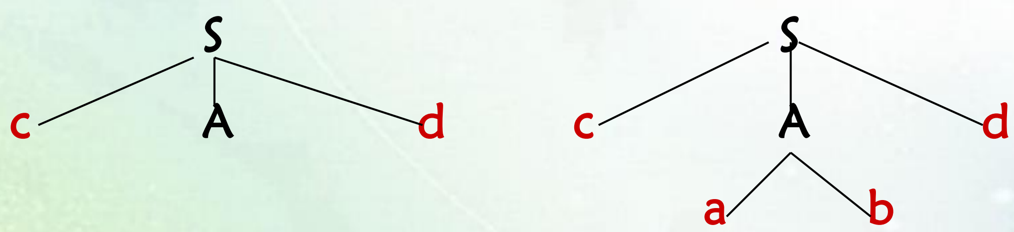
自顶而下语法分析的问题

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：$S \Rightarrow cAd$

$\Rightarrow c\underline{ab}d$

自顶而下语法分析的问题

例，文法G[S]：

$S \rightarrow cAd$

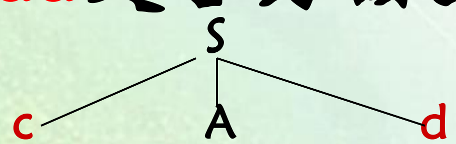
$S \rightarrow cB$

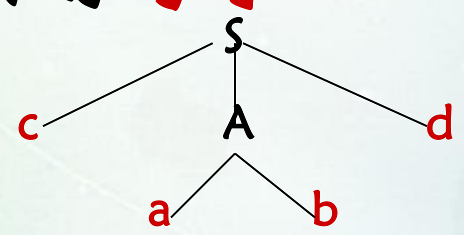
$A \rightarrow ab$

$A \rightarrow a$

$B \rightarrow aa$

分析输入串w=caa是否为该文法的句子





推导过程：$S \Rightarrow cAd$

$\Rightarrow cabd$

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： $A \rightarrow \alpha\beta|\alpha\gamma$

或 $A \rightarrow \alpha\beta1|\alpha\beta2|\ldots|\alpha\beta n$

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a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：$A \rightarrow \alpha\beta1|\alpha\beta2|\ldots|\alpha\beta n$

提取左公因子后变为：

$A \rightarrow \alpha(\beta1|\beta2|\ldots|\beta n)$

或：

$A \rightarrow \alpha A'$

$A' \rightarrow \beta1|\beta2|\ldots|\beta n$

TINY 语言的文法规则列表

program→stmt-sequence

stmt-sequence→ stmt-sequence ; statement |statement

statement→ if-stmt |repeat-stmt |assign-stmt | read-stmt|write-stmt

if-stmt→if exp then stmt-sequence end

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

repeat-stmt→repeat stmt-sequence until exp

assign-stmt→identifier := exp

read-stmt→read identifier

write-stmt→write exp

exp→simple-exp comparison-op simple-exp |simple-exp

comparison-op→ < | =

simple-exp→simple-exp addop term | term

addop→ + |-

term→term mulop factor | factor

mulop→ * | /

factor→(exp) | number | identifier

EBNF中TINY语言的文法

program → *stmt-sequence*
stmt-sequence → *statement* { **;** *statement* }
statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt → **if** *exp* **then** *stmt-sequence* [**else** *stmt-sequence*] **end**
repeat-stmt → **repeat** *stmt-sequence* **until** *exp*
assign-stmt → **identifier :=** *exp*
read-stmt → **read identifier**
write-stmt → **write** *exp*
exp → *simple-exp* [*comparison-op simple-exp*]
comparison-op → **<** | **=**
simple-exp → *term* { *addop term* }
addop → **+** | **-**
term → *factor* { *mulop factor* }
mulop → ***** | **/**
factor → **(** *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→B1|B2|…|Bn，那么如何确定用哪个右部去替代A？

例，文法G[S]：

S → Ad | Bc
A → ab
B → 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

G[S]：　　$S \rightarrow Ad \mid Bc$

$A \rightarrow ab$

$B \rightarrow aa$

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

$x_i \overset{*}{\Rightarrow} T_i v$　和　$x_j \overset{*}{\Rightarrow} T_j w$,

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

结 论

为了

(1) 提高分析效率,避免回溯

(2) 解决诸如 S→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first(C)=?

（2）求出first(A)=?

first集合计算方法

考虑情况2:

G[S]={

S→AB|CD

A→aB|dD

B→cC|bD

C→ef |gh

D→i | j

}

求出first(A)=?

求出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first(D)=?

求出first(A)=?

求出first(S)=?

First集合计算方法之归纳1

$$
\text{First}(x)=\begin{cases}
x & x\text{为终结符号}\\
x & x=\varepsilon\\
\text{First}(x_1)-\{\varepsilon\} & X\rightarrow X_1X_2\ldots X_n\\
\quad\cup\ \text{First}(x_2)-\{\varepsilon\} & X\rightarrow X_2\ldots X_n \quad \varepsilon\in\text{first}(x_1)\\
\qquad\cup\ \ldots\ldots\ldots & \\
\qquad\quad\cup\ \text{First}(x_n)-\{\varepsilon\} & X\rightarrow X_n \quad \varepsilon\in\text{first}(x_i)\ 1\le i<n\\
\qquad\qquad\cup\ \varepsilon & \varepsilon\in\text{first}(xi)\quad 1\le i\le n
\end{cases}
$$

对于规则X→x1x2…xn，first(x)的计算算法如下：

```
First(x)={};
K=1;
While (k<=n)
 {   if （xk 为终结符号或ε） first(xk)=xk;
     first(x)=first(x) ∪  first(xk) - {ε}
     If ( ε ∉ first(xk) ) break;
     k++;
 }
 If (k==n+1) first(x)=first(x) ∪ ε
```

First集合计算方法之归纳2

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

递归表达

请自行完成算法！

EBNF中TINY语言的文法

program → *stmt-sequence*
stmt-sequence → *statement* { **;** *statement* }
statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt → **if** *exp* **then** *stmt-sequence* [**else** *stmt-sequence*] **end**
repeat-stmt → **repeat** *stmt-sequence* **until** *exp*
assign-stmt → **identifier** **:=** *exp*
read-stmt → **read** **identifier**
write-stmt → **write** *exp*
exp → *simple-exp* [*comparison-op simple-exp*]
comparison-op → **<** | **=**
simple-exp → *term* { *addop term* }
addop → **+** | **-**
term → *factor* { *mulop factor* }
mulop → ***** | **/**
factor → **(** *exp* **)** | **number** | **identifier**

自顶向下分析法的问题

文法G[S]:

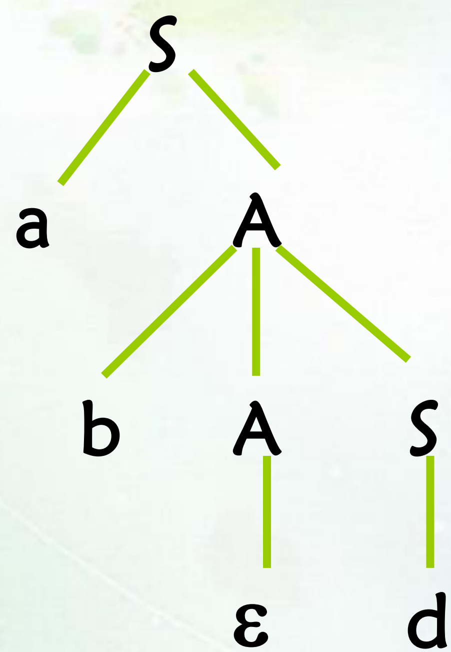
$$S \rightarrow aA \mid d$$
$$A \rightarrow bAS \mid \varepsilon$$

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→ε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→ε。
- ——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Follow (A)＝？
- 2. 若存在规则$B\rightarrow\alpha A\gamma$，则Follow($A$)＝？
- 3. 若存在规则$B\rightarrow\alpha A\gamma$，且$\varepsilon$在First($\gamma$)中，则Follow($A$)＝？

Follow 集合的定义

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

集合Follow(A)的定义如下：

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

Follow 计算的算法设计

G[S]={

S→ABC

A→aB|bB

B→cC| ε

C→ef | gh

}

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

计算Follow集合的算法

1.初始化：

1.1 Follow(开始符号)={ \$ }

1.2 其他任何一个非终结符号A，则执行 Follow(A)={ }

2.循环：反复执行

2.1 循环：对于文法中的每条规则 $A \rightarrow X_1X_2 \dots X_n$ 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 $First(X_{i+1}X_{i+2} \dots X_n) - \{\varepsilon\}$ 添加到 $Follow(X_i)$

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

直到任何一个Follow集合的值都没有发生变化为止。

$$A \rightarrow X_1X_2 \dots X_iX_{i+1} \dots X_n$$

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]:

$A \rightarrow Aa|Bb|Cc$

$B \rightarrow Ab|Bb$

$C \rightarrow Ac|Cb$

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

$A1 , ... , Am$;

(2)执行循环语句：

```
for( i=1; i<=m ; i++ )
{
    for ( j=1; j<=i-1; j++)
      将规则Ai→Ajα改写；
        //改写方法如下：如果Aj→β1|β2|...|βk
        //  Ai →β1α| β2α | ... | βkα
      消除Ai 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]：

$Z \rightarrow (Sbc \mid dS)Z'$ $\qquad Z' \rightarrow aZ' \mid \varepsilon$

$S \rightarrow (dSZ'ef \mid gSh)S'$ $\qquad S' \rightarrow bcZ'efS' \mid \varepsilon$

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

$A \rightarrow B\alpha \mid \ldots$

$B \rightarrow A\beta \mid \ldots$

- 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**

$A \to A\alpha 1|A\alpha 2|\ldots|A\alpha n|\beta 1|\beta 2|\ldots|\beta m$

其中β1，. . . .，βm 均不以A开头。

可先改为：

$A \to A(\alpha 1|\alpha 2|\ldots|\alpha n)|(\beta 1|\beta 2|\ldots|\beta m)$

接着使用情况1的方法进行改造，如改为：

$A \to \beta 1A' \mid \beta 2A' \mid \ldots \mid \beta mA'$

$A' \to \alpha 1A' \mid \alpha 2A' \mid \ldots \mid \alpha n A' \mid \varepsilon$

- 情况3：更复杂的左递归——间接左递归

$$A \rightarrow Bb \mid ...$$

$$B \rightarrow Aa \mid ...$$

（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 语言的文法规则列表

program→stmt-sequence

stmt-sequence→ stmt-sequence ; statement |statement

statement→ if-stmt |repeat-stmt |assign-stmt | read-stmt|write-stmt

if-stmt→if exp then stmt-sequence end

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

repeat-stmt→repeat stmt-sequence until exp

assign-stmt→identifier := exp

read-stmt→read identifier

write-stmt→write exp

exp→simple-exp comparison-op simple-exp |simple-exp

comparison-op→ < | =

simple-exp→simple-exp addop term | term

addop→ + |-

term→term mulop factor | factor

mulop→ * | /

factor→(exp) | number | identifier

EBNF中TINY语言的文法

program → *stmt-sequence*
stmt-sequence → *statement* { **;** *statement* }
statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt → **if** *exp* **then** *stmt-sequence* [**else** *stmt-sequence*] **end**
repeat-stmt → **repeat** *stmt-sequence* **until** *exp*
assign-stmt → **identifier** **:=** *exp*
read-stmt → **read** **identifier**
write-stmt → **write** *exp*
exp → *simple-exp* [*comparison-op* *simple-exp*]
comparison-op → **<** | **=**
simple-exp → *term* { *addop* *term* }
addop → **+** | **-**
term → *factor* { *mulop* *factor* }
mulop → ***** | **/**
factor → **(** *exp* **)** | **number** | **identifier**

3型文法与有穷自动机（FA）

3型文法(右线性)与有穷自动机（FA)

3型文法产生的语言是有穷自动机（FA）所接受的集合.

<u>定理</u> 设G=（V_N，V_T，P，S）是3型文法，则存在一个有穷自动机 M=(K, Σ , f, A, Z)，使得L(M)=L(G)

有穷自动机NFA M 这样构造:

- $\Sigma = V_T$
- K= $V_N \cup \{N\}$, N为一个新状态,它不在V_N中
- A=S
- Z={N}
- 对G中的形如 D→tB的产生式,t为终结符或ε,有f(D,t)=B;

  对G中形如D→t的产生式， t为终结符或ε,有f(D,t)=N;

  对V_T中的每一个a ,有f(N,a)= ϕ

3型文法与有穷自动机（FA）

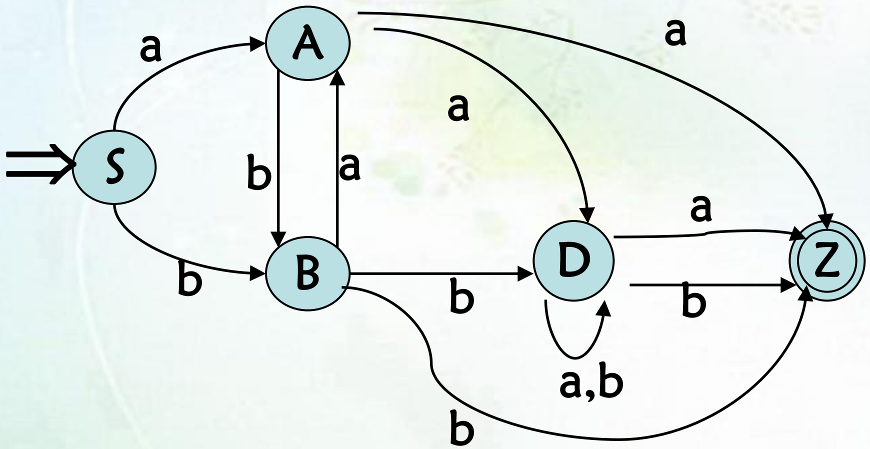
G［S］：

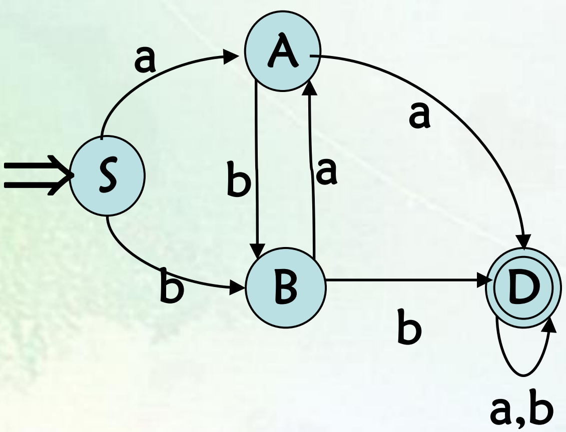
S→aA|bB

A→bB|aD|a

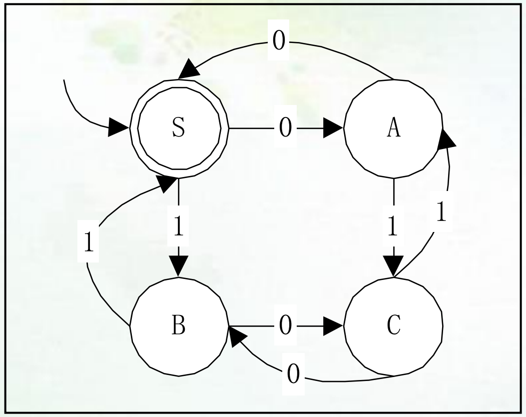
B→aA|bD|b

D→aD|bD|a|b





- 例. 给定右线性文法
- G[S]：
- S→0A|1B|ε
- A→0S|1C
- B→1S|0C
- C→0B|1A
- 试构造相应的DFA。
- 解：根据构造FA的算法，可得：



3型文法(右线性)与有穷自动机（FA）

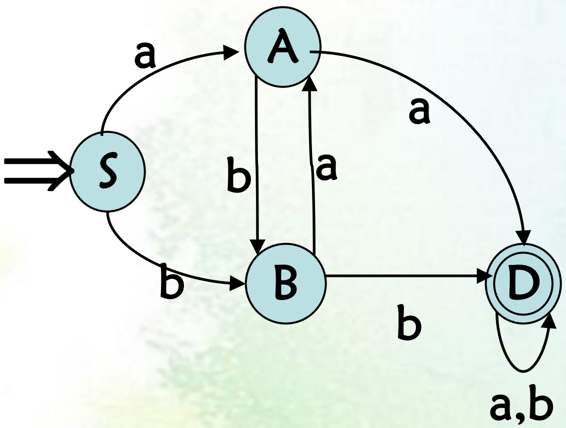
定理 已知一有穷自动机$M=(K, \Sigma, f, A, Z)$,存在有一个3型文法$G=(V_N, V_T, P, S)$,使得$L(G)=L(M)$

G的定义：

- $V_T=\Sigma$
- $V_N=K$
- $S=A$
- 若$f(D,t)=B$，则$D\rightarrow tB$在P中

  若$f(D,t)=B$，且B在Z中，则$D\rightarrow t$在P中

3型文法与有穷自动机（FA）



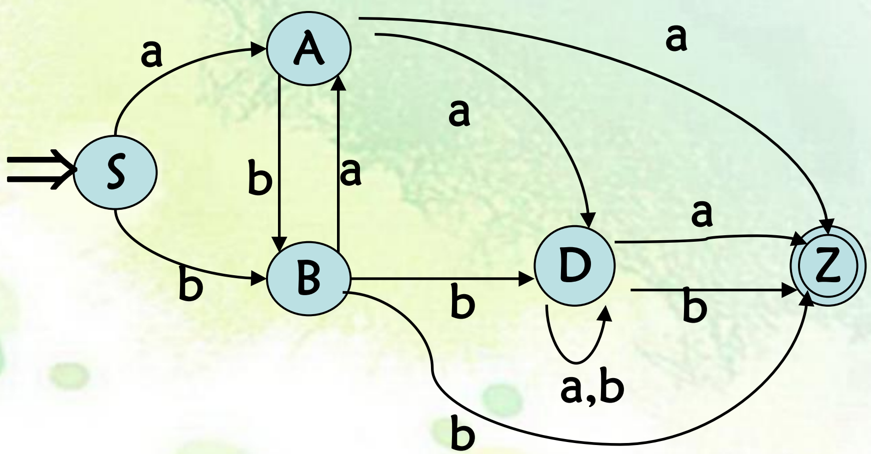
G［S］：

S→aA|bB

A→bB|aD|a

B→aA|bD|b

D→aD|bD|a|b



左线性规则与有穷自动机（FA）

有穷自动机（FA）的构造步骤如下：

步骤1 以符号S为开始状态作结点(假定文法的字符表中不包含符号S)；

步骤2 以每一个非终结符号为状态作结点；

步骤3 对于形如Q→a的每个规则，引一条从开始状态S到状态Q的弧，其标记为a；

对于形如Q→Ra的规则引一条从状态R到Q的弧，其标记为a。其中R为非终结符号，a为终结符号；

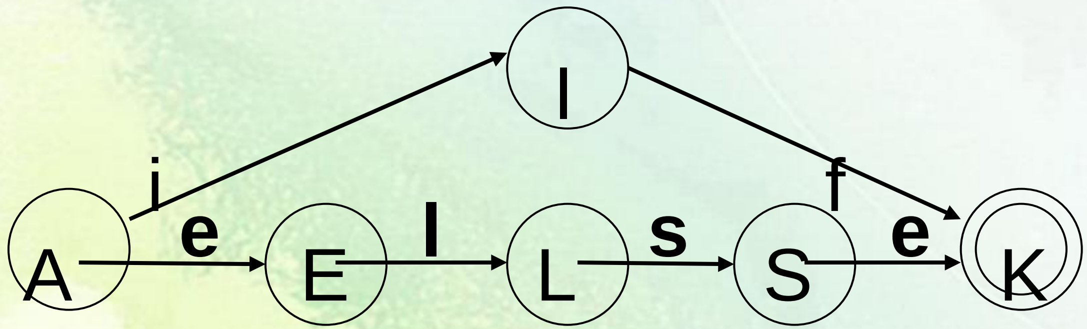
步骤4 文法开始符号相应的状态作为终止状态。

例　画出正规文法G[K]的FA

K→If | Se　　I→i

S→Ls　　　L→El　　E→e

FA：



例 画出下面正规文法的FA。

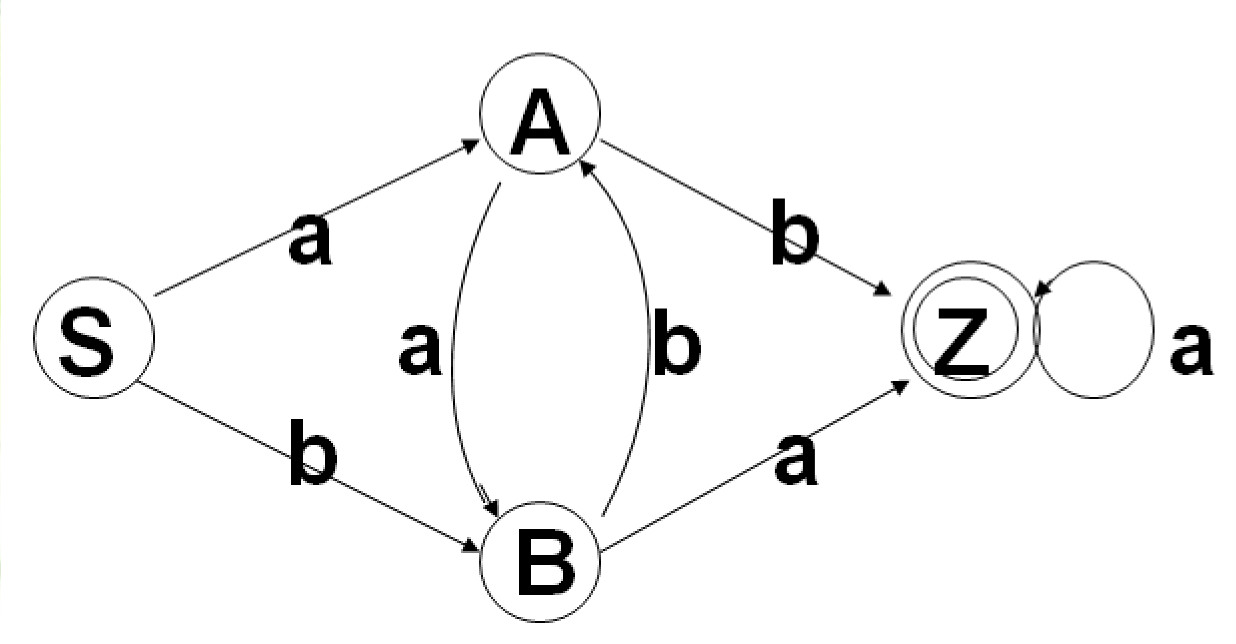
G[Z]:

Z→Za | Ab | Ba

A→Bb | a

B→Aa | b

FA:



正则表达式转换为正规文法

对$\sum$上的正则表达式r，存在一个RG=(V_N,V_T,P,S)：

$L(G)=L(r)$

初始化， $V_T=\sum, S\in V_N$，生成正则表达式 $S\to r$

(R.1) 对形如 $A\to r_1r_2$的正则表达式： $A\to r_1B$

$B\to r_2$

$B\in V_N$

(R.2)对形如$A\to r^*r_1$的正则表达式： $A\to rB$

$A\to r_1$

$B\to rB$

$B\to r_1$ $B\in V_N$

(R.3)对形如$A\to r_1|r_2$的正则表达式: $A\to r_1$

$A\to r_2$

不断应用(R.x)做变换，直到每个产生式右端至多有一个V_N

例 $r=a(a|d)^*$

(1) $S \to a(a|d)^*$

(2) $S \to aA \qquad A \to (a|d)^*$

(3) $A \to (a|d)B$

$A \to \varepsilon$

$B \to (a|d)B$

$B \to \varepsilon$

G[s]:

$S \to aA$

$A \to \varepsilon$

$A \to aB$

$A \to dB$

$B \to aB$

$B \to dB$

$B \to \varepsilon$

$V_T=\{a,d\}$

$V_N=\{S,A,B\}$

正规文法转换为正则表达式

对$G=(V_N,V_T,P,S)$,存在一个$\Sigma=V_T$上的正则表达式r：

$L(r)=L(G)$

$A\rightarrow xB$ ， $B\rightarrow y$ 形成正则表达式 $A=xy$

$A\rightarrow xA|y$ 形成正则表达式 $A=x^*y$

$A\rightarrow x|y$ 形成正则表达式 $A=x|y$

正规文法转换为正则表达式

$G[s]: S \rightarrow aA \mid a$

$A \rightarrow aA \mid a \mid dA \mid d$

$A \rightarrow (a \mid d)A \mid (a \mid d)$

$A \rightarrow (a \mid d)^*(a \mid d)$

$S = a(a \mid d)^*(a \mid d) \mid a$

$= a((a \mid d)^*(a \mid d) \mid \varepsilon)$

$= a((a \mid d)^* \mid \varepsilon)$

$R = a(a \mid d)^*$