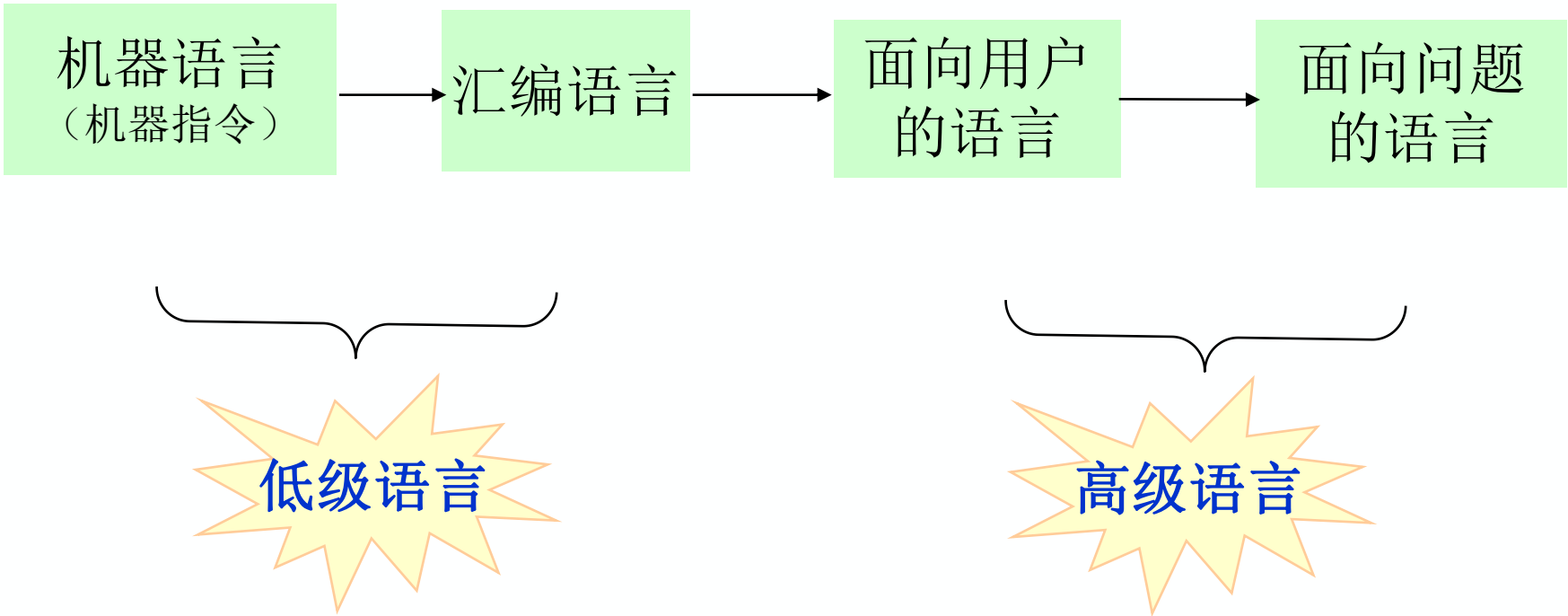
第一章　概 论

- 编译的起源：程序设计语言的发展
- 基本概念
- 编译过程和编译程序构造
- 编译技术的应用

1.1 程序设计语言的发展



- 低级语言（Low level Language)
  - 字位码、机器语言、汇编语言
    - 特点：与特定的机器有关，效率高、灵活，但使用复杂、繁琐、编写费时、易出错
- 高级语言
  - -- Fortran、Pascal、C 语言等
    - 特点：不依赖具体机器，移植性好、便于描述问题处理过程和算法、易使用、易维护等。

用高级语言编制的程序，计算机不能立即执行，必须通过一个“翻译程序”加工，转化为与其等价的机器语言程序，机器才能执行。

这种翻译程序，称之为“编译程序”。

1.2 基本概念

- **源程序**

  用汇编语言或高级语言编写的程序称为源程序。

- **目标程序**

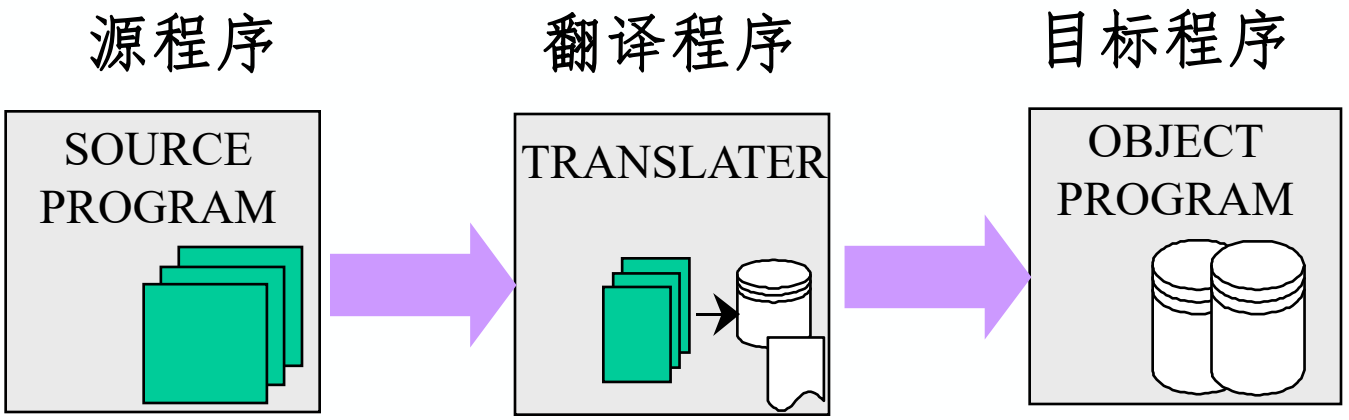
  用目标语言所表示的程序。

  目标语言：可以是某种机器的机器语言或汇编语言，也可以是介于源语言和机器语言之间的“中间语言”，甚至可以是另一种高级语言。

- **翻译程序**

  将源程序转换为目标程序的程序称为翻译程序。它是指各种语言的翻译器，包括汇编程序和编译程序，是汇编程序、编译程序以及各种变换程序的总称。

源程序、翻译程序、目标程序 三者关系：



即源程序是翻译程序的输入，目标程序是翻译程序的输出

- **汇编程序**

若源程序用汇编语言书写，经过翻译程序得到用机器语言表示的程序，这时的翻译程序就称之为汇编程序，这种翻译过程称为“汇编”（Assemble）

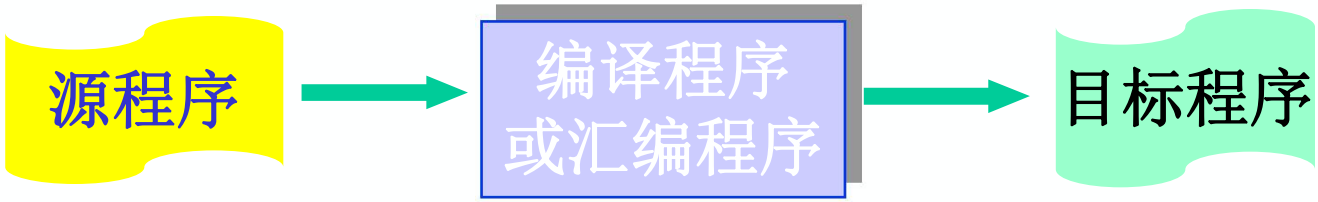
- **编译程序**

若源程序是用高级语言书写，经加工后得到目标程序，这种翻译过程称“编译”（Compile)

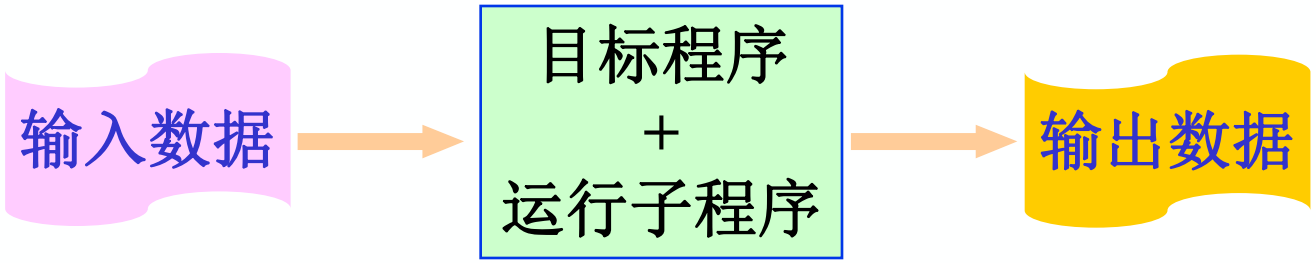
汇编程序与**编译程序**都是**翻译程序**，主要区别是加工对象的不同。

源程序的编译和运行

- 编译或汇编阶段



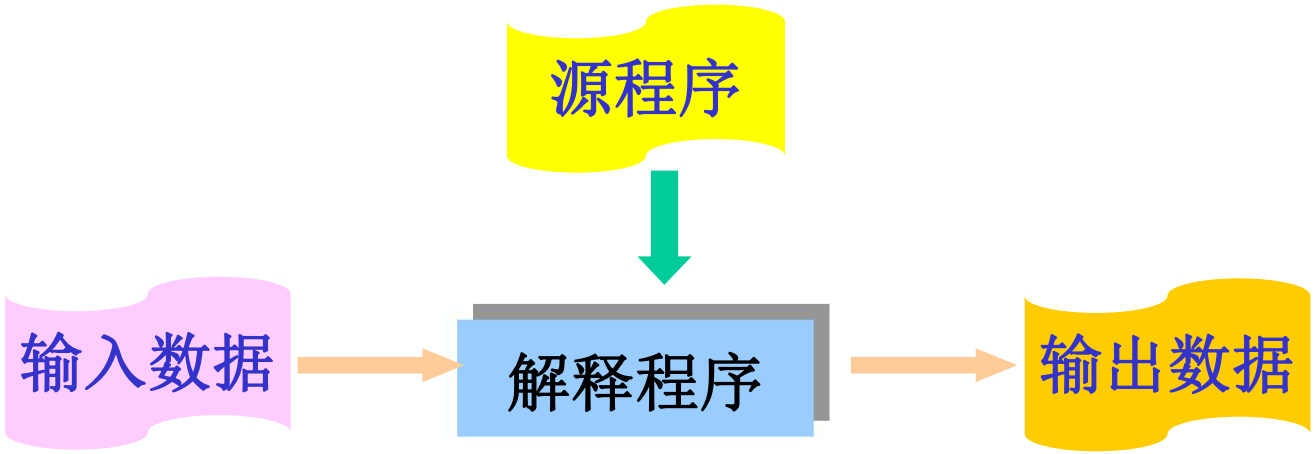
- 运行阶段



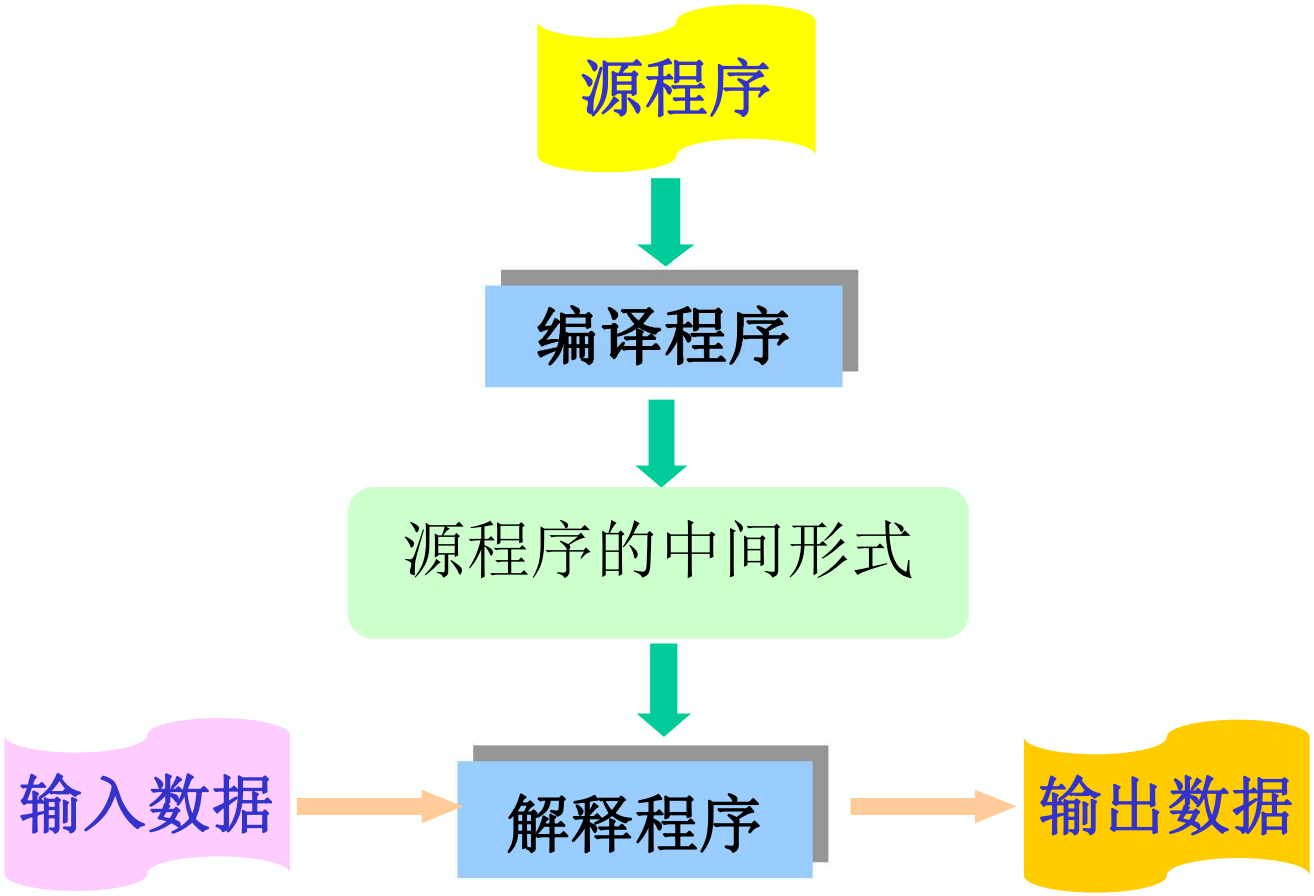
- 解释程序（Interpreter）

  对源程序进行解释执行的程序。

- 工作过程



“编译-解释执行”系统



1.3.1 编译过程

编译过程是指将高级语言程序翻译为等价的目标程序的过程。

习惯上是将编译过程划分为5个基本阶段：

一、词法分析

任务：分析和识别单词。

源程序是由字符序列构成的，词法分析扫描源程序(字符串),根据语言的词法规则分析并识别单词，并以某种编码形式输出。

- 单词：是语言的基本语法单位，一般语言有四大类单词

<1>语言定义的关键字或保留字（如BEGIN、END、IF）

<2>标识符

<3>常数

<4>分界符（运算符） （如+、-、*、/、;、（、） ……）

二、语法分析

任务：根据语法规则（即语言的文法），分析并识别出各种语法成分，如表达式、各种说明、各种语句、过程、函数等，并进行语法正确性检查。

X1:= (2.0 + 0.8) * C1

赋值语句的文法：

<赋值语句>→<变量><赋值操作符><表达式>
<变量>→<简单标识符>
<赋值操作符>→:=
<表达式>→ ……

三、语义分析、生成中间代码

任务：对识别出的各种语法成分进行语义分析，
并产生相应的中间代码。

- 中间代码：一种介于源语言和目标语言之间的中间语言形式
- 生成中间代码的目的：
  〈1〉便于做优化处理；
  〈2〉便于编译程序的移植。
- 中间代码的形式：编译程序设计者可以自己设计，常用的有
  四元式、三元式、逆波兰表示等。

★ 四元式（三地址指令）

X1:= （ 2.0 + 0.8 ） * C1

	运算符	左运算对象	右运算对象	结果
(1)	+	2.0	0.8	T1
(2)	*	T1	C1	T2
(3)	:=	X1	T2	

其中T1和T2为编译程序引入的工作单元

四元式的语义为：
2.0 + 0.8 → T1
T1 * C1 → T2
T2 → X1

这样所生成的四元式与原来的赋值语句在语言的形式上不同，但语义上等价。

四、代码优化

目的：是为了得到高质量的目标程序。

例如：前面的四元式中第一个四元式是计算常量表达式值，该值在编译时就可以算出并存放在工作单元中，不必生成目标指令来计算，这样四元式可优化为：

编译时：　　2.0 + 0.8 → T1

（1）*　T1　　C1　　T2

（2）:=　X1　　T2

五、生成目标程序

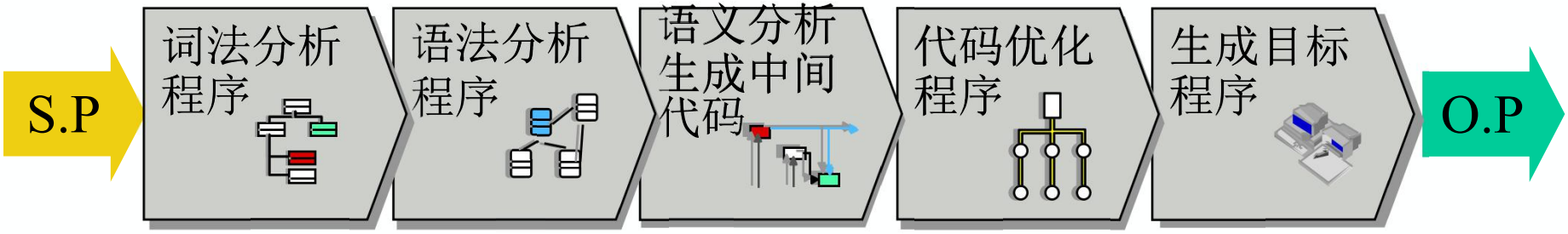
由中间代码很容易生成目标程序（地址指令序列）。这部分工作与机器关系密切 ，所以要根据机器进行。在做这部分工作时（要注意充分利用累加器），也可以进行优化处理。 X1:=（2.0 + 0.8）* C1

注意：在翻译成目标程序的过程中，要切记保持语义的等价性。

1.3.2 编译程序构造

一、 编译程序的逻辑结构

按逻辑功能不同，可将编译过程划分为五个基本阶段，与此相对应，我们将实现整个编译过程的编译程序划分为五个逻辑阶段（即五个逻辑子过程）。



在上列五个阶段中都要做两件事：

（1）建表和查表；（2）出错处理；

所以编译程序中都要包括符号表管理和出错处理两部分

★ 符号表管理

在整个编译过程中始终都要贯穿着建表（填表）和查表的工作。即要及时地把源程序中的信息和编译过程中所产生的信息登记在表格中，而在随后的编译过程中同时又要不断地查找这些表格中的信息。

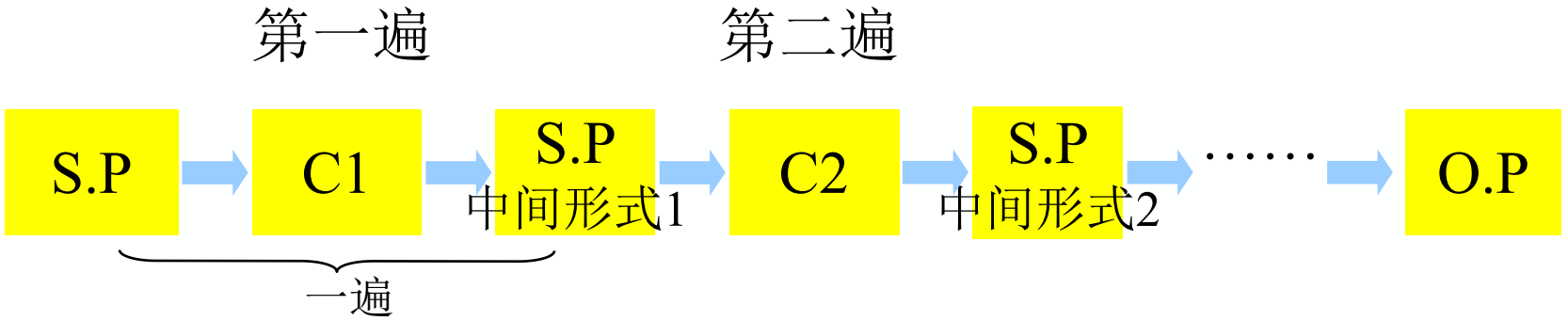
★ 出错处理

规模较大的源程序难免有多种错误，编译程序必须要有出错处理的功能。即能诊察出错误，并能报告用户错误的性质和位置，以便用户修改源程序。出错处理能力的大小是衡量编译程序质量好坏的一个重要指标。

典型的编译程序具有7个逻辑部分

二、 遍（PASS）

遍：对源程序（包括源程序中间形式）从头到尾扫描一次，并做有关的加工处理 ， 生成新的源程序中间形式或目标程序，通常称之为一遍。



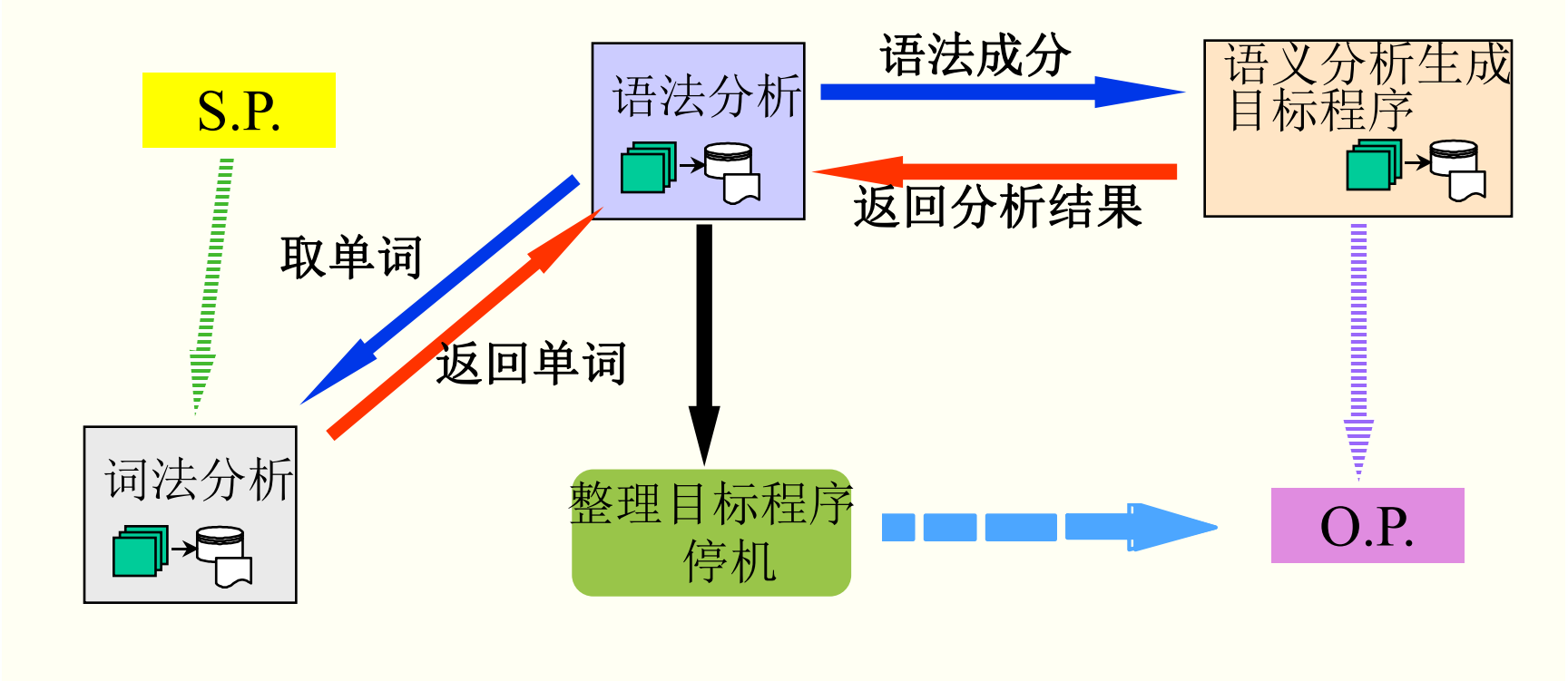
☆ **要注意遍与基本阶段的区别**

五个基本阶段：是将源程序翻译为目标程序在逻辑上要完成的工作。

遍：是指完成上述5个基本阶段的工作，要经过几次扫描处理。

一遍扫描即可完成整个编译工作的称为一遍扫描编译程序

其结构为：



三、前端和后端

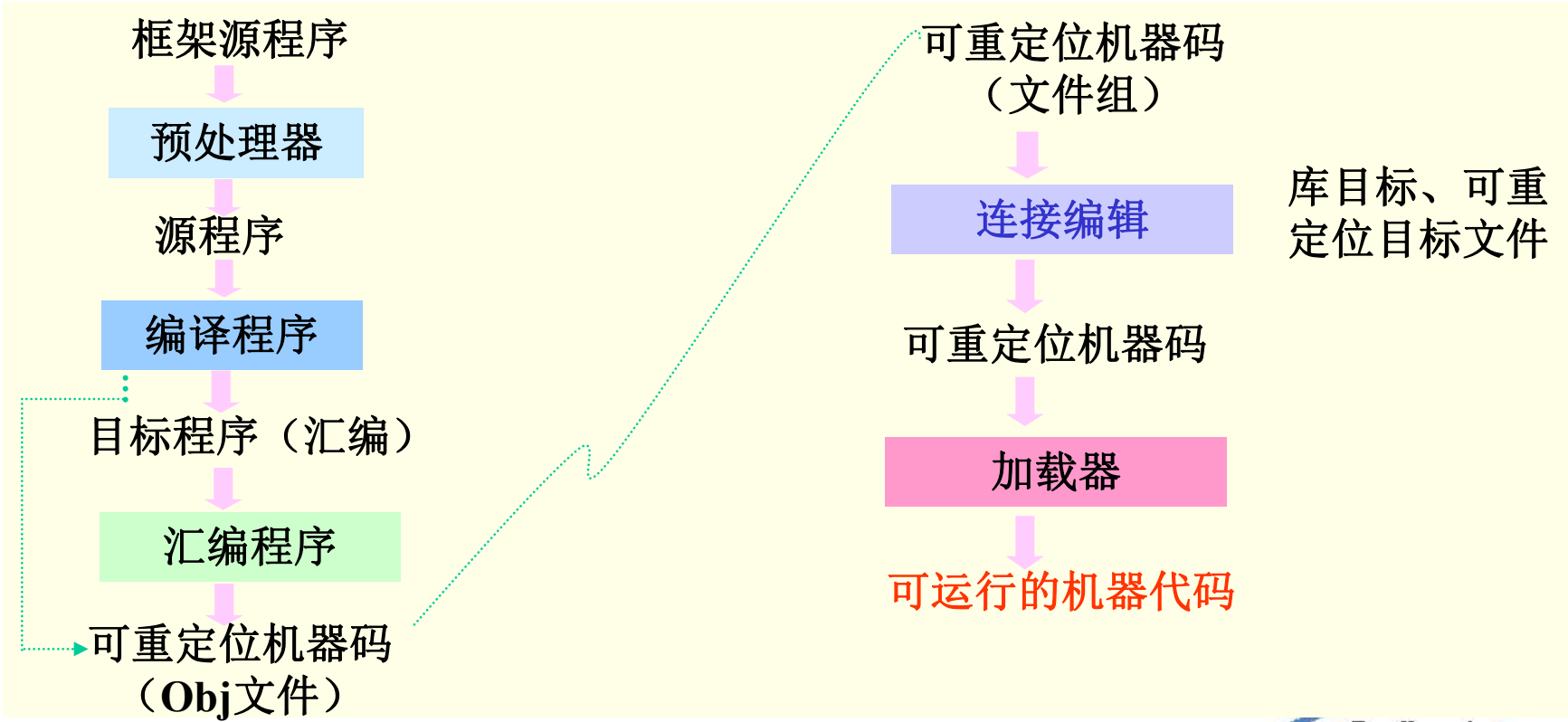
根据编译程序各部分功能，将编译程序分成前端和后端。

前端：通常将与源程序有关的编译部分称为前端。
词法分析、语法分析、语义分析、中间代码生成、代码优化 --------分析部分
特点：与源语言有关

后端：与目标机有关的部分称为后端。
目标程序生成（与目标机有关的优化）
--------综合部分
特点：与目标机有关

四、 编译程序的前后处理器

源程序：多文件、宏定义和宏替换（调用），包含文件
目标程序：一般为汇编程序或可重定位的机器代码



1.4 编译技术的应用

- 语法制导的结构化编辑器
- 程序格式化工具
- 软件分析与测试工具
- 程序理解工具
- 高级语言的翻译工具
- 等等

第二章 文法和语言的概念和表示

- 预备知识 – 形式语言基础
- 文法和语言的定义
- 若干术语和重要概念
- 文法的表示：扩充的BNF范式和语法图
- 文法和语言的分类

2.1 预备知识

一、字母表和符号串

字母表：　符号的非空有限集　例：∑={a，b，c}

符号：　字母表中的元素　例：a，b，c

符号串：　符号的有穷序列　例：a，aa，ac，abc，..

空符号串：无任何符号的符号串（ε）

符号串的形式定义

有字母表∑，定义：

（1）ε是∑上的符号串；

（2）若x是∑上的符号串，且a∈ ∑，则ax或xa是∑上的符号串；

（3）y是∑上的符号串，iff（当且仅当）y可由（1）和（2）产生。

符号串集合：由符号串构成的集合。

- 通常约定：
  - 用英文字母表开头的小写字母和字母表靠近末尾的大写字母来表示符号
    如：a, b, c, d, … , r 和 S, T , U, V , W， X, Y, Z
  - 用英文字母表靠近末尾的小写字母来表示符号串
    如： s, t, u, v, w , x, y, z
  - 用英文字母表开头的大写字母来表示符号串集合
    如：A, B, C, D, … , R

二、符号串和符号串集合的运算

1. 符号串相等：若x、y是集合上的两个符号串，则x＝y iff（当且仅当）组成x的每一个符号和组成y的每一个符号依次相等。

2. 符号串的长度：x为符号串，其长度|x|等于组成该符号串的符号个数。

例： x＝STV ， |x|=3

3. 符号串的联接：若x、y是定义在Σ上的符号串，　且x＝XY，y＝YX，则x和y的联接 xy＝XYYX也是Σ上的符号串。

注意：一般xy≠yx，而 ε x＝x ε

4. 符号串集合的乘积运算：令A、B为符号串集合，

定义　　$AB=\{xy \mid x\in A, y\in B\}$

例：A＝{s, t}, B = {u,v},　AB= ？

{su，sv，tu，tv}

因为 ε x＝x ε ＝x，所以{ ε }A=A{ ε } =A

5. 符号串集合的幂运算：有符号串集合A，定义

$A^0 = \{ \varepsilon \}$, $A^1 = A$, $A^2 = AA$, $A^3 = AAA$,

…… …… $A^n = A^{n-1}A = AA^{n-1}$ ， $n>0$

6. 符号串集合的闭包运算：设A是符号串集合，定义

$A^+ = A^1 \cup A^2 \cup A^3 \cup \ldots\ldots \cup A^n \cup \ldots\ldots$

称为集合A的正闭包。

$A^* = A^0 \cup A^+$

称为集合A的闭包。

例： A={x,y}

$A^+ = \{\underbrace{x,y}_{A^1}, \underbrace{xx,xy,yx,yy}_{A^2}, \underbrace{xxx,xxy,xyx,xyy,yxx,yxy,yyx,yyy,}_{A^3} \ldots\ldots\}$

$A^* = \{\underbrace{\varepsilon}_{A^0}, \underbrace{x,y}_{A^1}, \underbrace{xx,xy,yx,yy}_{A^2}, \underbrace{xxx,xxy,xyx,xyy,yxx,yxy,yyx,yyy,}_{A^3} \ldots\ldots\}$

★为什么对符号、符号串、符号串集合以及它们的运算感兴趣？

若A为某语言的基本字符集　　（把字符看作符号）

A＝{a, b, ……z, 0, 1, ……, 9, +, －, ×, _, /, (,), =, ……}

B为单词集　　（单词是符号串）

B ={begin, end, if, then, else, for, ……,<标识符>,<常量>, ……}

则$B \subset A^*$ 。

（把单词看作符号，句子便是符号串）

语言的句子是定义在B上的符号串。

若令C为句子集合，则$C \subset B^*$，程序$\subset C$

2.2文法的非形式讨论

1.什么是文法：文法是对语言结构的定义与描述。即从形式上用于描述和规定语言结构的称为“文法”（或称为“语法”）。

例：有一句子：“我是大学生”。这是一个在语法、语义上都正确的句子，该句子的结构（称为语法结构）是由它的语法决定的。在本例中它为“主谓结构”。

如何定义句子的合法性？
- 有穷语言
- 无穷语言

2. 语法规则：我们通过建立一组规则，来描述句子的语法结构。规定用“::=”表示“由...组成”（或“定义为...”）。

<句子>::=<主语><谓语>

<主语>::=<代词>|<名词>

<代词> ::=你|我|他

<名词>::= 王民|大学生|工人|英语

<谓语>::=<动词><直接宾语>

<动词>::=是|学习

<直接宾语>::=<代词>|<名词>

3. 由规则推导句子：有了一组规则之后，可以按照一定的方式用它们去推导或产生句子。

推导方法：从一个要识别的符号开始推导，即用相应规则的右部来替代规则的左部，每次仅用一条规则去进行推导。

<句子> => <主语><谓语>

<主语><谓语> => <代词><谓语>

…… ……

这种推导一直进行下去，直到所有带< >的符号都由终结符号替代为止。

推导方法：从一个要识别的符号开始推导，即用相应规则的右部来替代规则的左部，每次仅用一条规则去进行推导。

<句子> => <主语><谓语>

=> <代词><谓语>

=> 我<谓语>

=>我<动词><直接宾语>

=>我是<直接宾语>

=>我是<名词>

=>我是大学生

例：有一英语句子：The big elephant ate the peanut.

<句子>::=<主语><谓语>

<主语>::=<冠词><形容词><名词>

<冠词> ::=the

<形容词>::=big

<名词>::=elephant

<谓语>::=<动词><宾语>

<动词>::=ate

<宾语>::=<冠词><名词>

<名词> ::=peanut

<句子> => <主语><谓语>

=> <冠词><形容词><名词><谓语>

=> the <形容词><名词><谓语>

=> the big <名词> <谓语>

=> the big elephant <谓语>

=> the big elephant <动词><宾语>

=> the big elephant ate <宾语>

=> the big elephant ate <冠词><名词>

=> the big elephant ate the <名词>

=> the big elephant ate the peanut

上述推导可写成<句子> $\overset{+}{\Rightarrow}$ the big elephant ate the peanut

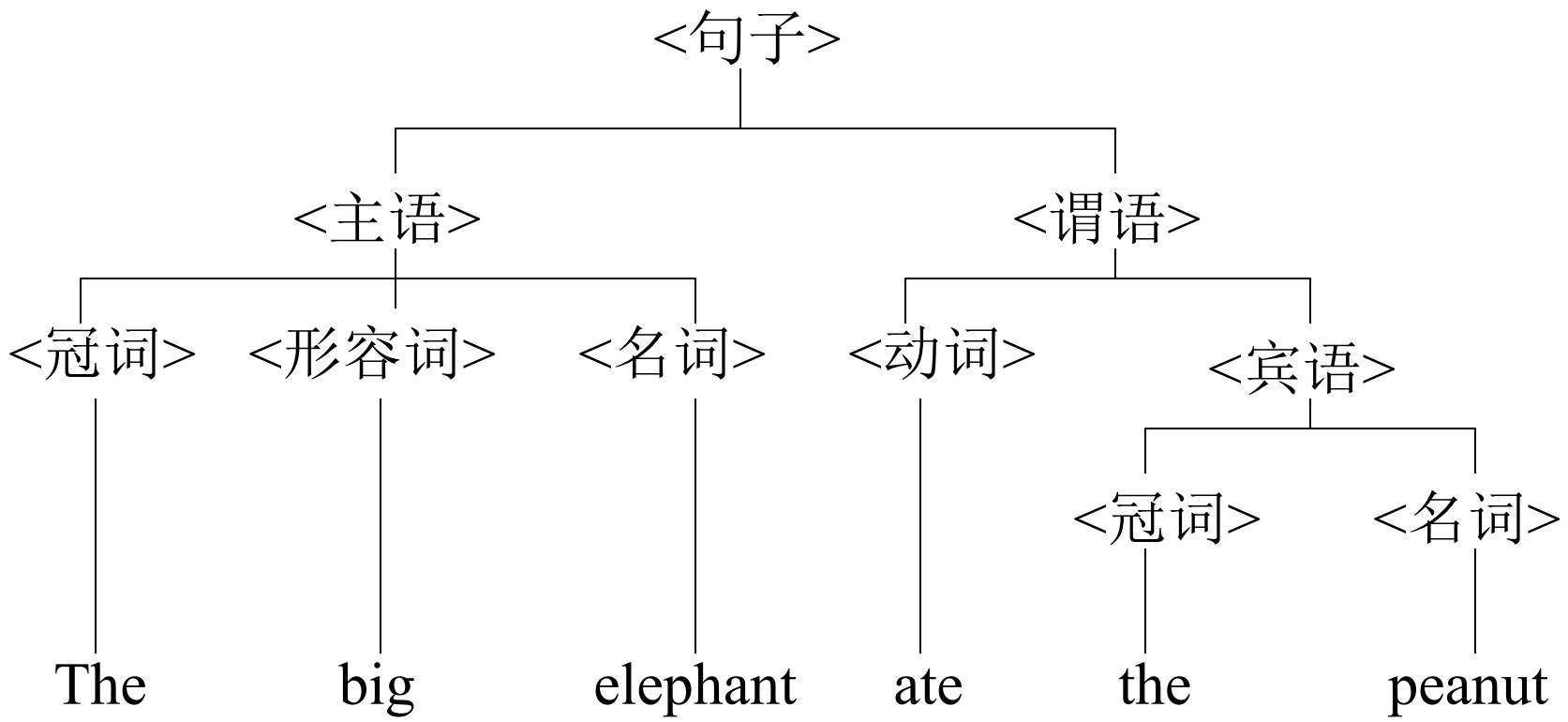
说明：

(1) 有若干语法成分同时存在时，我们总是从最左的语法成分进行推导，这称之为最左推导，类似的有最右推导(还有一般推导）。

(2) 从一组语法规则可推出不同的句子，如以上规则还可推出“大象吃象”、“大花生吃象”、“大花生吃花生”等句子，它们 在语法上都正确，但在语义上都不正确。

所谓文法是在形式上对句子结构的定义与描述，而未涉及语义问题。

4. 语法（推导）树：我们用语法（推导）树来描述一个句子的语法结构。



2.3 文法和语言的形式定义

2.3.1文法的定义

定义1. 文法G=（Vn，Vt，P，Z）

Vn：非终结符号集

Vt：终结符号集

P：产生式或规则的集合

Z：开始符号（识别符号） $Z \in Vn$

规则的定义：

规则是一个有序对(U, x), 通常写为:

$U ::= x$ 或 $U \rightarrow x$ ， $|U| = 1$ $|x| \geq 0$

例：无符号整数的文法：

G[<无符号整数>]=（Vn，Vt，P，Z）

Vn＝{<无符号整数>,<数字串>, <数字>}

Vt ＝{0, 1, 2, 3, …… 9}

P＝{<无符号整数> → <数字串>，

<数字串> → <数字串><数字>，

<数字串> → <数字>，

<数字> →0，

<数字> →1，

……

<数字> →9 }

Z＝<无符号整数>

★ 几点说明：

产生式左边符号构成集合Vn，且 Z ∈Vn

有些产生式具有相同的左部，可以合在一起

例：<无符号整数> → <数字串>
<数字串> → <数字串><数字>|<数字>
<数字> →0|1|2|3|……|9

给定一个 文法，需给出产生式（规则）集合，并指定识别符号

例： G[<无符号整数>]:
<无符号整数> → <数字串>
<数字串> → <数字串><数字>|<数字>
<数字> →0|1|2|3|……|9

2.3.2 推导的形式定义

定义2：文法G：v＝xUy，w＝xuy,

其中x、y $\in V^*$，U$\in$Vn，u $\in V^*$,

若U ::= u$\in$P，则v $\underset{G}{\Rightarrow}$w。

若x＝y＝ ε，有U ::= u，则U $\underset{G}{\Rightarrow}$ u

根据文法和推导定义，可推出终结符号串，所谓通过文法能推出句子来。

例如：G[<无符号整数>]

(1) <无符号整数> → <数字串>

(2) <数字串> → <数字串><数字>

(3) <数字串> → <数字>

(4) <数字> →0

(5) <数字> →1

……

(13) <数字> →9

<无符号整数> ==(1)==> <数字串> ==(2)==> <数字串><数字>

==(3)==> <数字><数字> ==(4)==> 1 <数字>

==(5)==> 1 0

当符号串已没有非终结符号时，推导就必须终止。因为终结符不可能出现在规则左部，所以将在规则左部出现的符号称为非终结符号。

定义3：文法G，$u_0, u_1, u_2, \ldots\ldots, u_n \in V^+$

if $v = u_0 \underset{G}{\Longrightarrow} u_1 \underset{G}{\Longrightarrow} u_2 \underset{G}{\Longrightarrow} \ldots\ldots \underset{G}{\Longrightarrow} u_n = w$

则 $v \overset{+}{\underset{G}{\Longrightarrow}} w$

例：<无符号整数>==> <数字串> ==> <数字串> <数字>

==> <数字><数字> ==>1 <数字>

==>1 0

即 <无符号整数> $\overset{+}{\underset{G}{\Longrightarrow}}$ 10

定义4：文法G，有v，w $\in V^+$

if v $\overset{+}{\underset{G}{\Rightarrow}}$ w , 或v=w，则 v $\overset{*}{\underset{G}{\Rightarrow}}$ w

定义5：规范推导：有xUy ==> xuy，若 y $\in V_t^*$,则此推导为规范的，记为 xUy $=\!|\!\Rightarrow$ xuy

直观意义：规范推导＝最右推导

最右推导：若规则右端符号串中有两个以上的非终结符时，先推右边的。

最左推导：若规则右端符号串中有两个以上的非终结符时，先推左边的。

若有v = u_0 $=\!|\!\Rightarrow$ u_1 $=\!|\!\Rightarrow$ u_2 $=\!|\!\Rightarrow$ …… $=\!|\!\Rightarrow$ u_n = w, 则 v $\overset{+}{=\!|\!\Rightarrow}$ w

2.3.3 语言的形式定义

定义6：文法G[Z]

（1）句型：x是句型 $\Leftrightarrow Z \overset{*}{\Rightarrow} x$，且 $x \in V^*$；

（2）句子：x是句子 $\Leftrightarrow Z \overset{+}{\Rightarrow} x$，且 $x \in V_t^*$；

（3）语言：$L(G[Z]) = \{x \mid x \in V_t^*, Z \overset{+}{\Rightarrow} x\}$；

形式语言理论可以证明以下两点：

（1）G → L(G)；

（2）L(G) → G1，G2，……，Gn；

已知文法，求语言，通过推导；

已知语言，构造文法，无形式化方法，更多是凭经验。

例：{ $ab^na \mid n \geq 1$},构造其文法

$G_1[Z]$：

$Z \rightarrow aBa,$

$B \rightarrow b \mid bB$

$G_2[Z]$：

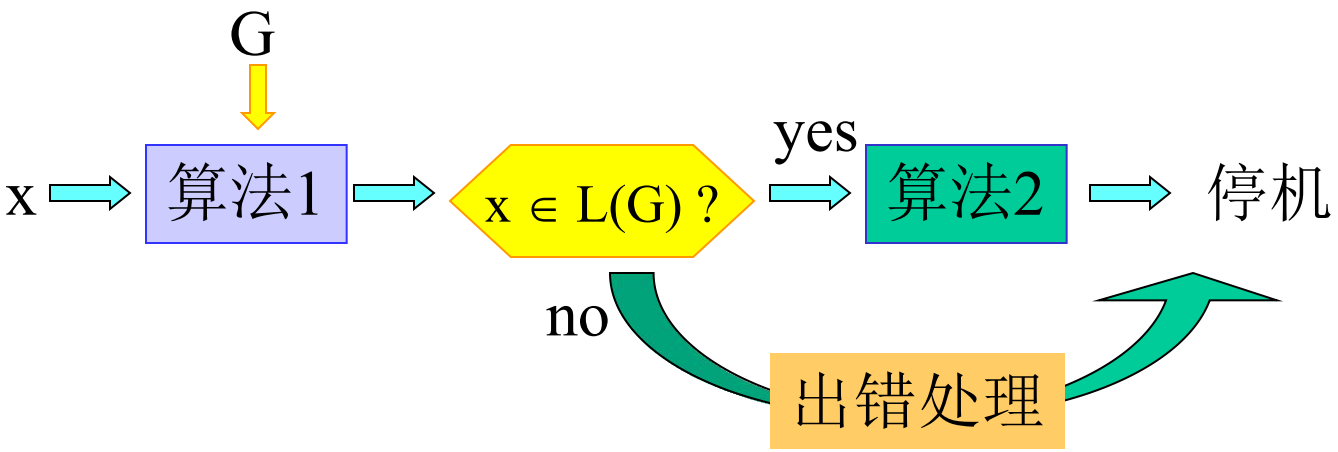
$Z \rightarrow aBa,$

$B \rightarrow b \mid Bb$

定义7. G和G’是两个不同的文法，若 L(G) = L(G’) ，则G和G’为等价文法。

编译感兴趣的问题是:

- 给定句子 x 以及文法 G ， 求x ∈ L(G) ？



2.3.4 递归文法

1.递归规则：规则右部有与左部相同的符号（非终结符）

对于 U::= xUy

若x= ε，　即U::= Uy，　左递归

若y= ε，　即U::= xU，　右递归

若x, y≠ ε，即U::= xUy，自嵌入递归

2.递归文法：文法G，存在U ∈Vn

if　$U \overset{+}{\Longrightarrow} \ldots U \ldots$，则G为递归文法；

if　$U \overset{+}{\Longrightarrow} U \ldots$，　则G为左递归文法；

if　$U \overset{+}{\Longrightarrow} \ldots U$，　则G为右递归文法。

3. 递归文法的优点：可用有穷条规则，定义无穷语言

会造成死循环（后面将详细论述）

4. 左递归文法的缺点：不能用自顶向下的方法来进行语法分析

例：对于前面给出的无符号整数的文法是左递归文法，用13条规则就可以定义出所有的无符号整数。若不用递归文法，那将要用多少条规则呢？

<无符号整数> → <数字串>
<数字串> → <数字串><数字> | <数字>
<数字> →0 | 1 | 2 | 3 | …… | 9

2.3.5 句型的短语、简单短语和句柄

定义8. 给定文法G[Z]，$w = xuy \in V^+$，为该文法的句型，

若 $Z \overset{*}{\Longrightarrow} xUy$, 且$U \overset{+}{\Longrightarrow} u$, 则u是句型w相对于U的短语；

若 $Z \overset{*}{\Longrightarrow} xUy$, 且$U \Longrightarrow u$, 则u是句型w相对于U的简单短语。

其中$U \in V_n$，$u \in V^+$，$x, y \in V^*$

直观理解：短语是前面句型中的某个非终结符所能推出的符号串。

任何句型本身一定是相对于识别符号Z的短语

定义9. 任一句型的最左简单短语称为该句型的句柄。

给定句型找句柄的步骤：

短语 ⟶ 简单短语 ⟶ 句柄

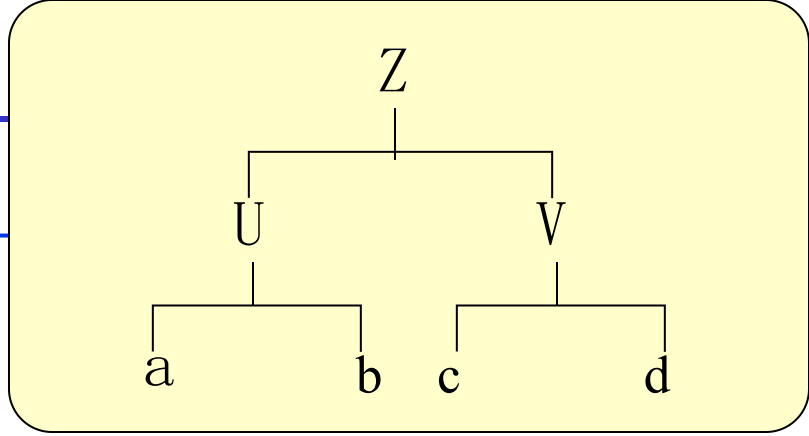
例: 文法G[<无符号整数>]，w = <数字串>1

<无符号整数> => <数字串> => <数字串><数字>
=> <数字串>1

短语：<数字串>1，1 ； 简单短语：1；句柄：1

注意：短语、简单短语是相对于句型而言的，一个句型可能有多个短语、简单短语，而句柄只能有一个。

2.4 语法树与二义性文法



2.4.1 推导与语法（推导）树

（1）语法（推导）树：句子(句型）结构的图示表示法，
它是有向图，由结点和有向边组成。

结点：符号

根结点：　识别符号（非终结符）
中间结点：非终结符
叶结点：　终结符或非终结符

有向边：表示结点间的派生关系

(2) 句型的推导及语法树的生成（自顶向下）

给定G[Z]，句型w：

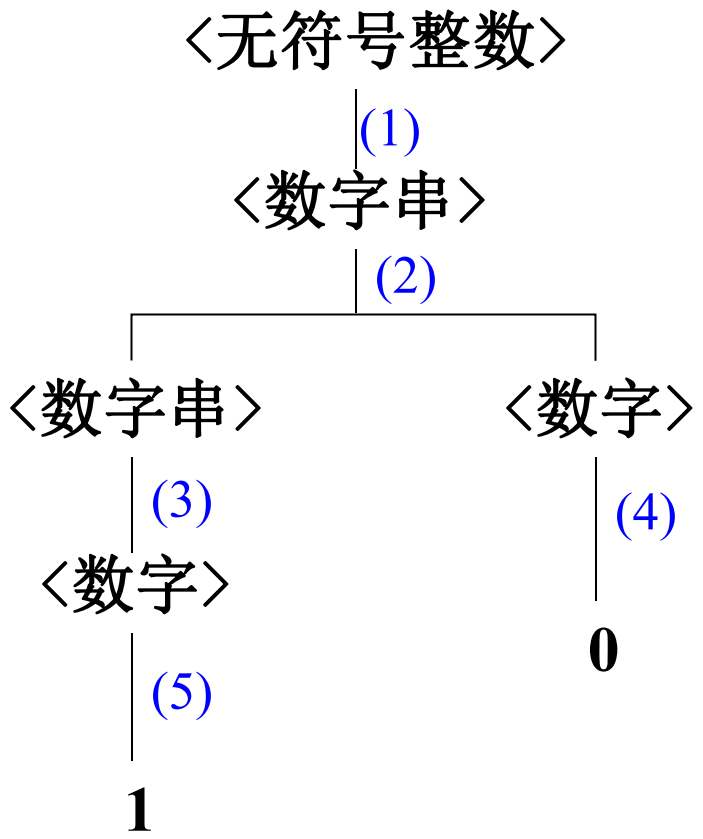
可建立推导序列，$Z \overset{*}{\underset{G}{\Longrightarrow}} w$

可建立语法树，以Z为树根结点，每步推导生成语法树的一枝，最终可生成句型w的语法树。

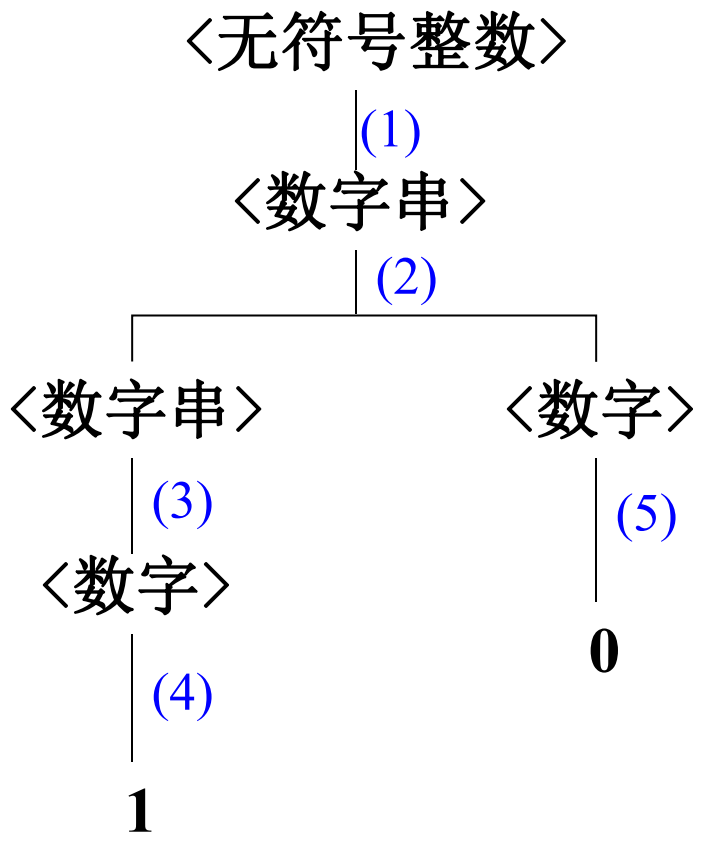
注意一个重要事实：文法所能产生的句子，可以用不同的推导序列（使用产生式顺序不同）将其推导出来。语法树的生长规律不同，但最终生成的语法树形状完全相同。某些文法有此性质，而某些文法不具此性质。

G[<无符号整数>]:
<无符号整数> → <数字串>
<数字串> → <数字串><数字> | <数字>
<数字> →0 | 1 | 2 | 3 | …… | 9

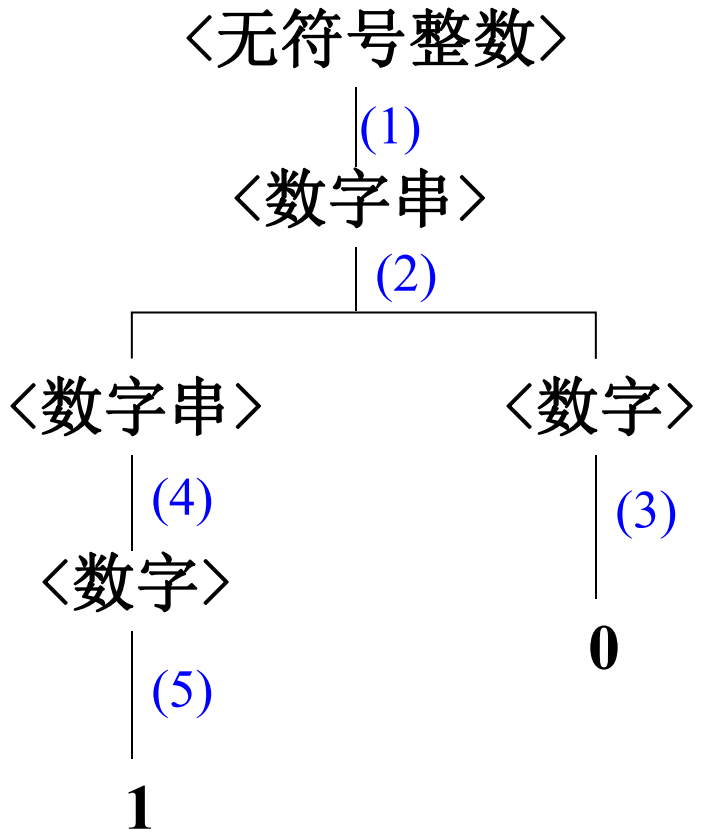
一般推导：



最左推导：



最右推导



(3) 子树与短语

子树：语法树中的某个结点（子树的根）连同它向下派生的部分所组成。

定理 某子树的末端结点按自左向右顺序为句型中的符号串，则该符号串为该句型的相对于该子树根的短语。

只需画出句型的语法树，然后根据子树找短语→简单短语→句柄。

(4) 树与推导

句型推导过程 <==> 该句型语法树的生长过程

1 由推导构造语法树

从识别符号开始，自左向右建立推导序列。

由根结点开始，自上而下建立语法树。

2 由语法树构造推导

自下而上地修剪子树的某些末端结点（短语），直至把整棵树剪掉（留根），每剪一次对应一次归约。

从句型开始，自右向左地逐步进行归约，建立推导序列。

定义12. 对句型中最左简单短语（句柄）进行的归约称为规范归约。

定义13. 通过规范推导或规范归约所得到的句型称为规范句型。

句型<数字><数字>不是文法的规范句型，因为：

<无符号整数>=+><数字串>

=+><数字串><数字>

==><数字><数字>

不是规范推导

2.4.2 文法的二义性

定义14.1 若对于一个文法的某一句子（或句型）存在两棵不同的语法树，则该文法是二义性文法，否则是无二义性文法。

换而言之，无二义性文法的句子只有一棵语法树，尽管推导过程可以不同。

二义性文法举例：

G[E]: E ::= E+E | E*E | (E) | i

V_n={E}

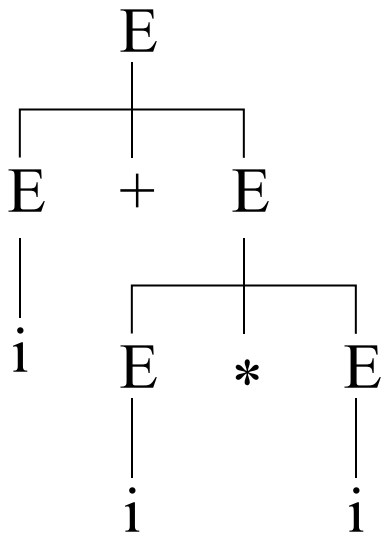
V_t={ +, * , (,) , i }

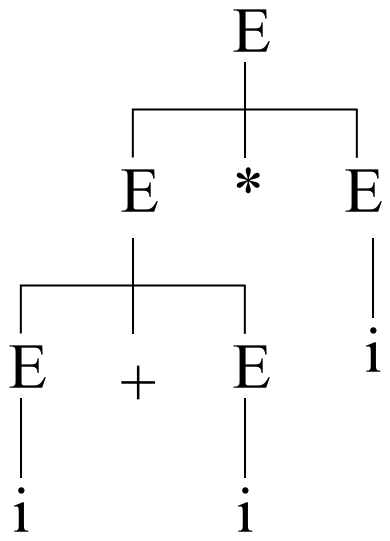
对于句子S＝i＋i ＊i ∈ L(G[E])，存在不同的规范推导：

(1) E=⇒>E+E=⇒>E+E*E =⇒>E+E*i =⇒>E+i*i =⇒> i＋i ＊i

(2) E=⇒> E*E =⇒> E*i =⇒> E+E*i =⇒> E+i*i =⇒> i＋i＊i

这两种不同的推导对应了两棵不同的语法树：



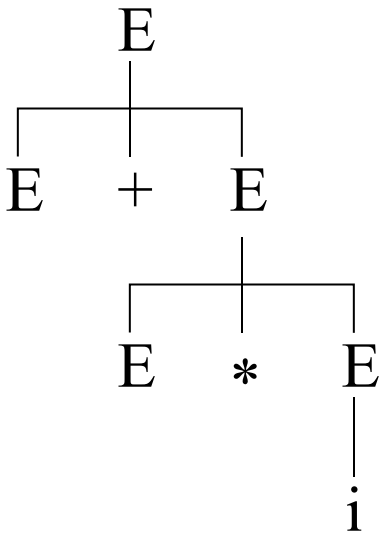


定义14.2 若一个文法的某句子存在两个不同的规范推导，则该文法是二义性的，否则是无二义性的。

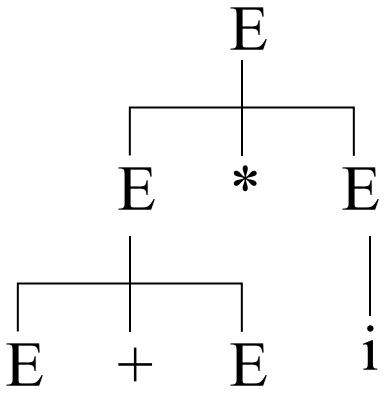
(1) E=|=>E+E=|=>E+E*E =|=>E+E*i =|=>E+i*i =|=> i+i * i

(2) E=|=> E*E =|=> E*i =|=> E+E*i =|=> E+i*i =|=> i+i * i

从自底向上的归约过程来看，上例中规范句型E+E*i 是由i+i * i通过两步规范归约得到的，但对于同一个句型E+E* i，它有两个不同的句柄（对应上述两棵不同的语法树）：i 和 E+E。因此，文法的二义性意味着句型的句柄不唯一。



句柄：i



句柄：E＋E

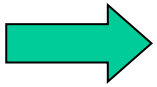
定义14.3 若一个文法的某规范句型的句柄不唯一，则该文法是二义性的，否则是无二义性的。

若文法是二义性的，则在编译时就会产生不确定性，遗憾的是在理论上已经证明：文法的二义性是不可判定的，即不可能构造出一个算法，通过有限步骤来判定任一文法是否有二义性。

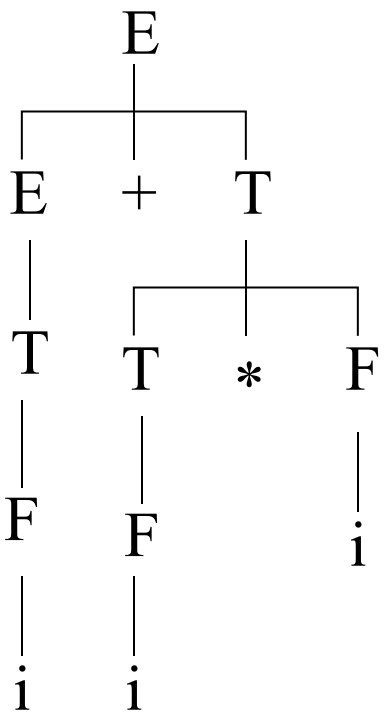
现在的解决办法是：提出一些限制条件，称为无二义性的充分条件，当文法满足这些条件时，就可以判定文法是无二义性的。

例:算术表达式的文法

E::= E+E | E*E | (E) | i



E::= E+T | T
T ::= T*F | F
F ::= (E) | i



句子： i+ i * i

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

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

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

也可以采用另一种解决办法：即不改变二义性文法，而是确定一种编译算法，使该算法满足无二义性充分条件。

例: Pascal 条件语句的文法

<条件语句>::= If <布尔表达式>then<语句> |

If <布尔表达式> then <语句> else <语句>

<语句> ::= <条件语句> | <非条件语句> |.....

If B then If B then stmt else stmt

2.5 句子的分析

任务：给定G[Z]：$S \in V_t^*$，　判定是否有 $S \in L(G[Z])$？

这是词法分析和语法分析所要做的工作，将在第三、第四章中详细介绍。

2.6 有关文法的实用限制

若文法中有如**U::=U**的规则，则这就是有害规则，它会引起二义性。

例如存在U::=U， U::= a | b,则有两棵语法树：

多余规则:（1）在推导文法的所有句子中，始终用不到的规则。即该规则的左部非终结符不出现在任何句型中（不可达符号）

（2）在推导句子的过程中，一旦使用了该规则，将推不出任何终结符号串。即该规则中含有推不出任何终结符号串的非终结符（不活动符号）

例如给定G[Z]，若其中关于U的规则只有如下一条：

U::=xUy

该规则是多余规则。

若还有U::=a，则此规则并非多余

若某文法中无有害规则或多余规则，则称该文法是压缩过的。

例1：G[<Z>]：

<Z> ::= <B> e

<A> ::= <A> e | e

<B> ::= <C> e | <A> f

<C> ::= <C> f

<D> ::= f

例2：　G[S]：

S ::= ccc

S ::= Abccc

A ::= Ab

A ::= aBa

B ::= aBa

B ::= AD

D ::= Db

D ::= b

2.7 文法的其它表示法

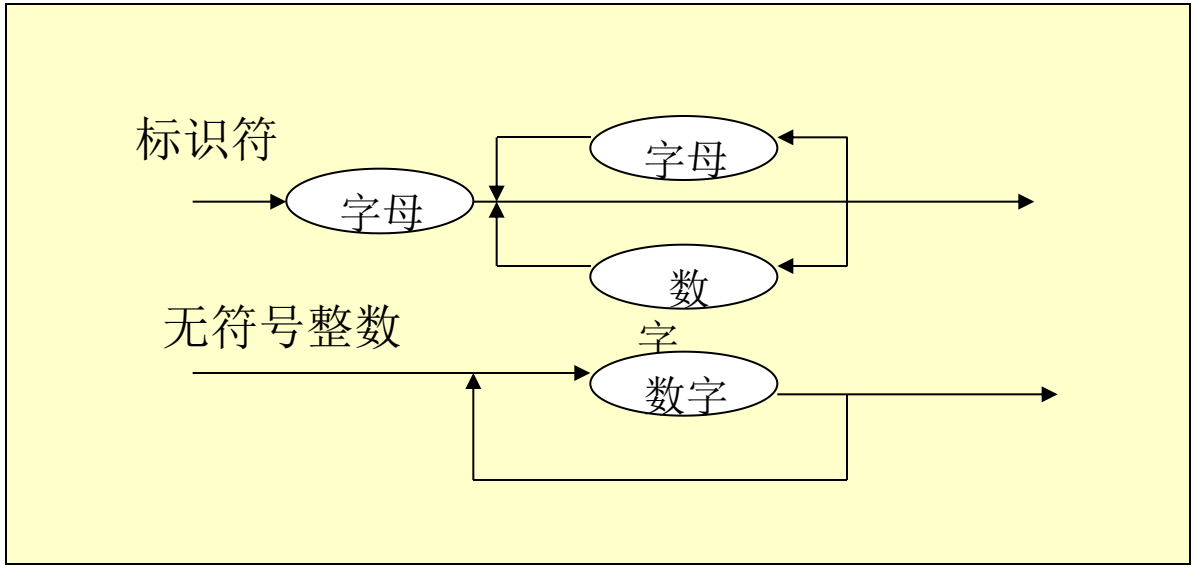
<标识符>::＝字母{字母|数字}

<无符号整数>::=数字{数字}

1、扩充的BNF表示

- BNF的元符号： < , >, ::= , |
- 扩充的BNF的元符号： < , >, ::= , | , { , } , [,] , (,)

2、语法图



2.8 文法和语言分类

形式语言：用文法和自动机所描述的没有语义的语言。

文法定义：乔姆斯基将所有文法都定义为一个四元组：

G=（Vn，Vt，P，Z）

Vn：非终结符号集

Vt：终结符号集

P：产生式或规则的集合

Z：开始符号（识别符号） $Z \in Vn$

语言定义： $L(G[Z]) = \{x \mid x \in Vt^*, Z \overset{+}{\Longrightarrow} x\}$

文法和语言分类：0型、1型、2型、3型

这几类文法的差别在于对产生式（语法规则）施加不同的限制。

0型：　　P：　$u ::= v$

其中　$u \in V^+$，$v \in V^*$　　$V = V_n \cup V_t$

0型文法称为短语结构文法。规则的左部和右部都可以是符号串，一个短语可以产生另一个短语。

0型语言：L0　这种语言可以用图灵机(Turing)接受。

1型：　　P：　xUy ::= xuy

其中 U∈Vn，

x、y、u∈V*

称为上下文敏感或上下文有关。也即只有在x、y这样的上下文中才能把U改写为u

1型语言：L1　这种语言可以由一种线性界限自动机接受。

2型： P： U ::= u

其中 U∈Vn，

u∈V*

称为上下文无关文法。也即把U改写为u时，不必考虑上下文。

(1型文法的规则中x，y均为 ε 时即为2型文法）

注意：2型文法与BNF表示相等价。

2型语言：L2 这种语言可以由下推自动机接受。

3型文法：

（左线性）

P： U ::= t

或 U ::= Wt

其中 U、W∈Vn

t∈Vt

（右线性）

P： U ::= t

或 U ::= tW

其中 U、W∈Vn

t∈Vt

3型文法称为正则文法。它是对2型文法进行进一步限制。

3型语言：L3　又称正则语言、正则集合

这种语言可以由有穷自动机接受。

- 根据上述讨论，L0 ⊃ L1 ⊃ L2 ⊃ L3
- 0型文法可以产生L0、L1、L2、L3，
- 但2型文法只能产生L2，L3不能产生L0，L1
- 3型文法只能产生L3

第三章　词法分析

- 词法分析程序的功能及实现方案
- 单词的种类及词法分析程序的输出形式
- 正则文法和状态图
- 词法分析程序的设计与实现
- 正则表达式与有穷自动机
- 词法分析程序的自动生成器

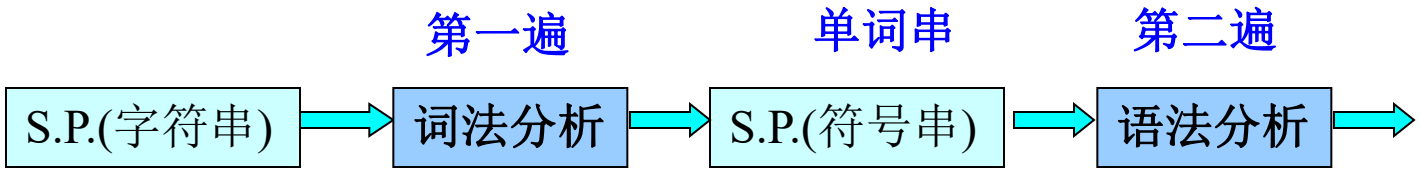
3.1 词法分析程序的功能及实现方案

词法分析程序的功能

- 词法分析：根据词法规则识别及组合单词，进行词法检查。
- 对数字常数完成数字字符串到二进制数值的转换。
- 删去空格字符和注释。

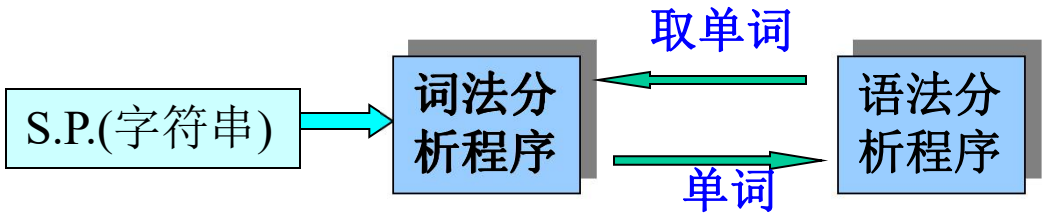
实现方案：基本上有两种

1.词法分析单独作为一遍



优点：结构清晰、
各遍功能单一
缺点：效率低

2.词法分析程序作为单独的子程序



优点：效率高

3.2 单词的种类及词法分析程序的输出形式

单词的种类

1. 保留字：begin、end、for、do...
2. 标识符：由用户定义，表示各种名字
3. 常　数：无符号数、布尔常数、字符串常数等
4. 分界符：+、-、*、/、...

词法分析程序的输出形式-----单词的内部形式

单词类别	单词值

几种常用的单词内部形式：

1、按单词种类分类

2、保留字和分界符采用一符一类

3、标识符和常数的单词值又为指示字（指针值）

1、按单词种类分类

单词名称	类别编码	单词值
标识符	1	内部字符串
无符号常数(整)	2	整数值
无符号浮点数	3	数值
布尔常数	4	0 或 1
字符串常数	5	内部字符串
保留字	6	保留字或内部编码
分界符	7	分界符或内部编码

2、保留字和分界符采用一符一类

单词名称	类别编码	单词值
标识符	1	内部字符串
无符号常数(整)	2	整数值
无符号浮点数	3	数值
布尔常数	4	0 或 1
字符串常数	5	内部字符串
BEGIN	6	-
END	7	-
FOR	8	-
DO	9	-
……	…….	…..
:	20	-
+	21	-
*	22	-
,	23	-
(	…….	--

3.3 正则文法和状态图

- 状态图的画法（根据文法画出状态图）

例如：正则文法

Z::= U0 |V1

U ::=Z1 |1

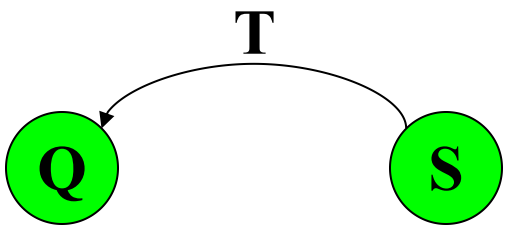
V ::=Z0 | 0

左线性文法。该文法所定义的语言为：

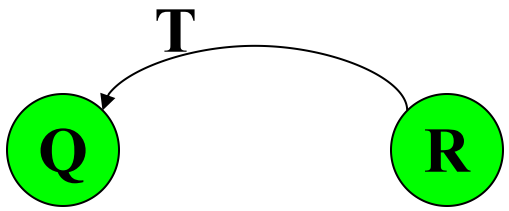
$L(G[Z]) = \{ B^n \mid n>0 \}$，其中 B= {01,10}

左线性文法的状态图的画法：

1. 令G的每个非终结符都是一个状态；
2. 设一个开始状态S；
3. 若Q::=T, Q ∈Vn,T ∈Vt, 则：



4. 若Q::=RT, Q、R∈Vn,T ∈Vt, 则：



5. 按自动机方法，可加上开始状态和终止状态标志。

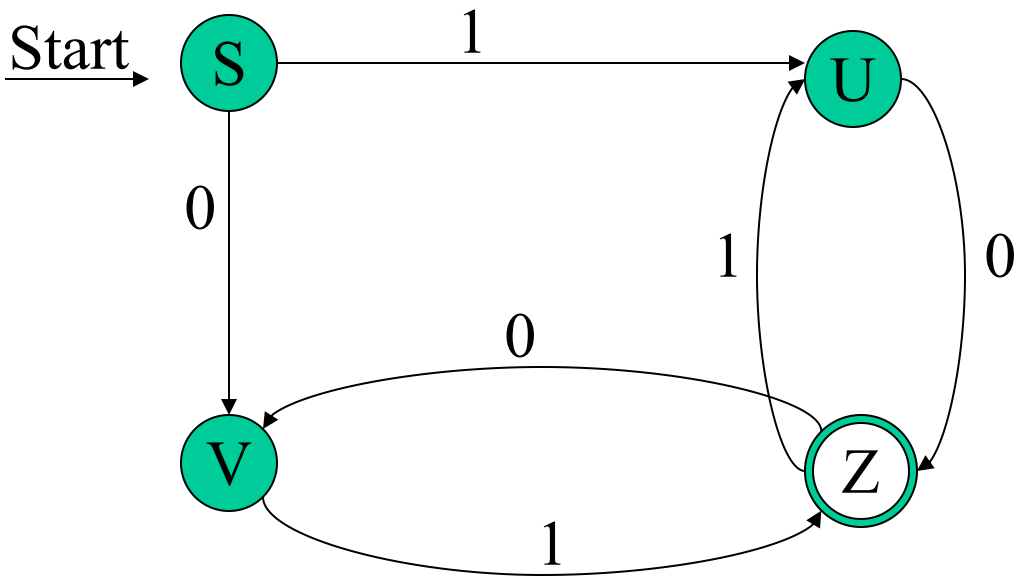
例如：正则文法

Z::= U0 |V1

U ::=Z1 |1

V ::=Z0 | 0

其状态图为：



- 识别算法

利用状态图可按如下步骤分析和识别字符串x：

1、置初始状态为当前状态，从x的最左字符开始，重复步骤2，直到x右端为止。

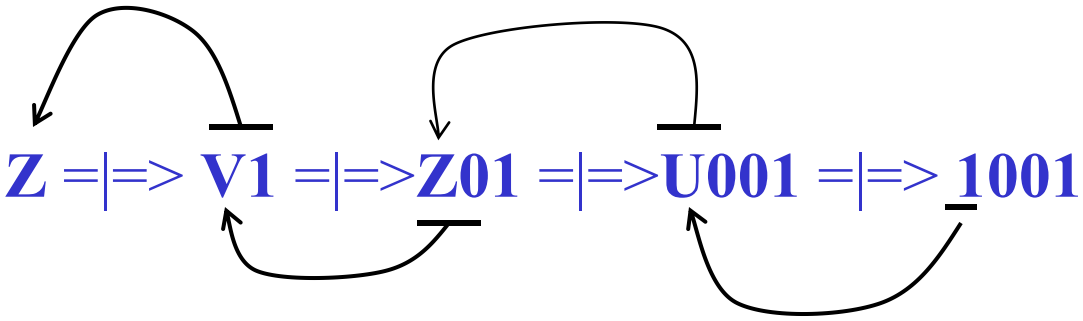
2、扫描x的下一个字符，在当前状态所射出的弧中找出标记有该字符的弧，并沿此弧过渡到下一个状态；如果找不到标有该字符的弧，那么x不是句子，过程到此结束；如果扫描的是x的最右端字符，并从当前状态出发沿着标有该字符的弧过渡到下一个状态为终止状态Z，则x是句子。

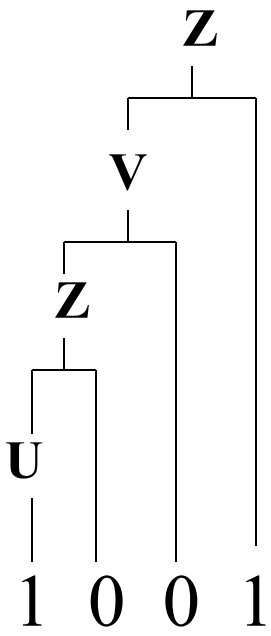
例：x=0110 和1011

- 问题：

1、上述分析过程是属于自底向上分析？还是自顶向下分析？

2、怎样确定句柄？





3.4 词法分析程序的设计与实现

词法规则 → 状态图 → 词法分析程序

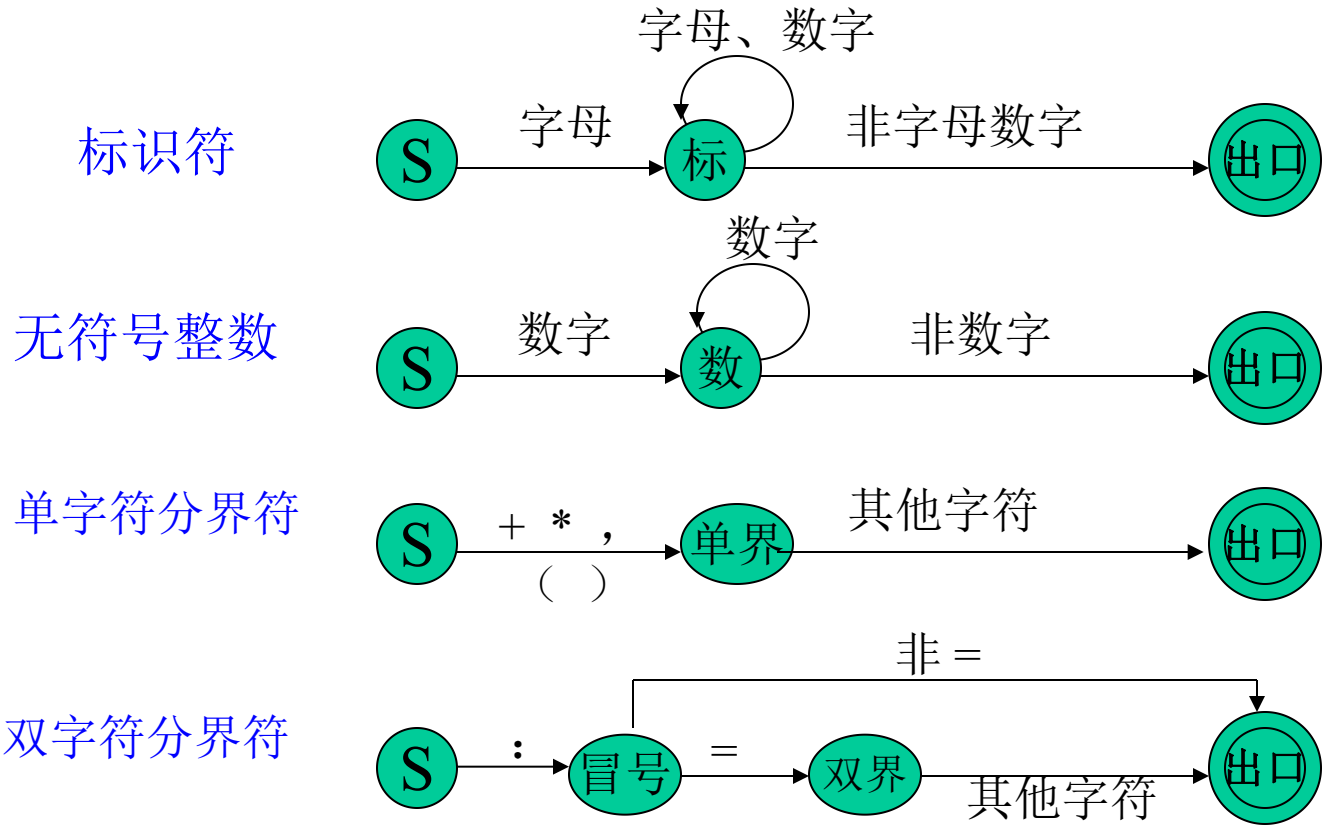
3.4.1 文法及其状态图

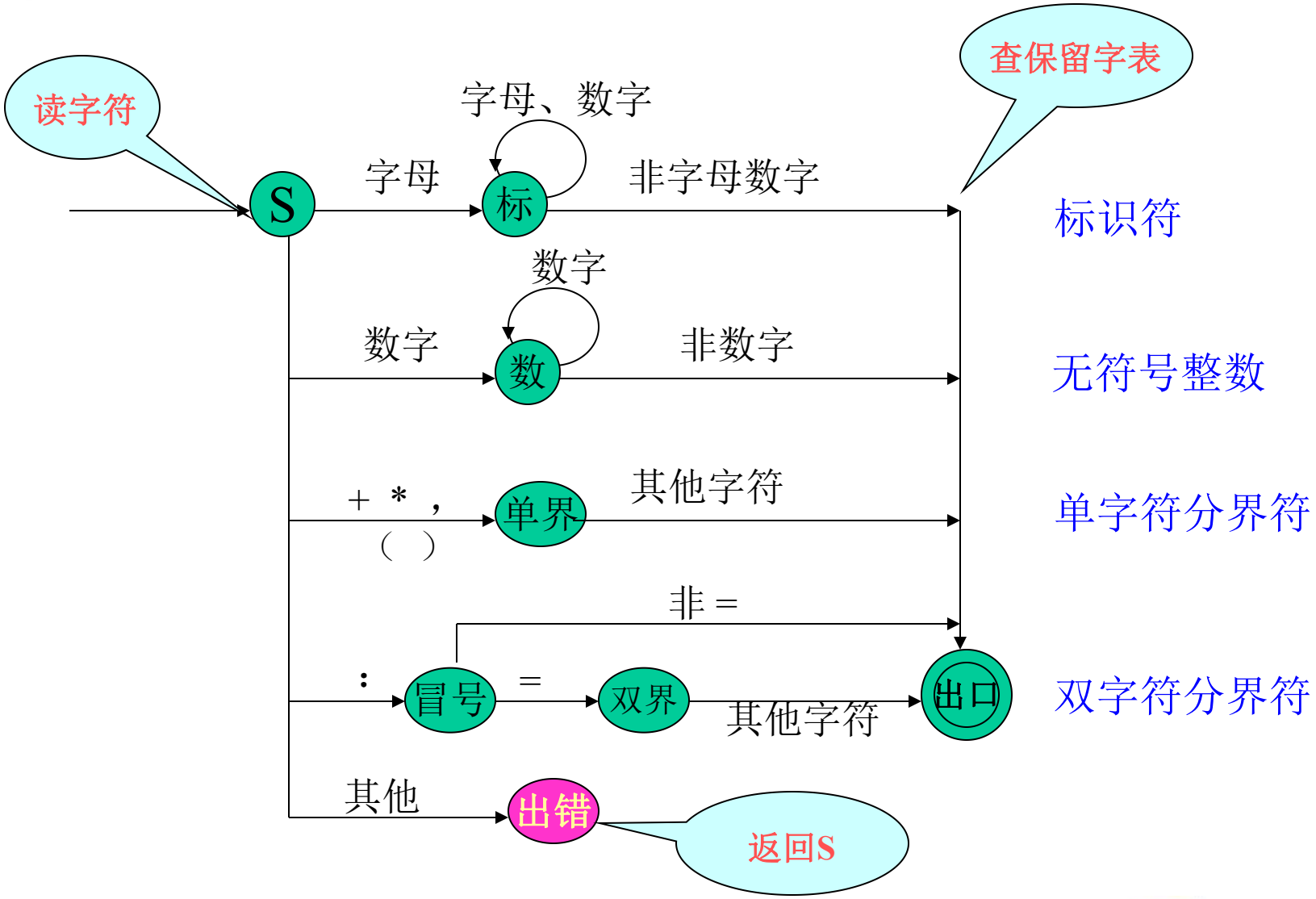
语言的单词符号

标识符
保留字(标识符的子集)
无符号整数
单分界符　+　*　：，（ ）
双分界符　　:=

两点说明：1. 空格的作用　　2. 实数的表示

- 文法：1. <标识符>::= 字母 | <标识符>字母 | <标识符>数字
  2. <无符号整数>::=数字 | <无符号整数>数字
  3. <单字符分界符>::= : | + | * | , | (|)
  4. <双字符分界符>::= <冒号>=
  5. <冒号>::= ：




读字符
S
字母
字母、数字
标
非字母数字
查保留字表
标识符
数字
数字
数
非数字
无符号整数
+ * ,
( )
单界
其他字符
单字符分界符
非 =
:
冒号
=
双界
出口
其他字符
双字符分界符
其他
出错
返回S

3.4.2 状态图的实现——构造词法分析程序

1. 单词及内部表示

2. 词法分析程序需要引用的公共（全局）变量和过程

3. 词法分析程序算法

1.单词及内部表示:　保留字和分界符采用一符一类

单词名称	类别编码	记忆符	单词值
BEGIN	1	BEGINSY	-
END	2	ENDSY	-
FOR	3	FORSY	-
DO	4	DOSY	-
IF	5	IFSY	-
THEN	6	THENSY	-
ELSE	7	ELSESY	-
标识符	8	IDSY	内部字符串
常数(整)	9	INTSY	整数值
:	10	COLONSY	-
+	11	PLUSSY	-
*	12	STARSY	-
,	13	COMSY	-
(	14	LPARSY	-
)	15	RPARSY	-
:=	16	ASSIGNSY	-

2.词法分析程序需要引用的公共（全局）变量和过程

名称	类别	功能
CHAR	字符变量	存放当前读入的字符
TOKEN	字符数组	存放单词字符串
GETCHAR	读字符过程	读字符到CHAR,移动指针
GETNBC	过程	反复调用GETCHAR, 直至CHAR进入一个非空白字符
CAT	过程	CHAR与TOKEN连接
ISLETTER 和 ISDIGIT	布尔函数	判断
UNGETCH	过程	读字符指针后退一个字符
RESERVE	布尔函数	判断TOKEN中的字符串 是保留字, 还是标识符
ATOI	函数	字符串到数字的转换
ERROR	过程	出错处理

3、词法分析程序算法

```
START:  TOKEN := ‘ ‘;     /*置TOKEN为空串*/
        GETCHAR;  GETNBC;
  CASE  CHAR OF
 ‘A’..’Z’: BEGIN
             WHILE  ISLETTER OR ISDIGET DO
                    BEGIN CAT; GETCHAR  END;
              UNGETCH;
              C:= RESERVE;
              IF C=0 THEN RETURN(‘IDSY’: TOKEN)
              ELSE RETURN (C,-)      /*C为保留字编码*/
            END;
 ‘0’..’9’:  BEGIN
             WHILE   DIGIT   DO
                     BEGIN CAT; GETCHAR  END;
               UNGETCH;
               RETURN (‘INTSY’,ATOI)
            END;
‘+’ :   RETURN(‘PLUSSY’,-) ;
```

```
‘*’ :   RETURN(‘STARSY’,-) ;
‘,’ :   RETURN(‘COMMASY’,-) ;
‘(’ :   RETURN(‘LPARSY’,-) ;
‘)’ :   RETURN(‘RPARSY’,-) ;
‘:’ :   BEGIN
            GETCHAR;
            if  CHAR=‘=‘ THEN  RETURN(‘ASSIGNSY’,-) ;
            UNGETCH;
            RETURN(‘COLONSY’,-) ;
        END
END OF CASE;
ERROR;
GOTO START;
```

3.5 正则表达式与有穷自动机

3.5.1 正则表达式和正则集合的递归定义

有字母表∑，定义在∑ 上的正则表达式和正则集合递归定义如下：

1. ε和ϕ都是∑ 上的正则表达式，其正则集合分别为：{ε}和ϕ；
2. 任何a∈ ∑ ， a是∑ 上的正则表达式，其正则集合为：{a}；
3. 假定U和V是 ∑ 上的正则表达式，其正则集合分别记为L(U)和L(V)，那么U|V，U•V和U*也都是∑ 上的正则表达式，其正则集合分别为L(U) ∪L(V)、 L(U) • L(V)和L(U)*；
4. 任何∑ 上的正则表达式和正则集合均由1、2和3产生。

正则表达式中的运算符：

| -----或（选择）　　• ----连接

* 或｛｝---重复　　（） ----括号

运算符的优先级：

先*， 后•， 最后 |

• 在正则表达式中可以省略.

正则表达式相等⇔这两个正则表达式表示的语言相等

如：b{ab} = {ba}b

{a|b} = {{a} {b}} = (a*b*)*

例：设 $\Sigma = \{ a, b \}$，下面是定义在Σ上的正则表达式和正则集合

正则表达式	正则集合
ba*	
a(a|b)*	
(a|b)*(aa|bb)(a|b)*	

正则表达式的性质：

设e1, e2和e3均是某字母表上的正则表达式, 则有:

单位正则表达式: ε　　εe = eε = e

交换律:　e1 | e2 = e2 | e1

结合律:　e1|(e2|e3) = (e1|e2)|e3

　　e1(e2e3) = (e1e2)e3

分配律:　e1(e2|e3) = e1e2|e1e3

　　(e1|e2)e3 = e1e3|e2e3

此外:　　r* = (r|ε)*　　r** =r*

　　(r|s)* = (r*s*)*

正则表达式与3型文法等价

例如：

正则表达式： ba*　　a(a|b)*

3型文法： Z ::= Za|b　　Z::=Za|Zb|a

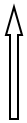
例：

3型文法　　正则表达式

S::= aS|aB

B::= bC

C::= aC|a



aS|aba*a → a*aba*a

ba*a

a*a

3.5.2 确定的有穷自动机（DFA）— 状态图的形式化
(Deterministic Finite Automata)

一个确定的有穷自动机（DFA）M是一个五元式:

$$M=(S, \Sigma, \delta, s_0, Z)$$

其中:

1. S —有穷状态集
2. Σ —输入字母表
3. δ —映射函数(也称状态转换函数)

   $S \times \Sigma \rightarrow S$

   $\delta(s,a)=s'$, $s, s' \in S$, $a \in \Sigma$
4. s_0—初始状态　　$s_0 \in S$
5. Z—终止状态集　　$Z \subseteq S$

例如： M:（{0, 1, 2, 3}, {a, b}, δ , 0, {3}）

δ（0, a）=1　　δ（0, b）=2

δ（1, a）=3　　δ（1, b）=2

δ（2, a）=1　　δ（2, b）=3

δ（3, a）=3　　δ（3, b）=3

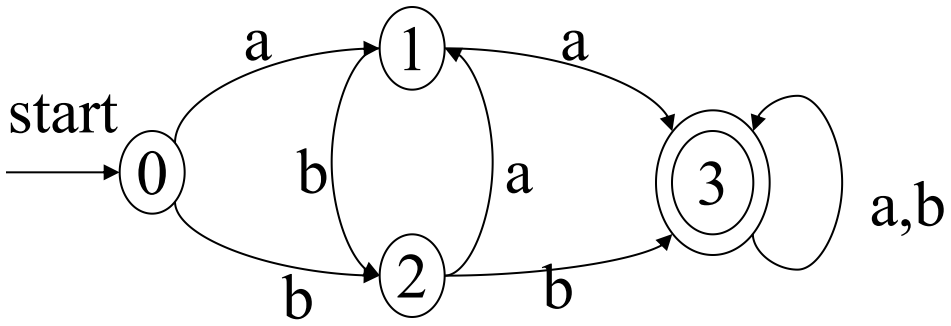
状态转换函数δ可用一矩阵来表示：

输入字符 / 状态	a	b
0	1	2
1	3	2
2	1	3
3	3	3

所谓确定的状态机，其确定性都表现在状态转换函数是单值函数！

DFA也可以用一状态转换图表示：

DFA的状态图表示：



DFA M所接受的符号串：

令 $\alpha = a_1a_2\cdots a_n$，$\alpha \in \Sigma$，若 $\delta^*(\delta(\cdots\delta(\delta(s_0,a_1),a_2)\cdots),a_{n-1}),a_n) = s_n$，且 $s_n \in Z$，则可以写成 $\delta(s_0,\alpha) = s_n$，我们称 α 可为M所接受。

$$\delta(s_0,a_1) = s_1$$

$$\delta(s_1,a_2) = s_2$$

$$\cdots\cdots$$

$$\delta(s_{n-2},a_{n-1}) = s_{n-1}$$

$$\delta(s_{n-1},a_n) = s_n$$

换言之：若存在一条初始状态到某一终止状态的路径，且这条路径上能有弧的标记符号连接成符号串 α，则称 α 为DFA M（接受）识别。

DFA M所接受的语言为：$L(M)=\{\alpha \mid \delta(s_0,\alpha) = s_n,\ s_n \in Z\}$

3.5.3 不确定的有穷自动机(NFA) (Nondeterministic Finite Automata)

若δ是一个多值函数，且输入可允许为ε，则有穷自动机是不确定的，即在某个状态下，对于某个输入字符存在多个后继状态。

从同一状态出发，有以同一字符标记的多条边，或者有以ε标记的特殊边的自动机。

NFA的形式定义为:

一个非确定的有穷自动机NFA M'是一个五元式:

NFA M'=(S, Σ ∪{ ε }, δ , s_0, Z)

其中 S—有穷状态集

Σ ∪{ ε }—输入符号加上 ε ,

即自动机的每个结点所射出的弧可以是Σ中的一个字符或是 ε

s_0—初态 $s_0 \in S$

Z—终态集 $Z \subseteq S$

δ —转换函数 $S \times \Sigma \cup \{ \varepsilon \} \rightarrow 2^S$

(2^S --S的幂集—S的子集构成的集合）

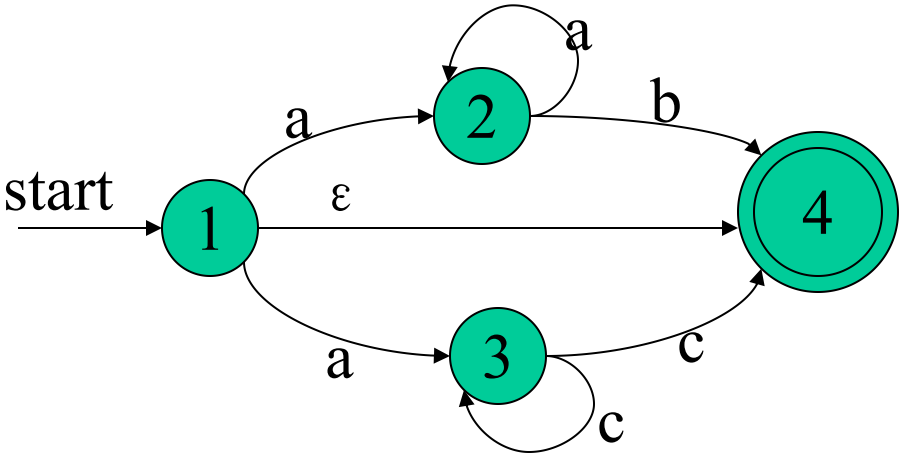
NFA M'所接受的语言为:

$$L(M')=\{\alpha \mid \delta(S_0,\alpha)=S' \quad S'\cap Z\neq\Phi\}$$

例： NFA M'=({1,2,3,4},{a,b,c} ∪{ ε }, δ ,1,{4})

状态 \ 符号	ε	a	b	c
1	{4}	{2，3}	Φ	Φ
2	Φ	{2}	{4}	Φ
3	Φ	Φ	Φ	{3，4}
4	Φ	Φ	Φ	Φ

上例题相应的状态图为：



M’所接受的语言（用正则表达式）　R=aa*b|ac*c| ε

3.5.4 NFA的确定化

已证明：**不确定的有穷自动机与确定的有穷自动机从功能上来说是等价的**，也就是说能够从：

$$\text{NFA}\ \ M' \xrightarrow{\text{构造成一个}} \text{DFA}\ \ \ M$$

使得　$L(M)=L(M')$

为了使得NFA确定化，首先给出两个定义：

定义1、集合I的 ε -闭包：

令I是一个状态集的子集，定义 ε -closure（I）为：
1）若s∈I，则s∈ ε -closure（I）；
2）若s∈I，则从s出发经过任意条 ε 弧能够到达的任何状态都属于 ε -closure（I）。
状态集 ε -closure（I）称为I的 ε -闭包。

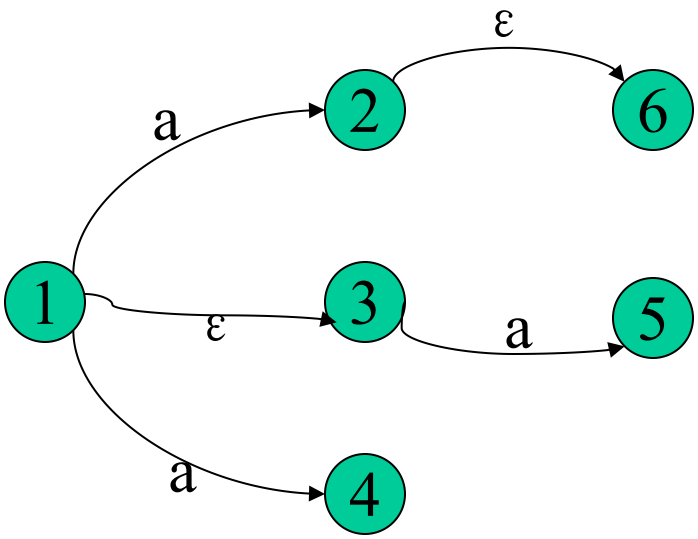
可以通过一例子来说明状态子集的 ε –闭包的构造方法

例：

如图所示的状态图：

令I={1}，

求 ε -closure（I）=?



根据定义：

ε -closure（I）= {1，3}

定义2：令I是NFA M'的状态集的一个子集，$a \in \Sigma$

定义：$I_a = \varepsilon\text{-closure}(J)$

其中$J = \bigcup_{S \in I} \delta(s,a)$

-- J是从状态子集I中的每个状态出发,经过标记为a的弧而达到的状态集合。

-- I_a是状态子集, 其元素为J中的状态, 加上从J中每一个状态出发通过 ε 弧到达的状态。

同样可以通过一例子来说明上述定义, 仍采用前面给定的状态图为例

例：令　I={1}

I_a = ε -closure(J)

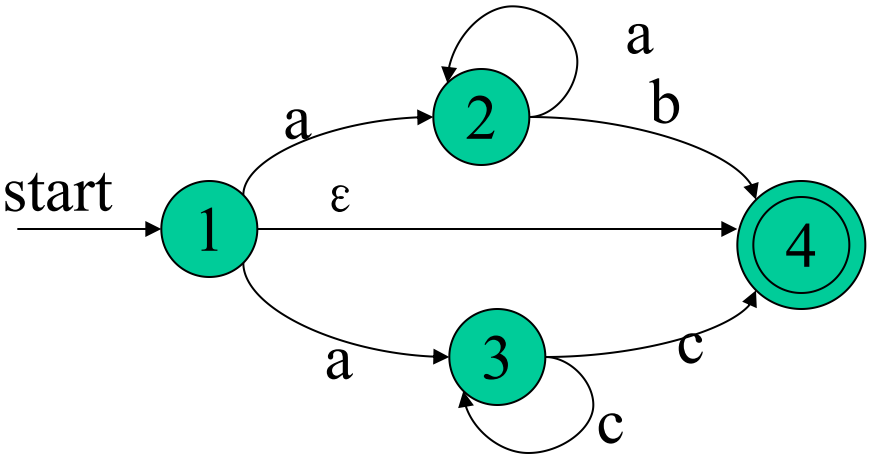
= ε -closure(δ （1, a）)

= ε -closure({2, 4})

={2, 4, 6}

根据定义1，2，可以将上述的M'确定化（即可构造出状态转换矩阵）

例：有NFA M’



I= ε -closure({1})={1,4}

I_a= ε -closure(δ (1,a)∪ δ (4,a))

= ε -closure({2,3}∪ ϕ)

= ε -closure ({2,3})

={2,3}

I_b= ε -closure(δ (1,b)∪ δ (4,b))

= ε -closure(ϕ)

= ϕ

I_c= ε -closure(δ (1,c)∪ δ (4,c))

= ϕ

I={2,3}, I_a={2},I_b={4},I_c={3,4}…

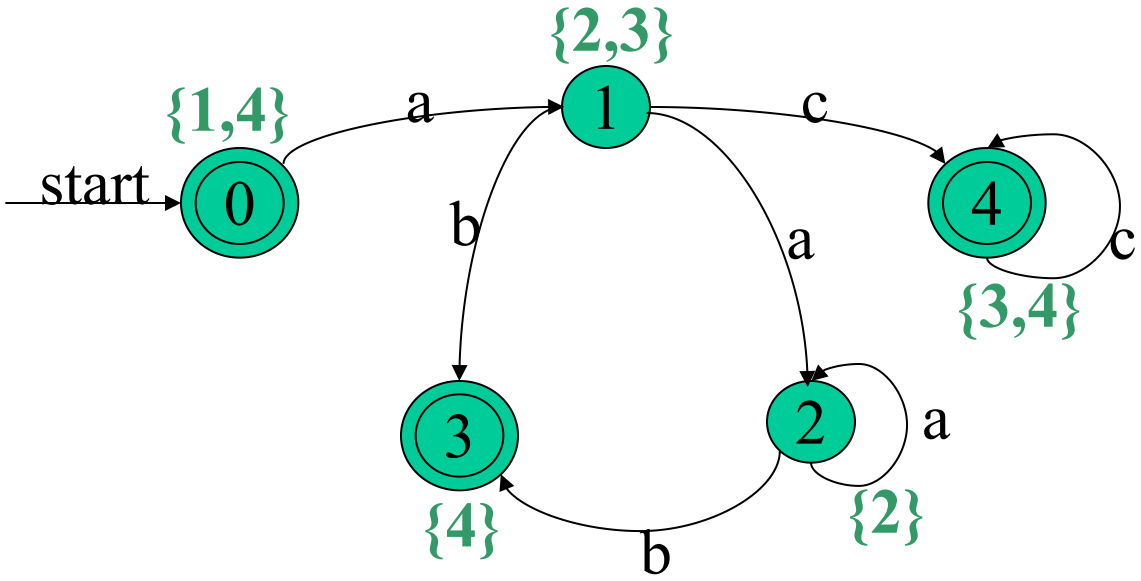
I	I_a	I_b	I_c
{1,4}	{2,3}	ϕ	ϕ
{2,3}	{2}	{4}	{3,4}
{2}	{2}	{4}	ϕ
{4}	ϕ	ϕ	ϕ
{3,4}	ϕ	ϕ	{3,4}

将求得的状态转换矩阵重新编号

DFA M状态转换矩阵：

状态 \ 符号	a	b	c
0	1	–	–
1	2	3	4
2	2	3	–
3	–	–	–
4	–	–	4

DFA M的状态图：



☆ 注意：包含原初始状态1的状态子集为DFA M的初态
包含原终止状态4的状态子集为DFA M的终态。

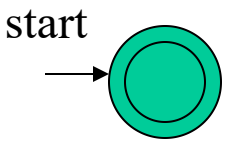
3.5.5 正则表达式与DFA的等价性

定理：在Σ上的一个子集V（$V \subseteq \Sigma^*$）是正则集合，当且仅当存在一个DFA M，使得V=L(M)

V是正则集合，
R是与其相对应的正则表达式 $\Leftrightarrow$ DFA M
V=L(R) L(M)=L(R)

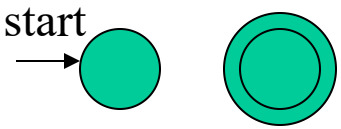
所以 正则表达式R $\Rightarrow$ NFA M' $\Rightarrow$ DFA M
L(R) = L(M') = L(M)

证明: 根据定义。



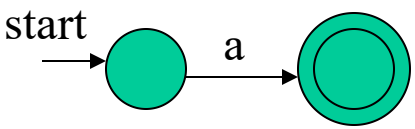
正则表达式 ε

正则集合 {ε}



ϕ

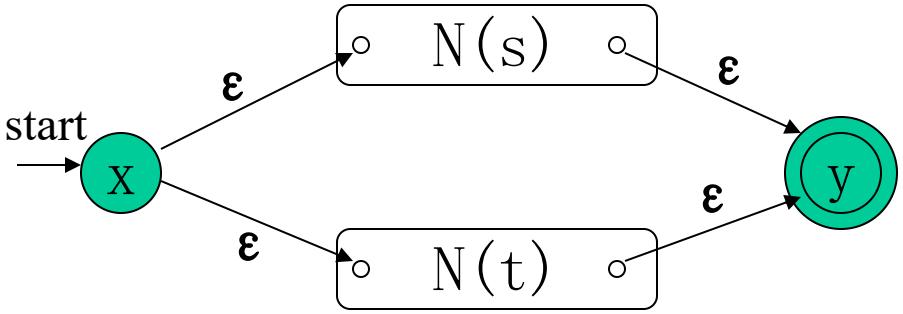
ϕ



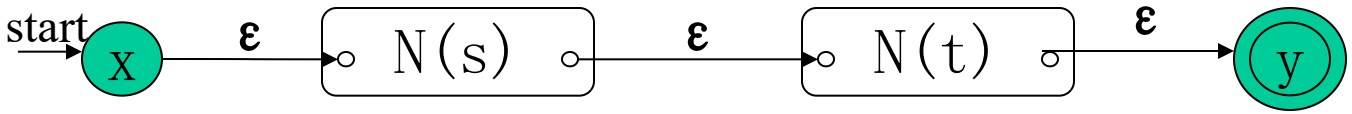
a

{a}

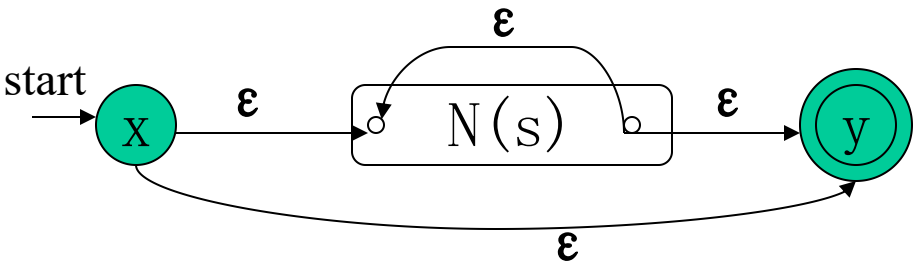
R=s|t NFA(R)：



R=st NFA(R)：



R=s* NFA(R)：



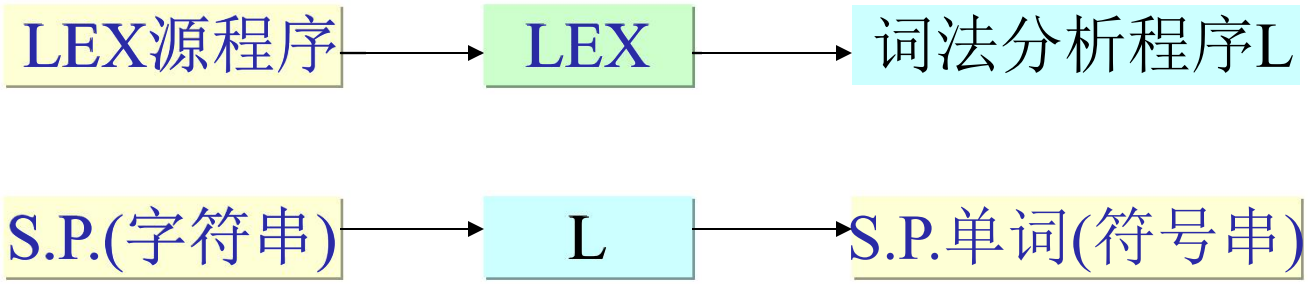
3.6 词法分析程序的自动生成器—LEX（LEXICAL）

LEX的原理：

正则表达式与DFA的等价性

根据给定的正则表达式自动生成相应的词法分析程序。

LEX的功能：



3.6.1 LEX源程序

一个LEX源程序主要由三个部分组成:

1. 辅助定义式
2. 识别规则
3. 用户子程序

各部分之间用%%隔开

辅助定义式是如下形式的LEX语句：

$$D_1 \longrightarrow R_1$$
$$D_2 \longrightarrow R_2$$
$$\vdots$$
$$\vdots$$
$$D_n \longrightarrow R_n$$

其中：

R_1, R_2，……，R_n 为正则表达式。

D_1, D_2，……，D_n 为正则表达式名字，称简名。

例：标识符：　letter→A|B|……|Z
digit →0|1|……|9
iden → letter(letter|digit)*

带符号整数：　integer→digit (digit)*
sign → +| - | ε
sign_integer → sign integer

识别规则：是一串如下形式的LEX语句：

P_1 $\{A_1\}$
P_2 $\{A_2\}$
：
：
P_m $\{A_m\}$

P_i：定义在 $\Sigma \cup \{D_1, D_2, \cdots D_n\}$ 上的正则表达式，也称词形。

$\{A_i\}$：A_i为语句序列，它指出，在识别出词形为P_i的单词以后，词法分析器所应作的动作。

其基本动作是返回单词的类别编码和单词值。

下面是识别某语言单词符号的LEX源程序：

例：LEX 源程序

```
AUXILIARY    DEFINITIONS   /*辅助定义*/
    letter → A|B|…|Z
    digit  →  0|1|…|9
%%
RECOGNITION RULES          /*识别规则*/
  1.BEGIN                  {RETURN(1,—) }
  2.END                    {RETURN(2,—) }
  3.FOR                    {RETURN(3,—) }
```

4.DO	{RETURN(4,—) }
5.IF	{RETURN(5,—) }
6.THEN	{RETURN(6,—) }
7.ELSE	{RETURN(7,—) }
8.letter(letter |digit)*	{RETURN(8,TOKEN) }
9.digit(digit)*	{RETURN(9,DTB }
10. :	{RETURN(10,—) }
11. +	{RETURN(11,—) }
12. “*”	{RETURN(12,—) }

13. , {RETURN(13,—) }

14. “（” {RETURN(14,—) }

15. “）” {RETURN(15,—) }

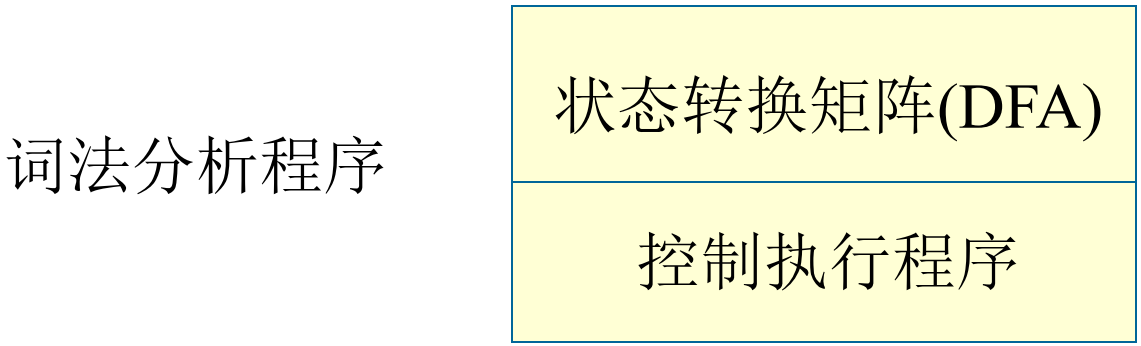
16. := {RETURN(16,—) }

17. = {RETURN(17,—) }

3.6.2 LEX的实现

LEX的功能是根据LEX源程序构造一个词法分析程序，该词法分析器实质上是一个有穷自动机。

LEX生成的词法分析程序由两部分组成:



∴LEX的功能是根据LEX源程序生成状态转换矩阵和控制程序

LEX的工作过程：

- 扫描每条识别规则Pi，构造相应的不确定有穷自动机M_i

¨将各条规则的有穷自动机Mi合并成一个新的NFA M

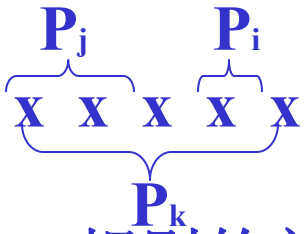
·¨确定化 NFA⇒DFA

生成该DFA的状态转换矩阵和控制执行程序

LEX二义性问题的两条原则：

1.最长匹配原则

在识别单词过程中，有一字符串



根据最长匹配原则，应识别为这是一个符合P_k规则的单词，而不是Pj和Pi规则的单词。

2.优先匹配原则

如有一字符串，有两条规则可以同时匹配时，那么用规则序列中位于前面的规则相匹配，所以排列在最前面的规则优先权最高。

例：字符串…begin…

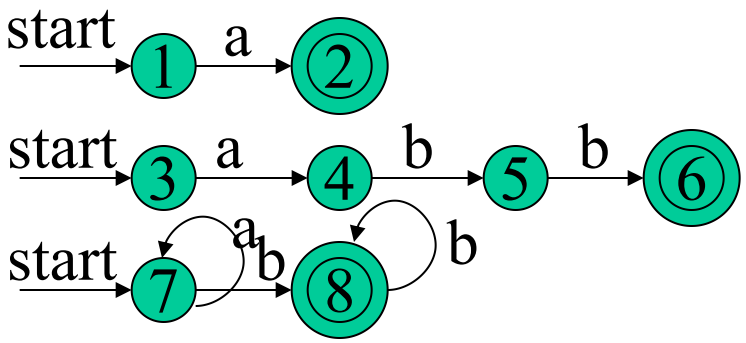
P_1 … P_8

根据原则，应该识别为关键字begin，所以在写LEX源程序时应注意规则的排列顺序。另要注意的是，优先匹配原则是在符合最长匹配的前提下执行的。

可以通过一个例子来说明这些问题：

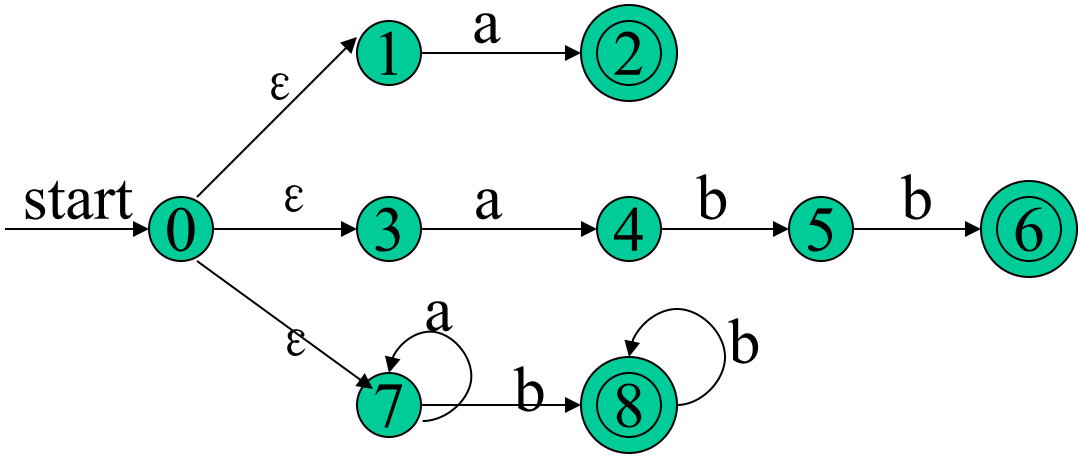
例： LEX源程序

a	{	}
abb	{	}
a^*bb^*	{	}



一.读LEX源程序，分别生成NFA，用状态图表示为：

二.合并成一个NFA：



三.确定化　　给出状态转换矩阵

状态		a	b	到达终态所识别的单词
初态	{0,1,3,7}	{2,4,7}	{8}	
终态	{2,4,7}	{7}	{5,8}	a
终态	{8}	Φ	{8}	$a^* bb^*$
	{7}	{7}	{8}	
终态	{5,8}	Φ	{6,8}	$a^* bb^*$
终态	{6,8}	Φ	{8}	abb

在此DFA中　　初态为{0,1,3,7}

终态为{2,4,7},{8},{5,8},{6,8}

词法分析程序的分析过程

令输入字符串为aba…

(1) 吃进字符ab

(2) 按反序检查状态子集
检查前一次状态是否含有原NFA的终止状态

读入字符	进入状态
开始	{0,1,3,7}
a	{2,4,7}
b	{5,8}
a	无后继状态(退掉输入字符a)

·即检查{5,8},含有终态8，因此断定所识别的单词ab是属于a*bb*中的一个。

··若在状态子集中无NFA的终态，则要从识别的单词再退掉一个字符(b)，然后再检查上一个状态子集。

…若一旦吃进的字符都退完，则识别失败，调用出错程序，一般是跳过一个字符然后重新分析。（应打印出错信息）

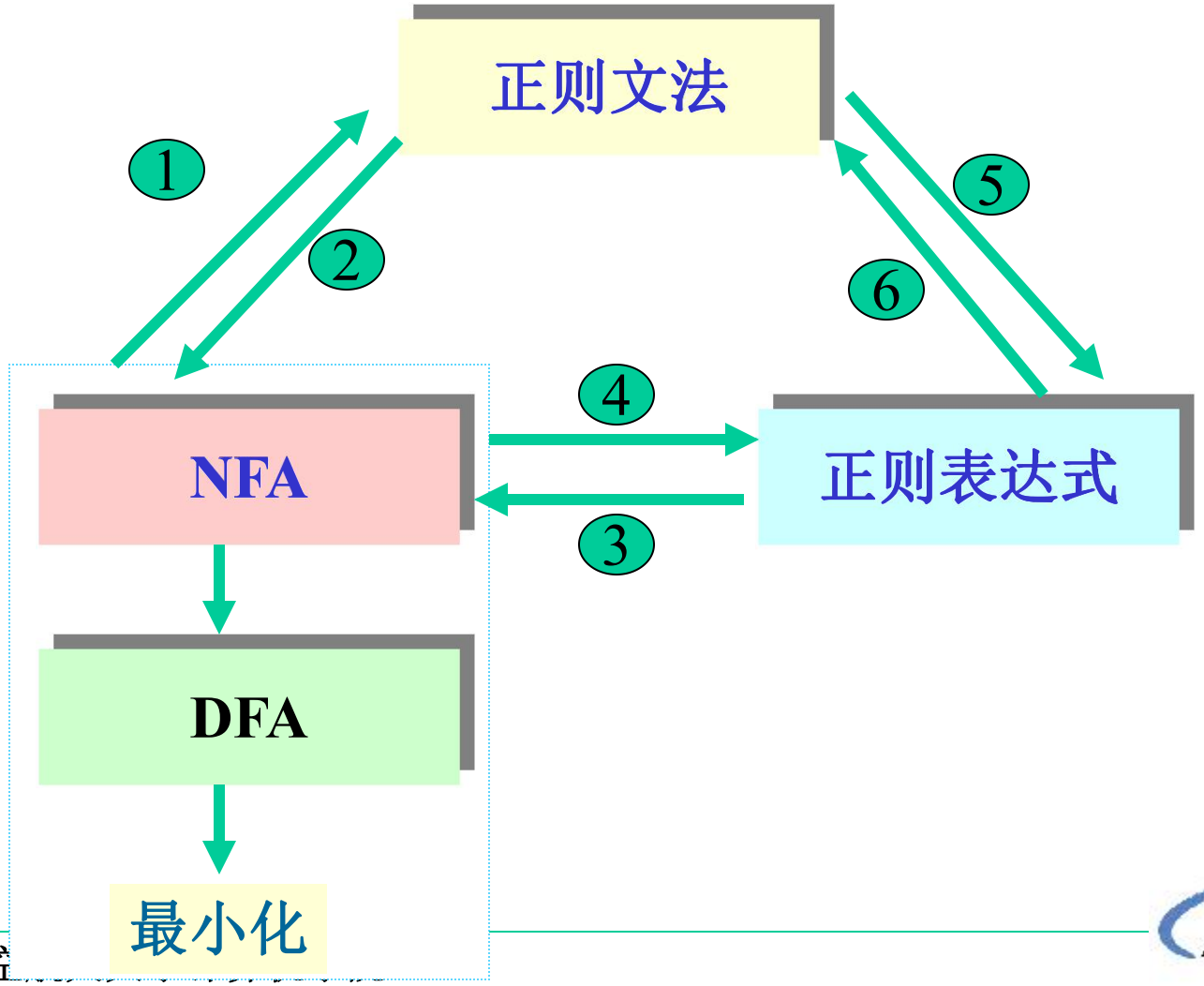
三点说明：

1) 以上是LEX的构造原理，虽然是原理性的，但据此就不难将LEX构造出来。

2) 所构造出来的LEX是一个通用的工具，用它可以生成各种语言的词法分析程序，只需要根据不同的语言书写不同的LEX源文件就可以了。

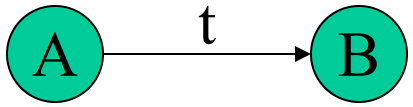
3) LEX不但能自动生成词法分析器，而且也可以产生多种模式识别器及文本编辑程序等。

补充



（1）有穷自动机⇒正则文法

算法：

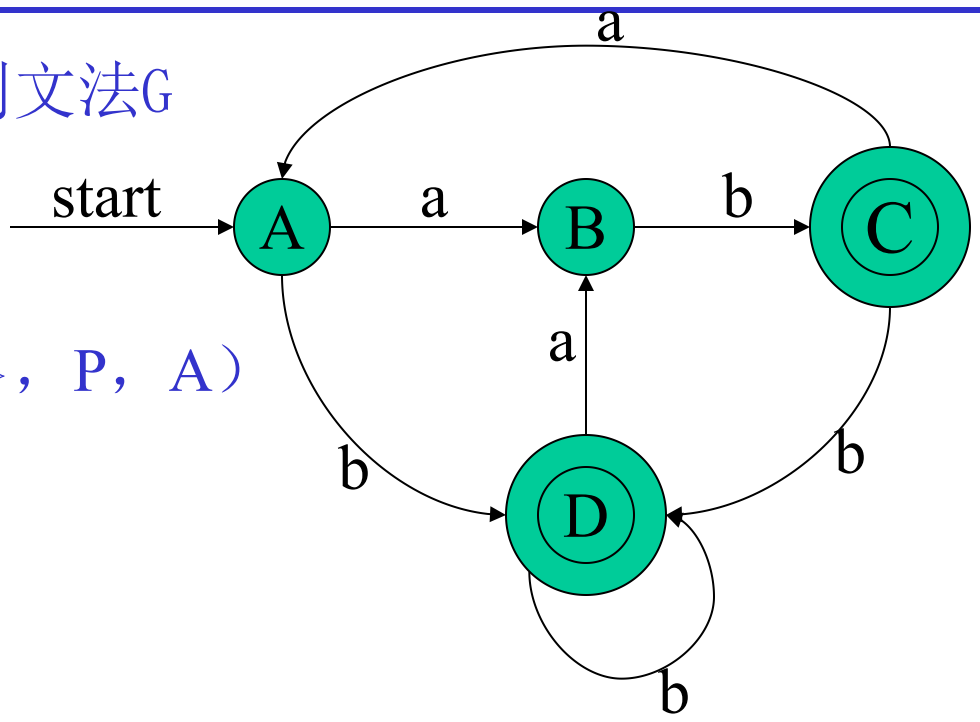


1. 对转换函数f(A, t)=B，可写成一个产生式:A→tB

2. 对可接受状态Z, 增加一个产生式:Z→ ε

3. 有穷自动机的初态对应于文法的开始符号(识别符号），有穷自动机的字母表为文法的终结符号集。

例:给出如图NFA等价的正则文法G



G=（{A，B，C，D}，{a，b}，P，A）

其中P: A→aB
A→bD
B→bC
C→aA
C→bD
C→ ε
D→aB
D→bD
D→ ε

（2）正则文法⇒有穷自动机M

算法：

1. 字母表与G的终结符号相同；
2. 为G中的每个非终结符生成M的一个状态，G的开始符号S是开始状态S；
3. 增加一个新状态Z，作为NFA的终态；
4. 对G中的形如A→tB，其中t为终结符或 ε，A和B为非终结符的产生式，构造M的一个转换函数f(A, t)=B；
5. 对G中的形如A→t的产生式，构造M的一个转换函数f(A, t)=Z。

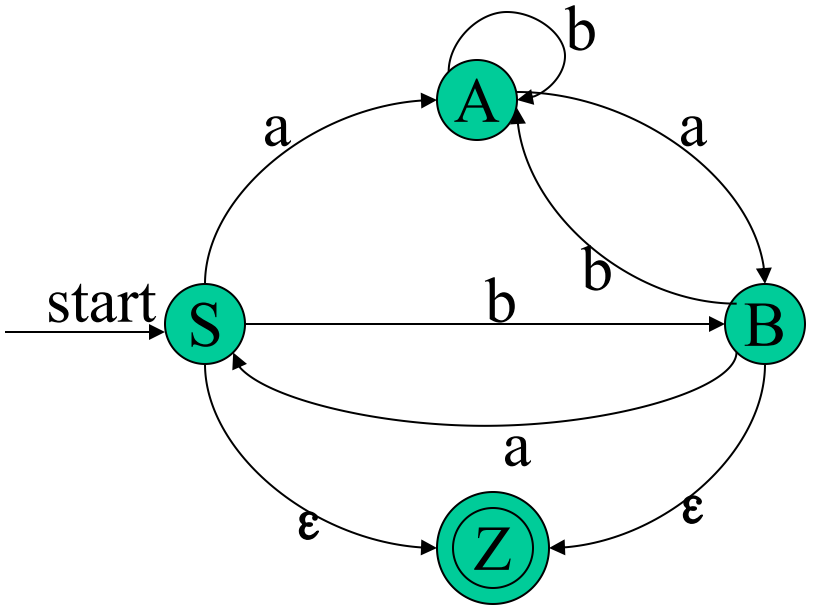
例:求与文法G[S]等价的NFA

G[S]: S→aA|bB| ε

A→aB|bA

B→aS|bA| ε

求得:

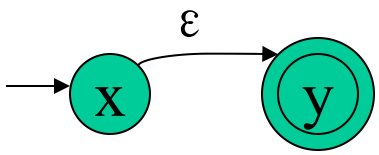


（3）正则式⇒有穷自动机

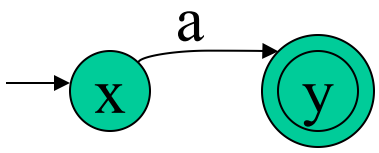
语法制导方法

1.(a)对于正则式 ϕ ,所构造NFA:

(b)对于正则式 ε ,所构造NFA:

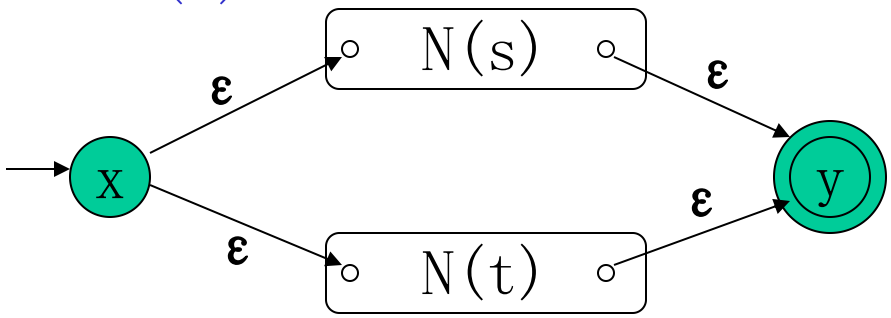


(c)对于正则式a,a∈ Σ ,则 NFA:

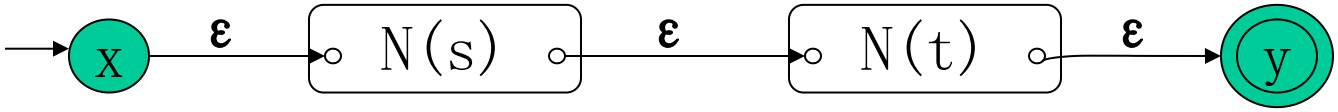


2. 若s, t为Σ上的正则式, 相应的NFA分别为N(s)和N(t);

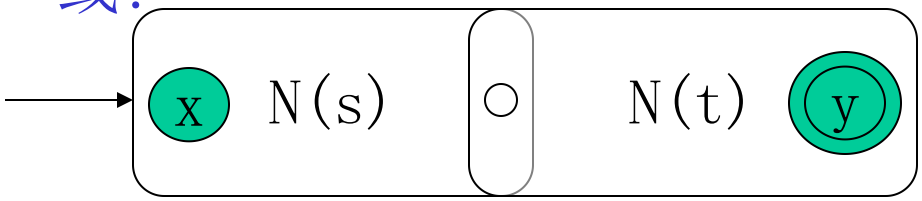
(a) 对于正则式R=s|t,　　NFA(R)



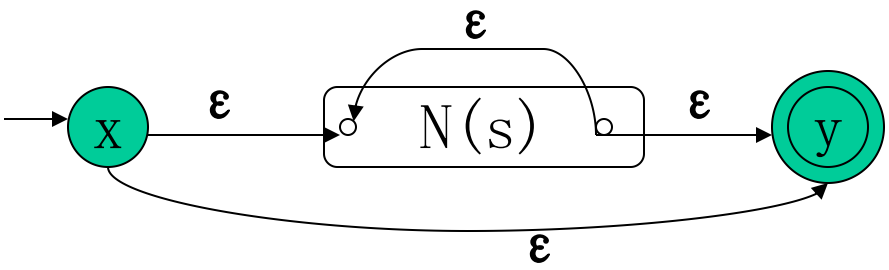
(b) 对正则式R=st,　　NFA(R)



或:



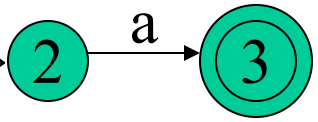
(c)对于正则式R=s*,　　NFA(R)



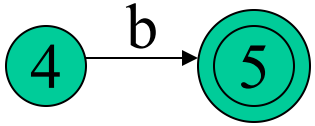
(d)对R=(s),与R=s的NFA一样.

例:为R=(a|b)* abb构造NFA N,使得L(N)=L(R)

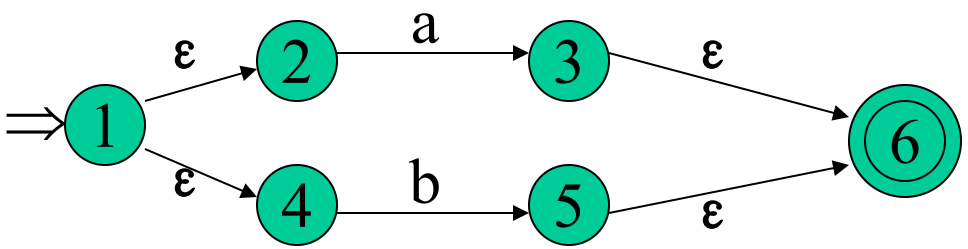
从左到右分解R,令r_1=a,第一个a,则有⇒



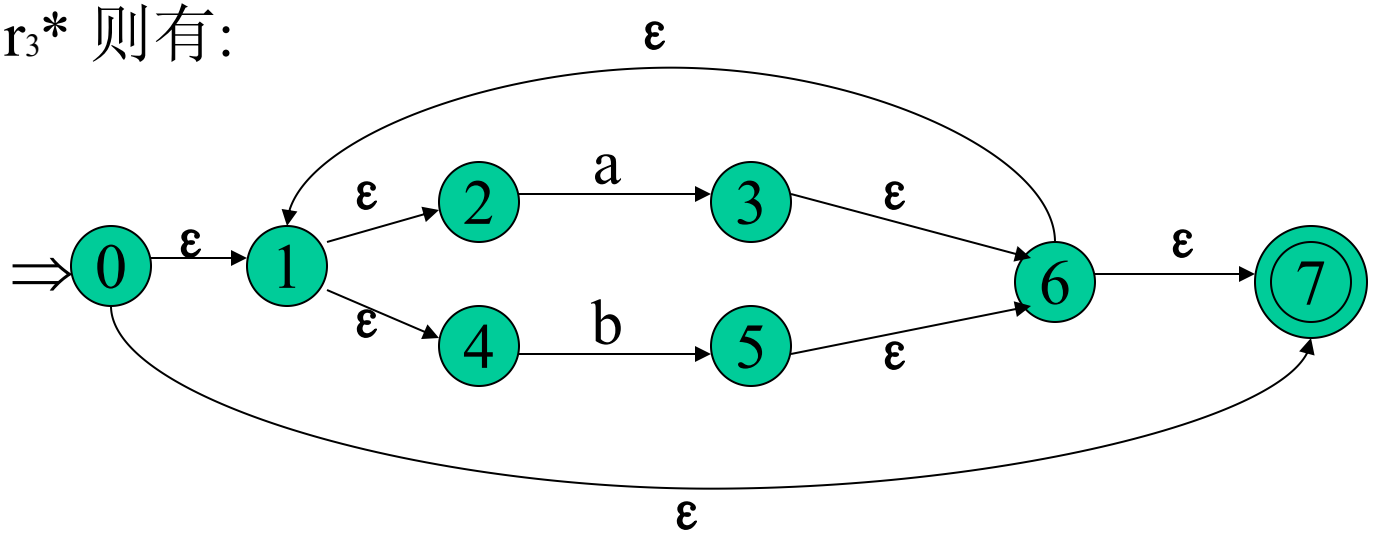
令r_2=b,则有⇒



令r_3= r_1|r_2,则有



令$r_4= r_3^*$ 则有:



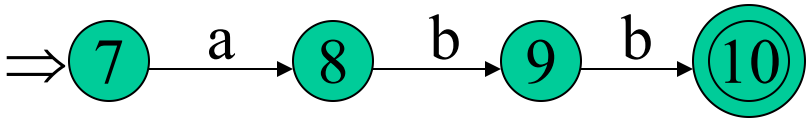
令r_5=a,

令r_6=b

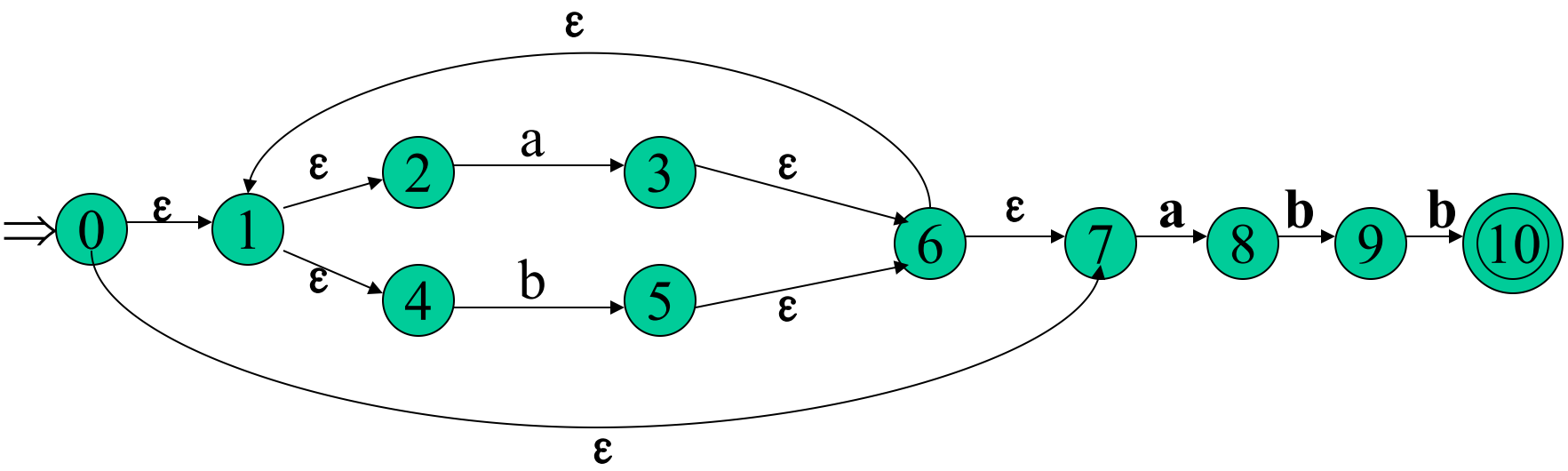
令r_7=b

令$r_8= r_5 r_6$

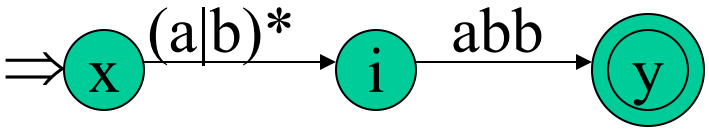
令$r_9= r_8 r_7$ 则有



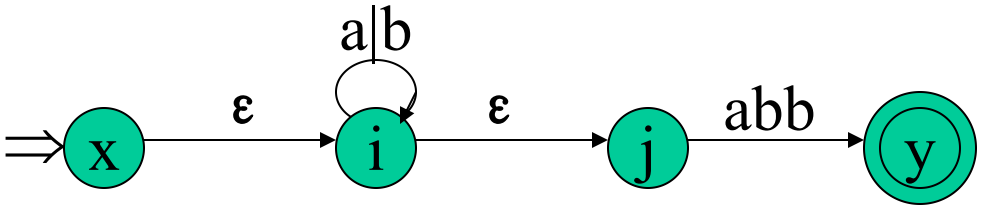
令$r_{10}= r_4 r_9$ 则最终得到NFA N:



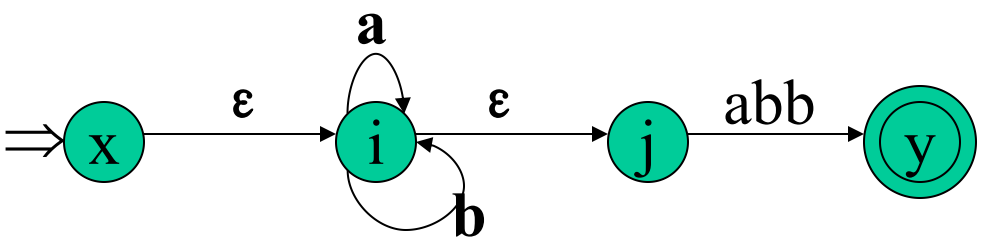
分解R的方法有很多种，下面给出另一种分解方式和所构成的NFA



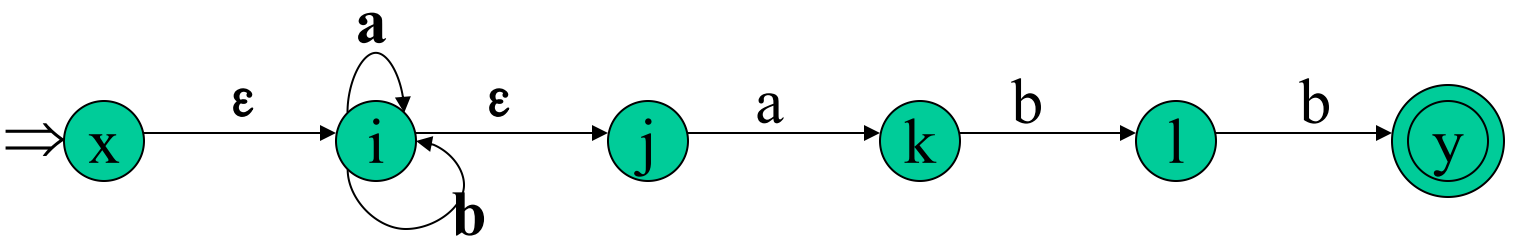
(a)



(b)



(c)



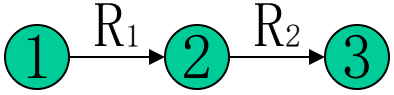
(d)

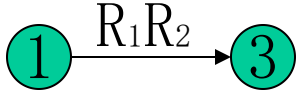
（4）有穷自动机⇒正则式R

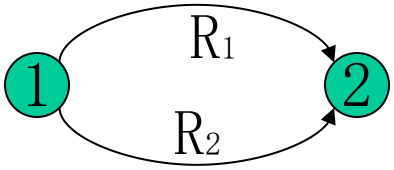
算法：

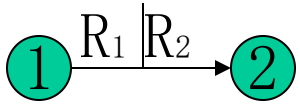
1）在M上加两个结点x, y, 从x结点用 ε 弧到M的所有初态,从M的所有终态用 ε 到y结点形成与M等价的M',M'只有一个初态x和一个终态y。

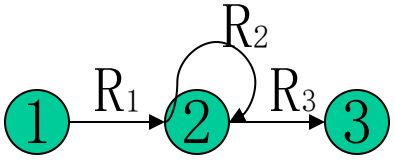
2）逐步消去M'中的所有结点,直至剩下x和y结点,在消结过程中,逐步用正则式来记弧,其消结规则如下：

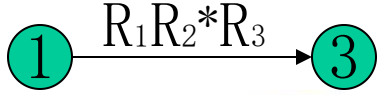
1.对于 

代之为 

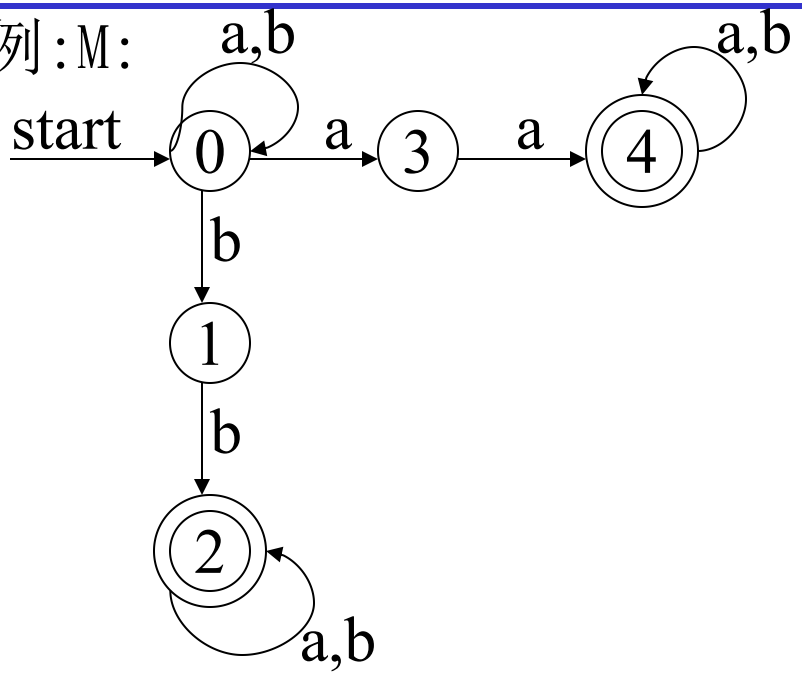
2.对于 

代之为 

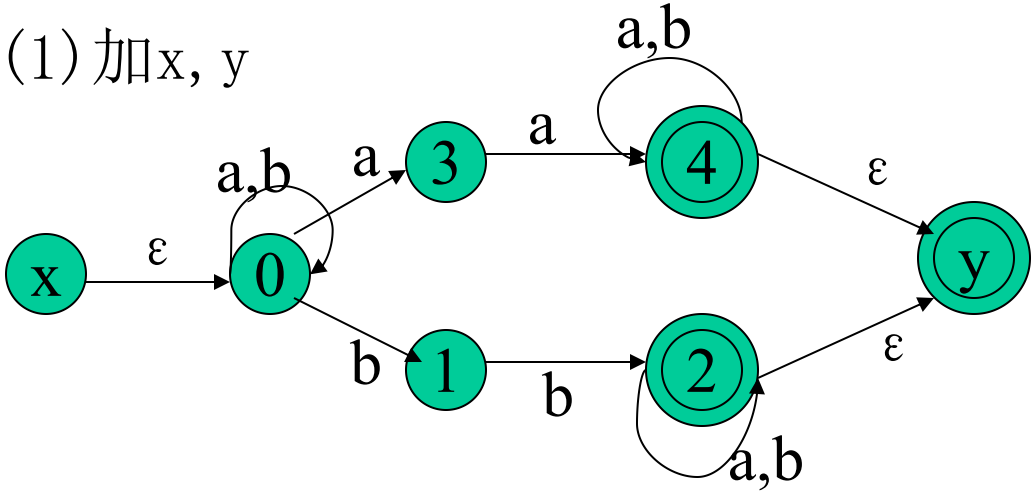
3.对于 

代之为 

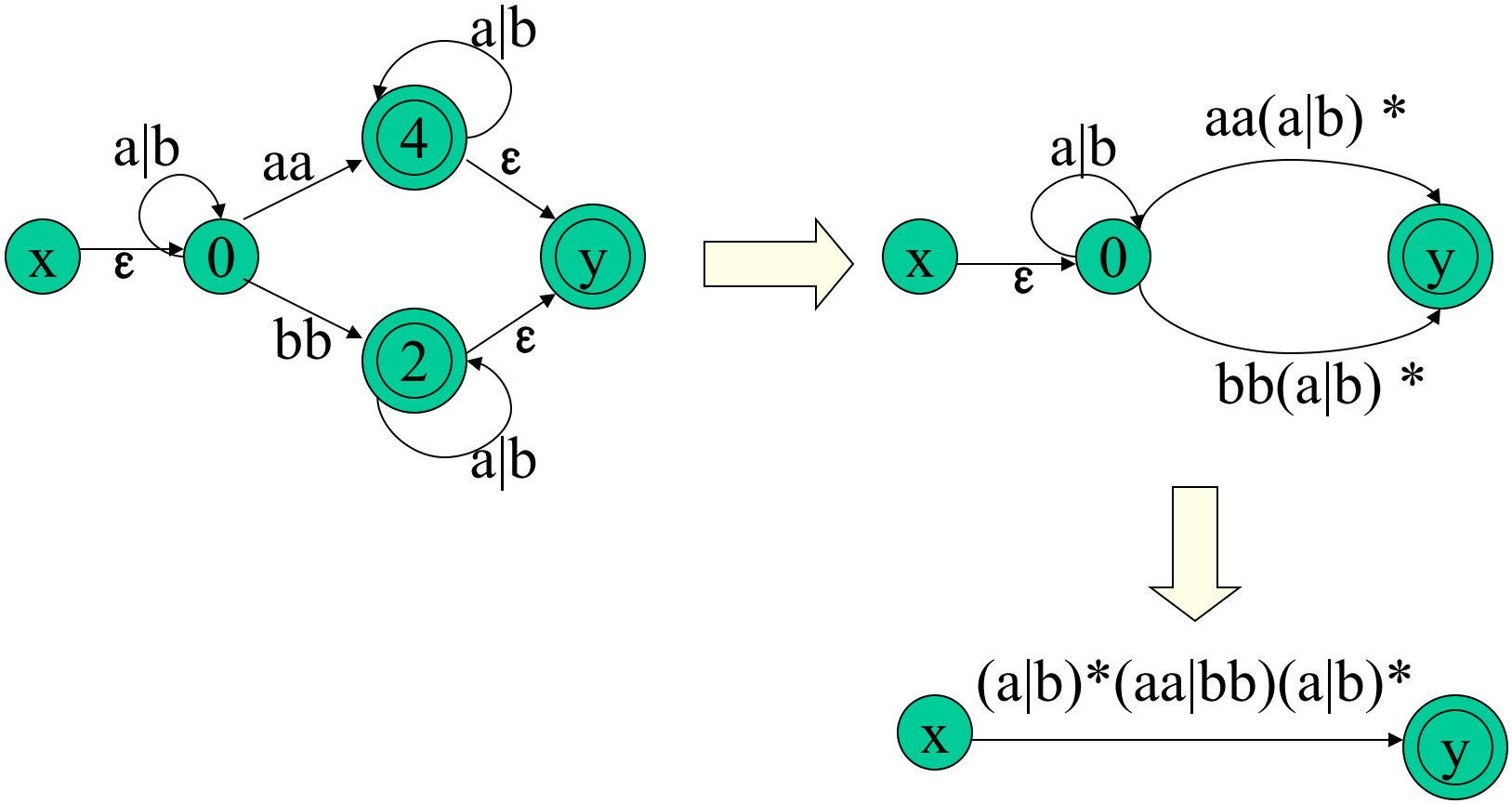
例:M:



解: (1)加x, y



(2) 消除M中的所有结点



（5）正则文法⇒正则式

利用以下转换规则，直至只剩下一个开始符号定义的产生式，并且产生式的右部不含非终结符。

规则	文法产生式	正则式
规则1	A→xB, B→y	A=xy
规则2	A→xA|y	A=x*y
规则3	A→x, A→y	A=x|y

例:有文法G[s]

S→aA|a,

A→aA|dA|a|d

于是: S=aA|a

A=(aA|dA)|(a|d)⇒A=(a|d)A|(a|d)

由规则二: A=(a|d)*(a|d)

代入:S=a(a|d)*(a|d)|a

于是:S=a((a|d)*(a|d)| ε)

（6）正则式⇒正则文法

算法:

1) 对任何正则式r,选择一个非终结符S作为识别符号,并产生产生式　S→r

2) 若x,y是正则式,对形为A→xy的产生式,重写为
A→xB　B→y,其中B为新的非终结符,$B \in V_n$
同样: 对于　A→x*y ⇒ A→xA　A→y
A→x|y ⇒ A→x　A→y

例:将R=a(a|d)*转换成相应的正则文法

解:1) S→a(a|d)*

2) S→aA
A→(a|d)*

3) S→aA
A→(a|d)A
A→ ε

4) S→aA
A→aA|dA
A→ ε

补充: DFA的简化(最小化)

“对于任一个DFA，存在一个唯一的状态最少的等价的DFA”

一个有穷自动机是化简的 ⇔ 它没有多余状态并且它的状态中没有两个是互相等价的。

一个有穷自动机可以通过消除多余状态和合并等价状态而转换成一个最小的与之等价的有穷自动机

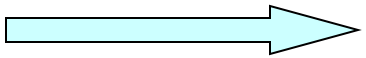
定义：

(1) **有穷自动机的多余状态：**从该自动机的开始状态出发,任何输入串也不能到达那个状态

例：

	0	1
S_0	S_1	S_5
S_1	S_2	S_7
S_2	S_2	S_5
S_3	S_5	S_7
S_4	S_5	S_6
S_5	S_3	S_1
S_6	S_8	S_0
S_7	S_0	S_1
S_8	S_3	S_6

画状态图可以看出S_4,S_6,S_8为不可达状态应该消除



	0	1
S_0	S_1	S_5
S_1	S_2	S_7
S_2	S_2	S_5
S_3	S_5	S_7
S_5	S_3	S_1
S_7	S_0	S_1

(2)等价状态⟺状态s和t的等价条件是:

1)一致性条件：状态s和t必须同时为可接受状态或不接受状态。

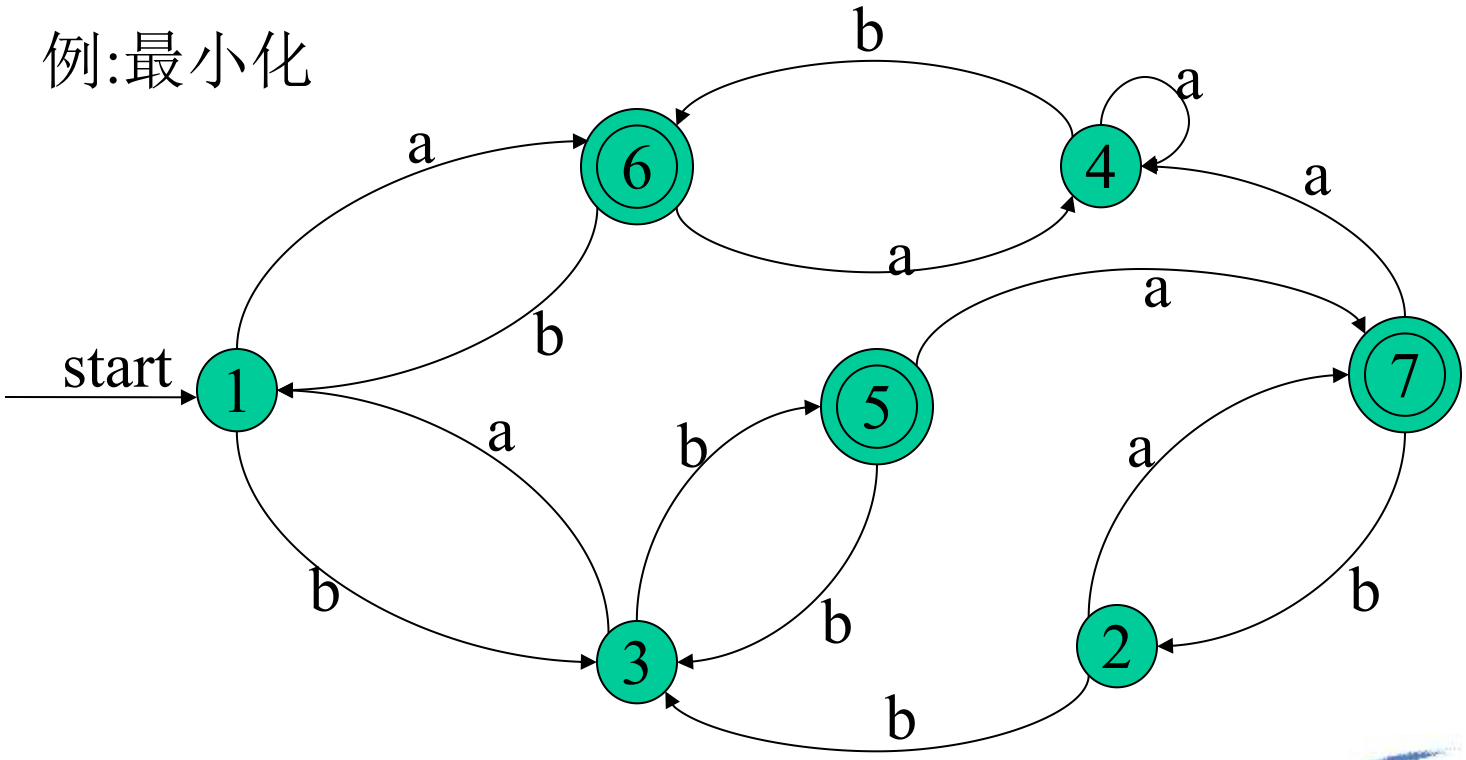
2)蔓延性条件：对于所有输入符号,状态s和t必须转换到等价的状态里。

对于所有输入符号c，$I_c(s)=I_c(t)$，即状态s、t对于c具有相同的后继，则称s，t是等价的。
（任何有后继的状态和任何无后继的状态一定不等价）

有穷自动机的状态s和t不等价,称这两个状态是可区别的。

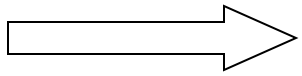
“分割法”：把一个DFA(不含多余状态)的状态分割成一些不相关的子集，使得任何不同的两个子集状态都是可区别的，而同一个子集中的任何状态都是等价的。

例:最小化

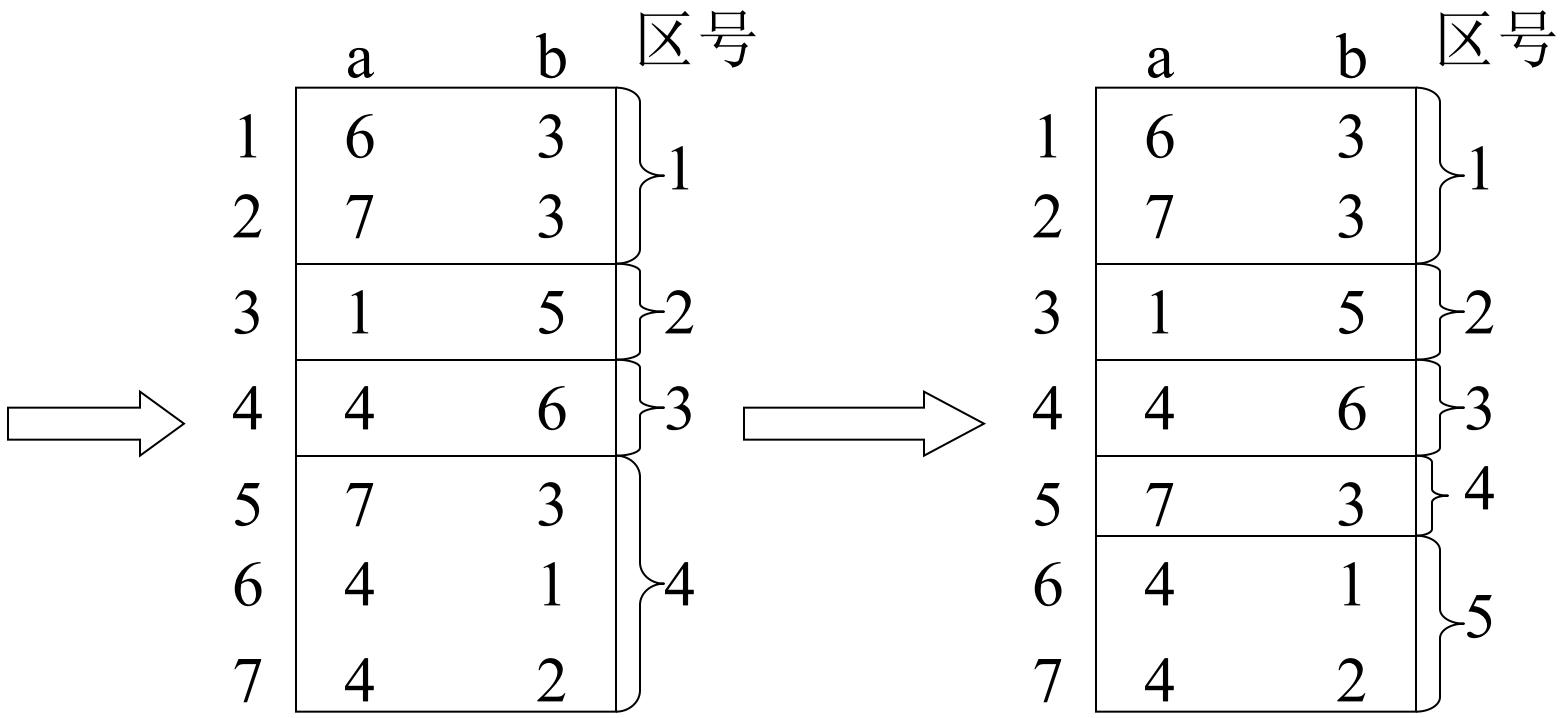


解: (一) 区分终态与非终态

	a	b	区号
1	6	3	1
2	7	3	
3	1	5	
4	4	6	
5	7	3	2
6	4	1	
7	4	2	

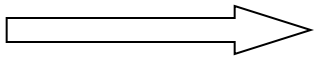


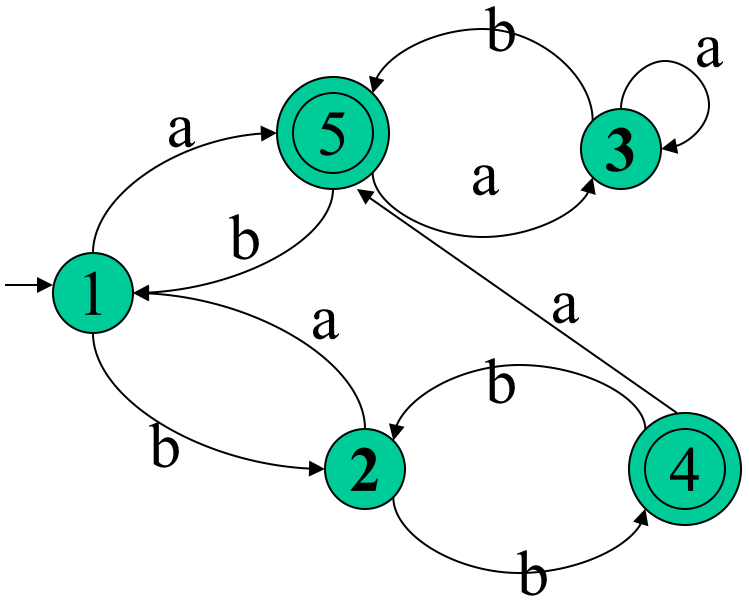
	a	b	区号
1	6	3	1
2	7	3	
3	1	5	2
4	4	6	
5	7	3	3
6	4	1	
7	4	2	


a
b
区号
1 6 3 1
2 7 3
3 1 5 2
4 4 6 3
5 7 3 4
6 4 1
7 4 2
a
b
区号
1 6 3 1
2 7 3
3 1 5 2
4 4 6 3
5 7 3 4
6 4 1 5
7 4 2

将区号代替状态号得:

	a	b
1	5	2
2	1	4
3	3	5
4	5	2
5	3	1





第四章　语法分析

- 语法分析的功能、基本任务
- 自顶向下分析法
- 自底向上分析法

4.1 语法分析概述

功能：根据文法规则，从源程序单词符号串中识别出语法成分，并进行语法检查。

基本任务：识别符号串S是否为某语法成分。

两大类分析方法：

自顶向下分析

自底向上分析

自顶向下分析算法的基本思想为：

若 $Z \overset{+}{\underset{G[Z]}{\Rightarrow}} S$ 则 $S \in L(G[Z])$ 否则 $S \notin L(G[Z])$

- ? 主要问题：
  - 左递归问题
  - 回溯问题
- ■ 主要方法：
  - 递归子程序法
  - LL分析法

自底向上分析算法的基本思想为：

若 $Z \overset{+}{\underset{G[Z]}{\Leftarrow}} S$ 则 $S \in L(G[Z])$ 否则 $S \notin L(G[Z])$

- ？主要问题：
  - 句柄的识别问题
- ▪主要方法：
  - 算符优先分析法
  - LR分析法

4.2 自顶向下分析

4.2.1 自顶向下分析的一般过程

给定符号串S，若预测是某一语法成分，则可根据该语法成分的文法,设法为S构造一棵语法树，
若成功,则S最终被识别为某一语法成分,即

$S \in L(G[Z])$，其中G[Z]为某语法成分的文法

若不成功, 则 $S \notin L(G[Z])$

- 可以通过一例子来说明语法分析过程

例：

S = cad

G[Z]:

Z::=cAd

A::=ab|a

求解 S∈L(G[Z])？

分析过程是设法建立一棵语法树,使语法树的末端结点与给定符号串相匹配。

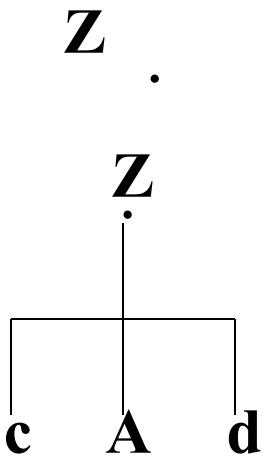
1. 开始:令Z为根结点

2. 用Z的右部符号串去匹配输入串

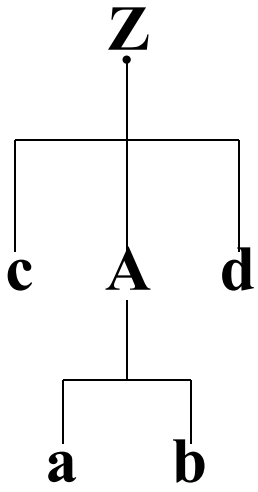
完成一步推导Z⇒cAd

检查，c-c匹配

A是非终结符,将匹配任务交给A

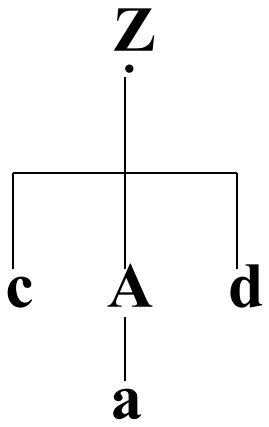


3. 选用A的右部符号串匹配输入串
A有两个右部,选第一个



完成进一步推导A⇒ab
检查,a-a匹配,b-d不匹配(失败)
但是还不能冒然宣布S∉L(G[Z])

4. 回溯 即砍掉A的子树
改选A的第二右部
A⇒a 检查 a-a匹配
d-d匹配



建立语法树,末端结点为cad,与输入cad相匹配,
建立了推导序列 Z⇒cAd⇒cad
∴cad∈L(G(Z))

自顶向下分析方法特点：

1. 分析过程是带预测的，对输入符号串要预测属于什么语法成分，然后根据该语法成分的文法建立语法树。

2. 分析过程是一种试探过程，是尽一切办法(选用不同规则) 来建立语法树的过程, 由于是试探过程, 难免有失败, 所以分析过程需进行回溯, 因此也称这种方法是带回溯的自顶向下分析方法。

3. 最左推导可以编写程序来实现, 但带溯的自顶向下分析方法在实际上价值不大, 效率低。

4.2.2 自顶向下分析存在的问题及解决方法

1、 左递归文法：

有如下文法：

令U是文法的任一非终结符，文法中有规则

$U::=U\cdots$ 或者 $U \Rightarrow^{+} U\cdots$

这个文法是左递归的。

自顶向下分析的基本缺点是：

不能处理具有左递归性的文法

为什么？

如果在匹配输入串的过程中，假定正好轮到要用非终结符U直接匹配输入串，即要用U的右部符号串U‴去匹配，为了用U‴去匹配，又得用U去匹配，这样无限的循环下去将无法终止。

如果文法具有间接左递归，则也将发生上述问题，只不过环的圈子兜得更大。

要实行自顶向下分析，必须要消除文法的左递归，下面将介绍直接左递归的消除方法，在此基础上再介绍一般左递归的消除方法。

消除直接左递归

方法一，使用扩充的BNF表示来改写文法

例：(1) E:: =E+T|T ⇒ E:: =T{+T}

(2) T:: =T*F|T/F|F ⇒ T :: =F{*F|/F}

a. 改写以后的文法消除了左递归。

b. 可以证明，改写前后的文法是等价的，表现在

$L(G_{改前}) = L(G_{改后})$

如何改写文法能消除左递归，又前后等价，
可以给出两条规则：

规则一（提因子）

若：U::=xy|xw|….|xz
则可改写为：U::=x(y|w|….|z)
若：$y=y_1y_2$, $w=y_1w_2$
则　U::=x(y_1(y_2|w_2)|….|z)

若有规则：U::=x|xy
则可以改写为：U::=x(y|ε)
注意：不应写成U::=x(ε|y)

使用提因子法，不仅有助于消除直接左递归，而且有助于压缩文件的长度，使我们能更有效地分析句子。

规则二

若有文法规则：U∷ =x|y|……|z|Uv

其特点是：具有一个直接左递归的右部并位于最后，这表明该语法类U是由x或y……或z其后随有零个或多个v组成。

U⇒Uv ⇒ Uvv ⇒ Uvvv ⇒ ……

∴ 可以改写为U∷ =(x|y|……|z){v}

通过以上两条规则，就能消除文法的直接左递归，并保持文法的等价性。

方法二，将左递归规则改为右递归规则

规则三

若：$P ::= P\alpha | \beta$

则可改写为：$P ::= \beta P'$

$P' ::= \alpha P' | \varepsilon$

例1　　E:: =E+T|T

右部无公因子，所以不能用规则一。

为了使用规则二，

令E:: =T|E+T

∴由规则二可以得到

E:: =T{+T}

例2　　T:: =T*F|T/F|F

T:: =T(*F|/F) | F　规则一

T:: =F|T(*F|/F)

T:: =F{(*F|/F)}　规则二

即T:: =F{*F|/F}

右递归：

T ::= FT’

T’ ::= *FT’ | /FT’ | ε

消除一般左递归

一般左递归也可以通过改写文法予以消除。

消除所有左递归的算法：

1. 把G的非终结符整理成某种顺序A_1，A_2，……A_n，使得：

$A_1 ::= \delta_1 | \delta_2 | \ldots\ldots \delta_k$

$A_2 ::= A_1 r\ldots\ldots$

$A_3 ::= A_2 u | A_1 v\ldots..$

……

2. For i:=1 to n do
   begin
     for j :=1 to i-1 do
       把每个形如$A_i ::= A_j r$的规则替换成
       $A_i ::= (\delta_1 | \delta_2 | \ldots\ldots | \delta_k)\ r$,
       其中$A_j ::= \delta_1 | \delta_2 | \ldots\ldots | \delta_k$是当前全部$A_j$ 的规则;
     消除A_i规则中的直接左递归
   end

3. 化简由2得到的文法即可。

例：文法G[s]为

S ::= Qc|c

Q ::= Rb|b

R ::= Sa|a

该文法无直接左递归，但有间接左递归

$S \Rightarrow Qc \Rightarrow Rbc \Rightarrow Sabc$　　$\therefore S \overset{+}{\Rightarrow} Sabc$

非终结符顺序重新排列

R::=Sa|a

Q::=Rb|b

S::=Qc|c

1. 检查规则R是否存在直接左递归　　R:: =Sa|a

2. 把R代入Q的有关选择，改写规则Q　　Q:: =Sab|ab|b

3. 检查Q是否存在直接左递归

4. 把Q代入S的右部选择　　S:: =Sabc|abc|bc|c

5. 消除S的直接左递归　　S:: =(abc|bc|c){abc}

最后得到文法为:

S:: =(abc|bc|c){abc}
Q:: =Sab|ab|b
R:: =Sa|a

可以看出其中关于Q和R的规则是多余的规则
∴经过压缩后 S:: =(abc|bc|c){abc}
可以证明改写前后的文法是等价的

应该指出,由于对非终结符的排序不同,最后得到的文法在形式上可能是不一样的,但是不难证明它们的等价。

2、回溯问题

什么是回溯?

分析工作要部分地或全部地退回去重做叫回溯。

造成回溯的条件:

文法中，对于某个非终结符号的规则其右部有多个选择，并根据所面临的输入符号不能准确地确定所要的选择时，就可能出现回溯。

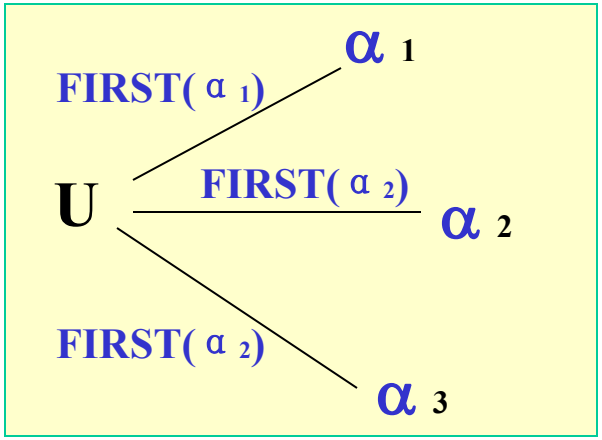
回溯带来的问题:

严重的低效率，只有在理论上的意义而无实际意义

效率低的原因

1) 语法分析要重做

2) 语义处理工作要推倒重来



设文法G（不具左递归性），$U \in V_n$

$U ::= \alpha_1 | \alpha_2 | \alpha_3$

[定义] $FIRST(\alpha_i) = \{a \mid \alpha_i \overset{*}{\Rightarrow} a\ldots, a \in V_t\}$

为避免回溯，对文法的要求是:

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi \quad (i \neq j)$$

消除回溯的途径：

1.改写文法

对具有多个右部的规则反复提取左因子

例1　U:: =xV|xW　　$U, V, W \in V_n$，　$x \in V_t^+$

改写为U:: =x(V|W)

更清楚地表示为：

U:: =xZ

Z:: =V|W

注意：问题到此并没有结束，还需要进一步检查V和W的首符号是否相交

若V:: =ab|cd　　FIRST(V) = {a,c}

W:: =de|fg　　FIRST(W) = {d,f}

只要不相交就可以根据输入符号确定目标，若相交，则要代入，并再次提取左因子。如：V::= ab　w::= ac

则：Z::= a (b|c)

例2：文法G[<程序>]

<程序> :: = <分程序>| <复合语句>

<分程序>:: = begin<说明串>；<语句串> end

<复合语句>:: = begin<语句串> end

FIRST(<分程序>) = {begin }

FIRST(<复合语句>) = {begin }

改写文法：

<程序> :: = begin （<说明串>；<语句串> end |

<语句串> end ）

引入 <程序*>

<程序> :: = begin <程序*>

<程序*> :: = <说明串>；<语句串> end | <语句串> end

<程序> :: = begin <程序*>
<程序*> :: = <说明串>；<语句串> end | <语句串> end

对于：<程序*>
FIRST(<说明串>；<语句串> end)
= {real, integer, boolean, array, function, procedure }
FIRST(<语句串>end)
= {标识符， goto, begin, if, for}
不相交。

2.超前扫描

当文法不满足避免回溯的条件时，即各选择的首符号相交时，可以采用超前扫描的方法，即向前侦察各输入符号串的第二个、第三个符号来确定要选择的目标

这种方法是通过向前多看几个符号来确定所选择的目标，从本质上来讲也有回溯的味道，因此比第一种方法费时，但是假读仅仅是向前侦察情况，不作任何语义处理工作。

例：

<程序> :: = <分程序>| <复合语句>

<分程序>:: = begin<说明串>；<语句串> end

<复合语句>:: = begin<语句串> end

这两个选择的首符号是相交的，故读到begin时并不能确定该用哪个选择，这时可采用向前假读进行侦察，此例题只需假读一次就可以确定目标。

因为<说明串>的首符集为{real，integer，……，procedure}

而<语句串>的首符集为{标识符，if，for，……，begin}

∴只要超前假读得到的是“说明”的首符，便是第一个选择；若是“语句”的首符，就是第二个选择。

文法的两个条件

为了在不采取超前扫描的前提下实现不带回溯的自顶向下分析，文法需要满足两个条件：

1、文法是非左递归的；

2、 对文法的任一非终结符，若其规则右部有多个选择时， 各选择所推出的终结符号串的首符号集合要两两不相交。

[定义] 设文法G（不具有左递归性），$U \in V_n$

$U ::= \alpha_1 | \alpha_2 | \alpha_3$

$FIRST(\alpha_i) = \{a \mid \alpha_i \overset{*}{\Rightarrow} a\ldots,\ a \in V_t\}$

为避免回溯，对文法的要求是：

$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi \quad (i \neq j)$

在上述条件下，就可以根据文法构造有效的、不带回溯的自顶向下分析器。

4.2.3 递归子程序法（递归下降分析法）

具体做法：对语法的每一个非终结符都编一个分析程序，当根据文法和当时的输入符号预测到要用某个非终结符去匹配输入串时，就调用该非终结符的分析程序。

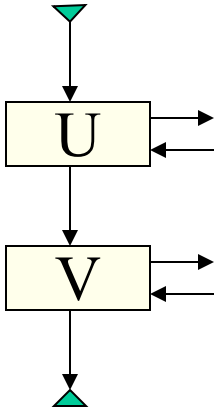
下面通过举例说明如何根据文法构造该文法的语法分析程序

如文法G[Z]:　Z ::= UV

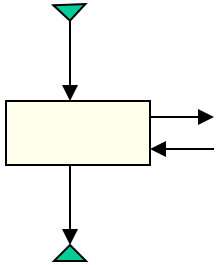
U ::= ….

V ::= ….

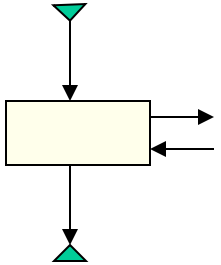
Z的分析程序



U的分析程序



V的分析程序



注：消除左递归后，可有其它递归：

U ::= …U…　　U ::= …W…

W::= …U…

例：文法G[Z]

Z:: =‘(’U‘)’|aUb

U:: =dZ|Ud|e

1.检查并改写文法

Z:: =‘ (’U‘) ’|aUb

U:: =(dZ|e){d}

改写后无左递归且首符集不相交：

{(}∩{a}= Φ

{d}∩{e}= Φ

2.检查文法的递归性

$Z \Rightarrow \cdots U \cdots \Rightarrow \cdots Z \cdots$ ∴$Z \Rightarrow \cdots Z^{+} \cdots$

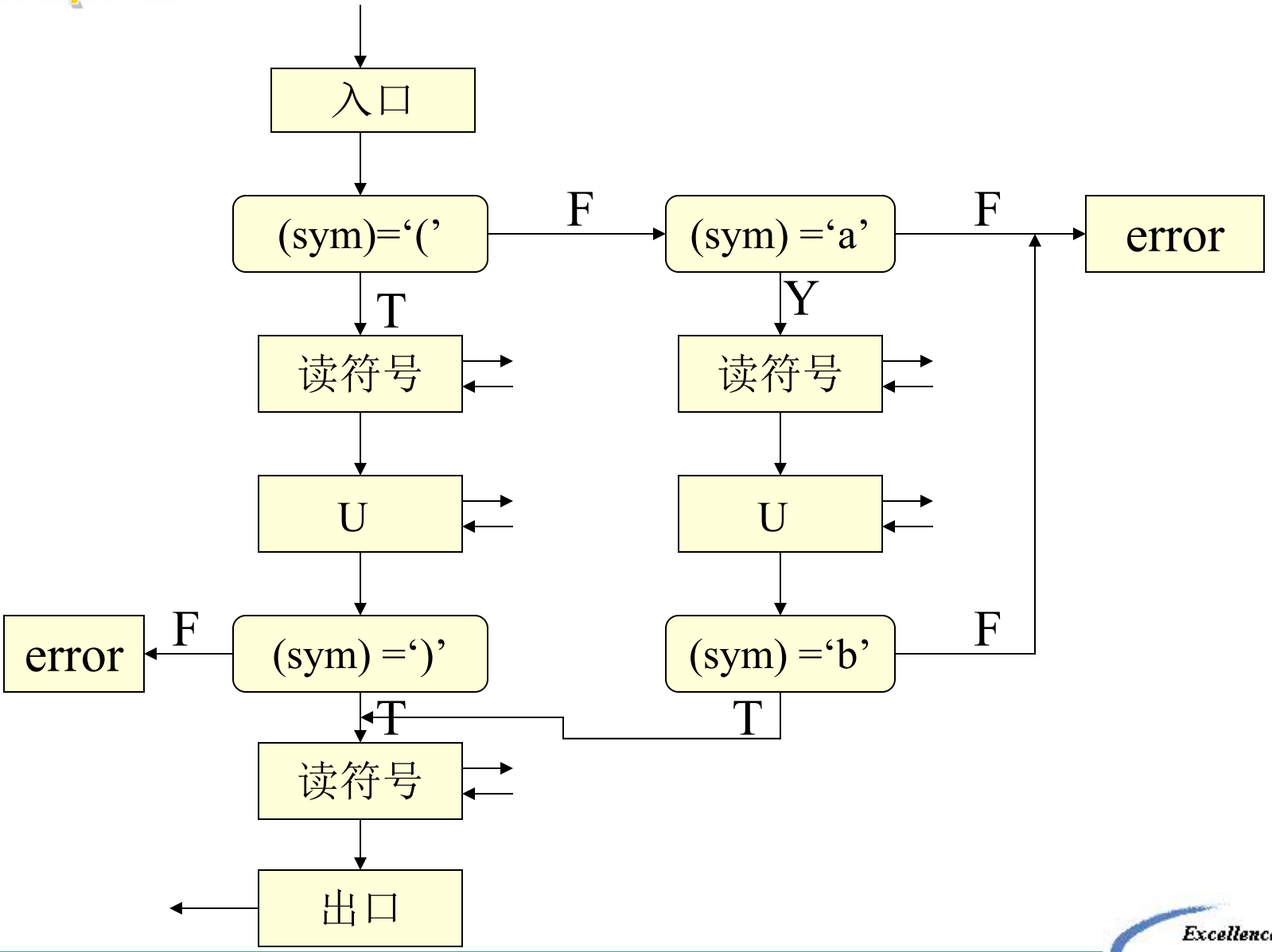
$U \Rightarrow \cdots Z \cdots \Rightarrow \cdots U \cdots$ ∴$U \Rightarrow \cdots U^{+} \cdots$

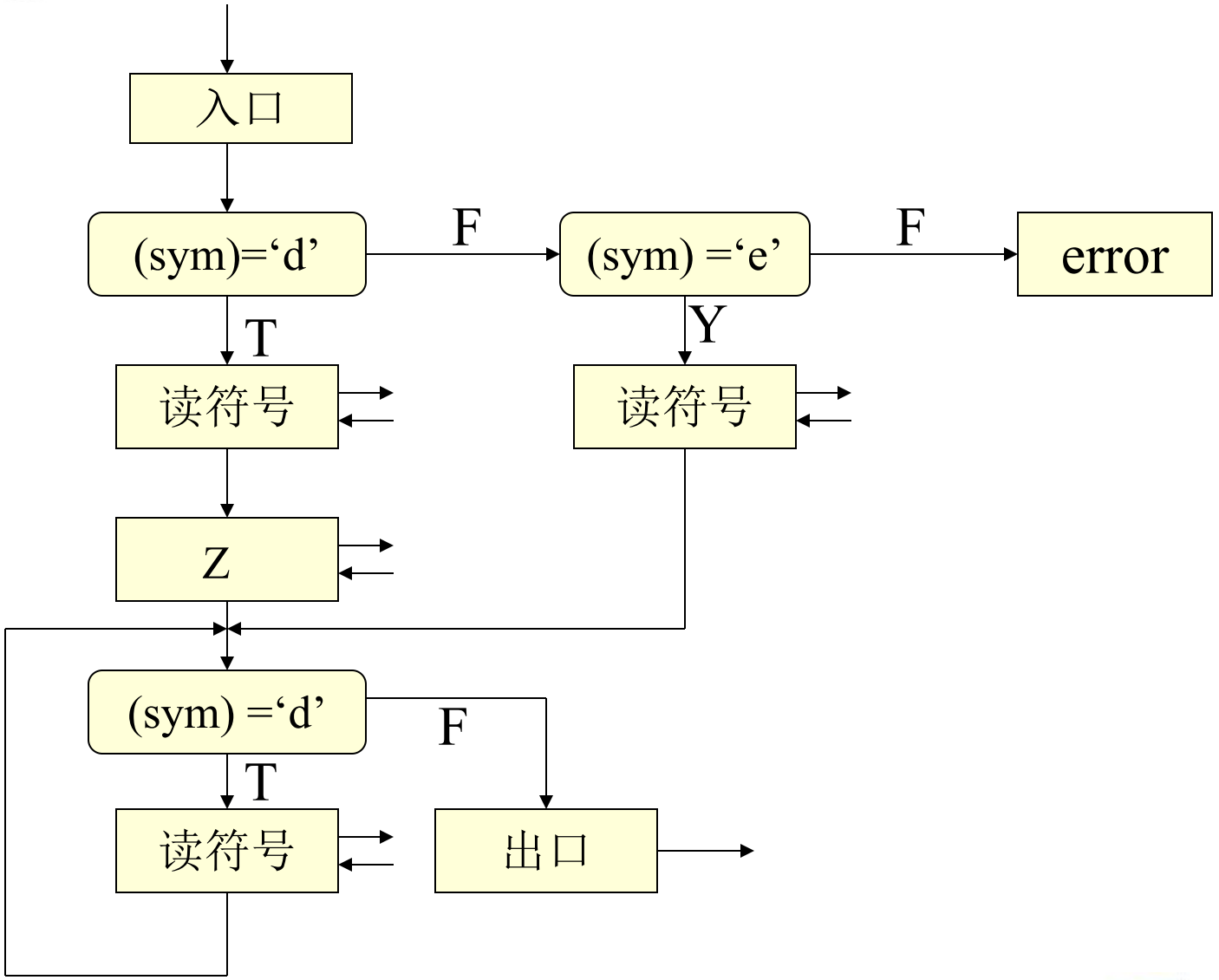
因此，Z和U的分析程序要编成递归子程序

3.算法框图

非终结符号的分析子程序的功能是：
用规则右部符号串去匹配输入串。

以下是以框图形式给出的两个子程序：


入口
(sym)=‘(’
F
(sym) =‘a’
F
error
T
Y
读符号
读符号
U
U
error
F
(sym) =‘)’
(sym) =‘b’
F
T
T
读符号
出口


入口
(sym)='d'
F
(sym) ='e'
F
error
T
Y
读符号
读符号
Z
(sym) ='d'
F
T
读符号
出口

说明

- 要注意子程序之间的接口,在程序编制时进入某个非终结符的分析程序时其所要分析的语法成分的第一个符号已读入sym中。

递归子程序法对应的是最左推导过程

4.2.4 用递归子程序法构造语法分析程序的例子

文法: <语句>:: = <变量>: =<表达式>
| IF <表达式> THEN <语句>
| IF <表达式> THEN <语句> ELSE <语句>
<变量>:: = i | i '['<表达式>']'
<表达式>:: = <项>| <表达式> + <项>
<项>:: = <因子>| <项>* <因子>
<因子>:: = <变量>| '('<表达式>')'

改写文法: <语句>:: = <变量>: =<表达式>
| IF<表达式>THEN<语句>[ELSE <语句>]
<变量>:: = i['['<表达式>']']
<表达式>:: = <项>{+<项>}
<项>:: = <因子>{*<因子>}
<因子>:: = <变量>|'('<表达式>')'

语法分析程序所要调用的子程序:

nextsym: 词法分析程序, 每调用一次读进一个单词, 单词的类别码放在sym中。

error: 出错处理程序。

```
PROCEDURE    state;                          /*语句分析子程序*/
   IF  sym =‘IF’ THEN
      BEGIN    nextsym;  expr;
          IF  sym≠‘THEN’  THEN   error
              ELSE  BEGIN  nextsym; state;
                       IF  sym= ‘ELSE’
                          THEN   BEGIN
                                      nextsym;
                                      state;
                                   END
                    END
      END
   ELSE  BEGIN    var;
               IF  sym≠‘：  =’
                  THEN  error
                   ELSE  BEGIN
                             nextsym;
                             expr;
                          END
          END
```

```
PROCEDURE     var;                                  /*变量*/
    IF  sym ≠‘i’     THEN   error
        ELSE BEGIN  nextsym;
                    IF   sym=‘[’ THEN
                         BEGIN    nextsym;
                                  expr;
                                   IF  sym ≠‘]’
                                   THEN   error
                                   ELSE    nextsym;
                         END
                END
```

<语句>:: = <变量>: =<表达式>
| IF<表达式>THEN<语句>[ELSE <语句>]
<变量>:: =i[‘[’<表达式>‘]’]
<表达式>:: =<项>{+<项>}
<项>:: =<因子>{*<因子>}
<因子>:: =<变量>|‘(’<表达式>‘)’

```
PROCEDURE    expr;                          /*表达式*/
    BEGIN   term;
        WHILE  sym=‘+’   DO
            BEGIN   nextsym;
                term;
            END
    END;
```

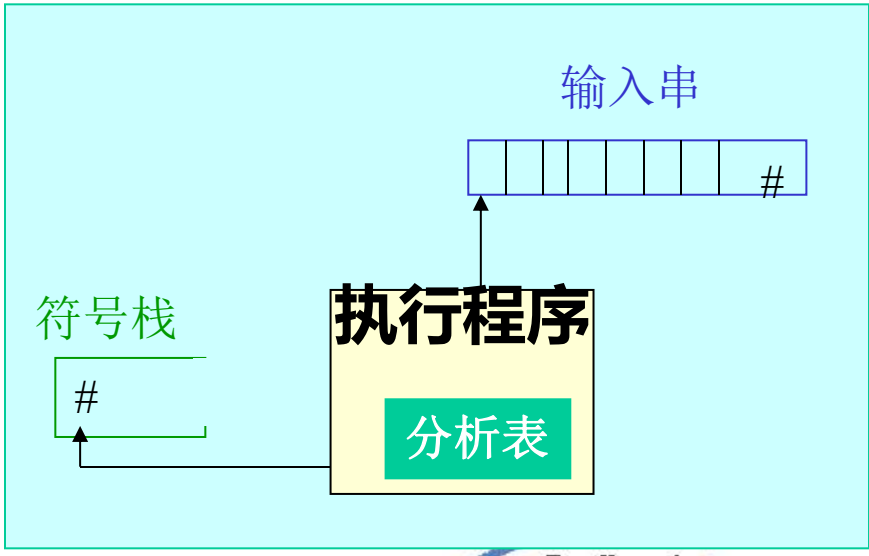
```
PROCEDURE    term;                          /*项*/
   BEGIN    factor;
      WHILE    sym=‘*’ DO
            BEGIN  nextsym;   factor  END
   END;


PROCEDURE    factor;                        /*因子*/
   BEGIN
     IF   sym=‘(’   THEN
          BEGIN  nextsym;  expr;
             IF  sym ≠‘)’
            THEN  error
            ELSE   nextsym
          END
     ELSE var;
   END
```

4.2.5 LL分析法

LL－自左向右扫描、自左向右地分析和匹配输入串。

∴ 分析过程表现为最左推导的性质。

1、LL分析程序构造及分析过程



由三部分组成:

分析表

执行程序 (总控程序）

符号栈 （分析栈）

在实际语言中，每一种语法成分都有确定的左右界符，为了研究问题方便，统一以‘#’表示。

（1）、分析表：二维矩阵M

$$M[A,a]=\begin{cases} A::=\alpha_i & \alpha_i \in V^* \\ \text{或} & A \in V_n \\ \text{error} & a \in V_t \text{ or } \# \end{cases}$$

M[A,a] = A::= α_i

表示当要用A去匹配输入串时，且当前输入符号为a时，可用A的第i个选择去匹配。

即 当 $\alpha_i \neq \varepsilon$ 时，有 $\alpha_i \overset{*}{\Rightarrow} a\ldots$;

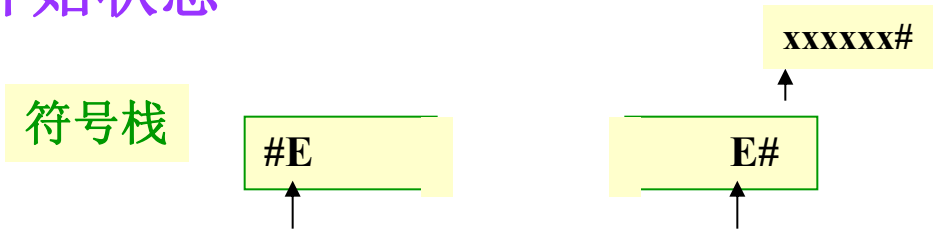
当 $\alpha_i = \varepsilon$ 时，则a为A的后继符号。

M[A,a] = error

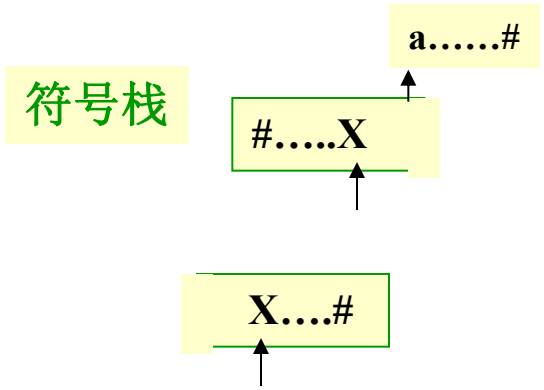
表示当用A去匹配输入串时，若当前输入符号为a，则不能匹配，表示无$A \overset{*}{\Rightarrow} a\ldots$,或a不是A的后继符号。

（2）符号栈： 有四种情况

- 开始状态



- 工作状态



	a	
X	$X::=\alpha_i$	

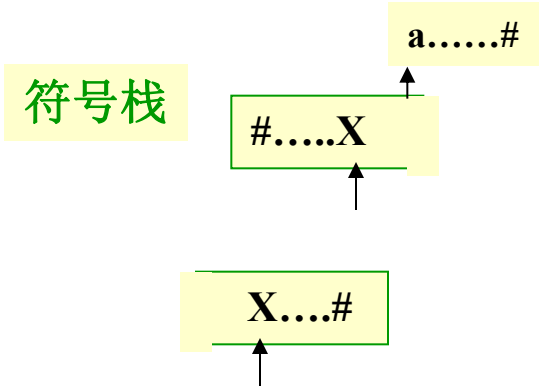
查分析表得：

$X \in V_n$, $M[X,a] = X::=\alpha_i$ 若 $X \overset{+}{\Rightarrow} a\ldots$

$X \in V_t$, $X = a$

	a	
X	error	

- 出错状态



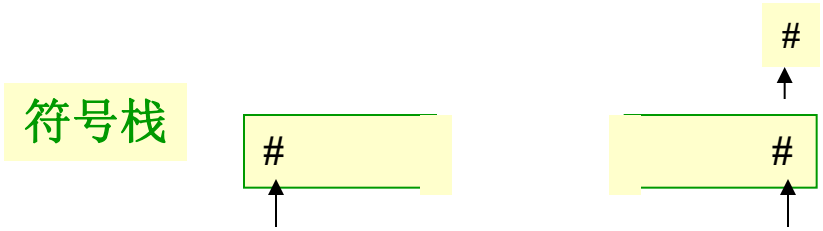
查分析表得：

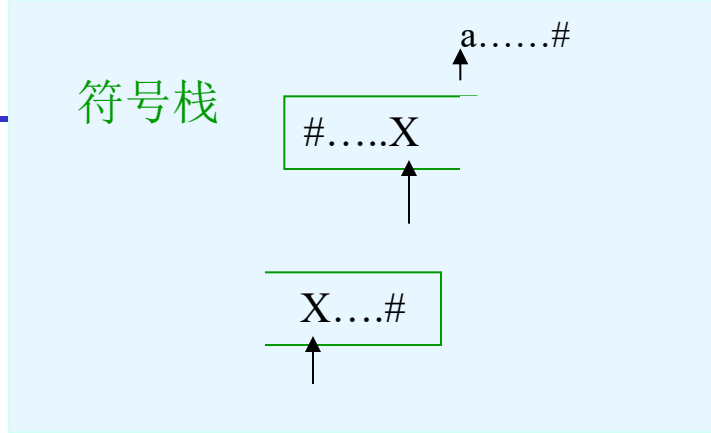
$X \in V_n$, $M[X,a] = error$

无$X \overset{+}{\Rightarrow} a\ldots$

$X \in V_t$, $X \neq a$

- 结束状态





（3）执行程序

执行程序主要实现如下操作：

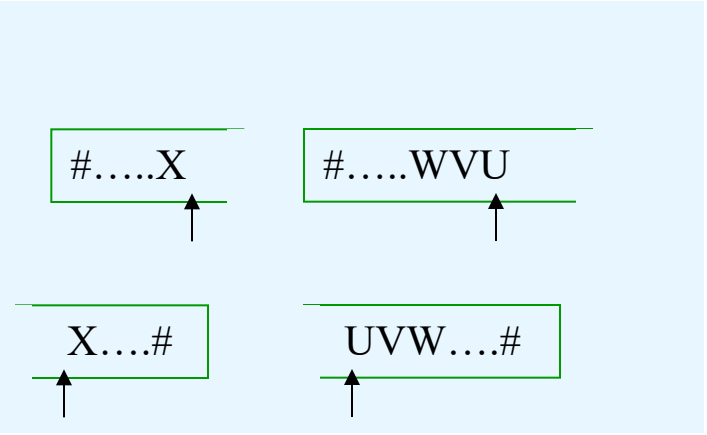
1. 把#和文法识别符号E推进栈，读入下一个符号，重复下述过程直到正常结束或出错。

2. 测定栈顶符号X和当前输入符号a,执行如下操作：

(1) 若X=a=#，分析成功，停止。E匹配输入串成功。
(2) 若X=a≠#，把X推出栈，再读入下一个符号。
(3) 若X∈V_n，查分析表M。

(3) 若X∈V_n，查分析表M

a) M[X,a] = X:: = UVW
则将X弹出栈，将UVW压入
注：U在栈顶 （最左推导）



b) M[X, a] = error 转出错处理

c) M[X, a] = X::= ε ，
— a为X的后继符号
则将X弹出栈 (不读下一符号)
继续分析。

	a	
X	X::=UVW	

例：文法G[E]

E ::= E +T | T

T ::= T * F | F

F ::= (E)|i

消除左递归

E ::= TE’

E ::= +TE’| ε

T ::= FT’

T’ ::= *FT’| ε

F ::= (E)|i

分析表

	i	+	*	(	)	#
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::= i			F::= (E)		

★ 注：矩阵元素空白表示Error

输入串为: i+i*i#

步骤	符号栈		读入符号	剩余符号串	使用规则
1.	#E	E#	i	+i*i#	
2.	#E'T	TE'#	i	+i*i#	E::=TE'
3.	#E'T'F	FT'E'#	i	+i*i#	T::=FT'
4.	#E'T' i	iT'E'#	i	+i*i#	F::= i
5.	#E'T'	T'E'#	+	i*i#	(出栈，读下一个符号)
6.	#E'	E'#	+	i*i#	T::= ε
7.	#E'T+	+TE'#	+	i*i#	E'::=+TE'

步骤	符号栈	读入符号	剩余符号串	使用规则
8.	#E’T	i	*i#	
9.	#E’T’F	i	*i#	T::=FT’
10.	#E’T’ i	i	*i#	F::= i
11.	#E’T’	*	i#	
12.	#E’T’F*	*	i#	T’::=*FT’
13.	#E’T’F	i	#	
14.	#E’T’ i	i	#	F::= i
15.	#E’T’	#		
16.	#E’	#		T’::= ε
17.	#	#		E’::= ε

推导过程：

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

最左推导。

2、分析表的构造

设有文法G[Z]:

定义：**FIRST(α)**=$\{a|\ \alpha \overset{*}{\Rightarrow} a\ldots,\ a\in V_t\}$

$\alpha \in V^*$, 若 $\alpha \overset{*}{\Rightarrow} \varepsilon$ ， 则 $\varepsilon \in$FIRST(α)

该集合称为α 的头符号集合。

定义：**FOLLOW(A)**=$\{a|\ Z \overset{*}{\Rightarrow} \ldots Aa\ldots,\ a\in V_t\}$

$A\in V_n$ ， Z识别符号

该集合称为A的后继符号集合。

特殊地：若$Z \overset{*}{\Rightarrow} \ldots A$ 则 #∈FOLLOW(A)

构造集合FIRST的算法

设 $\alpha = X_1X_2...X_n$，　　$X_i \in V_n \cup V_t$

求FIRST(α)=?

首先求出组成 α 的每一个符号X_i的FIRST集合

(1) 若$X_i \in V_t$，则 FIRST(X_i) = {X_i}

(2) 若$X_i \in V_n$ 且 $X_i ::= a..... | \varepsilon$ ， $a \in V_t$

则 FIRST(X_i)={a， ε }

(3) 若$X_i \in V_n$且$X_i ::= y_1 y_2 \ldots y_k$，则按如下顺序计算FIRST($X_i$)

FIRST(X_i) $\Leftarrow$ FIRST(y_1) $-\{\varepsilon\}$;

若 $\varepsilon \in$FIRST(y_1) 则将FIRST(y_2)-$\{\varepsilon\}$加入FIRST(X_i);

若 $\varepsilon \in$FIRST(y_1)

$\varepsilon \in$FIRST(y_2) 则将FIRST(y_3)-$\{\varepsilon\}$加入FIRST(X_i)

.....

若 $\varepsilon \in$FIRST(y_{k-1}) 则将FIRST(y_k)-$\{\varepsilon\}$加入FIRST(X_i)

若 $\varepsilon \in$FIRST(y_1) ～ FIRST(y_k)

则将 ε 加入FIRST(X_i)

★ 注意：要顺序往下做，一旦不满足条件，过程就要中断进行

★ 得到FIRST(X_i)，即可求出FIRST(α)

2.构造集合FOLLOW的算法

设$S, A, B \in V_n$，
算法：连续使用以下规则，直至FOLLOW集合不再扩大

(1) 若S为识别符号,则把“#”加入FOLLOW(S)中

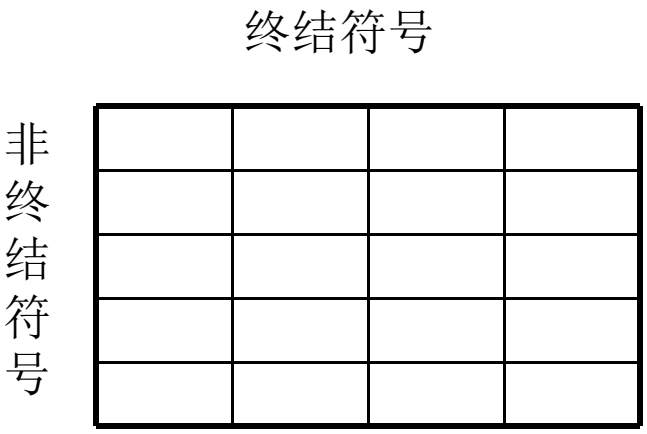
(2) 若$A::=\alpha B \beta$ $(\beta \neq \varepsilon)$,则把FIRST(β)-{ε}加入FOLLOW(B)

(3) 若$A::=\alpha B$ 或$A::=\alpha B \beta$， 且$\beta \overset{*}{\Rightarrow} \varepsilon$ 则把FOLLOW(A)加入FOLLOW(B)

注：FOLLOW集合中不能有 ε

2、构造分析表

基本思想是:
当文法中某一非终结符呈现在栈顶时,根据当前的输入符号,分析表应指示要用该非终结符的哪一条规则去匹配输入串(即进行一步最左推导)



根据这个思想, 不难把构造分析表算法构造出来!

算法：

设A::＝α_i为文法中的任意一条规则，a为任一终结符或#。

1、若a ∈FIRST（ α_i ），则A::= α_i ==> M[A,a]

表示：A在栈顶，输入符号是a，应选择α_i去匹配

2、若 $\alpha_i = \varepsilon$ 或 $\alpha_i \overset{+}{\Rightarrow} \varepsilon$,而且a ∈FOLLOW（A），

则A::= α_i ==>M[A,a]，表示A已经匹配输入串成功，

其后继符号终结符a由A后面的语法成分去匹配。

3、把所有无定义的M[A,a]都标上error

例：G[E]分析表的构造

E:: =TE’

E’:: =+TE’| ε

T:: =FT’

T’:: =*FT’| ε

F:: =(E)|i

求FIRST：

FIRST(F)={(,i}

FIRST(T’)={*, ε }

FIRST(T)=FIRST(F)-{ ε }={(,i}

FIRST(E’)={+, ε }

FIRST(E)= FIRST(T)-{ ε }={(,i}

∴FIRST(TE’)=FIRST(T)-{ ε }={(,i}

FIRST(+TE’)={+}　　FIRST(ε)={ ε }

FIRST(FT’)= FIRST(F)-{ ε }={(,i}

FIRST(*FT’)={*}　　FIRST(ε)={ ε }

FIRST(（E）)={(}　　FIRST(i)={i}

E:: =TE’

E’:: =+TE’| ε

T:: =FT’

T’:: =*FT’| ε

F:: =(E)|i

求FOLLOW

FOLLOW(E)={#,)} ∵因为E是识别符号∴#∈FOLLOW(E)

又F:: =(E) ∴)∈FOLLOW(E)

FOLLOW(E’)={#,)} ∵E:: =TE’ ∴FOLLOW(E)加入FOLLOW(E’)

FOLLOW(T)={+,), #} ∵E’:: =+TE’ ∴FIRST(E’)-{ ε }加入FOLLOW(T)

又E’⇒ ε， ∴ FOLLOW(E’)加入FOLLOW(T)

FOLLOW(T’)= FOLLOW(T)= {+,),#}

∵T:: =FT’ ∴ FOLLOW(T)加入FOLLOW(T’)

FOLLOW(F)={*, + ,), #} ∵T:: =FT’ ∴ FOLLOW(F)=FIRST（T’）-{ ε }

又T’$\overset{*}{\Rightarrow}$ ε ∴ FOLLOW(T)加入FOLLOW(F)

构造分析表

	i	+	*	(	)	#
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:: =i			F:: =(E)		

注意:用上述算法可以构造出任意给定文法的分析表,但不是所有文法都能构造出上述那种形状的分析表即M[A,a]=一条的规则或Error。**对于能用上述算法构造分析表的文法称为LL(1)文法**

3、LL(1)文法

定义：一个文法G，其分析表M不含多重定义入口(即分析表中无二条以上规则)，则称它是一个LL(1)文法.

定理：文法G是LL(1)文法的充分必要条件是：对于G的每一个非终结符A的任意两条规则$A::=\alpha \mid \beta$,下列条件成立：

1、$FIRST(\alpha) \cap FIRST(\beta) = \Phi$

2、若 $\beta \overset{*}{\Rightarrow} \varepsilon$，则$FIRST(\alpha) \cap FOLLOW(A) = \Phi$

用此构造分析表的算法,可以构造任何文法的分析表,但对于某些文法,有些M[A,a]中可能有若干条规则,这称为分析表的多重定义或者多重入口。

可以证明: 如果G是左递归的,或者是二义性的文法,则至少有一个多重入口。

左递归:

U:: =U…| a …

则有：FIRST(U…)∩FIRST(a…)≠Φ

∴M[U,a]={U:: =U…, U:: =a…}

二义文法:

对文法所定义的某些句子存在着两个最左推导,即在推导的某些步上存在多重定义,有两条规则可用，所以分析表是多重定义的。

4、LL分析的错误恢复----补充（不要求)

当符号栈顶的终结符和下一个输入符号不匹配,或栈顶是非终结符A，输入符号a,而M[A,a]为空白(即error)时，分析发现错误。

错误恢复的基本思想是: 跳过一些输入符号,直到期望的同步符号之一出现为止。

同步符号(可重新开始继续分析的输入符号)集合通常可按以下方法确定:

1) 把FOLLOW(A)的所有符号加入A的同步符号集合，跳过输入符号直到出现FOLLOW(A)的元素, 便把A从栈中弹出, 继续往下分析。

2) 为了避免仅按1)来确定同步符号集合会使跳读过多(如输入串中缺少语句结束符号“;”)，可将程序高层语法结构(成分)的开始符号(通常是关键词)加入到低层语法结构的同步集合中。

3) 把FOLLOW(A)的符号加入A的同步集合中。

4) 如果栈顶的非终结符号A可以产生空串,可以将A从栈中弹出。

5) 如果终结符在栈顶而不能匹配,则可弹出该终结符,继续分析，这好比把所有其他符号均作为该符号的同步集合元素。

4.3 自底向上分析

基本算法思想:

若采用自左向右的描述和分析输入串,那么自底向上的基本算法是:

从输入符号串开始，通过重复查找当前句型的句柄(最左简单短语)，并利用有关规则进行归约，若能归约为文法的识别符号，则表示分析成功，输入符号串是文法的合法句子，否则有语法错误。

分析过程是重复以下步骤：

1、找出当前句型的句柄 x （或句柄的变形）

2、找出以 x 为右部的规则 X::= x

3、把 x 归约为X，产生语法树的一枝

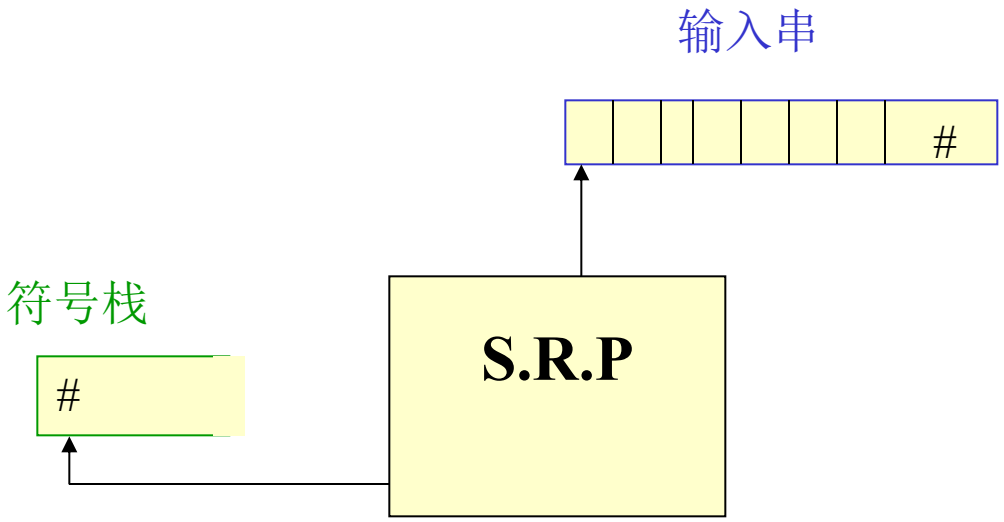
关键：找出当前句型的句柄 x (或其变形），这不是很容易。

主要内容：

- 自底向上分析的一般过程（移进-归约分析）
- 算符优先分析法
- LR分析法

4.3.1 移进—归约分析（Shift-reduce parsing)

要点：建立符号栈，用来记录分析的历史和现状，并根据所面临的状态，确定下一步动作是移进还是归约。



分析过程：把输入符号串按扫描顺序一一地移进符号栈（一次移一个），检查栈中符号，当在栈顶的若干符号形成当前句型的句柄时，就根据规则进行归约，将句柄从符号栈中弹出，并将相应的非终结符号压入栈内（即规则的左部符号），然后再检查栈内符号串是否形成新的句柄，若有就再进行归约，否则移进符号。分析一直进行到读到输入串的右界符为止。最后，若栈中仅含有左界符号和识别符号，则表示分析成功，否则失败。

例：G[S]:　　输入串为：

S ::= aAcBe　　abbcde

A::= b

A::= Ab

B::= d

输入串为abbcde，检查是否是该文法的合法句子：

若采用自底向上分析，即能否一步步归约当前句型的句柄，最终归约到识别符号S。先设立一个符号栈，将符号“#”作为待分析的符号串的左右分界符。

作为初始状态，先将符号串的左分界符推进符号栈，作为栈底符号。

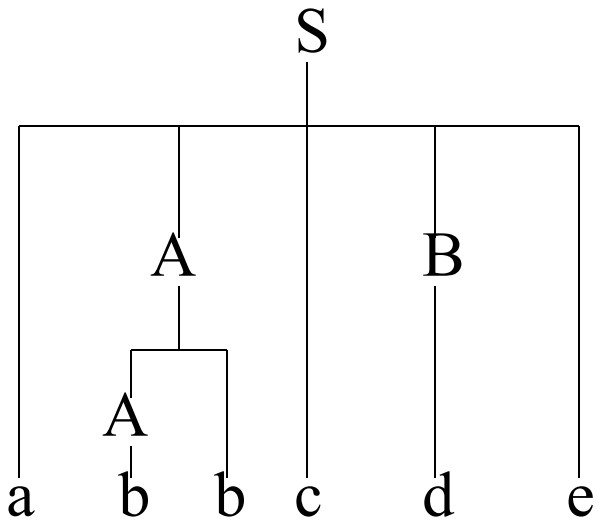
分析过程如下表:

步骤	符号栈	输入符号串	动作
1	#	abbcde#	准备,初始化
2	#a	bbcde#	移进
3	#ab	bcde#	移进
4	#aA	bcde#	归约(A:: =b)
5	#aAb	cde#	移进
6	#aA	cde#	归约(A:: =Ab)
7	#aAc	de#	移进
8	#aAcd	e#	移进
9	#aAcB	e#	归约(B:: =d)
10	#aAcBe	#	移进
11	#S	#	归约(S:: =aAcBe)
12	#S	#	成功

这一方法简单明了,不断地进行移进归约,关键是确定当前句型的句柄。

说明: 1) 例子的分析过程是一步步地归约当前句型的句柄

该句子的唯一语法树为:



注意两点：

（1）栈内符号串 + 未处理输入符号串 = 当前句型

（2）句柄都在栈顶

实际上，以上分析过程并未真正解决句柄的识别问题

2) 未真正解决句柄的识别。

上述分析过程是怎样识别句柄的，主要看栈顶符号串是否形成规则的右部。

这种做法形式上是正确的，但在实际上不一定正确。举例的分析过程可以说是一种巧合。

因为不能认为: 对句型 xuy 而言

若有U::＝u，即U⇒u 就断定u是简单短语，

u 就是句柄，而是要同时满足 $Z \overset{*}{\Rightarrow} xUy$

4.3.2 算符优先分析(Operator-Precedence Parsing)

1) 这是一种经典的自底向上分析法，简单直观，并被广泛使用，开始主要是对表达式的分析，现在已不限于此。可以用于一大类上下无关的文法。

2) 称为算符优先分析是因为这种方法是仿效算术式的四则运算而建立起来的，作算术式的四则运算时，为了保证计算结果和过程的唯一性，规定了一个统一的四则运算法则，规定运算符之间的优先关系。

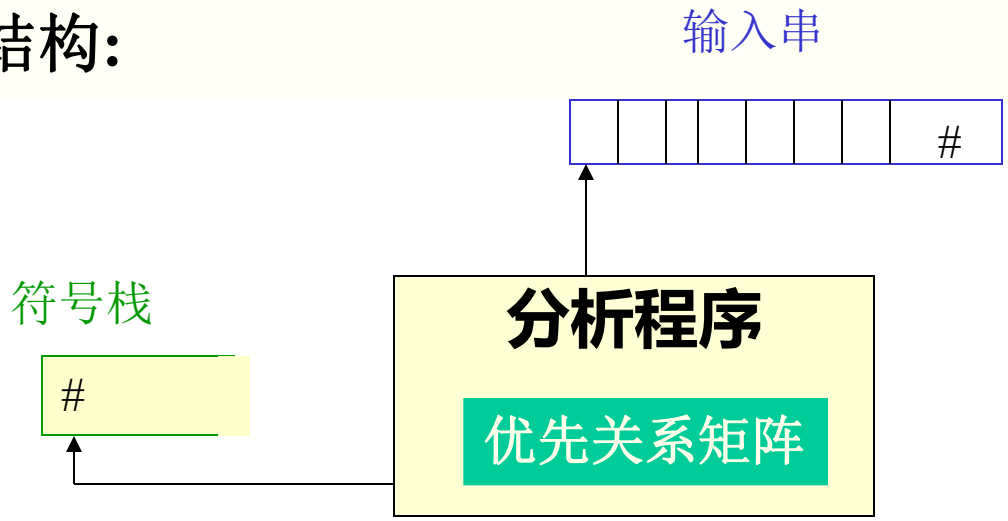
运算法则:

1.乘除的优先级大于加减
2.同优先级的运算符左大于右
3.括号内的优先级大于括号外
于是： 4+8-6/2*3 运算过程和结果唯一

3) 算符优先分析的特点：

仿效四则运算过程，预先规定相邻终结符之间的优先关系，然后利用这种优先关系来确定句型的“句柄”，并进行归约。

4) 分析器结构：



例： G[E]

E:: =E+E | E*E | (E) | i

V_t={+, *, (,), i}

这是一个二义文法，要用算符优先法分析由该文法所确定的语言句子， 如：i+i*i

(1) 先确定终结符之间的优先关系

优先关系的定义:

设 a, b 为可能相邻的终结符

定义: a = b　　a的优先级等于b

a < b　　a的优先级小于b

a > b　　a的优先级大于b

1) 例中文法终结符之间的优先关系可以用一个矩阵M来表示

a(左,栈内) \ b(右,栈外)	+	*	i	(	)	#
+	⋗	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋖	⋖	⋗	⋗
i	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	=	
)	⋗	⋗			⋗	⋗
#	⋖	⋖	⋖	⋖		

2) 矩阵元素空白处表示这两个终结符不能相邻,故没有优先关系

(2) 分析过程　　i+i*i

算法:

当栈顶项(或次栈顶项)终结符的优先级大于栈外的终结符的优先级，则进行归约，否则移进。

E:: =E+E | E*E | (E) | i

步骤	符号栈	输入串	优先关系	动作
1	#	i+i*i#	#<i	移进
2	#i	+i*i#	i>+	归约
3	#E	+i*i#	#<+	移进
4	#E+	i*i#	+<i	移进
5	#E+i	*i#	i>*	归约
6	#E+E	*i#	+<*	移进
7	#E+E*	i#	*<i	移进
8	#E+E*i	#	i>#	归约
9	#E+E*E	#	*>#	归约
10	#E+E	#	+>#	归约
11	#E	#		接受

分析过程是从符号串开始,根据相邻终结符之间的优先关系确定句型的“句柄”,并进行归约,直到识别符号E,最后分析成功:　　i+i*i∈L(G[E])

出错情况:

1. 相邻终结符之间无优先关系
2. 对双目运行符进行归约时,符号栈中无足够项
3. 非正常结束状态

重要说明

(1) 上述分析过程不一定是严格的最左归约（即不一定是规范归约）也就是每次归约不一定是归约当前句型的句柄，而是句柄的变形，但也是短语。

(2) 文法的终结符优先关系可以用一个矩阵表示,也可以用两个优先函数来表示：

f—栈内优先函数

g—栈外优先函数

若 $a \lessdot b$ 则令 $f(a) < g(b)$

$a \doteq b$ $f(a) = g(b)$

$a \gtrdot b$ $f(a) > g(b)$

根据这些原则,构造出上述文法的优先函数:

算符优先函数值的确定方法

1. 把各算符优先级由小到大定为j=0～n

	#	(	+	*	)	i
	0	0	1	2	3	

2. 对于各算符的优先顺序

   若为左结合,则 f (op)=2j　g (op)=2j-1

   若为右结合,则 f (op)=2j　g (op)=2j

   设m>2n,　则 f (j) = f (i) = m+1

   g (j) = g (i) = m，其他为0

   f(#) = f(() = g(#) = g()) = 0

	+	*	(	)	i	#
f(栈内)	2	4	0	5	5	0
g(栈外)	1	3	6	0	6	0

f(+) > g(+)　　左结合
f(+) < g(*)　　先乘后加
f(+) < g(()　　先括号内后括号外
:
:

特点:

(1) 优先函数值不唯一

(2) 优点:

- 节省内存空间
  若文法有n个终结符，则关系矩阵为n^2
  而优先函数为2n
- 易于比较：算法上容易实现,数与数比,不必查矩阵。

(3) 缺点：可能掩盖错误。

(3) 可以设立两个栈来代替一个栈

运算对象栈(OPND)　　运算符栈(OPTR)

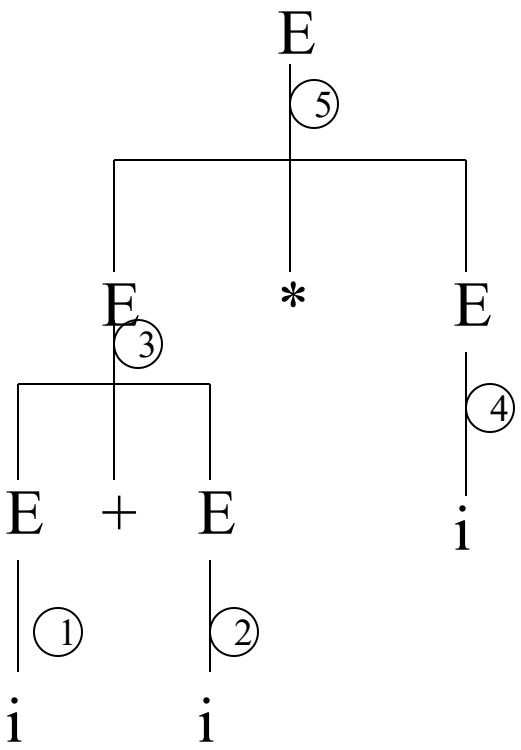
优点：便于比较,只需将输入符号与运算符栈的栈顶符号相比较

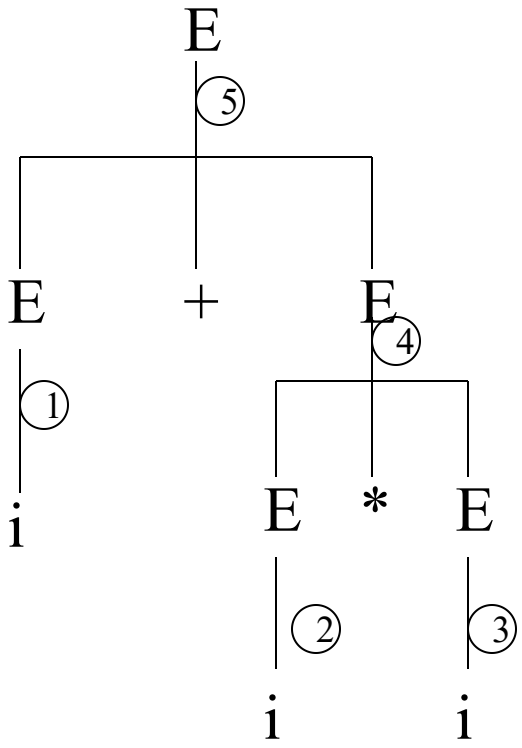
(4) 使用算符优先分析方法可以分析二义性文法所产生的语言

二义性文法按规范分析，其句柄不唯一

例：　G[E]
E:: =E+E|E*E|(E)|i
V_t={+, *, (,), i}

这是一个二义性文法,
i+i*i有两棵语法树





按规范归约,句柄不唯一, E+E*i　所以整个归约过程就不唯一,编译所得的结果也将不唯一。

4.3.3 算符优先分析法的进一步讨论

三个问题:

(1) 算符优先文法(OPG)

(2) 构造优先关系矩阵

(3) 算符优先分析算法的设计

（1）算符优先文法（OPG－Operator Precedence Grammar）

算符文法（OG）的定义

若文法中无形如U::＝·¨VW·¨的规则，这里$V,W \in V_n$
则称G为OG文法，也就是算符文法。

优先关系的定义

若G是一OG文法，$a,b \in V_t$，　$U,V,W \in V_n$

分别有以下三种情况:

1) a=b　iff 文法中有形如 U::＝·¨ab·¨或U::＝·¨aVb·¨的规则。

2) a<b　iff 文法中有形如 U::=·¨aW·¨的规则，其中 $W \overset{+}{\Rightarrow} b\cdot\text{¨}$ 或 $W \overset{+}{\Rightarrow} Vb\cdot\text{¨}$ 。

3) a>b　iff 文法中有形如 U::=·¨Wb·¨的规则，其中 $W \overset{+}{\Rightarrow} \cdot\text{¨}a$ 或 $W \overset{+}{\Rightarrow} \cdot\text{¨}aV$ 。

例：文法G[E]

E::= E + T | T

T::= T * F | F

F::= (E) | i

E::=E + T　　E => E + T　　∴ + ⋗ +

T => T * F　　∴ + ⋖ *

T =>F =>(E)　　∴ + ⋖ (

T =>F =>i　　∴ + ⋖ i

F::= (E)　　E => E + T　　∴ + ⋗)

∴ (⩵)

∴ (⋖ +

算符优先文法（OPG）的定义

设有一OG文法，如果在任意两个终结符之间，至多只有上述关系中的一种，则称该文法为算符优先文法(OPG)

对于OG文法的几点说明:

(1) 运算是以中缀形式出现的

(2) 可以证明，若文法为OG文法，则不会出现两个非终结符相邻的句型。

(3) 算法语言中的表达式以及大部分语言成分的文法均是OG文法

（2）构造优先关系矩阵

- 求 “ $\doteq$ ” 检查每一条规则，若有U::=…ab…或U::=…aVb…, 则 a=b

- 求 “ $\lessdot$ ”、“ $\gtrdot$ ”，需定义两个集合

$$FIRSTVT(U) = \{b \mid U \overset{+}{\Rightarrow} b\ldots \text{或} U \overset{+}{\Rightarrow} Vb\ldots, b \in V_t, V \in V_n\}$$

$$LASTVT(U) = \{a \mid U \overset{+}{\Rightarrow} \ldots a \text{或} U \overset{+}{\Rightarrow} \ldots aV, a \in V_t, V \in V_n\}$$

- 求“$\lessdot$”、“$\gtrdot$”：

若文法有规则
W:: =...aU... ， 对任何b, b∈FIRSTVT(U)
则有： a $\lessdot$ b

若文法有规则
W:: =...Ub... ， 对任何a, a∈LASTVT(U)
则有： a $\gtrdot$ b

构造FIRSTVT(U)的算法

1) 若有规则U::＝b…或U::＝Vb…(存在$U \overset{+}{\Rightarrow} b\ldots$或$U \overset{+}{\Rightarrow} Vb\ldots$)
则b∈FIRSTVT(U)

2) 若有规则U::＝V…且b∈FIRSTVT(V), 则b∈FIRSTVT(U)

说明:因为$V \overset{+}{\Rightarrow} b\ldots$或$V \overset{+}{\Rightarrow} Wb\ldots$, 所以有$U \Rightarrow V\ldots \overset{+}{\Rightarrow} b\ldots$或
$U \Rightarrow V\ldots \overset{+}{\Rightarrow} Wb\ldots$

具体方法如下：

设一个栈S和一个二维布尔数组F

F[U,b]=TRUE　iff　b∈FIRSTVT(U)

```
PROCEDURE   INSERT(U,b)
          IF  NOT  F[U,b]   THEN
             BEGIN
               F[U,b]:=TRUE;
               把(U,b)推进S栈     /* b∈FIRSTVT(U) */
             END
BEGIN  {main}
     FOR  每个非终结符号U和终结符b   DO
               F[U,b]:=FALSE;
     FOR 每个形如U::=b…或U::=Vb… 的规则  DO
               INSERT(U,b);
```

```
WHILE   S栈非空      DO
    BEGIN
        把S栈的栈顶项弹出,记为 (V,b)  /* b∈FIRSTVT(V)*/
        FOR    每条形如U::=V…的规则   DO
                    INSERT (U,b);   /* b∈FIRSTVT(U)*/
    END OF WHILE
END
```

上述算法的工作结果是得到一个二维的布尔数组F,从F可以得到任何非终结符号U的FIRSTVT

$$FIRSTVT(U) = \{ b \mid F[U,b] = TRUE \}$$

构造LASTVT(U)的算法

1. 若有规则U::=…a或U::=…aV,则a∈LASTVT(U)

2. 若有规则U::=…V,且a∈LASTVT(V), 则a∈LASTVT(U)

设一个栈ST,和一个布尔数组B

```
PROCEDURE    INSERT(U,a)
        IF  NOT  B[U,a]    THEN
            BEGIN
                B[U,a]::=TRUE; 把(U,a)推进ST栈;
            END;
```

```
BEGIN
    FOR 每个非终结符号U和终结符号a    DO
                B[U,a]:=FALSE;
    FOR  每个形如U::=…a或U::=…aV的规则  DO
                INSERT (U,a);
    WHILE  ST栈非空   DO
        BEGIN
            把ST栈的栈顶弹出,记为(V,a);
            FOR 每条形如U::=…V的规则    DO
                INSERT(U,a);
        END  OF    WHILE;
END;
```

构造优先关系矩阵的算法

FOR 每条规则$U::=x_1x_2\ldots x_n$ DO
  FOR i:=1 TO n-1 DO
    BEGIN
      IF x_i和x_{i+1}均为终结符,THEN 置 $x_i=x_{i+1}$
      IF $i\leq n-2$，且x_i和x_{i+2}都为终结符号但
        x_{i+1}为非终结符号 THEN 置 $x_i=x_{i+2}$
      IF x_i为终结符号,x_{i+1}为非终结符号 THEN
        FOR FIRSTVT(x_{i+1})中的每个b DO
          置$x_i<b$
      IF x_i为非终结符号,x_{i+1}为终结符号 THEN
        FOR LASTVT(x_i)中的每个a DO
          置$a>x_{i+1}$
    END

（3）算符优先分析算法的实现

先定义优先级，在分析过程中通过比较相邻运算符之间的优先级来确定句型的“句柄”并进行归约。

？ --最左素短语

[定义] 素短语：文法G的句型的素短语是一个短语，它至少包含有一个终结符号，并且除它自身以外不再包含其他素短语。

例：文法G[E]

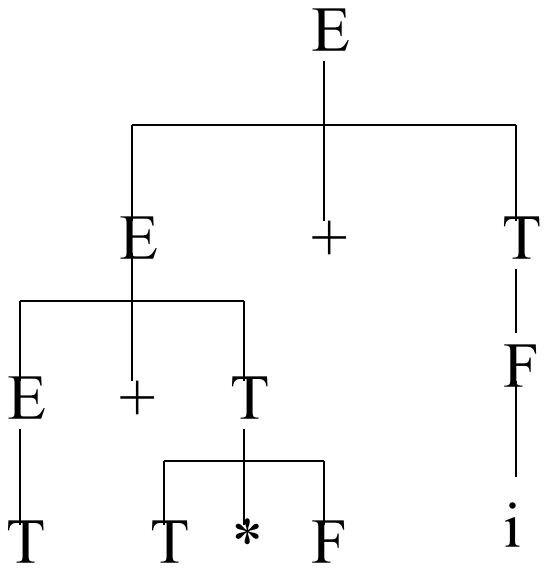
E::=E+T|T

T::=T*F|F

F::=(E)|i

求句型T+T*F+i 的素短语

文法的语法树：



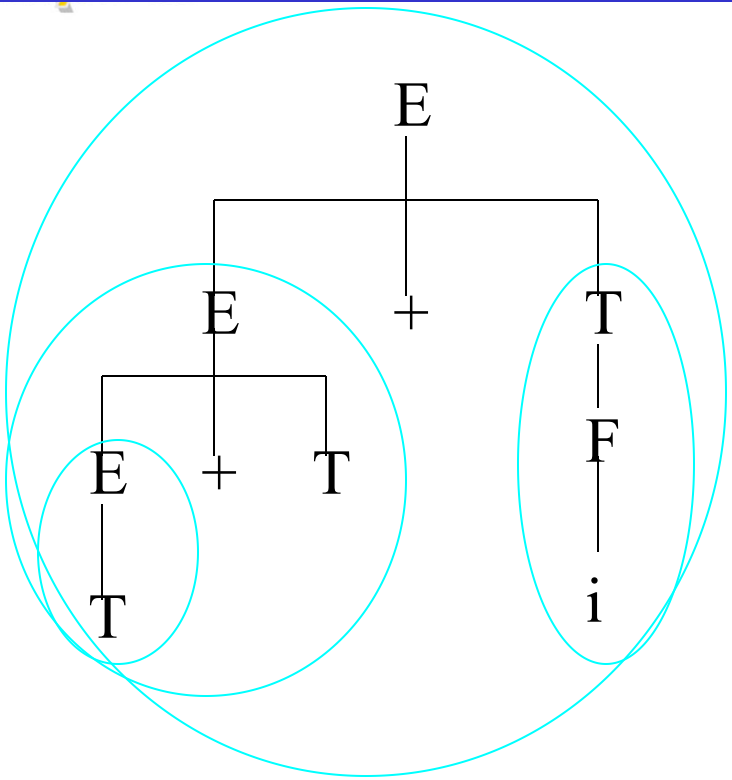
短语:T+T*F+i, T+T*F

T(最左), T*F, i

其中T不包含终结符,T是句柄

而T+T*F+i和T+T*F包含其他素短语。

只有T*F和i为素短语,其中T*F为最左素短语,而该句型句柄为T。



句型：T + T + i

短语：T+T+i

T +T

T

i

句柄：T

素短语：T+T, i

算符优先分析法如何确定当前句型的最左素短语？

设有OPG文法句型为:

$\#N_1a_1N_2a_2\ldots N_na_nN_{n+1}\#$

其中N_i为非终结符(可以为空), a_i为终结符

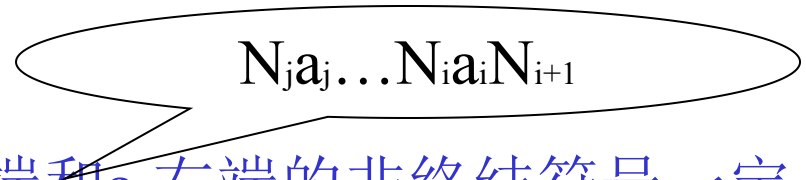
定理：一个OPG句型的最左素短语是满足下列条件的最左子串：$a_{j-1}N_ja_j\ldots N_ia_iN_{i+1}a_{i+1}$

其中 $a_{j-1}<a_j$

$a_j=a_{j+1}$, $a_{j+1}= a_{j+2}$,…, $a_{i-2}= a_{i-1}$, $a_{i-1}= a_i$

$a_i> a_{i+1}$

根据该定理,要找句型的最左素短语就是要找满足上述条件的最左子串。



★注意:出现在a_j左端和a_i右端的非终结符号一定属于这个素短语,因为运算是中缀形式给出的(OPG文法的特点)NaNaNaN ⇒NaWaN

例: 文法G[E]

E::=E+T|T

T::=T*F|F

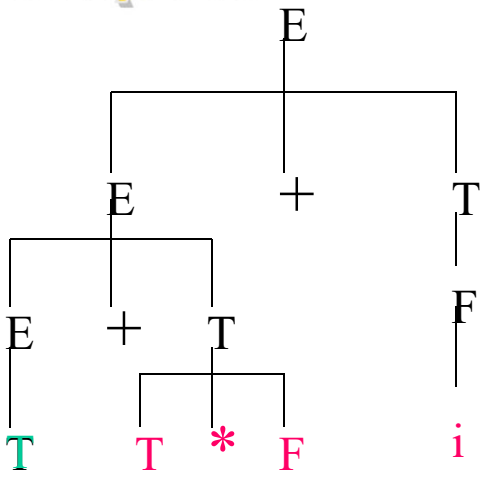
F::=(E)|i

分析文法的句型T+T*F+i

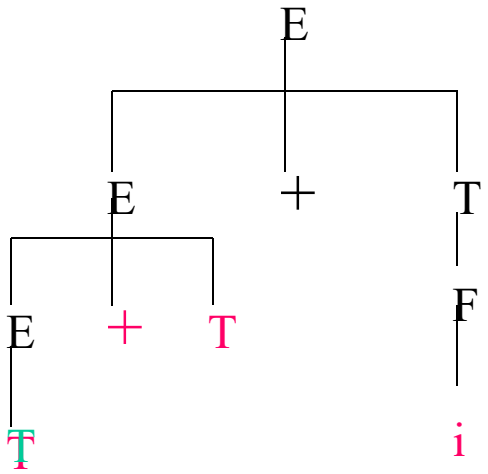
步骤	句型	关系	最左子串	归约符号
1	#T+T*F+i#	#<+<*>+<i>>#	T*F	T
2	#T+T+i#	#<+>+<i>#	T+T	E
3	#E+i#	#<+<i>#	i	F
4	#E+F#	#<+>#	E+F	E

可以看出：

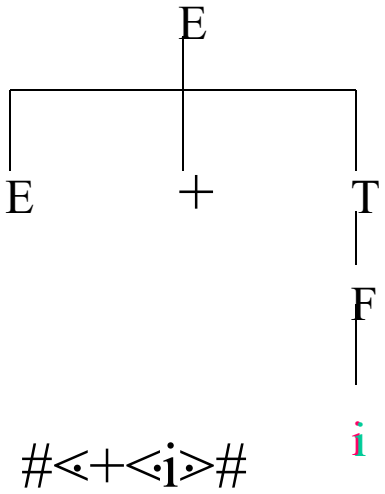
1. 每次归约最左子串,确实是当前句型的最左素短语(语法树）
2. 归约的不都是真句柄（仅i归约为F是句柄，但它是最左素短语)
3. 没有完全按规则进行归约,因为素短语不一定是简单短语



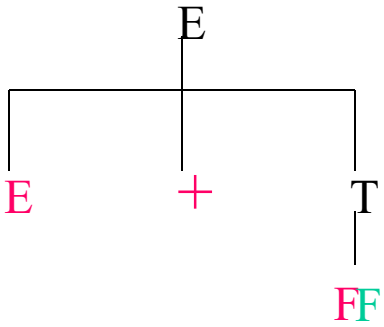
#<+<*>+<i>#



#<+>+<i>#



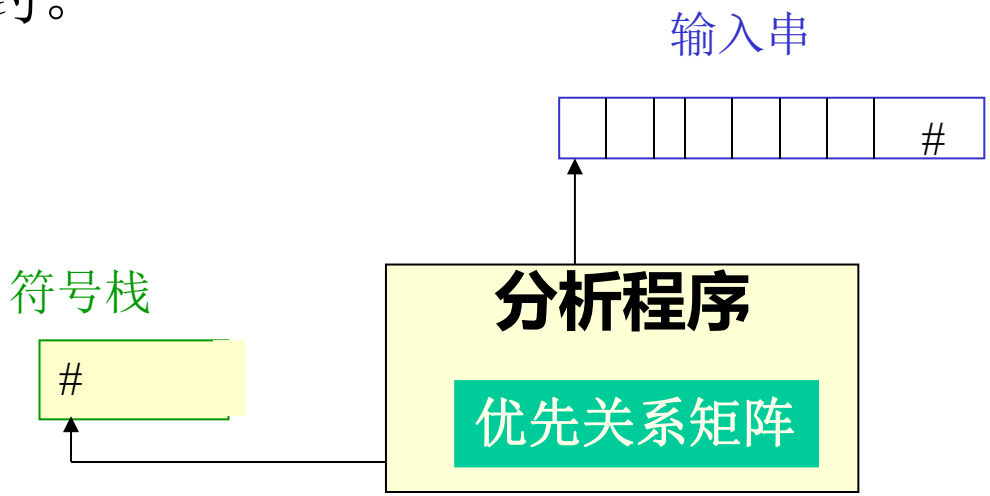
#<+<i>#



#<+>#

算符优先分析法的实现：

基本部分是找句型的最左子串（最左素短语）并进行归约。



当栈内终结符的优先级＜＝栈外的终结符的优先级时，移进；栈内终结符的优先级＞栈外的终结符的优先级时，表明找到了素短语的尾，再往前找其头，并进行归约。

4.3.4 LR分析法

1、概述

什么是LR分析：从左到右扫描(L)自底向上进行归约(R)(是规范归约），是自底向上分析方法的高度概括和集中

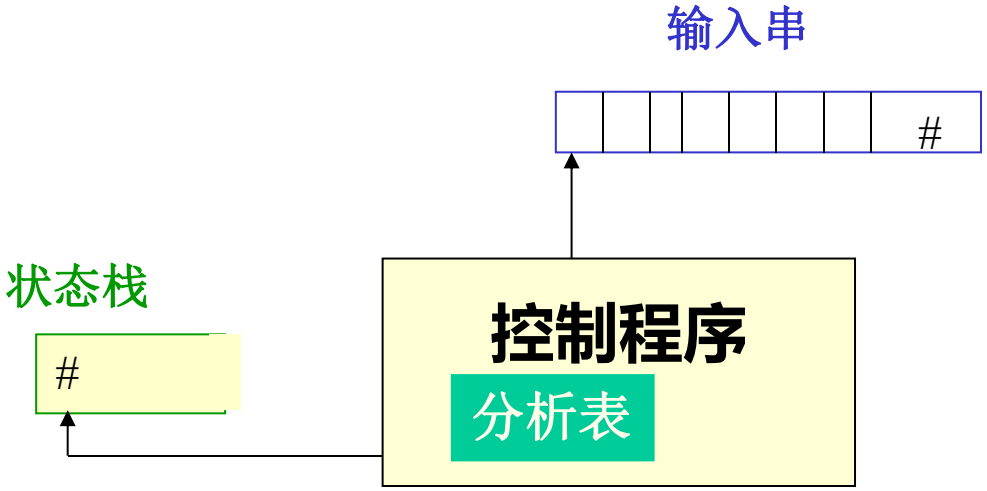
历史 + 展望 + 现状 => 句柄

(1) LR分析法的优缺点:

1) 适合文法类足够大
2) 分析效率高
3) 报错及时
4) 可以自动生成
5) 手工实现工作量大

(2) LR分析器有三部分：状态栈　分析表　控制程序

状态栈：放置分析器状态和文法符号。



分析表：由两个矩阵组成，其功能是指示分析器的动作，是移进还是归约，根据不同的文法类要采用不同的构造方法。

控制程序：执行分析表所规定的动作，对栈进行操作。

(3) 分析表的种类

a) SLR分析表(简单LR分析表)

构造简单,最易实现,大多数上下文无关文法都可以构造出SLR分析表,所以具有较高的实用价值。使用SLR分析表进行语法分析的分析器叫SLR分析器。

b) LR分析表(规范LR分析表)

适用文法类最大,几乎所有上下文无关文法都能构造出LR分析表, 但其分析表体积太大，实用价值不大。

c) LALR分析表(超前LR分析表)

这种表适用的文法类及其实现上难易在上面
两种之间,在实用上很吸引人。
使用**LALR**分析表进行语法分析的分析器叫**LALR**分析器。

例：**UNIX----YACC**

(4) 几点说明

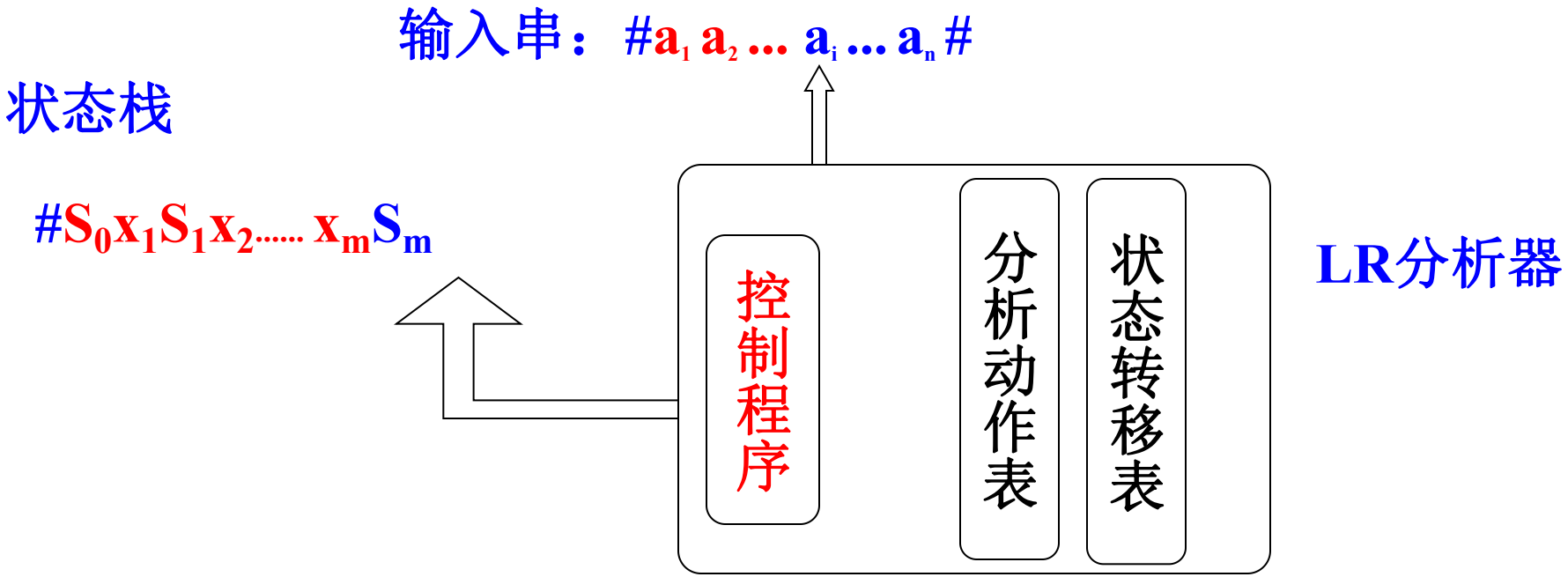
1. 三种分析表对应三类文法

2. 一个SLR文法必定是LALR文法和LR文法

3. 仅讨论SLR分析表的构造方法

2、LR分析

（1）逻辑结构



$\#S_0x_1S_1x_2\cdots\cdots x_mS_m \Longrightarrow \dfrac{S_0S_1\cdots\cdots S_m}{\#\ x_1x_2\cdots\cdots x_m}$

状态栈： $S_0,S_1,\ldots,S_m$ 状态

S_0---初始状态　　S_m---栈顶状态

栈顶状态概括了从分析开始到该状态的全部分析历史和展望信息。

符号串：$x_1x_2\cdots\cdots x_m$

为从开始状态(S_0)到当前状态(S_m)所识别的规范句型的活前缀。

规范句型: 通过规范归约得到的句型。

规范句型前缀: 将输入串的剩余部分与其连接起来就构成了规范句型。

如： $x_1x_2\ldots x_m a_i \ldots a_n$为规范句型

活前缀: 若分析过程能够保证栈中符号串均是规范句型的前缀，则表示输入串已分析过的部分没有语法错误，所以称为规范句型的活前缀。

规范句型的活前缀：

对于句型 $\alpha\ \beta\ t$，β 表示句柄,如果 $\alpha\ \beta = u_1u_2\ldots u_r$ 那么符号串 $u_1u_2\ldots u_i(1\le i\le r)$ 即是句型 $\alpha\ \beta\ t$ 的活前缀

例: 有文法G[E]: E→T|E+T|E-T
T→i|(E)
拓广文法G'[S]: S→E#
E→T|E+T|E-T
T→i|(E)
句型E-(i+i) #
活前缀：E, E-, E-(, E-(i 是句型E-(i+i)# 的活前缀。

- 分析表

a. 状态转移表 (GOTO表)

是一个矩阵：
行---分析器的状态
列---文法符号

GOTO表

状态＼符号	E	T	F	i	+	*	(	)	#
S_0									
S_1									
S_2									
:									
S_n									

GOTO表

状态＼符号	E	T	F	i	+	*	(	)	#
S_0									
S_1									
S_2									
:									
S_n									

$GOTO[S_{i-1}, x_i]=S_i$

S_{i-1}---当前状态(栈顶状态)

x_i--- 新的栈顶符号

S_i----新的栈顶状态(状态转移)

$\#S_0x_1S_1x_2\ldots\ldots x_{i-1}S_{i-1}\ x_iS_i$

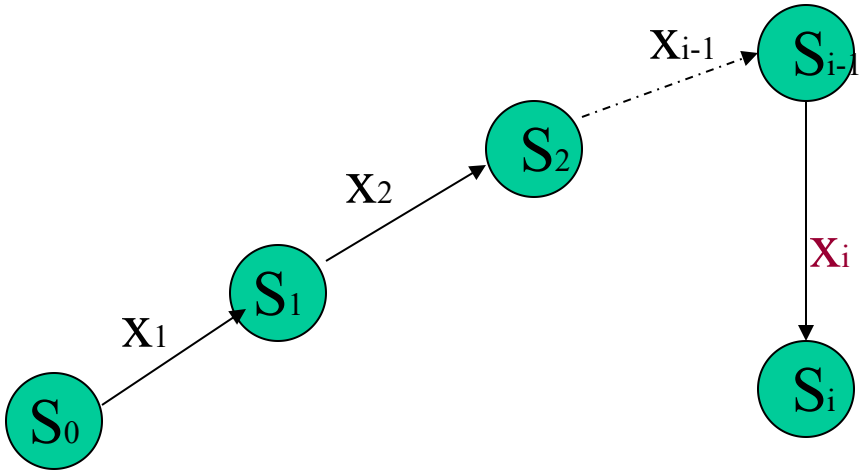
S_i需要满足条件是:

若$X_1\ X_2\ldots.\ X_{i-1}$是由S_0到S_{i-1}所识别的规范句型的活前缀,则$X_1\ X_2\ldots.\ X_i$是由S_0到S_i所识别的规范句型的活前缀。

通过对有穷自动机的了解, 可以看出:

状态转移函数GOTO是定义了一个以文法符号集为字母表的有穷自动机，该自动机识别文法所有规范句型的活前缀。

$M=(S,\ V,\ GOTO,\ S_0,\ Z)$



b. 分析动作表(ACTION表)

ACTION表

输入符号a / 状态s	+	*	i	(	)	#
S_0						
S_1						
S_2						
:						
S_n						

ACTION[S_i,a] = 分析动作　　$a \in V_t$

分析动作：

(1) 移进(shift)

$ACTION[S_i,a] = s$

动作：将a推进栈，并设置新的栈顶状态S_j

$S_j = GOTO[S_i,a]$，将指针指向下一个输入符号

(2) 归约(reduce)

$ACTION[S_i,a] = r_d$

d：文法规则编号　　(d) $A \rightarrow \beta$

动作：将符号串 β (假定长度为n)连同状态从栈内弹出，把A推进栈，并设置新的栈顶状态S_j

$S_j = GOTO[S_{i-n},A]$

(3) 接受(accept)

$ACTION[S_i,\#]=accept$

(4) 出错(error)

$ACTION[S_i,a]=error$

控制程序: (Driver Routine)

1、 根据栈顶状态和现行输入符号，查分析动作表(ACTION表)，执行由分析表所规定的操作；

2、并根据GOTO表设置新的栈顶状态(即实现状态转移)。

（2） LR分析过程

例：文法G[E]

(1) E ::= E+T

(2) E ::= T

(3) T ::= T*F

(4) T ::= F

(5) F ::= (E)

(6) F ::= i

该文法是SLR文法,故可以构造出SLR分析表(ACTION表和GOTO表)

GOTO表

状态 \ 文法符号	E	T	F	i	+	*	(	)
0(S_0)	1	2	3	5			4	
1(S_1)					6			
2(S_2)						7		
3(S_3)								
4(S_4)	8	2	3	5			4	
5(S_5)								
6(S_6)		9	3	5			4	
7(S_7)			10	5			4	
8(S_8)					6			11
9(S_9)						7		
10(S_{10})								
11(S_{11})								

ACTION 表　　GOTO 表

状态 \ 输入符号	i	+	*	(	)	#	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

分析过程 i*i+i

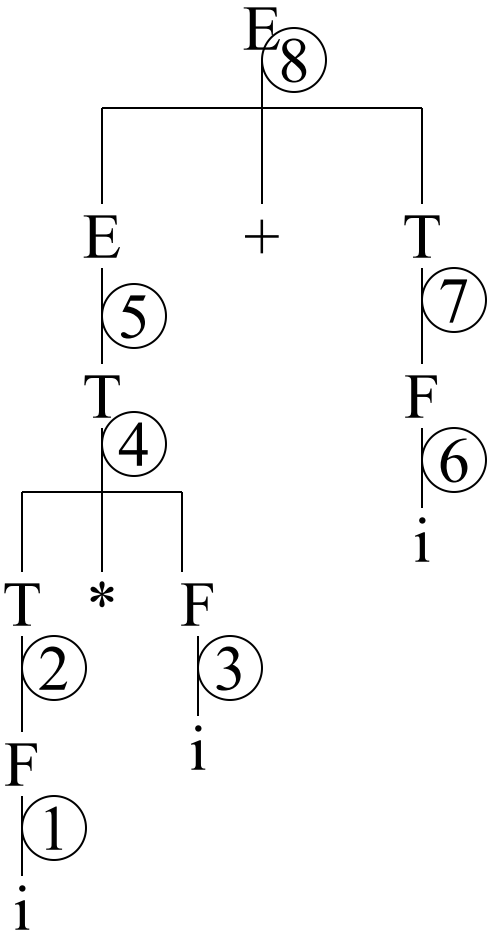
步骤	状态栈	符号	输入串	动作
1	#0	#	i*i+i#	初始化
2	#0i5	#i	*i+i#	S
3	#0F3	#F	*i+i#	r6
4	#0T2	#T	*i+i#	r4
5	#0T2*7	#T*	i+i#	S
6	#0T2*7i5	#T*i	+i#	S
7	#0T2*7F10	#T*F	+i#	r6

8	#0T2	#T	+i#	r3
9	#0E1	#E	+i#	r2
10	#0E1+6	#E+	i#	S
11	#0E1+6i5	#E+i	#	S
12	#0E1+6F3	#E+F	#	r6
13	#0E1+6T9	#E+T	#	r4
14	#0E1	#E	#	r1
15		#E		accept

由分析过程可以看到:

(1) 每次归约总是归约当前句型的句柄,是规范归约。
(算符优先分析归约最左素短语)

(2) 分析的每一步栈内符号串均是规范句型的活前缀,与输入串的剩余部分构成规范句型。



3、构造SLR分析表

构造LR分析器的关键是构造其分析表。

构造LR分析表的方法是:

（1）根据文法构造识别规范句型活前缀的有穷自动机DFA

（2）由DFA构造分析表

(1) 构造DFA

① DFA 是一个五元式

$$M=(S,\ V,\ GOTO,\ S_0,\ Z)$$

S：有穷状态集

在此具体情况下，S = LR(0)，项目集规范族。

项目集规范族：其元素是由项目所构成的集合。

V：文法字汇表

S_0：初始状态 $S_0 \in S$

GOTO: 状态转移函数

$GOTO[S_i,X]=S_j$

$S_i, S_j \in S$　　S_i, S_j为项目集合

$X \in V_n \cup V_t$

表示当前状态S_i面临文法符号为X时，应将状态转移到S_j

Z：终态集合　$Z = S-\{S_0\}$

即除S_0以外，其余全部是终态

构造DFA：

一、确定**S**集合，即**LR（0）项目集规范族**，同时确定$\mathbf{S_0}$

二、确定**状态转移函数GOTO**

② 构造LR（0）的方法

LR(0) 是DFA的状态集,其中每个状态又都是项目的集合。

项目:文法G的每个产生式(规则)的右部添加一个圆点就构成一个项目。

例:产生式:A→XYZ

项　目: A→.XYZ
A→X.YZ
A→XY.Z
A→XYZ.

项目的直观意义：指明在分析过程中的某一时刻已经归约的部分和等待归约部分。

产生式:A→ ε

项　目: A→.

构造LR(0)的方法(三步)

1) 将文法拓广

目的：使构造出来的分析表只有一个接受状态,这是为了实现的方便。

方法：修改文法，使识别符号的规则只有一条。

G[E]
E→…
E→…
…

拓广 ⇒

G'[E']
E'→E
E→…
E→…
…

L(G(E))=L(G'[E'])

2) 根据文法列出所有的项目

3) 将有关项目组合成集合，即DFA中的状态；
所有状态再组合成一个集合，即LR（0）项目集规范族

通过一个具体例子来说明LR(0)的构造以及DFA的构造方法。

例：G[E]

E→E+T | T

T→T*F | F

F→(E) | i

① 将文法拓广为G'[E']

(0) E'→E
(1) E→E+T
(2) E→T
(3) T→T*F
(4) T→F
(5) F→(E)
(6) F→i

② 列出文法的所有项目

(1) E'→.E
(2) E'→E.
(3) E→.E+T
(4) E→E.+T
(5) E→E+.T
(6) E→E+T.
(7) E→.T
(8) E→T.
(9) T→.T*F
(10) T→T.*F
(11)T→T*.F
(12)T→T*F.
(13)T→.F
(14)T→F.
(15)F→.(E)
(16)F→(.E)
(17)F→(E.)
(18)F→(E).
(19)F→.i
(20) F→i.

③ 将有关项目组成项目集,所有项目集构成的集合即为LR(0)

为实现这一步，先定义：

- 项目集闭包closure
- 状态转移函数GOTO

A.项目集闭包closure的定义和计算:

令I是文法G'的任一项目集合，定义closure(I)为项目集合I的闭包，可用一个过程来定义并计算closure(I)：

```
Procedure  closure(I);
  begin
        将属于I的项目加入closure(I);
         repeat
             for  closure(I)中的每个项目A→α.Bβ (B∈Vn)   do
                  将B→.r(r∈V* )加入closure(I)
        until   closure(I)不再增大
  end
```

B. 状态转移函数GOTO的定义:

GOTO(I,X) = closure(J)

I：项目集合

X：文法符号，X∈V

J：项目集合

J = { 任何形如A→ α X. β 的项目| A→ α .X β ∈I }

closure(J):项目集J的闭包, 仍是项目集合

所以,GOTO(I,X) = closure(J) 的直观意义是:

它规定了识别文法规范句型活前缀的DFA, 从状态I(项目集)出发,经过X弧所应该到达的状态(项目集合)

例:

I={E'→E. , E→E.+T} 求GOTO(I , +) =?

GOTO(I , +) = closure(J)

∵J = {E→E+.T}

∴GOTO(I , +) = {E→E+.T , T→.T*F , T→.F ,
F→.(E) , F→.i }

LR(0)和GOTO的构造算法:

```
G'→LR(0), GOTO
Procedure   ITEMSETS(G')
   begin
        LR(0) := {closure({E'→.E})};
        repeat
           for  LR(0)中的每个项目集I和G'的每个符号X  do
              if  GOTO(I , X)非空,且不属于LR(0)
              then   把GOTO(I , X)放入LR(0)中
        until   LR(0)不再增大
   end
```

例:求G'[E']的LR(0)　　V={E, T, F, i, +, *, (,)}

G'[E']共有20个项目

LR(0)={I_0,I_1,I_2,…I_{11}}　　由12个项目集组成:

I_0:
- E'→.E
- E→.E+T
- E→.T
- T→.T*F
- T→.F
- F→.(E)
- F→.i

closure({E'→.**E**}) = I_0

I_1:
- E'→E.
- E→E.+T

GOTO(I_0,E) = closure({E'→E.
E→E.+T})
= I_1

I_2: $E \rightarrow T.$ $GOTO(I_0,T) = closure(\{E \rightarrow T.\ \ T \rightarrow T.*F\}) = I_2$

$T \rightarrow T.*F$

I_3: $T \rightarrow F.$ $GOTO(I_0,F) = closure(\{T \rightarrow F.\}) = I_3$

I_4: $F \rightarrow (.E)$ $GOTO(I_0,() = closure(\{F \rightarrow (.E)\}) = I_4$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .i$

I_5: $F \rightarrow i.$ $GOTO(I_0,i) = closure(\{F \rightarrow i.\}) = I_5$

$GOTO(I_0,*) = \phi$

$GOTO(I_0,+) = \phi$

$GOTO(I_0,)) = \phi$

I_6:
- E→E+.T
- T→.T*F
- T→.F
- F→.(E)
- F→.i

GOTO(I_1,+) = closure({E→E+.T}) = I_6
GOTO(I_1,其他符号)为空

I_7:
- T→T*.F
- F→.(E)
- F→.i

GOTO(I_2,*) = closure({T→T*.F}) = I_7
GOTO(I_2,其他符号)为空
GOTO(I_3,所有符号)为空

I_8: $\begin{cases} F \rightarrow (E.) \\ E \rightarrow E.+T \end{cases}$

$GOTO(I_4,E) = closure(\{F \rightarrow (E.), E \rightarrow E.+T\}) = I_8$

$GOTO(I_4,T) = I_2 \in LR(0)$

$GOTO(I_4,F) = I_3 \in LR(0)$

$GOTO(I_4,() = I_4 \in LR(0)$

$GOTO(I_4,i) = I_5 \in LR(0)$

$GOTO(I_4,+) = \phi$

$GOTO(I_4,*) = \phi$

$GOTO(I_4,)) = \phi$

$GOTO(I_5,$所有符号$) = \phi$

I_9: $E \rightarrow E+T.$
$T \rightarrow T.*F$

$GOTO(I_6,T) = closure(\{E \rightarrow E+T., T \rightarrow T.*F\}) = I_9$

$GOTO(I_6,F) = I_3$

$GOTO(I_6,() = I_4$

$GOTO(I_6,i) = I_5$

I_{10}:T→T*F.　　GOTO(I_7,F) = closure({T→T*F .}) = I_{10}

GOTO(I_7,() = I_4

GOTO(I_7,i) = I_5

I_{11}:F→(E).　　GOTO(I_8,)) =closure({F→(E) .}) = I_{11}

GOTO(I_8,+) = I_6

GOTO(I_9,*)= I_7

GOTO(I_{10},所有符号) = Φ，　GOTO(I_{11},所有符号) = Φ

③ 构造DFA

$M = (S, V, GOTO, S_0, Z)$

$S = \{I_0, I_1, I_2, \ldots, I_{11}\} = LR(0)$

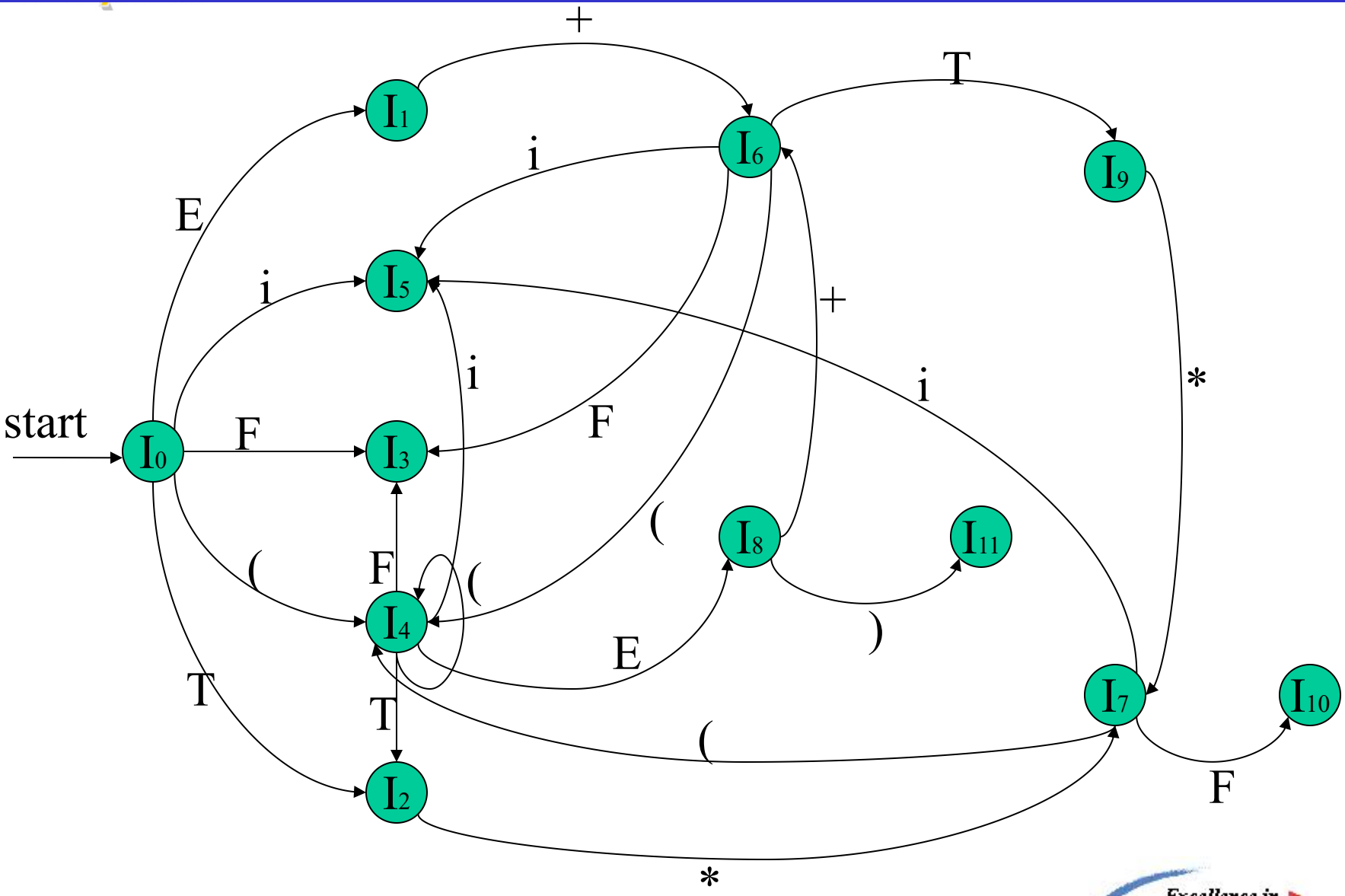
$V = \{+, *, i, (,), E, T, F\}$

$GOTO(I_m, X) = I_n$

$S_0 = I_0$

$Z = S-\{I_0\} = \{I_1, I_2, \ldots, I_{11}\}$

M的图解表示如下:


+
T
I1
I6
I9
E
i
i
I5
+
i
*
start
I0
F
I3
F
(
I8
I11
F
(
I4
E
)
T
T
I7
I10
(
F
I2
*

关于自动机的说明:

①除I_0以外,其余状态都是终态,从I_0到每一状态的每条路径都识别和接受一个规范句型的活前缀

如对文法句子i+i*i 进行规范归约所得到的规范句型的活前缀都可以由该自动机识别,如:

I_0~I_1 识别规范句型的活前缀E(+i*i)

I_0~I_6 识别规范句型的活前缀E+(i*i)

I_0~I_7 识别规范句型的活前缀E+T*(i)

I_0~I_9 识别规范句型的活前缀E+T(*i)

I_0~I_{10} 识别规范句型的活前缀E+T*F

② 状态中每个项目对该状态能识别的活前缀都是有效的。

有效项目定义:若项目$A\rightarrow \beta_1 . \beta_2$对活前缀$\alpha\ \beta_1$有效，其条件是存在规范推导

$$E' \overset{*}{\Rightarrow} \alpha Aw \Rightarrow \alpha\ \beta_1 \beta_2 w$$

其中$\alpha, \beta 1, \beta 2 \in V^*, w \in Vt^*$

注意：项目中圆点前的符号串成为活前缀的后缀

③ 有效项目能预测分析的下一步动作：

$E\rightarrow E+T.$ 表示已将输入串归约为E+T,下一步应该将E+T归约为E

$$E' \overset{*}{\Rightarrow} (E) \Rightarrow (E+T)$$

$T\rightarrow T.*F$ 表示已将输入串归约为T,下一步动作是移进输入符号*

注意：经移进或归约后,在栈内仍是规范句型的活前缀

④ DFA中的状态,既代表了分析历史又提供了展望信息

每条规范句型的活前缀都代表了一个确定的的规范归约过程,故由状态可以代表分析历史。

由于状态中的项目都是有效项目,所以提供了下一步可能采取的动作。

历史+展望+现实⇒ 句柄

（2）由DFA构造SLR分析表

* GOTO表在求LR（0）时已求出

GOTO表

状态 \ 文法符号	E	T	F	i	+	*	(	)
0(S_0)	1	2	3	5			4	
1(S_1)					6			
2(S_2)						7		
3(S_3)								
4(S_4)	8	2	3	5			4	
5(S_5)								
6(S_6)		9	3	5			4	
7(S_7)			10	5			4	
8(S_8)					6			11
9(S_9)						7		
10(S_{10})								
11(S_{11})								

* 求ACTION表

设k为状态编号,E为原文法识别符号，
E’为扩充文法识别符号

1、求出文法每个非终结符的FOLLOW集合

2、若项目$A \rightarrow \alpha .a\beta \in k$,且$a \in V_t$,则置
ACTION[k,a] = s (移进）

3、若项目A→ α .∈k, 那么对输入符号a,若a∈FOLLOW(A),则置ACTION[k,a]=r_j
其中A→ α 为文法G'的第j个产生式。

4、若项目E'→E.∈k, 则置
ACTION[k , #] =accept

5、ACTION表中不能用步骤2~4填入信息的空白格，均置error

在状态中可有三种类型的项目,其中只有两种有移进或归约动作:

$A \rightarrow \alpha .a \beta$	$a \in V_t$	移进项目	分析动作:移进
$A \rightarrow \alpha .$		归约项目	分析动作:归约
$A \rightarrow \alpha .B \beta$	$B \in V_n$	待约项目	无动作

根据上述算法,可以构造出文法G'[E']的ACTION

对文法G'[E']

k=2 (I_2)
有效项目 E→T.
T→T.*F
FOLLOW(E)={#, +,)}

k=1 (I_1)
E'→E.
E→E.+T
FOLLOW(E')={#}

k=0 (I_0)
E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.i

根据算法造出的ACTION表为:

ACTION表

输入符号a / 状态s	i	+	*	(	)	#
0	S_5			S_4		
1		S_6				accpet
2		r_2	S_7		r_2	r_2
3						
4						
5						
6						
7						
8						
9						
10						
11						

两点说明:

1. 由DFA构造出的SLR分析表,在造表时, 只需向前看一个符号就能确定分析的动作是移进还是归约，所以称为SLR(1)分析表，简称SLR分析表，使用SLR分析表的分析器叫SLR分析器。
2. 对文法G,若应用上述算法所造出的分析表具有多重定义入口,分析动作不唯一, 则文法G就不是SLR的，需要用别的方法来构造分析表。

复 习

语法分析方法：
- 自顶向下分析法 $Z \overset{+}{\Rightarrow} S$
- 自底向上分析法 $S \overset{+}{\Leftarrow} Z$

$S \in L[Z]$

(一) 自顶向下分析

① 概述自顶向下分析的一般过程

存在问题：
- 左递归问题 ——— 消除左递归的方法
- 回溯问题 ———
  - 无回溯的条件
  - 改写文法
  - 超前扫描

② 两种常用方法:

(1) 递归子程序法
- a) 改写文法,消除左递归,回溯
- b) 写递归子程序

(2) LL(1)分析法
- LL(1)分析器的逻辑结构及工作过程
- LL(1)分析表的构造方法
  - 1.构造FIRST集合的算法
  - 2.构造FOLLOW集合的算法
  - 3.构造分析表的算法
- LL(1)文法的定义以及充分必要条件

(二) 自底向上分析

归约过程:

(1)一般过程:　　移进-归约过程

问题:如何寻找句柄

(2)算法:

i)算符优先分析法:

1.分析器的构造,分析过程

根据算符优先关系矩阵来决定是移进还是归约。

2.算符优先法的进一步讨论

1) 适用的文法类----------引出的算符优先法的定义
2) 优先关系矩阵的构造
3) 什么是“句柄”,如何找
   由句柄引出的最左素短语的概念。
   最左素短语的定理,如何找。

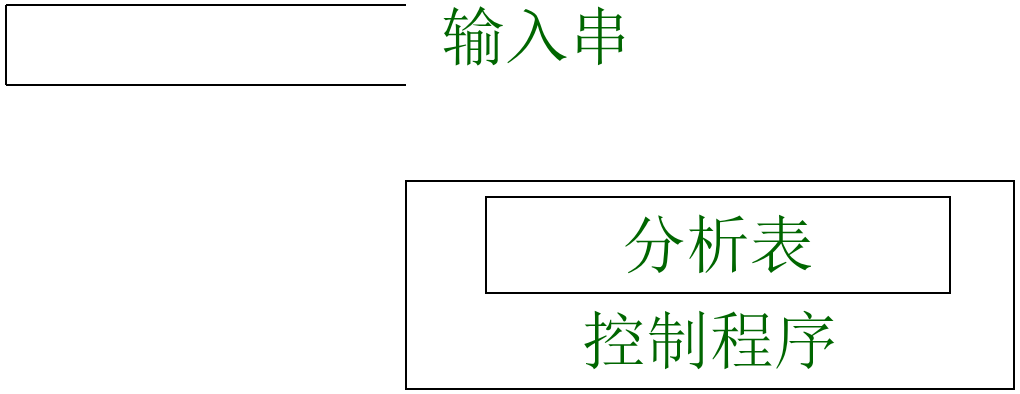
ii)LR分析法

1.概述----概念、术语　(活前缀、项目)

- 2.LR分析
  - 1)逻辑结构
    - 状态栈
    - 分析表
      - GOTO表
      - 分析动作表
    - 控制程序
  - 2)分析过程

- 3.如何构造SLR分析表
  - 1)构造DFA
  - 2)由DFA构造分析表

除了递归子程序法，其他几种方法逻辑结构很象：



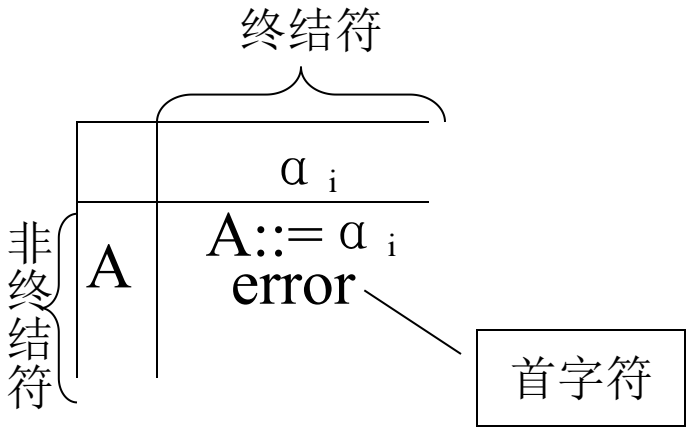
(1) 对于LL(1)分析法

符号栈

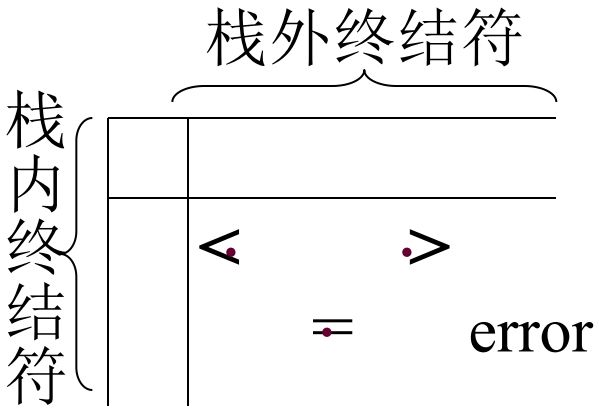
α_i

(自顶向下，保证最左推导)

LL(1)分析表



(2)对于算法优先分析:　符号栈



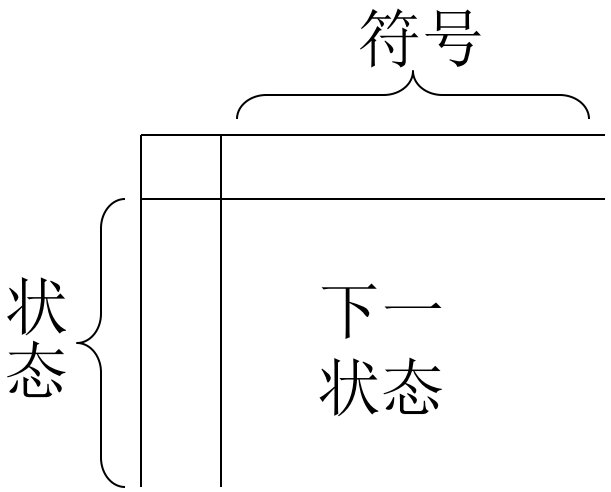
(3)LR分析:

符号栈

$S_0,S_1 \ldots S_m$
$X_1 \ldots X_m$

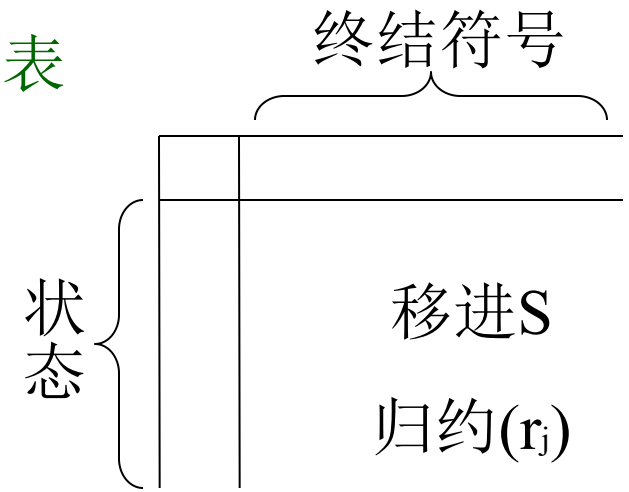
分析表 { 状态转移GOTO表；分析动作表

GOTO表



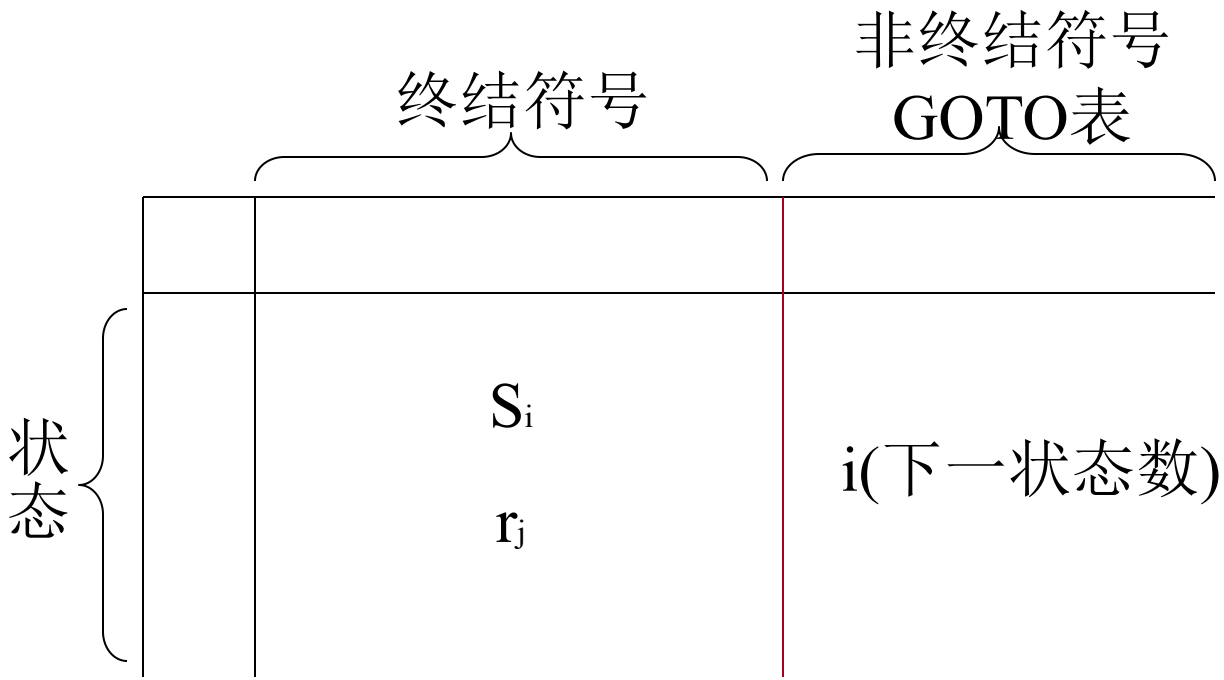
根据栈顶状态和栈顶符号推导出下一状态

分析动作表



根据栈顶状态和输入符号推导出下一动作

将GOTO表和分析动作表压缩后得:



第五章　符号表管理技术

- 概述
- 符号表的组织与内容
- 非分程序结构语言的符号表组织
- 分程序结构语言的符号表组织

5.1 概述

(1) 什么是符号表?

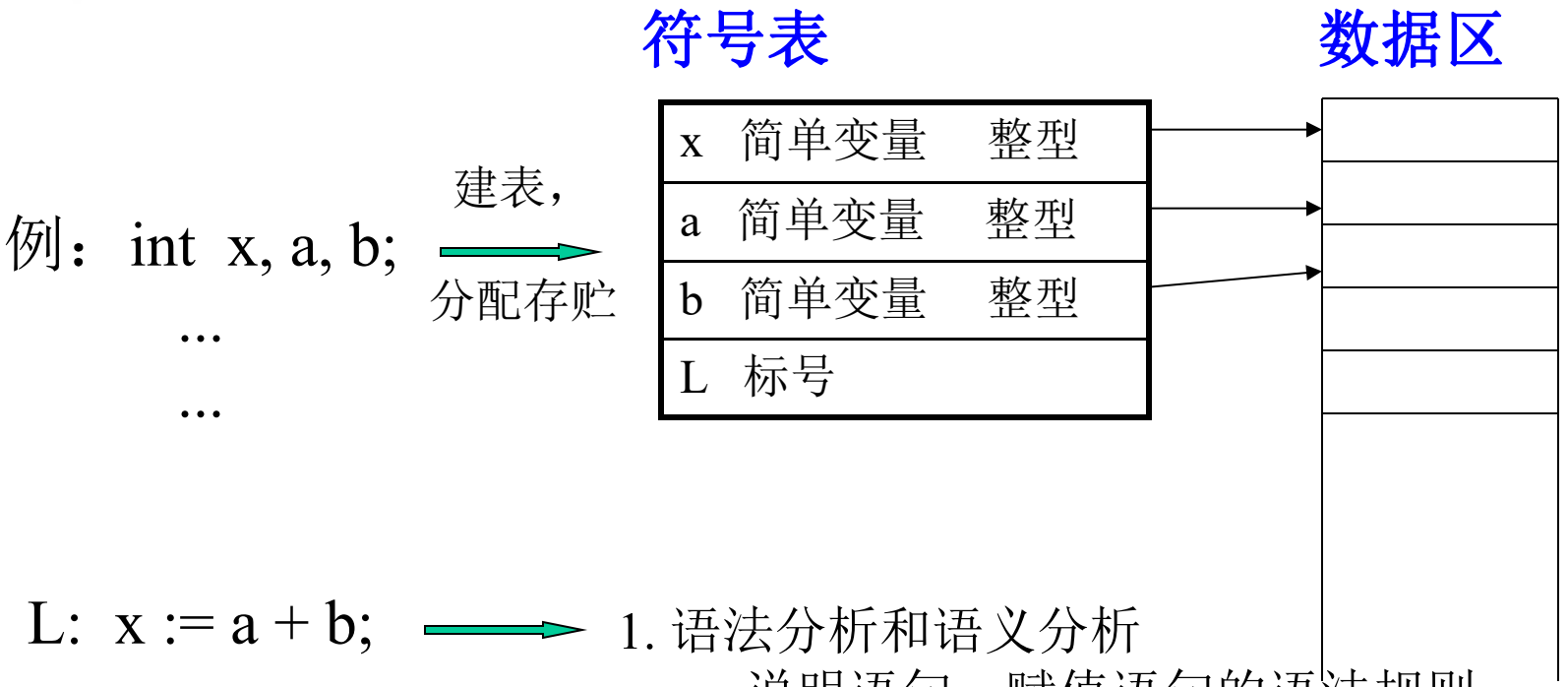
在编译过程中,编译程序用来记录源程序中各种名字的特性信息, 所以也称为名字特性表。

名　　字: 程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等。

特性信息: 上述名字的种类、类型、维数、参数个数、数值及目标地址（存储单元地址）等。

(2)建表和查表的必要性(符号表在编译过程中的作用)

- 源程序中变量要先声明，然后才能引用。
- 用户通过声明语句，声明各种名字，并给出它们的类型、维数等信息，编译程序在处理这些声明语句时，应该将声明中的名字及其信息登录到符号表中，同时编译程序还要给变量分配存储单元，而存储单元地址也必须登录在符号表中。
- 当编译程序编译到引用所声明的变量时(赋值或引用其值)，要进行语法语义正确性检查(类型是否符合要求)和生成相应的目标程序，这就需要查符号表以取得相关信息。



L: x := a + b;
...

1. 语法分析和语义分析
   - 说明语句、赋值语句的语法规则
   - 上下文有关分析：是否声明
   - 类型一致性检查
2. 生成目标代码

```
LOAD  a的地址
ADD   b的地址
STO   x的地址
```

(3)有关符号表的操作：填表和查表

填表：当分析到程序中的说明或定义语句时，应将说明或定义的名字，以及与之有关的信息填入符号表中。

例：**Procedure P( )**

查表：(1) 填表前查表，检查在程序的同一作用域内名字是否重复定义；

(2) 检查名字的种类是否与说明一致；

(3) 对于强类型语言，要检查表达式中各变量的类型是否一致；

(4) 生成目标指令时，要取得所需要的地址。

……

5.2 符号表的组织与内容

(1)符号表的结构与内容

符号表的基本结构:

名字	特性(信息)

“名字”域: 存放名字，一般为标识符的符号串,也可为指向标识符字符串的指针。

名字	特性(信息)

“特性”域：可包括多个子域，分别表示标识符的有关信息，如：

名字(标识符)的种类：简单变量、函数、过程、数组、标号、参数等

类型：如整型、浮点型、字符型、指针等

性质：变量形参、值形参等

值：常量名所代表的数值

地址：变量所分配单元的首址或地址位移

大小：所占的字节数

作用域的嵌套层次：

对于数组：维数、上下界值、计算下标变量地址所用的信息（数组信息向量）以及数组元素类型等。

对于记录（结构、联合）：域的个数，每个域的域名、地址位移、类型等。

对于过程或函数：形参个数、所在层次、函数返回值类型、局部变量所占空间大小等。

对于指针：所指对象类型等。

(2)符号表的组织方式

1. 统一符号表:不论什么名字都填入统一格式的符号表中

符号表表项应按信息量最大的名字设计,填表、查表比较方便, 结构简单, 但是浪费大量空间。

2. 对于不同种类的名字分别建立各种符号表

节省空间, 但是填表和查表不方便。

3. 折中办法:大部分共同信息组成统一格式的符号表,
特殊信息另设附表, 两者用指针连接。

例：begin

A : real;

B : array [1:100] of real;

:

:

end

A	简变	实型	地址	
B	数组	实型		

指针连接

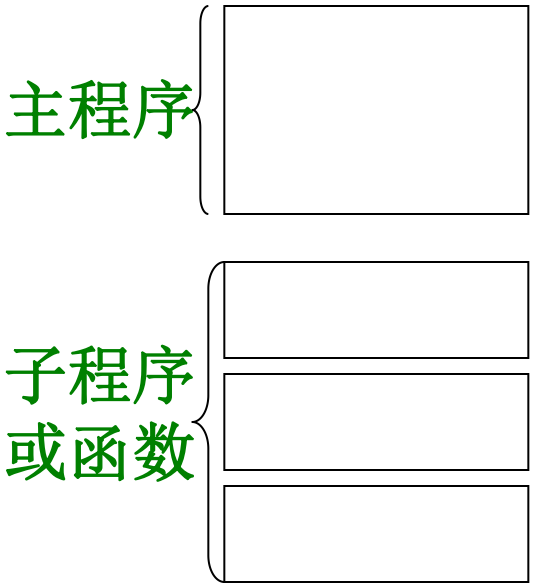
补充

维数	上下界	首地址

5.3 非分程序结构语言的符号表组织

(1)非分程序结构语言：每个可独立进行编译的程序单元是一个不包含有子模块的单一模块，如FORTRAN语言。

FORTRAN程序构造

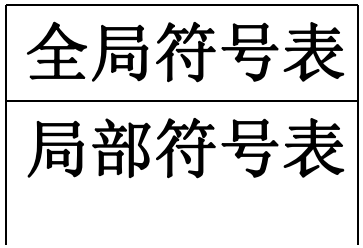


主程序和子程序中可定义common语句

(2) 标识符的作用域及基本处理办法

1. 作用域: 全局:子程序名,函数名和公共区名。
局部: 程序单元中定义的变量。

2. 符号表的组织:



3. 基本处理办法:
<1> 子程序、函数名和公共区名填入全局符号表。

<2> 在子程序（函数）声明部分读到标识符，造局部符号表。

查本程序单元局部符号表，有无同名？
- 有,重复声明,报错
- 无,造表

<3> 在语句部分读到标识符,查表:

查本程序单元局部符号表，有无同名？
- 有,即已声明
- 无,查全局变量表
  - 有,全局量
  - 无, 无定义标识符

4. 程序单元结束: 释放该程序单元的局部符号表。

5. 程序编译完成: 释放全部符号表。

(3) 符号表的组织方式

1. 无序符号表: 按扫描顺序建表,查表要逐项查找

查表操作的平均长度为$n+1/2$

2. 有序符号表：符号表按变量名进行字典式排序

线性查表： $n+1/2$

折半查表： $Log_2^n - 1$

3. 散列符号表(Hash表)：符号表地址 = Hash(标识符)

解决：冲突

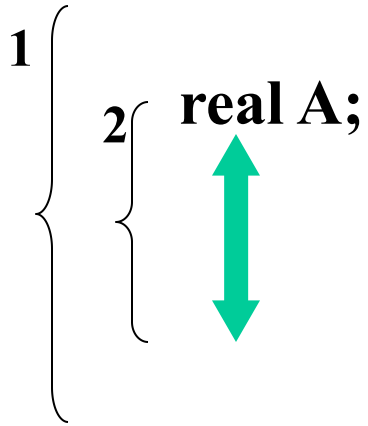
5.4 分程序结构语言的符号表组织

(1) 分程序结构语言:模块内可嵌入子模块

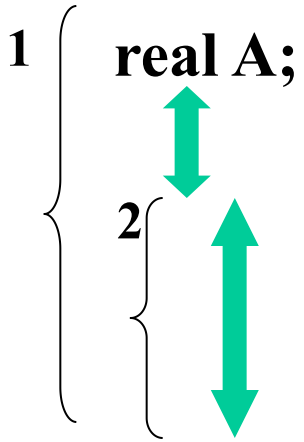
(2) 标识符的作用域和基本处理方法:

作用域：标识符局部于所定义的模块(最小模块)

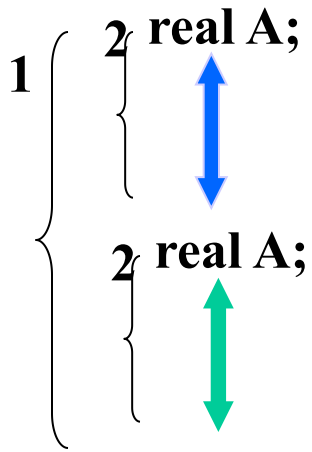
① 模块中所定义的标识符作用域是定义该标识符的子程序

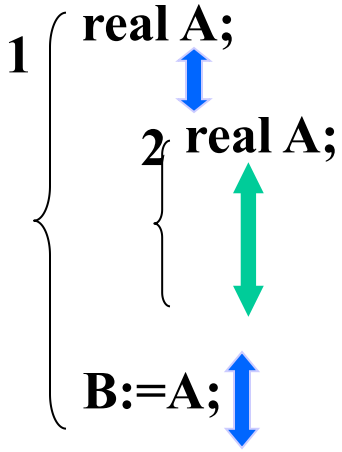


A为内分程序局部变量



A为可作用于内分程序的全局变量

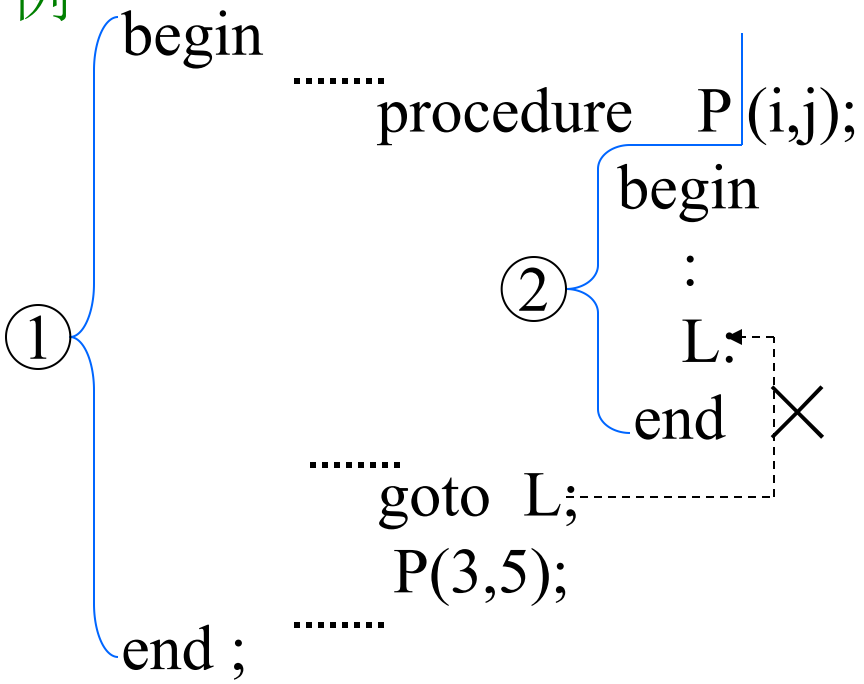




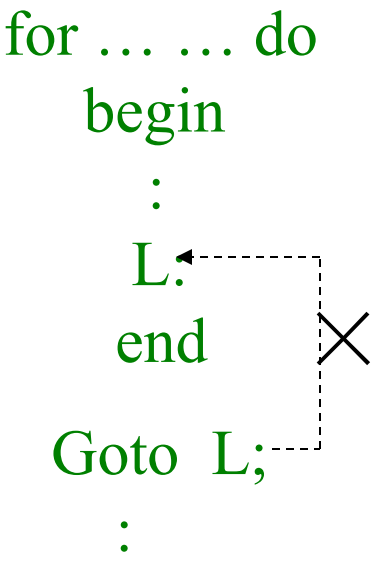
都是局部变量

② 过程或函数说明中定义的标识符(包括形参)其作用域为本过程体。

例



③ 循环语句中定义的标识符,其作用域为该循环语句。

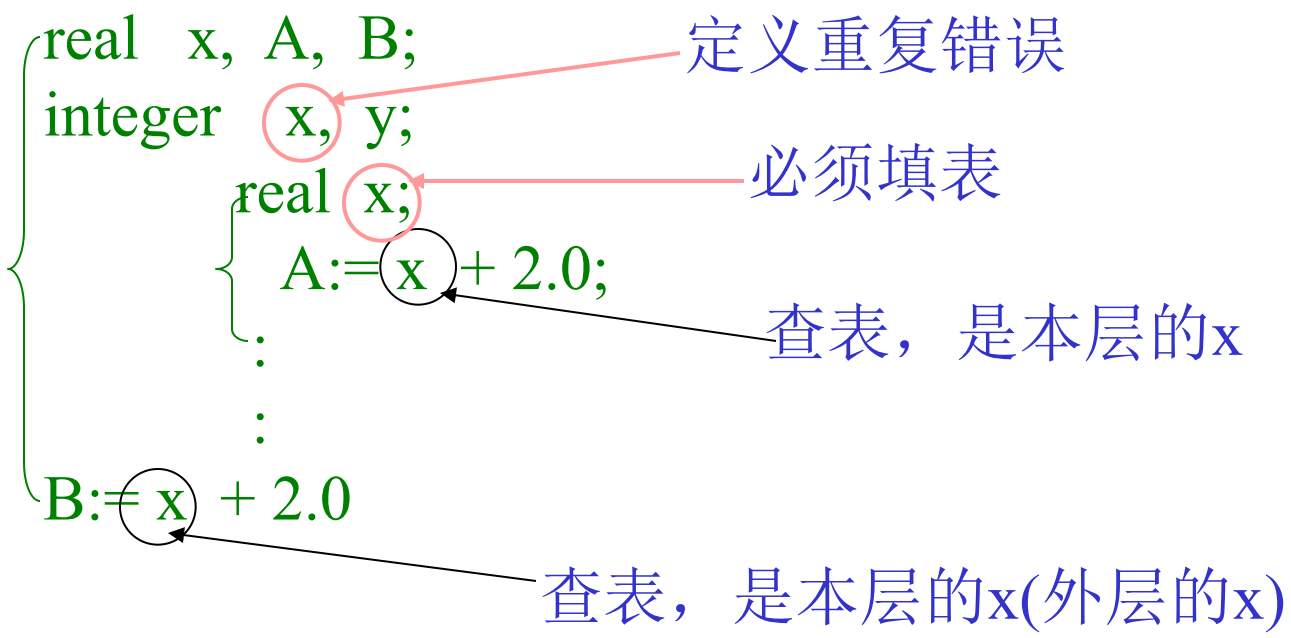


基本处理办法:

建查符号表均要遵循标识符的作用域规定进行。

建表：不能重复,不能遗漏

查表：按标识符作用域

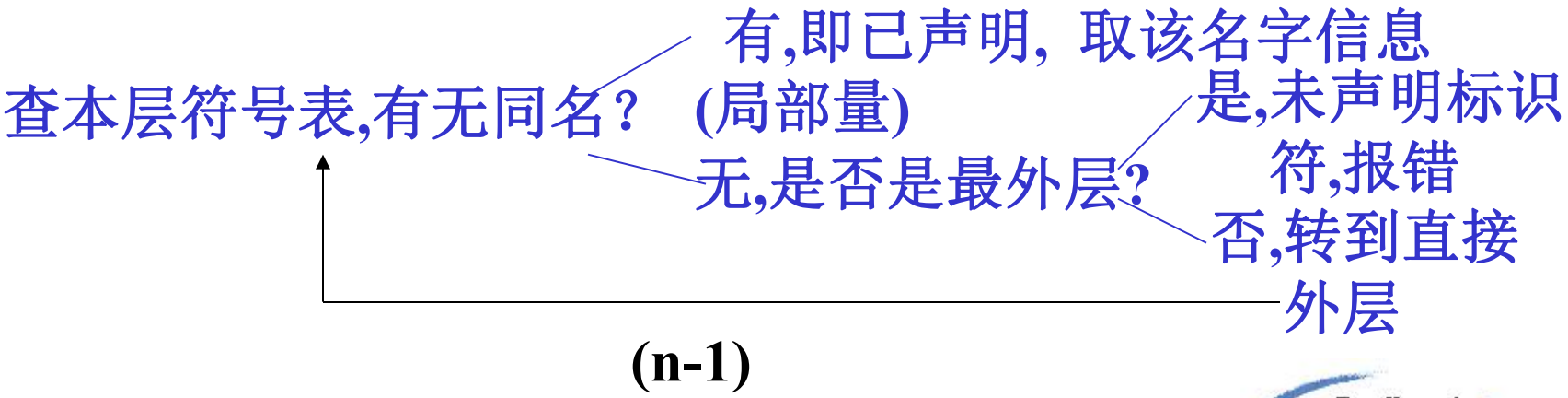


处理方法:

a. 在程序声明部分读到标识符时(声明性出现),建表:

查本层符号表,有无同名?
- 有,重复声明,报错
- 无,填入符号表

b. 在语句中读到标识符(引用性出现),查表:



c. 标准标识符的处理

主要是语言定义的一些标准过程和函数的名字，它们是标识符的子集。

如 **sin con abs….**

特点：1) 用户不必声明,就可全程使用

2) 设计编译程序时，标准名字及其数目已知

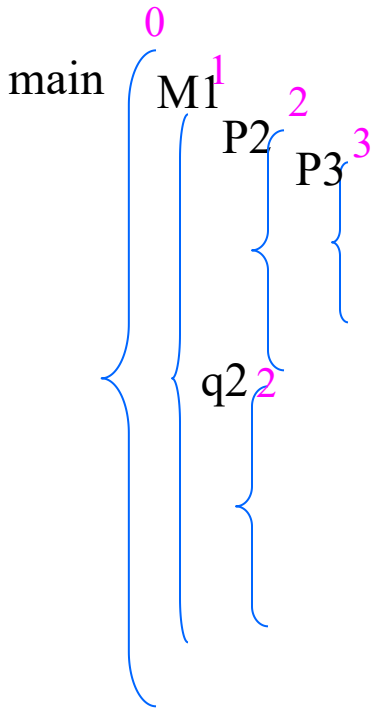
处理方法：1) 单独建表：使用不便，费时。

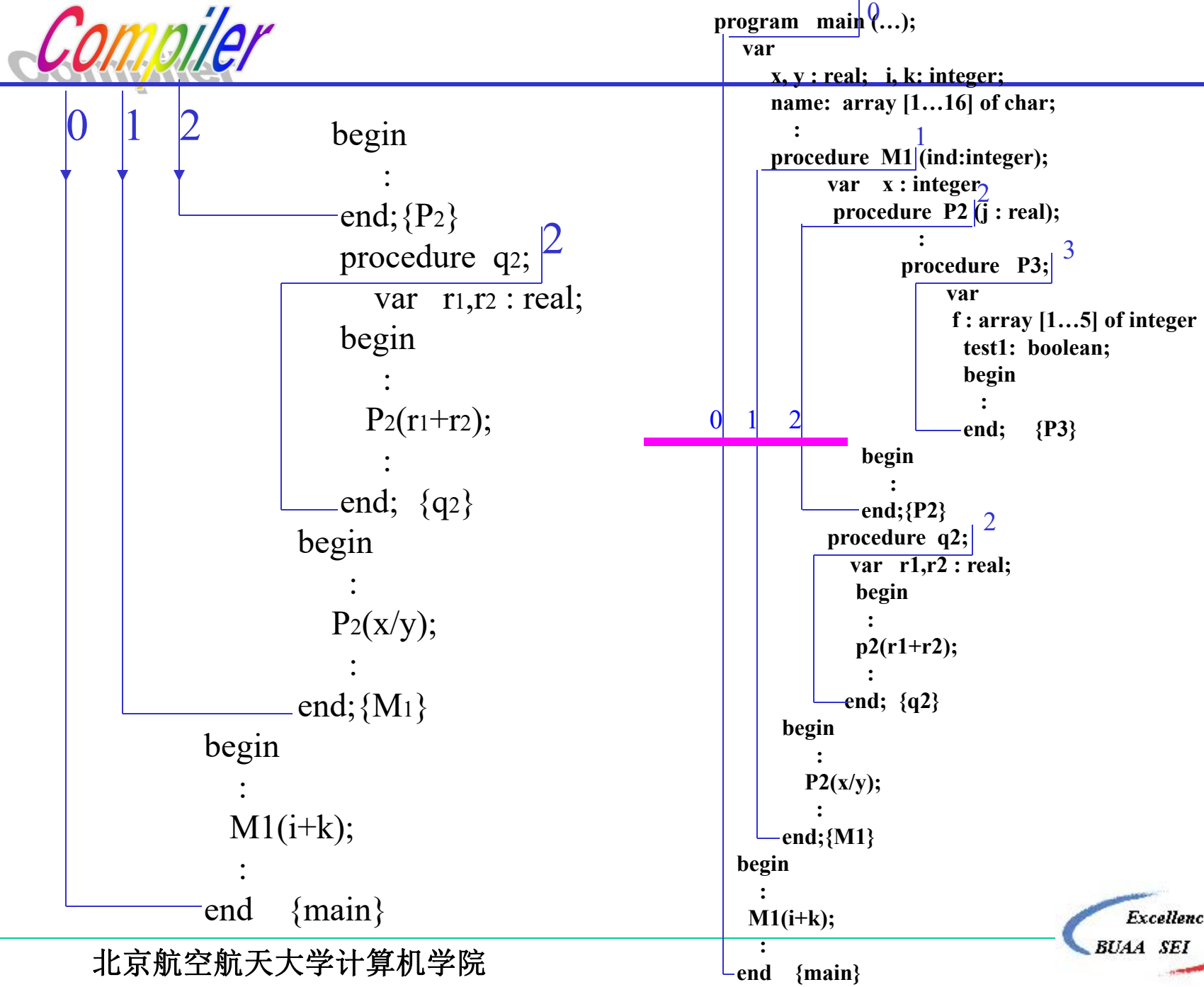
2) 预先将标准名填入名字表中

最外层

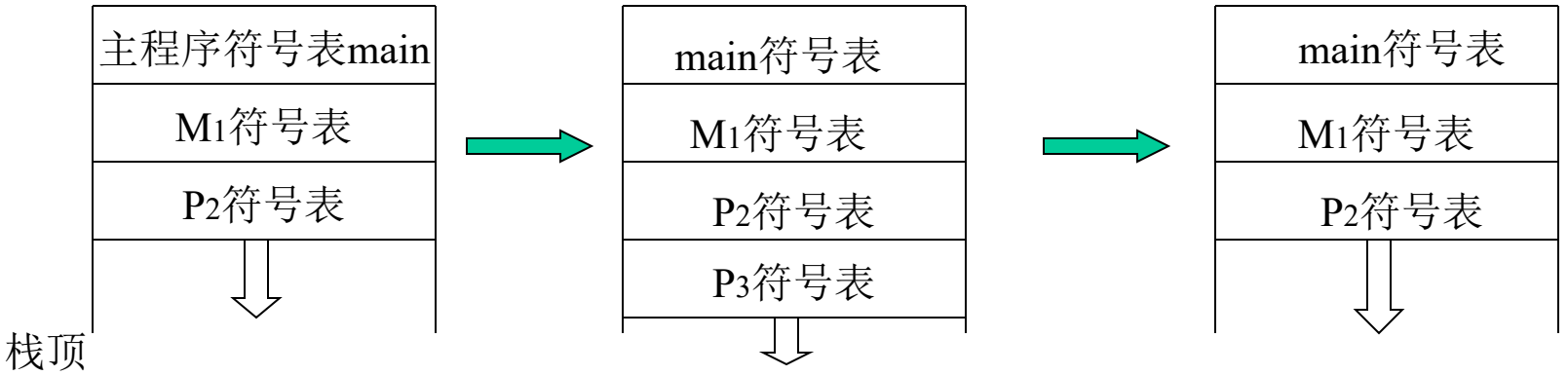
例:Pascal程序的分程序结构示例如下:

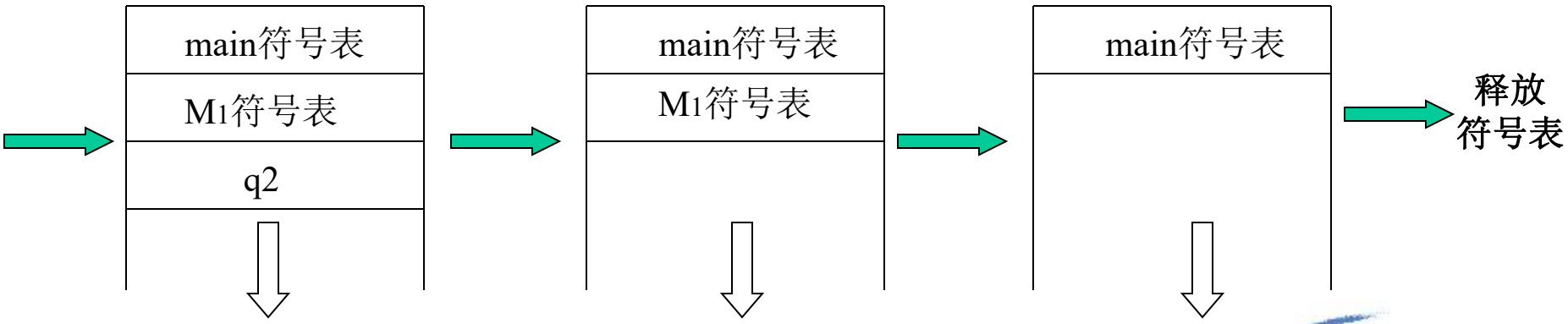
```
program  main (…);                                  0
    var
        x, y : real;   i, k: integer;
        name:  array [1…16] of char;
          :                                         1
        procedure  M1 (ind:integer);
              var    x : integer
               procedure  P2 (j : real);            2
                     :
                  procedure  P3;                    3
                       var
                        f : array [1…5] of integer
                          test1:  boolean;
                          begin
                            :
                          end;     {P3}
:
```




0 1 2
begin
:
end;{P2}
procedure q2;
var r1,r2 : real;
begin
:
P2(r1+r2);
:
end; {q2}
begin
:
P2(x/y);
:
end;{M1}
begin
:
M1(i+k);
:
end {main}
program main (…);
var
x, y : real; i, k: integer;
name: array [1…16] of char;
:
procedure M1 (ind:integer);
var x : integer
procedure P2 (j : real);
:
procedure P3;
var
f : array [1…5] of integer
test1: boolean;
begin
:
end; {P3}
begin
:
end;{P2}
procedure q2;
var r1,r2 : real;
begin
:
p2(r1+r2);
:
end; {q2}
begin
:
P2(x/y);
:
end;{M1}
begin
:
M1(i+k);
:
end {main}
0 1 2
0
1
2
2
3

栈式符号表结构


主程序符号表main
M1符号表
P2符号表
栈顶
main符号表
M1符号表
P2符号表
P3符号表
main符号表
M1符号表
P2符号表


main符号表
M1符号表
q2
main符号表
M1符号表
main符号表
释放
符号表

符号表

	name	kind	type	lev	other inf
1	x	var	real	0	
2	y	var	real	0	
3	i	var	int	0	
4	k	var	int	0	
5	name	var	array	0	
6	M_1	proc		0	
7	ind	para	int	1	
8	x	var	int	1	
9	P_2	proc		1	
10	j	para	real	2	
11	P_3	proc		2	
12	f	var	array	3	
13	test1	var	boolean	3	

main

M_1

P_2

P_3

分程序索引表

0	1
1	7
2	10
3	12

编译q_2说明部分后:

7	ind	para	int	1		M_1
8	x	var	int	1		
9	P_2	proc		1		
10		para	real	1		
11	q_2	proc		1		
12	r_1	var	real	2		q_2
13	r_2	var	real	2		

编译完q_2过程体:

7	ind	para	int	
8	x	var	int	
9	P_2	proc		
10		para	real	
11	q_2	proc		

当过程和函数体编译完成后，应将与之相应的参数名和局部变量名以及后者的特性信息从符号表中删去。

要求：给出一段程序，会画出其栈式符号表

第七章 源程序的中间形式

- 波兰表示
- N一元表示
- 抽象机代码

7.1 波兰表示

一般编译程序都生成中间代码，然后再生成目标代码，主要优点是可移植(与具体目标程序无关)，且易于目标代码优化。有多种中间代码形式：

波兰表示　　N－元组表示　　抽象机代码

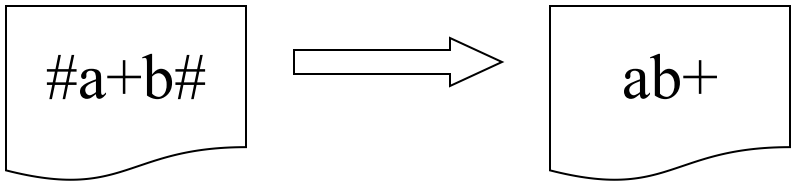
波兰表示

算术表达式：　　F*3.1416*R*(H+R)

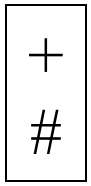
转换成波兰表示：　　F3.1416*R*HR+*

赋值语句：　　A := F * 3.1416 * R * (H + R)

波兰表示：　　AF3.1416 * R * HR + * :=



操作符栈

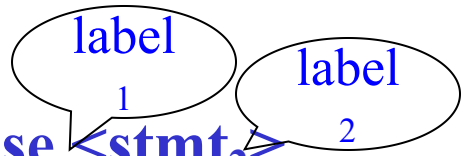


#优先级最低

算法：

设一个操作符栈；当读到操作数时，立即输出该操作数，当扫描到操作符时，与栈顶操作符比较优先级，若栈顶操作符优先级高于栈外，则输出该栈顶操作符，反之，则栈外操作符入栈。

if 语句的波兰表示



if 语句 ： if <expr> then <$stmt_1$> else <$stmt_2$>

波兰表示为 ： <expr><$label_1$>BZ<$stmt_1$><$label_2$>BR<$stmt_2$>

BZ: 二目操作符

若<expr>的计算结果为0 (false)，
则产生一个到<$label_1$>的转移

BR: 一目操作符

产生一个到< $label_2$>的转移

波兰表示为 ：<expr><label$_1$>BZ<stmt$_1$><label$_2$>BR<stmt$_2$>

由if语句的波兰表示可生成如下的目标程序框架：

<expr>
BZ label$_1$
<stmt$_1$>
BR label$_2$
label$_1$： <stmt$_2$>
label$_2$：

其他语言结构也很容易将其翻译成波兰表示，
使用波兰表示优化不是十分方便。

7.2 N一元表示

在该表示中，每条指令由n个域组成，通常第一个域表示操作符，其余为操作数。

常用的n元表示是：　三元式　四元式

三元式

操作符	左操作数	右操作数

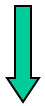
表达式的三元式：w * x + (y + z)

(1) *, w, x
(2) +, y, z
(3) +, (1), (2)

第三个三元式中的操作数(1)(2)表示第(1)和第(2)条三元式的计算结果。

条件语句的三元式：

```
if  x > y  then
    z := x;
else   z := y+1;
```



(1)	－,	x,	y
(2)	BMZ,	(1),	(5)
(3)	:=,	z,	x
(4)	BR,	,	(7)
(5)	+,	y,	1
(6)	:=,	z,	(5)
(7)			
	:		
	:		

其中：

BMZ：是二元操作符,测试第二个域的值，若≤0，则按第3个域的地址转移，若>0,则顺序执行。

BR：一元操作符，按第3个域作无条件转移。

使用三元式不便于代码优化，因为优化要删除一些三元式，或对某些三元式的位置要进行变更，由于三元式的结果(表示为编号)，可以是某个三元式的操作数，随着三元式位置的变更也将作相应的修改，很费事。

间接三元式：

为了便于在三元式上作优化处理，可使用间接三元式

三元式的执行次序用另一张表表示,这样在优化时，三元式可以不变，而仅仅改变其执行顺序表。

例： A:=B+C*D/E
F:=C*D

用间接三元式表示为：

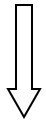
操作	三元式
1. (1)	(1) *, C, D
2. (2)	(2) /, (1), E
3. (3)	(3) +, B, (2)
4. (4)	(4) :=, A, (3)
5. (1)	(5) :=, F, (1)
6. (5)	

四元式表示

操作符	操作数1	操作数2	结果

结果：通常是由编译引入的临时变量，可由编译程序分配一个寄存器或主存单元。

例：　　(A + B) * (C + D) - E



+, A, B, T1
+, C, D, T2
*, T1, T2, T3
—, T3, E, T4

式中T1，T2，T3，T4为临时变量，由四元式优化比较方便

7.3 抽象机代码

许多pascal编译系统生成的中间代码是一种称为P－code的抽象代码，P－code的“P”即“Pseudo”

抽象机：

寄存器

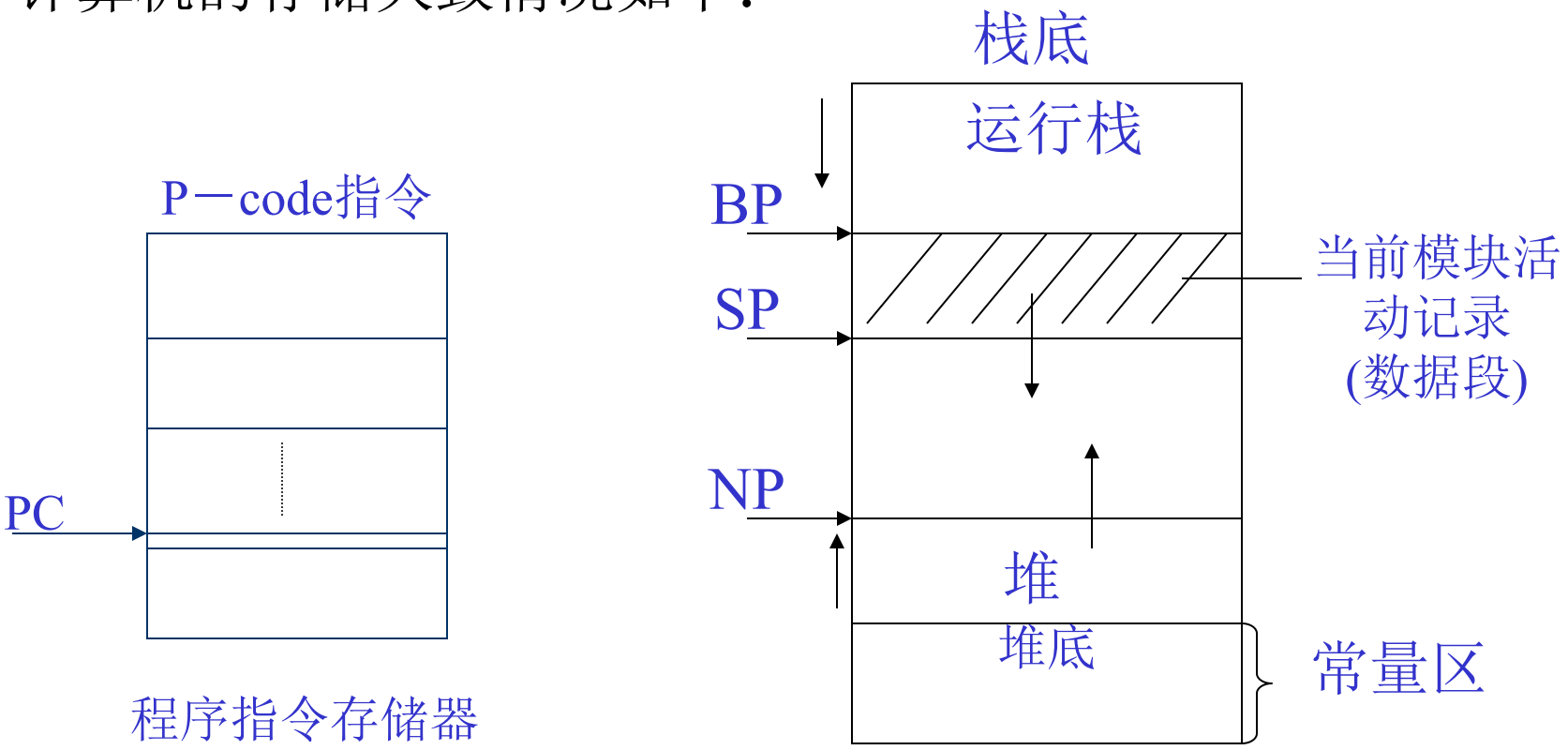
保存程序指令的存储器

堆栈式数据及操作存储

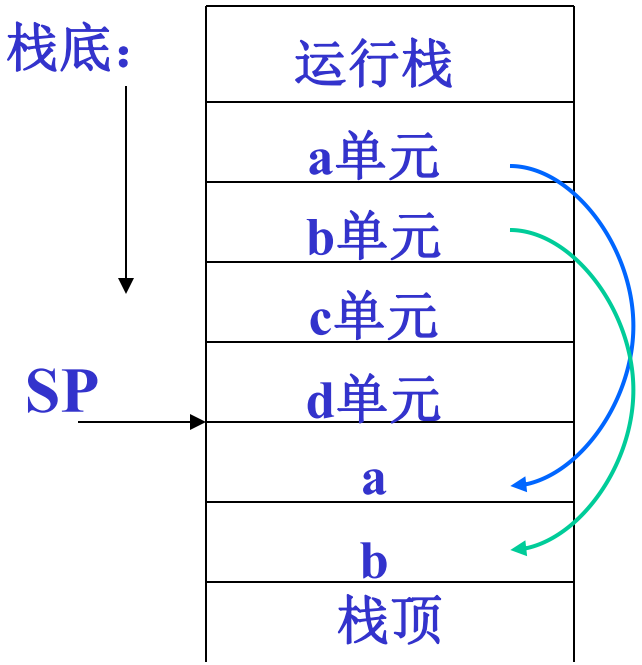
寄存器有：

1. PC－程序计数器
2. NP－New指针，指向“堆”的顶部。“堆”用来存放由New生成的动态数据。
3. SP－ 运行栈指针，存放所有可按源程序的数据声明直接寻址的数据。
4. BP－基地址指针，即指向当前活动记录的起始位置指针。
5. 其他，（如MP－栈标志指针，EP－极限栈指针等）

计算机的存储大致情况如下：



运行P－code的抽象机没有专门的运算器或累加器，所有的运算(操作)都在运行栈的栈顶进行，如要进行d:=(a+b)*c的运算，生成P－code序列为：



取a	LOD a
取b	LOD b
+	ADD
取c	LOD c
*	MUL
送d	STO d

P－code实际上是波兰表示形式的中间代码

第八章　错误处理

- 概述
- 错误分类
- 错误的诊察和报告
- 错误处理技术

8.1 概述

1. 必备功能之一

正确的源程序：通过编译生成目标代码

错误的源程序：通过编译发现并指出错误

2. 错误处理能力

(1) 诊察错误的能力

(2) 报错及时准确

(3) 一次编译找出错误的多少

(4) 错误的改正能力

(5) 遏止重复的错误信息的能力

8.2 错误分类

从编译角度，将错误分为两类：语法错误和语义错误

语法错误：源程序在语法上不合乎文法

如：

A[I， J := B +* C

语义错误主要包括：程序不符合语义规则或
超越具体计算机系统的限制

语义规则：

- 标识符先说明后引用
- 标识符引用要符合作用域规定
- 过程调用时实参与形参要一致
- 参与运算的操作数类型一致
- 下标变量下标不能越界

超越系统限制：

- 数据溢出错误
- 符号表、静态存储分配数据区溢出
- 动态存储分配数据区溢出

8.3 错误的诊察和报告

错误诊察：

1. 违反语法和语义规则以及超过编译系统限制的错误。

编译程序：语法和语义分析时

（语义分析要借助符号表）

2. 下标越界，计算结果溢出以及动态存储数据区溢出。

目标程序：目标程序运行时

对此，编译程序要生成相应的目标程序作检查和进行处理

错误报告：

1. 出错位置：即源程序中出现错误的位置

   实现：行号计数器　　line_no

   　　单词序号计数器　char_no

一旦诊察出错误，当时的计数器内容就是出错位置

2. 出错性质：

   可直接显示文字信息

   可给出错误编码

3. 报告错误的两种方式：

(1) 分析完以后再报告(显示或者打印)

编译程序可设一个保存错误信息的数据区(可用记录型数组)，将语法语义分析所诊察到的错误送数据区保存，待源程序分析完以后，显示或打印错误信息。

例：A[x , y :=B+*C

源程序行号	错误序号	错误性质
x x	6	缺少“]”
x x	10	表达式语法错误

(2) 边分析边报告

可以在分析一行源程序时若发现有错，立即输出该行源程序，并在其下输出错误信息。

Line－no A[x , y := B+ *C

缺 “]”on　　表达式语法错 m

错误编号

有时候报错不一定十分准确
（位置和性质），需进一步分析

例

```
begin
    ........
    i := 1  step 1    until  n  do
      ............
end
```

8.4 错误处理技术

发现错误后，在报告错误的同时还要对错误进行处理，以方便编译能进行下去。目前有两种处理办法：

1. 错误改正：指编译诊察出错误以后，根据文法进行错误改正。

如：A[i , j :=B+*C

要正确地改正错误是很困难的

但不是总能做到,如A:=B-C*D+E)

2. 错误局部化处理：指当编译程序发现错误后，尽可能把错误的影响限制在一个局部的范围，避免错误扩散和影响程序其他部分的分析。

(1) 一般原则

当诊察到错误以后，就暂停对后面符号的分析，跳过错误所在的语法成分然后继续往下分析。

词法分析：发现不合法字符，显示错误，并跳过该标识符(单词)继续往下分析。

语法语义分析：跳过所在的语法成分(短语或语句)，一般是跳到语句右界符，然后从新语句继续往下分析。

(2) 错误局部化处理的实现（递归下降分析法）

cx: 全局变量，存放错误信息。

•用递归下降分析时，如果发现错误，便将有关错误信息（字符串或者编号）送CX，然后转出错误处理程序；

•出错程序先打印或显示出错位置以及出错信息，然后跳出一段源程序，直到跳到语句的右界符(如：end）或正在分析的语法成分的合法后继符号为止，然后再往下分析。

例:条件语句分析: if<B> then <stmt>[else< stmt >];

有如下分析程序:

```
procedure  if_stmt;
 begin
    nextsym;                          /*读下个单词符号*/
    B;                                /*调用布尔表达式处理程序*/
    if  not  class=‘then’ then
      begin
         cx :=‘缺then’                /*错误性质送cx*/
         error;                       /*调用出错处理程序*/
      end;
    else
      begin
            nextsym;
            statement
      end;
    if class=‘else’then
      begin
            nextsym;
            statement;
      end
 end  if_stmt;
```

局部化处理的出错程序为:

```
procedure error;
      begin
        write(源程序行号, 序号, cx)
        repeat
          nextsym;
        until class = ';' or class = 'end' or class = 'else'
end  error;
```

real x, 3a, a, bcd, 2fg;

(3) 提高错误局部化程度的方法

设 S_1：合法后继符号集 (某语法成分的后继符号)

S_2：停止符号集　　(跳读必须停止的符号集)

进入某语法成分的分析程序时：

S_1:= 合法后继符号

S_2:= 停止符号

当发现错误时: error(S_1,S_2)

```
procedure   error(S1,S2)
        begin
                write(line_no, char_no, cx);
                repeat
                        nextsym
                until(class in  S1 or class in  S2 );
        end
```

if<B> then <stmt>[else< stmt >]

若<B>有错,则可跳到then,

若<stmt>有错,则可跳到else。

3.目标程序运行时错误检测与处理.

下标变量下标值越界

计算结果溢出

动态存储分配数据区溢出

· 在编译时生成检测该类错误的代码。

对于这类错误,要正确的报告出错误位置很难,因为目标程序与源程序之间难以建立位置上的对应关系

一般处理办法:

当目标程序运行检测到这类错误时,就调用异常处理程序,打印错误信息和运行现场(寄存器和存储器中的值)等, 然后停止程序运行。

第九章　语法制导翻译技术

- 翻译文法(TG)和语法制导翻译
- 属性翻译文法(ATG)
- 自顶向下语法制导翻译
  - 翻译文法的自顶向下语法制导翻译
  - 属性文法的自顶向下语法制导翻译
- 自底向上的语法制导翻译（自学）

9.0 本章导言

★ 词法分析，语法分析：解决单词和语言成分的识别及词法和语法结构的检查。语法结构可形式化地用一组产生式来描述。给定一组产生式，能够很容易地将其分析器构造出来。

本章要介绍的是语义分析和代码生成技术。

★ 程序语言的语义形式化描述目前有三种基本描述方法，即：

- 操作语义
- 指称语义
- 公理语义

9.1 翻译文法和语法制导翻译

有上下无关文法G[E]:

1. E →E+T
2. E →T
3. T →T*F
4. T →F
5. F →(E)
6. F →i

此文法是一个中缀算术表达式文法

翻译的任务是: 中缀表达式 ⟶ 逆波兰表示

a+b*c ⟶ abc*+

假如翻译任务是要将中缀表达式简单变换为波兰后缀表示，只需在上述文法中插入相应的动作符号。

1. E →E+T@+
2. E →T
3. T →T*F @*
4. T →F
5. F →(E)
6. F →i@i

其中：

@+,@*,@i 为动作符号。@为动作符号标记，后面为字符串。

在本例中，其对应语义子程序的功能是要输出打印动作符号标记后面的字符串。

所以，产生式1：E→E+T@+ 的语义是分析E, + 和T，输出+

产生式6：F→i @ i的语义是分析i，输出i

下面给出输入文法和翻译文法的概念：

输入文法： 未插入动作符号时的文法。

由输入文法可以通过推导产生输入序列。

翻译文法： 插入动作符号的文法。

由翻译文法可以通过推导产生活动序列。

⇩

输入序列

动作序列

例：(i+i) * i

可以用输入文法推导：

$E \Rightarrow T \Rightarrow T*F \Rightarrow F*F \Rightarrow (E)*F \Rightarrow (E+T)*F$

$\overset{*}{\Rightarrow}(i+i)*i$

用相应的翻译文法推导，可得：

$E \Rightarrow T$

$\Rightarrow T*F@*$

$\Rightarrow F*F@*$

$\Rightarrow (E)*F@*$

$\Rightarrow (E+T@+)*F@* \quad \overset{*}{\Rightarrow}(i@i+i@i\ @+)*i@i@*$

活动序列：由翻译文法推导出的符号串，由终结符和动作符号组成。

- 从活动序列中，抽去动作符号，则得输入序列（i+i)*i
- 从活动序列中，抽去输入序列，则得动作序列，执行动作序列，则完成翻译任务：

@i@i@+@i@* ⇒ ii+i*

定义9.1

翻译文法是上下文无关文法，其终结符号集由输入符号和动作符号组成。由翻译文法所产生的终结符号串称为活动序列。

上例题中的翻译文法为：

$G_T = (V_n, V_t, P, E)$

$V_n = \{E, T, F\}$

$V_t = \{i, +, *, (,), @+, @*, @i\}$

$P = \{E \rightarrow E+T@+,\ E \rightarrow T,\ T \rightarrow T*F@*,\ T \rightarrow F,\ F \rightarrow (E),$
$F \rightarrow i@i\}$

符号串翻译文法：若插入文法中的动作符号对应的语义子程序是输出动作符号标记@后的字符串的文法。

语法导制翻译：按翻译文法进行的翻译。
给定一输入符号串，根据翻译文法获得翻译该符号串的动作序列，并执行该序列所规定的动作的过程。

语法导制翻译的实现方法：

在文法的适当位置插入语义动作符号，当按文法分析到动作符号时就调用相应的语义子程序，完成翻译任务。

翻译文法所定义的翻译是由输入序列和动作序列组成的对偶集。

如：(i+i)*i,　@i@i@+@i@*　→ii＋i*

i+i*i　　@i@i@i@*@+

因此，给定一个翻译文法，就给定了一个对偶集。

9.2 属性翻译文法

在翻译文法的基础上，可以进一步定义属性文法，
翻译文法中的符号，包括终结符、非终结符和动作符号
均可带有属性，这样能更好的描述和实现编译过程。

属性可以分为两种：

综合属性

继承属性

9.2.1 综合属性

基本操作数带有属性的表达式文法G[E]

1.E→E+F　　4.T→F

2.E→T　　5.F→(E)

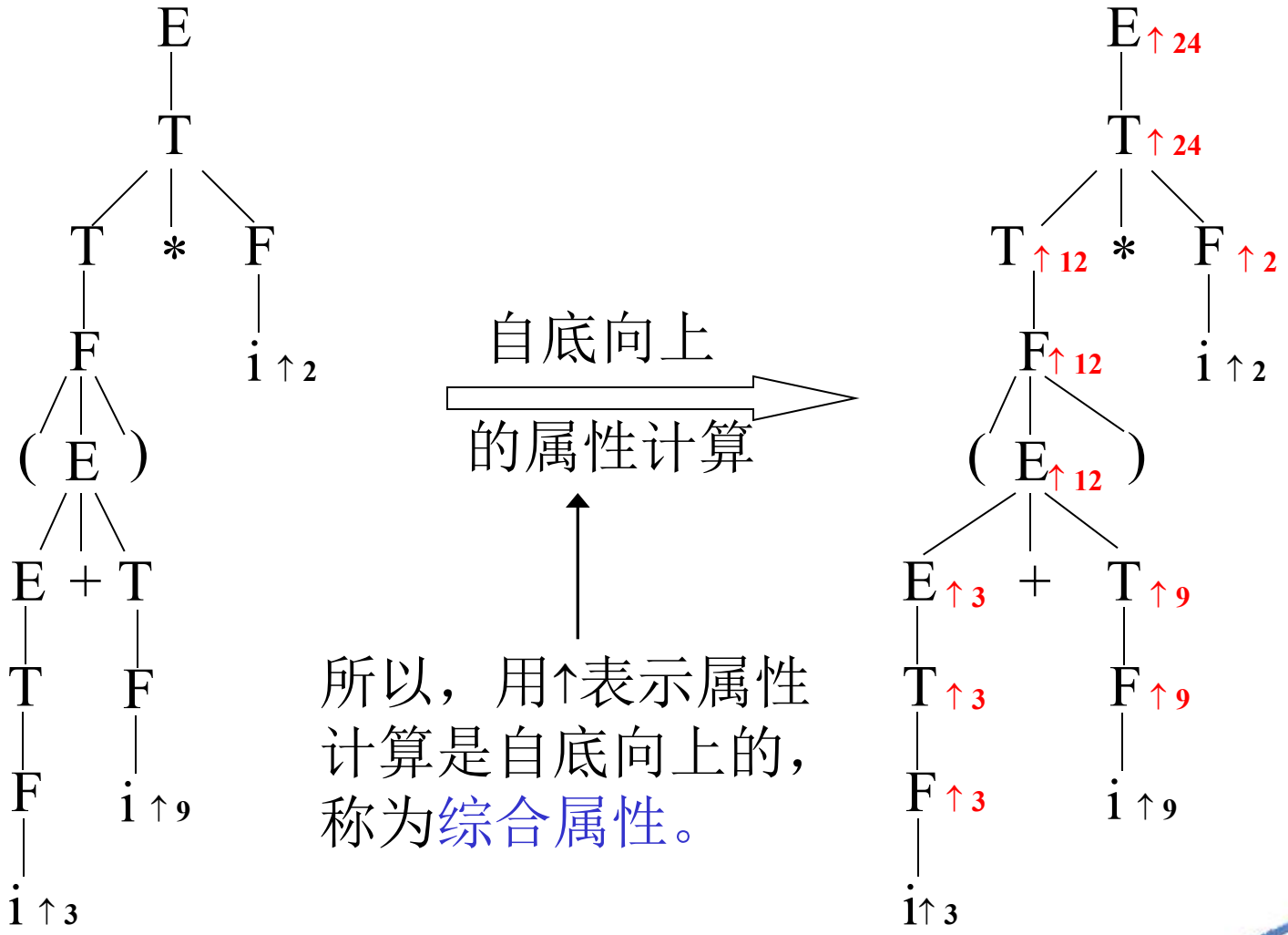
3.T→T*F　　6.F→i↑c

其中↑c是综合属性符号，↑为综合属性标记，c为属性变量或者属性值。

此文法能够产生如下的输入序列：

$$(i\uparrow_3 + i\uparrow_9) * i\uparrow_2$$

根据给定的文法，可写出该输入序列的语法树



所以，用↑表示属性计算是自底向上的，称为综合属性。

为了形式地表示上述表达式的属性求值过程，可以改写上述文法：

产生式	求值规则
1.$E\uparrow_{p4} \longrightarrow E\uparrow_{q5} + T\uparrow_{r2}$	$p_4 := q_5 + r_2$;
2.$E\uparrow_{p3} \longrightarrow T\uparrow_{q4}$	$p_3 := q_4$;
3.$T\uparrow_{p2} \longrightarrow T\uparrow_{q3} * F\uparrow_{r1}$	$p_2 := q_3 * r_1$;
4.$T\uparrow_{p2} \longrightarrow F\uparrow_{q2}$	$p_2 := q_2$;
5.$F\uparrow_{p1} \longrightarrow (E\uparrow_{q1})$	$p_1 := q_1$;
6.$F\uparrow_{p1} \longrightarrow i\uparrow_{q1}$	$p_1 := q_1$;

说明：

- p,q,r为属性变量名。
- 属性变量名局部于每个产生式，也可使用不同的名字。

- 求值规则：综合属性是自右向左，自底向上求值。

9.2.2 继承属性

考虑下列文法：G[<说明>]：

1. <说明> → Type id <变量表>
2. <变量表> → , id <变量表>
3. <变量表> → ε

其中

Type: 类型名（值：int, real, bool等）

id： 变量名（值：指向该变量符号表项的指针）

上述文法所产生的语句：int A,BC

该文法的翻译任务：将声明的变量填入符号表

完成该工作的动作符号：@set_table

符号表
A 整型
BC 整型

翻译文法：

1. <说明> → Type id @set_table <变量表>
2. <变量表> → ， id @set_table <变量表>
3. <变量表 > → ε

填表时需要的信息：类型，名字，以及填的位置（可以用全程变量或指针）

如何得到？

类型和名字在词法分析时得到，可设两个综合属性。

Type ↑t　　t中放类型值

id ↑n　　n中放变量名

填表动作符号也可带有属性：

@set_table ↓ t_1, n_1　　↓ t_1,n_1　　可从前面得到，所以称为继承属性，继承前面的值

<变量表> ↓ t_2　　↓ t_2　　同上

属性翻译文法：

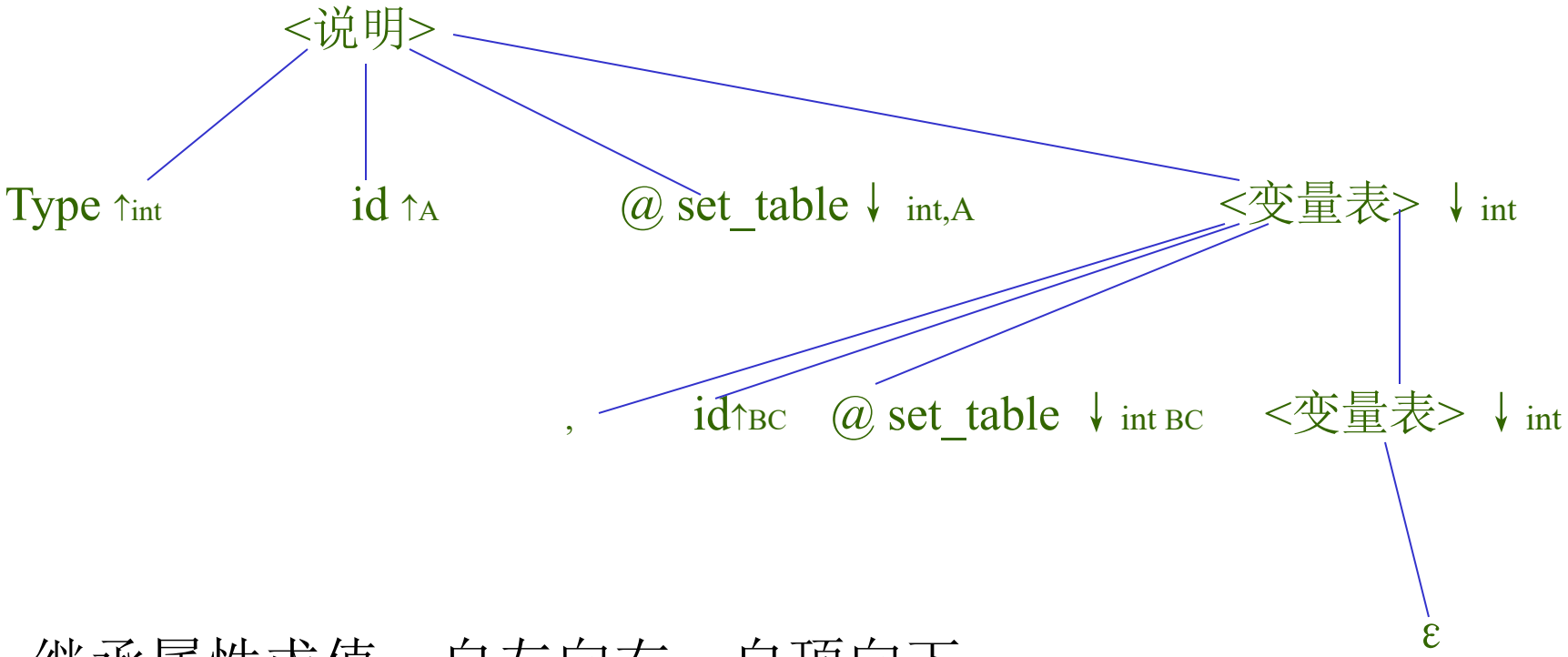
1.<说明> → Type↑t　id↑n @set_table ↓ t1,n1 <变量表> ↓ t2　　$t_2,t_1 := t$;　$n_1 := n$;

2.<变量表> ↓ t2 → ，　id ↑ n @set_table ↓ t1,n1 <变量表> ↓ t3　　$t_3,t_1 := t_2$;　$n_1 := n$;

3.<变量表 > ↓ t2 → ε

例：int A,BC ⇨ Type ↑int id ↑A , id↑BC

语法树：



继承属性求值：自左向右，自顶向下

综合属性求值：自右向左，自底向上

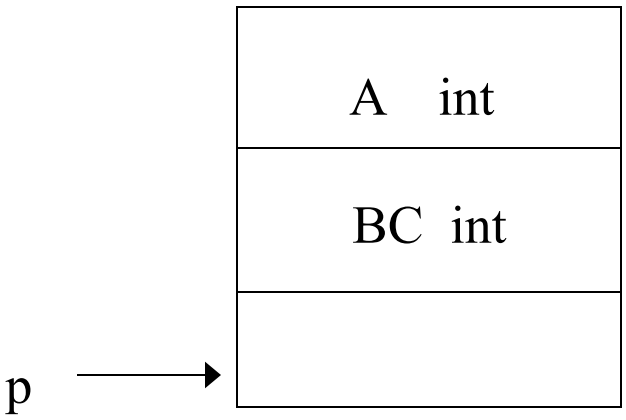
int A, BC 的分析翻译过程：

<说明> ⇒ Type ↑t id ↑n1@ set_table ↓t,n1 <变量表> ↓t

⁺⇒ Type ↑t id ↑n1 @ set_table ↓t,n1

, id ↑n2 @ set_table ↓t,n2

符号表



9.2.3 (1) L-属性翻译文法（L-ATG）

这是属性翻译文法中较简单的一种。其输入文法要求是LL(1)文法，可用自顶向下分析构造分析器。在分析过程中可进行属性求值。

定义9.2:

L-属性翻译文法是带有下列说明的翻译文法：

1. 文法中的终结符，非终结符及动作符号都带有属性，且每个属性都有一个值域。
2. 非终结符及动作符号的属性可分为继承属性和综合属性。
3. 开始符号的继承属性具有指定的初始值。
4. 输入符号（终结符号）的每个综合属性具有指定的初始值。
5. 属性的求值规则：

属性的求值规则：

继承属性：

（1） 产生式左部非终结符号的继承属性值，取前面产生式右部该符号已有的继承属性值。

（2）产生式右部符号的继承属性值，用该产生式左部符号的继承属性或出现在该符号左部的符号的属性值进行计算。

综合属性：

（1）产生式右部非终结符号的综合属性值，取其下部产生式左部同名非终结符号的综合属性值。

（2）产生式左部非终结符号的综合属性值，用该产生式左部符号的继承属性或某个右部符号的属性进行计算。

（3）动作符号的综合属性用该符号的继承属性或某个右部符号的属性进行计算。

适合在自顶向下分析过程中求值

例：A→BC

求值顺序：

1) A的继承属性 （若A为开始符号，则有指定值,否则由上面产生式右部符号的继承属性求得）
2) B的继承属性 （由A的继承属性求得）
3) B的综合属性 （由下面产生式中左部符号为B的综合属性求得）
4) C的继承属性 （由A的继承属性和B的属性求得）
5) C的综合属性 （由下面产生式中左部符号为C的综合属性求得）
6) A的综合属性 （由前述(2)，即A的继承属性或产生式某右部符号属性计算）

(2) 简单赋值形式的L_属性翻译文法(SL-ATG)

- 一般属性值计算： x:=f（ y ， z ）

  SL-ATG属性值计算： x:=某符号的属性值或常量。

例 x:=y, x,y,z:=17 —— 称为复写规则

为了实现上的方便，常希望文法符号的属性求值规则为上述简单形式的。为此，对现有的L-ATG的定义做一点改变，从而形成一个称为简单赋值形式的L-ATG。

•定义9.4　一个L-ATG被定义为简单赋值形式的(SL-ATG)，当且仅当满足如下条件：

1. 产生式右部符号的继承属性是一个常量，它等于左部符号的继承属性值或等于出现在所给符号左边符号的一个综合属性值。

2. 产生式左部非终结符号的综合属性是一个常量，它等于左部符号的继承属性值或等于右部符号的综合属性值。

因此，一个简单赋值形式的L-ATG除动作符号外，其余符号的属性求值规则其右部是属性或是常量。

- L-ATG ⇒ SL-ATG

给定一个L-ATG，如何找一个等价的赋值形式的L-ATG?

考虑产生式:

$$<A> \rightarrow a\ {}_{\uparrow R}<B>{}_{\uparrow S}<C>{}_{\downarrow I}, \qquad I := f(R, S)$$

显然: 该属性求值规则不是简单赋值形式的，因为它需要对f求值。

第一步：设动作符号“@ f”表示函数f求值，该动作符号有两个继承属性和一个综合属性。

$@f_{\downarrow I_1, I_2 \uparrow s_1}$　　且 $S_1 := f(I_1, I_2)$

第二步：修改产生式

1. 插入“@ f”（在适当位置）
2. 引进新的复写规则（将R , S 赋给I_1和I_2, f值赋给S_1）
3. 删去原有包含f 的规则

$$<A> \rightarrow a \uparrow_R <B> \uparrow_S @ f \downarrow_{I_1, I_2} \uparrow_{S_1} <C> \downarrow_I,$$

$$I_1 := R, \quad I_2 := S, \quad S_1 := f(I_1, I_2), \quad I := S_1.$$

该文法是简单赋值形式的L-ATG.

注意： 无参函数过程作为常数处理，如

$$<A> \rightarrow <B> \uparrow_x <C> \uparrow_y \qquad x, y := NEWT$$

9.3 自顶向下语法制导翻译

9.3.1 翻译文法的自顶向下翻译

——递归下降翻译器

9.3.2 属性翻译文法的自顶向下翻译的实现

——递归下降属性翻译器

9.3.1 翻译文法的自顶向下翻译——递归下降翻译器

按翻译要求，在文法中插入语义动作符号，在分析过程中调用相应的语义处理程序，完成翻译任务。

例：输入文法	翻译文法（符号串翻译文法）
1. $<S> \longrightarrow a<A><S>$	$<S> \longrightarrow a<A>@x<S>$
2. $<S> \longrightarrow b$	$<S> \longrightarrow b@z$
3. $<A> \longrightarrow c<A><S>b$	$<A> \longrightarrow c@y<A><S>@vb$
4. $<A> \longrightarrow \varepsilon$	$<A> \longrightarrow @w$

主程序

```
if CLASS = a or b then
    PROCS;
if CLASS ≠ 右界符 then
    ERROR;
ACCEPT;
```

过程 PROCS

```
case CLASS of
a : P1;
b : P2;
其它：ERROR;
end of case;
```

主程序

```
if CLASS = a or b then
    PROCS;
if CLASS ≠ # then
    ERROR;
ACCEPT;
```

过程 PROCS

```
case CLASS of
a : P1 ;
b : P2 ;
其它：ERROR;
end of case.
```

```
P1:          /*产生式1的代码*/
    NEXTSYM;
    PROCA;
    PROCS;
    RETURN;
P2:          /*产生式2的代码*/
    NEXTSYM;
    RETURN;
过程 PROCA
    …
```

```
P1:          /*产生式1的代码*/
    NEXTSYM;
    PROCA;
    OUT( x ) ;
    PROCS;
    RETURN;
P2:          /*产生式2的代码*/
    NEXTSYM;
    OUT( z ) ;
    RETURN;
过程 PROCA
    …
```

9.3.2 属性文法自顶向下翻译的实现——递归下降翻译器

方法：

- 对于每个非终结符号都编写一个翻译子程序（过程）。根据该非终结符号具有的属性数目，设置相应的参数。

继承属性：声明为赋值形参

综合属性：声明为变量形参

$U_{\downarrow x,\ \uparrow y} \rightarrow \ldots$

Procedure U(x,y);

x—赋值形参

y—变量形参

- 过程调用语句的实参：

  继承属性 ：继承属性值

  综合属性 ：属性变量名（传地址，返回时有值）

- 关于属性名的约定：

1）产生式左部的同名非终结符使用相同的属性名。

（递归下降分析法所必须）

$<L>_{\uparrow a \downarrow b} \rightarrow e_{\downarrow I} <R>_{\downarrow J}$　　$<L>_{\uparrow x \downarrow y} \rightarrow e_{\downarrow I} <R>_{\downarrow J}$

$<L>_{\uparrow x \downarrow y} \rightarrow <H>_{\downarrow z \uparrow w}$　　$<L>_{\uparrow x \downarrow y} \rightarrow <H>_{\downarrow z \uparrow w}$

2）具有相同值的属性取相同的属性名。

具有简单赋值形式的属性变量名取相同的属性名，可删去属性求值规则。

$<S> \rightarrow I_{\uparrow a} <B>_{\downarrow b} <C>_{\downarrow c}$　b,c:= a

$<S> \rightarrow I_{\uparrow x} <B>_{\downarrow x} <C>_{\downarrow x}$

下面通过一个例子，较详细地介绍如何构造属性文法 的递归下降翻译器。

例：有如下属性翻译文法G[< S>]

1. $< S >_{\downarrow R_1} \rightarrow a \uparrow_{T_1} < A > \uparrow_{Q_1} @ x \downarrow_{T_2, R_2} < S > \downarrow_{Q_2}$ $R_2 := R_1$ $T_2 := T_1$ $Q_2 := Q_1$

2. $< S >_{\downarrow R_1} \rightarrow b @ z \downarrow_{R_2}$, $R_2 := R_1$

3. $< A > \uparrow_P \rightarrow C \uparrow_{U_1} @ y \downarrow_{U_2} < A > \uparrow_Q < S > \downarrow_z @ v \downarrow_P b$

$U_2 := U_1, P := Q + U_1, z := U_1 - 3$

4. $< A > \uparrow_P \rightarrow @ w$ $P := 8$

对简单赋值形式的属性变量取相同的属性名，其求值规则可以删去。开始符号的继承属性 R_1=7。

1. $<S>_{\downarrow R} \rightarrow a\uparrow_T <A>\uparrow_Q @\, x_{\downarrow T,R} <S>_{\downarrow Q}$

2. $<S>_{\downarrow R} \rightarrow b\, @\, z_{\downarrow R}$,

3. $<A>\uparrow_P \rightarrow C\uparrow_u @\ y_{\downarrow u} <A>\uparrow_Q <S>_{\downarrow z} @\, v_{\downarrow P}\, b$

   $P := Q + U,\ z := U-3$

4. $<A>\uparrow_P \rightarrow @\, w \qquad P := 8$

全局变量和过程声明：

CLASS; /* 存放单词类别码 */

TOKEN; /* 存放单词值 */

NEXTSYM; /*词法分析程序，每调用一次单词类别码 ⇒ CLASS,
单词值 ⇒ TOKEN,该符号指针指向下一个单词 */

主程序：

NEXTSYM;

PROCS(7);

if CLASS ≠ 右界符 then ERROR;

ACCEPT

过程 PROCS(R)

R; /* 值形参声明 */

```
    case     CLASS      of
    a:       P1 ;
    b:       P2 ;
    其它：   ERROR;
    end      of      case;
P1 :                                  /* 产生式1的代码 */
    T ， Q ;                          /* 局部变量声明 */
    T := TOKEN;                       /* 单词值赋给终结符的综合属性 */
    NEXTSYM;
    PROCA(Q)
    OUT(X ↓T,R）；
    PROCS(Q)
    RETURN;
```

P_2: /* 产生式2的代码 */

NEXTSYM;

OUT(Z ↓R);

RETURN;

过程 PROCA(P)

P; /*变量形参声明*/

case CLASS of

C : p3

其它： p4

end of case;

P3:

```
U , Q , Z ;        /*局部变量声明*/
U := TOKEN;
NEXTSYM;
Z := U – 3 ;       插在U已知，使用Z之前
OUT(y↓U);
PROCA(Q);          返回时有值
P := Q+U;          插在Q,U已知，使用P之前。
PROCS(Z);
OUT(V↓P);
if  CLASS ≠ b  then  ERROR;
NEXTSYM;
RETURN;
```

P4:

```
P := 8;
OUT(w);
RETURN;
```

一个例子

例：构造将算术表达式翻译成四元式的属性翻译文法，并写出递归下降分析程序。由该属性翻译文法来描述翻译过程。

翻译的输入：　算术表达式　　a + b

翻译的输出：　四元式　　ADD , P_a , P_b , P_r

其中P_a , P_b , P_r 为变量a , b和结果单元的地址。

表达式：　(a + b) * c

输入：　(Id $\uparrow_1$ + Id $\uparrow_2$) * Id $\uparrow_4$

Id由词法分析程序返回，

$\uparrow_1$.....综合属性，变量在数据区地址。

输出：　ADD, 1, 2, 3

MULT, 3, 4, 5

数据区：

((1))翻译文法设计：

E →E+T@ADD　　T →F

E →T　　F →(E)

T →T*F @MULT　　F →Id

@ADD为输出ADD四元式的动作符号

@MULT为输出MULT四元式的动作符号

} 对应于完成翻译的语义动作程序

在文法中的插入位置：在分别处理完成两个操作数之后

输入序列：(a+b)*c

翻译文法产生的活动序列: (a+b@ADD)*c@MULT

动作符号序列：@ADD　@MULT　　反映生成四元式的顺序，语法分析过程中语义程序的调用顺序。

(2) 属性翻译文法的设计

- 输入符号（操作数）有一个综合属性，它是该符号在数据区的地址。
- 每个非终结符有一个综合属性，该属性是由它产生的代表该子表达式在数据区的地址。
- 动作符号有三个继承属性，它们分别是左右操作数和运算结果在数据区地址。

这样可得表达式的属性翻译文法

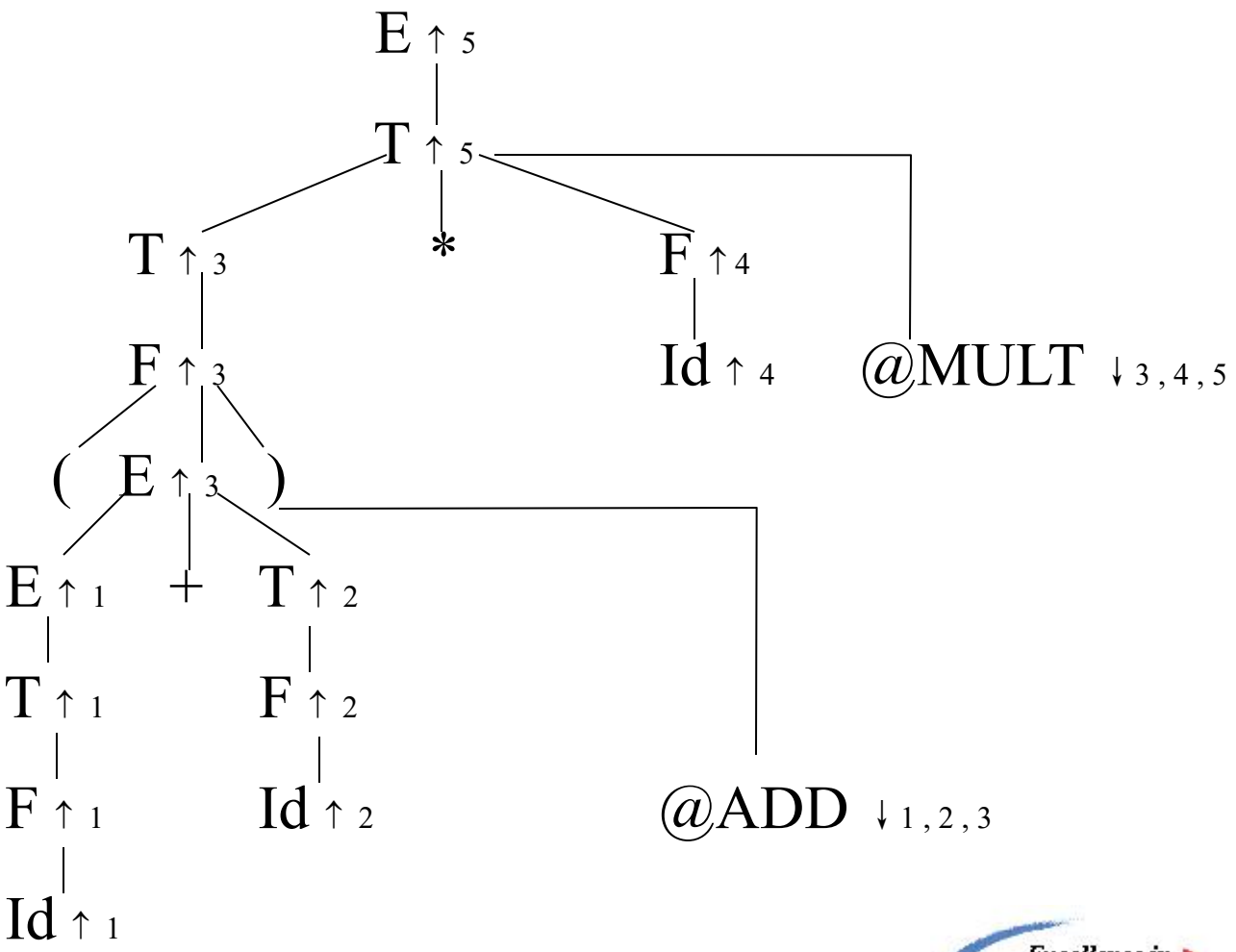
——可将中缀表达式翻译成四元式

1. $E\uparrow_x \rightarrow E\uparrow_q + T\uparrow_r @ADD\downarrow_{y,z,p}$ x, p := NEW y := q z := r
2. $E\uparrow_x \rightarrow T\uparrow_p$ x := p
3. $T\uparrow_x \rightarrow T\uparrow_q * F\uparrow_r @MULT\downarrow_{y,z,p}$ x, p := NEW y := q z := r
4. $T\uparrow_x \rightarrow F\uparrow_p$ x := p
5. $F\uparrow_x \rightarrow (E\uparrow_p)$ x := p
6. $F\uparrow_x \rightarrow Id\uparrow_p$ x := p

说明：

Id的综合属性p是数据区地址，NEWT为系统过程，返回数据区地址。

反映属性求值的语法树：



语义动作程序如何设计在后面介绍

@ADD ↓ y , z , p ⇒ fprintf(objfile, “ADD %d %d %d \n”, y, z, p)

((3))写递归下降翻译程序

(留作作业）

小结：

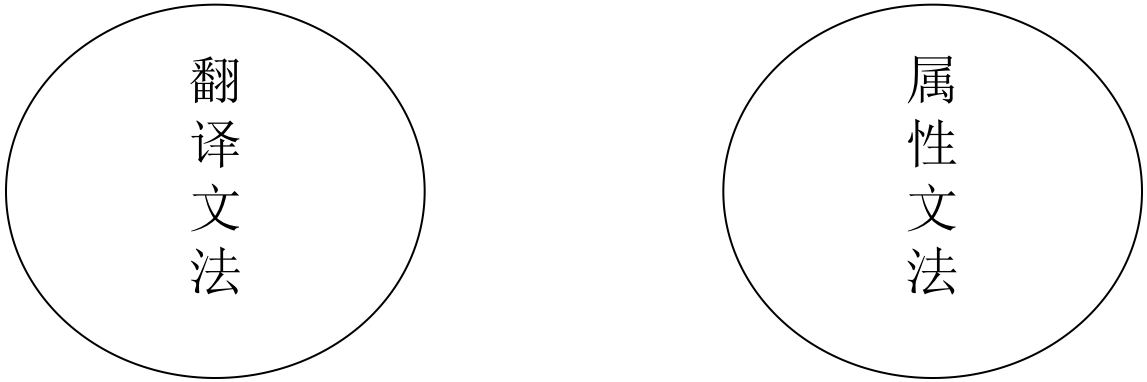
本章介绍了语法制导翻译的概念和技术，是在抽象层次上讲的。

第十章对过程语言的编译就是采用该法。

翻译文法：在输入文法中插入语义动作符号（完成翻译任务的语义子程序）

属性翻译文法：文法符号（包括动作符号）可带有属性，并定义相应的属性求值规则，就成为属性翻译文法。比翻译文法能更细地描述翻译过程。（属性有综合属性和继承属性之分）

自顶向下的语法制导翻译（递归下降翻译）



在本章的基础上，第十章将介绍典型的过程语言的语法制导翻译。

本章未讲的部分不要求

第十章　语义分析和代码生成

- 语义分析的概念
- 栈式抽象机及其汇编指令
- 声明的处理
- 表达式的处理
- 赋值语句的处理
- 控制语句的处理
- 过程调用和返回

假定：

- 源语言：　通用的过程语言
- 生成代码：栈式抽象机的（伪）汇编程序
- 翻译方法：自顶向下的属性翻译
- 语法成分翻译子程序参数设置：
  - 继承属性为值形参
  - 综合属性为变量形参
- 语法成分翻译动作子程序参数设置：
  - 继承属性为值形参
  - 综合属性不设形参，而作为动作子程序的返回值（由RETURN语句返回）

10.1 语义分析的概念

1、上下文有关分析：即标识符的作用域
2、类型的一致性检查
3、语义处理：

声明语句：其语义是声明变量的类型等，并不要求做其他的操作。
编译程序的工作是填符号表，登录名字的特征信息，分配存储。

执行语句：语义是要做某种操作。
语义处理的任务：按某种操作的目标结构生成代码。

用上下文无关文法只能描述语言的语法结构，而不能描述其语义。

例如，对于有嵌套子程序结构的程序段：

BEGIN … BEGIN α INT I β I END … I … END

若存在文法规则：VAR ::= I

BEGIN … <BLOCK> … I ... END

BEGIN … δ VAR ... END

第一次I的归约正确
第二次I的归约错误

δ ∈V*且不包含变量I的声明

文法规则应改为：INT I β VAR ::= INT I β I

然而上下文有关文法不仅构造困难，而且其分析器十分复杂，分析效率又低，显然是不实用的

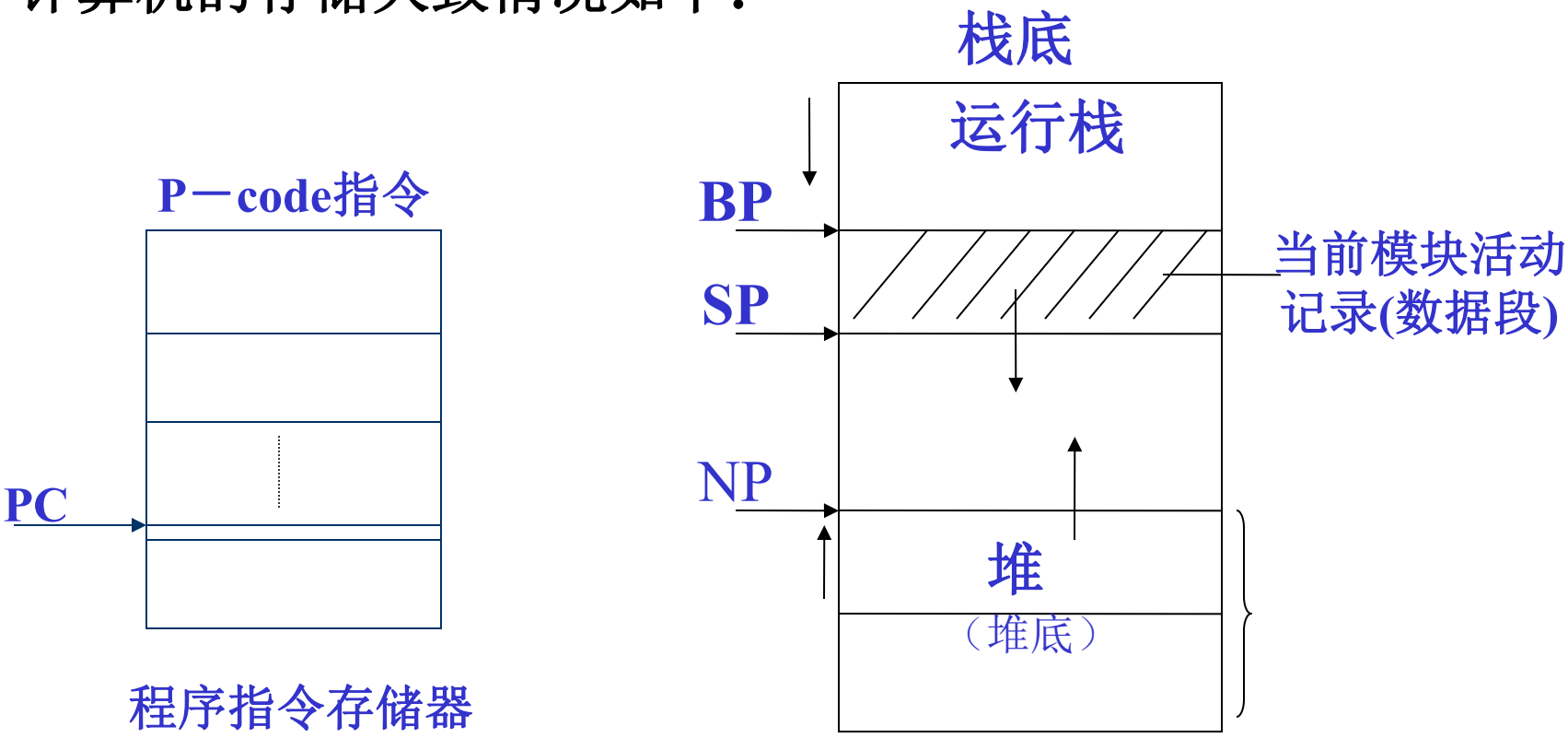
因此，通常我们把与语义相关的上下文有关信息填入符号表中，并通过查符号表中的这些信息来分析程序的语义是否正确

10.2 栈式抽象机及其汇编指令

栈式抽象机：由三个存储器、一个指令寄存器和多个地址寄存器组成。

存储器：
- 数据存储器 （存放AR的运行栈）
- 操作存储器 （操作数栈）
- 指令存储器

计算机的存储大致情况如下：



栈式抽象机指令代码如下：

指令名称	操作码	地址	指令意义
加载指令	LOD	D	将 D 的内容→栈顶
立即加载	LDC	常量	常量→栈顶
地址加载	LDA	(D)	变量 D 的地址→栈顶
存储	STO	D	栈顶内容 存入 →变量D
间接存	ST	@D	将栈顶内容→D 所指单元
间接存	STN		将栈顶内容→次栈顶所指单元
加	ADD		栈顶和次栈顶内容相加，结果留栈顶
减	SUB		次栈顶内容减栈顶内容
乘	MUL		
·····			

指令名称	操作码	地址	指令意义
等于比较	EQL		次栈顶内容与栈顶内容比较，结果（1 或 0）留栈顶
不等比较	NEQ		
大于比较	GRT		
小于比较	LES		
大于等于	GTE		
小于等于	LSE		
逻辑与	AND		
逻辑或	ORL		
逻辑非	NOT		
转子	JSR	lab	
分配	ALC	M	在运行栈顶分配大小为 M 的活动记录区

10.3 声明的处理

语义的表示：

给出语言结构的属性翻译文法来说明其语义及语义动作，并把这些语义动作插入属性翻译文法产生式中的适当位置。

编译程序的任务：

- 编译程序处理声明语句要完成的主要任务为：

  1) 分离出每一个被声明的实体，并把它们的名字填入符号表中
  2) 把被声明实体的有关特性信息尽可能多地填入符号表中

- 对于已声明的实体，在处理对该实体的引用时要做的事情：

  1) 检查对所声明的实体引用（种类，类型等）是否正确
  2) 根据实体的特征信息，例如类型，所分配的目标代码地址（可能为数据区单元地址，或目标程序入口地址）生成相应的目标代码

声明有常量声明，变量（包括简单变量，数组变量和记录变量等）和过程（函数)声明等，这里主要讨论常量声明和简单变量、数组声明的处理。

声明的两种方式：

(1) 类型说明符放在变量的前面。如：C语言： int a;
在填表时已知类型和a的值（名字）：直接填入符号表。

(2) 类型说明符放在变量的后面，如：Pascal, PL/1，Ada等，需要返填。

如PL/I声明语句：

DECLARE(X, Y(N), YTOTAL) FLOAT;

声明语句的输入文法为：

<declaration>→DECLARE ‘ (’<entity list>‘)’<type>
<entity list>→<entity name> | <entity name> , <entity list>
<type>→FIXED | FLOAT | CHAR

属性翻译文法为：

<declaration>→DECLARE @dec_on $\uparrow_x$‘ (’<entity list> ‘)’
<type> $\uparrow_t$ @fix_up $\downarrow_{x,t}$
<entity list>→<entity name> $\uparrow_n$ @name_defn $\downarrow_n$
| <entity name> $\uparrow_n$, @name_defn $\downarrow_n$ <entity list>
<type> $\uparrow_t$→FIXED $\uparrow_t$ | FLOAT $\uparrow_t$ | CHAR $\uparrow_t$

动作程序

@dec_on $\uparrow$ x 是把符号表当前可用表项的入口地址（指向符号表入口的指针，或称 表项下标值）赋给属性变量**x** 。

@name_defn $\downarrow$ n是将由各实体名所得的**n**继承属性值，依次填入从**x**开始的符号表中。

注：显然应有内部计数器或内部指针，指向下一个该填的符号表项。

@fix_up $\downarrow$ x, t 是将类型信息**t** 和相应的数据存储区分配地址填入从 **x** 位置开始的符号表中。（反填）

当然，如果声明语句中，类型说明符放在头上，就无需“反填”处理了。

10.3.1 常量类型声明处理

常量标识符通常被看作是全局名。

常量声明的ATG如下：

<const del> →constant <type> ↑t <entity> ↑n :=<const expr> ↑c, s
@insert ↓t, n, c, s ;

<type> ↑t → real ↑t |integer ↑t |string ↑t

<const expr> ↑c, s →<integer const > ↑c, s |<real const> ↑c, s
|<string const> ↑c, s

由该文法产生的一个声明实例为：

constant integer SYMBSIZE := 1024;

翻译处理过程为：

<const del> →constant <type> ↑t <entity> ↑n :=

<const expr> ↑c, s @insert ↓ t，n，c，s ;

先识别类型（integer），将它赋给属性t；然后识别常量名字（SYMBSIZE), 将它赋给属性n；最后识别常量表达式，并将其值赋给c，其类型赋给属性s 。

★ @insert 的功能是：

① 检查声明的类型t 和常量表达式的类型s 是否一致，若不一致，则输出错误信息

② 把名字n，类型t 和常量表达式的值c 填入符号表中

10.3.2 简单变量声明处理

ATG文法：

<svar del> → <type> ↑t , i <entity> ↑n @svardef ↓ t, i, n
@allocsv ↓i ;

<type> ↑t, i → real ↑t, i |integer ↑t, i |character ↑t (<number>) ↑i
| logical ↑t, i

n: 变量名
t: 类型值
i: 该类型变量所需
数据空间的大小

简单变量声明的例子：

```
real   x ;
integer  j;
character  ( 20 )  s ;
```

@svardef动作符号是把n，i 和t 填入符号表中。

```
procedure  svardef( t, i, n );
    j := tableinsert ( n, t, i  );      /*将有关信息填入符号表*/
    if   j = 0                          //填表时要检查是否重名
    then errmsg ( duplident , statementno);
    else  if   j = -1                   //符号表已满
          then errmsg( tblovflow, statementno);
end svardef;

procedure allocsv( i );
    codeptr := codeptr + i ;    //codeptr 为分配地址指针
end allocsv;
```

@allocsv 和 @svardef 可以合并

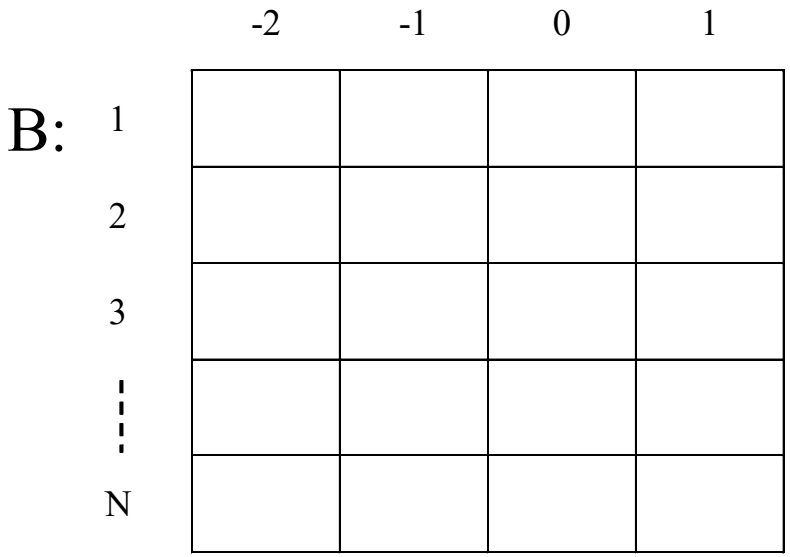
对于变长字符串（或其它大小可变的数据实体），往往需要采用动态申请存储空间的办法把可变长实体存储在堆中。我们可通过指向存放该实体数据区的指针来引用该实体，有时还应得到该实体存储空间的大小信息，并一起填入符号表内。

10.3.3 数组变量声明的处理

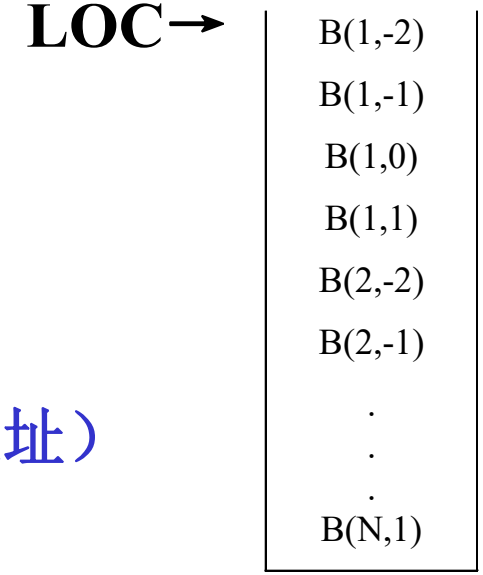
对于静态数组，即数组的大小在编译时是已知的，编译程序在处理数组声明时，可建立一个数组模板(又称为数组信息向量）以便以后的程序中引用该数组元素时，可按照该模板提供的信息，计算数组元素(下标变量）的存储地址。

对于动态数组，其大小只有在运行时才能最后确定。我们在编译时仅为该模板分配一个空间，而模板本身的内容将在运行时才能填入。

大部分程序设计语言，数组元素是按行（优先）存放在存储器中的*，如声明数组 **array B (N, -2: 1) char ;**



实际数组B各元素的存储次序为：



LOC是数组首地址
(该数组第一个元素的地址）

*** FORTRAN 例外，**
它按列（优先）存放数组元素

a）n维数组的地址计算公式

设数组的维数为n， 各维的下界和上界为L(i）和U(i)

例如， 上例二维数组B

L(1) = 1 (隐含值）, U(1) = N

L(2) = -2 , U(2) = 1

还假定n维数组元素的下标为V(1), V(2),…, V(n）

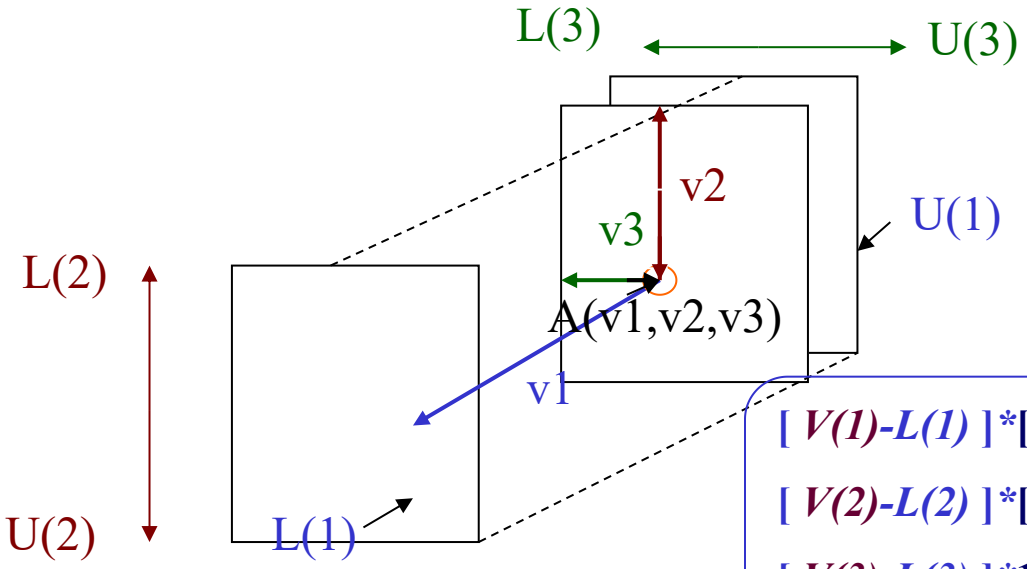
则该数组元素的地址计算公式为：

$$ADR = LOC + \sum_{i=1}^{n} [V(i) - L(i)] \times P(i) \times E$$

注：E为数组元素大小（字节数）

其中

$$P(i) = \begin{cases} 1 & \text{当i = n时} \\ \prod_{j=i+1}^{n} [U(j) - L(j) + 1] & \text{当1 ≤i < n 时} \end{cases}$$



P(1)

[*V(1)-L(1)*]*[U(2)-L(2)+1]*[U(3)-L(3)+1]**E*+

[*V(2)-L(2)*]*[U(3)-L(3)+1]**E*+

[*V(3)-L(3)*]*1**E*

P(2)

P(3)

$$ADR = LOC + \sum_{i=1}^{n} [V(i) - L(i)] \times P(i) \times E$$

其中

$$P(i) = \begin{cases} 1 & \text{当 } i = n \text{ 时} \\ \prod_{j=i+1}^{n} [U(j) - L(j) + 1] & \text{当 } 1 \le i < n \text{ 时} \end{cases}$$

若令 $RC = -\sum_{i=1}^{n} L(i) \times P(i) \times E$ (不变部分)

则地址 $ADR = LOC + RC + \sum_{i=1}^{n} V(i) \times P(i) \times E$

RC为数组元素地址计算公式中的不变部分。因为，只要知道数组的维数和每一维的上下界值，便可求得RC值。

以前面所举的二维数组B为例，若N = 3

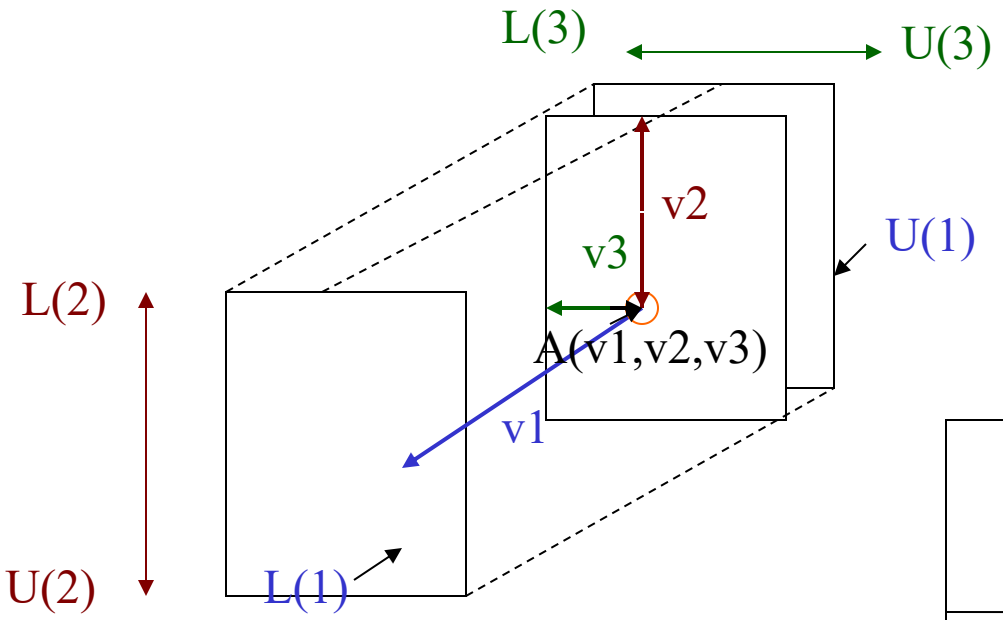
则 P(1) = [U(2) - L(2) +1]
= 1- (-2)+1
= 4

P(2) = 1

$RC = -\sum_{i=1}^{2} L(i) P(i) * E$
= -[1×4 +(-2)×1]×E
= -2E

因此， 若有数组元素B(2 , 1), 则它的地址为：

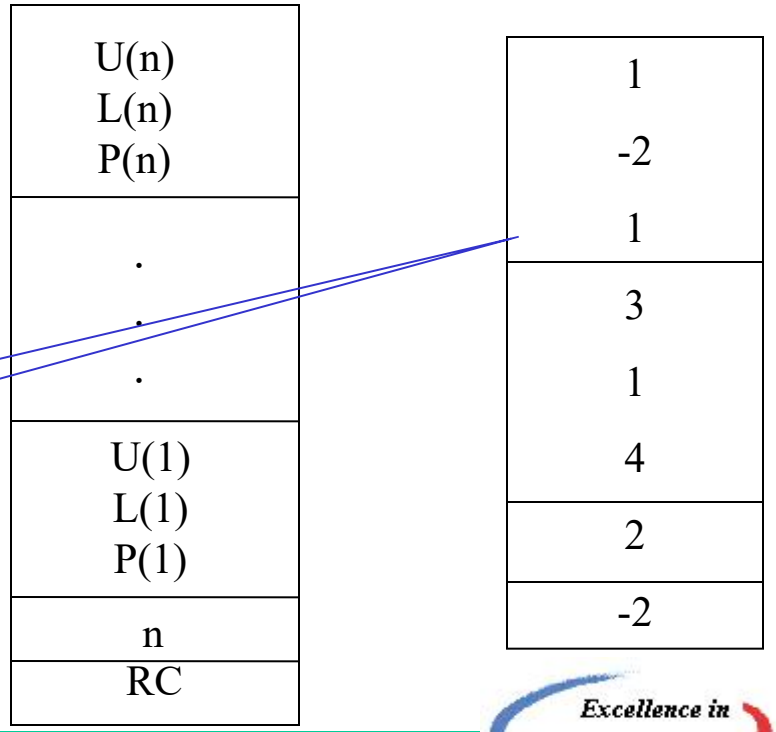
$ADR = LOC - 2E + \sum_{i=1}^{2} V(i) \times p(i) \times E$ = LOC – 2E+(2×4+1×1)×E
=LOC + 7×E



数组模板的一般形式如下左图所示，而对于数组B的模板如下右图所示：

array B (3, -2: 1) char ;

三维数组的例子

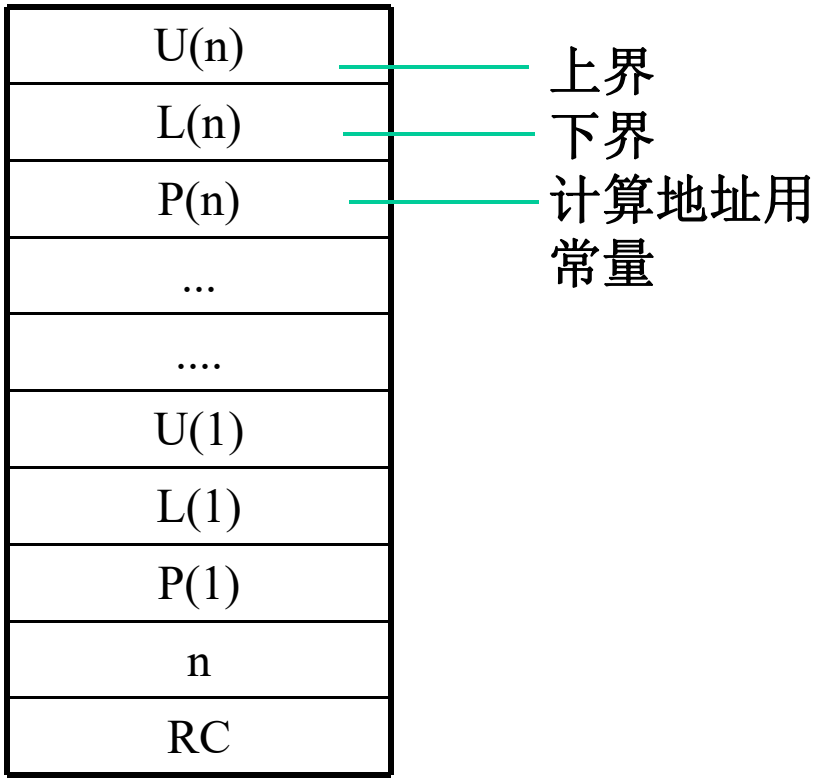


数组信息向量表

b) 数组信息向量表（模板）

功能：1、用于计算下标变量地址
2、检查下标是否越界

一般形式：



大小：3n + 2

注：1、数组模板所需的空间大小取决于数组的维数，即3n+2

∴无论是常界或变界数组，在编译时就能确定数组模板的大小

2、常界数组，在编译时就可造信息向量表；而变界数组信息向量表要在目标程序运行时才能造。编译程序要生成相应的指令

以前面所举的二维数组B为例，若N = 3

P(2) = 1

P(1) = [U(2) - L(2) +1]
= 1- (-2)+1
= 4

$$RC = -\sum_{i=1}^{2} L(i)P(i)$$

= -[1×4 +(-2)×1]
= -2

数组信息向量表

1
-2
1
3
1
4
2
-2

U(2)--上界
L(2)--下界
P(2)--计算地址常量
U(1)--上界
L(1)--下界
P(1)--计算地址常量
n---维数
RC

数组声明的ATG文法：

<array del> → array $\uparrow_k$ @init $\uparrow_j$ <entity> $\uparrow_n$ (<sublist> $\uparrow_j$)
<type> $\uparrow_t$ @symbinsert $\downarrow_{j, n, t}$

<sublist> $\uparrow_j$ → <subscript> @dimen# $\uparrow_j$
| <subscript>, <sublist> $\uparrow_j$ @dimen# $\uparrow_j$

<subscript> → <integer expr> $\uparrow_u$ @bannds $\downarrow_u$
| <integer expr> $\uparrow_l$: @lowerbnd $\downarrow_l$
<integer expr> $\uparrow_u$ @upperbnd $\downarrow_{u, l}$

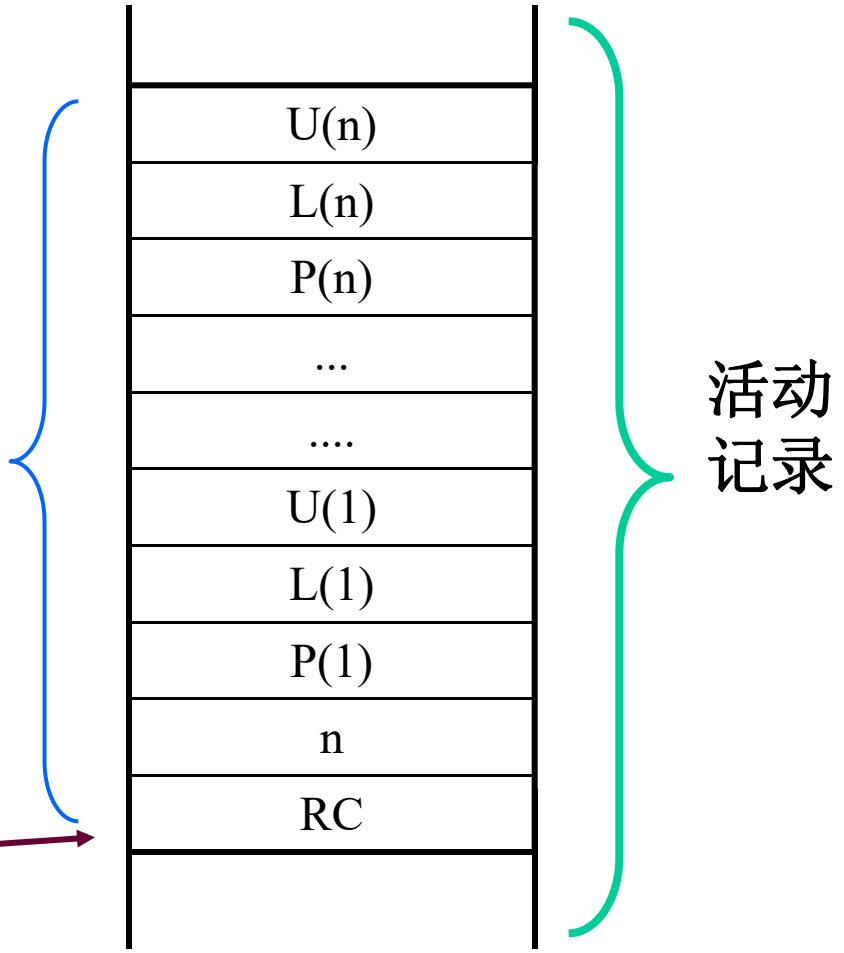
1) 动作程序 @init 的功能为在分配给数组模板区中保留两个存储单元，用来放 RC 和 n， 并将维数计数器 j 清0。

2) @dimen# $\uparrow_j$: j := j + 1, 即统计维数

1) @init:

p:= p+2;

j := 0; **/*维数计数器*/**



3) @bannds将省略下界表达式情况的u => U(i),但应把相应的L(i)置成隐含值1, 然后计算P(i)

实际P(i)计算公式可利用 P(i)＝ [U(i+ 1) - L(i +1) +1]×P(i+1)

$$P(i)=\begin{cases}1 & \text{当} i=n \text{ 时}\\ \prod_{j=i+1}^{n}[U(j)-L(j)+1] & \text{当} 1\le i<n \text{ 时}\end{cases}$$

注：由于P(i)的计算要依赖于P(i +1), 所以实际P(i)的值是反填的

4) @lowerbnd 把 l => L(i)

@upperbnd 把 u => U(i), 并计算 P(i)

5) 最后的动作程序@symbinsert是把数组名n，数组维数 j 和数组

元素类型 t 及数组标志 k 填入符号表中；为数组分配存储空间

对于变界数组：

4) @lowerbnd ↓ l

生成将 l => L(i) 的代码

@upperbnd ↓ u

生成 把 u => U(i)的代码,

生成计算 P（i ）的代码；

生成将P（i ）的值送模板区的代码；

5) @symbinsert ↓ j, n, t

a) 把n, j, t, 填入符号表中

b) 生成调用运行子程序代码（计算RC, 并将计算结果和数组名一起存入模板区；计算数组所需数据区大小，为数组分配存储空间，并将头地址填入符号表。）

10.4 表达式的处理

分析表达式的主要目的是生成计算该表达式值的代码。通常的做法是把表达式中的操作数装载（LOAD）到操作数栈（或运行栈）栈顶单元或某个寄存器中，然后执行表达式所指定的操作，而操作的结果保留在栈顶或寄存器中。

注：操作数栈即操作栈，它可以和前述的运行栈（动态存储分配）合而为一，也可单独设栈。

本章中所指的操作数栈实际应与动态运行（存储分配）栈分开。

请看下面的整型表达式ATG文法：

1.<expression>→<expr>
2.<expr>→<term><terms>
3.<terms>→ ε
4. | +<term>@add<terms>
5. | - <term>@sub<terms>
6.<term>→<factor><factors>
7.<factors>→ ε
8. | *<factor>@mul<factors>
9. | /<factor>@div<factors>
10.<factor>→<variable>↑n@lookup↓n↑j
@push↓j
11. | <integer>↑i@pushi↓i
12. | (<expr>)

有关的语义动作为：

```
procedure add;
    emit('ADD');
end;
```

```
procedure mul;
 emit('MUL');
end;
```

```
procedure lookup(n);
    string n; integer j;
    j:= symblookup( n);
     /*名字n表项在符号表中的位置*/
    if j < 1
    then  /*error*/
    else  return (j);
end;
```

```
procedure push(j);
   integer j;
   emit('LOD', symbtbl (j).objaddr);
end;
```

```
procedure pushi(i); /*压入整数*/
   integer i;
   emitl('LDC', i) ;
end;
```

对于输入表达式 x + y * 3, 可以生成如下目标代码：

```
LOD, < ll, on> x
LOD, < ll, on> y
LDC, 3
MUL
ADD
```

上面所述的表达式处理实际上忽略了出现在表达式中各操作数类型的不同，且变量也仅限于简单变量。

下面假定表达式中允许整型和实型混合运算，并允许在表达式中出现下标变量（数组元素）。因此应该增加有关类型一致性检查和类型转换的语义动作，也要相应产生计算下标变量地址和取下标变量值的有关指令。

<expression>→<expr> ↑t

<expr> ↑t →<term> ↑s <terms> ↓s ↑t

<terms> ↓s ↑u →@echo ↓s ↑u

| + <term> ↑t @add ↓t, s ↑v <terms> ↓v ↑u

| - <term> ↑t @sub ↓t, s ↑v <terms> ↓v ↑u

<term> ↑u →<factor> ↑s <factors> ↓s ↑u

<factors> ↓s ↑u → @echo ↓s ↑u

| *<factor> ↑t @mul ↓t, s ↑v <factors> ↓v ↑u

| /<factor> ↑t @div ↓t, s ↑v <factors> ↓v ↑u

<factor> ↑t →<variable> ↑i @type ↓i ↑t

| <integer> ↑i @pushi ↓i ↑t

| <real> ↑r @pushi ↓r ↑t

<variable> ↑j →<identifier> ↑n @lookup ↓n ↑j @push ↓j

| <identifier> ↑n @lookup ↓n ↑j (@template ↓j ↑k <sublist> ↓k, j)

<sublist> ↓k, j → <subscript> ↑t @offset ↓k, t ↑i <subscripts> ↓i, j

<subscripts> ↓k, j →@checkdim ↓k, j

| , <subscript> ↑t @offset ↓k, t ↑i <subscripts> ↓i, j

<subscript> ↑t → <expr> ↑t

语义动作add等应作相应改变：

```
procedure add( t, s);
  string t,  s;
  if t = ‘real ’ and s = ‘integer’
  then begin
         emit( ‘CVN’); /*次栈顶转为实数*/
         emit( ‘ADD’);
         return ( ‘real’);
        end;
  if t = ‘integer ’ and s = ‘real’
  then begin
         emit( ‘CNV’); /*栈顶转为实数*/
         emit( ‘ADD’);
         return ( ‘real’);
        end;
   emit( ‘ADD’);
   return ( t);
end;
```

```
procedure offset( k, t );
    integer k;  string t;
    k := k+1;
    if t ≠ ‘ integer’
     then errmsy(‘数组下标应为整
                型表达式’ , statno);
     else  emitl( ‘OFS’, k );
     return (k);
end;
```

```
procedure checkdim( k, j);
    integer k, j;
    if k ≠ symbtbl (j).dim
    then errmsy(‘数组维数与
          声明不匹配’ , statno);
    else  begin
           emit( ‘ARR’);
           emit( ‘DER’);
          end;
end;
```

```
procedure template(j);
    integer j;
    emitl( ‘TMP’, symbtbl (j). objaddr);
    k:= 0; /*维数计数器初始化*/
    return(k);
end;
```

★过程template发送一条目标机指令‘TMP’, 该指令把数组的模板地址加载到操作数栈顶，并将下标（维数）计数器k清0。

★ offset过程要确保每一个下标都是整型，而且发送一条‘OFS’指令，该指令在运行时要完成以下功能：

1. 检查第k个下标值是否在栈顶并是否在上下界范围内

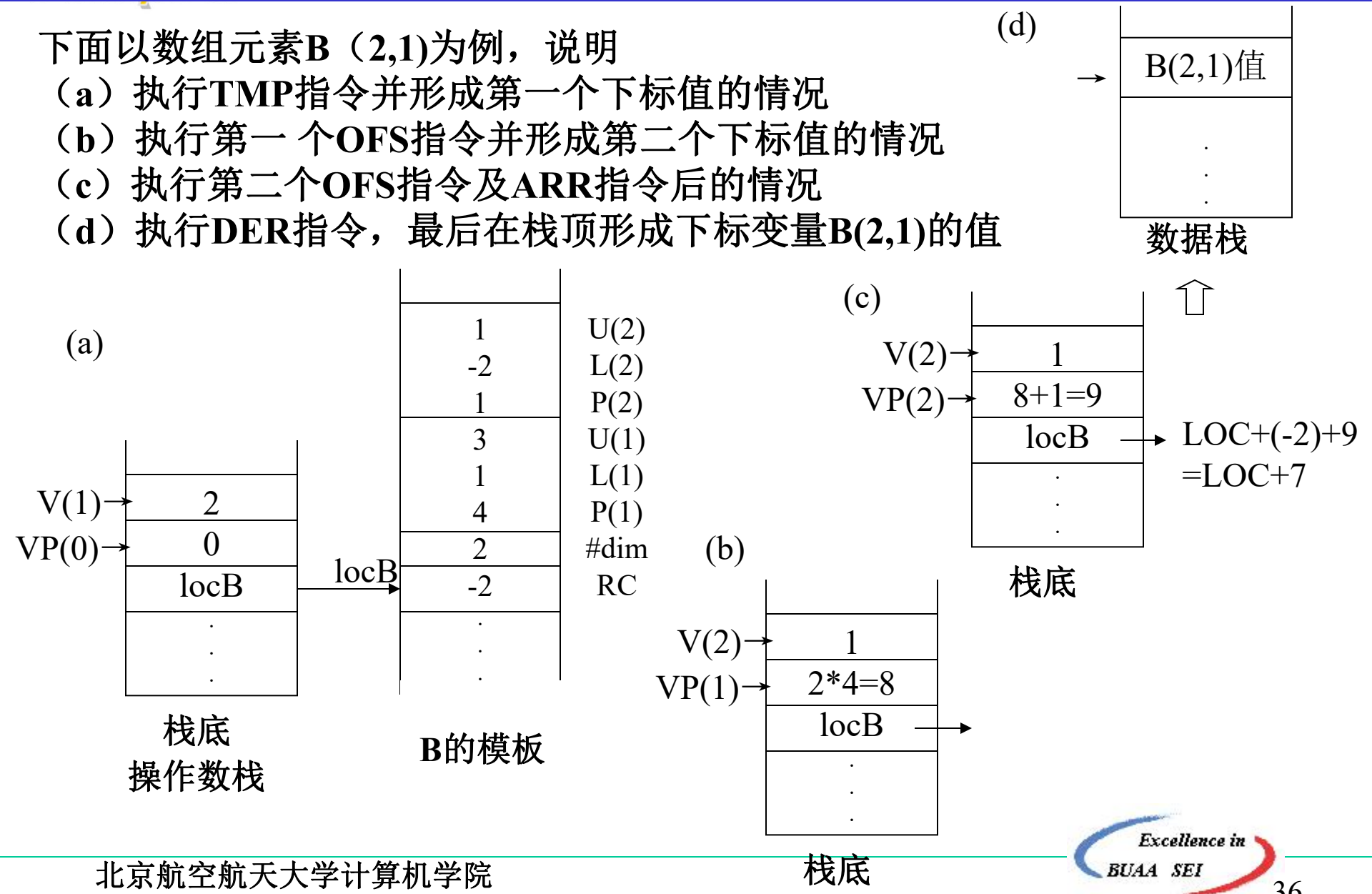
2.使用下列递归函数，计算地址计算公式中可变部分：

VP(0) = 0;

VP(k) = VP(k-1) + V(k) *P(k)　　1≤k≤n

该VP函数是由计算公式 $\sum_{k=1}^{n} V(k) \times P(k)$ 导出的

下面以数组元素B（2,1)为例，说明
（a）执行TMP指令并形成第一个下标值的情况
（b）执行第一 个OFS指令并形成第二个下标值的情况
（c）执行第二个OFS指令及ARR指令后的情况
（d）执行DER指令，最后在栈顶形成下标变量B(2,1)的值



处理逻辑表达式(关系表达式）的方法与处理算术表达式的方式基本相同。下面是逻辑表达式～(x =y & y≠z | z＜ x) 生成的指令序列：

```
LOD,  ( ll, on )x
LOD,  ( ll, on )y
EQL
LOD,  ( ll, on )y
LOD,  (ll, on )z
NEQ
AND
LOD,  ( ll, on )z
LOD,  ( ll, on )x
LES
ORL
NOT
```

10.5 赋值语句的处理

X:= Y+X;

```
LDA  ( ll, on )x
LOD  ( ll, on )y
LOD  ( ll, on )x
ADD
STN
```

<assignstat > → @setL ↑L<variable> ↓L ↑t:=
@resetL ↑L<expr> ↑s@storin ↓t,s ;

@setL是设置变量为“左值”（被赋变量），即将属性L置true

@resetL是设置变量为非被赋变量，即把属性L置成false

```
procedure setL;
    return (true );
end;
```
指示取变量地址

```
procedure resetL;
    return (false );
end;
```
指示取变量之值

```
procedure storin(t,s);
  string t, s;
  if t ≠ s
  then /*生成进行类型转换的指令*/
  emit('STN');
end;
```

10.6 控制语句的处理

10.6.1 if语句

JMF, labA

IF <log_expr> THEN <stat_list> labA:

JMF, labA

IF <log_expr> THEN <stat_list> ELSE labA: <stat_list> labB:

JMP, labB

其属性翻译文法及相应的语义动作程序：

1. <if_stat>→<if_head>$\uparrow_y$ <if_tail>$\downarrow_y$

2. <if_head>$\uparrow_y$→IF <log_expr>@brf$\uparrow_y$ THEN <stat list>

3. <if_tail>$\downarrow_y$→@labprod$\downarrow_y$
   |ELSE @br$\uparrow_z$@labprod$\downarrow_y$<stat_list>@labprod$\downarrow_z$

动作程序@brf的功能是生成JMF指令，并将转移标号返回给属性y

```
procedure  brf;
   string labx;
   labx := genlab;
            /*产生一标号赋给labx*/
   emitl('JMF', labx);
   return (labx);
end;
```

动作程序@labprod是把从继承属性y得到的标号设置到目标程序中

```
procedure labprod(y);
   string y;
   setlab(y);
/*在目标程序当前位置设标号*/
end;
```

动作程序@br是是生成JMP指令，并将转移标号返回给属性z

```
procedure br;
   string labz;
   labz := genlab;
   emitl( 'JMP', labz);
   return( labz);
end;
```

10.6.4 for 循环语句

for 语句例子：

```
for i:= 1 to n by z do
    <statement>
    ...
end for;
```

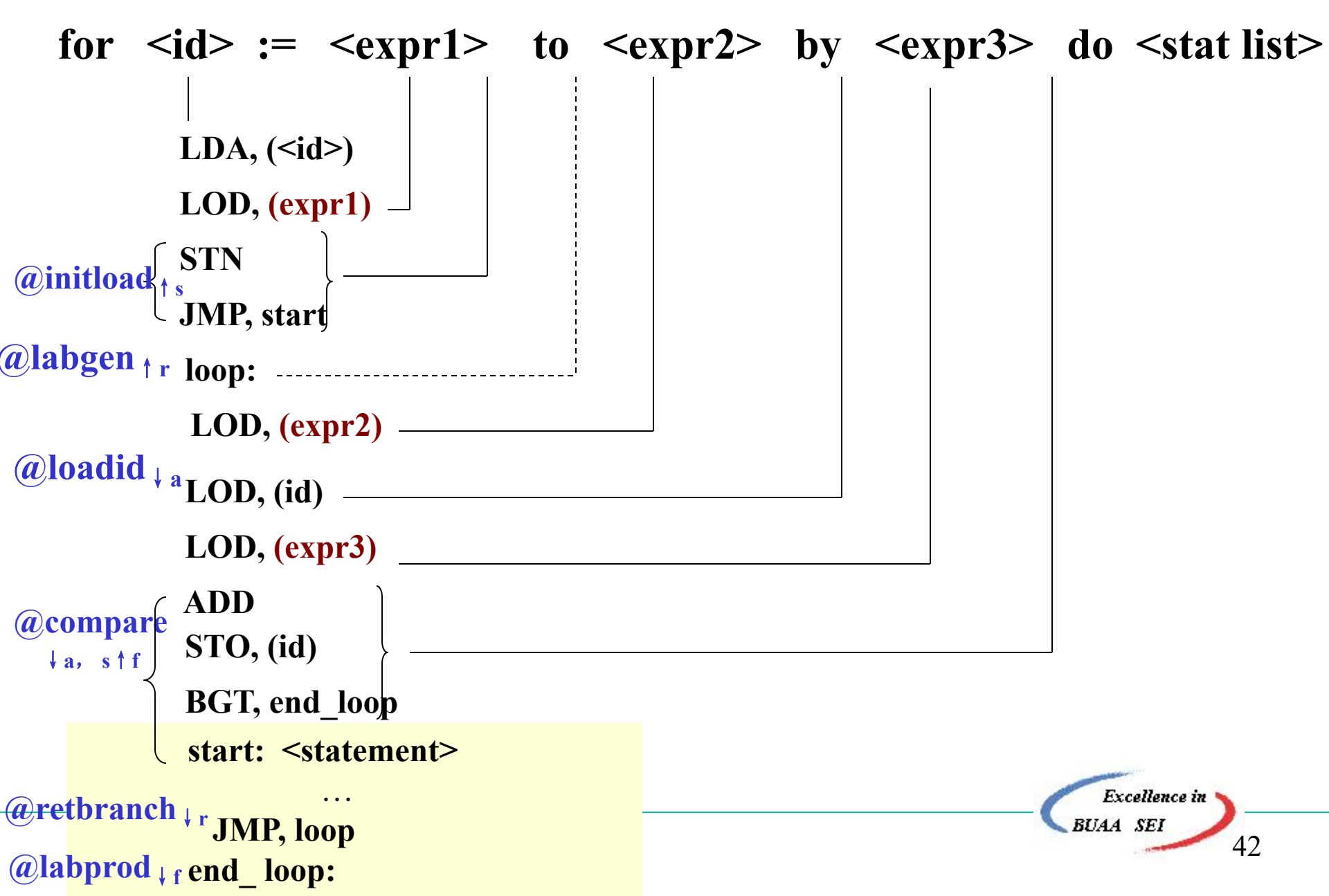
ATG文法

1.<for loop>→<for head>$\uparrow_{a,f,r}$ < rest of loop>$\downarrow_{a,f,r}$

2.<for head>$\uparrow_{a,f,r}$ →for <id>$\uparrow_{a}$:= <expr> @initload$\uparrow_{s}$
to @labgen$\uparrow_{r}$ <expr> by
@loadid$\downarrow_{a}$<expr> @compare$\downarrow_{a,s}\uparrow_{f}$

3.<rest of loop>$\downarrow_{a,f,r}$ →do <stat list> end for
@retbranch$\downarrow_{r}$ @labemit$\downarrow_{f}$

@initload 只生成给循环变量赋初值的指令。


for <id> := <expr1> to <expr2> by <expr3> do <stat list>
LDA, (<id>)
LOD, (expr1)
@initload ↑s
STN
JMP, start
@labgen ↑r
loop:
LOD, (expr2)
@loadid ↓a
LOD, (id)
LOD, (expr3)
@compare ↓a, s↑f
ADD
STO, (id)
BGT, end_loop
start: <statement>
...
@retbranch ↓r
JMP, loop
@labprod ↓f
end_ loop:

```
procedure labgen
    string r;
    r := genlab;
    setlab(r);
    return ( r );
end;
```

```
procedure loadid( a )
    address a;
    emitl( 'LOD', a);
end;
```

```
procedure  compare( a, s);
    address a;    string f, s;
    emit( 'ADD');
    emitl( 'STO', a );
    f := genlab;
    emitl( 'BGT', f );
    setlab( s );
    return( f );
end;
```

```
procedure  labprod( f )    // 即labemit
    string  f;
    setlab( f );
end;
```

10.7 过程调用和返回

10.7.1 参数传递的基本形式

1.传值（call by value)　— 值调用

实现：

调用段（过程语句的目标程序段）：

计算实参值 => 操作数栈栈顶

被调用段（过程说明的目标程序段）：

从栈顶取得值 => 形参单元

过程体中对形参的处理：

对形参的访问等于对相应实参的访问

特点：

数据传递是单向的

如C语言，
Ada语言的in参数，
Pascal 的值参数。

2.传地址（call by reference) — 引用调用

实现：

调用段：

计算实参地址 => 操作数栈栈顶

被调用段：

从栈顶取得地址 => 形参单元

如：FORTRAN, Pascal 的变量形参。

过程体中对形参的处理：

通过对形参的间接访问来访问相应的实参

特点：

结果随时送回调用段

如：ALGOL 的换名形参。

3. 传名（call by name）

又称“名字调用”。即把实参名字传给形参。这样在过程体中引用形参时，都相当于对当时实参变量的引用。

当实参变量为下标变量时，传名和传地址调用的效果可能会完全不同。

传名参数传递方式，实现比较复杂，其目标程序运行效率较低, 现已很少采用。

10.7.2 过程调用处理

与调用有关的动作如下：

1. 检查该过程名是否已定义（过程名和函数名不能用错） 实参和形参在类型、顺序、个数上是否一致。(查符号表）

2. 加载实参（值或地址）

3. 加载返回地址

4.转入过程体入口地址

例：有过程调用：

process_symb(symb, cursor,replacestr);

调用该过程生成的目标代码为：

LOD, (addr of symb)
LOD, (addr of cursor)
LOD, (addr of replacestr)
JSR, (addr of process_symb)
<retaddr>:….

传值调用

若实参并非上例中所示变量，而是表达式，则应生成相应计算实参表达式值的指令序列。

JSR指令先把返回地址压入操作数栈，然后转到被调过程入口地址。

设过程说明的首部有如下形式：

procedure process_symb(string:symbal, int: cur, string: repl);

则过程体目标代码的开始处应生成以下指令，以存储返回地址和形参的值。

```
ALC, 4 + x   /* x为定长项空间 */
STO, <actrec loc1>  /* 保存返回地址 */
STO, <actrec loc4>  /* 保存replacestr */
STO, <actrec loc3>  /* 保存cursor */
STO, <actrec loc2>  /* 保存symb */
```

过程调用时，实参加载指令是把实参变量内容（或地址）送入操作数栈顶，过程声明处理时，应先生成把操作数栈顶的实参送运行栈AR中形参单元的指令。

将操作数栈顶单元内容存入运行栈（动态存储分配的数据区）当前活动记录的形式参数单元。

可认为此时运行栈和操作数栈不是一个栈（分两个栈处理）

过程调用的ATG文法：

<proc call>→ <call head>↑i,z @initm↑m <args> ↓i,z @genjsr↓i

<call head>↑i,z → <id>↑n @lookupproc↓n↑i,z

<args>↓i,z → @chklength↓i,z | (<arg list>↓i,z)

<arg list>↓i,z → <expr>↑t @chktype↓t,i,m,z↑z <exprs>↓i,z

<exprs>↓i,z → @chklength↓i,z

| , <expr>↑t @chktype↓t,i,m,z↑z <exprs>↓i,z

```
procedure  lookupproc(n);
   string n; integer i, z;
   i := lookup(n);  /*查符号表*/
   if   i < 1
   then begin
        error( '过程', n ,'未定义', statno);
        errorrecovery( panic );   /*应急处理过程 */
        return ( i := 0, z:= 0);
       end
   else  return( i , z:= symtbl [i].dim);     /* z为形参数目*/
end;
```

LOD, (addr of symb)

LOD, (addr of cursor)

LOD, (addr of replacestr)

JSR, (addr of process_symb)

<retaddr>:….

```
procedure  chktype(t, i, m, z);
   string t; integer m, i, z;
   if z < 1
   then begin
      error( ‘实参数大于形参数’, symtbl [i].name, statno);
      return ( z);
    end
    m := m+1;          /* 实参计数 */
    if  t ≠symtbl [i+m].type
     then error(‘实参和形参类型不匹配’, symtbl [i+m].name, statno);
     z := z-1;          /* 减去已匹配的形参数 */
     return (z);        /* 剩下待匹配的形参数 */
end;
```

@chklenth 应检验z最后值为0。否则表示实参数目小于形参数目。

@genjsr 生成JSR指令。该指令转移地址为 symbtbl [i] .addr

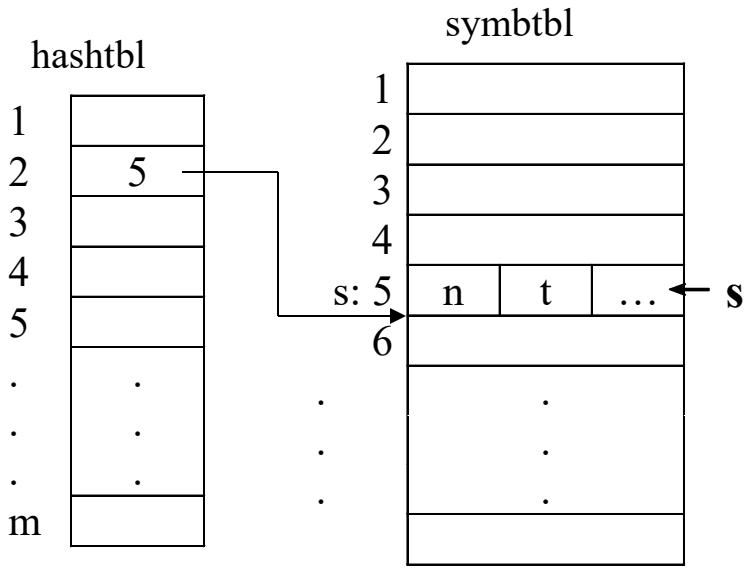
过程说明（定义）的ATG文法如下：

<proc defn>→<proc defn head> @initcnt ↑j
<parameters>↓j↑k @emitstores ↓k
<proc defn head>→procedure ↑t <id>↑n @tblinsert ↓t, n
<parameters>↓j↑k→@echo ↓j↑k | (<parm list>↓j↑k)
<parm list>↓j↑l→<type>↑t : <id>↑n @tblinsert ↓t, n
@upcnt ↓j↑k <parms>↓k↑l
<parms>↓j↑l→@echo ↓j↑l | , <type>↑t : <id>↑n @tblinsert ↓t, n
@upcnt ↓j↑k <parms>↓k↑l

@tblinsert 是把过程名和它的形参名填入符号表中：

```
procedure tblinsert( t, n );
 string t, n; integer hloc;
  if lookup ( n ) > 0
  then error( '名字定义重复', statno);
  else begin
    hloc := hashfctn(n); /*求散列函数值*/
    hashtbl[hloc] := s; /*s为符号表指针
                          （下标），  为全局量*/
    symbtbl [s].name:= n;
    symbtbl [s].type:= t;
    s := s+1;
end;
```

```
procedure emitstores(k);
  integer  k;
  emitl( 'ALC', k + x +… );
  emitl( 'STO', < ll, x+1 >);
                /*保存返回地址*/
  for i := k + x+1 down to x+2
                /*保存参数值*/
      emitl( 'STO', < ll , i > )
  end;
end;
```



注：实际ALC指令所分配的空间应在所有局部变量定义处理完以后，并考虑固定空间（前述‘x’）大小，反填回去。

ALC, 4 + x /* x为定长项空间 */

STO, <actrec loc1> /* 保存返回地址 */

STO, <actrec loc4> /* 保存replacestr */

STO, <actrec loc3> /* 保存cursor */

STO, <actrec loc2> /* 保存symb */

10.7.3 返回语句和过程体结束的处理

其语义动作有：

1) 若为函数过程，应将操作数栈（或运行栈）顶的函数结果值送入（存回）函数值结果单元

2) 生成无条件转移返回地址的指令（JMP　RA)

3) 产生删除运行栈中被调用过程活动记录的指令（只要根据DL—活动链，把abp退回去即可）

第十一章 代 码 优 化

- 概述
- 优化的例子
- 基本块的优化
- 循环优化

11.1 概述

代码优化（code optimization）

指编译程序为了生成高质量的目标程序而做的各种加工和处理。

目的：提高目标代码运行效率
- 时间效率（减少运行时间）
- 空间效率（减少内存容量）

原则：进行优化必须严格遵循“不能改变原有程序语义”原则。

分类：

从优化的层次，与机器是否有关，分为：

- 独立于机器的优化：即与目标机无关的优化，通常是在中间代码上进行的优化。
- 与机器有关的优化：充分利用系统资源，（指令系统，寄存器资源）。

从优化涉及的范围，又分为：

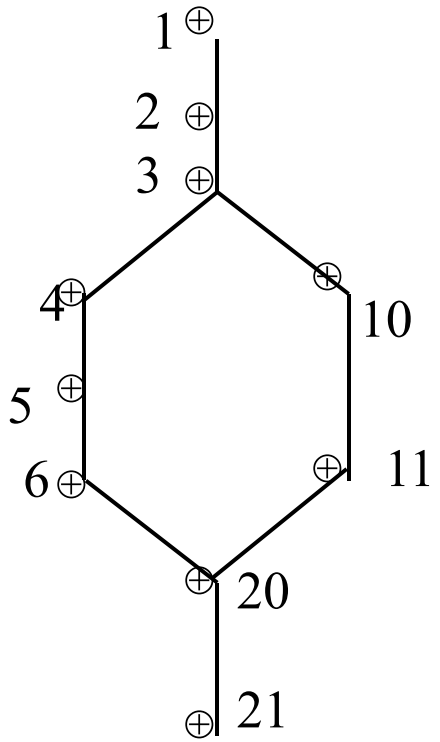
- 局部优化：是指在基本块内进行的优化。
- 循环优化：对循环语句所生成的中间代码序列上所进行的优化。
- 全局优化：顾名思义，跨越多个基本块的全局范围内的优化。因此它是指在非线性程序段上（包括多个基本块, GOTO, 循环）的优化。需要进行全局控制流和数据流分析，复杂。

[定义] 基本块（ basic block ）

满足以下三个条件的程序段，称为基本块：

- 只有一个入口和一个出口，且语句为顺序执行的程序段。
- 所有转移语句的目的语句都是基本块的第一条语句。
- 转移语句的下一条语句是基本块的第一条语句。
- 如果块中任一语句被执行，则该块内的所有语句也将被执行（无分支），且执行次数一样（无循环）。

例：书上的例子




```
1.   FACTOR = 2
2.   EXP 1 =…
3.   IF (  )GO TO  10
4.   BASE = 2.0
5.   FACTOR = FACTOR**2
6.   GO TO  20
10.  BASE =…
11.  FACTOR…
20.  Q =
21.  RETURN
```

基本块

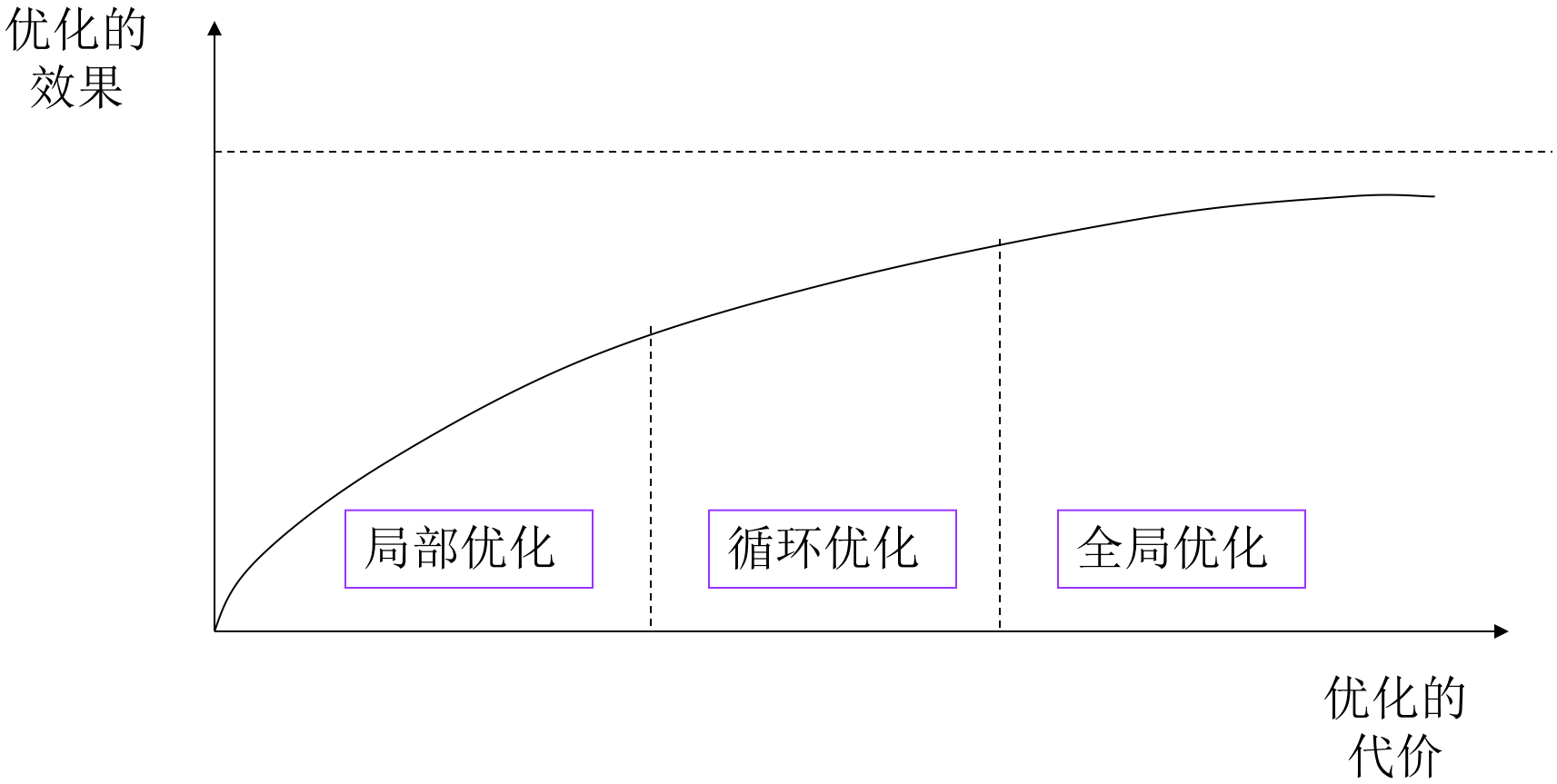
基本块

基本块

基本块

（先编号，后画图→决定基本块）

优化所花费的代价和优化产生的效果可用下图表示：



- 图的左部表示只要做些简单的处理，便能得到明显的优化效果。这相当于局部优化。
- 若要进一步提高优化效果，就要逐步付出更大的代价——循环优化。
- 全局优化进一步提高。

为什么要优化？

- 有的大型计算程序一运行就要花上几十分钟，甚至好几小时，这时为优化即使付出些代价也是值得的。
- 另外，程序中的循环往往要占用大量的计算时间。所以为减少循环执行时间所进行的优化对减少整个程序的运行时间有很大的意义。——尤其有实时要求的程序。如市场决策，供需及求益的平衡
- 至于（像学生作业之类的）简单小程序（占机器内存，运行速度均可接受），或在程序的调试阶段，花费许多代价去进行一遍又一遍的优化就毫无必要了。

11.2 优化的基本方法和例子

注：实际的优化应在中间代码或目标代码上进行。但为了便于说明，这里用源程序形式举例。

（1）利用代数性质（代数变换）

- 编译时完成常量表达式的计算，整数类型与实型的转换。

例： a := 5+6+x → a := 11+x

又如： 设x为实型，x := 3+1 可变换成x := 4.0

- 下标变量引用时，其地址计算的一部分工作可在编译时预先做好（运行时只需计算“可变部分”即可）。

- 运算强度削弱：用一种需要较少执行时间的运算代替另一种运算，以减少运行时的运算强度时、空开销）

如　　x**2 → x*x

3*x → x+x+x

8*x ， 4*x 等换成左移运算

x/2 ， x/16 等换成右移运算

x:=x+1 变为INC x指令

x/5 → x*0.2 等

利用机器硬件所提供的一些功能，如左移，右移操作，利用它们做乘法或除法，具有更高的代码效率。

（2） 复写(copy)传播

如x:=y这样的赋值语句称为复写语句。由于x和y值相同，所以当满足一定条件时，在该赋值语句下面出现的x可用y来代替。

例如：

```
x:=y ;              x:=y ;
u:=2*x ;     →      u:=2*y ;
v:=x+1 ;            v:=y+1  ;
```

这就是所谓的复写传播。(copy propagation)

若以后的语句中不再用到x时，则上面的x:=y可删去。

若上例中不是x := y而是x := 3。则复写传播变成了常量传播，即

```
x := y;                x := 3;          u := 6;        v := 4;
u := 2*x;      ⇨       u := 2*x;
v := x+1;              v := x+1;
```

又如　　t_1 := y/z;　　x := t_1;

若这里t_1为暂时（中间）变量，以后不再使用，则可变换为

x := y/z;

此外常量传播，引起常量计算，如：

pi = 3.14159　　r = pi/180.0

此时：pi = 3.14159　　r = 0.0174644　　（常量计算）

（3） 删除公共子表达式

具有相同值的子表达式在两个以上地方出现时，称它为公共子表达式(common subexpression)

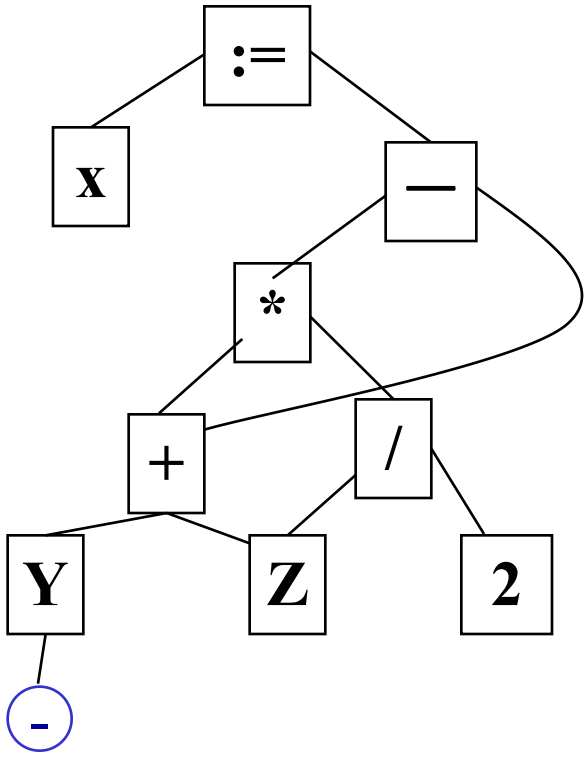
如赋值语句:

x := (-y + z) * z/2 - (-y + z)

其中: (-y+z)是公共子表达式。

可用DAG （ directed acyclic graph,有向无循环图）来表示具有公共子表达式的抽象语法树。

x := (-y+z)*z/2-(-y+z)



显然，对于公共子表达式只要计算1次即可

（4） 删除冗余代码

冗余代码就是毫无实际意义的代码，又称死代码(dead code)或无用代码(useless code)。

例如： x := x + 0 ; x := x * 1 ; 等

又例： FLAG := TRUE

IF FLAG THEN…

…

ELSE…

} FLAG永真

另外在程序中为了调试常有如下：

if debug then … 的语句。

但当debug为false时，then后面的语句便永远不会执行，这就是可删去的冗余代码。

（可用条件编译 #if DEBUG 编写程序，而源代码中还应留着）

（5） 循环优化

经验规则告诉我们：“程序运行时间的80%是由仅占源程序20%的部分执行的”。这20%的源程序就是循环部分，特别是多重循环的最内层的循环部分。因为减少循环部分的目标代码对提高整个程序的时间效率有很大作用。

```
for  i = 1      to        10
     for      j = 1       to        100
              x := x+0 ;
              y := 5+7+x ;
```

优化一条，少10*100次运算

除了对循环体进行优化，还有专用于循环的优化

a) 循环不变式的代码外提

不变表达式：

不随循环控制变量改变而改变的表达式或子表达式。

如： FOR I := E_1 STEP E_2 TO E_3 DO

BEGIN

S := 0.2*3.1416*R

P := 0.35*I

V := S*P

……

不变表达式可外提

不能外提

如　　while　…　do

x := …　（b*b - 4.0*a*c) …

若a,b,c的值在该循环中不改变时，则可将循环不变式移到循环之外，即变为：

t_1 := b*b - 4.0*a*c

while　…　do

x:= …(t_1) …

从而减少计算次数——也称为频度削弱

b) 循环展开

循环展开是一种优化技术。它将构成循环体的代码（不包括控制循环的测试和转移部分），重复产生许多次（这可在编译时确定），而不仅仅是一次，以空间换时间。

例 PL/1中的初始化循环

```
DO    I = 1    TO    30
      A[ I ] = 0.0
END
```

展开

```
      I := 1
L1:   IF  I > 30  THEN
      GOTO    L2
      A[ I ] = 0.0
      I = I+1
      GOTO    L1
L2:
```

代码5条语句，共执行5*30条语句

```
A[1] = 0.0
A[2] = 0.0
……
A[30] = 0.0
```

30条语句（指令）执行也是30条语句

- 循环一次执行5条语句才给一个变量赋初值。展开后，一条语句就能赋一个值，运行效率高。
- 优化在生成代码时进行，并不是修改源程序。
- 必须知道循环的终值，初值及步长。
- 但并不是所有展开都是合适的。如上例中循环展开后节省执行了转移和测试语句：2*30=60语句 (其实，还不止节省60条）。

∴增加29条省60条

但若循环体中不是一条而是40条语句，则展开后将有40*30条=1200，但省的仍是60条，就不算优化了。

∴判断准则：1. 主存资源丰富
处理机时间昂贵
2. 循环体语句越少越好

⇨ 循环展开有利（大型机）

实现步骤：

1. 识别循环结构，确定循环的初值，终值和步长。
2. 判断。以空间换时间是否合算来决定是否展开。
3. 展开。重复产生循环体所需的代码个数。

比较复杂：

∵在对空间与时间进行权衡时，还可以考虑一种折衷的办法，即部分展开循环。如上例展为：

```
DO    I = 1   TO    30    BY   3
      A[I] = 0.0
      A[I+1] = 0.0
      A[I+2] = 0.0
END;
```

空间只多二条，但省了20次测试时间（只循环10次）

c) 归纳变量的优化和条件判断的替换

归纳变量(induction varable): 在每一次执行循环迭代的过程中，若某变量的值固定增加（或减少）一个常量值，则称该变量为归纳变量(induction variable)。即若当前执行循环的第j次迭代。归纳变量的值应为c*j+c’, 这里c和c’都循环不变式。

例： for i := 1 to 10 do

a[i] := b[i] + c[i]

```
for  i:= 1     to     10     do
     a[i] := b[i]  +  c[i]
```

```
1)      i := 1
2)      labb:
3)      if  i > 10     goto   labe
4)      t1 :=  4*i
5)      t2 :=  b [ t1]
6)      t3 :=  4*i
7)      t4 :=  c [ t3]
8)      t5 := t2 +  t4
•       t6 :=  4*i
7)      a[t6] := t5
8)      i := i+1
9)      goto      labb
13) labe:
```

优化：

```
1)      u := 4
2)      labb:
3)      if u >40   goto    labe
4)      tb := b [u]
5)      tc := c [u]
6)      t  :=  tb + tc
7)      a [ u] := t
8)      u := u+4
9)      goto          labb
10)     labe :
```

中间变量t1， t3， t6 都是归纳变量　　t1 := 4*i，t3 := 4*i，t6 := 4*i

d) 其它循环优化方法

- 把多重嵌套的循环变成单层循环。
- 把n个相同形式的循环合成一个循环等。

对于循环优化的效果是很明显的。某FORTRAN 77 编译程序，在进行不同级别的优化后所得的目标代码指令数为：

优化级别	循环内的指令数（包括循环条件判断）
0（不优化）	21
1	16
2	6
3	5

（6） in_line 展开

把过程（或函数）调用改为in_line展开可节省许多处理过程（函数）调用所花费的开销。

如：procedure m(i , j : integer ; max : integer);

begin if i > j then max:=i else max:=j end;

若有过程调用 m (k , 0, max);

则内置展开后为：

if k > 0 then max := k else max := 0;

省去了函数调用时参数压栈，保存返回地址等指令。

这也仅仅限于简单的函数。

（7）其他， 如控制流方法

如

```
        BR      L
        …
L :
```

无条件转移

——为不可达代 码

又如：转移到转移指令的指令

```
        BR      L1
        …
L1:     BR      L2
```

优化

```
        BR      L2
        …
L1:     BR      L2
```

还有：

BR_{CC} L1 当条件CC成立，转到L1

BR L2

L1: ….

可改进为：

BR'_{CC} L2 当条件不能成立时，转到L2

(L1:) …

总结：

优化分为两大类

- 与机器无关的优化，即独立于机器的（中间）代码优化
  - 局部优化：（一个入口，一个出口，线性）——基本块
    - 方法：常数合并
      - 冗余子表达式的削除等
  - 循环优化：对循环语句所生成的中间代码序列上所进行的优化
    - 方法：循环展开
      - 频度削弱
      - 循环不变表达式的外提
      - 强度削弱
  - 全局优化：顾名思义，跨越多个基本块的全局范围内的优化。因此它是在非线性程序段上（包括多个基本块,GOTO循环）的优化。
- 与机器有关的优化，即目标代码上的优化（与具体机器有关）

第十二章 编译程序生成方法和工具

- 编译程序的书写语言
- 自编译性
- 自展
- 编译程序的移植
- 编译程序的自动生成

12.1 编译程序的书写语言

- 机器语言或汇编语言

  主要优点：编出来的程序效率高。

  主要缺点：编程效率低，可读性差，不便于修改和移植。

- 高级程序设计语言已基本取代汇编语言

  优点：编程效率高，可读性好，利于移植。

  缺点：编译程运行效率较低。

12.2 自编译性

自编译性：如果一个高级语言能用来书写自己的编译程序，则该语言具有自编译性，并称该语言为自编译语言。

两点说明：

1. 通常用自编译语言除可编写本语言的编译程序以外，也可用来编写别的语言的编译程序。

   ∴如果某台机器上已配备有某种自编译语言，则可利用这种语言为本台机器配置其它的高级语言。

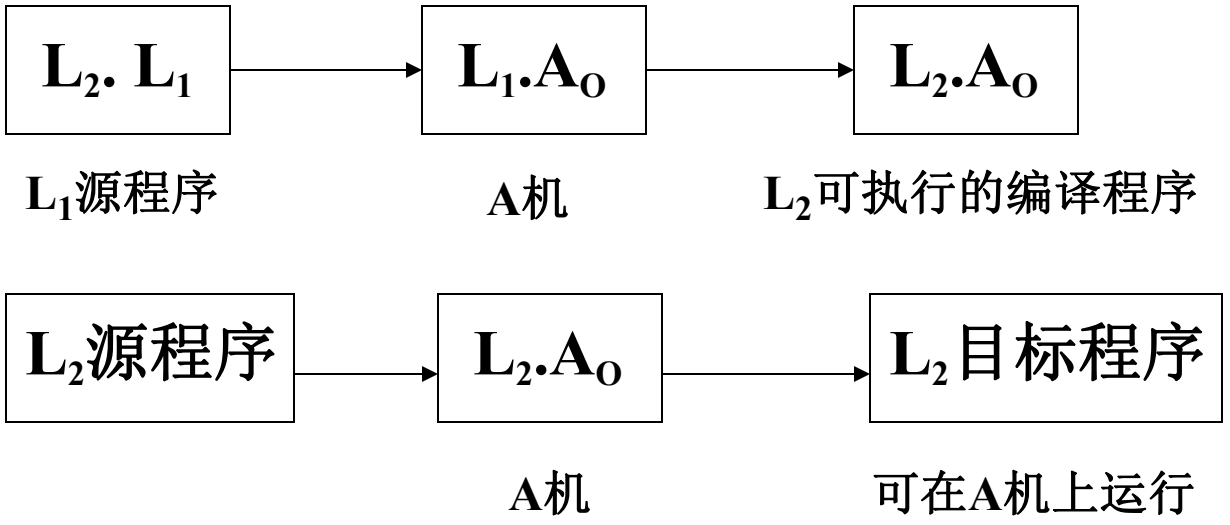
例：A机上有自编译语言L_1的编译程序

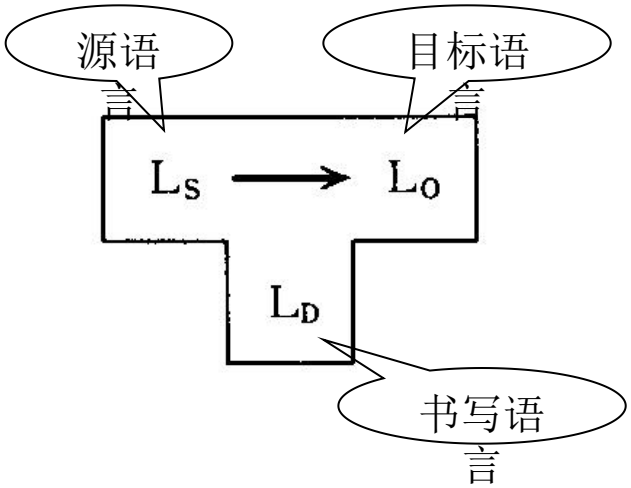
$L_1 . A_O$

L_1——语言L_1的编译程序

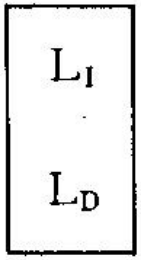
A_O——以A机的机器指令形式给出

利用语言L_1可为A机生成语言L_2的编译程序



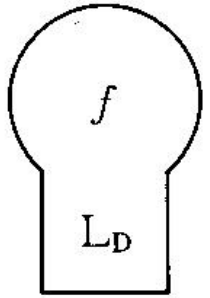


(a)编译程序

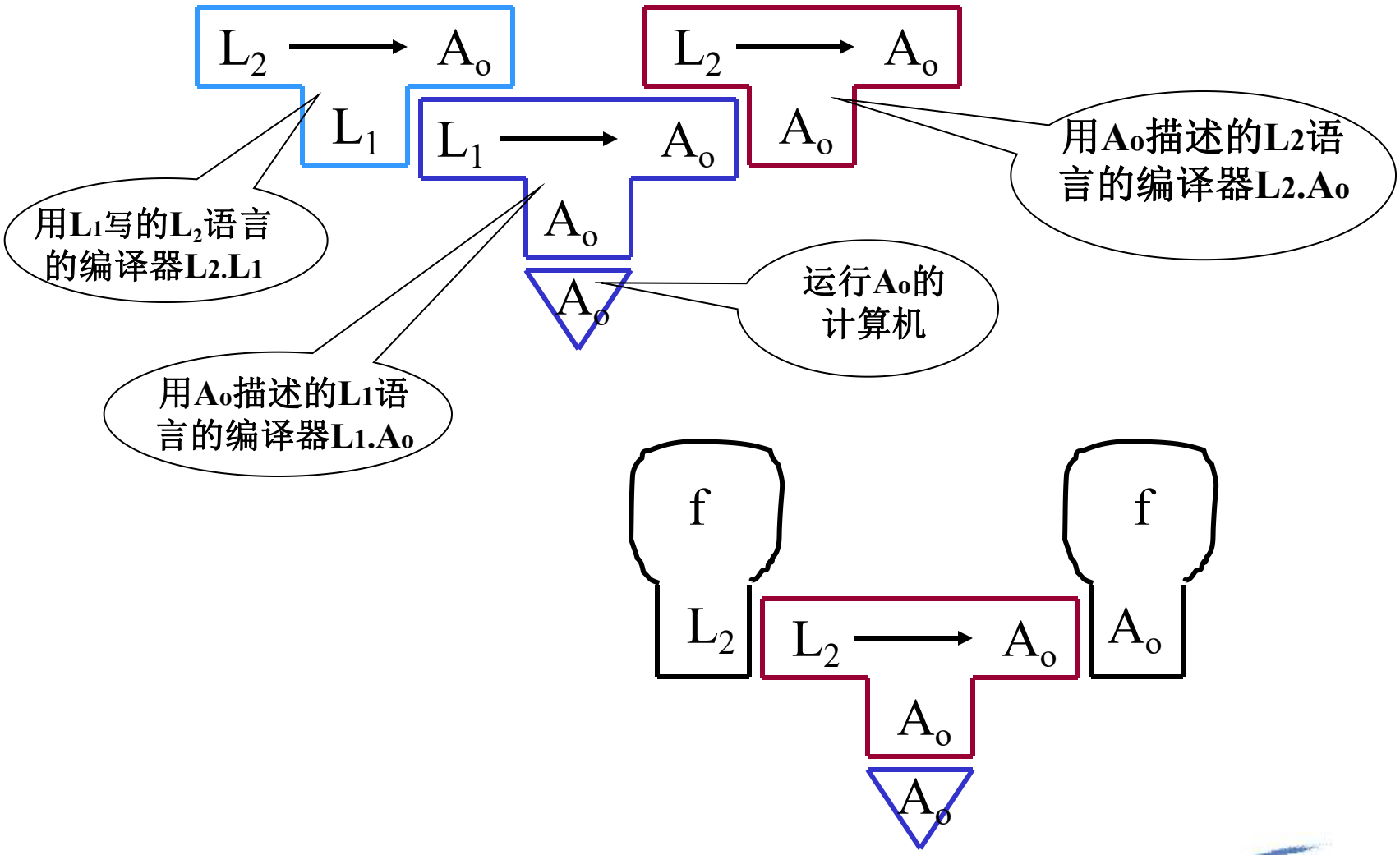


(b)解释程序

(c)计算机



(d)程序


L2 → Ao
L1
L1 → Ao
Ao
Ao
L2 → Ao
Ao
用L1写的L2语言的编译器L2.L1
用Ao描述的L2语言的编译器L2.Ao
运行Ao的计算机
用Ao描述的L1语言的编译器L1.Ao
f
L2
f
Ao
L2 → Ao
Ao
Ao

2.自编译性不是绝对的，只是强弱不同

数据类型丰富的语言
控制结构丰富的语言
} 自编译性强

数据类型：除一般的外还有字符串类型，数组，结构，枚举，指针等类型。

控制结构：应适于进行多分支的程序设计，如有CASE语句等

FORTRAN , ALGOL——自编译性差

PASCAL , C , ADA，C＋＋，JAVA——自编译性强

实践示例：用PASCAL语言编写一个简单的编译程序，就是利用PASCAL的自编译性。

12.3 自展

利用高级语言的自编译性，还可以通过自展方式生成语言的编译程序。

设L为自编译语言，自展生成

L. Ao（A机目标形式的语言L的编译器，可在A机上运行）

步骤：1.首先，将语言划分为N个部分：

$$L = L_1 + L_2 + \ldots + L_n$$

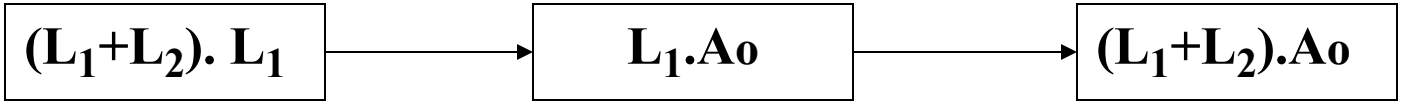
L_1 ——核心部分

L_2～ L_n——扩充部分

2.先用A机上的汇编编写L_1的编译程序，L_1.Aa

L_1.Aa→Assember→L_1.Ao

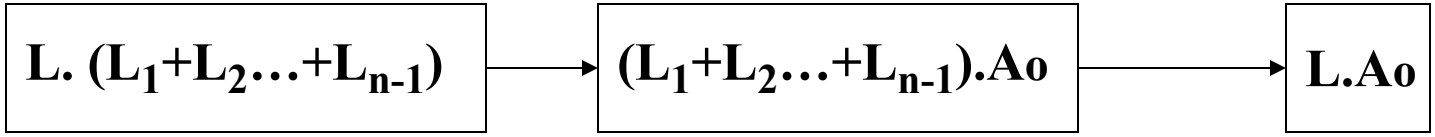
3.用L_1编写L_1+L_2的编译程序

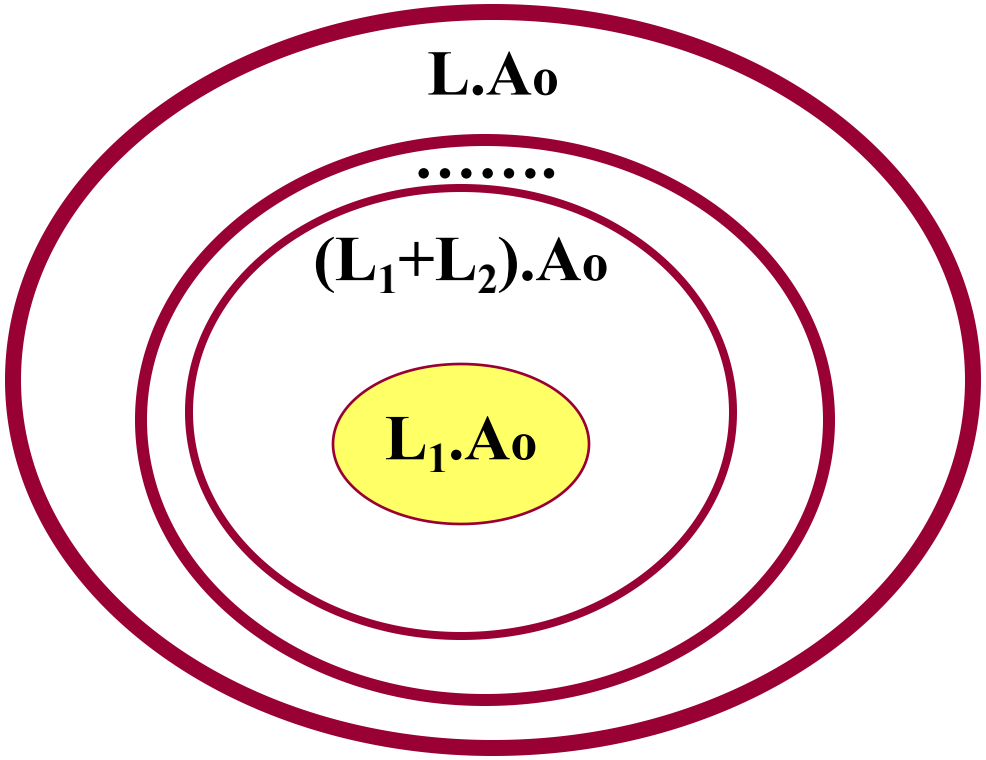


4.用(L_1+L_2)编写$L_1+L_2+L_3$的编译程序

…

5.



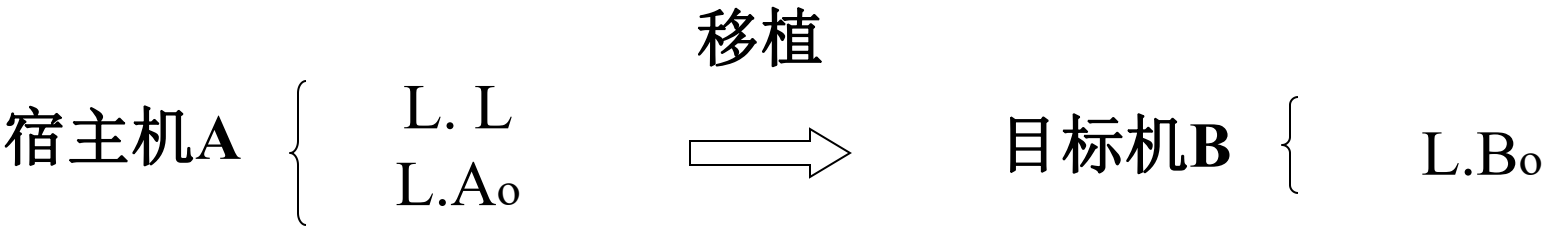


滚雪球式

用自展方式进行编译，可提高生产率。因核心语言小，可用汇编实现。其余部分高级语言编写。比全用低级语言效率高。

12.4 编译程序的移植

移植：将某台机上的成熟软件移植到另一台机器上，也就是将宿主机上的软件移植到目标机上。如果使用具有自编译性的高级语言来书写程序，则移植是方便的。



通过移植，在B机上可得到语言L的编译程序，具B机目标形式，可在B机上运行。

移植步骤：

1. 将L. L分为两部分：

一部分与机器无关 F.L　一部分与机器有关　A.L

∴ L. L = F.L+ A.L

2. 根据目标机用语言L改写与具体机器有关的部分：

A.L $\xrightarrow{L}$ B.L

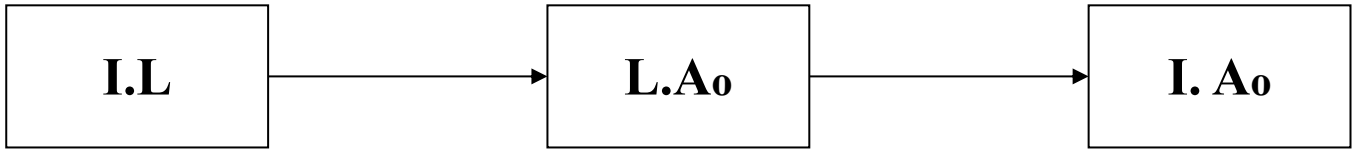
产生A机代码　　产生B机代码

∴交叉编译器：　　I.L= F.L+ B.L

用A机上的L语言所写的能生成B机目标代码的语言L的编译程序。

3. 第一次编译

将I.L在宿主机A上用L的编译程序进行编译，生成能在宿主机A上运行的语言L的交叉编译器，它能生成目标机B的代码。

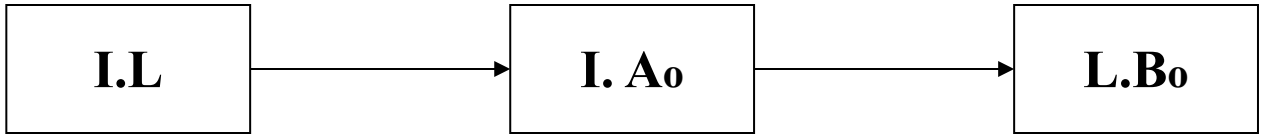


用L所写的生成目标机B代码的L语言交叉编译器源程序

宿主机A的L编译程序

语言L的交叉编译器，能在宿主机A上运行，生成目标机B的代码

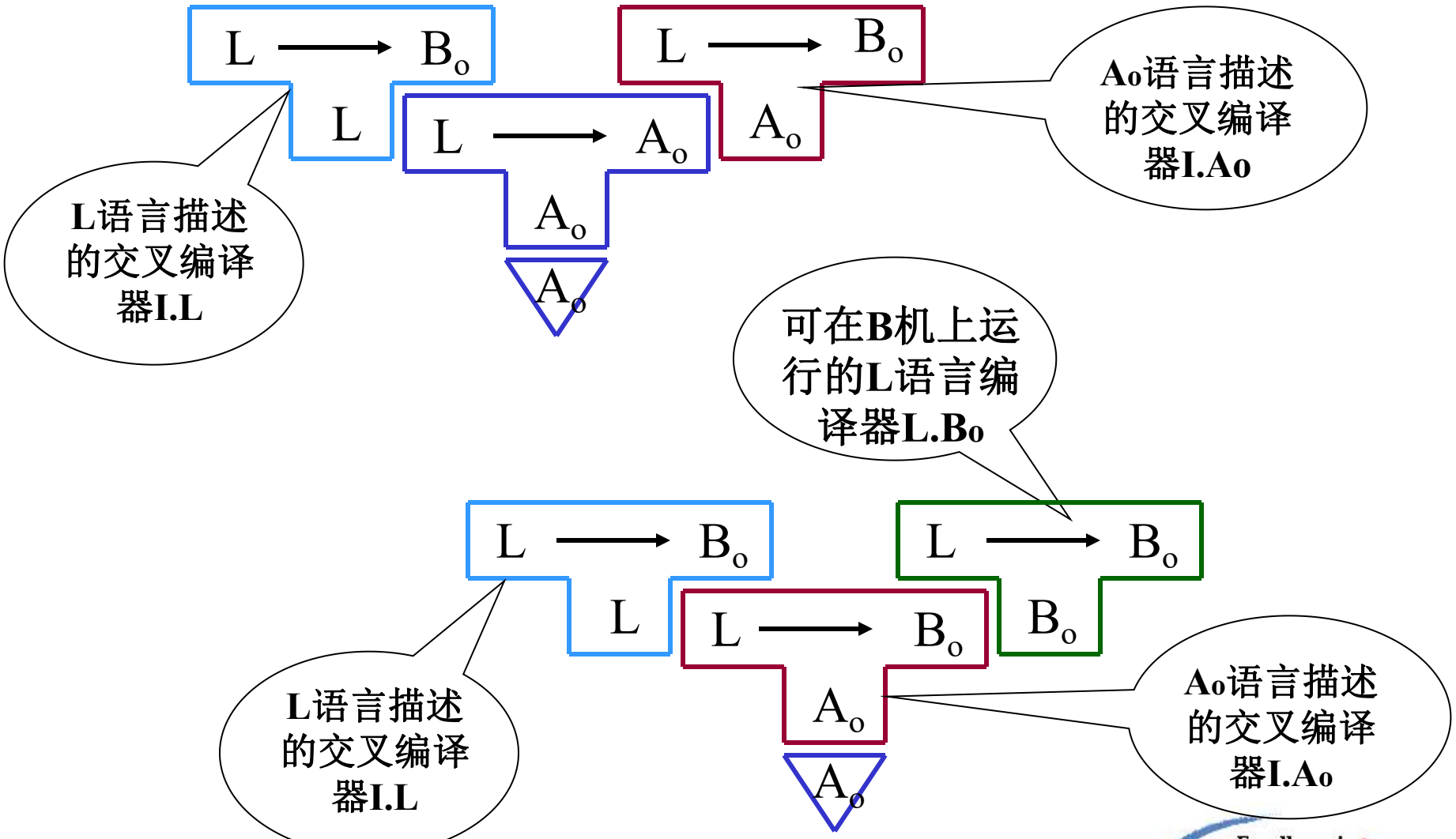
4. 第二次编译（交叉编译）



可在目标机B上运行并生成目标机B代码的L编译程序

A机

交叉编译程序：在宿主机A上运行，但所生成的目标只能在目标机B（另一台机器）上运行。


L → Bo
L
L → Ao
Ao
Ao
L → Bo
Ao
L语言描述的交叉编译器I.L
Ao语言描述的交叉编译器I.Ao
可在B机上运行的L语言编译器L.Bo
L → Bo
L
L → Bo
Bo
Ao
L → Bo
Bo
L语言描述的交叉编译器I.L
Ao语言描述的交叉编译器I.Ao

可以设想，只要在某台机器上为某目标机配置一个L语言的交叉编译程序，就能将宿主机上的L语言所写的所有软件移植到其他目标机上。

采用软件移植的办法来开发软件，可提高软件生产率，并提高软件的可靠性。由于上述优点，所以软件的可移植性是软件开发所追求的目标之一。

目前有许多编译程序都考虑到可移植性的要求。

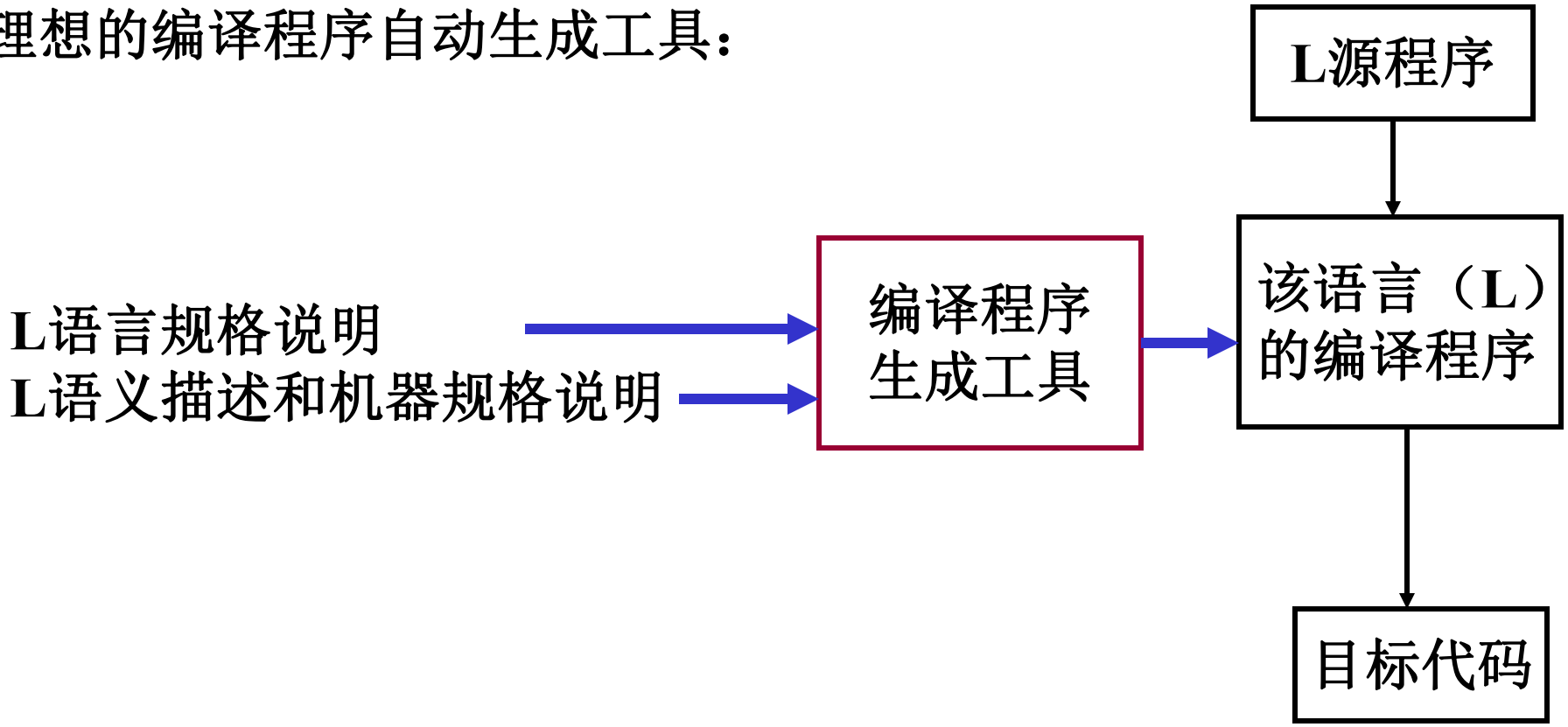
例如有：可移植的PASCAL编译程序。

P.J.Brown ， Software Portablility.

朱关铭等译，1982.12

12.5 编译程序的自动生成

理想的编译程序自动生成工具：

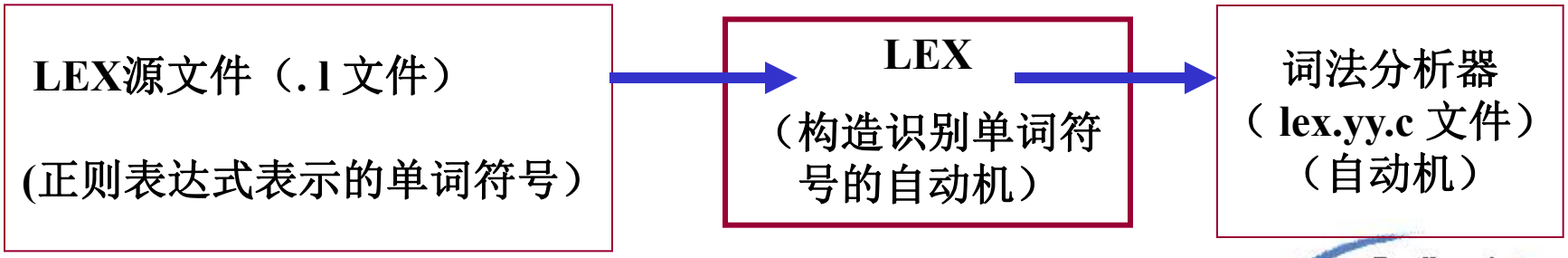


目前还没有一个系统能自动生成整个编译系统。

早期的工作集中在分析部分，即针对语法规则的形式化描述。对编译程序后端，即与目标机有关的代码生成与代码优化部分，由于对语义和目标机进行形式化描述方面所存在的困难，最近有所突破，但未见到流行的产品。（样机——未形成真正产品）

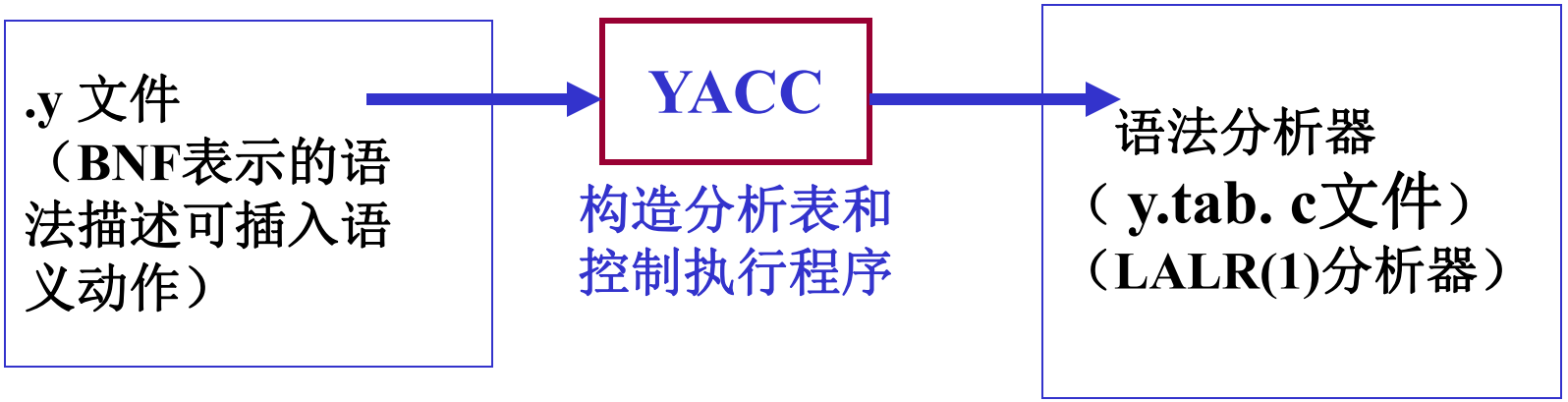
- 有词法分析器的自动生成器和语法分析器的自动生成器。

  词法分析器生成器（在第三章已作介绍） **LEX:**



语法分析器生成器：

YACC (YET ANOTHER COMPILER - COMPILER)



Bison: 美国GNU开发的语法分析器生成器)和YACC一样都在UNIX系统下运行。（已有PC版)

用yacc建立翻译程序

yacc源程序：translate.y

1. 键入命令：

yacc translate.y

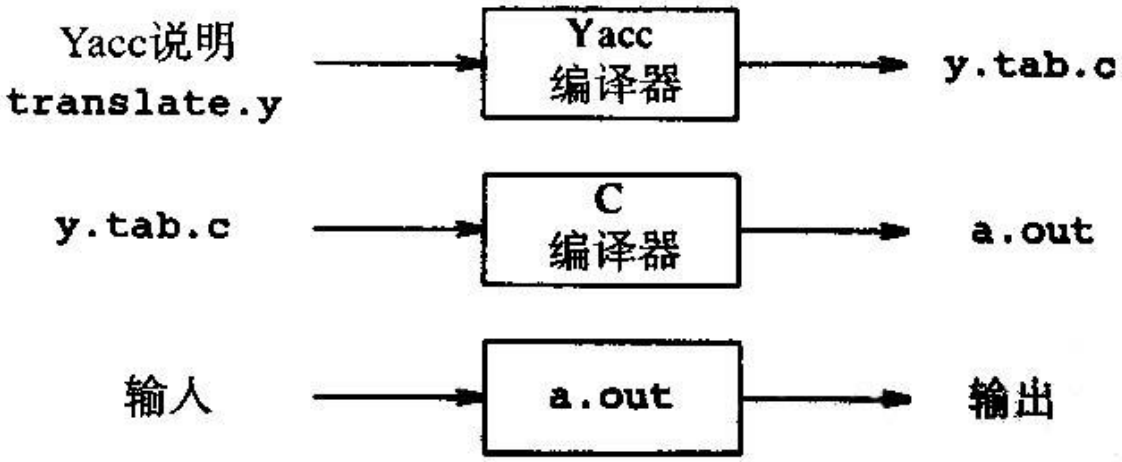
2. 生成进行LALR分析的翻译程序： y.tab.c

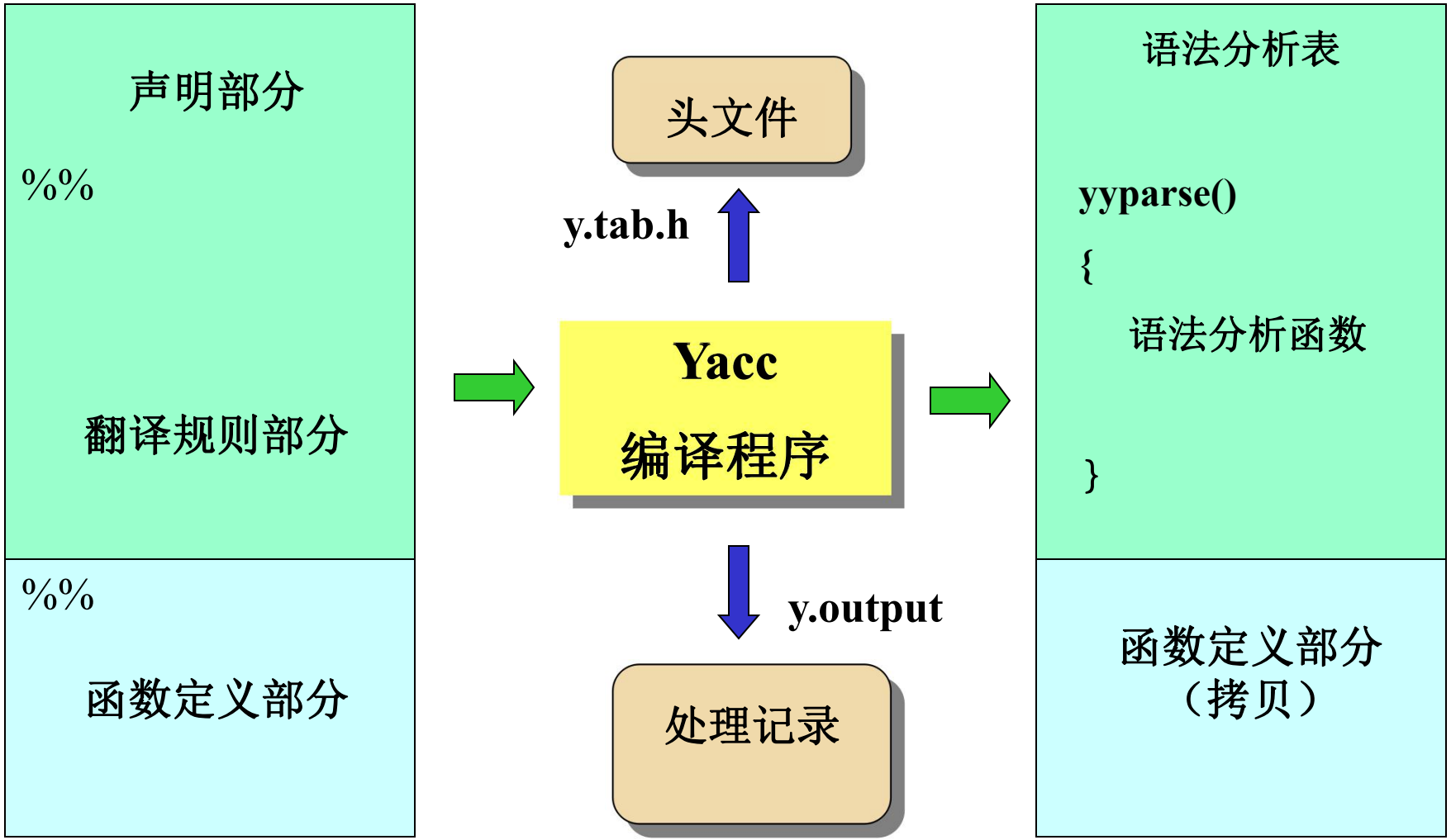
3. 对生成的分析器进行编译：

cc y.tab.c -ly

(ly为使用LR分析器的库）

生成可执行的 翻译程序 a.out




声明部分
%%
翻译规则部分
%%
函数定义部分
yacc源程序
(*. y)
头文件
y.tab.h
Yacc
编译程序
y.output
处理记录
语法分析表
yyparse()
{
语法分析函数
}
函数定义部分
（拷贝）
y.tab.c

```
1.   /*  表达式计算  */
2.   % token  NUM
3.   %%
4.   line   : expr  '\n'          { printf ('\n', $1);}
5.          ;
6.   expr   : expr '+' term       { $$ = $1 + $3; }
7.          | expr '-' term       { $$ = $1 - $3; }
8.          | term                /* $$ = $1 */
9.          ;
10.  term : term  '*' factor      { $$ = $1 * $3; }
11.         | term '/' factor     { $$ = $1 / $3; }
12.         | factor              /* $$ = $1 */
13.         ;
14.  factor:  '(' espr ')'        { $$ = $2; }
15.         | NUM                 /* $$ = $1 */
16.         ;
```

```
%%
#include   <ctype.h>
yylex( )
{
        int  c;
        while (( c = getchar( )) == ' ' );
        if ( isdigit ( c ) )  {
            yylval = c – '0';
            while ( isdigit( c = getchar ( ) ) )
                yylval = yylval*10 + ( c-'0' );
            ungetc ( c, stdin );
            retun  NUM;      }
         else   return c;
}
```