编译原理

- **第一章. 绪论**
  - 前端：与源程序有关的部分。
    - 包括词法语法语义中间代码生成与优化。 —— 分析部分
    - 特点：与源语言有关。
  - 后端：与目标机有关的部分。
    - 目标程序生成 —— 综合部分
    - 特点：与目标机有关。
  - 遍：对遍程序从头到尾扫描一次，并做有关的加工处理，生成新的源程序中间形式或者目标程序，通常称之为一遍。
  - 七个组成部分
    - 词法分析，语法分析，语义分析生成中间代码，程序优化，生成目标程序，符号表管理，出错处理
- **第二章. 文法和语言**
  - 文法
    - 定义： 文法G=(V_N,V_T,P,Z)
    - P：产生式或者规则的集合
    - Z：开始符号
    - V：终结符和非终结符的集合

- ~~V*：V的闭包，0到多个终结符或非终结符构成的串的集合。~~
- V+：V的正则闭包，一个或者多个终结符或非终结符构成的串的集合，等于V*-{ε}。

- 语言
  - 形式化定义

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

- BNF范式
- 推导
  - 1

$Z \overset{*}{\underset{G}{\Rightarrow}} w$

定义4：文法G，$v, w \in V^+$

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

  - 2

$\overset{+}{\Rightarrow}$

$\overset{+}{\underset{G}{\Rightarrow}}$

  - 3

$\Rightarrow$

  - 4

$\overset{+}{\Rightarrow}$

- 归约
- 文法和语言的关系：多对一

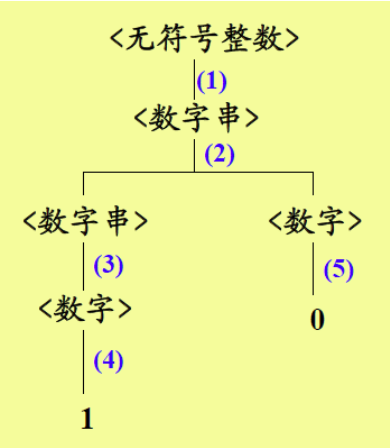
- 规范推导
  - 等于最右推导
  - 最右推导：字符串中有两个以上的V_n的时候，先推右边的
  - 最左推导：字符串中有两个以上的V_n的时候，先推左边的
- 规范归约

> 对句型中最左简单短语（句柄）进行的规约称为规范规约。

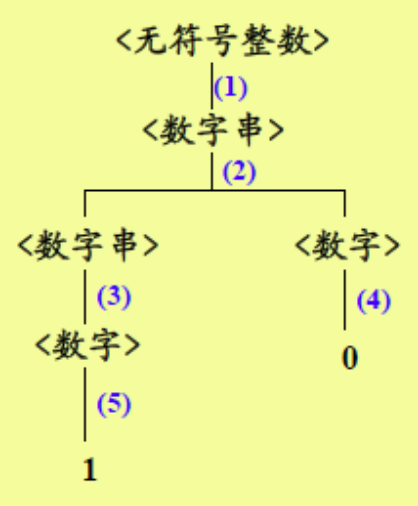
  - 由语法树建立推导：自下而上的修剪子树的末端节点，直到剪掉整棵树，每剪一次对应一次规约。
  - 从句型开始，自左向右逐步进行规约，建立推导序列。
- 规范句型

> 通过规范推导或规范规约所得到的句型称为规范句型。

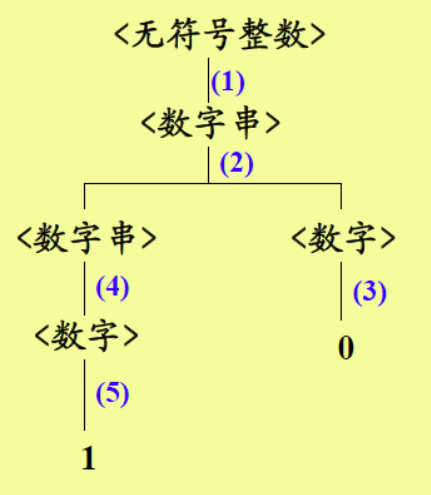
- 递归文法
  - 左递归和右递归
  - 左递归：不能自顶向下
  - 优点：可以用有穷条规则，定义无穷语言
- 语法树
  - 语法树生成的形状完全相同
  - 最左推导产生



- 一般推导（?



- 最右推导



- 子树

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。

- 某子树末端节点按照从左向右的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

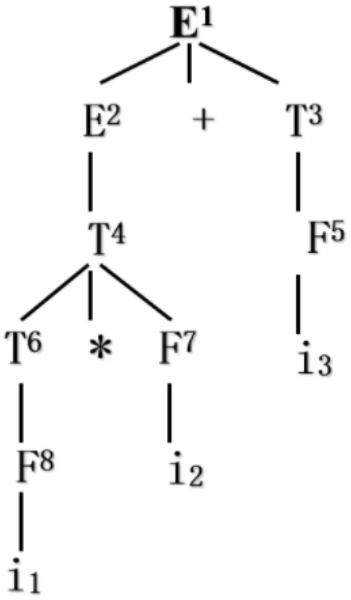
树根的短语。

- **短语与简单短语与句柄**

利用语法树寻找短语、句柄课堂练习

例 文法G[E]： E→E+T|T T→T*F|F F→(E)|i

句型 η=i_1*i_2+i_3 的语法树 BEGIN



8个内部节点—— 8棵子树

句型η有8个短语：

i_1*i_2+i_3是句型η相对于E^1的短语

i_1*i_2是句型η相对于E^2，T^4的短语

i_1是句型η相对于T^6，F^8的短语

i_2是句型η相对于F^7的短语

i_3是句型η相对于T^3, F^5的短语

直接短语 3个： i_1， i_2， i_3

句柄： i_1

- 定义

定义8. 给定文法G[Z], w::=xuy∈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；

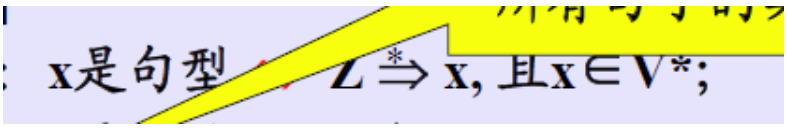
若 $Z \overset{*}{\Longrightarrow} xUy$, 且$U \Longrightarrow u$, 则u是句型w相对于U的简单短语。

其中U ∈V_n， u ∈V^+， x, y ∈V^*

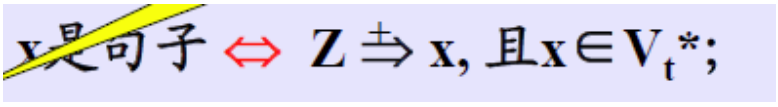
- 短语是前面句型中某个非终结符所能推出的符号串
- 任一句型最左简单短语称为该句型的句柄
- 给定句型找句柄的步骤
  - 短语→简单短语→句柄
- 注意

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

- 句型
  - 对于文法G[Z]，x是句型，等价于



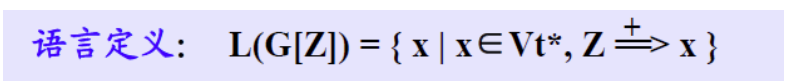
- 句子
  - 对于文法G[Z]，x是句子



- 二义性
  - 定义

定义14.1 若对于一个文法的某一句子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。

  - [难点]二义性文法的算法/判断
    - 随便找一个句子，对它的推导存在两棵不同的语法树
    - 存在两个不同的规范推导（最右推导，最左推导也可可）过程
    - 对该句子的某个规范句型存在两个不同的句柄
  - 文法的二义性是不可判定的
- 文法的乔姆斯基分类
  - 语言定义



  - 文法定义

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∈Vn

- ## 文法分类

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施加不同的限制。

- 0型文法
  - 定义

    0型：　P：　u::=v
    其中u∈V⁺，v∈V*
  - 被称为短语结构文法，左部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，可以被图灵机接受。
- 1型文法
  - 定义

    P：　xUy::= xuy
    其中 U∈Vn，
    x、y、u∈V*
  - 称为上下文敏感/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，可以由一种线性界限自动机接受。
- 2型文法
  - 定义

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不用考虑上下文。
- 与BNF表示等价。
- 2型语言：L2，可以由下推自动机接受。

- 3型文法
  - 定义

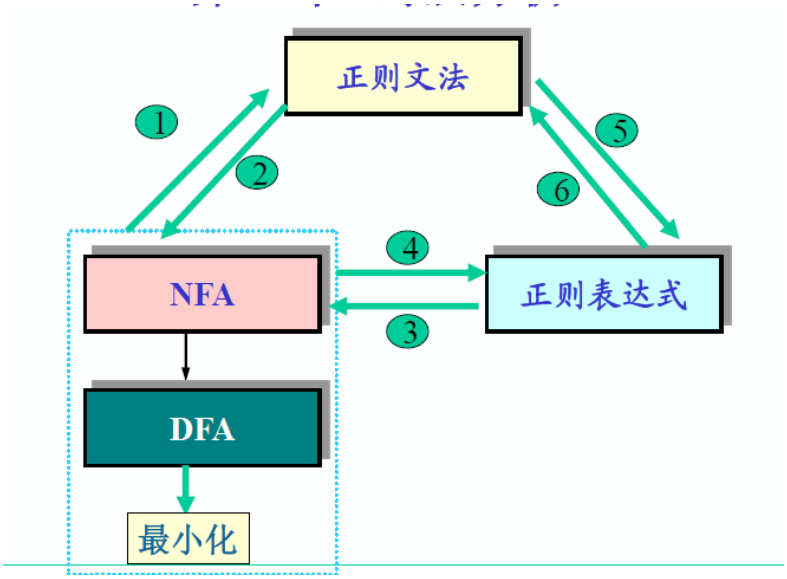
（左线性）	（右线性）
P: U::=T 或 U::=wT 其中 U、w∈Vn T∈Vt	P: U::=T 或 U::=Tw 其中 U、w∈Vn T∈Vt

  - 称为正则文法，对2加以限制。
  - 3型语言：L3，又称正则语言，正则集合，可以由有穷自动机接受。
- 包含关系

根据上述讨论，L0⊃L1⊃L2⊃L3

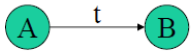
0型文法可以产生L0、L1、L2、L3，但2型文法只能产生L2，不能产生L1。

- **第三章. 词法分析**



- 转换算法
  - 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

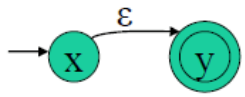
3. 有穷自动机的初态对应于文法的开始符号，
   有穷自动机的字母表为文法的终结符号集。

  - 正则式=>有穷自动机

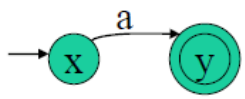
语法制导方法

1.(a)对于正则式φ,所构造NFA:

(b)对于正则式ε,所构造NFA:

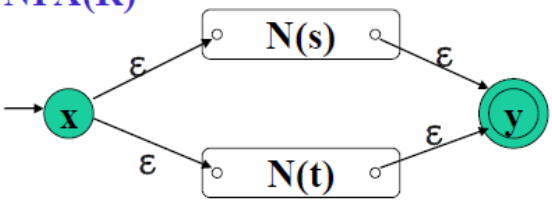


(c)对于正则式a，a∈Σ,则 NFA:

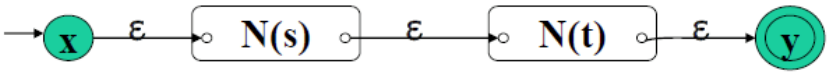


2.若s,t为Σ上的正则式,相应的NFA分别为N(s)和N(t);

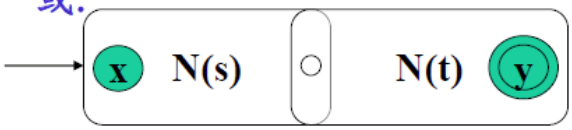
(a)对于正则式R = s | t， NFA(R)



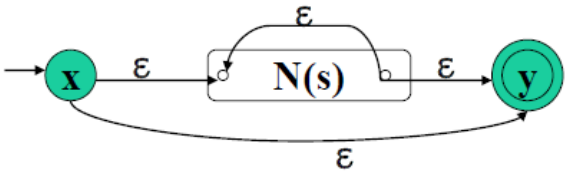
(b)对正则式R = s t， NFA(R)



或:



(c)对于正则式R = s*， NFA(R)



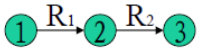
(d)对R = (s)，与R = S的NFA一样。

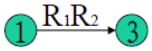
- 有穷自动机=>正则式R

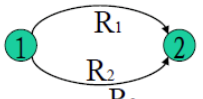
算法:

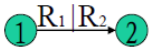
1) 在M上加两个结点x, y。从x用ε弧连接到M的所有初态结点，从M的所有终态结点用ε弧连接到y，形成与M等价的M'。M'只有一个初态x和一个终态y。

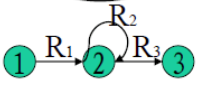
2) 逐步消去M'中的所有结点，直至剩下x和y为止。在消除结点的过程中，逐步用正则式来标记箭弧。其消结规则如下:

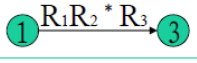
1.对于 

代之为 

2.对于 

代之为 

3.对于 

代之为 

- 正则文法=>正则式

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

- 正则式=>正则文法

（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→x y的产生式，重写为A→x B和B→y，其中B为新的非终结符，B∈V_N

同样，对于 A→x* y ⇒ A→x A A→y

A→x | y ⇒ A→x A→y

例：将S = 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

- 左线性文法状态图的画法

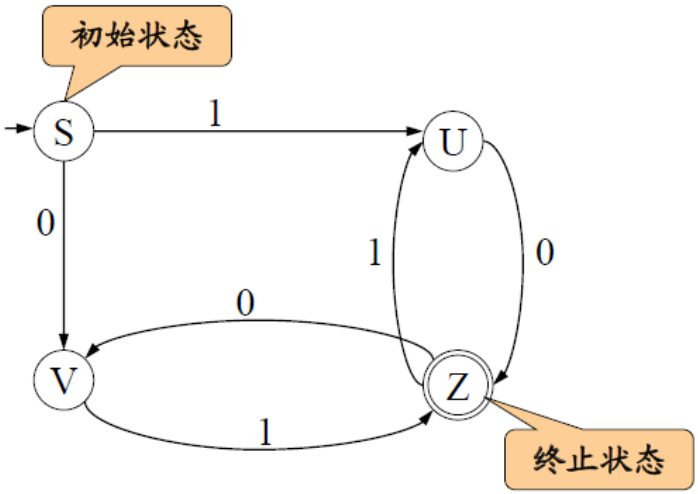
例如：正则文法

Z::= U0 | V1

U ::=Z1 | 1

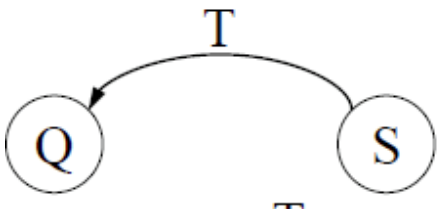
V ::=Z0 | 0

其状态图为:

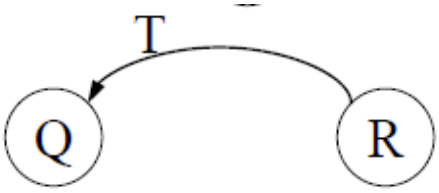


- 令G的每个V_N都是一个状态

- 设一个开始状态S
- 若Q::=T，Q∈V_n，T∈V_t，则



- 若Q::=RT，Q、R∈V_n，T∈V_t，则



- 按自动机方法，可以加上开始状态和终止状态标志

- 单词的划分
  - 标识符，关键字，界符（可分为单双分界符），常量
- 正则文法、正则表达式、NFA的相互转换
-
- [考试必考]NFA确定化，NFA转换为DFA的方法：子集法
  - DFA的定义

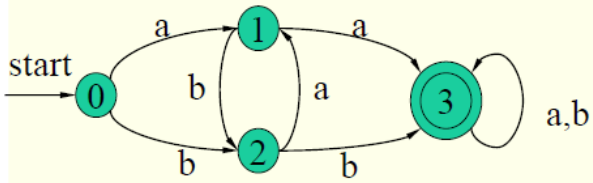
一个确定的有穷自动机（DFA）M是一个五元式：

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

一个DFA也可以用一个状态转换图表示：

状态 \ 输入字符	a	b
0	1	2
1	3	2
2	1	3
3	3	3

DFA的状态图表示：



- NFA的状态转换图

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

- NFA的确定化
  - 集合I的ε闭包

    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。

  - 定义2 I_a

定义2：令 I 是NFA M'的状态集的一个子集，a∈Σ

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

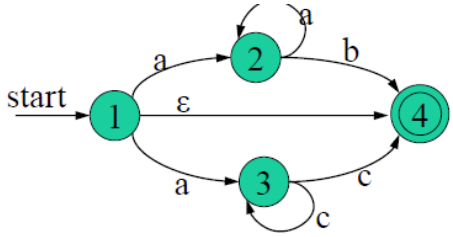
其中 $J = \bigcup_{s \in I} \delta(s, a)$

——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。

我们同样可以通过一例子来说明上述定义，仍采用前面给定状态图为例。

- 【算法】NFA求法
- 1. 求出



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}

- 2.

将求得的状态转换矩阵重新编号

DFA M状态转换矩阵：

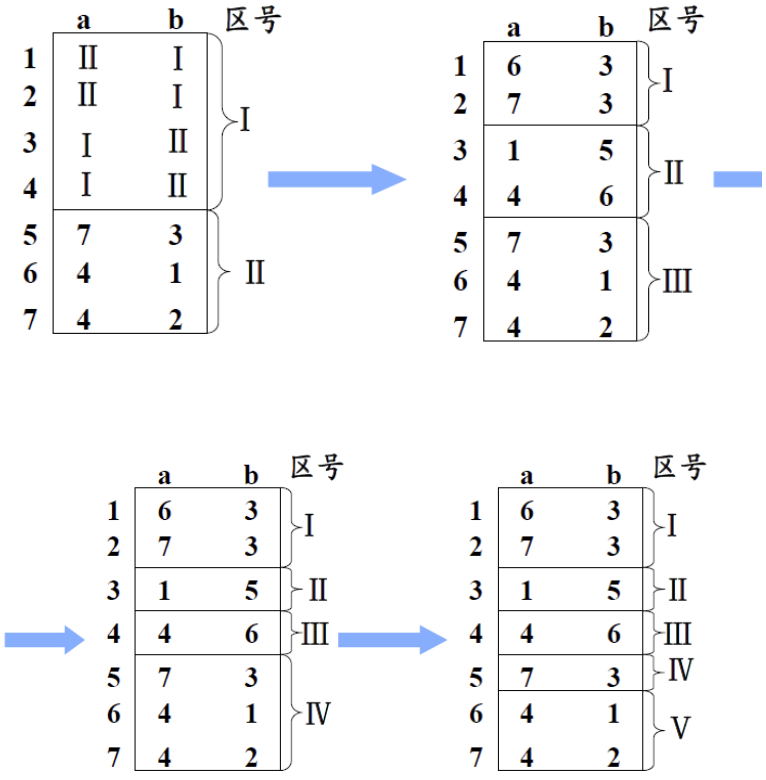
- 3.

原初始状态 1 的 ε-closure 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。

- [考试必考]DFA的最小化：分割法
  - 【算法】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。

(一) 区分终态与非终态



- **第四章. 语法分析**
  - 自顶向下
    - 基本思想

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

    - 一般过程
      - 根据该语法成分的文法，为符号串S构造一棵语法树
      - 分析过程是一个试探过程，需要进行回溯

- 这里的最左推导的效率低
- 问题
  - 无法处理左递归
  - 回溯
- 消除直接左递归
  - 方法1.用扩充的BNF表示来改写文法

(1) E∷= E + T | T ⇒ E∷= T { + T }
(2) T∷= T * F | T / F | F ⇒ T∷= F { * F | / F }

    - 规则
      - 提公因子，把epsilon放最后
      - 规则2

若有文法规则：U∷= x | y |...| z | U v

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打头，其后跟随零个或多个 v 组成。

U ⇒ Uv ⇒ Uvv ⇒ Uvvv ⇒ .....
∴ 可以改写为U∷= (x | y |...| z) { v }

  - 方法2.改为右递归

若： P∷= P α | β
则可改写为：
P ∷= β P'
P' ∷= α P' | ε

- 避免回溯的要求

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

- 方法
  - 递归下降子程序
    - 消除左递归
    - 求FIRST和FOLLOW集的算法
      - 【算法】FIRST，一般从下往上

设$\alpha = X_1 X_2 \dots X_n$，　　$X_i \in V_n \cup V_t$

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 $X_i ::= a \dots | \varepsilon$， $a \in V_t$

则 FIRST(X_i) = { a , ε }

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

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

且若$\varepsilon \in$ FIRST(y_2) 则将FIRST(y_3) – {ε}加入FIRST(X_i)

.....

$\varepsilon \in$ FIRST(y_{k-1}) 则将FIRST(y_k) – {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

若$\varepsilon \in$FIRST(y_1) ~ FIRST(y_k) 则将ε加入FIRST(X_i)

得到FIRST(X_i)，即可求出FIRST(α)。

- V_T的FIRST就是它本身，V_N的FIRST就是开头的V_T和epsilon
- 分两种情况，若FIRST(Y1)中无epsilon则将其加入X的FIRST，停止。

- 若有则将FIRST(Y1)-epsilon和FIRST(Y2)-{epsilon}加入X的FIRST，依次进行下去，不满足条件则终止。
- 若epsilon属于FIRST(Y_1)~FIRST(Y_K)则将其加入，最后得到FIRST集。_

- 【算法】FOLLOW，一般从上往下

设S, A, B∈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::=αBβ(β≠ε)，则把FIRST(β)-{ε}加入FOLLOW(B);

(3) 若A::=αB 或A::=αBβ，且β⇒ε则把FOLLOW(A)加入FOLLOW(B) 中去。

注意：FOLLOW集合中不能有ε!!!

  - U为识别符号，则将#加入FOLLOW(U)
  - A→...Ua...，则将a加入FOLLOW(U)
  - A→...UP...，则将FIRST(P)加入FOLLOW(U)
  - A→...UP，同上，但P含有epsilon的时候将FOLLOW(A)加入FOLLOW(U)
  - →...U，将FOLLOW(A)加入FOLLOW(U)
- 求有效项目和有效项目集族的算法
- 构造递归下降分析表和递归下降子程序框图的算法

- LL分析
- LL(1)文法

- 分析表中不含多重定义入口/表中无两条以上规则，则称它是一个LL(1)文法。
- 【算法】LL(1)文法的充要条件：

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

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

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

- 【算法】LL(1)分析器的构造
  - 1、遍历所有产生式A→α。
  - 2、遍历a ∈ First(α)，将 A --> α 填入M[A,a]
  - 3、如果ε ∈ First(α)，对于任意a ∈ Follow(A)，将 A --> α 填入M[A,a]
    - 如果ε ∈ First(α) 且 # ∈Follow(A)， 则将 A --> α 填入M[A,#]
  - 4、将所有没有定义的M[A,b] 标上出错标志 （留空也可以）

- 自底向上
  - 基本思想

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

  - 当采用自左向右扫描分析输入串的时候，算法为：

- 从输入符号串开始，通过反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规约，若能规约为文法的识别符号，则表示分析成功。

- 问题
  - 如何找句柄
  - 句柄α已找到，但是如果既存在A→α又存在B→α的话要怎么办?

- 方法
  - 算符优先分析法
  - LR分析
    - LR(0)
    - [重点]SLR(1)
    - LR(1)
    - [大概不考]LALR(1)

- 算符优先分析法
  - 运算法则
    - 与四则运算法则相同
  - 优先关系的定义

1) 优先关系的定义:
设 a，b 为可能相邻的终结符
定义: $a \doteq b$ — a的优先级等于b
$a \lessdot b$ — a的优先级小于b
$a \gtrdot b$ — a的优先级大于b

  - 算符优先函数相关

- 优点：节省内存空间，n个终结符关系矩阵为n^2，优先函数为2n；易于比较，算法上容易实现。
- 缺点：可能掩盖错误，将ii识别成一个合法的句子。

- 可以分析二义性文法所产生的语言。按规范分析，其句柄不唯一。

- 算符优先文法
  - 定义：无形如U::=...VW...，VW∈V_n的规则，称之为OG文法即算符文法。
  - 优先关系的定义（？

若G是一个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⇒ b...或W⇒ Vb... 。
3) a>b　iff 文法中有形如 U::=...Wb...的规则，其中W⇒...a或W⇒...aV 。

  - 【算法】求FIRSTVT和LASTVT
    - FIRSTVT
      - 若有规则 U::= b... 或 U::= V b...，则 b∈FIRSTVT(U)；
      - 若有规则 U::= V... 且 b∈FIRSTVT(V), 则 b∈FIRSTVT(U)。

- LASTVT
  - 若有规则 U::=…a 或 U::=…aV，则 a∈LASTVT(U)；
  - 若有规则 U::=…V，且 a∈LASTVT(V)，则 a∈LASTVT(U)。
- 【算法】构造算符优先矩阵（？
  - 1. 找出形如aQb和ab的部分，在ab对应的表格填=。
  - 2. 找出形如aQ的部分，竖排a与横排Q的FIRSTVT中每一个元素<。
  - 3. 找出形如Qa的部分，竖排Q的LASTVT集合与a填>。
- LR分析法

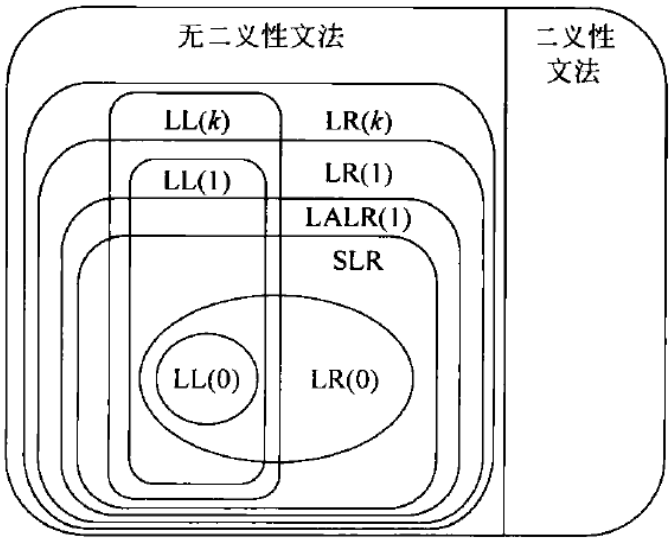


图 4.10　不同文法类的层次结构[10]

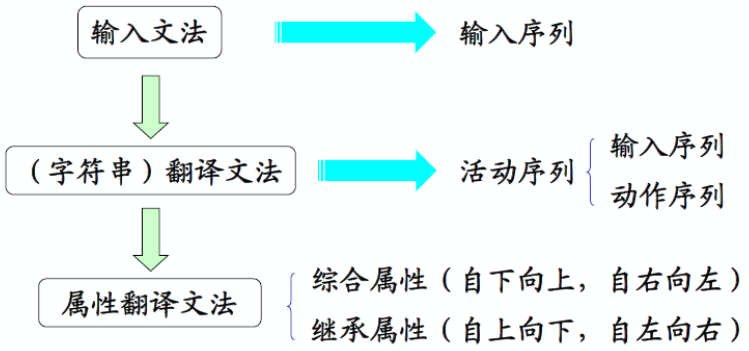
- 规约的是当前句型的句柄，是规范规约。（而算符优先规约的是最左素短语）

- 分析的每一步栈内符号串均为规范句型的活前缀，且与输入串的剩余部分构成了规范句型。
- 规约过程
  - ACTION[Si, ,a] = s，将a 推进栈，并设置新的栈顶状态为 S j 。 Sj = GOTO[S i , a]，并将指针指向下一个输入符号。
  - ACTION[Si, a] = r_d, 弹出β个符号和状态，(d) A->β，此时栈顶状态为S'，再把A推进栈，并设置新的栈顶状态为Sj，Sj=GOTO[s', A]
- 【算法】LR分析器构造
  - LR(0)的构造
    - 1. 将文法扩充，使得识别符号的规则只有一条。
    - 2. 列出所有的项目。
    - 3. （废弃）将有关项目组成集合，即DFA中的状态，所有的状态构成了LR(0)项目集规范族。
    - 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)。
      - A. 构造项目集闭包【待补充】
      - B. 定义GOTO函数【待补充】
- 有效项目
-

- 重点掌握
  - 理解简单优先文法和算符优先文法的概念
  - 根据语法树求素短语、最左素短语、简单优先文法的符号比较、算符优先文法的符号比较
  - 掌握构造递归下降程序框图的算法
  - 熟练应用消除左递归、FIRST集、FOLLOW集、求有效项目、有效项目集族的算法
- 优化 也有1/6

第五章. 语法制导的翻译

- 总体



- 输入文法：未插入动作符号时候的文法，可以通过推导产生输入序列。
- 翻译文法：插入后的文法，可以通过推导产生活动序列。
- 输入序列：就是输入序列，(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。
- 动作序列：从活动序列中去掉输入序列，则得到动作序列。
- 语法制导翻译：按翻译文法进行的翻译。
- 属性翻译文法
  - 综合属性：↑c，自底向上的计算。（自右向左
  - 继承属性：↓t1,n1，自左向右，自顶向下。@语法树

第六章. 符号表管理

- 主要内容

- 非分程序结构语言的符号表组织
- [重点和难点]分程序结构语言的符号表组织

- 最常执行的操作是插表和查表，遇到声明语句或者定义语句的时候，先查再插。
- 重点理解分程序结构程序栈式符号表。
- 题型：要求给出一个程序，能够分析程序运行到不同节点的时候符号栈的情况。

```
/*
|   数组 S   |
|    Y     |
|    X     |
|    J     |
|S 的模板/模块 |
| prev abp |
| ret addr |
|  abp(1)  | <-- abp(2)
|    Y     |
|    X     |
|    I     | <-- abp(1)
*/
```

点 2：

```
/*
|   数组 R   |
|R 的模板/模块 |
| prev abp |
| ret addr |
|  abp(2)  |
|  abp(1)  | <-- abp(3)
|   数组 S   |
|    Y     |
|    X     |
|    J     |
|S 的模板/模块 |
| prev abp |
| ret addr |
|  abp(1)  | <-- abp(2)
|    Y     |
|    X     |
|    I     | <-- abp(1)
*/
```

P166 2

考虑下面的类 ALGOL 程序：

```
BEGIN
    INTEGER I;
    REAL X, Y;
    READ(I, X, Y);
    EXAMPLEPROC(I, X, Y);
    PROCEDURE EXAMPLEPROC(INTEGER J, REAL X, Y)
    BEGIN
        STRING S(8);       ! 点 1
        ...
        BEGIN
            REAL R(J, J);  ! 点 2
            ...
        END
    END
END
```

画出执行到点 1 和点 2 时，运行时栈的内容。

答案解析

点 1：

- 符号表的使用贯穿编译的整个过程，不存在于运行的时候。
- 符号表分成四大类。标识符常量函数分类（？
- 重点和难点：分程序结构语言的符号表组织。

- **第七章. 运行时的存储组织及管理**
  - 主要内容
    - 静态存储
      - 在编译阶段由编译程序实现对存储空间的管理，和为源程序中的变量分配存储的方法。
    - 动态存储分配
      - 在目标程序 运行阶段 由 目标程序 实现对存储空间的组织与管理，和为源程序中的变量分配存储的方法。
  - 运行栈
- **第八章. 源程序的中间形式**

- 主要内容
  - 波兰表示
  - N元表示（三元，四元式）
  - 抽象机代码（专属于栈式指令集CPU体系架构）
- **第九章. 错误处理**
  - 主要内容
    - 错误处理程序的作用
    - 如何诊察和报告语法、语义错误
    - 错误处理技术
- **第十章. 语义分析和中间代码生成**
  - 主要内容
    - 语义分析的概念
    - 栈式抽象机及其汇编指令
    - 声明的处理
    - 数组信息向量
- **第十一章. 中间代码优化**
  - **主要内容**
    - 与机器有关的优化和与机器无关的优化
    - 局部优化、全局优化、循环优化
    - 基本块
    -
    - 全局数据流分析
  - **优化分类方法**

★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，比较复杂。

- 从优化层次
  - 与机器无关的优化技术：数据流分析，常量分析，公共子表达式删除，死代码删除，循环交换，代码内联等等。
  - 与机器相关：略，指令系统，寄存器资源。
- 从优化范围
  - 局部优化：基本块内进行的优化，如局部公共子表达式的删除。
  - 全局优化：函数/过程内进行的优化，例如全局数据流分析。
  - 跨函数优化：整个程序，例如跨程序别名分析，逃逸分析。
- **【算法】划分基本块**
  - 第一条语句，跳转语句能转移到的语句，跟在跳转语句之后的语句为入口语句。
  - 到下一个语句为止的所有语句都属于同一个基本块。
- **DAG图**
  - 作用：消除局部公共子表达式
  - 从DAG图导出中间代码的启发式算法
- **【算法】DAG图的生成**
  - 对于一个基本块内的中间代码序列，首先建立节点表（空），记录变量名和常量值和对应DAG图中节点的序号。
  - 然后按照以下规则：z = x op y，则在表中找x，若找到则记录节点号i，未找到则在DAG图中新建一个叶节点，假设节点号

仍然为i，标记为x（x为变量名则更改为x_0），在表中新增(x,i)，y同理。

- 在图中寻找中间结点并标记为op，且左节点i右j，若找到，记录其节点号k，未找到则新建中间结点将i和j作为左右子节点其相连。
- 在表中找z，若找到将z对应的节点号改为k，找不到则新建(z,k)。
- 重复以上步骤。

- **【算法】从DAG图导出中间代码的启发式算法**
  - 初始化一个放置DAG图中间结点的队列。
  - 选取一个未进队但父节点进队，或者没有父节点的中间结点n进队。
  - 若n的最左子节点符合上述条件让他进队，循环访问最左子节点进队。出现不符合条件则执行上个操作。
  - 逆序输出中间节点队列，即为计算顺序，整理即可。
- **数据流分析**
  - 到达定义分析（先算gen和kill集，然后用in算out）

$$in[B] = \cup_{B的前驱基本块P} out[P]$$
$$out[B] = gen[B] \cup (in[B] - kill[B])$$

  - 活跃变量分析（先算def和use集，然后用out算in）

$$out[B] = \cup_{B的所有后继S} in[S]$$
$$in[B] = use[B] \cup (out[B] - def[B])$$

- **定义-使用链→网→冲突图**
- **循环优化**
  - 循环不变式的代码外提
  - 循环展开
  - 归纳变量的优化
  - 条件判断的替换
  - in_line 展开（函数内联）

- **第十二章. 目标代码生成及优化**
  - 主要内容
    - 微处理器体系结构基础知识（4种现代微处理器体系结构）
    - 栈式
    - 累加器式
    - 寄存器-内存式（CISC）
    - 寄存器-寄存器式（RISC）
  - 构建冲突图
    - 变量冲突的基本概念
    - 通过活跃变量分析构建冲突图
  - 寄存器的分配和指派
    - 全局寄存器分配算法
      - 引用计数
      - [标红了]图着色
  - 指令选择
    - 套用模板
    - 窥孔优化
- **考试**
  - 综合属性求值规则
- **TODO**
  - LR与SLR的构建及证明
    - LR(0)的移进-规约，规约-规约冲突的判断?

- SLR，过程可以解决规约-规约冲突，FOLLOW集为空则为SLR文法。两种冲突都不存在也是SLR文法。
  - 先计算规范族，然后找出冲突的，
- LL的构建及证明
  - 文法的充要条件：

> 定理：文法G是LL(1)文法的充分必要条件是：对于G的每个非终结符A的任意两条规则A::=α|β,下列条件成立：
>
> 1、FIRST(α) ∩ FIRST(β) = Φ
>
> 2、若β=*=> ε，则FIRST(α) ∩ FOLLOW(A) = Φ

  - 不带回溯的证明↑
- 数据流分析
- ~~数组求法？~~
- 符号表管理？
- 冲突图，图着色算法
- ~~重看：12-11(3)，短语直接短语和句柄的过程~~
- 求活动序列和字符串？
- 编译过程中引入中间语言的优点？
- 三元式？
- 有效项目
  - 构造自动机
- 求活前缀集合

- 写出规范句型的推导过程，最后一步中确定αβ
- ~~直接求素短语？~~
- 算符优先文法，优先关系构造？
-

以上内容整理于 幕布文档