编译原理复习笔记

杨承昊[1]

(1.北京航空航天大学软件学院，北京 100191)

目录

1 引论

1.1 名词解释

1. 汇编语言写的也叫源程序
2. 低级语言：字位码，机器语言，汇编语言
3. 目标程序：目标语言表示的程序，**目标语言**：介于源语言和机器语言的语言，“中间语言” **一般为汇编程序或可重定位的机器代码**
4. 翻译程序：将源程序翻译为目标程序的软件工具，包括**汇编程序**和**编译程序**，以及其他各种变换程序。**翻译的实质：语义的等价**
5. 汇编程序：将目标程序转换为用机器语言表示的程序的工具，这个过程被称为汇编（较为容易，目标语言常与机器语言有一对一的关系）
6. 编译程序：将高级语言表示的程序转换为目标程序的工具，这个过程被称为编译（较为复杂）
7. 解释程序：对源程序/目标程序解释执行的程序。
8. 编译-解释执行系统：易于debug，易于移植，适合需要动态确定数据的程序

9. ”五阶段，七模块”：注意出错处理和符号表管理（登记信息，提供信息查询功能）贯穿五个阶段

10. 遍（pass）：对源程序从头到尾扫描一遍，生成中间代码或目标程序的过程称为遍（几遍=几次扫描）

11. 前端(**分析部分**)：与源程序有关的编译部分，包括词法分析，语法分析，语义分析，中间代码生成，代码优化

12. 后端(**综合部分**)：与目标机有关，包括目标代码生成

2 形式语言与自动机

2.1 名词解释

1. 单词：语言基本语法单位，包括:（**注释不算单词**）

    - 保留字
    - 标识符
    - 分界符
    - 常量

2. 符号串集合的0次幂是仅包含空串的集合

3. 符号串的形式定义：1）空串属于符号串，2）可以在已有的符号串基础上左加或右边加上符号集的符号

4. 闭包：星闭包与正闭包(并集中无符号串集合的0次幂)

5. 由...组成： ::=或者=>

6. 推导，最左推导(最左语法成分开始)，最右推导（最右推导）

7. **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

8. 文法形式化定义：四元组(V_n,V_t,P,z)

   - V_n：所有规则的左部，
   - 字汇表$V = V_n \cap V_t$，
   - 规则集P可唯一确定文法

9. 扩充的BNF（{}：0到多次重复，可以写上下标确定上下界；[]：0-1次；$(a|b)$：提公因式，小心括号可能不是**元符号**）

10. 推导的形式化定义：X=aUb,Y=aub,a,b均属于V的星闭包，存在规则U::=u时，有X=>G

11. 直接推导，间接推导，0步推导，*推导（0步推导+间接推导）

12. 句型(V*)，句子(V_t^*)，语言(V_t^*的元素(**句子**中，可由起始符号经过**间接推导**得到，其实说的就是句子的集合)

13. 字母表：符号的**非空**有限集

14. 一个文法对应一个语言，一个语言对应多个文法（这些文法等价：**文法等价性证明**）

15. 左递归，右递归，递归文法（有穷规则定义无穷语言），自嵌入递归（左部符号出现在右部中间）

16. 短语，简单短语，句柄（最左简单短语）（**相对于句型而言**）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 (简单短语要求只有一步推导)

    注意除非V_t中包含空串，否则短语不能为空串

17. 语法树（其实从语法树找句柄很快）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。

    一个推导对应一个语法树（从拓扑结构的意义上，形状可以不同）

18. 文法二义性：对于一个**句子**

    - 存在两种推导或者存在两棵语法树
    - 存在两个不同的最右推导
    - 存在两个不同的最左规约(**规范规约**)

      规范规约的过程中是否存在**两个句柄**

    文法二义性是不可判定的

19. 多余文法与有害文法 多余文法：规则左部和右部完全相同有害文法：存在着满足以下条件中其中一个的文法

    - 永远用不到的规则
    - 用到了就会无法停止推导（死循环）的规则(含有无法推出任何终结符号串的非终结符号)

    没有多余文法和有害文法的称为**压缩过**的文法（基本只考虑这个）

20. 乔姆斯基文法

    0型文法—短语文法–图灵机

$$(U ::= v, U \in V^+, v \in V^*) \tag{2.1}$$

    1型文法—上下文有关文法–线性界限自动机

$$(xUy ::= xvy, U \in V_n, x, v, y \in V^*) \tag{2.2}$$

2型文法—上下文无关文法–下推自动机

$$(U ::= v, U \in V_n, v \in V^*) \tag{2.3}$$

3型文法—正则文法–有限状态自动机

$$(U ::= Xv|v^{-}\Pi U ::= v|vX, U, X \in V_n, v \in V_t) \tag{2.4}$$

(**注意，3型文法只能是左线性或者右线性，否则是2型文法**)

$$L3 \subset L2 \subset L1 \subset L0 \tag{2.5}$$

2.2 DFA

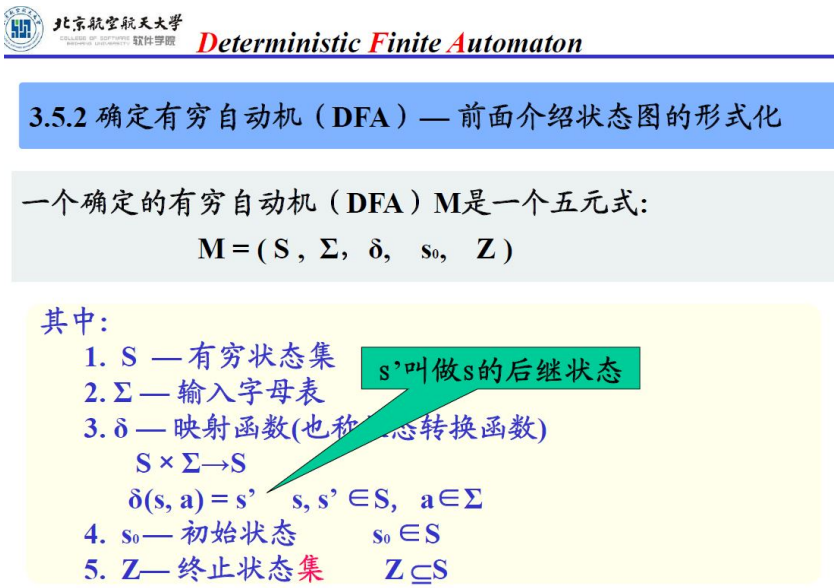

图 1: 确定有限状态自动机形式定义

若存在一条从初始状态到某一终止状态的路径，且这条路径上所有弧的标记符连接成符号串 α ,则称 α 为DFA M(接受)识别。

2.3 NFA

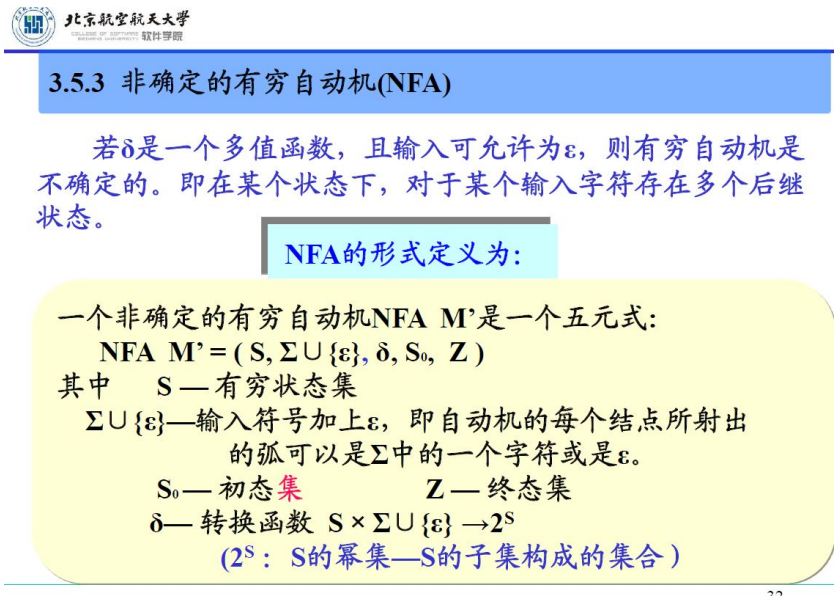

图 2: 非确定有限状态自动机形式定义

2.4 DFA与正则表达式的等价性

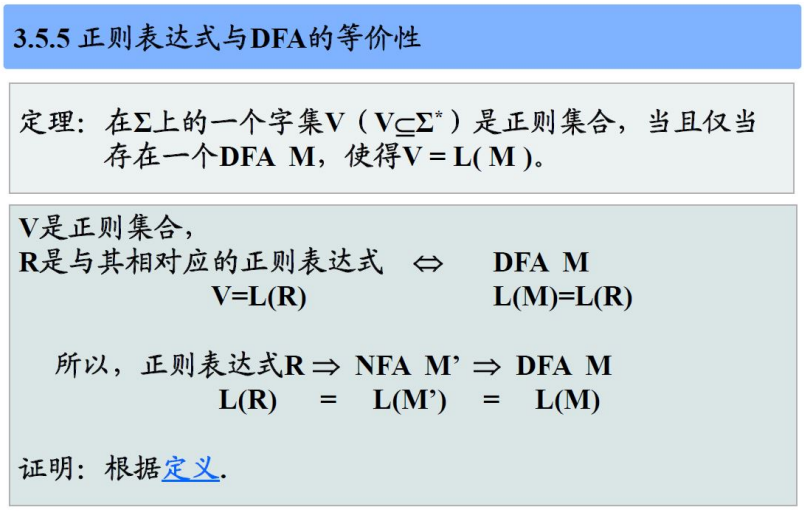

图 3: DFA与正则表达式的等价性

2.5 FA，正则表达式，3型文法互相转换

1. DFA to type-3: 定起始状态为起始符号，**终止状态Z改为规则:Z::=ϵ**
2. type-3 to DFA: （先将文法改为右线性）定起始符号位起始状态，设立终止状态Z，**对所有规则A::=t（$t \in V_t$），连边（A，t，Z）**，对所有规则A::=tB，连边（A,t,B），如此，得到NFA，确定化即可
3. REG to DFA: 语法制导方法，
   (a) 空串和纯字符：连边（X，a/ϵ，Y）
   (b) 连接：左连起始，右连终止，两个符号对应的NFA依次连空弧
   (c) 选择：起始态并发两个空弧连到两个符号对应的NFA，两个NFA分别以空弧汇聚在终止态
   (d) 重复：别忘了可以0次重复，所以起始要跳过NFA连空弧到终止，因为重复之间无符号，所以需要NFA终止态反向连空弧到初态
4. NFA to REG:先加上初态和末态，连空弧，反着语法制导方法，采用“缩圈法”，**注意，消去节点后，应保证边上信息正确，不漏掉任何一条边，假设消去的节点入度为n，出度为m，消去一个点将删去一个点和（m+n）条边，增加m*n条边**
5. type-3 to REG:（一样先化为右线性），然后”代入法/观察法”求解（基本范式有，但是一般直接观察应该就能猜个7788）
6. REG to type-3：（谨记3型文法特征：规则右部只可能为一个V_t或者V_tV_n（右线性）或者V_nV_t（左线性，但一般不转换为左线性））造一个开始符号S，对任何正则式r，有S− >r，然后选择（一个非终结符推导两个非终结符），重复（右递归构造），连接（新造一个非终结符顶替连接右端符号）

2.6 DFA最小化

对任意一个DFA，存在唯一一个最小化的DFA（无等价状态和多余状态）

- 多余状态：从初态开始无论如何不会经过的状态
- 等价状态：
  - 一致性条件：必须同为接受状态或同为不接受状态
  - 蔓延性条件：对于相同的字符输入，必须转换到相同的状态（无后继的和有后继的绝不等价）
- 可区别状态：若两个状态不等价，则称为可区别的。

“分割法”最小化DFA：同一个子集的状态等价，不同子集的状态可区别

（核心原理：每次迭代中用大区号来替代转移的目标状态，然后根据此确定是否分出新类）

3 编译过程

3.1 词法分析

输入 ：源程序

输出 ：合法单词流

任务 ：分析并识别单词，进行词法检查

3.1.1 单词存储格式

名-类别-值-其他

3.1.2 类别标识方案

按单词种类分类或者保留字和分界符采用一符一类

3.1.3 状态图

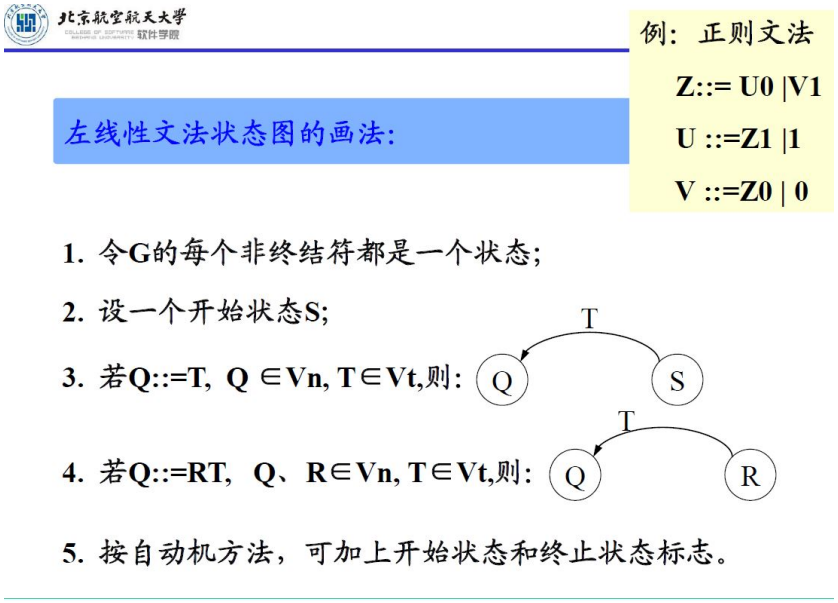

图 4: 左线性文法状态图画法

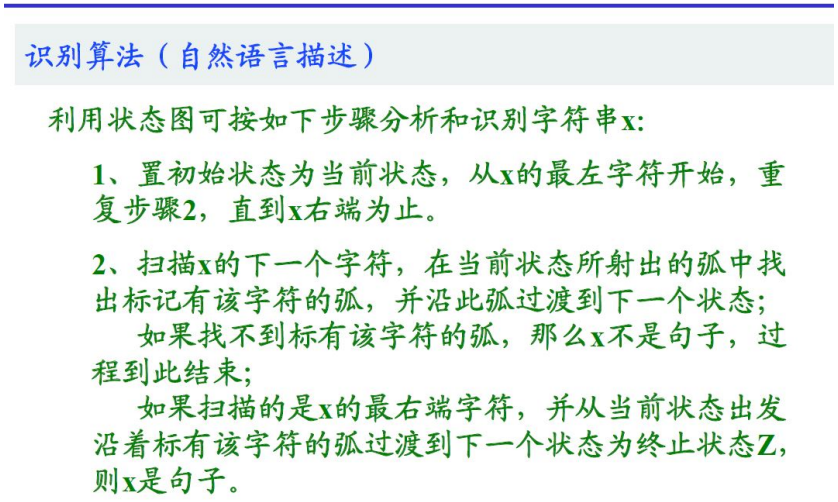

图 5: 识别过程

3.1.4 正则表达式

与3型文法等价，用3型文法描述的语言都可以用正则表达式描述

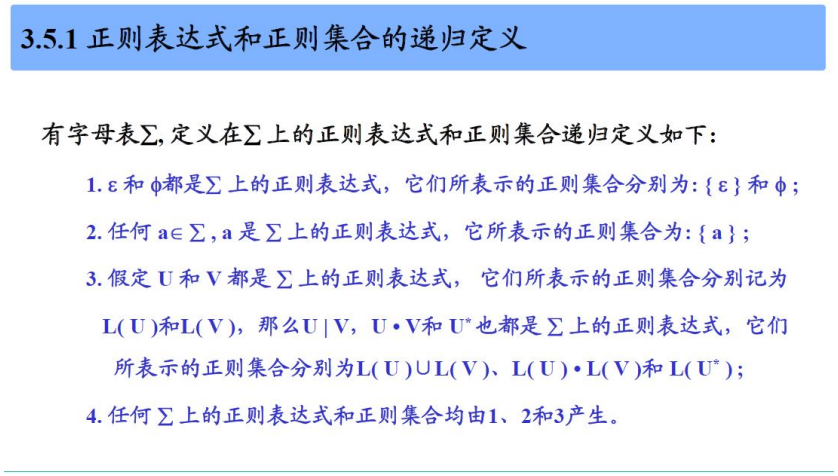

图 6: 正则表达式形式定义

ϕ指空正则表达式，ϵ指空串，不要搞混了

注意正则集合的概念（类似于文法识别的语言）

运算符优先顺序：

$$\mathrm{fl}''^{-} \tag{3.1}$$

正则表达式相等–正则表达式所表示的语言相等

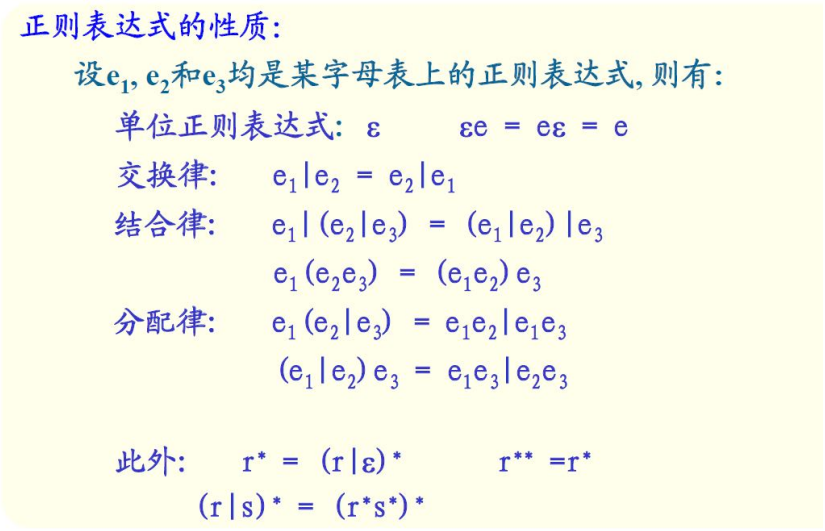

图 7: 正则表达式性质

可用正则表达式、NFA/DFA来辅助自动实现词法分析程序，具体来说，可以先设定文法，画NFA，再确定化（子集法（ϵ闭包,状态子集（吸收一个符号后再求ϵ闭包））），再最小化

LEX工作原理：构造DFA和生成控制管理程序

LEX会对每一个规则生成一个DFA，最后加一个初始状态，向所有DFA的初态连空弧，再将这个DFA确定化，然后生成DFA和控制程序

LEX二义性处理：1. 最长匹配原则 2.最优匹配原则（规则排列靠前的优先）

注意：最优匹配原则在最长匹配原则下进行

3.2 语法分析

输入 ：合法单词流

输出 ：语法树

任务 ：分析并识别语法成分，进行语法检查

两大方法：自顶向下，自底向上

编写程序注意：由谁来负责读“下一个符号”（调用者，被调用者都可以，但必须统一）

自顶向下：

3.2.1 自顶向下方法

- 递归下降子程序（最左推导，虽然不能有左递归，但还是会有右递归和自嵌入递归，所以需要递归）
- LL分析法（最左推导，自左向右扫描，自左向右分析和匹配，要求对于能推导到ϵ的非终结符，First集合与Follow集合交集必须为空—一个更正式的表述是若A的一个产生式可以推导出空，则A的Follow集合和A的其他产生式的First集合交集必须为空—是等价的，可以通过反证法证明）

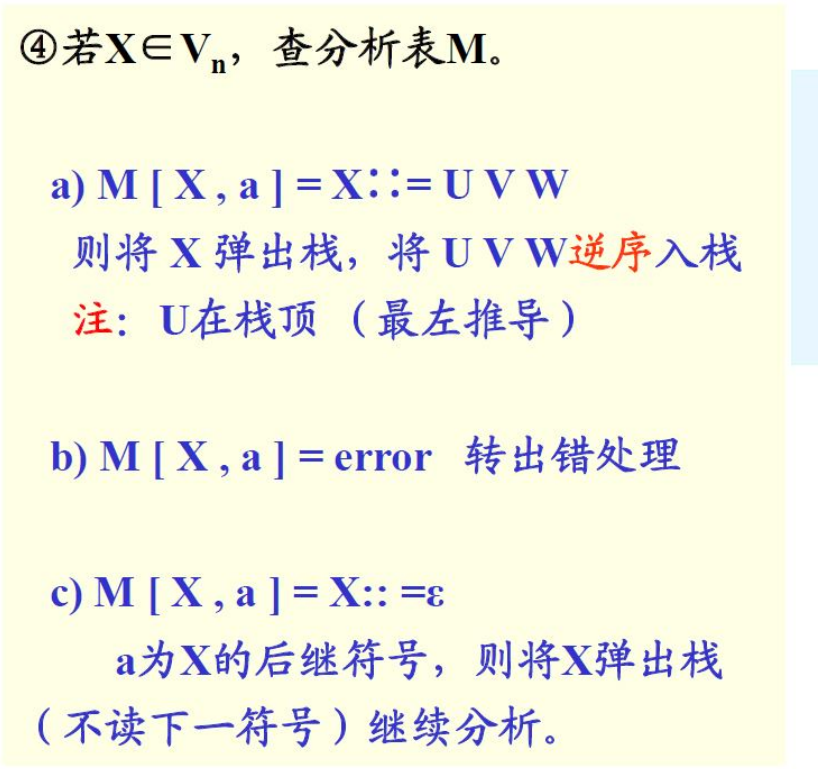

图 8: LL分析法（注意这里的**逆序入栈**）

– 分析程序运行：

  每次弹一个栈内符号a，和栈外符号b比较，

  若a是Vt，则a和b必须相等，并且仅在此时，读头向下一个字符移动（若为"#"，则结束），否则报错（b肯定是Vt）；

  若a是Vn，则查表，根据表中规则填入a推导的内容（注意逆序填入和推导出空串的情况，推导出空串不移入符号）（这时栈外字符和读头位置仍然保持）

– 与LR分析法不同：符号栈（LR是状态栈），分析顺序

– 构造表先要求First集合与Follow集合，求法简要概括如下：

  First集合：收集所有的first Vt，能穿透的要穿透（$Z ::= X1X2X3, X1, X2 ::=$

ϵ),如果都能穿透要把ϵ加入（First1，First2）（特殊情况：非终结符可以直接推导空串，那么空串必须属于该非终结符的First集合）

Follow集合：根据First集合可以求，注意对于A::=aB或者A::=aBC,C可以推导到空的情况，要将Follow（A）加入Follow（B）（Follow）

  * 求First集合，先求可以直接看到的Vt，记录集合子集依赖关系（产生式的第一个符号的First集合是产生式左部符号FIRST集合的子集），然后考虑穿透，最后整合
  * 求FOLLOW集合，先求可以直接看到的Vt，记录集合子集关系（后一个符号的First是前一个符号的Follow的子集），然后考虑结尾移入情况，最后整合，注意去掉空符号

- 有些非LL(1)的文法可以改写为LL（1），但并非都可以
- LL(1)文法的充分必要条件（其实是保证分析表中不会出现多重入口）：
  * 同一个非终结符的多条产生式的First集合交集为空
  * 若A的一个产生式可以推导出空，则A的Follow集合和A的其他产生式的First集合交集必须为空
  * LL(1)分析的过程中，将栈中符号倒过来看，从上往下排列，就是最左推导过程
- LL(1)文法的错误处理–构造同步符号集合

1) 把FOLLOW(A)的所有符号加入A的同步符号集。如果我们跳读一些输入符号直到出现FOLLOW(A)的符号，之后把A从栈中弹出，继续往下分析即可。

2) 只用FOLLOW(A)作为非终结符A的同步符号集是不够的(容易造成跳读过多，如输入串中缺少语句结束符分号时)。此时可将作为语句开头的关键字加入它的同步符号集，从而避免这种情况的发生。

3) 把FIRST(A)的符号加入非终结符A的同步符号集中。

4) 如果非终结符A可以产生空串，那么推导ε的产生式可以作为缺省的情况。这样做可以推迟某些错误检查，但不会漏过错误。

5) 如果终结符在栈顶而不能匹配，则可弹出该终结符并发出一条信息后继续分析。这好比把所有其他符号均作为该符号的同步集合元素。

图 9: LL（1）文法的同步符号集

注意这里FOLLOW集合不能含有空符号ϵ，注意"#"号加入Follow集合仅在与该符号串由初始符号退出，且位于初始符号末尾（实际上是将初始符号的FOLLOW集合移入）

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（即分析表中无两条以上规则），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$\overset{*}{\Rightarrow}$ ε，则FIRST(α) ∩ FOLLOW(A) = Φ

图 10: LL（1）文法的充分必要条件

左递归问题（死循，用扩充的BNF或者改为右递归），回溯问题

构造不需要超前扫描，无回溯的自顶向下分析器的条件：

1. 非左递归文法
2. First集不相交

- 左递归问题：

  （消除直接左递归）

  扩充的BNF：提公因子（注意把空串放在结尾），分析重复部分的pattern，用大括号扩出来

  改为右递归：

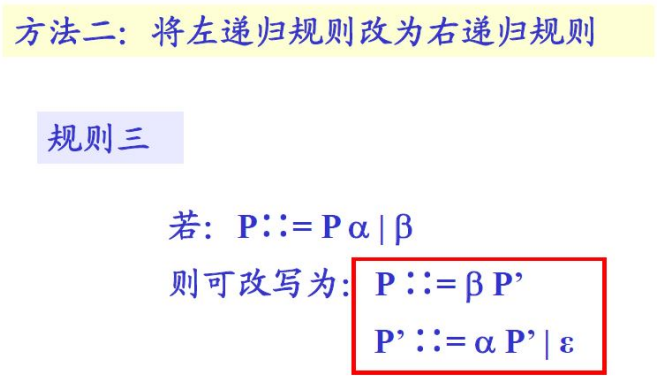

图 11: 改为右递归：消除直接左递归

（本质上是构造了一个递归符号做中间量）

（消除一般左递归（包括间接左递归））

先按照依赖顺序排列规则，第n条规则需要依赖的符号应该在前n-1条中，第1条规则是非终结符全部推导为终结符

然后从上往下扫描，逐条解决左递归，解决第i条左递归时，参考前i-1条，替换掉符号后，消除直接左递归

```
for i:=1 to n do
    begin
      for j :=1 to i-1 do
        把每个形如Ai::= Aj r的规则替换成
        Ai ::= (δ1|δ2|……|δk) r
        其中Aj ::=δ1|δ2|……|δk是当前Aj 的全部规则
        消除Ai规则中的直接左递归
    end
```

图 12: 消除一般左递归

- 回溯问题：提取First集公共元素A，拆规则S::=AB，B::=.....，以B为左侧的规则集中First集合不相交

  （虽然超前扫描可能可以解决回溯问题，但是效率低下（提前读取若干个字符以判断，本质上也有回溯问题（假读1个，2个....））

3.2.2 自底向上方法

（算符优先分析法，LR分析法）句柄识别问题，二义性问题（规约过程中两个句柄怎么办？）

- 算符优先分析法：比较两个终结符之间的优先级，确定规约/移进
  - OG文法：产生式集合中没有两个非终结符相邻的产生式
  - OPG文法：对OG文法，如果任意两个终结符之间的关系只有三种（等于，小于，大于），那么就是OPG文法
  - 优先函数（节省优先关系矩阵的存储成本，可以由栈里栈外两个函数，缺点是容易屏蔽问题）

- 注意两个符号之间的大小关系这时不对称，a<b不意味着b>a，a=b不意味着b=a
- 每次规约的不是句柄，实际上是最左素短语（只看终结符序列关系，$<, =, =, = \ldots\ldots, >$，注意不要取小于等于两端的终结符号，只取等于连接的中间部分）
- 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它素短语。
- 小于，等于移进，大于规约（向栈里找到第一个小于的情况）
- 优先关系确定

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 。

图 13: OPG优先关系

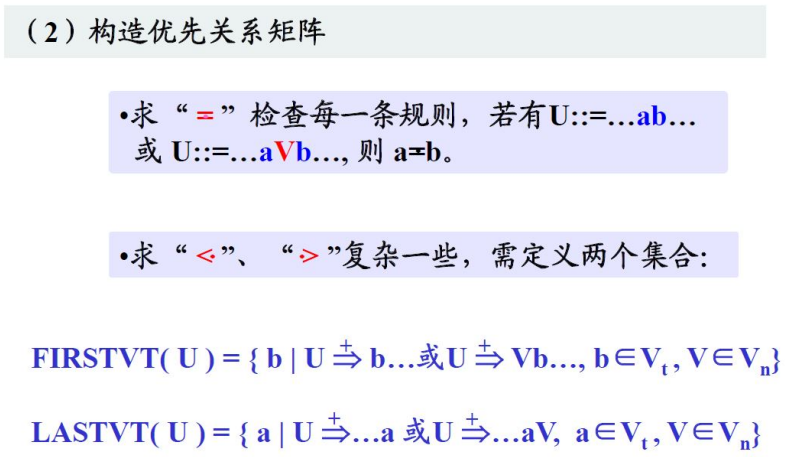

图 14: 构造优先关系矩阵

– 优先关系构造算法–构造FirstVT与LastVT集合

  * FisrtVT：从一步推导开始，对于U::=a...和U::=Va，（a是VT），将a加入FirstVT（U）；然后多步推导的情况，对于U::=V....，将FirstVT（V）加入First（U）
  * LASTVT：从一步推导开始，对于U::=...a和U::=aV，将a加入LASTVT(U)；对于多步推导的情况，对于U::=...V,将LASTVT（V）加入LASTVT（U）
  * 实际运行过程中，可以搞一个二元对(V,a)的栈，不断循环，用栈顶的二元对根据规则为各符号修改FIRSTVT/LASTVT，如果某个非终结符U的FIRSTVT/LASTVT有更新，将（U，a）压栈，弹栈，开始下一个循环。

– 优先关系构造算法–构造算符优先矩阵

  * 先找到U::=...aVb..的情况或者U::=..ab...的情况，设a=b
  * 对于U::=...aW..的情况，a<所有FIRSTVT（W）元素
  * 对于U::=..Wa..的情况，所有LASTVT(W)>a

– 算符优先分析法–结束条件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。

– LR分析法

  * 活前缀规范句型直到**第一个句柄的终结符号**的子串的任意前缀
  * 若一个2型文法构造的分析表没有多重入口，则称为LR文法
  * **并非所有的2型文法都是LR文法，只是大部分编程语言是**。

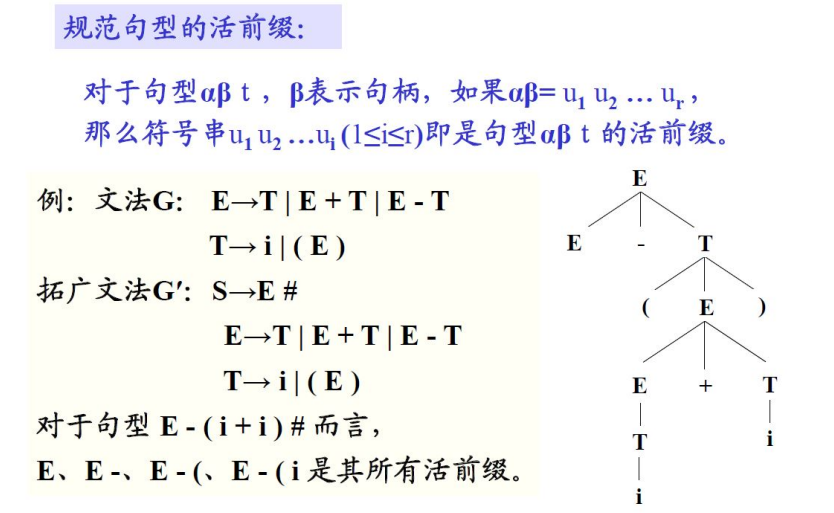

图 15: 活前缀

  * 项目：对每条产生式的右部从头到尾加点已经规约的和等待规约的部分 （注意，对于推导出空串的非终结符号，比如$A->\epsilon$，则对应项目为$A->.$
  * 构造项目->构造DFA（ϵ闭包与状态子集，这里的DFA因为要识别规范句型的活前缀，所以需要保证：
    · 除了初始状态以外，都是终止状态，从初始状态出发的任意路径都可以识别一个活前缀

- 状态中每个项目对该状态能识别的活前缀都有效
- 有效项目能预测分析的下一步动作(规约/移进/待约/接受/报错)
- DFA中的状态既代表了分析历史又提供了展望信息

)

* Action表的求法

*** ACTION表由项目集规范族求出**

根据圆点所在的位置和圆点后是终结符还是非终结符，把项目集规范族中的项目分为以下四种:

项目	种类	分析动作
$A\rightarrow\alpha.$	规约项目	规约
$E'\rightarrow\alpha.$ (E'为开始符号)	接受项目	接受
$A\rightarrow\alpha.a\beta$ $(a\in V_t)$	移进项目	移进
$A\rightarrow\alpha.B\beta$ $(B\in V_n)$	待约项目	无

图 16: Action表

* 项目集合（识别活前缀的DFA的状态）
* 项目集规范族（识别活前缀的DFA的所有状态的集合）
* LR分析表：识别规范句型的活前缀的DFA–构造GOTO-ACTION表
* LR(0)：在项目集合（注意不是规范族）中，不同时存在移进项目和规约项目，也不含有多个规约项目 (也就是GOTO-ACTION表一行过去全移进或者全规约)（太过简单没啥用）

* SLR(1)：对规约做限定，仅在follow集范围内规约（每个SLR（1）都是无二义的，但是无二义的不一定是SLR（1））

一般地，假定LR(0)规范族的一个项目集

$I = \{A_1 \to \alpha \cdot a_1\beta_1,\ A_2 \to \alpha \cdot a_2\beta_2,\ \ldots,\ A_m \to \alpha \cdot a_m\beta_m,$
$B_1 \to \alpha \cdot,\ B_2 \to \alpha \cdot,\ \ldots,\ B_n \to \alpha \cdot\}$

如果集合$\{a_1, \ldots, a_m\}$, $FOLLOW(B_1), \ldots, FOLLOW(B_n)$两两不相交（包括不得有两个FOLLOW集合有 # ），则：

1. 若输入a 是某个 a_i，$i = 1, 2, \ldots, m$，则移进；
2. 若$a \in FOLLOW(B_i)$，$i = 1, 2, \ldots, n$，则用产生式$B_i \to \alpha$进行归约；
3. 此外，报错。

——冲突的SLR(1)解决办法。

图 17: SLR(1)

* LR（1）：FOLLOW集范围太广，改用下一个的First集(必然是Follow集的子集，回想Follow集的构造过程）规约（向前看符号）

LR（1）适用的范围最广，但是状态数量巨大；每一个SLR（1）都是LR（1），反之不成立

LR(1)分析表的构造算法

1) 若项目[A→α•a β, b]属于I_k，且转换函数GOTO(I_k, a) = I_j 。
当a为终结符时，则置ACTION[k, a]为S_j。（b没有用处！）
2) 若项目[A→α•, a]属于I_k，则对a为任何终结符或“#”，置
ACTION[k, a] = r_j，j为产生式在文法G'中的编号。
（SLR文法中，是将FOLLOW(A)所对应的位置全置为规约）
3) 若GOTO(I_k, A) = I_j，这里则置GOTO[k, A] = j。其中A为
非终结符，j为某一状态号。
4) 若项目[S' →S •, #]属于I_k，则置ACTION[k, #] = acc
5) 其它填上“报错标志”。

图 18: LR(1)分析表构造方法

* LALR(1)：LR（1）+合并同心集（两个项目集只有向前看符号不一样）会迟报但不会漏报，能有效减少状态数量

3.3 符号表管理

符号表作用 语法规则，类型检查，变量地址分配，重定义检查，生成目标代码时参考

符号表种类 统一符号表，种类符号表，折中方案–共同部分放统一符号表，特殊部分在统一符号表拉个指针

非分程序（Fortran）符号表 线性，有序，散列

分程序符号表

- 作用域范围，循环也是一层作用域（外面不可以跳到循环体里面）
- 查表要按照作用域由内向外查

- 声明性出现：先检查是否重复，否，填表
- 引用性出现：从内向外逐层检查是否有定义，否，报错，是，取相关信息
- 标准标识符：填入最外层符号表（也可以单独建表，但更麻烦效率也更低）
- 分程序索引表的形成顺序——每个模块头的出现顺序
- 分程序符号表的形成顺序——每个模块(语法分析时的语法单位)识别顺序
- 栈式符号表：在生成正式的符号表之前，设一临时工作栈，存放临时工作表，分程序处理完成后，推栈，然后移动到正式符号表。

静态存储分配 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

- 不允许指针或动态分配
- 不允许递归调用过程
- 子程序数据区构造：隐式参数区（返回地址，返回值），显式参数区（形式参数，值或地址），局部变量和临时变量区

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。

- 栈式动态存储分配：进入子程序时，开辟堆栈，离开时，弹出堆栈
- 活动记录：局部数据区，参数区，display区

- 局部数据区：**注意数组要在栈上分配模板和地址指针（取决于实现，也可以将值放在栈上）**
- 参数区：显式参数区(形式参数的值或者地址）和隐式参数区（**prev abp**–调用模块的基地址，ret value–返回值，ret addr–返回地址，下一条执行指令的地址）
- display区：存放各外层活动记录基地址
- 变量二元地址（BL,ON）：BL==嵌套层次，非并列层次；ON==相对显式参数区的偏移
- **有没有可能prev abp不在display区？有可能，比如某个子程序B调用了与它并列的子程序C，B不是C的外层，故而B不在display区，但是prev abp指向B（CHAPTER7-18）**
- **建造display区的方法：**
  - **假如next=prev+1(嵌套层次上而言):复制prev的display区，再加上指向prev的指针**
  - **加入$next \leq prev$：复制prev的前next-1层**
- PPT上abp直接加偏移地址的说法不妥，直接看最后一页PPT，采用如下方法还原二元地址：

$$\begin{cases} abp + BL - 1 + NIP(\text{隐式参数区大小}) + ON & BL = LEV \\ display[BL] + BL - 1 + NIP + ON & BL < LEV \end{cases} \tag{3.2}$$

3.4 语义分析与中间代码生成

输出 ：中间代码

任务 ：分析语义正确性，产生中间代码（从定义上来看中间代码是一种目标语言，但是一般这里指的是诸如四元式、P-CODE之类的东西）

语义分析 上下文无关文法只能描述语法结构，语义分析需要上下文有关文法，然而这十分困难而且难以实现，所以一般使用语义动作（语法制导翻译，L-ATG/SL-ATG）

栈式计算机 BP：活动记录基指针 NP：堆指针 SP：栈指针

处理声明和引用时的语义动作（填表和查表）：

1. 处理声明时，将实体名字和有关信息尽可能填入符号表（当然，需要先检查有没有重定义）
2. 处理实体引用时，检查类型等信息是否正确，从符号表中获取地址等信息，生成目标代码

变长实体的动态存储方案 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

可变长实体存储在堆中，而其指针存放在符号表中

数组模版（数组向量） 3n+2：编译时不管是常界数组还是变界数组，数组模板的长度都不变（记得符号表分配空间的时候）

对于常界数组，编译时即可确定所有参数，而变界数组，要靠运行时指令来确定参数

3（L（i），U（i），P(i)）n+2(n–数组维数，RC（固定偏移量）)

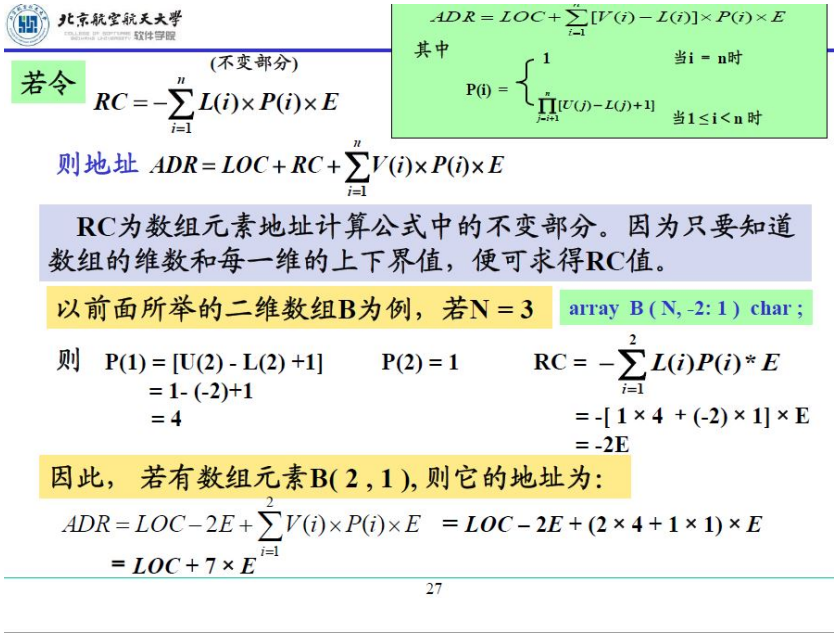

图 19: 数组模版

提示：为了加速计算，一连串的Vi*Pi*E可以通过递推公式来计算，能大大降低复杂度

中间代码的优势 ：

- 易于移植
- 易于优化

中间代码

- 波兰表示：不需要括号便可进行表达式计算（后缀表达式，语法树后序遍历，可用算符优先分析+符号串翻译文法生成），也可以用于其他编程语言的中间形式，但是优化起来不方便
- 三元式：操作指令，左操作数，右操作数。不方便优化：因为调整顺序和增加删除修改都需要调整中间代码–代码中含有指令的序号

- 间接三元式：将执行顺序和三元式顺序分割开来，执行的顺序就可以随意改动了
- 四元式：操作指令，操作数1，操作数2，结果。（结果是临时存储计算结果的位置，可以是寄存器等地方。）四元式优化较为容易。
- P-CODE:抽象栈式计算机。有寄存器，代码存储器，堆栈式数据及操作存储。 实际上是波兰表示形式的中间代码

语法制导翻译

1. 活动序列：输入序列+动作序列，活动序列由翻译文法（插入动作符号的文法）产生
2. 语法制导翻译：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过程。
3. 翻译文法是上下文无关文法
4. 符号串翻译文法：输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。
5. 属性翻译文法：只要求属性值计算的依赖关系不要成环，实际可能很难做
6. L-ATG:要求必须是LL（1）文法 属性值的计算可以用自顶向下，自左向右的方法计算（继承属性），也可以用自底向上，自右向左的方法计算（综合属性），开始符号的继承属性给定，终结符号的综合属性给定

   注意这里产生式右部的非终结符号的综合属性只靠以他为根的子树计算（靠下部以他为左部的产生式计算）

   “出现在产生式左边的继承属性和出现在产生式右边的综合属性不由

所给的产生式的属性求值规则进行计算。它们由其它产生式的属性规则计算。”

求值顺序：从产生式左部开始，求左部继承，然后对右部每一个符号，求完继承和综合后再求下一个，最后求左部综合

7. SL-ATG:属性值的计算为直接赋值的形式，对于L-ATG，总是可以转换为SL-ATG：

   (a) 设立动作符号@f，继承属性为输入参数，综合属性为输出

   (b) 修改产生式，加入新的动作符号，引入新的赋值规则

   (c) 继承属性：继承属性值（传实参值）

   (d) 综合属性：属性变量名（传地址，返回时有值）

   (e) 1)具有相同值的属性取相同的属性名。（这样可省去不少属性求值规则）

   (f) 2) 产生式左部的同名非终结符使用相同的属性名。（递归下降分析法规定每个非终结符只编写一个子程序!）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。

3.5 出错处理

- 语法错误(不符合语法规则)
- 语义错误（不合适的语义或者超过具体计算机限制，比如常数溢出，静态数据区溢出，动态数据区溢出
- 尽可能多的发现问题，不要一碰到错误就炸（同步符号集），尽可能地提供提示
- 违反语法、语义、编译系统限制的，由编译程序负责检出，而下界溢出，动态存储区溢出，计算溢出由**目标程序**检出

- 报告错误两种形式：边分析边报告，分析完了再报告
- 错误改正：很难实现
- 错误局部化处理：编译系统发现错误后，要尽可能把错误局限在小的范围内，避免错误扩散影响到程序其他部分的分析。一般原则：发现错误以后，跳过所在的语法成分(词法-跳过单词，语法和语义-跳过语句)，继续往下分析
- 运行时错误：保存现场，停止运行，报错

3.6 代码优化

要求 必须不能改变语义，在空间和时间上要提高代码运行的效率

优化的种类

- 机器相关：充分利用各种硬件资源
- 机器无关：一般指中间代码上的优化

或者，另外一种分类法：

- 局部优化：基本块内的优化
- 全局优化：数据流技术等
- 循环优化：循环语句生成的中间代码上的优化

基本块

- 无分支，无循环
- 所有转向该基本块的入口都指向该基本块的第一条语句

- 只有一个出口一个入口，顺序执行

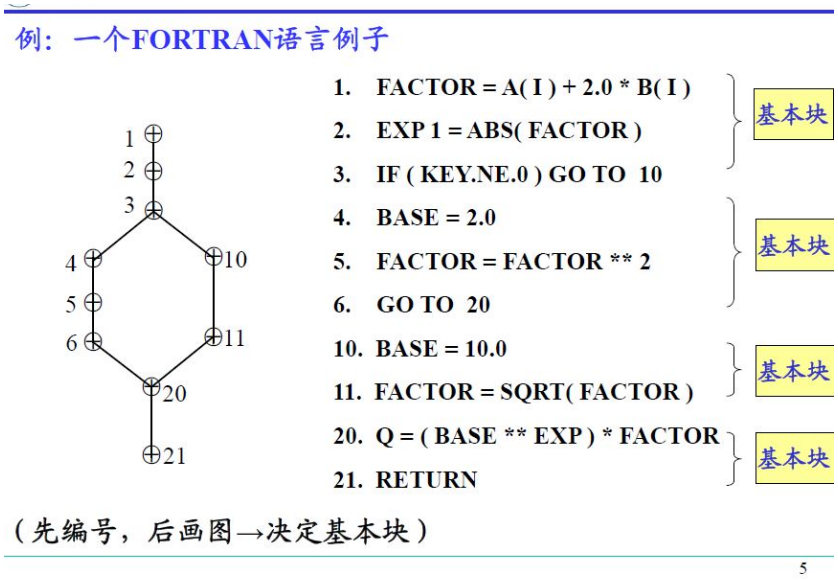

图 20: 基本块确定实例（先标号后画图）

局部优化

1. 代数变换（表达式中的常数预算，数组模版的参数）
2. 运算强度削弱（用位移，INC指令替代乘，某些除和+1）
3. 复写（copy）传播（相等变量如果可以安全替换，就直接替换成相同的，特别的，如果变量始终等于一个常数，那么可以将这个变量的所有出现替换为常数，并在可能的地方应用代数变换）
4. 删除公共子表达式（DAG（此时语法树变成语法图）中可以找出来）
5. 删除冗余代码（不会执行的代码，比如永真永假）

循环优化

1. 不变式代码外提（频度削弱）

2. 循环展开（空间换时间–生成更多代码，减少测试和转移代码执行，但也要权衡时间和空间是否合适，比如如果展开后代码巨多，减少的执行次数相比之下少的可怜，就没有必要（可以采用多路循环展开））

3. 归纳变量和条件判断的替换

4. 嵌套循环变成单层，相同形式循环合一

其它

- 内联函数
- 多步跳转转一步

3.7 目标代码生成