

编译原理速通

速通这玩意真的有可能吗?

但我们可以把它当小抄! 哈哈哈!

里面所有的"书"均指清华教材编译原理第3版(紫书)

这里都是总结性质的内容, 请结合书本观看.

里面所有的表格, 包括LL(1)驱动的预测分析表, 还有算符优先关系表等, 格子表达的二元关系的顺序是<纵轴值, 横轴值>

默认你懂符号串运算和离散数学

文法的定义

G={Vn, Vt, P, S} (常表示为G[S])

- V_n 是非终结符集合
- V_t 是终结符集合
- 下面用 V 表示 $V_n \cup V_t$
- P 是规则集合
- S 是开始符(不是集合), 属于 V_n , 至少在一个 P 的规则中作为左部出现

相关定义

- 句型是由 G 从 S 推导出来的符号串
- 所有符号都是终结符的句型是句子
- 语言 $L(G)$ 是所有 G 能推导出来的所有句子的集合

文法的分类 (语言从复杂到简单, 限制从少到多)

0型文法

- 对应图灵机
- 只要符合文法的基本定义就是0型文法, 即 P 中每一个规则 $\alpha \rightarrow \beta$ 中, α 至少包括一个非终结符

1型文法 (上下文有关文法)

- 是0型文法
- 对应于线性有界自动机(受到一点点限制的图灵机)
- 定义: P 中每一个规则 $\alpha \rightarrow \beta$ 有 $|\alpha| \leq |\beta|$ (特例: $|\alpha|=1, \beta=\epsilon$ 算符合)
- 每一个产生式都不会使长度变短

2型文法 (上下文无关文法)

- 是1型文法
- 对应于下推自动机(DFA加一个无限大的栈)
- 定义: P中每一个规则都为 $A \rightarrow \beta$, A是非终结符, $\beta \in V^*$

LR(k)文法

- 适用于LR分析
- 2型文法经过更强的限制得到:
- 考试中k=0或1

简单优先文法

- 是2型文法
- 任何语法符号组成的序偶存在最多一种优先关系(=,<,>)
- 适用于简单优先分析法

算符文法 (OG文法)

- 是2型文法
- 任何规则中不包含两个非终结符相邻的情况
- 性质:
 - 可以推出任何句型中也都不包含两个相邻的非终结符
 - 在一个句型中, 终结符若和一个非终结符相邻, 他们在句型的所有短语里一定总是一起出现
- 符号该文法的任意句型如下:

$N_1a_1N_2a_2\dots N_na_nN_{n+1}$

其中 N_i 要么是空, 要么是非终结符, a_i 是终结符

算符优先文法 (OPG文法)

- 是算符文法, 两个非终结符不会相邻
- 任何终结符组成的序偶存在最多一种优先关系(=,<,>)
- 适用于算符优先分析法

LL(k)文法

- 是2型文法
- 考试中只有k=1
- k=1时, 适用于确定的自顶向下分析
- 2型文法经过更强的限制得到: 它的 SELECT 集合满足 $\text{SELECT}(A \rightarrow \alpha) \cap \text{SELECT}(A \rightarrow \beta) = \emptyset$
- L:从左向右扫描句型; L:使用最左推导; k:只要向右看k个符号就可以决定使用哪个产生式推导

3型文法 (正则文法/正规文法)

- 是2型文法
- 对应于有限状态自动机, 即DFA, 也同样对应NFA
- 正则表达式用来表示此文法
- 分为左线性文法和右线性文法
- 定义: P中每一个规则都为 $A \rightarrow a \mid aB$ (左线性文法) 或 $A \rightarrow a \mid Ba$ (右线性文法), A和B都是非终结符, $a \in Vt^*$

对分类的理解

- 从0 \rightarrow 1 (限制 α 长度小于 β) \rightarrow 2 (限制 α 长度为1) \rightarrow 3 (限制 β), 限制逐渐增强, 因此可以说1型文法是0型文法, 2型文法是1型文法等
- LR(k), LL(k), 简单优先文法, OG 和 OPG 都是2型文法, 为了适用于各自的分析方式构造的
- LR(k) 比 LL(k) 和算符优先文法限制少
- "上下文无关"比"上下文有关"约束更为强烈, 上下文有关推出的语言更复杂, 比如 $a b(c)$;是一个变量定义还是一个函数声明取决于符号c的含义, 即先前c的定义, 这个语句的解析上下文无关比更为复杂
- **没有一种语法分析适合所有2型文法, 所有分析方式都有其对应的适用范围**
 - DFA/NFA - 3型文法
 - 确定的自顶向下语法分析(递归下降子程序/表驱动) - LL(1)文法
 - 简单优先分析法 - 简单优先文法
 - LR分析 - LR文法
 - 而大多数编程语言连2型文法都不是...所以编译原理这学科也就图一乐, 考过就行了
- 如果说一个语言不是上下文无关的, 它大概率是1型文法
- C/C++ 不是2型或3型文法
- Markdown 不是2型或3型文法, 因为其中*的含义取决于上下文
- p10语言是2型LL(1)文法, 不是正则文法
- html 是2型文法, 没有任何内嵌(css, javascript等)的html是正则文法

2型文法的语法树

- 由于P中每一个规则都为 $A \rightarrow \beta$, A是单个非终结符, 故可以对一个句子构建语法树
- 语法树用来表示2型文法的推导过程
- 如果每次推导都从最右边开始, 称为**最右推导**(规范推导), 得到右句型(规范句型)
- 若可以构造多个最左/最右推导过程, 称此文法是**二义的**
- 若一个语言的所有文法都是二义的, 称此语言先天二义
- 构造语法树有两种方法, 自顶向下和自底向上
- 对于每一棵子树, 称:
所有的叶子节点构成的符号串 是 该句型相对于根节点的**短语**, 若子树高度为2(直接推出), 则为相对
于该直接推出规则的**直接短语**
- 右句型的直接短语称为**句柄**

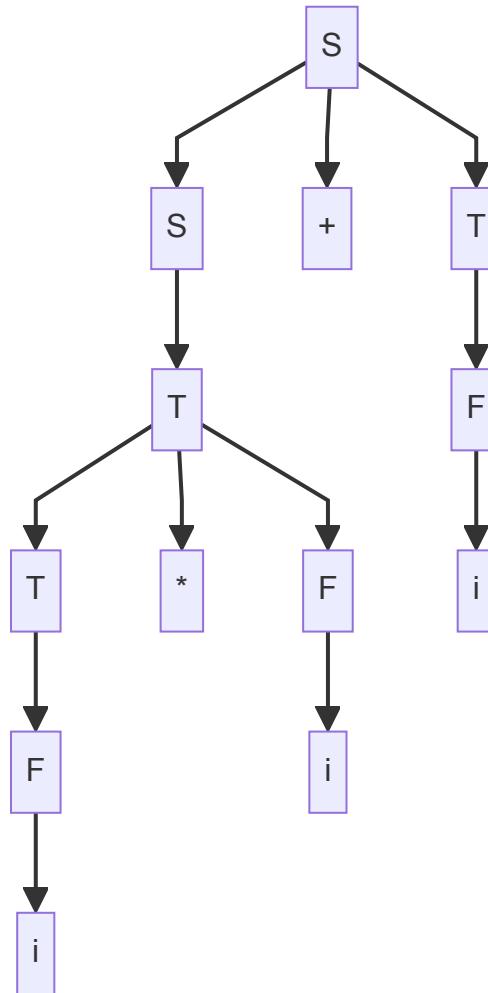
举一个例子:

G[S]:

- $S \rightarrow T|S+T$
- $T \rightarrow F|T^*F$
- $F \rightarrow (S)|i$

句型: i^*i+i

该句型的语法树: (用Typora看更香)



- 此时句型*i^{*}i+i*的短语有:
 - i^*i+i (相对于S)
 - i^*i (相对于T) (相对于S)
 - i (相对于F) (相对于T) (直接短语, 相对于规则 $F \rightarrow i$)

词法分析

- 把字符流变成单词序列
- 需要借助语言的词法规则, 通常是正规文法
- 通过正规语法构造NFA, 然后转为DFA, 再进行DFA的化简 (书p47)

确定的自顶向下语法分析 (LL(1)文法)

- 从S开始, 不断地选择表达式拓展成目标的样子, 关键在**避免错误的选择**

• 我们需要构造SELECT集以进行正确的选择

SELECT集由FIRST集和FOLLOW集构造, 结合二者信息共同判断是否选择该规则

- $\text{FIRST}(\alpha) = \{t \mid \alpha =^* > t\beta, t \in V_t, \alpha, \beta \in V^*\}$
(相对于句型) 句型推出的所有结果中的开头终结符
若 $\alpha \in V_t$, 则 $\text{FIRST}(\alpha) = \{\alpha\}$ (显然如此)
- $\text{FOLLOW}(A) = \{t \mid S =^* \dots A \dots, t \in V_t, \beta \in V^*, A \in V_n\}$
(相对于非终结符) 非终结符后面紧跟着的所有终结符
- $\text{SELECT}(A \rightarrow \alpha) = (\text{FIRST}(\alpha) - \{\epsilon\} \cup \text{FOLLOW}(A))$
- 不管是FIRST, FOLLOW还是SELECT, 里面只有终结符号和#和 ϵ

• LL(1)的分析法:

- 递归下降子程序法 (如果依靠程序员的本能写parser大概率会得到这个东西)
对语法的每一个非终结符都编一个分析程序, 它们之间相互调用形成递归
- 表驱动 (书p93) 看后面

• 判断给定2型文法是不是LL(1)

- 先求出能推出 ϵ 的非终结符, 如果某个非终结符A能推出 ϵ , 等下给他的FIRST集加上 ϵ (相当于单独考虑 ϵ)
- 其实SELECT集不包含 ϵ , 所以如果不用写出FIRST集, 你根本无需考虑 ϵ
- 求所有文法文法符号的FIRST集: 这里有一些技巧减少重复计算
 - 记住先不考虑FIRST包含 ϵ
 - 终结符的FIRST集只有它自己
 - 按定义求一些简单的FIRST集
 - 书本: 若 $X, Y_1, Y_2 \dots Y_n \in V_n$,
 $X \rightarrow Y_1, Y_2 \dots Y_n$,
且 $Y_1, Y_2 \dots Y_{i-1} =^* > \epsilon$,
则 $Y_1, Y_2 \dots Y_i$ 的FIRST集合包含在 $\text{FIRST}(X)$ 中 (所有FIRST集不含 ϵ) ($0 \leq i \leq n$)
意思是, 若有 $X =^* > Y$, 则 $\text{FIRST}(Y) \subseteq \text{FIRST}(X)$ (所有FIRST集不含 ϵ)
 - 最后给能推出 ϵ 的非终结符FIRST集加上 ϵ
- 求所有规则中右边符号串的FIRST
 - 若以非终结符号开头, 则其FIRST集就是该终结符
 - 其他的参照上面
- 计算所有非终结符的FOLLOW集
 - FOLLOW集中没有 ϵ
 - $\# \in \text{FOLLOW}(S)$, S为开始符
 - 若有 $A \rightarrow \dots B \beta$, 则有 $\text{FIRST}(\beta) - \{\epsilon\} \subseteq \text{FOLLOW}(B)$
 - 若有 $A =^* > \dots B$, 则有 $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$
- 根据定义写出所有规则的SELECT集
- 看看包含同一个非终结符的规则的SELECT有没有交集, 有则不是LL(1), 也就是LL(1)分析表的一个格子只能有一个规则

• 表驱动LL(1)分析程序

- 见书p94
- 先判断给定2型文法是不是LL(1), 这之后你有了SELECT集
- 构造LL(1)预测分析表:
 - 横轴为终结符, 纵轴为非终结符
 - 对于每个格子 $[A, \alpha]$, 若终结符 $\alpha \in \text{SELECT}(A \rightarrow \alpha)$, 把 $A \rightarrow \alpha$ 放到格子中
- 表驱动程序怎么运行?
 - 句子括号用#表示, 即在句子后面加上#作为结尾, 在栈顶加入开始符S作为开头
 - 先看分析栈顶的非终结符A, 再看剩余符号串的最左部终结符 α , 选取规则 $[A, \alpha]$
 - 运用该规则处理栈顶的非终结符. 即弹出它, 它根据规则产生的东西入栈
 - 如果分析栈顶产生终结符 β , 将其删去, 同时删去剩余符号串的最左部终结符, 它应该也是 β , 此时称"接受 β "
 - 栈顶的非终结符推出 ϵ 时, 弹出它即可
 - 栈的最后一个元素变为 ϵ 时, 程序结束
 - 我们可以看出, 分析栈经历的过程, 从S到目标字符串, 正是自顶向下

• 对于大部分非LL(1)的2型文法, 可以将其转换为LL(1), 使其适用确定的自顶向下语法分析

- 提取左公因子

例子:

$G[S]:$

- $S \rightarrow aSb$
- $S \rightarrow aS$

先提取a:

- $S \rightarrow aS(b \mid \epsilon)$

再把括号转换为非终结符:

- $S \rightarrow aSA$
- $A \rightarrow b$
- $A \rightarrow \epsilon$

- 消除左递归(变成右递归)

什么是左递归?

- 直接左递归:

- $S \rightarrow Sa$
- $S \rightarrow b$

- 间接左递归:

- $A \rightarrow Ba$
- $B \rightarrow Ab$
- $B \rightarrow \epsilon$

怎么消除左递归?

- 直接左递归: 改写上例变成右递归:
 - $S \rightarrow bS'$
 - $S' \rightarrow aS' | \epsilon$
- 间接左递归: 通过代入把间接左递归文法改写为直接左递归文法, 再消除
此例中把第一个规则代入第二个规则, 得到:
 - $B \rightarrow Bab$
 - $B \rightarrow \epsilon$

自底向上优先分析 (2型文法)

- 从目标句型开始, 不断地选择表达式进行归约, 最后只剩 S , 关键也在**避免错误的选择**
- 按一定规则求出该文法所有符号的**优先关系**, 然后按该关系求出**句柄**
- 上面提过, 右句型的直接短语称为**句柄**
- **简单优先分析法 (简单优先文法)**
 - 适用于简单优先文法
 - 简单优先关系是在 V 集合, 全体文法符号上的二元关系
 - 因为打不出某些符号, 这里直接用 $=, <, >$
 - 接下来的所有 X, Y 都属于全体文法符号, 即 V
 - 若有规则 $A \rightarrow \dots XY \dots$ 则称 X 和 Y 的优先性相等, 记作 $X=Y$ (注意是 X 在左, Y 在右, 后面也是)
 - 若有规则 $A \rightarrow \dots XB \dots$ 且 $B=+>Y$ 则称 X 的优先性小于 Y , 记作 $X < Y$ (Y 的层数比 X 大, Y 比 X 先归约)
 - 若有规则 $A \rightarrow \dots BD \dots$ 且 $B=+>X$ $D=*>Y$ 则称 X 的优先性大于 Y , 记作 $X > Y$ (X 的层数也许比 Y 大)
 - 这个关系不满足自反/对称/传递! 顺序很重要, $X=Y$ 不一定 $Y=X$, $X>Y$ 不一定 $Y < X$
 - 这个关系不满足三歧性(有可能没有关系), 但根据简单优先文法的定义, 确定顺序的两个元素间最多只有一个关系存在
 - 求出关系表, 表先纵轴后横轴
 - 怎么使用这个表格归约? (书p107)

• 算符优先分析法 (算符优先文法)

- 需要使用算符优先文法(OPG), 见上
- 算符优先文法只考虑终结符的优先关系
- 优先关系实际上是某种**结合律**的体现
- 算符优先关系可以简单地以下例子直观说明:
 - 一个算符优先于自己, 意味着它服从左结合
比如 $+>+$ 意味着 $1+2+3 = (1+2)+3$
 - 相反, 一个算符优先性低于自己, 意味着它服从右结合
 - 任何运算符优先性大于 $\#$, 除了它自己, 这意味着它右结合, 它又总是位于句子最右端, 因此被最后归约
 - 对于括号类的算符, 往往有 $A \rightarrow (B)$ 导致 $(=)$, 这代表他俩同时归约, 而 $)$ 和 $($ 的关系往往没有定义, 其次, $)$ 往往优先于其他所有算符, 其他所有算符也优先于 $)$, 这代表右括号左结合; 相反的, $)$ 优先性往往低于其他所有算符, 其他所有算符优先性也低于 $($, 这代表左括号右结合
- 如何计算任一终结符对的优先关系? (书p110)

- 我们需要FIRSTVT集和LASTVT集
 - $\text{FIRSTVT}(B) = \{b \mid B=+>b\dots \text{ 或 } B=+>Cb\dots\}$
非终结符B能推出的所有句型中的第一个终结符
 - $\text{LASTVT}(B) = \{b \mid B=+>\dots b \text{ 或 } B=+>\dots bC\}$
非终结符B能推出的所有句型中的最后一个终结符
- 对如下形式的规则

- $A->\dots ab\dots$
- $A->\dots aBb\dots$

(他们两个之间不隔着其他终结符, 因为这种奇怪的结合律只能应用于相邻终结符)

(空白处表示这两个终结符不能相邻, 故没有优先关系)

有 $a=b$ 成立

- 对 $A->\dots aB\dots, \text{ 有 } a < \text{FIRSTVT}(B)$

即对每一个 $b \in \text{FIRSTVT}(B)$, 有 $a < b$

(同样, 他们两个之间不隔着其他终结符)

- 对 $A->\dots Ab\dots, \text{ 有 } \text{FIRSTVT}(A) > b$

即对每一个 $a \in \text{FIRSTVT}(A)$, 有 $a > b$

(同样, 他们两个之间不隔着其他终结符, 记住这点就行了)

- 我们如果有 n 个函数, 存放优先关系要占用 $O(n^2)$ 的空间, 能不能把它优化一下? (书p118)

虽然优先关系不是偏序关系, 但我们可以构造两个用整数 f, g 表示的优先等级 (没给出证明, 不明白为什么可以构造捏), 使得:

- $a=b$ 有 $f(a)=g(b)$
- $a < b$ 有 $f(a) < g(b)$
- $a > b$ 有 $f(a) > g(b)$

总是成立.

这样就只需 $O(n)$ 的空间存储关系, f, g 称为优先函数

怎么构造这样的函数?

- 对每一终结符 $a \in V_t$ (包括#), 初始化 $f(a)=g(a)=1$, 随后的操作只增不减
- 若 $a > b$ 而 $f(a) <= g(b)$, 令 $f(a)=g(b)+1$
- 若 $a < b$ 而 $f(a) >= g(b)$, 令 $g(b)=f(a)+1$
- 若 $a=b$ 而 $f(a) \neq g(b)$, 令 $\min\{f(a), g(b)\} = \max\{f(a), g(b)\}$ (只增不减)

优先函数的直观含义

- **f意味着左结合性**(我自造的词), **g意味着右结合性**
- $f(\#)$ 和 $g(\#)$ 总是最低, 这意味着它最后结合
- 对于括号类, $f(\()$ 和 $g(\))$ 总是最低, $g(\()$ 和 $f(\))$ 总是最高, 因为左括号右结合, 右括号左结合
- 对于变量类(i), $f(i)$ 和 $g(i)$ 总是最高, 它总是和其他算符先结合

它有一个问题: 有的关系没有定义, 但这优先函数全能比较出来

LR分析 (LR文法)

- 一种自底向上分析
- 使用规范归约 (最左归约, 规范推导的逆过程)

分类	特点	优点	缺点	实际应用
SLR	使用 FOLLOW 集, 基于 LR(0) 项目	实现简单	分析能力弱, 易冲突	适合简单文法
LR(1)	使用 1 个向前看符号	分析能力强, 处理更多上下文无关文法	项目集大, 表格大	较为常用
LR(k)	使用 k 个向前看符号	理论上分析能力强	表格过大, 实用性差	很少使用
LALR	合并同核心的 LR(1) 项目	减小表格大小, 实际应用广	可能无法处理所有 LR(1) 文法	编译器中常用

LR(0)

- 使用两个栈(文法符号栈和状态栈), 一个分析表
- 我们需要根据不同文法构造ACTION-GOTO表
- 一个ACTION-GOTO表: (书p125) (书p136)

ACTION表中的第 $[S_i, a]$ (S_i 是状态, a 是输入串顶部终结符) 个格子可能有 :

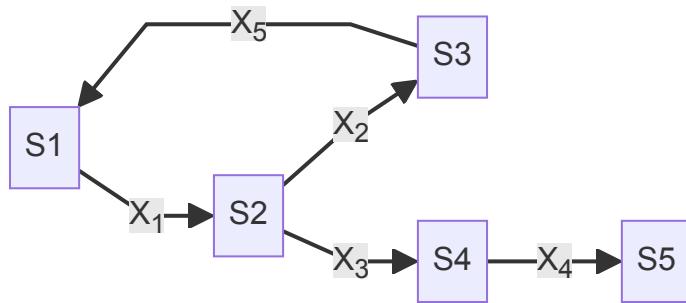
- S_j : 往状态栈压入状态 j
- r_j : 用第 j 个规则, 比如 $A \rightarrow \beta$, 归约符号栈最顶部, 同时状态栈弹出 $|\beta|$ 个状态
- 空的, 代表遇到错误

GOTO表中的第 $[S_i, A]$ (S_i 是状态, A 是符号栈顶部非终结符) 个格子可能有某个状态, 到了这个格子就往状态栈压入状态 i

先执行ACTION, 后执行GOTO

• 怎么构造ACTION-GOTO表?

- 文法拓广
 - 要使用LR分析, 我们需要先构造拓广文法.
 - 对于文法 $G[S]$, 加入规则 $S \rightarrow S'$, 得到拓广文法 $G[S']$, S' 为该文法的开始符.
 - 目的: 使构造出来的分析表只有一个接受状态, 即 $S' \rightarrow S \bullet$.
 - GOTO
 - GOTO表一般只用非终结符
 - 要构造GOTO表, 我们需要识别活前缀的DFA, DFA的状态转换矩阵就是GOTO表
 - 活前缀是输入串(规范句型)已分析过的、没有语法错误的前缀部分
- 例:
- 对于句型 $\alpha\beta t$, β 表示句柄, 如果 $\alpha\beta = u_1u_2\dots u_r$, 那么符号串 $u_1u_2\dots u_i$ ($1 \leq i \leq r$) 即是句型 $\alpha\beta t$ 的活前缀
- 这个DFA大概长这样:



- 其中 S_i 是状态, X_i 是非终结符.
- 为什么 GOTO 表不考虑终结符呢?
 - 因为要节省空间, 干脆放到 ACTION 表里面了

◦ ACTION

- ACTION 表一般只用终结符
- 一共有移进和归约, 接受和出错四种动作
- 当遇到 $ACTION[S_i, a]$ 时:
 - 移进 ($ACTION[S_i, a] = S_j$):
将 a 推进栈并设置新的栈顶状态 S_j
 - 归约 ($ACTION[S_i, a] = r_c$):
使用第 c 个文法规则 (假设是 $A \rightarrow \beta$) 归约符号栈最顶部的 β (弹出 β , 把 A 推进栈, 同时弹出和 β 相同长度的状态),
并设置新的栈顶状态 $S_j = GOTO[S_i - |\beta|, A]$
 - 接受 ($ACTION[S_i, a] = \text{accept}$)
 - 出错 ($ACTION[S_i, a]$ 为空白)

◦ 构造方法

- 首先求出每个表达式的项目

举例:

表达式: $S \rightarrow aAcBe$

它的项目有:

- $S \rightarrow \bullet aAcBe$
- $S \rightarrow a \bullet AcBe$
- $S \rightarrow aA \bullet cBe$
- $S \rightarrow aAc \bullet Be$
- $S \rightarrow aAcB \bullet e$
- $S \rightarrow aAcBe \bullet$

项目是分析过程中的某一状态, 用 \bullet 代表该状态, 它分割开了已经归约的部分和等待归约部分,

其中 \bullet 前已经被归约, \bullet 后等待归约

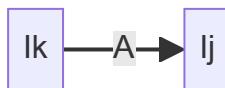
- 朴素的方法: 根据项目写出 NFA, 然后确定化为 DFA
 - 规定项目 $S \rightarrow \bullet aAcBe$ 为 NFA 的唯一初态
 - 任何项目均认为是 NFA 的终态(活前缀识别态)

- 对于状态 i (对应 $S \rightarrow a \bullet A \beta$) 和 j (对应 $S \rightarrow a A \bullet c \beta$) , 画一条从 $i \rightarrow j$ 的弧, 标志为 A .
- 更快的方法: 将有关项目组成项目集, 所有项目集构成的DFA即为LR(0)
 - 求出所有的项目集闭包closure (书p132)
 - closure是一个项目对应的集合, 包含该项目以及其状态后面可能面对的所有项目
 - 对于 $I: A \rightarrow a \bullet B \beta$, 将所有 $B \rightarrow \cdot r$ 加入closure(I)中
 - 例: $G'[E']$
 - $E' \rightarrow E$
 - $E \rightarrow E + T \mid T$
 - $T \rightarrow T^* F \mid F$
 - $F \rightarrow (E) \mid i$

令 $I = \{E' \rightarrow \cdot E\}$

$\text{closure}(I) = \{E' \rightarrow \bullet E, E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T^* F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet i\}$

- 这个过程正是直接构造化简后的DFA的过程.
- DFA的每个节点就是一个closure, 对于closure内的所有项目(状态), 它们在"下一个字符未知"的情况下是完全一样的. (LR(0)不向右观察下一个字符)
- 书上的转换函数GO(有时称其为GOTO)其实就是DFA的有向边.
- 通过构造closure, 我们将所有项目分组打包, 减少了节点数量, 正如之前化简DFA的时候. 我们将这个分组称作**规范项目族**.
- 包含 $S' \rightarrow \bullet S$ 的closure是DFA的初始状态.
- 一个完整的用项目集构造的DFA见书p133.
- 得到了识别活前缀的DFA后, 我们构造ACTION和GOTO表. (书p135)
 - 我们用 I_i 表示closure.
 - 对项目 $A \rightarrow a \bullet a \beta \in I_k$, 若在DFA中有 $I_k \rightarrow I_j$, 则令 $\text{ACTION}[k, a] = S_j$. (终结符转换)
 - 对项目 $A \rightarrow a \bullet \in$ 第 j 个产生式, 对任何终结符 a (包括#), 置 $\text{ACTION}[k, a]$ 为 r_j . (归约)
 - 在DFA中若有如下转换则使 $\text{GOTO}[k, A] = j$ (非终结符转换)



- 若 $S' \rightarrow S \bullet \in I_k$, 令 $\text{ACTION}[k, \#] = \text{acc}$ (接受)
- 若表每个格子元素唯一, 称该文法为LR(0)文法.

SLR(1) (书p138) (可以看作LR(0)的升级)

- 大多数2型文法不是LR(0), 即表某个格子元素不唯一, 故我们设计一种文法, 允许冲突的状态存在, 并向前看一个符号处理冲突, 这就是SLR(1).
- SLR(1)文法属于LR(1)文法.
- 有的冲突可以解决, 有的冲突不能, 毕竟SLR(1)文法只是2型文法的一个子集
- 如何处理冲突:
 - 可以被解决的冲突有: 移进-归约冲突, 归约-归约冲突

- 下面以同时解决两个冲突为例, 了解原理即可:
 - 该closure为 I , $I = \{X->...●a..., A->α●, B->β●\}$, 此时, 不知道是该等待下一个终结符 a 然后移进, 还是归约 A , 还是归约 B
 - 条件:
 - $\text{FOLLOW}(A) \cap \text{FOLLOW}(B) = \emptyset$
 - $\text{FOLLOW}(A) \cap \{a\} = \emptyset$
 - $\text{FOLLOW}(B) \cap \{a\} = \emptyset$
 - 即所有用来判断动作的终结符不能有交集
 - 假设下一个终结符为 x
 - $x=a$, 移进
 - $x \in \text{FOLLOW}(A)$, 归约 A
 - $x \in \text{FOLLOW}(B)$, 归约 B
- 总而言之, 我们可以沿用上面的ACTION-GOTO表, 只需要修改一个规则:
 - "对项目 $A->\alpha● \in$ 第 j 个产生式, 对任何终结符 a (包括#), 置 $\text{ACTION}[k,a]$ 为 r_j " 太粗暴了
 - 改为 "对项目 $A->\alpha● \in$ 第 j 个产生式, 对任何终结符 $a \in \text{FOLLOW}(A)$ (包括#), 置 $\text{ACTION}[k,a]$ 为 r_j "
 - 若表每个格子元素唯一, 称该文法为SLR(1)文法.

LR(1) (可以看作SLR(1)的升级)

- SLR(1) 有一个问题, 就是对于 $A->\alpha$, 下一个输入符号 $b \in \text{FOLLOW}(A)$ 只是归约 A 的必要条件而不是充分条件

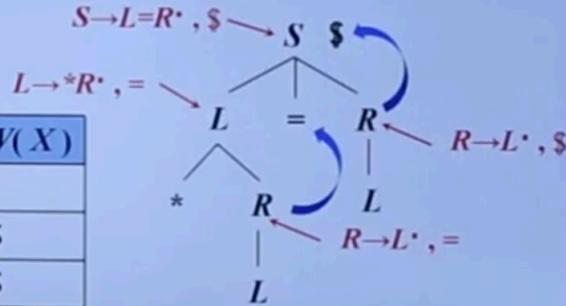
- LR(1) 对归约的条件作出更为严格的限制.

► LR(1)分析法的提出

➤ 对于产生式 $A \rightarrow \alpha$ 的归约, 在不同使用位置,
 A 会要求不同的后继符号

- | |
|------------------------------|
| 0) $S' \rightarrow S$ |
| 1) $S \rightarrow L = R$ |
| 2) $S \rightarrow R$ |
| 3) $L \rightarrow *R^*$ |
| 4) $L \rightarrow \text{id}$ |
| 5) $R \rightarrow L$ |

X	FOLLOW(X)
S	\$
L	=, \$
R	=, \$



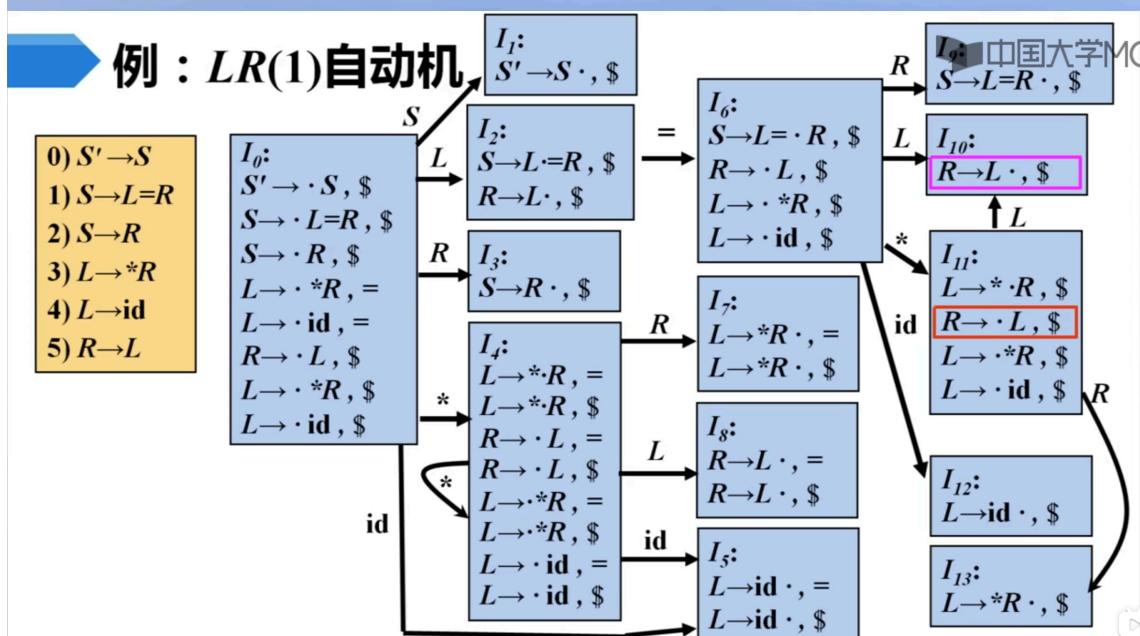
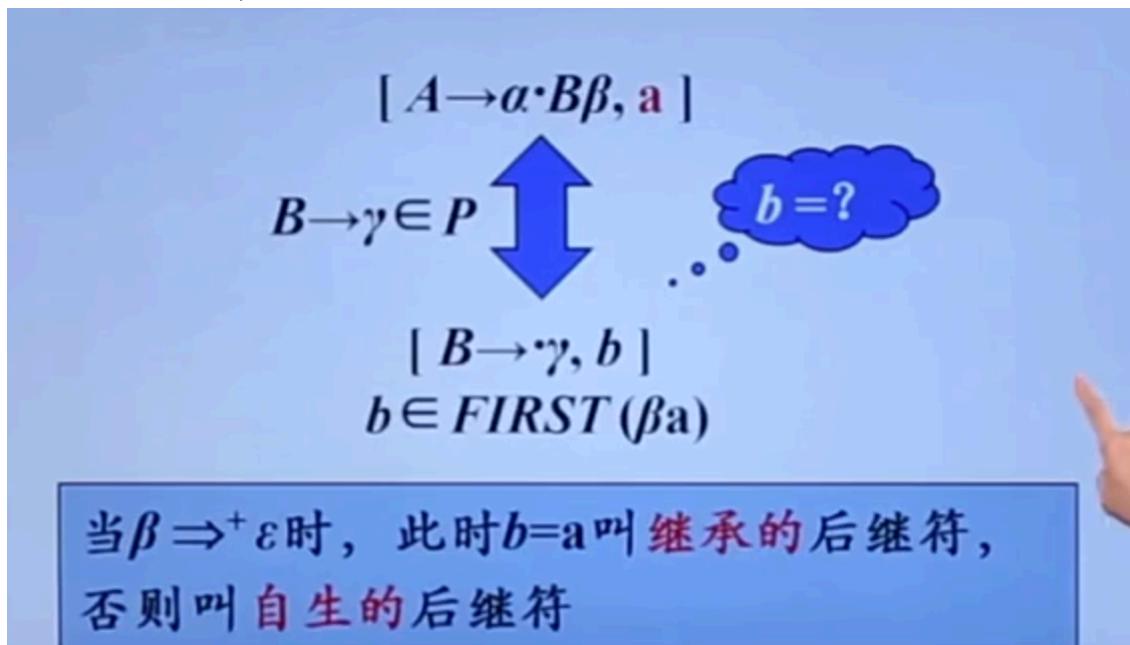
➤ 在特定位置, A 的后继符集合是 $FOLLOW(A)$ 的子集

► 规范LR(1)项目

中国大学MOOC

- 将一般形式为 $[A \rightarrow \alpha\cdot\beta, a]$ 的项称为 **LR(1) 项**, 其中 $A \rightarrow \alpha\beta$ 是一个产生式, a 是一个终结符(这里将\$视为一个特殊的终结符)它表示在当前状态下, A 后面必须紧跟终结符, 称为该项的**展望符(lookahead)**
- LR(1) 中的1指的是项的第二个分量的长度
- 在形如 $[A \rightarrow \alpha\cdot\beta, a]$ 且 $\beta \neq \epsilon$ 的项中, 展望符 a 没有任何作用
- 但是一个形如 $[A \rightarrow \alpha\cdot, a]$ 的项在只有在下一个输入符号等于 a 时才可以按照 $A \rightarrow \alpha$ 进行归约
- 这样的 a 的集合总是 $FOLLOW(A)$ 的子集, 而且它通常是一个真子集

- 看这张图。虽然我们说“ $A \rightarrow a \cdot B\beta, a$ ”里面a没有任何作用，但在递推的过程中，我们将得到 $A \rightarrow aB\bullet$ 的展望符是a，这就约束了归约条件。



LR(1)项目集闭包的计算

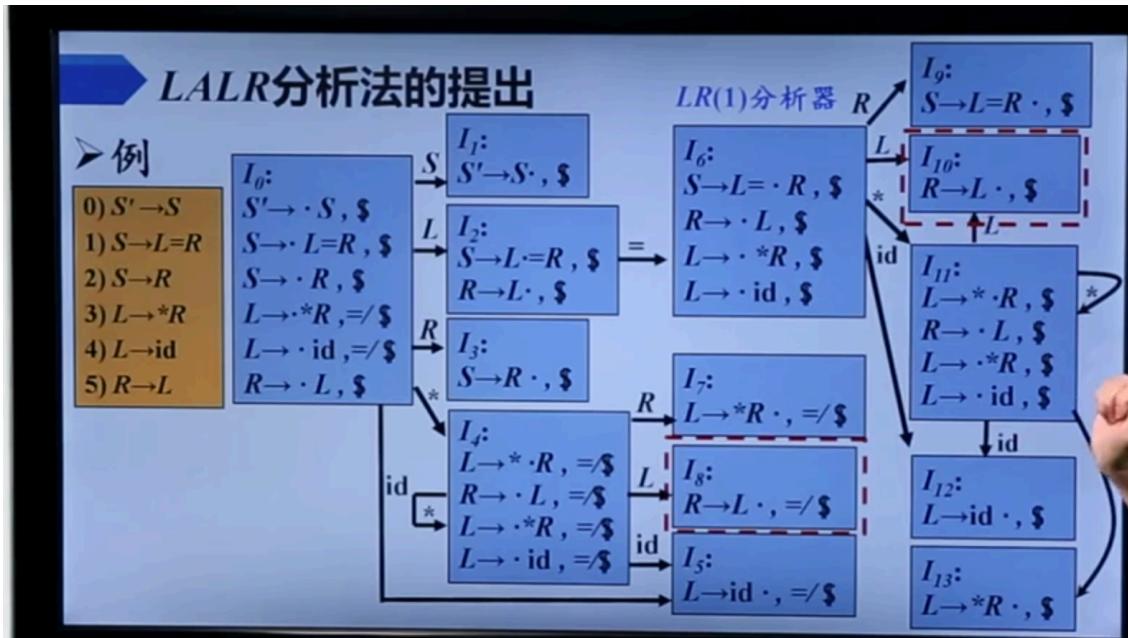
$CLOSURE(I) = I \cup \{ [B \rightarrow \cdot \gamma, b] \mid [A \rightarrow a \cdot B\beta, a] \in CLOSURE(I), B \rightarrow \gamma \in P, b \in FIRST(\beta a)\}$

```

SetOfItems CLOSURE (I) {
  repeat
    for (I中的每个项[A → a·B\beta, a])
      for (G'的每个产生式B → \gamma)
        for (FIRST(\beta a)中的每个符号b)
          将[B → \cdot \gamma, b]加入到集合I中;
  until 不能向I中加入更多的项;
  until I;
}
  
```

LALR(1) (可以看作LR(1)的升级)

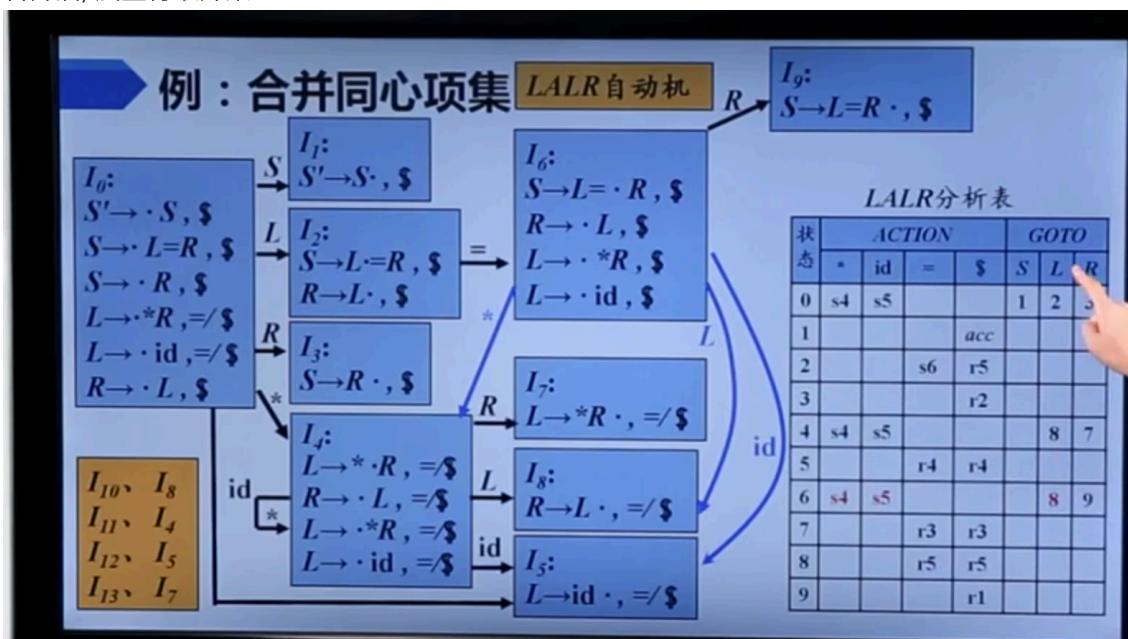
- 下图是一个LR(1)的DFA:



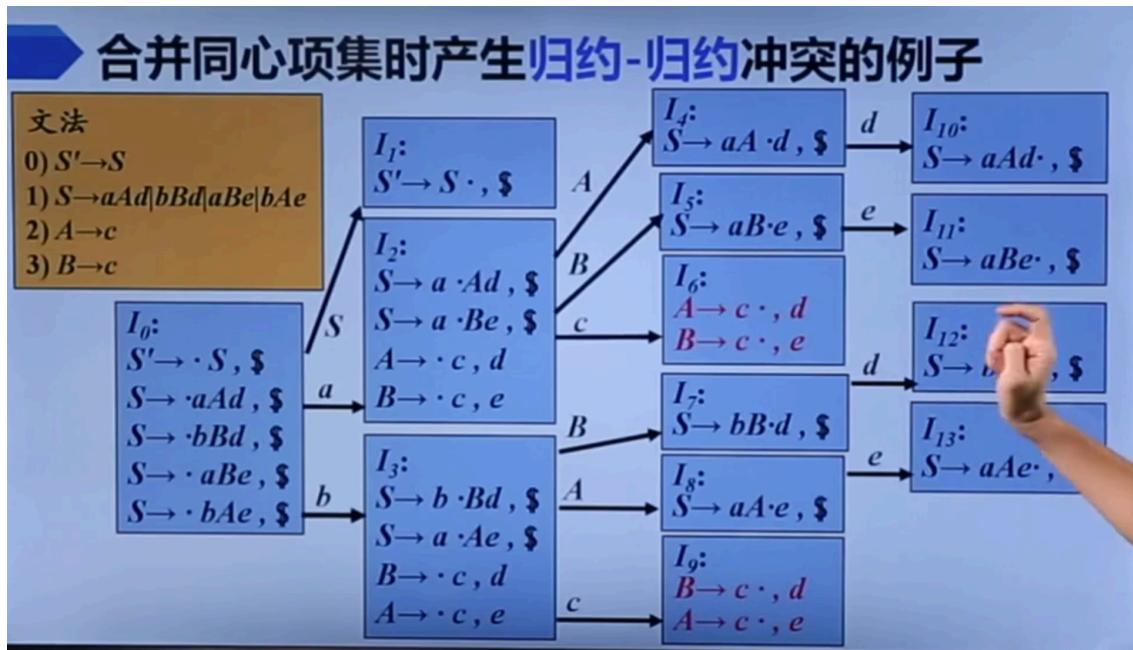
- LALR就是将其化简, 即合并除了展望符都相同的项目集, 以节省空间.

这张图中, 我们将合并(8,10), (7,13), (4,11), (5,12).

- 合并后, 展望符取并集.



- 合并可能会产生归约-归约冲突, 如下图的I₆和I₉:



- 不会产生移入-归约冲突, 因为展望符对移入不起作用, 而合并实际上是合并展望符
- 合并还可能延缓错误的发现.
- 分析能力SLR(1)<LALR(1)<LR(1)

语法制导 (2型文法)

- 简介



- SDD是一种特殊的,和文法同级的概念,可以认为它对文法的"规则"(产生式)作了拓展。它的规则基于2型文法,为文法符号关联有特定意义的属性,为产生式关联相应的对属性的操作,以扩充2型文法,使其可以上下文有关。

两个概念

- 将语义规则同语法规则(产生式)联系起来要涉及两个概念
- 语法制导定义(Syntax-Directed Definitions, SDD)
- 语法制导翻译方案(Syntax-Directed Translation Scheme, SDT)

- 下图中CFG是指上下文无关文法,即2型文法

语法制导定义(SDD)

SDD是对CFG的推广

- 将每个文法符号和一个语义属性集合相关联
- 将每个产生式和一组语义规则相关联,这些规则用于计算该产生式中各文法符号的属性值

例	产生式	语义规则
	$D \rightarrow T\ L$	$L.\ inh = T.\ type$
	$T \rightarrow \text{int}$	$T.\ type = \text{int}$
	$T \rightarrow \text{real}$	$T.\ type = \text{real}$
	$L \rightarrow L_1, \text{id}$	$L_1.\ inh = L.\ inh$

语法制导翻译方案(SDT)

- SDT是在产生式右部嵌入了程序片段的CFG,这些程序片段称为语义动作。按照惯例,语义动作放在花括号内

例	$D \rightarrow T \{ L.inh = T.type \} L$
	$T \rightarrow \text{int} \{ T.type = \text{int} \}$
	$T \rightarrow \text{real} \{ T.type = \text{real} \}$
	$L \rightarrow \{ L_1.inh = L.inh \} L_1, \text{id}$

- SDT的包含的信息和SDD大体相同，但多出了“操作属性的时间”这个信息。

SDD与SDT

➤ SDD

- 是关于语言翻译的高层次规格说明
- 隐蔽了许多具体实现细节，使用户不必显式地说明翻译发生的顺序

➤ SDT

- 可以看作是对SDD的一种补充，是SDD的具体实施方案
- 显式地指明了语义规则的计算顺序，以便说明某些实现细节

➤ 将S-SDD转换为SDT

- 将一个S-SDD转换为SDT的方法：将每个语义动作都放在产生式的最后

➤ 例

S-SDD

产生式	语义规则
(1) $L \rightarrow E \text{ n}$	$L.val = E.val$
(2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
(3) $E \rightarrow T$	$E.val = T.val$
(4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
(5) $T \rightarrow F$	$T.val = F.val$
(6) $F \rightarrow (E)$	$F.val = E.val$
(7) $F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

SDT

(1) $L \rightarrow E \text{ n} \{ L.val = E.val \}$
(2) $E \rightarrow E_1 + T \{ E.val = E_1.val + T.val \}$
(3) $E \rightarrow T \{ E.val = T.val \}$
(4) $T \rightarrow T_1 * F \{ T.val = T_1.val \times F.val \}$
(5) $T \rightarrow F \{ T.val = F.val \}$
(6) $F \rightarrow (E) \{ F.val = E.val \}$
(7) $F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

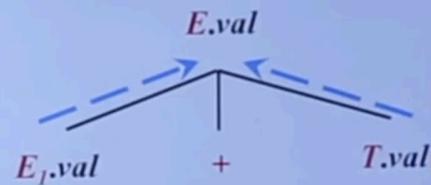
- 属性分为综合属性和继承属性

▶ 综合属性 (*synthesized attribute*)

在分析树结点 N 上的非终结符 A 的综合属性只能通过 N 的子结点或 N 本身的属性值来定义

例

产生式	语义规则
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$



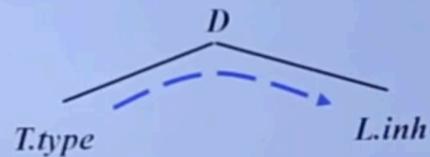
终结符可以具有综合属性。终结符的综合属性值是由词法分析器提供的词法值，因此在 SDD 中没有计算终结符属性值的语义规则

▶ 继承属性 (*inherited attribute*)

在分析树结点 N 上的非终结符 A 的继承属性只能通过 N 的父结点、 N 的兄弟结点或 N 本身的属性值来定义

例

产生式	语义规则
$D \rightarrow T L$	$L.inh = T.type$



终结符没有继承属性。终结符从词法分析器处获得的属性值被归为综合属性值

- 属性的计算有其顺序

SDD的求值顺序

- SDD为CFG中的文法符号设置语义属性。对于给定的输入串 x , 应用语义规则计算分析树中各结点对应的属性值
- 按照什么顺序计算属性值?
 - 语义规则建立了属性之间的依赖关系, 在对语法分析树节点的一个属性求值之前, 必须首先求出这个属性值所依赖的所有属性值



依赖图 (Dependency Graph)

- 依赖图是一个描述了分析树中结点属性间依赖关系的有向图
- 分析树中每个标号为 X 的结点的每个属性 a 都对应着依赖图中的一个结点
- 如果属性 $X.a$ 的值依赖于属性 $Y.b$ 的值, 则依赖图中有一条从 $Y.b$ 的结点指向 $X.a$ 的结点的有向边



- 属性值的计算顺序即是有向图的拓扑排序.

- 对于只具有综合属性的SDD, 可以按照任何自底向上的顺序计算它们的值
- 对于同时具有继承属性和综合属性的SDD, 不能保证存在一个顺序来对各个节点上的属性进行求值



- 如果依赖图有圈则无法计算顺序. (见书p164)
- S属性文法和L属性文法 (都属于SDD)
 - 这两个文法的依赖图不会出现回路

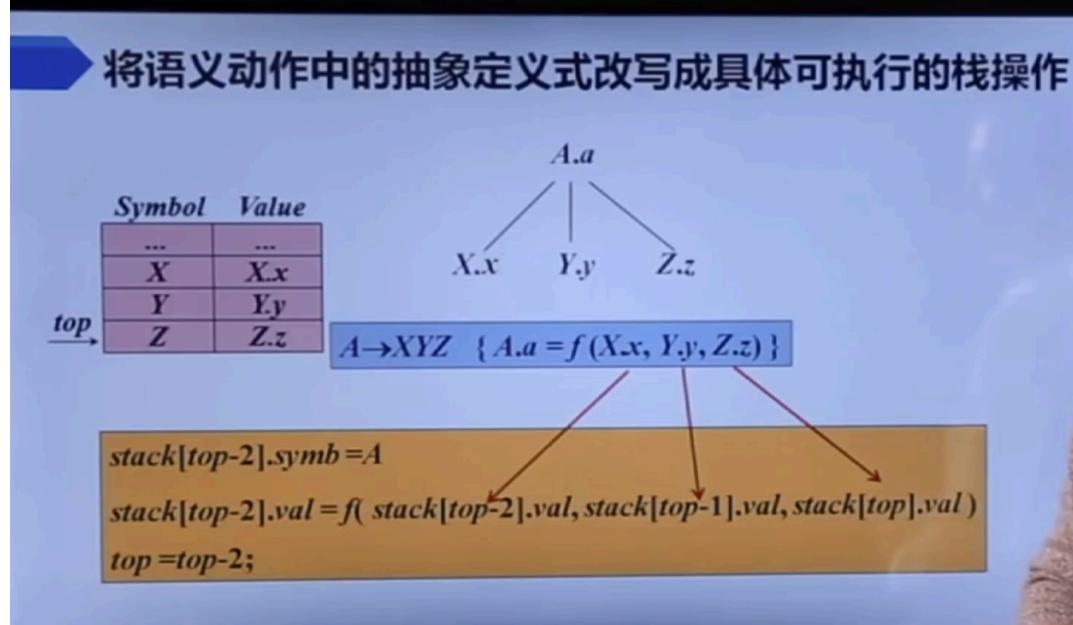
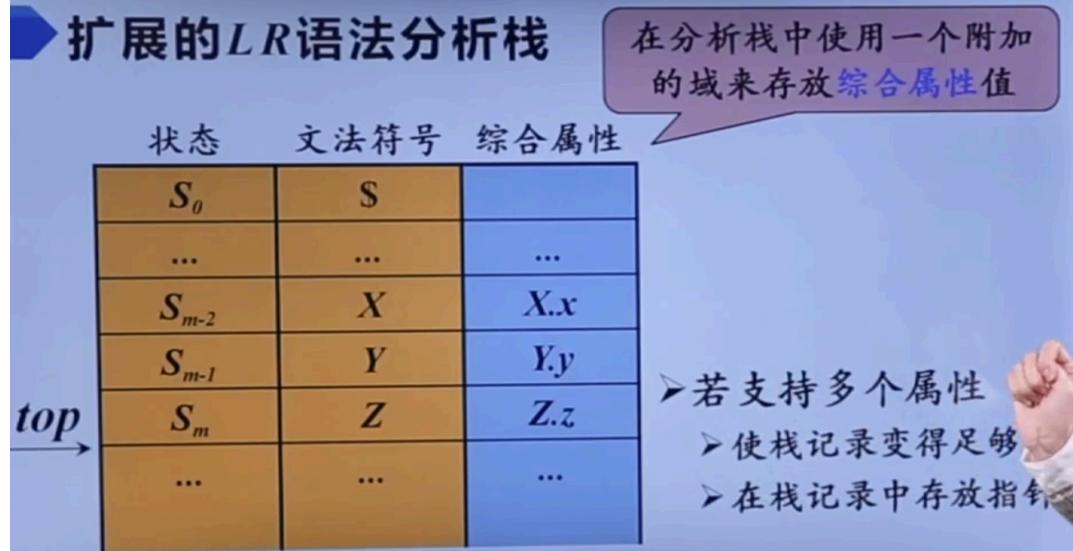
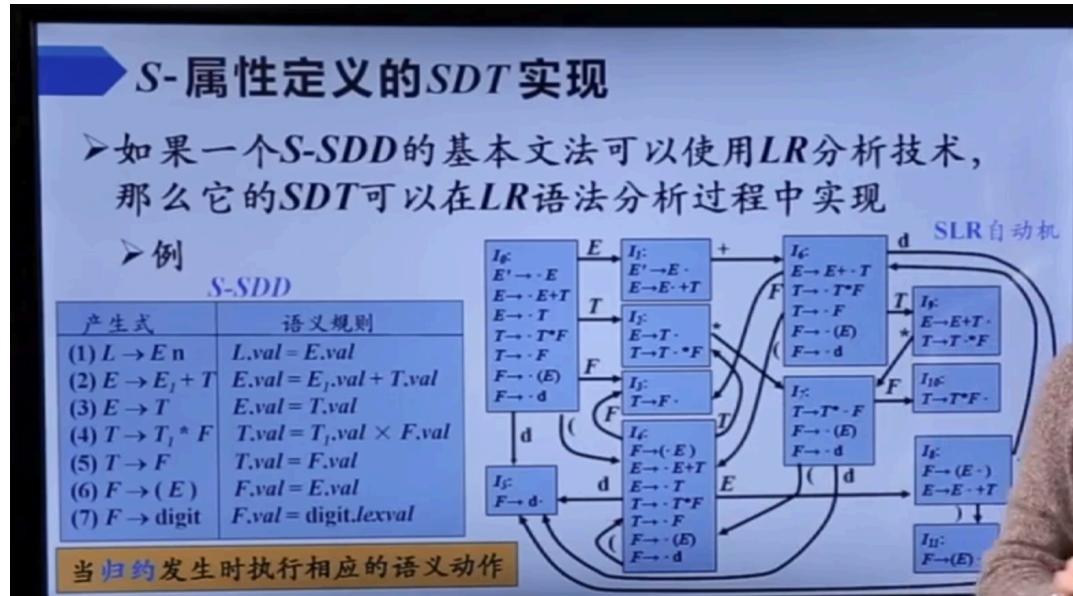
- 只有综合属性的文法称为S属性文法
- L属性文法定义见书p166, 依赖图的边可以从左到右, 但不能从右到左.
- 判断是否为L属性文法: 首先区分综合属性和继承属性, 被计算的产生式左部符号的属性是综合属性.

例 : *L-SDD*

The diagram illustrates the derivation of an LR(0) item set for an L-attributed grammar. It shows four items (1) through (4) with their corresponding LR(0) items and associated semantic rules. Item (1) has a red dashed arrow labeled '继承属性' (Inheritance Attribute) pointing to its LR(0) item. Item (4) has a blue dashed arrow labeled '综合属性' (Synthetic Attribute) pointing to its LR(0) item. A hand is pointing to the inheritance rule $T'.syn = T'_1.syn$.

	产生式	语义规则
(1)	$T \rightarrow F T'$	$\boxed{T'.inh} = F.val$
		$\boxed{T.val} = T'.syn$
(2)	$T' \rightarrow * F T'_1$	$\boxed{T'_1.inh} = T'.inh \times F.val$
		$\boxed{T'.syn} = T'_1.syn$
(3)	$T' \rightarrow \epsilon$	$\boxed{T'.syn} = T'.inh$
(4)	$F \rightarrow \text{digit}$	$\boxed{F.val} = \text{digit.lexval}$

- S属性定义的SDT实现



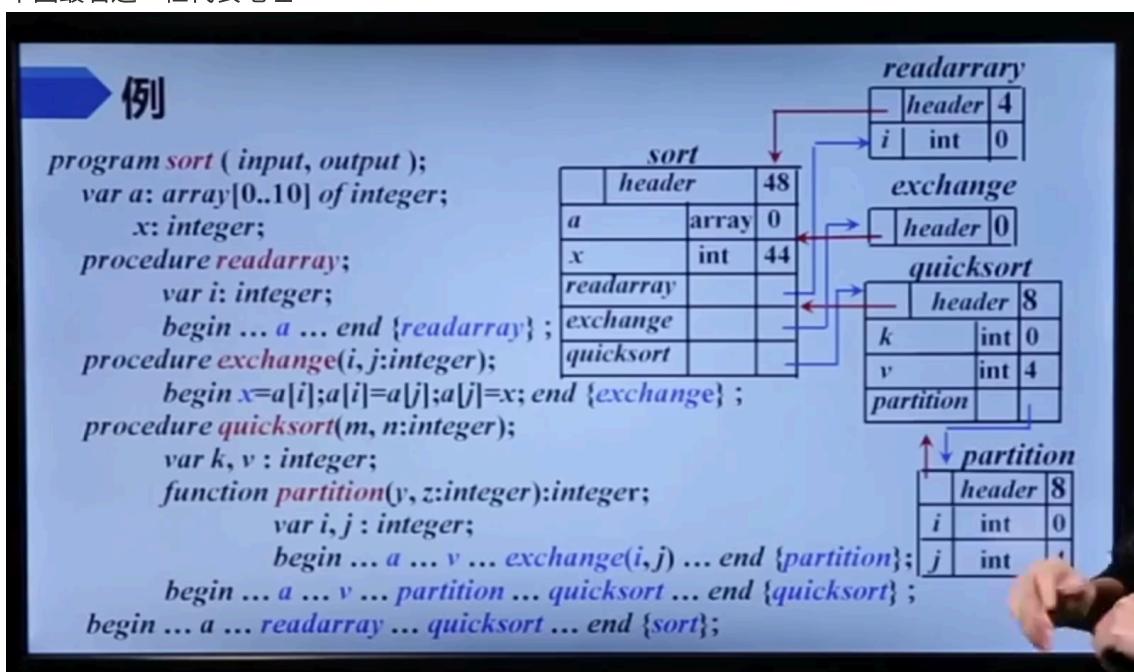
例：在自底向上语法分析栈中实现桌面计算器

产生式	语义动作	
(1) $E' \rightarrow E$	$\text{print}(E.\text{val})$	{ $\text{print}(\text{stack}[\text{top}].\text{val});$ }
(2) $E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$	{ $\text{stack}[\text{top}-2].\text{val} = \text{stack}[\text{top}-2].\text{val} + \text{stack}[\text{top}].\text{val};$ $\text{top}=\text{top}-2;$ }
(3) $E \rightarrow T$	$E.\text{val} = T.\text{val}$	{ $\text{stack}[\text{top}-2].\text{val} = \text{stack}[\text{top}-2].\text{val} \times \text{stack}[\text{top}].\text{val};$ $\text{top}=\text{top}-2;$ }
(4) $T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} \times F.\text{val}$	{ $\text{stack}[\text{top}-2].\text{val} = \text{stack}[\text{top}-2].\text{val} \times \text{stack}[\text{top}].\text{val};$ $\text{top}=\text{top}-2;$ }
(5) $T \rightarrow F$	$T.\text{val} = F.\text{val}$	{ $\text{stack}[\text{top}-2].\text{val} = \text{stack}[\text{top}-1].\text{val};$ $\text{top}=\text{top}-2;$ }
(6) $F \rightarrow (E)$	$F.\text{val} = E.\text{val}$	
(7) $F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{lexval}$	

语义分析和中间代码生成

符号表管理

- 为每个作用域建立一个独立的符号表
- 下图最右边一栏代表地址

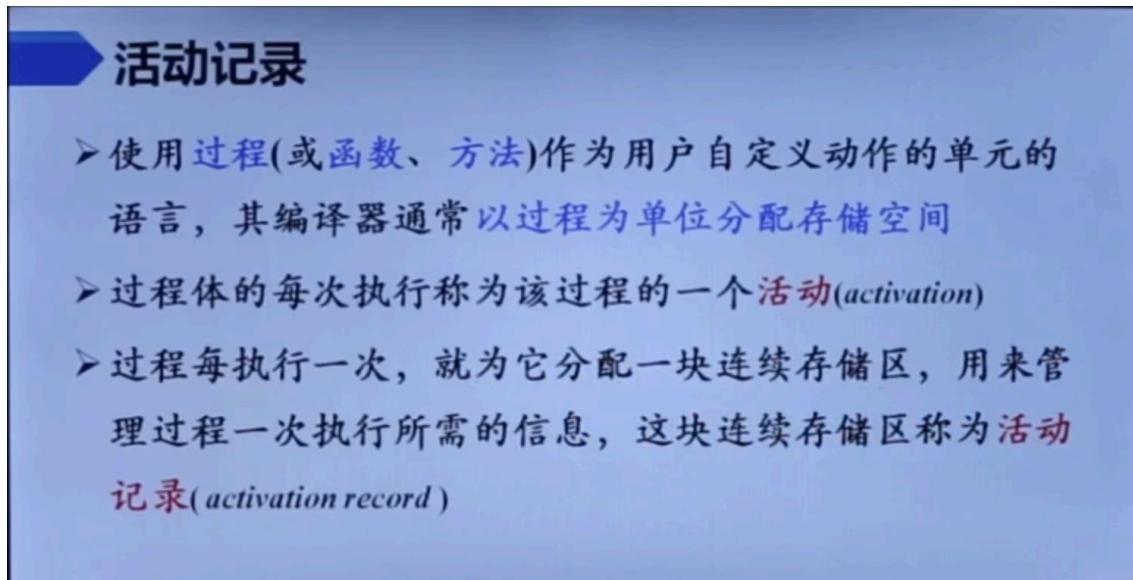


运行时存储

- 运行时内存的划分



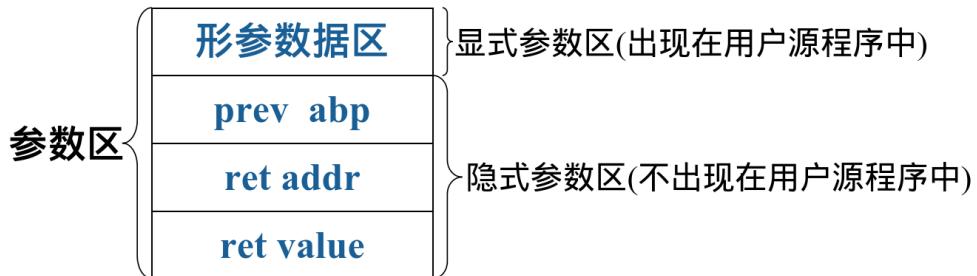
- 什么是活动记录



- 一个典型的活动记录可以分为三部分：
 - 局部数据区：存放模块中定义的各个局部变量

- 参数区

(2) 参数区：存放隐式参数和显式参数。



prev abp：存放调用模块记录的基地址，函数执行完时，释放其数据区，数据区指针指向调用前的位置

ret addr：返回地址，即调用语句的下一条执行指令地址

ret value：函数返回值（无值则空）

形参数据区：每一形参都要分配数据空间，形参单元中存放实参值或者实参地址

- display区：存放各外层模块活动记录的基地址
- 静态存储

静态存储分配

- 在静态存储分配中，编译器为每个过程确定其活动记录在目标程序中的位置
- 这样，过程中每个名字的存储位置就确定了
- 因此，这些名字的存储地址可以被编译到目标代码中
- 过程每次执行时，它的名字都绑定到同样的存储单元

静态存储分配的限制条件

- 适合静态存储分配的语言必须满足以下条件
- 数组上下界必须是常数
- 不允许过程的递归调用
- 不允许动态建立数据实体

代码优化

- 划分基本块
- 构造流图