

1

口 语法分析器

通用的 (Cocke-Younger-Kasami 算法、Earley 算法)
 { 自顶向下的
 自底向上的

口

上下文无关文法

{ 终结符号
 非终结符号
 - 开始符号
 - 一组产生式

$s \Rightarrow^* \alpha$, s 为文法的开始符号, α 是经此产生式零次或多次产生, 及该文法的一个句型
 (句型即可包括终结符, 又可包括非终结符)
 句子是特殊的句型, 只能包含终结符

最左推导 :: 总是选择每个句型的最左非终结符

最右推导 :: 总是选择每个句型的最右非终结符

口 上下文无关文法的表达能力

• 强于正则表达式

按右侧流程图将正则表达式对

应的NFA 转成文法

1) 为每个状态, 创建 对应非终结点:

 $i \rightarrow A_i$ 3) 若为接受状态, 则得 $A_i \rightarrow \epsilon$ • 文法可对 '两个个体进行计数' 正则表达式 4) 若为开始状态, 则 A_0 为开始符 $a^n b^n$ $S \rightarrow A$ aAb $A \rightarrow aAb | \epsilon$ $aaAbb$ \downarrow $aaaAbbb$

口 二义性 → 由文法可得到给定串的多棵语法分析树.

消除二义性

{ 制定规则

(如, 每个 else 和最近的尚未匹配的 then 匹配)

{ 改写成无二义性的文法,

② 左递归消除及提取左公因子

□ 左递归消除

- 立即左递归消除 $\Rightarrow A \rightarrow A\alpha$ 改写为 $A \rightarrow \beta A'$
 $A \rightarrow \beta$ $A' \rightarrow \alpha A' | \epsilon$

- 消除左递归 (没有环或 ϵ 产生的文法)

1) 按一定顺序把非终结符号排列

2) for (每个非终结符号 A_i [按序]) {

$\text{for } (A_j \leftarrow \{A_0, A_1, \dots, A_{i-1}\}) \{$

• 若存在 $A_i \rightarrow A_j r$ 的产生式替换为产生式组

$A_i \rightarrow S_1 r | S_2 r | \dots | S_n r$, 其中 $A_j \rightarrow S_1 | S_2 | \dots | S_n$

是所有的 A_j 产生式

}

消除 A_i 产生式之间的立即左递归

}

在外循环第*i*次迭代完成后, $\{A_0, \dots, A_i\}$ 的所有产生式的头部不以 A_0, \dots, A_i 开头, 故能保证这些产生式无左递归

□ 提取左公因子 \longrightarrow 可用于改写文法

$\text{stmt} \rightarrow \text{if expr then stmt else stmt}$

- 对每个非终结符 A , 找出它的两个或多个选

$| \text{if expr then stmt}$

项之间的最长公共前缀 α , 然后转换

$\text{stmt} \rightarrow \text{if expr then stmt } A'$

- 不断迭代上述步骤直至任意两个产生式无

每个非终结符

公共前缀

$A' \rightarrow \text{else stmt } | \epsilon$

③ 自顶向下的语法分析

□ 递归下降的语法分析

• 每个非终结字符对应一个函数，函数中对终结字符执行匹配
对非终结字符调用对应函数

- 对整段程序的分析，从用开始字符对应的函数开始
- 在选择非终结字符时，要么根据向前看字符决定，要么按照某个顺序逐个尝试这些产生式。若按后者，尝试意味着失败，因而需回溯

□ First & Follow

First(α)：文法符号 α 可推出串的首个字符所组成的集合

Follow(A)：某些句型中紧跟在 A 右边的终结符号的集合

$\left\{ \begin{array}{l} \text{若 } \alpha \text{ 为一个终结符号} \\ \text{则 } \text{First}(\alpha) = \alpha \end{array} \right.$

$\left\{ \begin{array}{l} \text{若 } \alpha \text{ 为非终结符号} \\ \alpha \rightarrow \beta_1 \beta_2 \dots \beta_n \Rightarrow \text{若有 } \epsilon \text{ 在所有 } \text{First}(\beta_1) \dots \text{First}(\beta_n) \\ \text{中，就把 } \text{First}(\beta_{n+1}) \text{ 放入 } \text{First}(\alpha) \text{ 中} \end{array} \right.$

$\left\{ \begin{array}{l} S \text{ 为开始符号，则 } \$ \text{ 在 } \text{Follow}(S) \text{ 中} \end{array} \right.$

$\left\{ \begin{array}{l} A \rightarrow \alpha B \beta, \text{ 则 } \text{First}(\beta) \text{ 中除 } \epsilon \text{ 外的所有符号都在 } \text{Follow}(B) \text{ 中} \end{array} \right.$

$\left\{ \begin{array}{l} A \rightarrow \alpha B \text{ 或 } (A \rightarrow \alpha B \beta \text{ 且 } \text{First}(\beta) \text{ 包含 } \epsilon), \text{ 则 } \text{Follow}(A) \text{ 中所有符} \\ \text{号都在 } \text{Follow}(B) \text{ 中} \end{array} \right.$

□ LL(1) 文法

$\left\{ \begin{array}{l} \text{从左至右扫描} \\ \text{最左推导} \end{array} \right.$

| 指向前看 1 步

\downarrow 当 ϵ 在 $\text{First}(\beta)$
“ $\in \text{First}(\beta_n)$ ”
才把 ϵ 加入到 $\text{First}(\alpha)$ ；
 $X \rightarrow \epsilon$, 则将 ϵ 加入到 $\text{First}(X)$

解决‘不知选择’判定非

⇒ 处理待定的哪一产生式

$$A \rightarrow \alpha | \beta \quad \left\{ \begin{array}{l} \text{First}(\alpha) \text{ 与 } \text{First}(\beta) \text{ 不存在交集} \\ \text{若 } \Sigma \in \text{First}(\alpha), \text{ 则需 } \text{Follow}(A) \text{ 与 } \text{First}(\beta) \text{ 不存在交集} \end{array} \right.$$

↓
LL(1)文法

□ LL(1) 语法分析

1) 先构建预测分析表

- 表为二维，一维作为非终结符描述符，另一维作为输入符号；表中元素为产生式，即表时某非终结符下，输入符号 a 选择哪一产生式
- LL(1)文法使得表中元素至多是 1 个产生式
- 构表过程

对文法 G 的每个产生式 $A \rightarrow \alpha$ ，进行如下处理：

1) 对 $\text{First}(\alpha)$ 每个终结符 a ，将 $A \rightarrow \alpha$ 加入到 $M[A, a]$ 中

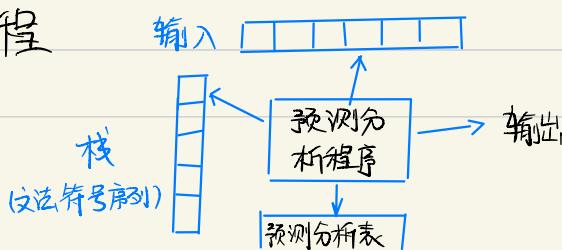
2) 若 $\Sigma \in \text{First}(\alpha)$ ，那么对 $\text{Follow}(A)$ 的每个终结符 b ，将
 $A \rightarrow \alpha$ 加入 $M[A, b]$

含 \$ 符号

注：按上填完预测分析表后，表中部分位置仍为空。
这些地方意味错误，可向其中填 error 或相应错误恢复机制。

另注：对 $A \rightarrow \Sigma$ 的产生式的动作，只前移文教符号，输入串上的指针不动

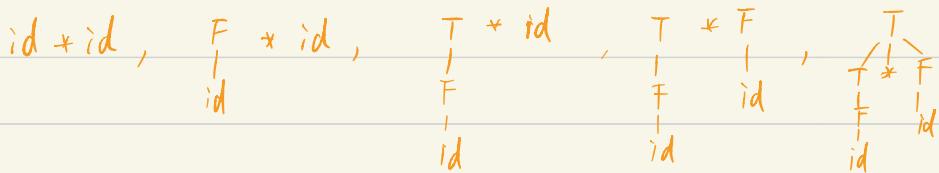
2) 非递归分析过程



④ 自底向上的语法分析

口 思想

- 语法分析即抽象语法树的构建，自底向上意味着先构叶子再一层层地往上。
- 归约推导的逆过程。推导是人文法到串的过程，而归约是串到文法的过程。
- 如对 $id * id$ 进行自底向上的语法分析过程



即, $\underbrace{id * id \Rightarrow F + id}_{\text{从中间看到“归约”}} \Rightarrow T * id \Rightarrow T * F \Rightarrow T \Rightarrow E$

因为每次归约右侧都是“终结符号”

口 过程 (移入-归约) 语法分析技术

栈(文法序列)

$\$$

 $\$ \$$

输入

$w \$$

 $\$$

初始状态,

结束状态, 表示

分析过程中,含4个动作

移入: 即把串从输入移到栈
归约: 将栈中句柄归约, 将归约得到的非
接受 (即分析成功)
报错

附: 句柄定义及位置 (见下页)

▲ 后续通过LR系列方法得出何时
移入, 何时归约以及句柄的位置

□ 句柄

- 句柄是能和某个产生式体匹配的子串，在语法分析过程中，归约即把句柄替换为相应产生的头部。
- 句柄的结尾处总是在栈顶 (证明从最左推导出发，列出所有情况)

总的来说，语法分析器在进行一次归约之后，都必须接着移入零个或多个符号才能在栈顶找到下一个句柄。

□ 分析过程存在的冲突

- { 移入 / 归约冲突 \rightarrow 即可移入又可归约
- 归约 / 归约冲突 \rightarrow 归约时不知道选哪一产生式

□ LR

- 目前最流行的自底向上的语法分析器都基于所谓的 $LR(k)$ 语法分析 $\Rightarrow L$ 从左向右扫描， R 最右推导， k 向前看 k 个输入符号
- LR 流行的原因

{ LR 文法是 LL 文法的真超集

$LR(k)$ 表明当我们在一个最右句型看到某个产生式的右部时，再向前看 k 个符号 就可以决定是否使用这个产生式进行归约；而对于

$LL(k)$ 文法，我们在决定是否使用某个产生式时，只能向前看该产生式右部推导出的 前 k 个符号 因最左推导

\rightarrow 只需可找到能推出这些的符号 \rightarrow 相当给定 非终结符串要求其必须产生的 (需该产生式能位于道产生式后的后边) $\xrightarrow{\text{相比约束小一点}}$ 推导出后续的 k 个符号

□ LR(0) 自动机 (即 SLR(1))

• 项

对于产生式 $A \rightarrow XY$, 可产生的项 $A \rightarrow \cdot XY$ 、 $A \rightarrow X \cdot Y$ 、 $A \rightarrow XY\cdot$ 。
可得到项 \cdot 给产生式加了点, 这些点分隔产生的右部表示哪些已归约得到哪些待归约得到。

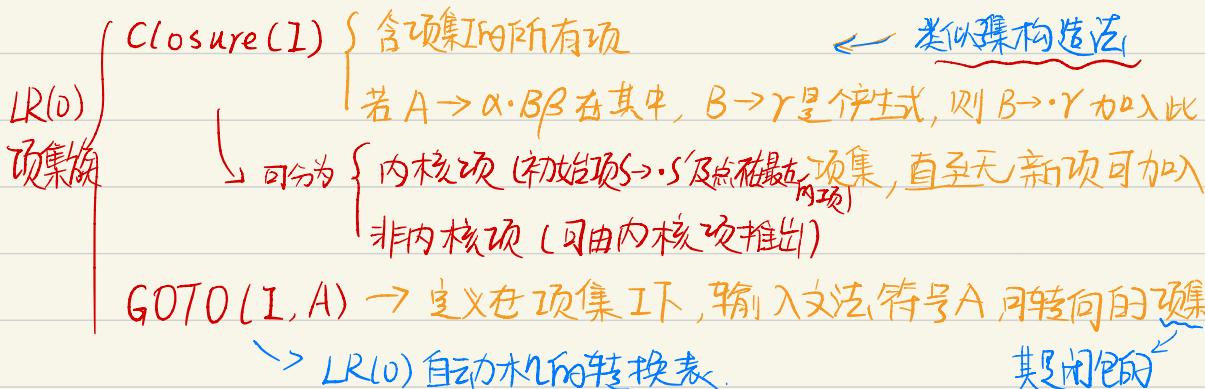
• LR(0) 自动机

先介绍长什么样; 再说明怎样用它做语法分析。最后‘证明’是什么给予它这样的作用

1) 长什么样

\rightarrow CLOSURE

LR(0) 作为一个 DFA, 它有状态 [项集], 还有边 [GOTO 函数]



LR(0) 项集族的构造算法,

$$C = \{ Closure(\{S' \rightarrow \cdot S\}) \};$$

repeat

for C 中每一项集

for (每个文法符号 X)

if ($GOTO(I, X)$ 非空且不在 C 中)

将 $GOTO(I, X)$ 加入 C 中;

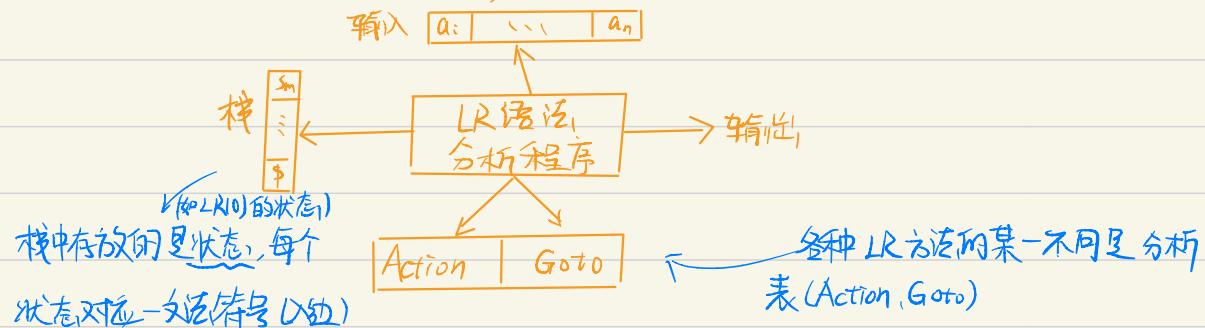
until 在某一轮中无新的项集被加入 C 中;

□

自动机

→ 怎样用 LR(0) 做语法分析 \Rightarrow SLR(1)

• LR 语法分析器通用模型



• SLR 的 Action, Goto 分析表

Action (状态, 终结字符) \rightarrow 给出在该状态下, 输入该终结字符要

进行的动作 (如移入、归约、接受、报错)

Goto (状态, 非终结字符) \rightarrow 给出归约后进入的状态

同时指出了移入后进入哪一状态,

同时指定了按照哪一产生式进行归约

• SLR 分析表与 LR(0) 自动机的关系

- 对于 LR(0) 中的 $GOTO(I, a)$ 即指定的文法符号是终结字符的直接构造时, 赋值给 $Action(i, a) \rightarrow$ 其中为项集 I 对应的状态
- 可通过选代项集做对每个项集做右侧分析
- 对于 LR(0) 中 $GOTO(I, A)$ 直接赋值给 $Action(i, A)$
- 若 $A \rightarrow \alpha \cdot$ 在 I 中, 那么对于 Follow(A) 中的所有 a, 将 $Action[i, a]$ 设置为‘归约 $A \rightarrow \alpha'$ (注, A 不等于 S')
- 若 $S' \rightarrow s \cdot$ 在 I 中, 则将 $Action[i, \$]$ 设为‘接受’
- SLR(1) 文法: 满足构成得的 SLR 表不存在冲突的文法

3) 为什么可用 LR(0) 自动机做移入-归约决定

- LR(0) 自动机的状态对应一顶集，也对应一文法符号（即入边对应）
- 语法分析通用模型中的栈记录了状态的变化，栈的符号中内容是当前状态序列，也就对应当前的文法符号路径，这一路径被称为“可行前缀”，它描述了目前归约到了哪一步
- 根据栈顶中的状态，我们可知它是可行前缀的后续（是移入还是归约等）如此能不断推进可行前缀。若最后可得到可行前缀 s' (为原始开始符号) 则表示分析成功
即 $s' \rightarrow s \cdot$ 的对应

原理的核心：如果我们在某个文法的 LR(0) 自动机中从初始状态开始沿着标号为某个可行前缀 s' 的路径到达一个状态，那么该状态对应的项集就是 s' 的有效项集（从而得知在某输入下执行怎样的动作）

从 LR(0) 自动机的角度看语法分析的整个过程，分析的成功意味着走到了 $s' \rightarrow s \cdot$ 项所在的项集对应的状态，分析的前进意味着状态的转换……

附：若一开始把项作为自动机的状态，那么将得到的是 NFA。用子集构造法，可把此 NFA 转为 DFA，会发现 DFA 等价 LR(0) 自动机。
回想 Closure(1) 的计算过程，它实际在求状态集合的 Σ 闭包。

□ 规范 LR(1)

- 目的：解决 SLR(1) 中的~~移入~~和~~归约~~的冲突，增强 SLR(1) 文法。

被称为LR(1)项

• 途径：为项添加一分量，如 $[A \rightarrow \alpha \cdot \beta, a]$ ， a 为添加的第二分量，它被称为向前看符号。当 $\beta \neq \varepsilon$ 时，第二分量无啥用；当 $\beta = \varepsilon$ 时，即 $[A \rightarrow \alpha \cdot, a]$ 只有当输入等于向前看符号 a 时才按照 $A \rightarrow \alpha \cdot$ 进行归约。

→ 加了此限制而在 SLR(1) 中只要输入在 $\text{Follow}(A)$ 中就进行归约，但这个 Follow 得到的集合是推导而来，而现在是归约，集合中的符号不能互适合。比如，
(考虑所有产生式)

$S \rightarrow L = R / R$	$\text{Follow}(R)$ 中包含 ' $=$ '，但对于项 $R \rightarrow L$ 遇
$L \rightarrow *R / id$	于 ' $=$ ' 时，不能把 $L =$ 归约为 $R =$ ，因为并 且 $R \rightarrow L$ 无 $R =$ 开头的产生式。

• 正式地讲，LR(1) 项 $[A \rightarrow \alpha \cdot \beta, a]$ 对一个可行前缀 r 有效的条件是存在 $S \xrightarrow{r} S \alpha w \xrightarrow{*} S \alpha \beta w$ ，其中 $r = S\alpha$ 且要么 a 是 w 的第一个符号，要么 w 为 ε 且 a 为 $\$$ 。

• LR(1) 项集的闭包运算

$[A \rightarrow \alpha \cdot \beta, a]$ 在 I 中，则对于 $[B \rightarrow \gamma, b]$ 其中 $b = \text{First}(\beta a)$ ，
(也在 I 中)

$\rightarrow S \xrightarrow{r} S \alpha w \xrightarrow{*} S \alpha \beta w$ (已知可行前缀为 $r = S\alpha$)，再利用 $B \rightarrow \gamma$ ，则 $S \xrightarrow{r} S \alpha B \beta w \xrightarrow{*} S \alpha \gamma \beta w$ 。而 $[B \rightarrow \gamma, \text{First}(\beta w)]$ 也是可行前缀 r 的有效项 (同一个可行前缀的有效项在同一子集中) 而 w 可表示为 αx ，所以 $\text{First}(\beta w) = \text{First}(\beta \alpha x) = \text{First}(\beta a)$ ，从而 $[B \rightarrow \gamma, \text{First}(\beta a)]$ 等价于 $[B \rightarrow \gamma, \text{First}(\beta w)]$

对于 $\text{First}(\beta a)$ 有两种情况

LR(1) 会用到
↓

$\left\{ \begin{array}{l} \beta \text{ 推出 } \varepsilon, \text{First}(\beta a) = a \rightarrow \text{认为该向前看符号是} \\ \text{传播得到的} \\ \beta \text{ 不为 } \varepsilon, \text{First}(\beta a) = \text{First}(\beta) \rightarrow \text{认为该向前看符号是生} \\ \text{成得到的} \end{array} \right.$

• LR(1) 项集的GOTO函数及 LR(1) 项集族的构建

$GOTO(I, X)$ {

将 I 初始化为空集；

for (I 中的每项 $[A \rightarrow \alpha \cdot A\beta, a]$)

将项 $[A \rightarrow \alpha \cdot X \cdot \beta, a]$ 加入集合 I 中；

return Closure(I);

}

$items(G')$ {

将 C 初始化为 $\{Closure([S' \rightarrow \cdot S, \$])\}$

repeat

for (C 中每个项集 I)

for (每行文法符号 X)

if ($GOTO(I, X)$ 非空且不在 C 中)

将 $GOTO(I, X)$ 加入 C 中；

until 不再有新项集加入到 C 中。

}

• LR(1) 语法分析表

→ 同 SLR(1)

1) 若项集 $[A \rightarrow \alpha \cdot A\beta, b]$ 在 I_i 中，并且 $GOTO(I_i, a) = I_j$ ，则将 Action $[i, a]$

设为“移入 j ”。 a 为一个终结符号。

2) 若 $[A \rightarrow \alpha \cdot, a]$ 在 I_i 中且 $A \neq S'$ ，那么将 Action $[i, a]$ 设为“由 $(A \rightarrow \alpha)$ ”

3) 若 $[S' \rightarrow s \cdot, \$]$ 在 I_i 中，那将 Action $[i, \$]$ 设为“接受”

4) 若 $GOTO(I_i, A) = I_j$ ，那么 $GOTO(i, A) = j$

↓
非终结字符

→ 向前看 □ LALR(1)

- 解决问题：LR(1)文法比SLR(1)描述能力加强，但LR(1)对应的项集族变得很大。若SLR(1)有几百个项集，则LR(1)会有几千个项集。
LALR(1)是在尽可能保持LR(1)的描述能力下提出而用于减少项集数的方法。
- 途径：LR(1)的项表示为[产生式，向前符号]，产生式部分被称为项的核心。为了缩减项集数，我们寻找具有相同核心的LR(1)项集，并将这些项集合并为一个项集。如此LALR(1)得到的项集数等于SLR(1)。
- 上述操作引入的冲突：只会引入归约/归约冲突（不会引入移入/归约冲突）
假设会引入移/归冲突，即存在项 $[A \rightarrow \alpha; a]$ 和项 $[B \rightarrow \beta \cdot ar, b]$ ，前者遇到 a 时归约，后者遇 a 时移入，故冲突。因为合并时要求核心相同，所以合并前至少有一项集存在 $[A \rightarrow \alpha; a]$ 、 $[B \rightarrow \beta \cdot ar, \dots]$ 并存，此时便已产生冲突，与前提至 LR(1) 文法矛盾。故假设不成立。也就说明不会引入移入/归约冲突。

• (LR 语法分析表构造 (低效版))

先构造 LA(1) 项集族 → 合并项集得 LALR(1) 项集族 →
构造 LALR(1) 语法分析表。

□ 高效构造 LALR 语法分析表

- 过程 | 1) 构造 LR(0) 项集族的内核
| 2) 利用‘生成’和‘传播’为内核生成向前看符号，从而得 LALR(1)
| 向内核项
| 3) 由内核项的闭包运算得完整的 LALR(1) 项集族。 15

• 思想

P173 最后为 LR(0) 内核项 添加向前瞻符号使其变成 LAR(1) 内核项.

怎么做呢？ 因为有图 4-40 的规范 LR(1) 构造过程，
其中 项的第二分量通过以下公式计算：

$GOTO(Closure([A \rightarrow \alpha \cdot \beta, a]), X)$

(注 这里只需考虑不同项集的)

对应计算结果有两种 项间的向前符号的关系。因为同一
情况。

情况1) 存在一个 闭包得到)

包含内核项 $[A \rightarrow \alpha \cdot \beta, a]$ 的次集 I，并且 $J = GOTO(I, X)$
~~不管 a 为归值~~ $GOTO(Closure([A \rightarrow \alpha \cdot \beta, a]), X)$ 时
得到结果中总包含 $[B \rightarrow \gamma \cdot \delta, b]$ \Rightarrow 因而 b 由

情况2) 与情况1) 条件相同但是呢，得到
的结果总包含 $[B \rightarrow \gamma \cdot \delta, b=a]$ 这因为项 $A \rightarrow \alpha \cdot \beta$ 有
一个向前瞻符号 b \Rightarrow 传播

再仔细想想 P167 的 GOTO 函数 针对内核项

时 只需做循环无需做 return 为 closure。所以
不同次集间的内核项的第二分量还是在 确定故
有2种情况，也对应 P166—167 所述的2种情况

• 确定向前看符号

由 I 中内核项组成的集合

输入：一个 $LR(0)$ 项集 J 的内核 K 及一个文法符号 X

输出： $GOTO(I, X)$ 的内核项目生成的向前看符号，以及 I 把向前看符号传至 $GOTO(J, X)$ 的那些内核项

for (K 中的每个项 $A \rightarrow \alpha \cdot \beta$) { → 文法不存在的符号

$J := Closure(\{A \rightarrow \alpha \cdot \beta, \# \})$;

if ($[B \rightarrow r \cdot X s, a]$ 在 J 中，并且 $a \neq \#$)

判定 $GOTO(I, X)$ 中的项 $B \rightarrow r \cdot X \cdot s$ 的向前看符号 a 是自发生 的；

if ($[B \rightarrow r \cdot X s, \#]$ 在 J 中)

判定向前看符号从 I 中的项 $A \rightarrow \alpha \cdot \beta$ 传播到 $GOTO(J, X)$

中 $r \cdot X \cdot s$ 上；

}

• 由初始值（即自生成的向前看符号）和‘传播路径’生成完整的 $LR(1)$ 的内核（因各） → 不断传播的过程

• 注： $LALR$ 分析器可能在 LR 语法分析报错之后继续执行一些归约动作（因为其合并了一些归约）但 $LALR$ 语法分析器决不会在 LR 语法分析器报错之后移入任务符号。

□ 使用二义性文法

• 注：每个二义性文法都是 LR 的

• {二义性文法能够提供比任何等价的无二义性文法更短、更自然的归约
易扩展}

- 但应注意要保守地使用二义性构造，且在严格控制之一
- 如何解决二义性文法对应语法分析表中的冲突 —— 对冲突的条目制定规则
 - | 用优先级、结合性解决冲突
 - | 语义（悬空-else' 的二义性处理）

□ LR 语法分析中的错误恢复

- 恶性模式

圈飞过错误

实践中通常选择代表了主要程序段内，非终结符
具体操作，从栈顶向下扫描，直至发现某个状态 s ，它 其中有一个 对应于某个非终结符号 A 的 GOTO 目标 $| \text{对应项 } A \rightarrow \alpha \cdot |$ 。然后我们丢弃零个或多个输入符号，直到发现一个可能合法地跟在 A 后的符号 α 为止。之后将 $GOTO(s, A)$ 压入栈中，继续进行正常的语法分析。

- 短语层次错误恢复

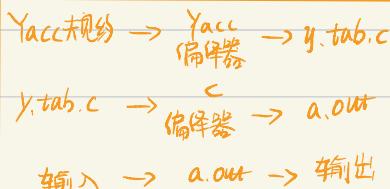
分析表的排错条目指向对应错误修改例程，在例程中可执行一些恢复动作，如修改栈顶状态和/或第一个输入符号。

注：避免 LR 语法分析陷入无限循环（如保证输入会前移才会缩小）

制度

- 附：为减少错误处理的工作量，可将一些错误条目替换为对应状态，下频繁使用归约，再错误延迟至移入时处理。

⑤ Yacc (yet another compiler-compiler)



使用 -V 选项查看有哪些冲突

• 二义性文法的 Yacc 规约

1) Yacc 默认处理冲突的方法

{ 对于归约/归约冲突，选择利用先出现的产生式进行归约
| 对于移入/归约冲突，总选择移入

2) Yacc 提供的自定义的解决移入/归约冲突的机制

{ 使用规定结合性来解决 (%left %right)

{ 使用规定优先级来解决 (根据它们在声明部分的出现顺序而定)

{ 产生式亦有优先级、结合性 (通常产生式的优先级被设为它的最右终结符号的优先级，亦可通过在产生式后添加 %prec <表达式> 来指明)
此时，这个产生式的优先级、结合性同这个终结符号相同

• Yacc 中的错误恢复

采用的是 恐慌模式 → 指那些生成表达式、语句、块等的非终结符号

需在“主要”非终结符号上添加“错误产生式” ($A \rightarrow \text{error} \alpha$)。当遇到错误时从根顶扫描，直至遇到含错误产生式的状态。 error 作为一个保留字，在此时分析器就好像看到 error ，将虚拟的词法单元 error 填入栈中。然后 Yacc 将压前跳过一些输入符号，寻找可被归约为 error 的串。最后若成功归约此错误产生式，则执行错误产生式对应的处理逻辑。

21.03.14

