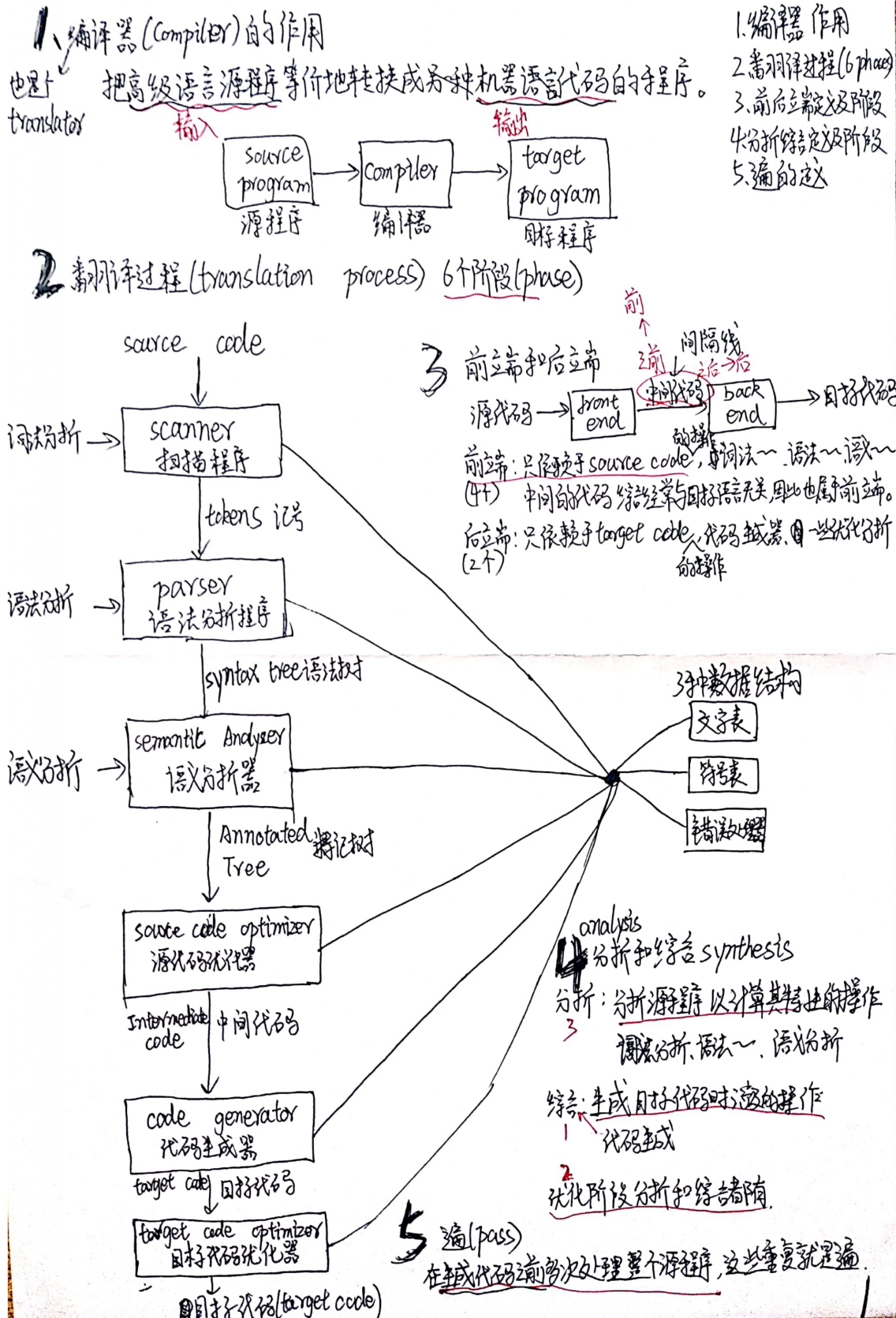


编译原理 (双语) 期末复习

注意: 部分资料来自 chouxianyu.github.io, 版权归原作者所有。

本文图片较多, 加载速度受图床服务器网络状况限制, 请耐心等待。

概论



词法分析

lexical analysis 又称为 scanning

正则表达式

regular expressions

概念

基本正则表达式 (3个)

基本正则表达式是字母表中的单个字符且与自身匹配

- 若 $a \in \Sigma$, 则 $L(a) = \{a\}$
- 空串 ϵ , $L(\epsilon) = \{\epsilon\}$
- 空集 \emptyset , $L(\emptyset) = \{\}$

基本正则表达式

$\begin{cases} a \\ \epsilon \\ \emptyset \end{cases}$

正则表达式的三个基本运算及优先级

重复 repetition : r^*
连接 concatenation : rs
选择 choice : $r|s$

三个基本运算的优先级顺序为: * 优先级最高, 连接 其次, | 优先级最低, 可以用 $()$ 改变这个优先级顺序

在该定义中, \emptyset , ϵ , $|$, $*$, $()$ 这6个符号都有元字符的含义。

- 选择 (choice) : $r|s$
 - $L(r|s) = L(r) \cup L(s)$, 例如 $L(a|b) = L(a) \cup L(b) = \{a, b\}$
- 连接 (concatenation) : rs
 - $L(rs) = L(r)L(s)$, 例如 $L((a|b)c) = L(a|b)L(r) = \{a, b\}\{c\} = \{ab, bc\}$
- 重复 (repetition or closure) : r^*
 - 有时称为Kleene闭包、克林闭包 (Kleene closure), 写作 r^* , r 的0次或多次连接
 - 例如 $S^* = \{\epsilon\} \cup S \cup SS \cup SSS \cup \dots$

正则表达式的两个事实

- ① 不同的正则表达式可以生成相同的语言 (在实际中从未尝试着证实已找到了最简单的, 这有两个原因)
- ② 并非用简单术语描述的所有串都可由正则表达式产生, 我们将正则表达式可匹配的串的集合称为 正则集合 (regular set)

正则表达式的扩展 (9个)

- 一个或多个重复: r^+
- 任意字符: $.$
- 字符范围: $[a-z]$, $[0-9]$, $[a-zA-Z]$
- 非: $\sim r$
- 可选的子表达式 (即0个或1个重复): $r?$

题：解释正则式

1. (10 points) Write English description for the languages generated by following regular expression:

1) $0+(0|1)1+$

001, 011, 0001, 0011, ...

~~至少有1个前导0, 后接1个0或1个1, 后接至少1个1~~
长度至少为3的, 至少1个0在至少1个1后面的字符串

any string of length 3 or greater, which is 1 or more 0's followed by 1 or more 1's.

2) $0^*(100^*)^*1^*$

ϵ , 0, 010, 0100, 01001, 100, 1, ...

不包含"110"子串的任意字符串.

Any string that has no substring 110.

题：检查匹配

- a. Please check out which strings can be generated by the regular expression $(ab|b)^*cc$?

~~abcc~~, ~~abab~~, ~~bcc~~, ~~babcc~~, ~~aabcc~~

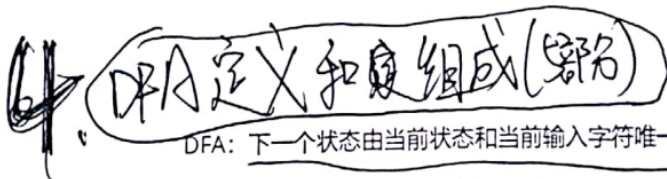
- b. Please check out which strings can be generated by the regular expression $(b|a)b+(ba)^*$?

~~aba~~, ~~abb~~, ~~ababa~~, ~~aab~~, ~~bbb~~

有限自动机

finite automata

DFA



DFA: 下一个状态由当前状态和当前输入字符唯一确定的自动机

一个确定有限自动机 M 由5部分组成:

- 字母表 Σ
- 状态集合 S
- 转换函数 $T: S \times \Sigma \rightarrow S$
 - $S \times \Sigma$ 指的是 S 和 Σ 的笛卡尔积或叉积: 集合对 (s, c) , 其中 $s \in S, c \in \Sigma$.
 - 例如: $T(s, c) = s'$
- 初始状态 s_0
 - $s_0 \in S$
- 接受状态集合 A
 - $A \subset S$

NFA

ϵ -转换

ϵ -transition 是无需考虑输入串就有可能发生的转换，它可以看作一个空串的“匹配”。

有两个好处：

- ① 方便合并 DFA
 - ② 描述空串的匹配
- 使最初的确定性有限自动机保持完整并且只添加一个新的初始状态就可以将几个确定性有限自动机合并
 - 有了它就可以清晰地描述出空串的匹配

非确定性有限自动机 (nondeterministic finite automata)

NFA和DFA有两点不同：

- ① 字母表 Σ
 - ② 转移函数 T
- 字母表 Σ 扩充为 $\Sigma \cup \epsilon$
 - 转移函数 $T : S \times (\Sigma \cup \{\epsilon\}) \rightarrow P(S)$ ，其中 $P(S)$ 为 S 的幂集 (power set)

任意个 ϵ 都可能在任一状态上引入到串中，并与 NFA 中 ϵ 转换的数量相对应。因此，NFA 并不表示算法，但是却可以通过在每个非确定性选择中回溯的算法来模拟它

正则表达式到 NFA

从正则表达式到 NFA

可以使用 Thompson 构造 (Thompson's construction)，其利用 ϵ 转换将正则表达式对应的自动机连接起来形成整个正则表达式的自动机。

基本正则表达式的 NFA

连接 (concatenation) 的 NFA

选择 (choice) 对应的 NFA

重复 (repetition) 的 NFA



NFA到DFA

从NFA到DFA

状态 s 的 ϵ -闭包 (ϵ -closure)

我们将单个状态 s 的 ϵ -closure定义为: 状态 s 可经过0个或多个 ϵ -closure达到的状态的集合, 并将这个集合记作 \bar{s} .

一个状态的 ϵ 总是包含着该状态本身

状态集合的 ϵ -closure

我们将一个状态集合的 ϵ -closure定义为集合中每个状态的 ϵ -closure的并.

记状态集合为 S , 则 $\bar{S} = \bigcup_{s \in S} \bar{s}$

子集构造 (subset construction)

设有NFA M , 则其DFA为 \bar{M} .

① M 开始状态的闭包即 \bar{M} 的初始状态

② 求每个状态在每字符 a 上的转换, 即求 $S \xrightarrow{a} S'$

例: A. 求 S_a . 若 S 中状态 t 的转换($s \xrightarrow{a} t$), 则 $t \in S_a$.

例: B. 求 \bar{S}_a . 求 S_a 的 ϵ 闭包.

例: C. 求 \bar{S} .

\bar{M} 中包含 M 接受状态的状态为 \bar{M} 的接受状态.

注意: 检查集合间的重复
一直计算新集合

最小化DFA

9. DFA状态数最小化

对于任意的DFA, 都有一个含有最少量的状态的等价DFA, 且这个最小状态的DFA是唯一的.

状态在同一个集合内代表着等价

方法:

① 将DFA状态分为两个集合: 非接受状态集合和接受状态集合

② 对于字符集上的每个字符 a .

检查每个集合中是否有 a 区分状态 s 和 t , 若有, 则拆分, 并重复步骤2

若无, 则定义了该集合到其自身的 a 转换

区分的定义: 若 a 区分 s 和 t ,

即 s 和 t 在 a 上有不同的转换, 有2种情况.

① 转换至不同集合

② 存在错误转换和空转换

题：正则->NFA->DFA

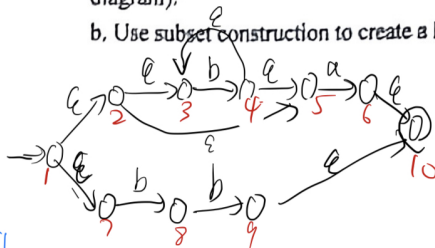
$\{a, b\}$

3. (12 points) Consider the following regular expression from the alphabet $\{a, b\}$:

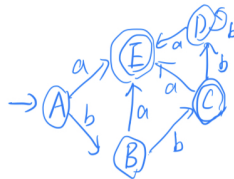
$(b^*a)(bb)$

a. Use Thompson's construction to make an NFA from the regular expression (show it as a state diagram).

b. Use subset construction to create a DFA equivalent to the NFA you gave for part A.



	a	b
A	E	B
B	E	C
C	E	D
D	E	D
E		



$\epsilon\text{-closure}[1] = [1, 2, 3, 4, 5, 7]$
 A $\text{mov}(1, 2, 3, 4, 5, 7, a) = [6]$
 $\epsilon\text{-closure}[6] = [6, 10]$ - Final state 终点
 A $\text{mov}(1, 2, 3, 4, 5, 7, b) = [4, 8]$
 $\epsilon\text{-closure}[4, 8] = [3, 4, 5, 8]$
 B $\text{mov}(3, 4, 5, 8, a) = [6]$
 $\epsilon\text{-closure}[6] = [6, 10]$ - Final state
 B $\text{mov}(3, 4, 5, 8, b) = [4, 9]$
 $\epsilon\text{-closure}[4, 9] = [3, 4, 5, 9, 10]$ - Final state
 C $\text{mov}(3, 4, 5, 9, 10, a) = [6]$
 $\epsilon\text{-closure}[6] = [6, 10]$ - Final state
 C $\text{mov}(3, 4, 5, 9, 10, b) = [4]$
 $\epsilon\text{-closure}[4] = [3, 4, 5]$
 D $\text{mov}(3, 4, 5, a) = [6]$
 $\epsilon\text{-closure}[6] = [6, 10]$ - Final state
 D $\text{mov}(3, 4, 5, b) = [4]$
 $\epsilon\text{-closure}[4] = [3, 4, 5]$

4. (6 points) Given the grammar:

$E \rightarrow T | E + T | E \cdot T$

$T \rightarrow F | T * F | T / F$

$F \rightarrow (E) | i$

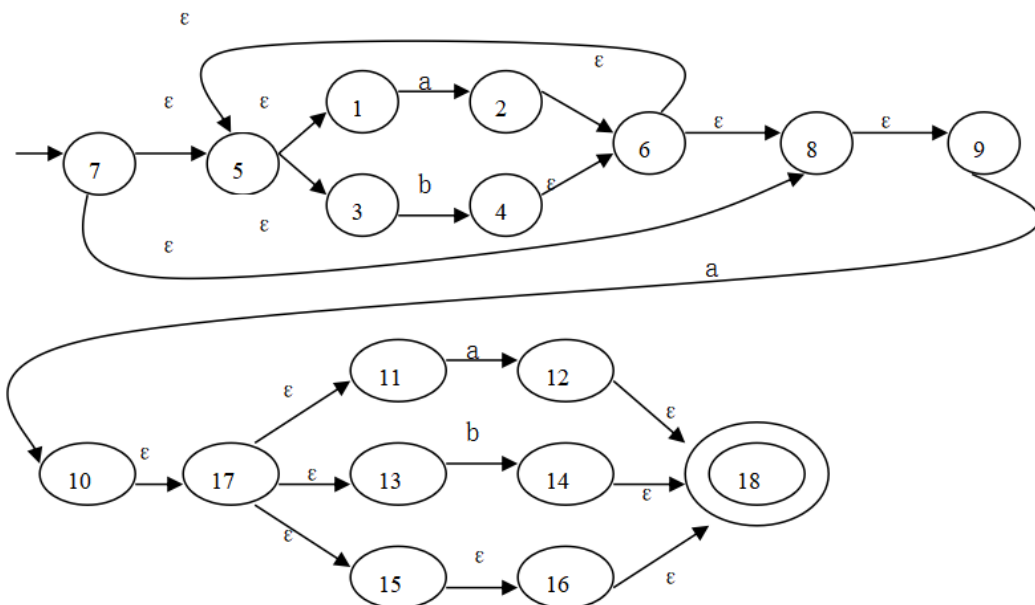
/ 11

2.12 a. Use Thompson's construction to convert the regular expression $(a|b)^*a(a|b)^*\epsilon$ into an NFA.

b. Convert the NFA of part (a) into a DFA using the subset construction.

[Solution]

a. An NFA of the regular expression $(a|b)^*a(a|b)^*\epsilon$



b. The subsets constructed as follows:

$$\underline{\{7\}} = \{7, 5, 1, 3, 8, 9\}$$

$$\underline{\{7\}}_a = \{2, 10\}$$

$$\underline{\{7\}}_b = \{4\}$$

$$\underline{\{2, 10\}} = \{2, 6, 5, 1, 3, 8, 9, 10, 17, 11, 13, 15, 16, 18\}$$

$$\underline{\{2, 10\}}_a = \{2, 10, 12\}$$

$$\underline{\{2, 10\}}_b = \{4, 14\}$$

$$\underline{\{2, 10, 12\}} = \{2, 6, 5, 1, 3, 8, 9, 12, 18, 10, 17, 11, 13, 15, 16\}$$

$$\underline{\{2, 10, 12\}}_a = \{2, 10, 12\}$$

$$\underline{\{2, 10, 12\}}_b = \{4, 14\}$$

$$\underline{\{4, 14\}} = \{4, 6, 5, 1, 3, 8, 9, 14, 18\}$$

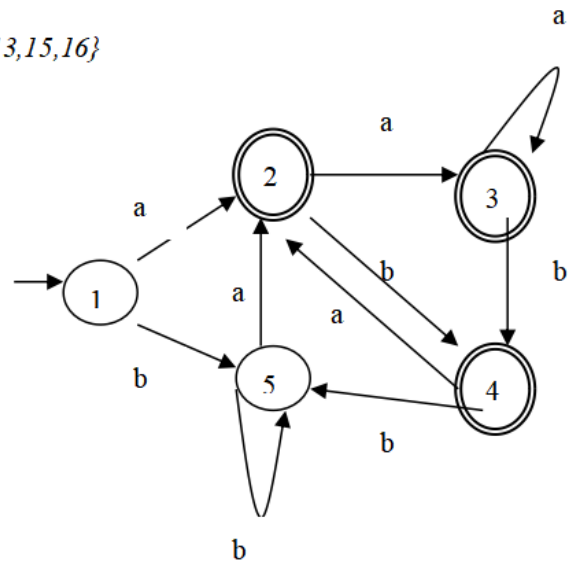
$$\underline{\{4, 14\}}_a = \{2, 10\}$$

$$\underline{\{4, 14\}}_b = \{4\}$$

$$\underline{\{4\}} = \{4, 6, 5, 1, 3, 8, 9\}$$

$$\underline{\{4\}}_a = \{2, 10\}$$

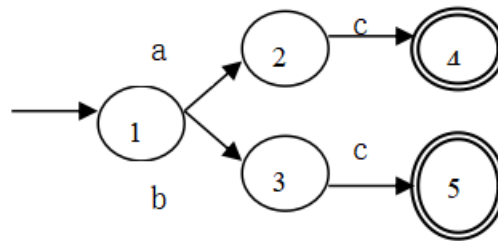
$$\underline{\{4\}}_b = \{4\}$$



题：最小化DFA

2.16 Apply the state minimization algorithm of section 2.4.4 to the following DFAs:

a.



[Solution]

a. Step 1: Divide the state set into two subsets:

$\{1, 2, 3\}$

$\{4, 5\}$

Step 2: Further divide the subset $\{1, 2, 3\}$ into two new subsets:

$\{1\}$

$\{2, 3\}$

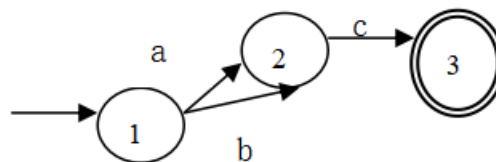
Step 3: Can not divide the subsets any more, finally obtains three subsets:

$\{1\}$

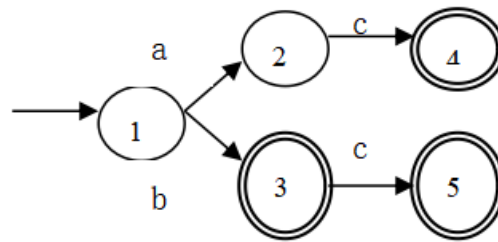
$\{2, 3\}$

$\{4, 5\}$

Therefore, the minimized DFA is:



b.



[Solution]

b. Step 1: Divide the state set into two subsets:

$\{1, 2\}$

$\{3, 4, 5\}$

Step 2: Further divide the subset $\{1, 2\}$ into two new subsets:

$\{1\}$

$\{2\}$

Step 2: Further divide the subset $\{3, 4, 5\}$ into two new subsets:

$\{3\}$

$\{4, 5\}$

Step 4: Can not divide the subsets any more, finally obtains three subsets:

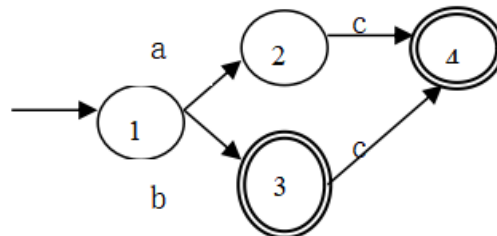
$\{1\}$

$\{2\}$

$\{3\}$

$\{4, 5\}$

Therefore, the minimized DFA is:



语法分析

上下文无关文法

概念

2 上下文无关文法

定义

上下文无关文法由以下各项组成:

令 G 为一个上下文无关文法, 即 $G = (T, N, P, S)$.

- 终结符集合 T
- 非终结符集合 N

非终结符集合和非终结符集合不相交

- 产生式集合 P

产生式 $A \rightarrow \alpha$, 其中 A 是一个非终结符, α 是终结符或非终结符的克林闭包, 即 $\alpha \in (T \cup N)^*$.

- 开始符号 S

开始符号是一个非终结符

最左推导 最右推导

最左推导

有推导 $S \Rightarrow^* \omega$, 若其中的每一个推导步骤 $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ 都有 $\alpha \in T^*$, 则其为最左推导, 即先推导左边再推导右边。

最右推导

有推导 $S \Rightarrow^* \omega$, 若其中的每一个推导步骤 $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ 都有 $\gamma \in T^*$, 则其为最右推导, 即先推导右边再推导左边。

文法 G 上的分析树

分析树

4. 文法G上的分析树 每层从右往左编

文法G上的分析树是一个带有以下属性的做了标记的树:

- 每个节点都用终结符、非终结符或 ϵ 标出
- 根节点都用开始符号标出
- 每个叶子节点都用终结符或 ϵ 标出
- 每个非叶子节点(内部节点)都用非终结符标出
- 每个内部节点到其所有子节点表示一个推导步骤中的相关非终结符的替换
- 如带有标记 $A \in N$ 的非叶子节点有 n 个带有标记 X_1, X_2, \dots, X_n 的孩子节点, 就有

4. 分析树与推导的关系
 $A \rightarrow X_1 X_2 \dots X_n \in P$
 每一个推导都引出一个分析树; 许多推导可引出相同的分析树, 但每个分析树只有唯一的一个最左推导和唯一的一个最右推导 (最左推导与分析树的前序编号相对应, 最右推导与分析树的后序编号相对应)

题: 列出文法信息

4. (6 points) Given the grammar:

$$E \rightarrow T | E + T | E - T$$

$$T \rightarrow F | T * F | T / F$$

$$F \rightarrow (E) | i$$

Please list all non-terminals and terminals in this grammar, and give the start symbol of the grammar.

\downarrow \downarrow E
 $\{+, -, *, /, (,), i\}$ $\{E, T, F\}$

题: 最左推导、最右推导

5. (10 points) Given the grammar

$$\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$$

$$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$$

$$\text{factor} \rightarrow (\text{exp}) \mid \text{number}$$

Write down leftmost derivations for: $3*(6-5)$ and rightmost derivations for $16*6/4$

$$\begin{aligned} \textcircled{1} \text{exp} &\Rightarrow \text{term} \Rightarrow \text{term} * \text{factor} \Rightarrow \text{factor} * \text{factor} \Rightarrow 3 * \text{factor} \Rightarrow 3 * (\text{exp}) \Rightarrow 3 * (\text{exp} - \text{term}) \\ &\Rightarrow 3 * (\text{term} - \text{term}) \Rightarrow 3 * (\text{factor} - \text{term}) \Rightarrow 3 * (6 - \text{term}) \Rightarrow 3 * (6 - \text{factor}) \Rightarrow 3 * (6 - \text{number}) \end{aligned}$$

$$\begin{aligned} \textcircled{2} \text{exp} &\Rightarrow \text{term} \Rightarrow \text{term} / \text{factor} \Rightarrow \text{term} / 4 \Rightarrow \text{term} * \text{factor} / 4 \Rightarrow \text{term} * 6 / 4 \Rightarrow \text{factor} * 6 / 4 \\ &\Rightarrow 16 * 6 / 4 \end{aligned}$$

题：分析树、抽象语法树

3.3 Given the grammar

$exp \rightarrow exp \text{ addop } term \mid term$

$addop \rightarrow + \mid -$

$term \rightarrow term \text{ mulop } factor \mid factor$

$mulop \rightarrow *$

$factor \rightarrow (exp) \mid \text{number}$

Write down leftmost derivations, parse trees, and abstract syntax trees for the following expression:

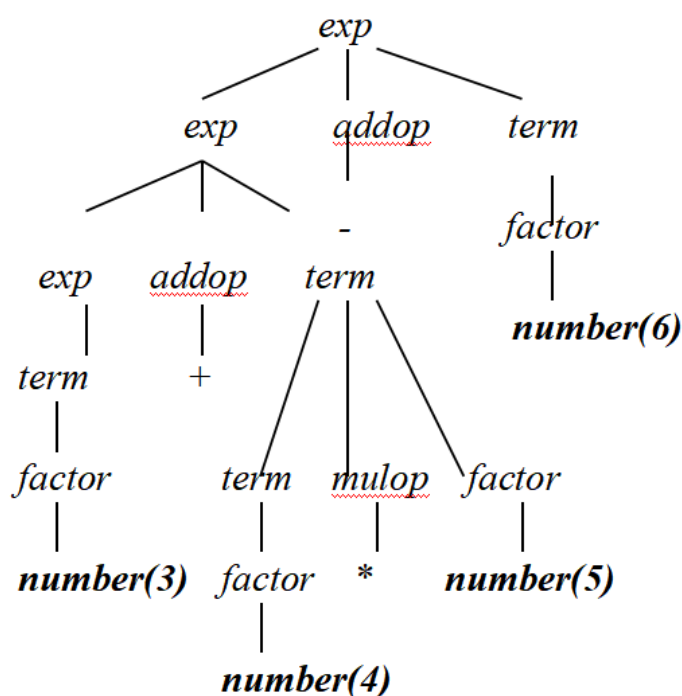
- a. $3+4*5-6$ b. $3*(4-5+6)$ c. $3-(4+5*6)$

[Solution]:

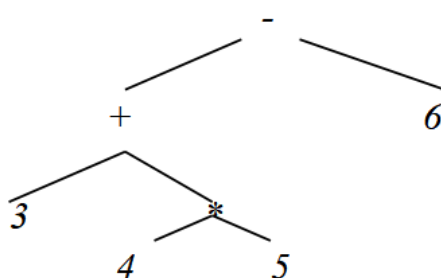
- a. The leftmost derivations for the expression $3+4*5-6$:

$Exp \Rightarrow exp \text{ addop } term \Rightarrow exp \text{ addop } term \text{ addop } term$
 $\Rightarrow term \text{ addop } term \text{ addop } term \Rightarrow factor \text{ addop } term \text{ addop } term$
 $\Rightarrow 3 \text{ addop } term \text{ addop } term \Rightarrow 3 + term \text{ addop } term$
 $\Rightarrow 3 + term \text{ mulop } factor \text{ addop } term \Rightarrow 3 + factor \text{ mulop } factor \text{ addop } term$
 $\Rightarrow 3 + 4 \text{ mulop } factor \text{ addop } term \Rightarrow 3 + 4 * factor \text{ addop } term$
 $\Rightarrow 3 + 4 * 5 \text{ addop } term \Rightarrow 3 + 4 * 5 - term \Rightarrow 3 + 4 * 5 - factor \Rightarrow 3 + 4 * 5 - 6$

The parse tree for the expression $3+4*5-6$:



The abstract syntax tree for the expression $3+4*5-6$:



自底向下分析

LL(1)文法

First集

First集是指对于给定的文法符号，它可能推导出的第一个终结符的集合。

- 对于一个非终结符A， $First(A)$ 表示A能够推导出的所有可能的第一个终结符的集合。
- 对于一个产生式，First集也可以用于表示该产生式右侧推导出的所有可能的第一个终结符的集合。

$First(E)$ 找E变成的产生式的第一个非终结符的集合

First集合 → 终结符和非终结符都有

含义 可以正规地开始这个串的token。
如 $First(A) = \{a, (\}$ ，则串A可以由a和(开始。

构造方法

若A是终结符，则 $First(A) = \{A\}$ 。
设有产生式 $A \rightarrow \alpha$ ，其中A是非终结符， $\alpha = X_1 X_2 \dots X_n$ 是终结符和非终结符的串，
 $First(A) = First(\alpha)$ 。

1. $First(\alpha)$ 包括 $First(X_1) - \{\epsilon\}$;
2. 对于每个 $X_i, i = 2, \dots, n$ ，如果所有的 $First(X_k), k = 1, \dots, i-1$ 都包括 ϵ ，则 $First(\alpha)$ 包括 $First(X_i) - \{\epsilon\}$;
3. 如果所有的 $First(X_i), i = 1, \dots, n$ 都包括 ϵ ，则 $First(\alpha)$ 包括 ϵ 。

需要 将文法规则消除左递归和提取公因子，然后将选择展开，生成新的文法。

按照一定顺序，对于每一个产生式，运行上面的三个步骤，直至First集合不再发生变化，First集合即构造成功。

文法规则	第1遍	第2遍	...
~	$First() = \{ \}$		
~			
~			
~			

Follow集

Follow集是指对于给定的非终结符，它在句子中可能紧跟在其后的终结符的集合。对于一个非终结符A， $Follow(A)$ 表示在任何一个句子中，A后面可能出现的终结符的集合。

$Follow(E)$ 找E后面可能跟哪些非终结符，如果E在产生式的最后，那么找这个式子的左部后面可能跟哪些非终结符。Follow集里不可能有 ϵ 。注意\$符号（或#号）

① 文法开始符，必有#

② $A \rightarrow \alpha B$ FOLLOW(B) B后为 ϵ ，将follow(A)加入到follow(B)中。

③ $A \rightarrow \alpha B \beta$ $\left\{ \begin{array}{l} \beta \text{ 是终结符，直接写下来} \\ \beta \text{ 是非终结符，} first(\beta) \text{ 加入到 follow(B) 中。} \end{array} \right.$

如果 $\beta \rightarrow \epsilon$ ，带入 $A \rightarrow \alpha B \beta$ 中，得到 $A \rightarrow \alpha B$

3. Follow集合 → 非终结符前，不是终结符 + \$

含义

可以正规地出现在非终结符A之后的token的集合，这样的token指出A可以恰当地在南非西中的这个点处消失。

构造方法

遍历产生式，关注产生式右边

设A为非终结符，那么Follow(A)由终结符组成，此外可能还有\$。

1. Follow(S)包括\$，其中S为开始符号；
2. 若存在产生式 $B \rightarrow \alpha A \gamma$ ，则Follow(A)包括Follow(γ) - $\{\epsilon\}$ ；
3. 若存在产生式 $B \rightarrow \alpha A \gamma$ 且First(γ)包括 ϵ ，则Follow(A)包括Follow(B)。

构造格式同First集合构造格式

First(ϵ) = $\{\epsilon\}$

Select集

Select集是用于确定某个产生式是否应该被应用的条件。对于一个产生式P，Select(P)表示当满足某些条件时，产生式P应该被应用。Select集可以通过First集和Follow集计算得出。Select集合里不可能有 ϵ 。

对于一个产生式 $E \rightarrow \alpha$ ， $\text{Select}(E \rightarrow \alpha) = \text{First}(\alpha) = \text{First}(\text{产生式右部})$

如果 α 可以为 ϵ ， $\text{Select}(E \rightarrow \alpha) = \text{First}(\alpha) - \epsilon + \text{Follow}(E) = \text{First}(\text{产生式右部}) - \epsilon + \text{Follow}(\text{产生式左部})$

构造Select集合：

Select集大白话理解：

- 定义与作用：Select集是针对产生式而言，对于一个产生式，它所产生的句子的句首符号所构成的集合称为Select集。其作用就是用于构建预测分析表，以便之后分析时选择正确的产生式。

构造方法

有了前面First集和Follow集的构造，Select集合的构造就相对比较简单了，它有两种情况：

- 第一种是产生式能产生一个句子，但是不能推导出 ϵ ，则 $\text{Select}(E \rightarrow \alpha) = \text{First}(\alpha)$
- 第二种是产生式能推导出 ϵ ，则 $\text{Select}(E \rightarrow \alpha) = \{\text{First}(\alpha) - \epsilon\} + \text{Follow}(E)$

例子：

E \rightarrow TA
A \rightarrow +TA | ϵ
T \rightarrow FB
B \rightarrow *FB | ϵ
F \rightarrow i | (E)

line1：只有一个候选产生式，且E无法推导出 ϵ ，所以 $\text{Select}(E \rightarrow \text{TA}) = \text{First}(T)$

line2：第二行有两个候选式，针对第一个候选式A \rightarrow +TA，因为以+开头，那么肯定就推不出 ϵ ，故 $\text{Select}(A \rightarrow +\text{TA}) = +$ ，而第二个产生式能推出 ϵ ，故 $\text{Select}(A \rightarrow \epsilon) = \text{Follow}(A)$

line3：与第一行类似，略

line4：与第二行类似，略

line5：针对第一个产生式F \rightarrow i，满足第一条规则，所以 $\text{Select}(F \rightarrow i) = i$ ；针对第二个产生式也是如此。

判定定理

- 文法 G 是 $LL(1)$ 的，当且仅当 G 的任意两个具有相同左部的产生式 $A \rightarrow \alpha \mid \beta$ 满足下面的条件：
 - 不存在终结符 a 使得 α 和 β 都能够推导出以 a 开头的串
 - α 和 β 至多有一个能推导出 ε
 - 如果 $\beta \Rightarrow^* \varepsilon$ ，则 $FIRST(\alpha) \cap FOLLOW(A) = \Phi$ ；
如果 $\alpha \Rightarrow^* \varepsilon$ ，则 $FIRST(\beta) \cap FOLLOW(A) = \Phi$ ；

同一非终结符的各个产生式的可选集互不相交

消除左递归

Left Recursion Removal

$A \rightarrow A\alpha \mid \beta$

Where, α and β are strings of terminals and non-terminals;
 β does not begin with A .

The grammar will generate the strings of the form $\beta\alpha^n$.

为了消除左递归，将这个文法规则重写为两个规则：一个是首先生成 β ，另一个是生成 α 的重复，它不用左递归却用右递归：

$A \rightarrow A\alpha \mid \beta$  $A \rightarrow \beta A'$
 $A' \rightarrow \alpha A' \mid \varepsilon$

例4.1 再次考虑在简单表达式文法中的左递归规则：

$exp \rightarrow exp \text{ addop } term \mid term$

它属于格式 $A \rightarrow A\alpha \mid \beta$ ，且 $A = exp$ ， $\alpha = \text{addop } term$ ， $\beta = term$ 。将这个规则重写以消除左递归，就可得到

$exp \rightarrow term exp'$
 $exp' \rightarrow \text{addop } term exp' \mid \varepsilon$

CASE2: General immediate left recursion

$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \beta_1 | \beta_2 | \dots | \beta_m$
Where none of β_1, \dots, β_m begin with A .

The solution is similar to the simple case:

$A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_m A'$
 $A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A' | \epsilon$

这种情况发生在有如下格式的产生式

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \beta_1 | \beta_2 | \dots | \beta_m$$

中。其中 β_1, \dots, β_m 均不以 A 开头。在这种情况下，其解法与简单情况类似，只需将选择相应地扩展：

$$A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_m A'$$
$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A' | \epsilon$$

提取左公因子

当两个或更多个产生式选择共享一个通用前缀串时，需要提取左因子。

如 $A \rightarrow \alpha\beta | \alpha\gamma$ 可重写为：

$\epsilon | \alpha A'$

$A \rightarrow \alpha A'$
 $A' \rightarrow \beta | \gamma$

二义性

可生成带有两个不同分析树的串的文法称作二义性文法 (ambiguous grammar)。由于这个文法并不能准确地指出程序的语法结构 (即使是完全确定正规串本身 (它是文法的语言成员))，所以它是分析程序表示的一个严重问题。在某种意义上，二义性文法就像是一个非确定的自动机，此时两个不同的路径都可接收相同的串。但是因为没有合适的算法，因此就不能像自动机中的情形一样 (第2章中讨论过的子集构造)，文法中的二义性就不能如有穷自动机中的非确定性一样轻易地被解决^②。

文法的二义性是指一个句子在该文法下有两个或多个不同的推导树。消除文法的二义性对于语法分析过程非常重要，因为它可以确保句子具有唯一的语法结构，从而避免歧义。以下是一些常用的方法来消除文法的二义性：

1. 重新定义文法规则：

尝试通过重新定义文法规则来消除二义性。这可以包括合并某些规则、拆分规则、重新排列规则等。目标是确保每个句子在新文法下只有一个唯一的推导树。

2. 引入优先级和结合性:

为文法中的运算符定义优先级和结合性规则。优先级规则确定了哪些运算符应该优先计算，而结合性规则确定了具有相同优先级的运算符的计算顺序。通过为文法中的运算符分配优先级和结合性，可以消除由于运算符顺序不明确导致的二义性。

3. 采用左递归或右递归:

根据需要，可以将文法规则改写为左递归或右递归的形式。左递归适用于左结合的运算符，而右递归适用于右结合的运算符。通过将文法规则改写为适当的递归形式，可以消除由于运算符结合性不明确导致的二义性。

分析表

可以在Select集合的基础上构建，也可以直接构建

LL(1)分析表

定义

一个二维数组，索引为 非终结符 与 终结符和 $\$$ ，这个表称为 $M[N, T]$ 。

作用

为替换栈顶的A选择产生式

构造方法

假设分析表在开始时，所有项目为空。

根据以下规则在这个表中添加产生式:

对于每个产生式 $A \rightarrow \alpha$ ，其中A是非终结符，重复以下两个步骤:

- ① 对于 $First(\alpha)$ 中的每个终结符 a ，都将 $A \rightarrow \alpha$ 添加至 $M[A, a]$ 中;
- ② 若 $First(\alpha)$ 包括 ϵ ，则对于 $Follow(A)$ 中的每个元素 a (记号或者是 $\$$)，都将 $A \rightarrow \alpha$ 添加至 $M[A, a]$ 。

分析过程

注意，替换时要反着压入parsing stack

反着压才可以利用select集的定义，让新串与input匹配



LL(1)分析

- 不用将BNF转换为EBNF，但仍需处理文法规则
 - LL(1)分析中的重复和可选也存在着与在递归下降程序分析中遇到的类似问题
 - 我们利用EBNF表示法解决了递归下降程序中的这些问题
 - 但却不能在LL(1)分析中使用相同的方法，而是使用左递归消除和提取左因子
 - 左递归消除和提取左因子不能保证将一个文法变成LL(1)文法，就像EBNF不能保证在编写递归下降程序中可以解决所有的问题
- 使用显示栈而不是递归调用来完成分析

过程

需要画4列

步骤	分析栈	输入	动作
1	$\$ \$$	$\dots \$$...
2
...
n	$\$$	$\$$	accept

1. 自顶向下的分析程序是从将开始符号放在栈中开始的；

2. 一系列步骤

- 生成 *generate*

选择 $A \rightarrow \alpha$ 将栈顶的非终结符 A 替换为 α (注意: α 要倒序进栈)

- 匹配 *match*

将栈顶的token与输入中的下一个token匹配 (要将匹配成功的两个token从栈中和输入中删除)

3. 栈和输入都空，即栈和输入中都只有 $\$$ ，此时接受 (accept)

题: LL(1)分析表

6. (25 point) Consider the following grammar:

$$\underline{S \rightarrow Sb} \quad S \rightarrow Ab \quad S \rightarrow b \quad \underline{A \rightarrow Aa} \quad A \rightarrow a$$

a. remove the left recursion. (5 point)

b. Construct First and Follow sets for the nonterminals of the resulting grammar. (6 point)

c. Construct the LL(1) parsing table for the resulting grammar. (6 point)

d. show the action of LL(1) parser that used the parsing table to recognize the following string: aaabb. (8 point)

$$\begin{aligned} \text{b. } \text{First}(S) &= \{a, b\} & \text{Follow}(S) &= \{\$ \} \\ \text{First}(S') &= \{b, \epsilon\} & \text{Follow}(S') &= \{\$ \} \\ \text{First}(A) &= \{a, \} & \text{Follow}(A) &= \{b\} \\ \text{First}(A') &= \{a, \epsilon\} & \text{Follow}(A') &= \{b, \} \end{aligned}$$

xi&:
$$\begin{aligned} S &\rightarrow Abs' & A &\rightarrow aA' \\ S &\rightarrow bs' & A' &\rightarrow aA' \\ S' &\rightarrow bs' & A' &\rightarrow \epsilon \\ S' &\rightarrow \epsilon \end{aligned}$$

$$\begin{aligned} \text{c. } \text{select}(S \rightarrow Abs') &= \{a\} \\ \text{select}(S \rightarrow bs') &= \{b\} \\ \text{select}(S' \rightarrow bs') &= \{b\} \\ \text{select}(S' \rightarrow \epsilon) &= \{\$ \} \\ \text{select}(A \rightarrow aA') &= \{a\} \\ \text{select}(A' \rightarrow aA') &= \{a\} \\ \text{select}(A' \rightarrow \epsilon) &= \{b\} \end{aligned}$$

LL(1) Parsing Table

	a	b	\$
S	$S \rightarrow Abs'$	$S \rightarrow bs'$	
S'		$S' \rightarrow bs'$	$S' \rightarrow \epsilon$
A	$A \rightarrow aA'$		
A'	$A' \rightarrow aA'$	$A' \rightarrow \epsilon$	

题: LL(1)分析过程

parsing stack	input	Action.
\$ S	aaabbb \$	$S \rightarrow ABS'$
\$ S' b A	aaabbb \$	$A \rightarrow aA'$
\$ S' b A' a	aaabbb \$	match
\$ S' b A'	aabbb \$	$A' \rightarrow aA'$
\$ S' b A' a	aaabbb \$	match
\$ S' b A'	abbb \$	$A' \rightarrow aA'$
\$ S' b A' a	abbb \$	match
\$ S' b A'	bbb \$	$A' \rightarrow \epsilon$
\$ S' b	bb \$	match
\$ S'	b \$	$S' \rightarrow bS'$
\$ S' b	b \$	match
\$ S'	\$	$S' \rightarrow \epsilon$
\$	\$	Accept

自底向上分析

LR(0)文法

构造分析表

Left to Right, Rightmost.
LR(0) 的构造

eg. $G[E]: E \rightarrow E+T \mid T$
 $T \rightarrow (E) \mid a$

① 拓广文法

$G[E']$

- (0): $E' \rightarrow E$
- (1): $E \rightarrow E+T$
- (2): $E \rightarrow T$
- (3): $T \rightarrow (E)$
- (4): $T \rightarrow a$

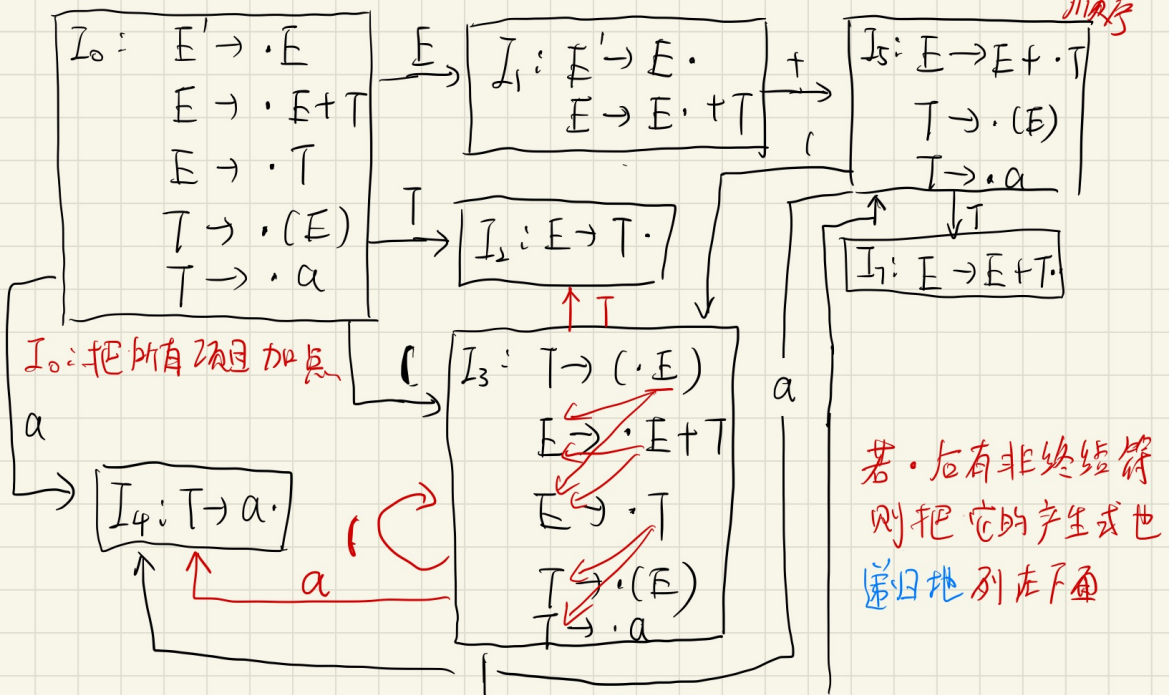
a. $E' \rightarrow$ 开始符号
b. 将多项拆开
c. 标示号

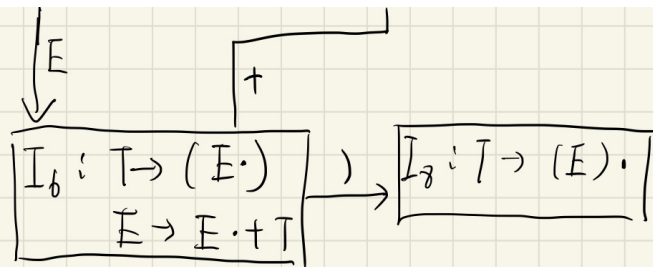
② 列项目 (可略过)

特别地, 若 $A \rightarrow \epsilon$, 只写 $A \rightarrow \cdot$

③ 项目集规范族 (直接构造 DFA)

分析顺序: I 的下标
顺序





④ 分析表

S: shift, 移进

R: Reduce, 归约

状态	Action 终结符					GOTO 非终结符	
	a	+	()	#	E	T
I_0	$S(I_4)$		$S(I_3)$			I_1	I_2
I_1		$S(I_5)$			Accept		
I_2	$R(2)$	$R(2)$	$R(2)$	$R(2)$	$R(2)$		
I_3	$S(I_4)$		$S(I_3)$			I_6	I_2
I_4	$R(4)$	$R(4)$	$R(4)$	$R(4)$	$R(4)$		
I_5	$S(I_4)$		$S(I_3)$				I_7
I_6		$S(I_5)$		$S(I_3)$			
I_7	$R(1)$	$R(1)$	$R(1)$	$R(1)$	$R(1)$		
I_8	$R(3)$	$R(3)$	$R(3)$	$R(3)$	$R(3)$		

注: ① $GOTO(I_0, E) = I_1$. $I_0 \xrightarrow{E} I_1$

② $Action(I_0, a) = S(I_4)$. $I_0 \xrightarrow{a} \text{shift to } I_4$

③ $Action(I_2, \#) = R(2)$. I_2 完成了第 (2) 条产生式

④ $Action(I_1, \#) = \text{Accept}$. I_1 完成了整个推广文法

SLR(1)文法

构造分析表

SLR(1) 的构造

Simple

引入: 为了解决“移进-归约”和“归约-归约”冲突

Reduce ① 归约项目: 在最后时需要归约 ($A \rightarrow \alpha \cdot$)

Shift ② 移进项目: 后面是终结符时需要移进 ($A \rightarrow \alpha \cdot x \beta$)

③ 待约项目: 后面是非终结符时等待被归约 ($A \rightarrow \alpha \cdot X \beta$)

Accept ④ 接收项目: 在开始产生式最右时, 为接收状态 ($S \rightarrow \alpha \cdot$)

① 拓广文法 ② 闭项目 ③ DFA ④ 分析表

与 LR(0) 一样

是 SLR(1) 文法

① $G[E] = E \rightarrow E+T \mid T$

$T \rightarrow F \mid F$

$F \rightarrow (E) \mid a$

$G'[E] = (0) S' \rightarrow E$

(1) $E \rightarrow E+T$

(2) $E \rightarrow T$

(3) $T \rightarrow F*$

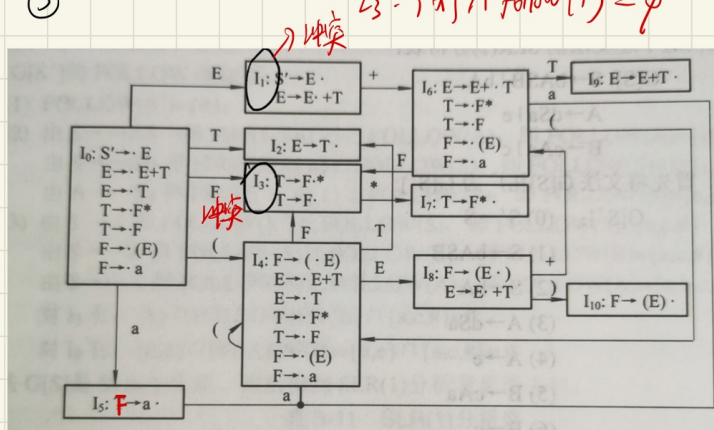
(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow a$

$Z_1: \{+\} \cap \text{Follow}(S') = \emptyset$

$Z_3: \{*\} \cap \text{Follow}(T) = \emptyset$



④ 求 $G'(E)$ 的 Follow 集

$\text{Follow}(S') = \{\#\}$

$\text{Follow}(E) = \{+,), \#\}$

$\text{Follow}(T) = \{+,), \#\}$

$\text{Follow}(F) = \{+, *,), \#\}$

在需要填 $R(x)$ 的行里, 只填 Follow 集内有的终结符的格子
 eg. $R(2), x=2, (2): E \rightarrow T, \text{Follow}(E) = \{+,), \#\}$

状态									
	a	(*	()	#	E	T	F
0	S_5			S_4			1	2	3
1		S_6				a_4			
2		r_2			r_2	r_2			
3			S_7						
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5		S_3	S_4				9	3
7		r_3			r_3	r_3			
8		S_6			S_{10}				
9		r_1			r_1	r_1			
10		r_5	r_5		r_5	r_5			

题: SRL(1)文法综合

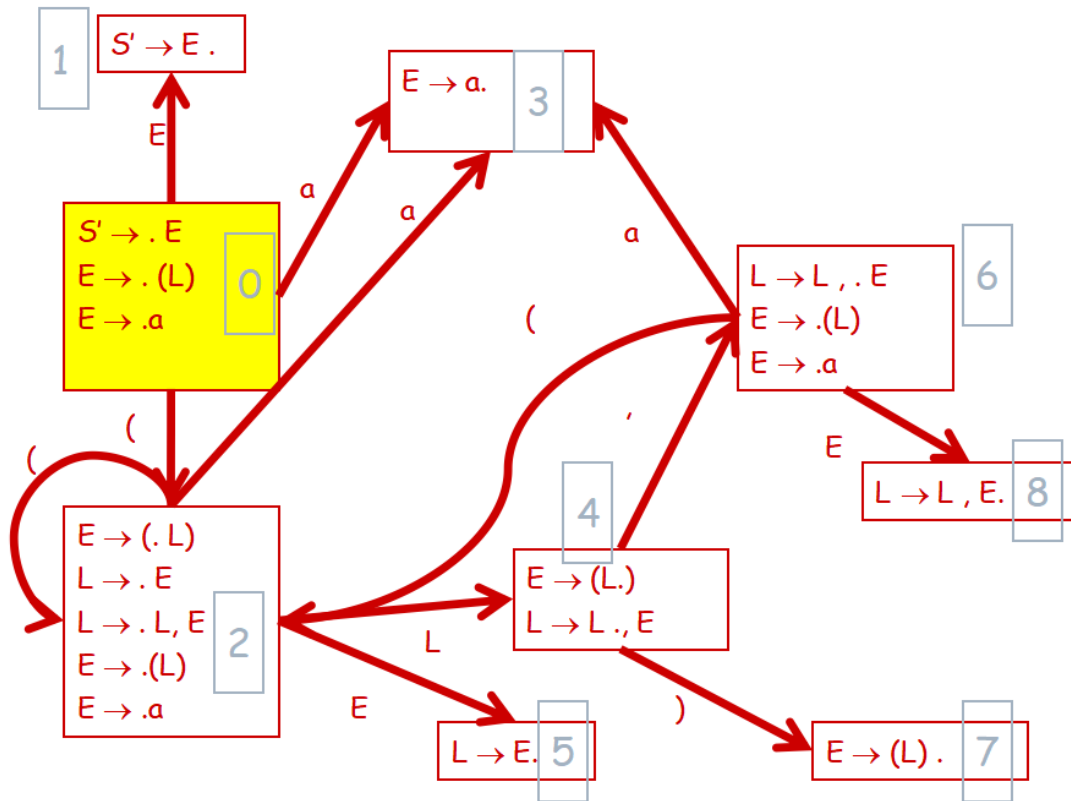
5.1 Consider the following grammar:

$E \rightarrow (L) \mid a$

$L \rightarrow L, E \mid E$

- Construct the DFA of LR(0) items for this grammar.
- Construct the general SLR(1) parsing table.
- Show the parsing stack and the actions of an SLR(1) parse for the input string $((a), a, (a, a))$
- Is this grammar an LR(0) grammar? If not, describe the LR(0) conflict, if so, construct the LR(0) parsing table, and descrie how a parse might differ from an SLR(1) parse.

a.



b. Construct the general SLR(1) parsing table.

Follow(E)={ \$, , ,) }

Follow(L)={ , ,) }

State	Input					Goto	
	(a)	,	\$	L	E
0	S2	S3					1
1					Accept		
2	S2	S3				4	5
3			r(E→a)	r(E→a)	r(E→a)		
4			S7	S6			
5			r(L→E)	r(L→E)			
6	S2	S3					8
7			r(E→L)	r(E→L)	r(E→L)		
8			r(L→L,E)	r(L→L,E)			

c. Show the parsing stack and the actions of an SLR(1) parse for the input string ((a),a,(a,a))

	Parsing stack	Input	Action
1	\$0	((a),a,(a,a)) \$	shift 2
2	\$0 (2	(a),a,(a,a)) \$	shift 2
3	\$0(2(2	a),a,(a,a)) \$	shift 3
4	\$0(2(2a3),a,(a,a)) \$	reduce $E \rightarrow a$
5	\$0(2(2E5),a,(a,a)) \$	reduce $L \rightarrow E$
6	\$0(2(2L4),a,(a,a)) \$	shift 7
7	\$0(2(2L4)7	,a,(a,a)) \$	reduce $E \rightarrow (L)$
8	\$0(2E5	,a,(a,a)) \$	reduce $L \rightarrow E$
9	\$0(2L4	,a,(a,a)) \$	shift 6
10	\$0(2L4,6	a,(a,a)) \$	shift 3
11	\$0(2L4,6a3	,a,a)) \$	reduce $E \rightarrow a$
12	\$0(2L4,6E8	,a,a)) \$	reduce $L \rightarrow L,E$
13	\$0(2L4	,a,a)) \$	shift 6
14	\$0(2L4,6	(a,a)) \$	shift 2
15	\$0(2L4,6(2	a,a)) \$	shift 3
16	\$0(2L4,6(2a3	,a)) \$	reduce $E \rightarrow a$
17	\$0(2L4,6(2E5	,a)) \$	reduce $L \rightarrow E$
18	\$0(2L4,6(2L4	,a)) \$	shift 6
19	\$0(2L4,6(2L4,6	a)) \$	shift 3
20	\$0(2L4,6(2L4,6a3)) \$	reduce $E \rightarrow a$
21	\$0(2L4,6(2L4,6E8)) \$	reduce $L \rightarrow L,E$
22	\$0(2L4,6(2L4)) \$	shift 7
23	\$0(2L4,6(2L4)7) \$	reduce $E \rightarrow (L)$
24	\$0(2L4,6E8) \$	reduce $L \rightarrow L,E$
25	\$0(2L4) \$	shift 7
26	\$0(2L4)7	\$	reduce $E \rightarrow (L)$
27	\$0E1	\$	accept

d. Is this grammar an LR(0) grammar? If not, describe the LR(0) conflict, if so, construct the LR(0) parsing table, and describe how a parse might differ from an SLR(1) parse.

This is a LR(0) grammar because in no shift reduce conflict and reduce reduce conflict.

State	Action	Rule	Input				<u>Goto</u>	
			(a)	,	E	L
0	shift	$E' \rightarrow E$	2	3			1	
1	reduce							
2	shift	$E \rightarrow a$	2	3			5	4
3	reduce							
4	shift	$L \rightarrow E$			7	6		
5	reduce							
6	shift	$E \rightarrow (L)$	2	3			8	
7	reduce							
8	reduce	$L \rightarrow L, E$						

LR(1)文法

构造分析表

LR(1) 的构造

① 拓广文法 ② 带向前搜索符的 DFA ③ LR(1) 分析表

① G[S]: $S \rightarrow BB$
 $B \rightarrow aB \mid b$

G'[S]: (0) $S' \rightarrow S$ $Fo(S') = \{ \# \}$
(1) $S \rightarrow BB$
(2) $B \rightarrow aB$ $Fi(B) = \{ a, b \}$
(3) $B \rightarrow b$

② look ahead DFA

$I_0: S' \rightarrow \cdot S, \#$
 $S \rightarrow \cdot BB, \#$
 $B \rightarrow \cdot aB, a/b$ $a \text{ 或者 } b$
 $B \rightarrow \cdot b, a/b$

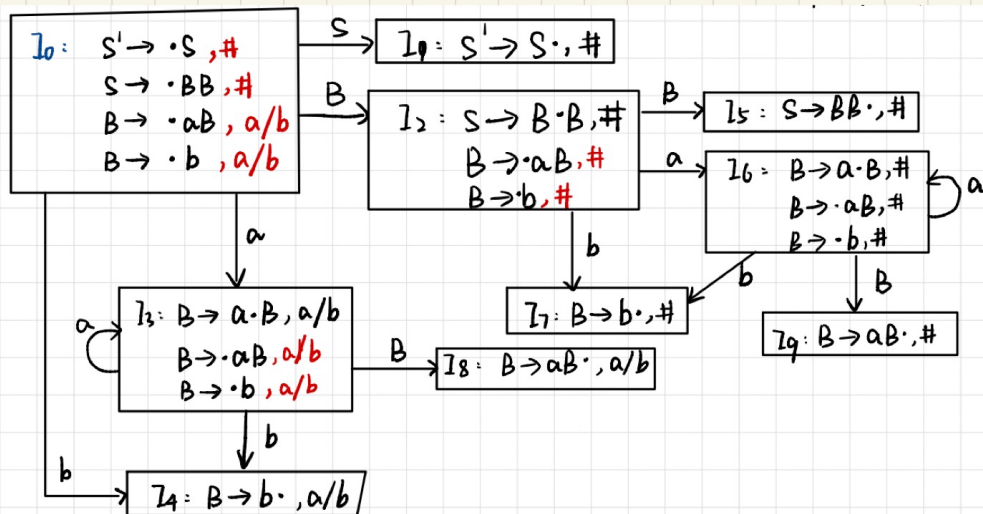
LR(1) 不能随意合并状态
look ahead 每个状态要重新判断
(除了第 1 个式子)

look ahead 求法
① Follow(S')

② 先看前再看后

eg. $A \rightarrow a \cdot B \beta$, a ① $\beta = \phi$, 照抄
 $B \rightarrow \sim$ ② $\beta \neq \phi$, $Fi(B)$
加) look ahead

eg. G'[S]: $I_0: S' \rightarrow \cdot S, \#$
 $S' \rightarrow S$ $S \rightarrow \cdot A, \#$
 $S \rightarrow A$ $A \rightarrow \cdot BA, \#$
 $A \rightarrow BA$ $A \rightarrow \cdot \#, \#$
 $A \rightarrow \epsilon$ $B \rightarrow \cdot aB, a/b/\#$
 $B \rightarrow aB$ $B \rightarrow \cdot b, a/b/\#$
 $B \rightarrow b$



③ 分析表

$I_4: B \rightarrow b \cdot, a/b$

	ACTION			GOTO	
	a	b	#	S	B
0	S ₃	S ₄		1	2
1			acc		
2	S ₆	S ₇			5
3	S ₃	S ₄			8
④	r ₃	r ₃			
5			r ₁		
6	S ₆	S ₇			9
7			r ₃		
8	r ₂	r ₂			
9			r ₂		

(0) $S' \rightarrow S$

(1) $S \rightarrow BB$

(2) $B \rightarrow aB$

(3) $B \rightarrow b$

~~LX(0)~~

~~$S' \rightarrow S$~~

LALR(1)文法

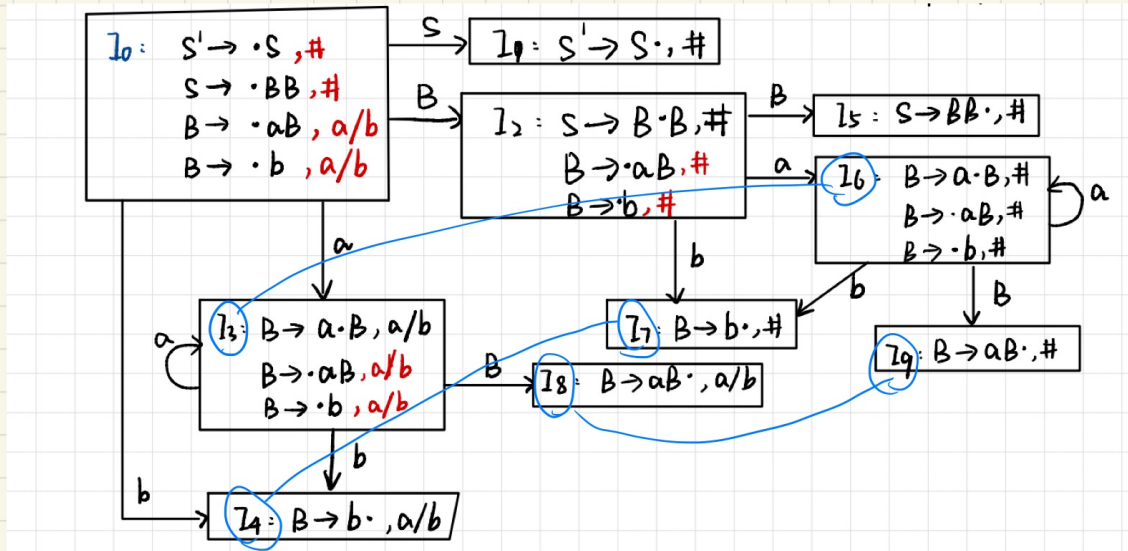
构造分析表

LALR(1) 构造

Look-Ahead

① 推广文法 ② lookahead DFA ③ 分析表

合并同心集 (表达式相同, lookahead 不同)



I_3, I_6 合并: I_{36} : $B \rightarrow a \cdot B, a/b \#$
 $B \rightarrow \cdot aB, a/b \#$
 $B \rightarrow \cdot b, a/b \#$

I_{47} $B \rightarrow b \cdot, a/b \#$
 I_{29} $B \rightarrow aB \cdot, a/b \#$

③ 构造分析表

与 LR(1) 一致

题: LALR(1) 文法综合

注: a, b 小题是 LR(1), c, d 小题是 LALR(1)

5.2 Consider the following grammar:

$$E \rightarrow (L) \mid a$$

$$L \rightarrow L, E \mid E$$

- Construct the DFA of LR(1) items for this grammar.
- Construct the general LR(1) parsing table.
- Construct the DFA of LALR(1) items for this grammar.
- Construct the LALR(1) parsing table.
- Describe any difference that might occur between the actions of a general LR(1) parser and an LALR(1) parser.

[Solution]

Augment the grammar by adding the production: $E' \rightarrow E$

a.

State 0: $[E' \rightarrow \cdot E, \$]$

$[E \rightarrow \cdot (L), \$]$

$[E \rightarrow \cdot a, \$]$

State 2: $[E \rightarrow (L) \cdot, \$]$

$[L \rightarrow \cdot L, E, \cdot]$

$[L \rightarrow \cdot E, \cdot]$

$[L \rightarrow \cdot L, E, \cdot, \cdot]$

$[L \rightarrow \cdot E, \cdot, \cdot]$

$[E \rightarrow \cdot (L), \cdot]$

$[E \rightarrow \cdot a, \cdot]$

$[E \rightarrow \cdot (L), \cdot, \cdot]$

$[E \rightarrow \cdot a, \cdot, \cdot]$

State 1: $[E' \rightarrow E \cdot, \$]$

State 3: $[E \rightarrow a \cdot, \$]$

State 4: $[E \rightarrow (L) \cdot, \$]$

$[L \rightarrow L \cdot, E, \cdot]$

$[L \rightarrow L \cdot, E, \cdot, \cdot]$

State 5: $[L \rightarrow E \cdot, \cdot]$

$[L \rightarrow E \cdot, \cdot]$

State 6: $[E \rightarrow (L) \cdot, \cdot]$

$[E \rightarrow (L) \cdot, \cdot, \cdot]$

$[L \rightarrow \cdot L, E, \cdot]$

$[L \rightarrow \cdot E, \cdot]$

$[L \rightarrow \cdot L, E, \cdot, \cdot]$

$[L \rightarrow \cdot E, \cdot, \cdot]$

$[E \rightarrow \cdot (L), \cdot]$

$[E \rightarrow \cdot a, \cdot]$

$[E \rightarrow \cdot (L), \cdot, \cdot]$

$[E \rightarrow \cdot a, \cdot, \cdot]$

State 7: $[E \rightarrow a \cdot, \cdot]$

$[E \rightarrow a \cdot, \cdot]$

State 8: $[E \rightarrow (L) \cdot, \cdot]$

State 9: $[L \rightarrow L \cdot, E, \cdot]$

$[E \rightarrow \cdot (L), \cdot]$

$[E \rightarrow \cdot a, \cdot]$

$[L \rightarrow L \cdot, E, \cdot, \cdot]$

$[E \rightarrow \cdot (L), \cdot, \cdot]$

$[E \rightarrow \cdot a, \cdot, \cdot]$

State 10: $[E \rightarrow (L) \cdot, \cdot]$

$[E \rightarrow (L) \cdot, \cdot, \cdot]$

$[L \rightarrow L \cdot, E, \cdot]$

$[L \rightarrow L \cdot, E, \cdot, \cdot]$

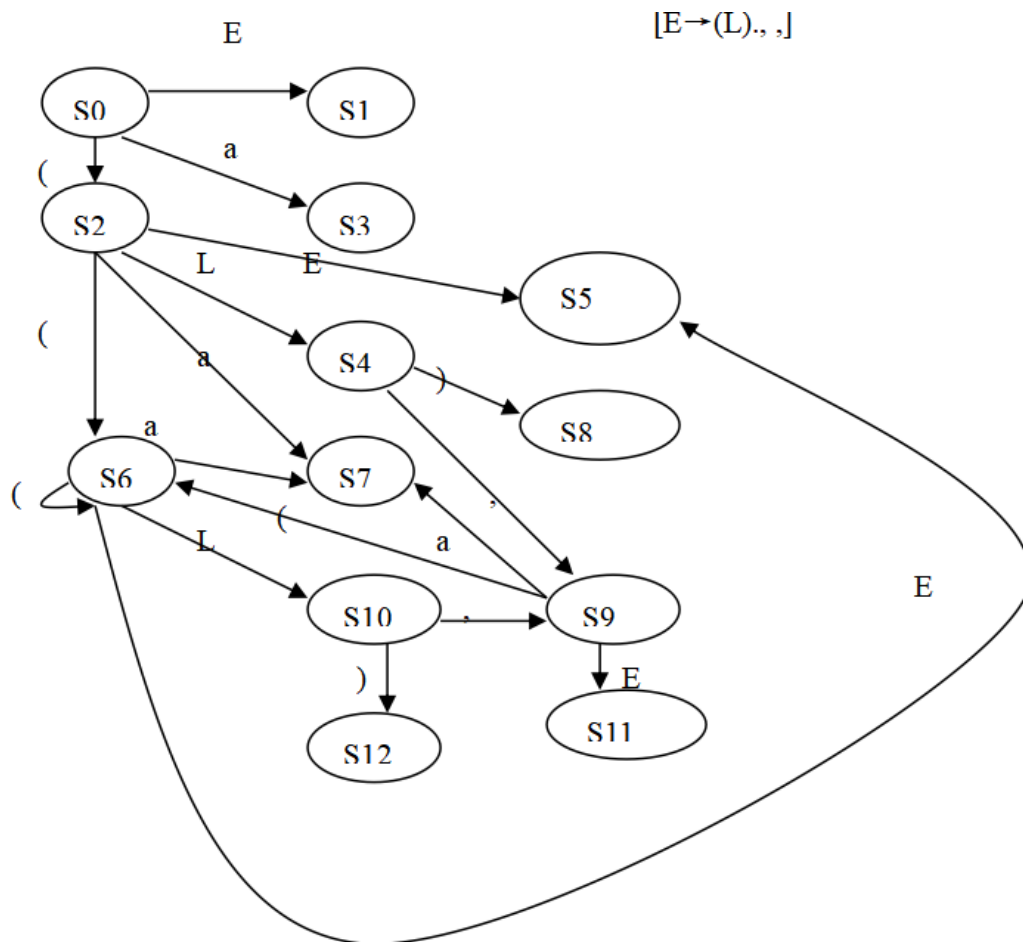
State 11: $[L \rightarrow L \cdot, E, \cdot]$

$[L \rightarrow L \cdot, E, \cdot, \cdot]$

State 12: $[E \rightarrow (L) \cdot, \cdot]$

$[E \rightarrow (L) \cdot, \cdot, \cdot]$

E



b. $r1: E \rightarrow (L)$ $r2: E \rightarrow a$ $r3: L \rightarrow L, E$ $r4: L \rightarrow E$

State	Input					Goto	
	(a)	,	\$	L	E
0	S2	S3					1
1					Accept		
2	S6	S7				4	5
3					r2		
4			S8	S9			

5			r4	r4			
6	S6	S7				10	5
7			r2	r2			
8					r1		
9	S6	S7					11
10			S12	S9			
11			r3	r3			
12			r1	r1			

c.

State 0: $[E' \rightarrow .E, \$]$
 $[E \rightarrow .(L), \$]$
 $[E \rightarrow .a, \$]$

State 1: $[E' \rightarrow E., \$]$

State 2/6: $[E \rightarrow (L), \$ /)/,]]$
 $[L \rightarrow .L, E ,)]$
 $[L \rightarrow .E ,)]$
 $[L \rightarrow .L, E , ,]]$
 $[L \rightarrow .E , ,]]$
 $[E \rightarrow .(L),)]$
 $[E \rightarrow .a,)]$
 $[E \rightarrow .(L), ,]]$
 $[E \rightarrow .a, ,]]$

State 3/7: $[E \rightarrow a., \$ /)/,]]$

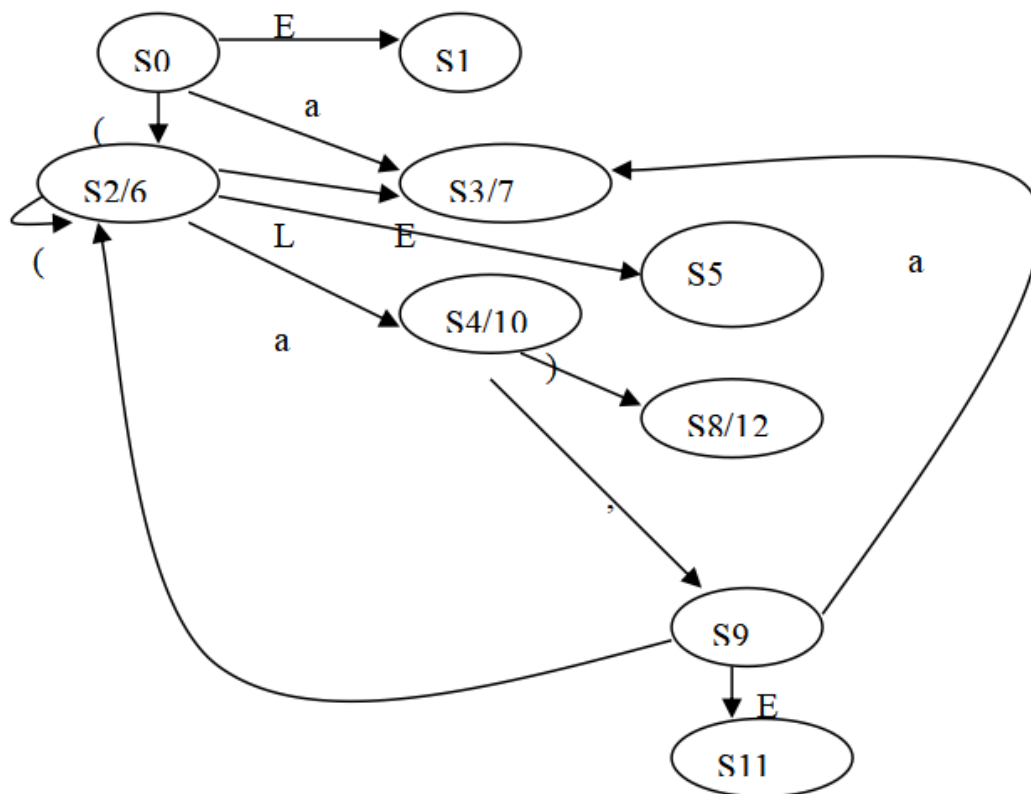
State 4/10: $[E \rightarrow (L.), \$ /)/,]]$
 $[L \rightarrow L., E ,)]$
 $[L \rightarrow L., E , ,]]$

State 5: $[L \rightarrow E.,)]$
 $[L \rightarrow E., ,]]$

State 8/12: $[E \rightarrow (L.), \$ /)/,]]$

State 9: $[L \rightarrow L., E ,)]$
 $[E \rightarrow .(L),)]$
 $[E \rightarrow .a,)]$
 $[L \rightarrow L., E , ,]]$
 $[E \rightarrow .(L), ,]]$
 $[E \rightarrow .a, ,]]$

State 11: $[L \rightarrow L., E.,)]$
 $[L \rightarrow L., E., ,]]$



d. $r1: E \rightarrow (L)$ $r2: E \rightarrow a$ $r3: L \rightarrow L, E$ $r4: L \rightarrow E$

State	Input					Goto	
	(A)	,	\$	L	E
0	S2/6	S3/7					1
1					Accept		
2/6	S2/6	S3/7				4/10	5
3/7			r2	r2	r2		
4/10			S8/12	S9			
5			r4	r4			
8/12			r1	r1	r1		
9	S2/6	S3/7					11
10			S8/12	S9			
11			r3	r3			

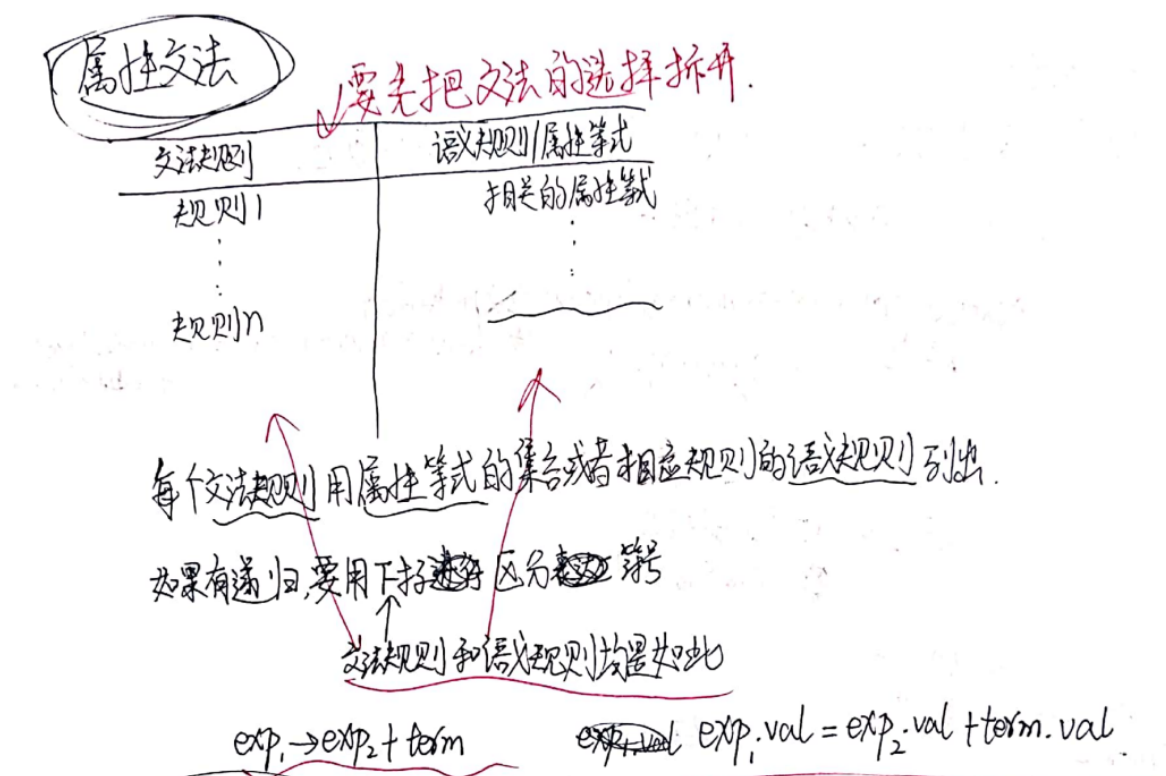
e. The consequence of using LALR(1) parsing over general LR(1) parsing is that , in the presence of errors, some spurious reduction may be made before error is declared. (Page 225)

e. 使用 LALR(1) 解析而不是一般的 LR(1) 解析的结果是，在存在错误的情况下，可能会在声明错误之前进行一些虚假的规约。

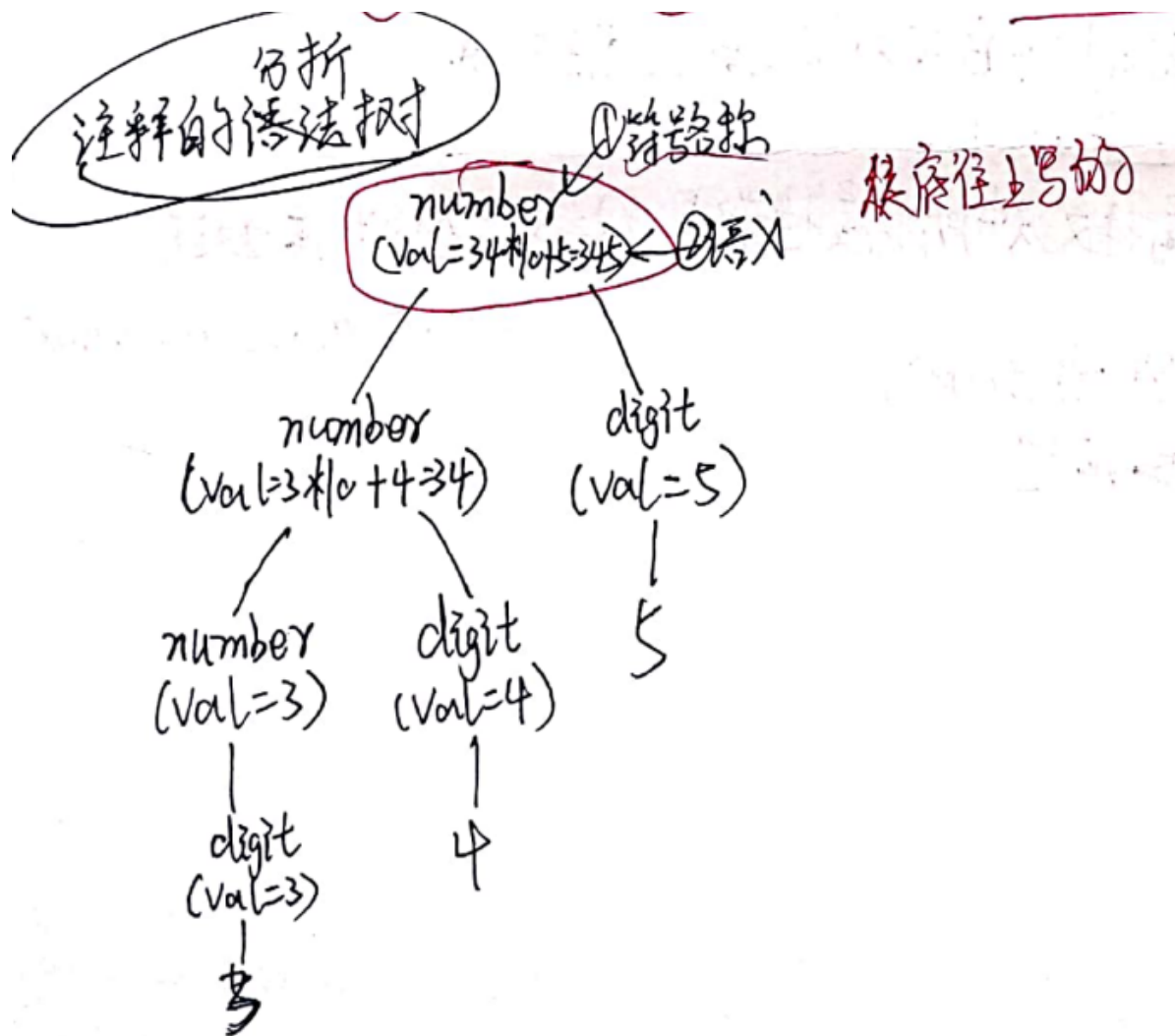
语义分析

属性文法

构造属性文法



分析树



题：构造属性文法

Example 6.1 consider the following simple grammar for unsigned numbers:

$\text{number} \rightarrow \text{number digit} \mid \text{digit}$

$\text{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

the most significant attribute: numeric value (write as val) , and the responding attribute grammar is as follows:

Grammar Rule	Semantic Rules
$\text{Number1} \rightarrow \text{number2 digit}$	$\text{number1.val} = \text{number2.val} * 10 + \text{digit.val}$
$\text{Number} \rightarrow \text{digit}$	$\text{number.val} = \text{digit.val}$
$\text{digit} \rightarrow 0$	$\text{digit.val} = 0$
$\text{digit} \rightarrow 1$	$\text{digit.val} = 1$
$\text{digit} \rightarrow 2$	$\text{digit.val} = 2$
$\text{digit} \rightarrow 3$	$\text{digit.val} = 3$
$\text{digit} \rightarrow 4$	$\text{digit.val} = 4$
$\text{digit} \rightarrow 5$	$\text{digit.val} = 5$
$\text{digit} \rightarrow 6$	$\text{digit.val} = 6$
$\text{digit} \rightarrow 7$	$\text{digit.val} = 7$
$\text{digit} \rightarrow 8$	$\text{digit.val} = 8$
$\text{digit} \rightarrow 9$	$\text{digit.val} = 9$

Example 6.2 consider the following grammar for simple integer arithmetic expressions:

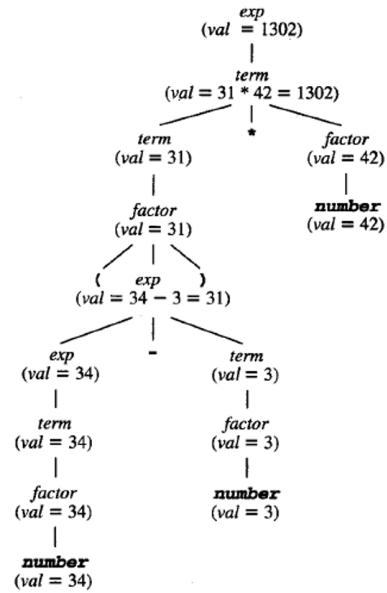
$$\begin{aligned} \text{exp} &\rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term} \\ \text{term} &\rightarrow \text{term} * \text{factor} \mid \text{factor} \\ \text{factor} &\rightarrow (\text{exp}) \mid \text{number} \end{aligned}$$

文法规则	语义规则
$\text{exp}_1 \rightarrow \text{exp}_2 + \text{term}$	$\text{exp}_1.\text{val} = \text{exp}_2.\text{val} + \text{term}.\text{val}$
$\text{exp}_1 \rightarrow \text{exp}_2 - \text{term}$	$\text{exp}_1.\text{val} = \text{exp}_2.\text{val} - \text{term}.\text{val}$
$\text{exp} \rightarrow \text{term}$	$\text{exp}.\text{val} = \text{term}.\text{val}$
$\text{term}_1 \rightarrow \text{term}_2 * \text{factor}$	$\text{term}_1.\text{val} = \text{term}_2.\text{val} * \text{factor}.\text{val}$
$\text{term} \rightarrow \text{factor}$	$\text{term}.\text{val} = \text{factor}.\text{val}$
$\text{factor} \rightarrow (\text{exp})$	$\text{factor}.\text{val} = \text{exp}.\text{val}$
$\text{factor} \rightarrow \text{number}$	$\text{factor}.\text{val} = \text{number}.\text{val}$

题：分析树

通过在语法树的节点上附加等式来表示属性文法包含的计算。例如，给定表达式 $(34-3) * 42$ ，可以用在其语法树上值的语义来表达，如图6 - 2所示。

文法规则	语义规则
$\text{exp}_1 \rightarrow \text{exp}_2 + \text{term}$	$\text{exp}_1.\text{val} = \text{exp}_2.\text{val} + \text{term}.\text{val}$
$\text{exp}_1 \rightarrow \text{exp}_2 - \text{term}$	$\text{exp}_1.\text{val} = \text{exp}_2.\text{val} - \text{term}.\text{val}$
$\text{exp} \rightarrow \text{term}$	$\text{exp}.\text{val} = \text{term}.\text{val}$
$\text{term}_1 \rightarrow \text{term}_2 * \text{factor}$	$\text{term}_1.\text{val} = \text{term}_2.\text{val} * \text{factor}.\text{val}$
$\text{term} \rightarrow \text{factor}$	$\text{term}.\text{val} = \text{factor}.\text{val}$
$\text{factor} \rightarrow (\text{exp})$	$\text{factor}.\text{val} = \text{exp}.\text{val}$
$\text{factor} \rightarrow \text{number}$	$\text{factor}.\text{val} = \text{number}.\text{val}$



题：有数据类型的文法

Example 6.3 consider the following simple grammar of variable declarations in a C-like syntax:

decl \rightarrow type var-list
type \rightarrow int|float
var-list \rightarrow id, var-list|id

Define a data type attribute for the variables given by the identifiers in a declaration and write equations expressing how the data type attribute is related to the type of the declaration as follows:

(we use the name *dtype* to distinguish the attribute from the nonterminal type)

Grammar Rule	Semantic Rules
decl \rightarrow type var-list	var-list.dtype = type.dtype
type \rightarrow int	type.dtype = integer
type \rightarrow float	type.dtype = real
var-list1 \rightarrow id, var-list2	id.dtype = var-list1.dtype var-list2.dtype= var-list1.dtype
var-list \rightarrow id	id.type = var-list.dtype

题：有不同进制的文法

注意 if 语句错误判断

Example 6.4 consider the following grammar, where number numbers may be octal or decimal, suppose this is indicated by a one-character suffix o(for octal) or d(for decimal):

Based-num \rightarrow Num basechar
Basechar \rightarrow o|d
Num \rightarrow Num digit | digit
Digit \rightarrow 0|1|2|3|4|5|6|7|8|9

In this case num and digit require a new attribute base, which is used to compute t val attribute. The attribute grammar for base and val is given as follows.

Grammar Rule	Semantic Rules
Based-num \rightarrow num basechar	based-num.val = num. val Num.base = basechar.base
Basechar \rightarrow o	basechar.base = 8
Basechar \rightarrow d	basechar.base = 10

Based-num → num basechar

Basechar → o

Basechar → d

Num1 → num2 digit

Num → digit

Digit → 0

Digit → 1

...

Digit → 7

Digit → 8

Digit → 9

based-num.val = num.val

Num.base = basechar.base

basechar.base = 8

basechar.base = 10

num1.val =

If digit.val = error or num2.val = error

Then error

Else num2.val * num1.base + digit.val

Num2.base = num1.base

Digit.base = num1.base

num.val = digit.val

Digit.base = num.base

digit.val = 0

digit.val = 1

...

digit.val = 7

digit.val = if digit.vase = 8 then error else 8

digit.val = if digit.vase = 8 then error else 9]
