第二章 词法分析

2.3 叙述由下列正规式描述的语言

(a) $0(0|1)^*0$

［解答］

该正规式所描述的语言为：在字母表{0, 1}上，首字符为 0，尾字符为 0，中间由零个或多个 0 或 1 所组成的字符串。

(b) ((ε|0)1*))*

［解答］

该正规式所描述的语言为：在字母表{0, 1}上，由 0 和 1 组成的所有字符串，包括空串。

2.4 为下列语言写正规定义

C 语言的注释，即以 /* 开始和以 */ 结束的任意字符串，但它的任何前缀（本身除外）不以 */ 结尾。

［解答］

other → a | b | …　　*other* 指除了*以外 C 语言中的其它字符

$other_1$ → a | b | …　　$other_1$ 指除了*和/以外 C 语言中的其它字符

comment → /* *other** (* ** $other_1$ *other**)* ** */

(f) 由偶数个 0 和偶数个 1 构成的所有 0 和 1 的串。

［解答］

由题目分析可知，一个符号串由 0 和 1 组成，则 0 和 1 的个数只能有四种情况：

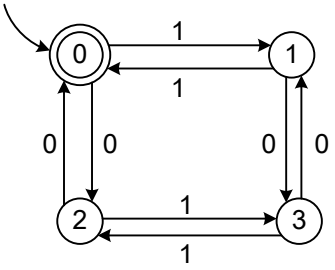
- 偶数个 0 和偶数个 1（用状态 0 表示）；
- 偶数个 0 和奇数个 1（用状态 1 表示）；
- 奇数个 0 和偶数个 1（用状态 2 表示）；
- 奇数个 0 和奇数个 1（用状态 3 表示）；

所以，

- 状态 0（偶数个 0 和偶数个 1）读入 1，则 0 和 1 的数目变为：偶数个 0 和奇数个 1（状态 1）
- 状态 0（偶数个 0 和偶数个 1）读入 0，则 0 和 1 的数目变为：奇数个 0 和偶数个 1（状态 2）
- 状态 1（偶数个 0 和奇数个 1）读入 1，则 0 和 1 的数目变为：偶数个 0 和偶数个 1（状态 0）
- 状态 1（偶数个 0 和奇数个 1）读入 0，则 0 和 1 的数目变为：奇数个 0 和奇数个 1（状态 3）
- 状态 2（奇数个 0 和偶数个 1）读入 1，则 0 和 1 的数目变为：奇数个 0 和奇数个 1（状态 3）

- 状态 2（奇数个 0 和偶数个 1）读入 0，则 0 和 1 的数目变为：偶数个 0 和偶数个 1（状态 0）
- 状态 3（奇数个 0 和奇数个 1）读入 1，则 0 和 1 的数目变为：奇数个 0 和偶数个 1（状态 2）
- 状态 3（奇数个 0 和奇数个 1）读入 0，则 0 和 1 的数目变为：偶数个 0 和奇数个 1（状态 1）

因为，所求为由偶数个 0 和偶数个 1 构成的所有 0 和 1 的串，故状态 0 既为初始状态又为终结状态，其状态转换图如下所示：



由此可以写出其正规文法为：

$S_0 \rightarrow 1S_1 \mid 0S_2 \mid \varepsilon \quad S_1 \rightarrow 1S_0 \mid 0S_3 \mid 1$

$S_2 \rightarrow 1S_3 \mid 0S_0 \mid 0 \quad S_3 \rightarrow 1S_2 \mid 0S_1$

在不考虑 $S_0 \rightarrow \varepsilon$ 产生式的情况下，可以将文法变形为：

$S_0 = 1S_1 + 0S_2 \qquad S_1 = 1S_0 + 0S_3 + 1$

$S_2 = 1S_3 + 0S_0 + 0 \qquad S_3 = 1S_2 + 0S_1$

所以：

$S_0 = (00|11)\ S_0 + (01|10)\ S_3 + 11 + 00 \qquad (1)$

$S_3 = (00|11)\ S_3 + (01|10)\ S_0 + 01 + 10 \qquad (2)$

解(2)式得：

$S_3 = (00|11)^*\ ((01|10)\ S_0 + (01|10))$

代入(1)式得：

$S_0 = (00|11)\ S_0 + (01|10)\ (00|11)^*((01|10)\ S_0 + (01|10)) + (00|11)$

$\Rightarrow S_0 = ((00|11) + (01|10)\ (00|11)^*(01|10))S_0 + (01|10)\ (00|11)^*(01|10) + (00|11)$

$\Rightarrow S_0 = ((00|11)|(01|10)\ (00|11)^*(01|10))^*((00|11) + (01|10)\ (00|11)^*\ (01|10))$

$\Rightarrow S_0 = ((00|11)|(01|10)\ (00|11)^*\ (01|10))^+$

因为 $S_0 \rightarrow \varepsilon$ 所以由偶数个 0 和偶数个 1 构成的所有 0 和 1 的串的正规定义为：

$S_0 \rightarrow ((00|11)|(01|10)\ (00|11)^*\ (01|10))^*$

(g) 由偶数个 0 和奇数个 1 构成的所有 0 和 1 的串。

［解答］

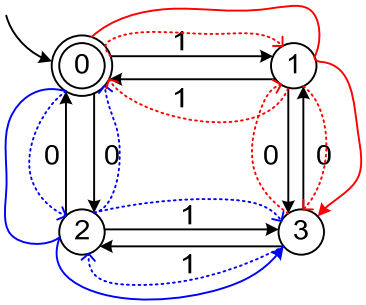
此题目我们可以借鉴上题的结论来进行处理。

对于由偶数个 0 和奇数个 1 构成的所有 0 和 1 的串，我们分情况讨论：

(1) 若符号串首字符为 0，则剩余字符串必然是奇数个 0 和奇数个 1，因此我们必须在上题偶数个 0 和偶数个 1 的符号串基础上再读入 10（红色轨迹）或 01（蓝色轨迹），又因为在 0→1 和 1→3 的过程中可以进行多次循环（红色虚线轨迹），同理 0→2 和 2→3（蓝色虚线轨迹），所以还必须增加符号串(00|11)*，我们用 S_0 表示偶数个 0 和偶数个 1，用 S 表示偶数个 0 和奇数个 1 则其正规定义为：

$S \rightarrow 0(00|11)^*(01|10)\ S_0$

$S_0 \rightarrow ((00|11)|(01|10)\ (00|11)^*\ (01|10))^*$



(2) 若符号串首字符为 1，则剩余字符串必然是偶数个 0 和偶数个 1，其正规定义为：

$S \rightarrow 1S_0$

$S_0 \rightarrow ((00|11)|(01|10)\ (00|11)^*\ (01|10))^*$

综合(1)和(2)可得，偶数个 0 和奇数个 1 构成的所有 0 和 1 串其正规定义为：

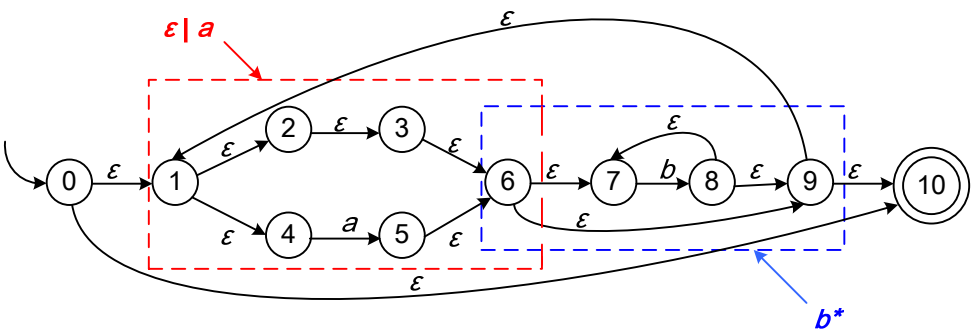
$S \rightarrow 0(00|11)^*(01|10)\ S_0|1S_0$

$S_0 \rightarrow ((00|11)|(01|10)\ (00|11)^*\ (01|10))^*$

2.7 用算法 2.4 为下列正规式构造非确定的有限自动机，给出它们处理输入串 ababbab 的状态转换序列。

(c) ((ε | *a*) *b)***

［解答］



根据算法 2.4 构造该正规式所对应的 *NFA*，如图所示。

则输入串 *ababbab* 的状态转换序列为：

0 → 1 → 4 → 5 → 6 → 7 → 8 → 9 → 1 → 4 → 5 → 6 → 7 → 8

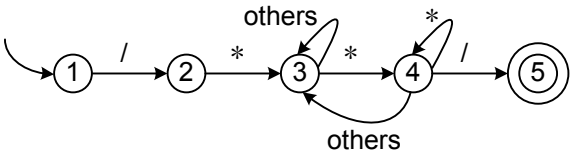
a *b* *a* *b*

→ 7 → 8 → 9 → 1 → 4 → 5 → 6 → 7 → 8 → 9 → 10

b *a* *b*

2.10 C 语言的注释是以 /* 开始和以 */ 结束的任意字符串，但它的任何前缀（本身除外）不以 */ 结尾。画出接受这种注释的 *DFA* 的状态转换图。

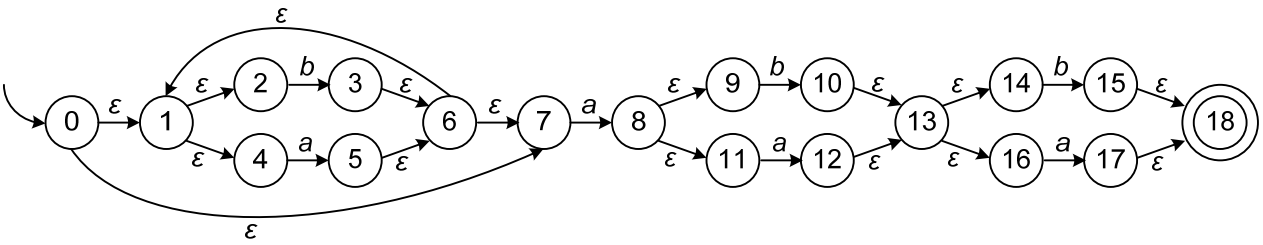
［解答］



2.12 为下列正规式构造最简的 *DFA*

(b) (*a*|*b*)* *a* (*a*|*b*) (*a*|*b*)

［解答］



(1) 根据算法 2.4 构造该正规式所对应的 *NFA*，如图所示。

(2) 根据算法 2.2（子集法）将 *NFA* 转换成与之等价的 *DFA*（确定化过程）

初始状态

S_0 = *ε-closure*(0) = {0, 1, 2, 4, 7}

标记状态 S_0

S_1 = *ε-closure*(move(S_0, *a*)) = *ε-closure*({5, 8}) = {1, 2, 4, 5, 6, 7, 8, 9, 11}

S_2 = *ε-closure*(move(S_0, *b*)) = *ε-closure*({3}) = {1, 2, 3, 4, 6, 7}

标记状态 S_1

S_3 = *ε-closure*(move(S_1, *a*)) = *ε-closure*({5, 8, 12}) = {1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16}

S_4 = *ε-closure*(move(S_1, b)) = *ε-closure*({3, 10}) = {1, 2, 4, 5, 6, 7, 10, 13, 14, 16}

标记状态 S_2

S_1 = *ε-closure*(move(S_2, a)) = *ε-closure*({5, 8}) = {1, 2, 4, 5, 6, 7, 8, 9, 11}

S_2 = *ε-closure*(move(S_2, b)) = *ε-closure*({3}) = {1, 2, 3, 4, 6, 7}

标记状态 S_3

S_5 = *ε-closure*(move(S_3, a)) = *ε-closure*({5, 8, 12, 17}) = {1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 18}

S_6 = *ε-closure*(move(S_3, b)) = *ε-closure*({3, 10, 15}) = {1, 2, 4, 5, 6, 7, 10, 13, 14, 15, 16, 18}

标记状态 S_4

S_7 = *ε-closure*(move(S_4, *a*)) = *ε-closure*({5, 8, 17}) = {1, 2, 4, 5, 6, 7, 8, 9, 11, 17, 18}

S_8 = *ε-closure*(move(S_4, *b*)) = *ε-closure*({3, 15}) = {1, 2, 3, 4, 6, 7, 15, 18}

标记状态 S_5

S_5 = *ε-closure*(move(S_5, a)) = *ε-closure*({5, 8, 12, 17}) = {1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 18}

S_6 = *ε-closure*(move(S_5, *b*)) = *ε-closure*({3, 10, 15}) = {1, 2, 4, 5, 6, 7, 10, 13, 14, 15, 16, 18}

标记状态 S_6

S_7 = *ε-closure*(move(S_6, a)) = *ε-closure*({5, 8, 17}) = {1, 2, 4, 5, 6, 7, 8, 9, 11, 17, 18}

S_8 = *ε-closure*(move(S_6, *b*)) = *ε-closure*({3, 15}) = {1, 2, 3, 4, 6, 7, 15, 18}

标记状态 S_7

S_3 = *ε-closure*(move(S_7, a)) = *ε-closure*({5, 8, 12}) = {1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16}

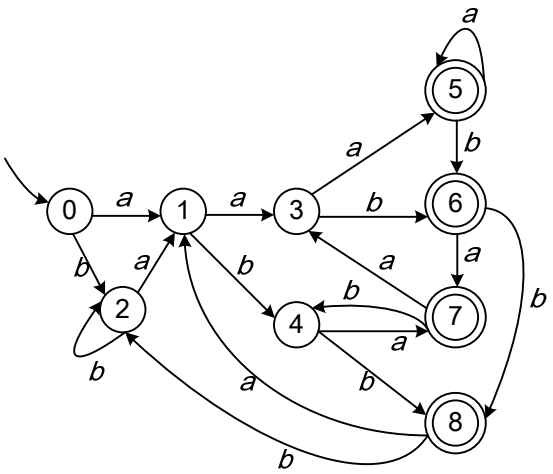
S_4 = *ε-closure*(move(S_7, b)) = *ε-closure*({3, 10}) = {1, 2, 4, 5, 6, 7, 10, 13, 14, 16}

标记状态 S_8

S_1 = *ε-closure*(move(S_8, *a*)) = *ε-closure*({5, 8}) = {1, 2, 4, 5, 6, 7, 8, 9, 11}

S_2 = *ε-closure*(move(S_8, *b*)) = *ε-closure*({3}) = {1, 2, 3, 4, 6, 7}

由以上可知，确定化后的 *DFA* 的状态集合 $S = \{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8\}$，输入符号集合 $\Sigma = \{a, b\}$，状态转换函数 move 如上，S_0 为开始状态，接收状态集合 $F = \{S_5, S_6, S_7, S_8\}$，其状态转换图如下所示：



(3) 根据算法 2.3 过将 *DFA* 最小化

第一次划分：{S_0, S_1, S_2, S_3, S_4}　　{S_5, S_6, S_7, S_8}

{S_0, S_1, S_2, S_3, S_4}*a* = {S_1, S_3, S_1, S_5, S_7}

第二次划分：{S_0, S_1, S_2}　　{S_3, S_4}　　{S_5, S_6, S_7, S_8}

{S_0, S_1, S_2}*a* = {S_1, S_3, S_1}

第三次划分：{S_0, S_2}　{S_1}　　{S_3, S_4}　{S_5, S_6, S_7, S_8}

{S_0, S_2}*a* = {S_1}　{S_0, S_2}*b* = {S_2}　　S_0, S_2 不可区分，即等价。

{S_5, S_6, S_7, S_8}*a* = {S_5, S_7, S_3, S_1}

第四次划分：{S_0, S_2}　{S_1}　　{S_3, S_4}　{S_5, S_6}　{S_7, S_8}

{S_3, S_4}*a* = {S_5, S_7}

第五次划分：{S_0, S_2}　{S_1}　　{S_3}　{S_4}　{S_5, S_6}　{S_7, S_8}

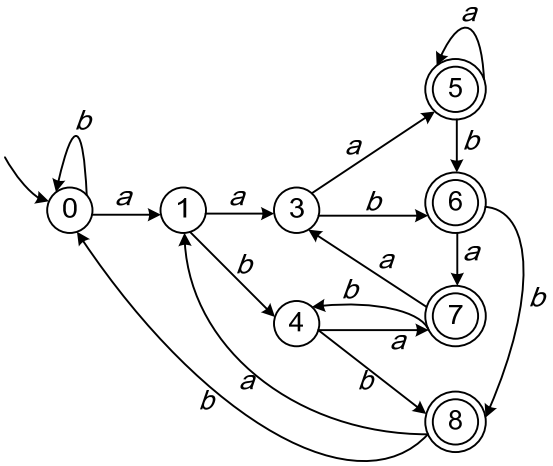
{S_5, S_6}*a* = {S_5, S_7}

第六次划分：{S_0, S_2}　{S_1}　　{S_3}　{S_4}　{S_5}　{S_6}　{S_7, S_8}

{S_7, S_8}*a* = {S_3, S_1}

第七次划分：{S_0, S_2} {S_1} {S_3} {S_4} {S_5} {S_6} {S_7} {S_8}

集合不可再划分，所以 S_0, S_2 等价，选取 S_0 表示{S_0, S_2}，其状态转换图，即题目所要求的最简 *DFA* 如下所示：



2.14 构造一个 *DFA*，它接受 Σ = {0，1}上能被 5 整除的二进制数。

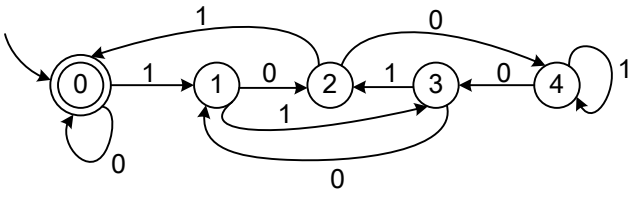
［解答］

分析题目可知，一个二进制数除以 5，其余数（十进制）只能是 0，1，2，3，4 五种，因此我们以 0，1，2，3，4 分别表示这五种状态。因为要求得能被 5 整除的二进制数，故状态 0 既为初始状态，又为终结状态。

二进制序列中只能有 0 或 1 组成，下面举例说明序列中增加 0 或 1 后余数的变化。

$(000)_2$	增加 0	$(0000)_2 \bmod 5 = (0)_{10}$	增加 1	$(0001)_2 \bmod 5 = (1)_{10}$
$(001)_2$	增加 0	$(0010)_2 \bmod 5 = (2)_{10}$	增加 1	$(0011)_2 \bmod 5 = (3)_{10}$
$(010)_2$	增加 0	$(0100)_2 \bmod 5 = (4)_{10}$	增加 1	$(0101)_2 \bmod 5 = (0)_{10}$
$(011)_2$	增加 0	$(0110)_2 \bmod 5 = (1)_{10}$	增加 1	$(0111)_2 \bmod 5 = (2)_{10}$
$(100)_2$	增加 0	$(1000)_2 \bmod 5 = (3)_{10}$	增加 1	$(1001)_2 \bmod 5 = (4)_{10}$

所以其 *DFA* 为下图所示：



第三章 语法分析

3.2 考虑文法

S → aSbS | bSaS | ε

(a) 为句子 *abab* 构造两个不同的最左推导，以此说明该文法是二义的。

［解答］

S =>lm aSbS =>lm aεbS =>lm aεbaSbS
=>lm aεbaεbS =>lm aεbaεbε =>lm abab
S =>lm aSbS =>lm abSaSbS =>lm abεaSbS
=>lm abεaεbS =>lm abεaεbε =>lm abab

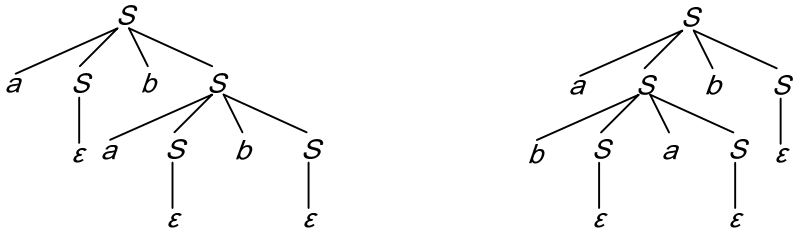
(b) 为 *abab* 构造对应的最右推导。

［解答］

S =>rm aSbS =>rm aSbaSbS =>rm aSbaSbε
=>rm aSbaεbε =>rm aεbaεbε =>rm abab
S =>rm aSbS =>rm aSbε => abSaSbε
=> abSaεbε => abεaεbε =>rm abab

(c) 为 *abab* 构造对应的分析树。

［解答］



3.10 构造下列文法的 LL(1)分析表

D → T L T → int | real L → id R R → , id R | ε

［解答］

D → *T L*	*FIRST*(*TL*)	= { int, real }
T → int	*FIRST*(int)	= { int }
T → real	*FIRST*(real)	= { real }
L → id *R*	*FIRST*(id *R*)	= { id }
R → , id *R*	*FIRST*(,id*R*)	= { , }
R → ε	*FIRST*(ε)	= { ε }
	FOLLOW(*R*)	= FOLLOW(*L*) = *FOLLOW*(*D*) = { $ }

	int	real	id	,	$
D	*D* → *TL*	*D* → *TL*			
T	*T* → int	*T* → real			
L			*L* → id *R*		
R				*R* → ,id*R*	*R* → ε

3.11 下面的文法是否为 **LL(1)**文法？说明理由。

S* → *A B* | *P Q x*　　*A* → *x y*　　*B* → *b c

P* → *d P* | ε　　*Q* → *a Q* | *ε

［解答］

S → *A B* | *P Q x*

FIRST(*AB*) = *FIRST*(*A*) = *FIRST*t(*xy*) = { *x* }

FIRST(*PQx*) = { *FIRST*(*P*) - { *ε* } } ∪ { *FIRST*(*Q*) - { *ε* } } ∪ { *x* }

= *FIRST*(*dP*) ∪ *FIRST*(*aQ*) ∪ { *x* }

= { *d, a, x* }

因为*FIRST*(*AB*) ∩ *FIRST*(*PQx*) = { *x* } ≠ Φ，不满足LL(1)文法的定义。所以此文法不是LL(1)文法。

3.16 给出接受文法

S* → (*L*) | *a*　　*L* → *L, S* | *S

的活前缀的一个 *DFA*

［解答］

拓广文法

S ' → *S*　　*S* → (*L*) | *a*　　*L* → *L, S* | *S*

构造项目集过程：

I_0 = *closure*({ [*S* '→*S* ·] }) = { *S* '→*S* ·, *S* → · (*L*), *S* → · *a* }

I_1 = *goto*(I_0, *S*) = *closure*({ [*S* '→*S* ·] }) = { *S* '→*S* · }

I_2 = *goto*(I_0, ( ) = *closure*({ [*S* → (· *L*)] })

= { *S* → (· *L*, *L* → · *L*, *S*, *L* → · *S*, *S* → · (*L*), *S* → · *a* }

I_3 = *goto*(I_0, a) = *closure*({ [*S* → *a* ·] }) = { *S* → *a* · }

I_4 = *goto*(I_2, *L*) = *closure*({ [*S* → (*L* ·)], [*L* → *L* · , *S*] })

= { *S* → (*L* ·), *L* → *L* · , *S* }

I_2 = *goto*(I_2, ( ) = *closure*({ [*S* → (· *L*)] })

= { *S* → (· *L*, *L* → · *L*, *S*, *L* → · *S*, *S* → · (*L*), *S* → · *a* }

I_5 = *goto*(I_2, *S*) = *closure*({ [*L* → *S* ·] }) = { *L* → *S* · }

I_6 = *goto*(I_4,)) = *closure*({ [*S* → (*L*) ·] }) = { *S* → (*L*) · }

I_7 = *goto*(I_4, ,) = *closure*({ [*L* → *L* , · *S*] }) = { *L* → *L* , · *S*, *S* → · (*L*), *S* → · *a* }

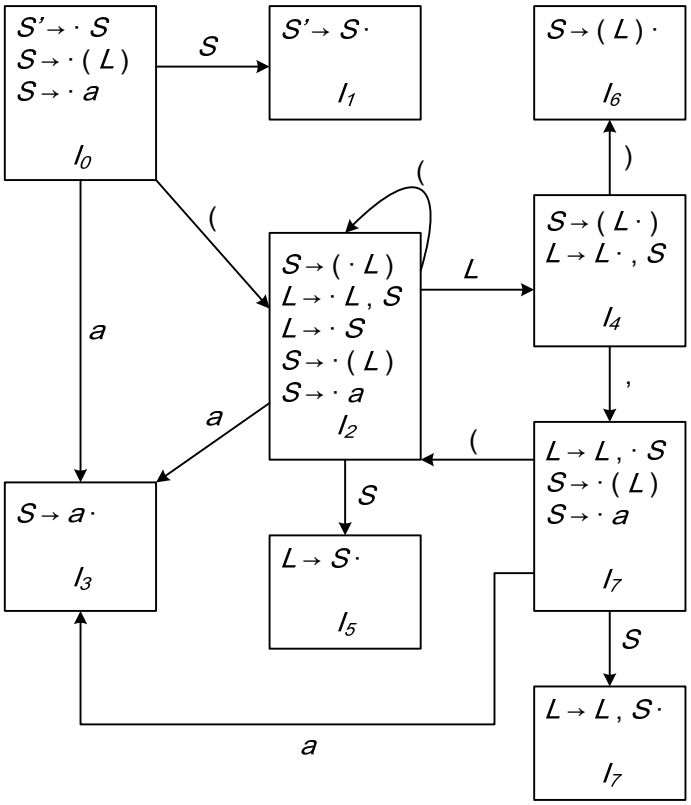
I_8 = *goto*(I_4, *S*) = *closure*({ [*L* → *L* , *S* ·] }) = { *L* → *L* , *S* · }

I_2 = *goto*(I_7, ( ) = *closure*({ [*S* → (· *L*)] })

= { $S \rightarrow (\cdot L, L \rightarrow \cdot L, S, L \rightarrow \cdot S, S \rightarrow \cdot (L), S \rightarrow \cdot a$ }

$I_3 = goto(I_7, a) = closure(\{[S \rightarrow a \cdot]\}) = \{S \rightarrow a \cdot\}$

识别其活前缀的 *DFA* 如图：



3.20 (a) 证明下面文法

S → A a A b | B b B a

A → ε B → ε

是 LL(1)文法，但不是 SLR(1)文法。

［解答］

(a)

S → A a A b | B b B a 有两个候选式，根据 LL(1)文法定义，

FIRST(*AaAb*) ∩ *FIRST*(*BbBa*) = { *a* } ∩ { *b* } = Φ

所以，该文法为 LL(1)文法。

A → ε 和 *B* → ε 进行归约时，我们考虑，

FOLLOW(*A*) = { *a, b* }

FOLLOW(*B*) = { *b, a* }

FOLLOW(*A*) ∩ *FOLLOW*(*B*) ≠ Φ

所以会出现归约—归约冲突，因此该文法不是 SLR(1)文法。

3.22 证明下面文法

S → A a | b A c | d c | b d a

$A \rightarrow d$

是 LALR(1)文法，但不是 SLR(1)文法。

［解答］

(1) 考虑项目[$A \rightarrow d \cdot$]和[$S \rightarrow d \cdot c$]

∵ $FOLLOW(A) \cap \{c\} = \{a, c\} \cap \{c\} \neq \Phi$

∴ 出现归约—移进冲突。

再考虑项目[$A \rightarrow d \cdot$]和[$S \rightarrow b\,d \cdot a$]

∵ $FOLLOW(A) \cap \{a\} = \{a, c\} \cap \{a\} \neq \Phi$

∴ 出现归约—移进冲突。

所以该文法不是 SLR(1)文法。

(2) 此文法比较简单，根据其定义我们可以知道，此文法能得到的句子有：

da，*bdc*，*dc*，*bda*

① 当分析栈栈顶为 *d* 时，当分析栈外为 *a* 时，*d* 才可以被归约为 *A*，即[$A \rightarrow d \cdot, a$]，所以上面的第一个冲突被解决；

② 当分析栈栈顶为 *bd* 时，当分析栈外为 *c* 时，*d* 才可以被归约为 *A*，即[$A \rightarrow d \cdot, c$]，所以上面的第二个冲突被解决；

所以此文法为 LR(1)文法，而此文法中没有同心集，所以该文法也为 LALR(1)文法。

3.25 证明下面文法

$S \rightarrow A\,a \mid b\,A\,c \mid B\,c \mid b\,B\,a$

$A \rightarrow d \qquad B \rightarrow d$

是 LR(1)文法，但不是 LALR(1)文法。

［解答］

(1) 此文法与 3.22 题中文法所定义语言集合相同，文法能得到的句子有：

da，*bdc*，*dc*，*bda*

由此我们可以看出：

① 分析栈顶为 *d* 时，栈外为 *a* 时，*d* 被归约为 *A*；栈外为 *c* 时，*d* 被归约为 *B*；

② 分析栈顶为 *bd* 时，栈外为 *c* 时，*d* 被归约为 *A*；栈外为 *a* 时，*d* 被归约为 *B*；

所以此文法为 LR(1)文法。

(2) 当析栈顶为 *d* 时，LR(1)对应的项目集为

{［$A \rightarrow d \cdot, a$ ］,［B $\rightarrow d \cdot, c$ ］}

当析栈顶为 bd 时，LR(1)对应的项目集为

{［$A \rightarrow d \cdot, c$ ］,［$B \rightarrow d \cdot, a$ ］}

这两个项目集为同心集，合并同心集后为：

{［$A \rightarrow d \cdot, a/c$］,［$B \rightarrow d \cdot, c/a$ ］}

产生归约—归约冲突，所以该文法不是 LALR(1)文法。

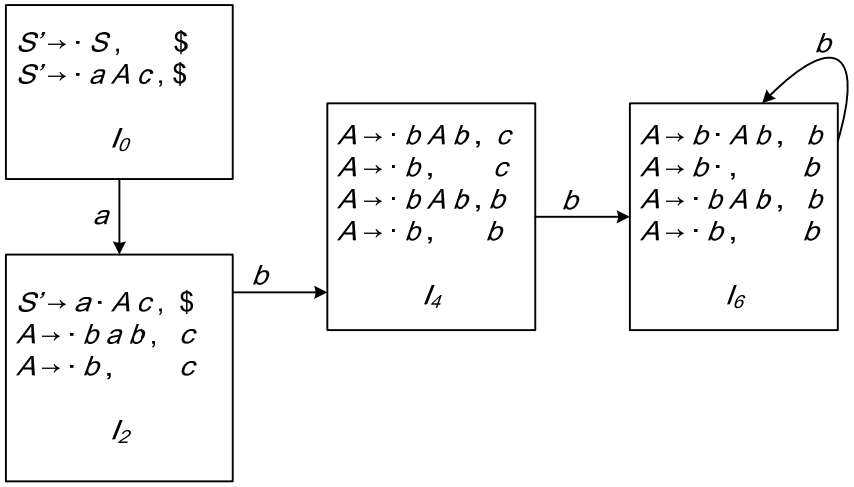
3.30 下面两个文法中哪一个不是 LR(1)文法？对非 LR(1)的那个文法，给出那个有移进—归约冲突的规范的 LR(1)项目集。

$S \rightarrow aAc$　　$S \rightarrow aAc$

$A \rightarrow Abb \mid b$　　$A \rightarrow bAb \mid b$

［解答］

文法 $S \rightarrow aAc$，$A \rightarrow bAb \mid b$ 不是 LR(1)文法。



有移进—归约冲突的项目集为 I_6。

第四章 语法制导的翻译

4.3 为文法 **S → (L) | a　　L → L, S | S**

(a) 写一个语法制导定义，它输出括号的对数。

［解答］

S' → S　　{ print (S.*num*); }
S → (L)　　{ S.*num* = L.*num* + 1; }
S → a　　{ S.*num* = 0; }
L → L_1 , S　　{ L.*num* = L_1.*num* + S.*num*; }
L → S　　{ L.*num* = S.*num*; }

(b) 写一个语法制导定义，它输出括号嵌套的最大深度。

［解答］

S' → S　　{ print (S.*max*); }
S → (L)　　{ S.*max* = L.*max* + 1; }
S → a　　{ S.*max* = 0; }
L → L_1, S　　{ L.*max* = if (L_1.*max* > S.*max*) then L_1.*max* else S.*max*; }
L → S　　{ L.*max* = S.*max*; }

4.6 给出把中缀表达式翻译成没有冗余括号的中缀表达式的语法制导定义。例如，因为**+**和*****是左结合，**((a * (b + c)) * (d))**可以重写成 **a * (b + c) * d**

［解答］

S' → E　　{ print (E.*code*); }
E → E_1+T　　{ if T.*op* = *plus* then
　　E.*code* = E_1.*code* || “+” || “(” || T.*code* || “)”;
　　else
　　E.*code* = E_1.*code* || “+” || T.*code*;
　　E.*op* = *plus*;
　　}
E → T　　{ E.*code* = T.*code*; E.*op* = T.*op*; }
T → T_1*F　　{ if (F.*op* = *plus*) or (F.*op* = *times*) then
　　if T_1.*op* = *plus* then
　　T.*code* = “(” || T_1.*code* || “)” || “*” || “(” || F.*code* || “)”;
　　else
　　T.*code* = T_1.*code* || “*” || “(” || F.*code* || “)”;
　　else if T_1.*op* = *plus* then
　　T.*code* = “(” || T_1.*code* || “)” || “*” || F.*code*;
　　else

T.*code* = T_1.*code* || “*” || F.*code*;
T.*op* = *times*
}
T → F { T.*code* = F.*code*; T.*op* = F.*op*; }
F → id { F.*code* = id.*lexeme*; F.*op* = id; }
F → (E) { F.*code* = E.*code*; F.*op* = E.*op*; }

4.10 文法如下：**S → (L) | a　L → L , S | S**

(a) 写一个翻译方案，它输出每个 **a** 的嵌套深度。例如，对于句子**(a, (a, a))**，输出的结果是 **1 2 2**。

［解答］

S’ → { S.depth = 0; } S
S → { L.depth = S.depth + 1; } (L)
S → a { print (S.depth); }
L → { L_1.depth = L.depth; } L_1 , { S.depth = L.depth; } S
L → { S.depth = L.depth; } S

(b) 写一个翻译方案，它打印出每个 **a** 在句子中是第几个字符。例如，当句子是**(a , (a , (a , a) , (a)))**时，打印的结果是 **2 5 8 10 14**。

［解答］

S’ → { S.in = 0; } S
S → { L.in = S.in +1; } (L) { S.out = L.out + 1; }
S → a { S.out = S.in + 1; print (S.out); }
L → { L_1.in = L.in; } L_1, { S.in = L_1.out + 1; } S {L.out = S.out; }
L → { S.in = L.in; } S { L.out = S.out; }

第五章 类型检查

5.3 下面是一个 **C** 语言程序，虽然 **main** 函数中两次调用函数 **gcd** 的参数个数都不对，但是该程序能够通过编译和连接装配而形成一个目标程序。试问为什么 **C** 编译器和连接装配器没能发现这样的错误？

```
long gcd (p, q)
long p, q;
{
    if ( p % q == 0 )
        return q;
    else
        return gcd (q, p%q);
}
main()
{
    printf ( "%ld, %ld \n", gcd(5), gcd(5, 10, 20) );
}
```

［解答］

题目中 gcd 函数采用的是旧式声明方式，调用时实际参数的数目与函数定义中形式参数的数目不相等，函数调用的结果是没有定义的。本题目中的 C 语言编译器和连接器没有进行函数参数和函数定义的类型是否一致的检查。

如果函数是以新式方式声明的，则实际参数数目必须与显式声明的形式参数数目相同（除函数的参数是可变参数）。

5.5 假如有下列 **C** 的声明

```
typedef struct {
    int a, b;
} CELL, *PCELL;
CELL foo[100];
PCELL bar (x, y) int x; CELL y; { ... }
```

为类型 **foo** 和 **bar** 写类型表达式。

［解答］

foo 的类型表达式：

array(0..99, *record* ((*a*×*integer*)×(*b*×*integer*)))

bar 的类型表达式：

(*integer*×*record* ((*a*×*integer*)×(*b*×*integer*)))

→ *pointer*(*record* ((*a*×*integer*)×(*b*×*integer*)))

第六章 运行时存储空间的组织和管理

6.4 下面给出一个 **C** 语言程序及其在 **X86/Linux** 操作系统上的编译结果。根据所生成的汇编程序来解释程序中 **4** 个变量的存储分配、作用域、生存期和置初值方式等方面的区别。

```
static long aa = 10;
short bb = 20;
func() {
    static long cc = 30;
    short dd = 40;
}
```

该 **C** 语言程序生成的汇编代码：

```
.file "static.c"
  .version “01.01”
gcc2_compiled.:
.data
    .align 4
    .type aa, @object
    .size aa, 4
aa:
    .long 10
.globl bb
    .align 2
    .type bb, @object
    .size bb, 2
bb:
    .value 20
    .align 4
    .type cc.2, @object
```

```
        .size cc.2, 4
cc.2:
        .long 30
.text
    .align 4
.globl func
        .type func, @function
func:
        pushl %ebp
        movl %esp, %ebp
        subl $4, %esp
        movw $40, -2(%ebp)
.L1:
        leave
        ret
.Lfe1:
        .size func, .Lfe1-func
.ident "GCC: (GNU) egc–2.91.66 19990314/Linux(egcs–1.1.2 release)
```

［解答］

变量名	类　型	存储分配	作用域	生存期	置初值方式
aa	静态外部变量	静态数据区	本文件	整个程序	静态
bb	外部变量	静态数据区	所有文件	整个程序	静态
cc	静态局部变量	静态数据区	func 函数	整个程序	静态
dd	局部变量	栈区	func 函数	func 函数活动期	动态

6.5 假定使用：（a）值调用；（b）引用调用；（c）值—结果调用；（d）换名调用。下面的程序打印的结果是什么?

```
program main(input, output);
var a, b: integer;
    procedure p(x, y, z : integer);
        begin
```

```
            y := y + 1;
            z := z + x;
        end;
    begin
        a := 2;
        b := 3;
        p(a+b, a, a);
        print a;
    end.
```

［解答］

(a) 2
(b) 8
(c) 3 或 7
(d) 9

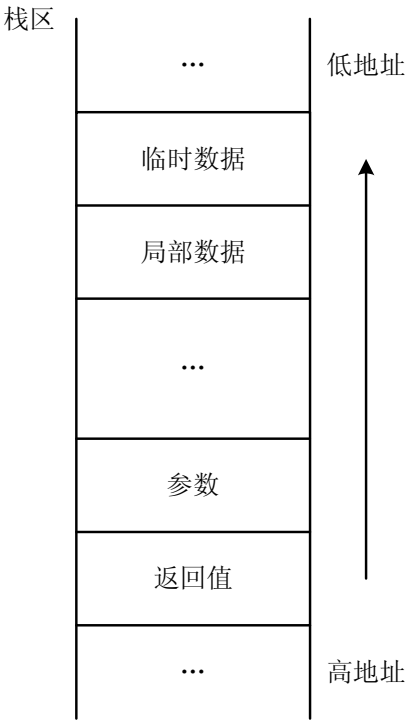
6.6 一个 **C** 语言程序如下：

```
func ( i1, i2, i3)
long i1, i2, i3;
{
    long j1, j2, j3;
    printf( "Addresses of i1, i2, i3 = %o, %o, %o \n", &i1, &i2, &i3);
    printf( "Addresses of j1, j2, j3 = %o, %o, %o \n", &j1, &j2, &j3);
}
main()
{
    long i1, i2, i3;
    func(i1, i2, i3);
}
```

该程序在 **X86/Linux** 机器上的运行结果如下：

```
Addresses of i1, i2, i3 = 27777775460, 27777775464, 27777775470
Addresses of j1, j2, j3 = 27777775444, 27777775440, 27777775434
```

从上面的结果可以看出，func 函数的 3 个形式参数的地址依次升高，而 3 个局部变量的地址依次降低。试说明为什么会有这个区别。注意，输出的数据是八进制的。



［解答］

在 gcc version3.4.2 下汇编代码如下（部分）：

```
_func:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        leal    16(%ebp), %eax
        movl    %eax, 12(%esp)
        leal    12(%ebp), %eax
        movl    %eax, 8(%esp)
        leal    8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $LC0, (%esp)
        call    _printf
        leal    -12(%ebp), %eax         // j3
        movl    %eax, 12(%esp)
        leal    -8(%ebp), %eax          // j2
        movl    %eax, 8(%esp)
        leal    -4(%ebp), %eax          // j1
        movl    %eax, 4(%esp)
        movl    $LC1, (%esp)
        call    _printf
        leave
        ret

_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        movl    %eax, -16(%ebp)
        movl    -16(%ebp), %eax
        call    __alloca
```

```
call    ___main
movl    -12(%ebp), %eax
movl    %eax, 8(%esp)          // i3
movl    -8(%ebp), %eax
movl    %eax, 4(%esp)          // i2
movl    -4(%ebp), %eax
movl    %eax, (%esp)           // i1
call    _func
leave
ret
```

由上面汇编代码可以看出在 main 函数中，实参进栈的顺序是 i3、i2、i1。一般情况下栈向低地址方向增长，所以 i1、i2 和 i3 的地址依次增大。

对于局部变量 j1、j2、j3，它们在局部数据区按照声明的顺序依次入栈，所以 j1、j2、j3 地址依次减小。

6.8 下面给出一个 **C** 语言程序及其在 **SPARC/SUN** 工作站上经某编译器编译后的运行结果。从运行结果看，函数 **func** 中 **4** 个局部变量 **i1, j1, f1, e1** 的地址间隔和它们类型的大小是一致的，而 **4** 个形式参数 **i, j, f, e** 的地址间隔和它们的类型的大小不一致，试分析不一致的原因。注意，输出的数据是八进制的。

```
func (i, j, f, e)
short i, j; float f, e;
{
    short i1, j1; float f1, e1;
    printf( "Address of i, j, f, e = %o, %o, %o, %o \n", &i, &j, &f, &e);
    printf( "Address of i1, j1, f1, e1 = %o, %o, %o, %o \n", &i1, &j1, &f1, &e1);
    printf( "Sizes of short, int, long, float, double = %d, %d, %d, %d, %d \n",
        sizeof(short), sizeof(int), sizeof(long), sizeof(float), sizeof(double) );
}
main()
{
    short i, j; float f, e;
    func(i, j, f, e);
}
```

运行结果是：

```
Address of i, j, f, e = 35777772536, 35777772542, 35777772544, 35777772554
```

Address of i1, j1, f1, e1 = 35777772426, 35777772424, 35777772420, 35777772414

Sizes of short, int, long, float, double = 2, 4, 4, 4, 8

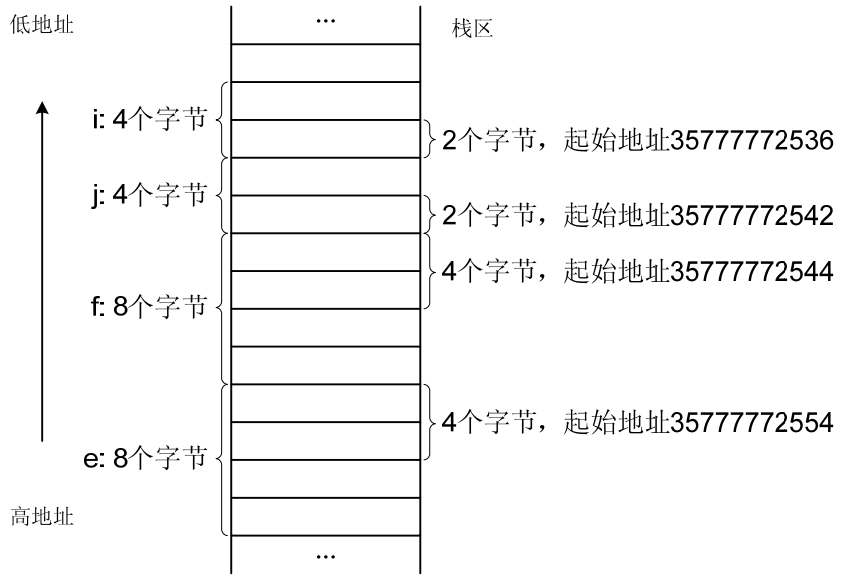
［解答］

在函数调用的作用域中，如果函数是以旧式方式声明的，按以下方式对每个实际参数进行默认参数提升：对每个整型参数进行整型提升；将每个 float 类型的参数转换为 double 类型。

int 类型数据占用 4 个字节，而 short 类型占用 2 个字节。因此，被调用函数把实在参数的高地址端的内容当成是自己所需要的数据。（SPARC/SUN 采用大端法，big endian）

对于实型，double 类型是 8 个字节，而 float 类型占用 4 个字节。被调用函数把实在参数的低地址端的内容当成是自己所需要的数据。

其参数空间分配可参考下图。



6.14 一个 C 语言程序如下：

```
int n;
int f(g)
int g();
{
    int m;
    m = n;
    if (m == 0) return 1;
    else {
        n = n - 1; return m * g(g);
    }
```

```
}
main()
{
    n = 5;
    printf( " %d factorial is %d \n ", n, f(f) );
}
```

该程序的运行结果不是我们所期望的

5 factroial is 120

而是

0 factroial is 120

[解答]

在 gcc version3.4.2 下 main 函数汇编代码如下：

```
_main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $24, %esp
    andl    $-16, %esp
    movl    $0, %eax
    addl    $15, %eax
    addl    $15, %eax
    shrl    $4, %eax
    sall    $4, %eax
    movl    %eax, -4(%ebp)
    movl    -4(%ebp), %eax
    call    __alloca
    call    ___main
    movl    $5, _n
    movl    $_f, (%esp)
    call    _f          ←———————— 函数 f 被调用
    movl    %eax, 8(%esp)
    movl    _n, %eax    ←———————— 参数 n 进栈
    movl    %eax, 4(%esp)
    movl    $LC0, (%esp)
    call    _printf
    leave
    ret
```

由上面的汇编代码可以看出在函数调用过程中，实参是逆序进栈，因此 f 函数先被调用，而在函数 f 中全局变量 n 值被减为 0，所以出现了题目中的输出结果。