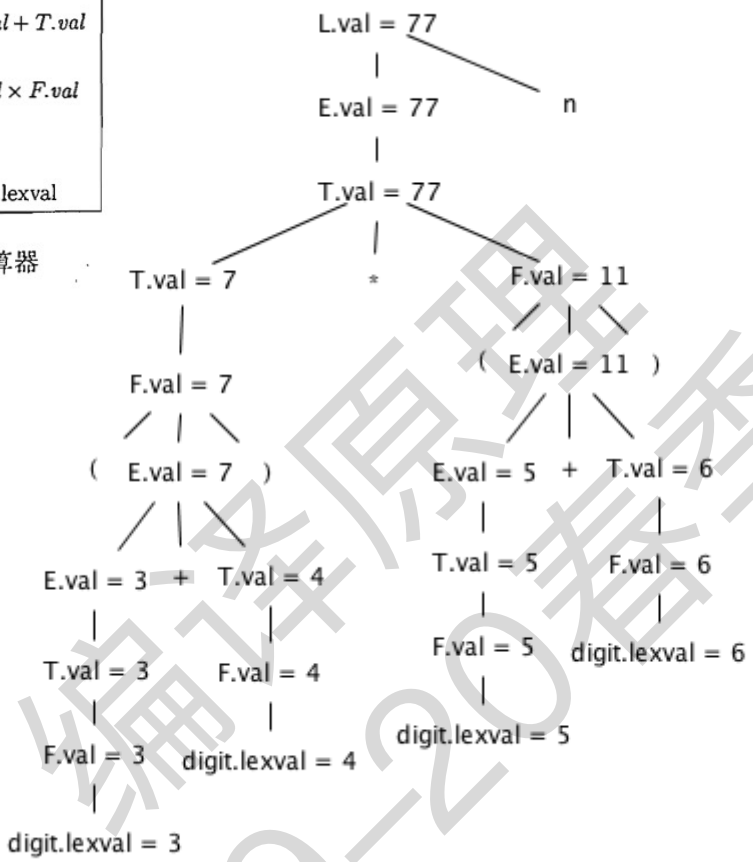
第五章作业

练习 **5.1.1**　5.1.1：对于图5-1中的SDD，给出下列表达式对应的注释语法分析树

1）（ 3 + 4 ）*（ 5 + 6 ）n

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

图 5-1　一个简单的桌上计算器的语法制导定义



练习 5.1.2　扩充图5-4中的SDD，使它可以像图5-1所示的那样处理表达式

产生式	语义规则
L -> En	L.val = E.val
E -> TE’	E’.inh = T.val E.val = E’.syn
E’ -> +TE$_1$’	E$_1$’.inh = E’.inh + T.val E’.syn = E$_1$’.syn
E’ -> ε	E’.syn = E’.inh
T -> FT’	T’.inh = F.val T.val= T’.syn
T’ -> *FT$_1$’	T$_1$’.inh = T’.inh*F.val T’.syn = T$_1$’.syn
T’ -> ε	T’.syn = T’.inh
F -> (E)	F.val = E.val
F -> digit	F.val = digit.lexval

产生式	语义规则
1) $T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow *\,F\,T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

图 5-4　一个基于适用于自顶向下语法分析的文法的 SDD

练习 5.2.3

5.2.3：假设我们有一个产生式A→BCD，A、B、C、D这四个非终结符号都有两个属性：s是一个综合属性，i是一个继承属性。对于下面的每组规则，指出：（i）这些规则是否满足S属性定义的要求。（ii）这些规则是否满足L属性定义的要求。（iii）是否存在和这些规则一致的求值过程？

1. A.s = B.i + C.s
2. A.s = B.i + C.s 和 D.i = A.i + B.s
3. A.s = B.s + D.s
4. A.s = D.i, B.i = A.s + C.s, C.i = B.s 和 D.i = B.i + C.i

（1）不是；是；存在
（2）不是；是；存在
（3）是；是；存在
（4）不是；不是；不存在

练习 5.3.1

◆ 5.3.1：下面是涉及运算符+和整数或浮点运算分量的表达式的文法。区分浮点数的方法是看它有无小数点。

E → E + T | T
T → num.num | num

1）给出一个SDD来确定每个项T和表达式E的类型

2）扩展1）中得到的SDD，使得它可以把表达式转换为后缀表达式。使用一个单目运算符intToFlost把一个整数转换为相等的浮点数

1)

产生式	语义规则

E→E$_1$+T	if (E$_1$.type==integer && T.type==integer){
	E.type = integer
	}else{
	E.type = real
	}

E→T	E.type = T.type
T→**num.num**	T.type = real
T→**num**	T.type = integer

2)

产生式产生式	语义规则
E→E$_1$+T	if (E$_1$.type==integer && T.type==integer){ E.type = integer

	E.post = E1.post || T.post || 'int+'
	}else{
	E.type = real
	if (E_1.type==integer){
	E_1.type = real
	E_1.post = E_1.post || "inttoreal"
	}
	if(T.type==integer){
	T.type =real
	T.post = T.post || "inttoreal"
	}
	E.post = E_1.post || T.post || 'float+"
	}

E→T	E.type = T.type E.post = T.post
T→**num.num**	T.type = real T.post = **num.num**
T→**num**	T.type = integer T.post:= **num**

其中 post 属性为后缀符号串，'||'符号为连接运算

练习 **5.4.2**

5.4.2：改写下面的SDT：

A → A { a } B | A B { b } | 0

B → B { c } A | B A { d } | 1

使得基础文法变成非左递归的。其中a、b、c和d是语义动作，0和1是终结符号

A -> 0A’

A' -> {a}BA' | B{b}A' | ε

B -> 1B'

B' -> {c}AB' | A{d}B' | ε

如果 a、b、c、d 涉及到属性计算的话，变换的结果要更复杂一些。

练习 **5.4.6**

5.4.6：修改图5-25中的SDD，使它包含一个综合属性B.le，即一个Box的长度。两个Box并列后得到的Box的长度是这两个Box的长度和。然后，将你的新规则加入到图5-26中SDT的合适位置上

SDD 与 SDT 中修改的部分使用粗体表示。

SDD：

S -> B　　B.ps = 10

B -> B_1B_2　　B_1.ps = B.ps

B_2.ps = B.ps

B.le = B_1.le + B_2.le

B.ht = max(B_1.ht, B_2.ht)

B.dp = max(B_1.dp, B_2.dp)

产生式	语义规则
1) $S \rightarrow B$	$B.ps = 10$
2) $B \rightarrow B_1\ B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
3) $B \rightarrow B_1$ **sub** B_2	$B_1.ps = B.ps$ $B_2.ps = 0.7 \times B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
4) $B \rightarrow (\ B_1\ )$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5) $B \rightarrow$ **text**	$B.ht = getHt(B.ps, \mathbf{text}.lexval)$ $B.dp = getDp(B.ps, \mathbf{text}.lexval)$

图 5-25　方框排版的 SDD

B -> B_1 sub B_2	B_1.ps = B.ps B_2.ps = 0.7 * B.ps **B.le = B_1.le + B_2.le** B.ht = max(B_1.ht, B_2.ht - 0.25 * B.ps) B.dp = max(B_1.dp, B_2.dp + 0.25 * B.ps)
B -> (B_1)	B_1.ps = B.ps **B.le = B_1.le** B.ht = B_1.ht B.dp = B_1.dp
B -> text	**B.le = getLe(B.ps, text.lexval)** B.ht = getHt(B.ps, text.lexval) B.dp = getDp(B.ps, text.lexval)

SDT：

S -> B	{ B.ps = 10;}
B -> B_1 B_2	{B_1.ps = B.ps;} {B_2.ps = B.ps;} {**B.le = B_1.le + B_2.le;** B.ht = max(B_1.ht, B_2.ht); B.dp = max(B_1.dp, B_2.dp);}
B -> B_1 sub B_2	{B_1.ps = B.ps;} {B_2.ps = 0.7 * B.ps;} {**B.le = B_1.le + B_2.le;** B.ht = max(B_1.ht, B_2.ht - 0.25 * B.ps); B.dp = max(B_1.dp, B_2.dp + 0.25 * B.ps);}
B -> (B_1)	{B_1.ps = B.ps;} {**B.le = B_1.le;** B.ht = B_1.ht B.dp = B_1.dp;}
B -> text	{**B.le = getLe(B.ps, text.lexval);** B.ht = getHt(B.ps, text.lexval); B.dp = getDp(B.ps, text.lexval);}

◆ 5.4.4：为下面的产生式写出一个和例5.19类似的L属性SDD。这里的每个产生式表示一个常见的C语言中那样的控制流结构。你可能需要生成一个三地址语句来跳转到某个标号L，此时你可以生成语句goto L

① $S \rightarrow$ if $(C)\ S_1$ else S_2

② $S \rightarrow$ do S_1 while (C)

③ $S \rightarrow$ '{' L '}' ; $L \rightarrow LS \mid \varepsilon$

5.4.4

清注意，列表中的任何语句都可能包含一条从它的内部跳转到下一条语句的跳转指令，因此简单地为各个语句按顺序生成代码是不够的

（1）

S -> if (C) S_1 else S_2

L_1 = newlabel()
L_2 = newlabel()
C.true = L_1
C.false = L_2
S_1.next = S.next
S_2.next = S.next
S.code = C.code || label || L_1 || S_1.code || goto S.next || label || L_2 || S_2.code

（2）

S -> do S_1 while(C)

L_1 = newlabel ()
L_2 = newlabel ()
C.true = L_1
C.false = S.next
S_1.next = L_2
S.code = label || L_1 || S_1.code || label || L_2 || C.code

（3）

S -> '{' L '}'

L.next = S.next
S.code = L.code

L -> L_1S

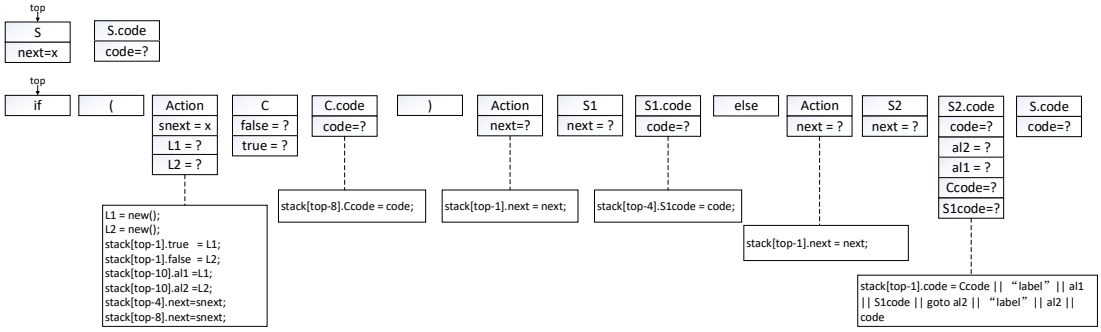
M = newlabel ()
L_1.next = M
S.next = L.next
L.code = L_1.code || label || M || S_1.code

L -> ε

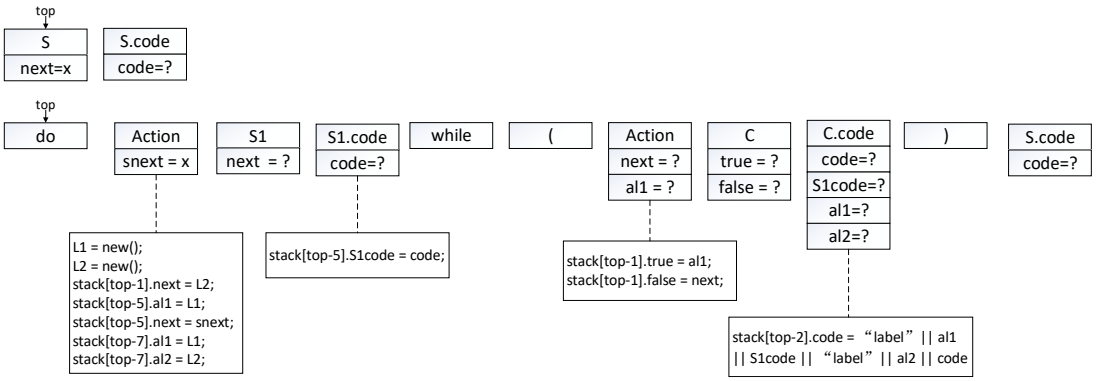
L.code = ""

5.5.4：图中用于申请标号的 **new()**，应该是 **newlabel()**，请注意。

a) **5.5.4：按照5.5.3节的风格，将5.4.4中得到的每个SDD和一个LL语法分析器一起实现，但是代码（或指向代码的指针）存放在栈中**



b)



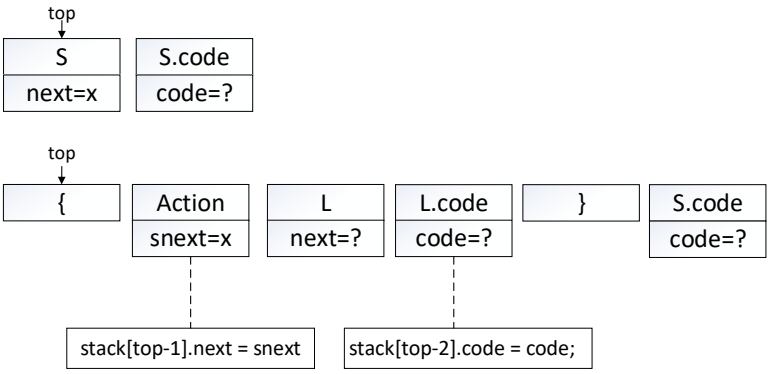
c) 原文法包含左递归，需要先消除左递归。

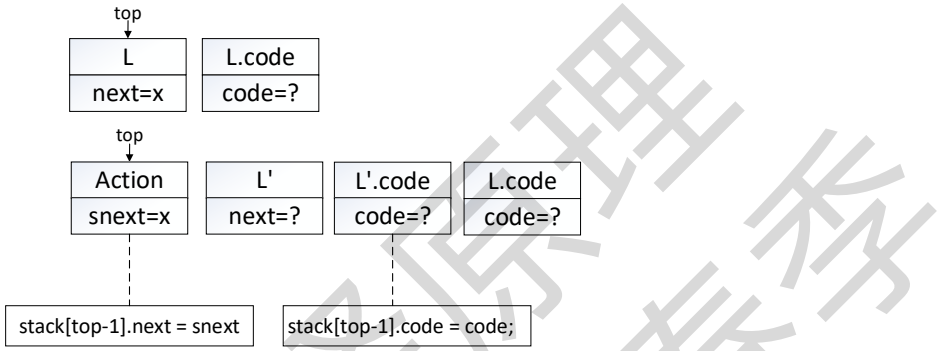
对 L –> LS | ε 消除左递归，得到文法：

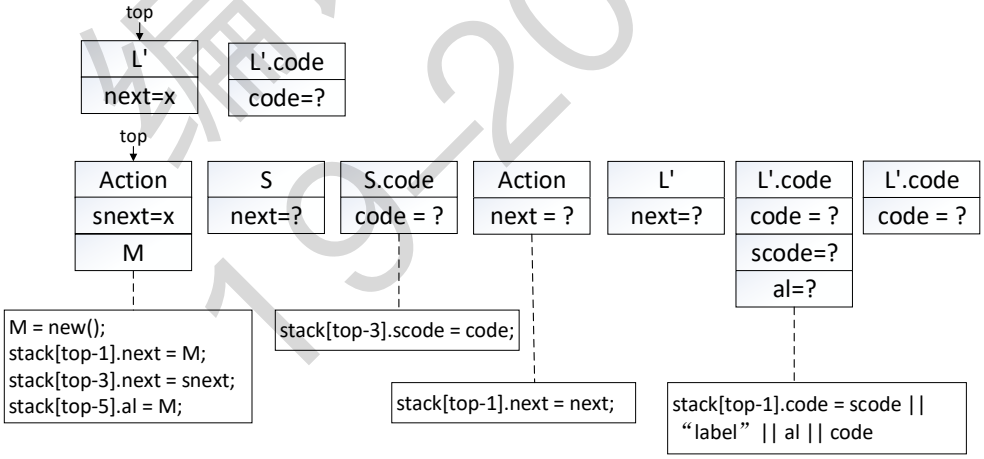
S -> ‘{’ L ’}’

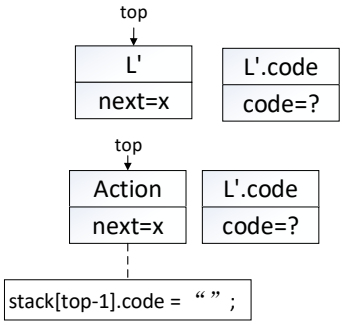
L ->L’

L’ –> SL’ | ε


top
S
next=x
S.code
code=?
top
{
Action
snext=x
L
next=?
L.code
code=?
}
S.code
code=?
stack[top-1].next = snext
stack[top-2].code = code;


top
L
next=x
L.code
code=?
top
Action
snext=x
L'
next=?
L'.code
code=?
L.code
code=?
stack[top-1].next = snext
stack[top-1].code = code;

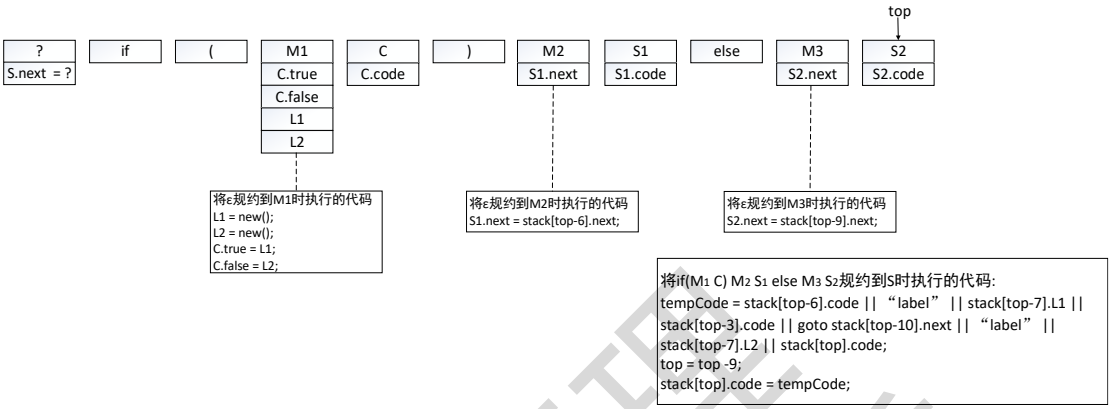

top
L'
next=x
L'.code
code=?
top
Action
snext=x
M
S
next=?
S.code
code = ?
Action
next = ?
L'
next=?
L'.code
code = ?
scode=?
al=?
L'.code
code = ?
M = new();
stack[top-1].next = M;
stack[top-3].next = snext;
stack[top-5].al = M;
stack[top-3].scode = code;
stack[top-1].next = next;
stack[top-1].code = scode ||
“label” || al || code


top
L'
next=x
L'.code
code=?
top
Action
next=x
L'.code
code=?
stack[top-1].code = “ ”;

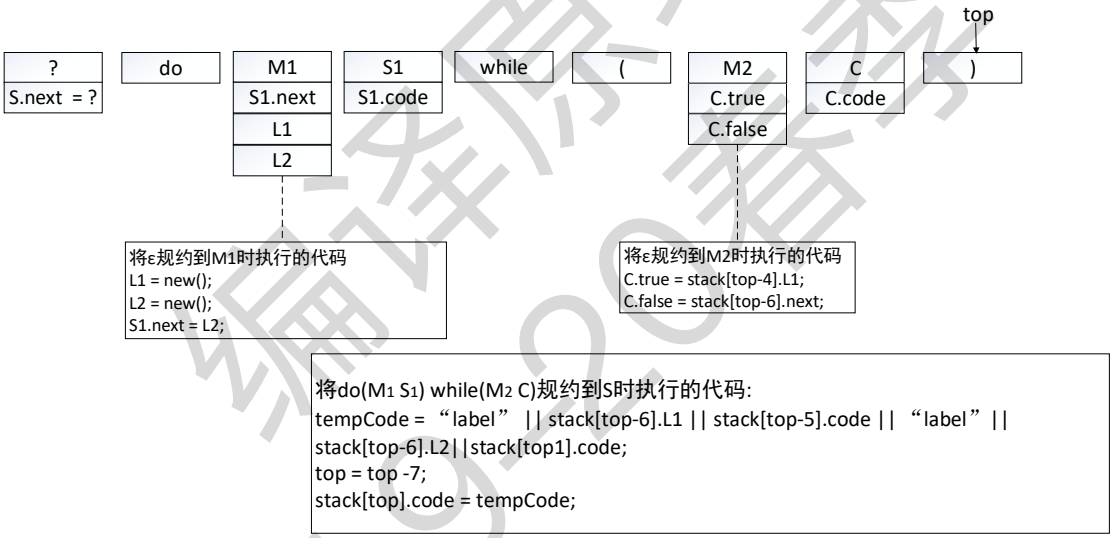
练习 **5.5.5** 图中用于申请标号的 **new()**，应该是 **newlabel()**，请注意。

5.5.5：按照5.5.4节的风格，将5.4.4中得到的每个SDD和一个LR语法分析器一起实现

a)



b)

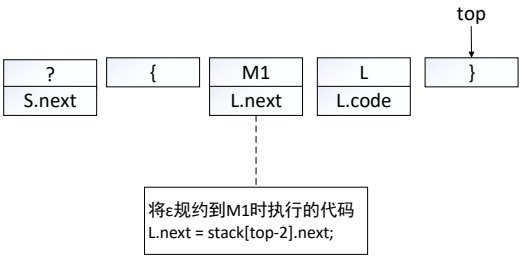


c)原文法包左递归，消除左递归后可得文法：

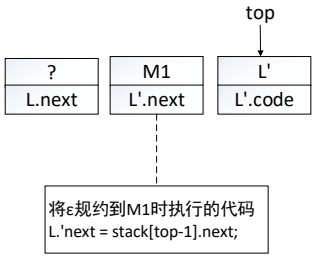
S -> ‘{’ L ’}’

L ->L’

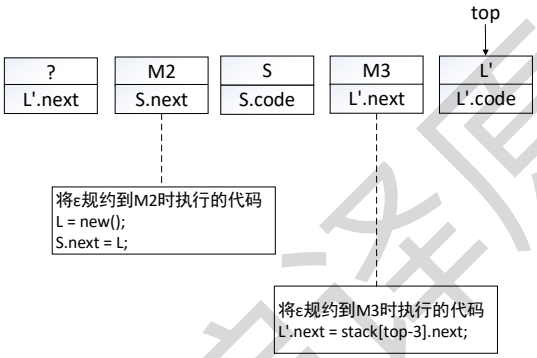
L’ –> SL’ | ε



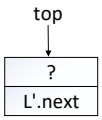
将{M1L}规约到S时执行的代码
tempCode = stack[top-1].code;
top = top － 3;
stack[top].code = tempCode;



将M1L'规约到L时执行的代码
tempCode = stack[top].code;
top = top － 1;
stack[top].code = tempCode;



将M2 S M3 L'规约到L'时执行的代码
tempCode = stack[top-2].code ||"label"||stack[top-3].next||
stack[top].code;
top = top － 3;
stack[top].code = tempCode;



将ε规约到L'时执行的代码
tempCode = “”;
top = top + 1;
stack[top].code = tempCode;

第六章作业

练习 **6.1.2**

6.1.2：为下列表达式构建DAG，且指出它们的每个字表达式的值编码
假定+是左结合的

3) a + a + (a + a + a + (a + a + a + a))

3) a + a + (a + a + a + (a + a + a + a))

子表达式的值编码如下：

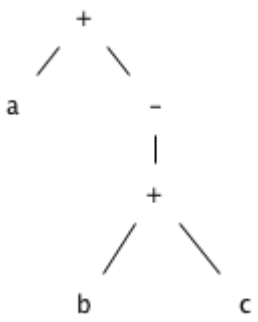
1	id	a	
2	+	1	1
3	+	2	1
4	+	3	1
5	+	3	4
6	+	2	5

练习 **6.2.1**

6.2.1：将算术表达式 a + - (b + c) 翻译成
1)抽象语法树
2)四元式序列
3)三元式序列
4)间接三元式序列

a + - (b + c)

1) 抽象语法树



2) 四元式序列

	op	arg1	arg2	result
0	+	b	c	t1
1	uminus	t1		t2
2	+	a	t2	t3

3) 三元式序列

	op	arg1	arg2
0	+	b	c
1	uminus	(0)	
2	+	a	(1)

4) 间接三元式序列

	statment
100	(0)
101	(1)
102	(2)

	op	arg1	arg2
0	+	b	c
1	uminus	(0)	
2	+	a	(1)

练习 **6.3.1**

6.3.1：确定下列声明序列中各个标识符的类型和相对地址

```
float x;
record { float x; float y } p;
record { int tag; float x; float y } q;
```

SDT

```
S ->                        {top = new Evn(); offset = 0;}
      D
D -> T id;                  {top.put(id.lexeme, T.type, offset);
                              offset += T.width}
      D1
D -> ε
T -> int                    {T.type = interget; T.width = 4;}
T -> float                  {T.type = float; T.width = 8;}
T -> record '{'
                            {Evn.push(top), top = new Evn();
                             Stack.push(offset), offset = 0;}
      D '}'                 {T.type = record(top); T.width = offset;
                             top = Evn.top(); offset = Stack.pop();}
```

标识符类型和相对地址

line id　　type　　offset　　Evn

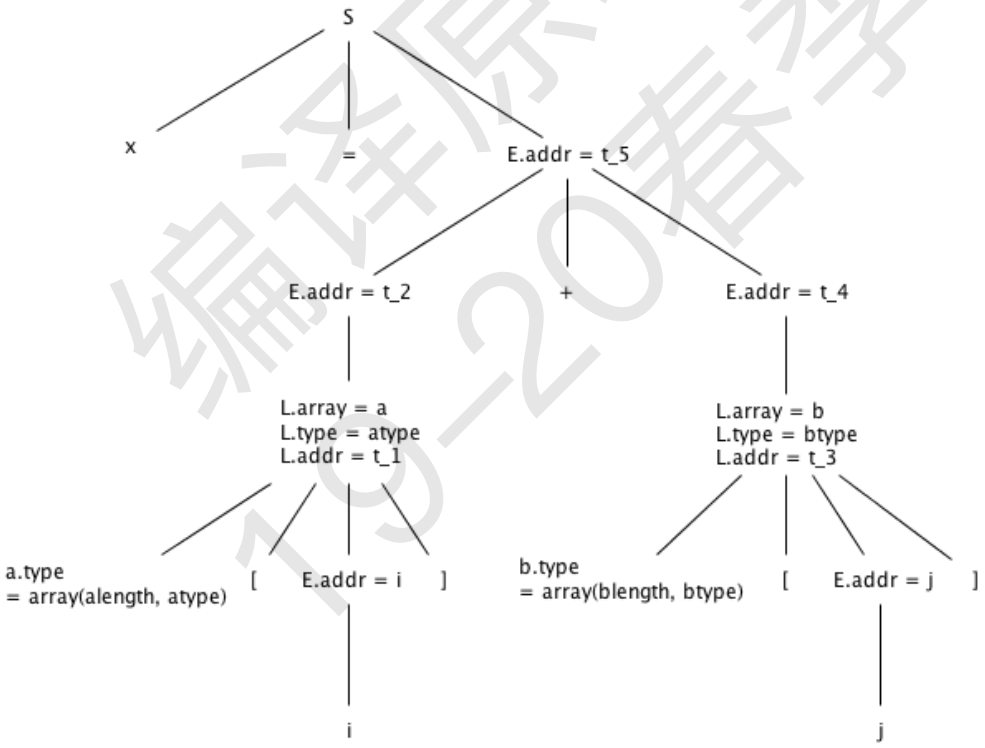
1)	x	float	0	1
2)	x	float	0	2
2)	y	float	8	2
2)	p	record()	8	1
3)	tag	int	0	3
3)	x	float	4	3
3)	y	float	12	3
3)	q	record()	24	1

6.4.3：使用图6-22的翻译方案翻译下列赋值语句

练习 **6.4.3**

1) x = a[i] + b[j]
2) x = a[i][j] + b[i][j]

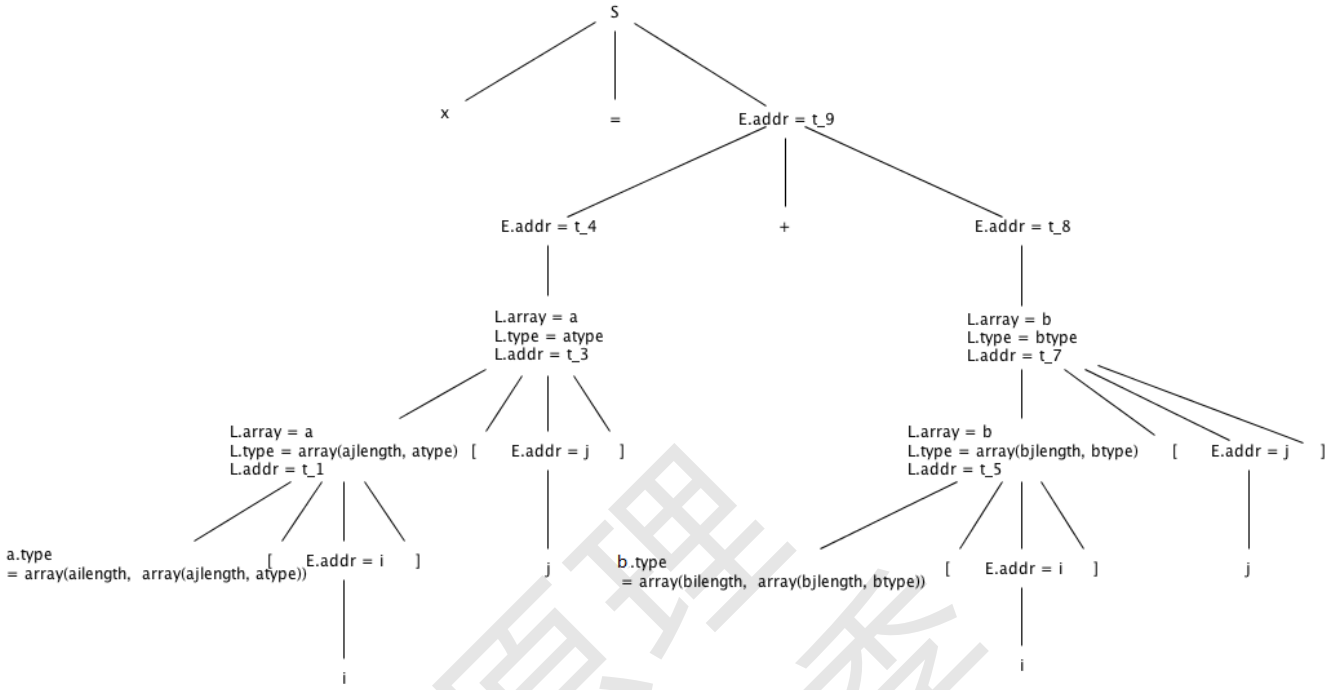
1) x = a[i] + b[j]
语法分析树如下：



三地址代码如下：

t_1 = i * awidth
t_2 = a[t_1]
t_3 = j * bwidth
t_4 = b[t_3]
t_5 = t_2 + t_4
x = t_5

2) x = a[i][j] + b[i][j]
语法分析树如下：



三地址代码如下：

t_1 = i * ai_width
t_2 = j * aj_width
t_3 = t_1 + t_2
t_4 = a[t_3]
t_5 = i * bi_width
t_6 = j * bj_width
t_7 = t_5 + t_6
t_8 = b[t_7]
t_9 = t_4 + t_8
x = t_9

练习 **6.5.1**

6.5.1：假定图6-26中的函数widen可以处理图6-26a的层次结构中的所有类型，翻译下列表达式。假定c和d是字符型，s和t是短整型，i和j是整型，x是浮点型

1) x = s + c
2) i = s + c
3) x = (s + c) * (t + d)

1) x = s + c
t1 = (int) s
t2 = (int) c
t3 = t1 + t2
x = (float) t3

2) i = s + c
t1 = (int) s
t2 = (int) c
i = t1 + t2

3) x = (s + c) * (t + d)
t1 = (int) s
t2 = (int) c
t3 = t1 + t2
t4 = (int) t
t5 = (int) d
t6 = t4 + t5
t7 = t3 * t6
x = (float) t7

练习 **6.6.1**

6.6.1：在图6-36的语法制导定义中添加处理下列控制流构造的规则：

1）一个repeat语句， repeat S while B

2）一个for循环语句， for (S_1 ; B ; S_2) S_3

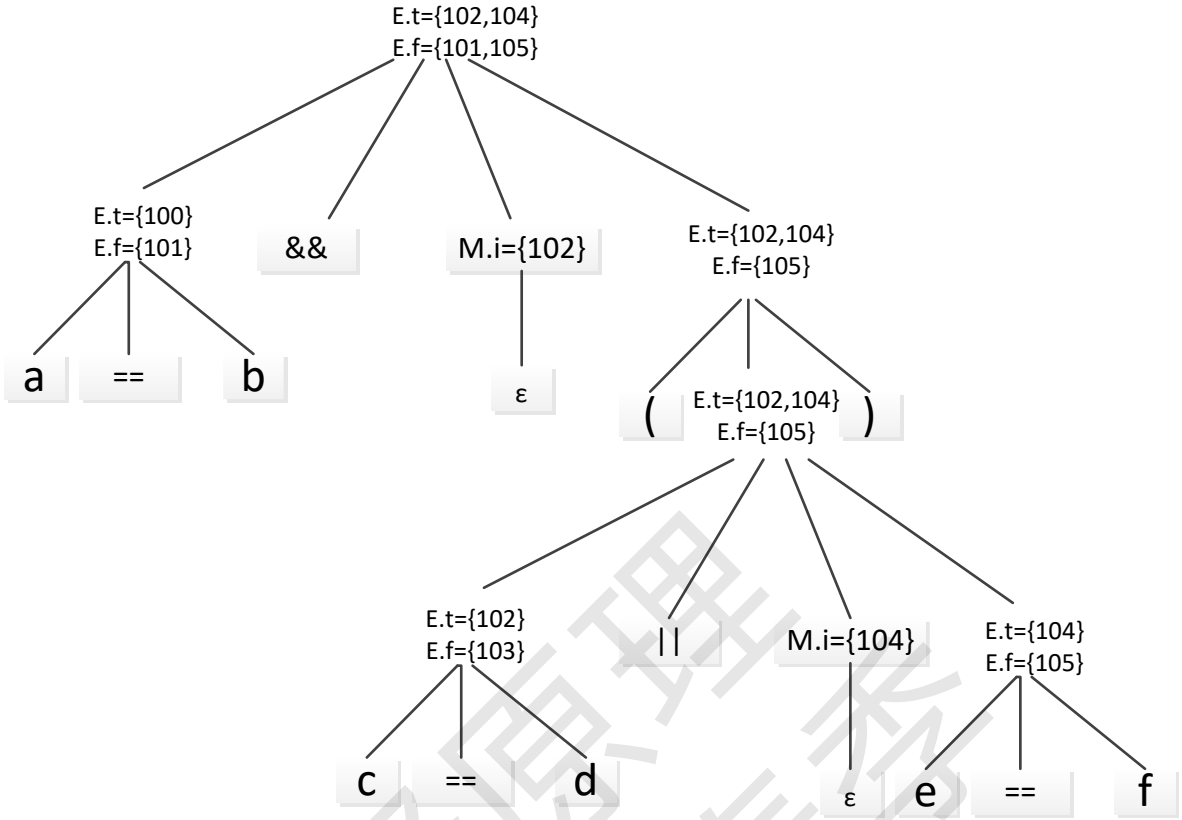
产生式	语义规则
S -> repeat S1 while B	S1.next = newlabel() B.true = newlabel() B.false = S.next S.code = label(B.true) || S1.code|| label(S1.next) || B.code
S -> for (S1; B; S2) S3	S1.next = newlabel() B.true = newlabel() B.false = S.next S2.next = S1.next S3.next = newlabel() S.code = S1.code|| lable(S1.next) || B.code|| lable(B.true) || S3.code || label(S3.next) || S2.code || gen('goto', S1.next)

练习 **6.7.1**

1) a == b && (c == d || e == f)
各个子表达式的 truelist 和 falselist 如下图所示：

6.7.1：使用图6-43中的翻译方案翻译下列表达式。给出每个子表达式的truelist和flaselist。你可以假设第一条被生成的指令的地址是100

1) a == b && (c == d || e == f)
2) (a == b || c == d) || e == f
3) (a == b && c == d) && e == f



具体分析过程如下：

首先，对于 a==b 按照 E → id1 relop id2 的语义动作进行规约，产生如下指令：

100: if a==b goto -

101: goto -

对于 E→ E1 and M E2 中的 M 记录了 E2 的入口，所以 M.i={102}；

对于 c==d 按照 E → id1 relop id2 的语义动作进行规约，产生如下指令：

102: if c==d goto -

103: goto -

对于 E→ E1 or M E2 中的 M 记录了 E2 的入口，所以 M.i={104}

对于 e==f 按照 E → id1 relop id2 的语义动作进行规约，产生如下指令：

104: if e==f goto -

105: goto -

接着，用产生式 E→ E1 or M E2 进行规约，执行以下动作：

```
{
    backpatch( E1.falselist, M.quad );
    E.truelist := merge(E1.truelist , E2.truelist );
    E.falselist := E2.falselist
}
```

先执行 backpatch(E1.falselist, M.quad)，即 backpatch({103}, 104)，所以有

100: if a==b goto -

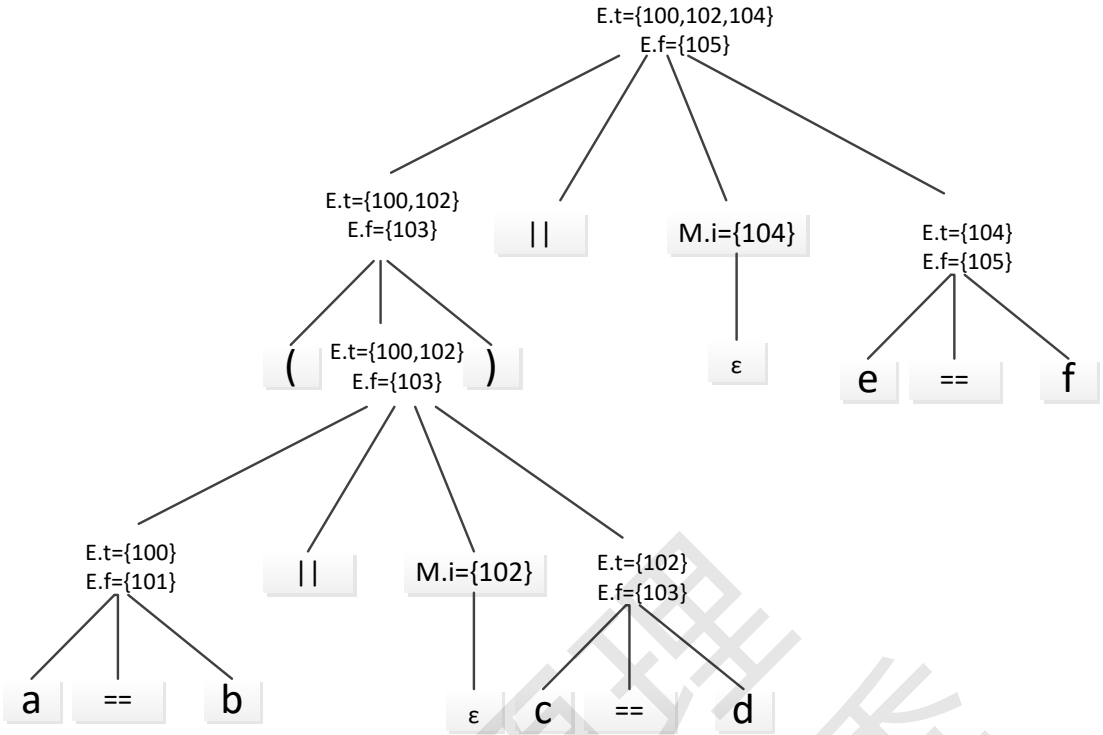
101: goto -

102: if c==d goto -

103: goto 104

104: if e==f goto -
105: goto -
接着执行 E.truelist := merge(E1.truelist , E2.truelist)，所以 E.t={102,104}，
接着执行 E.falselist := E2.falselist，所以 E.f={105}
然后用产生式 E→(E1)进行规约，执行以下动作：
{
    E.truelist := E1.truelist;
    E.falselist := E1.falselist
}
直接把各自的 truelist 和 falselist 拷贝过去，所以有 E.t={102,104}，E.f={105}
然后用产生式 E→E1 and M E2 进行规约，执行以下动作：
{
    backpatch(E1.truelist, M.quad);
    E.truelist := E2.truelist;
    E.falselist := merge(E1.falselist , E2.falselist)
}
先执行 backpatch(E1.truelist, M.quad)即 backpatch({100},102)，所以有：
100: if a==b goto 102
101: goto -
102: if c==d goto -
103: goto 104
104: if e==f goto -
105: goto -
接着执行 E.truelist := E2.truelist，所以 E.t={102,104}
接着执行 E.falselist := merge(E1.falselist , E2.falselist)，所以 E.f={101,105}
如果分别用 L1 和 L2 表示整个表达式的真、假两个出口，则最终生成的指令如下：
100: if a==b goto 102
101: goto L2
102: if c==d goto L1
103: goto 104
104: if e==f goto L1
105: goto L2

2) (a == b || c == d) || e == f
与 1)类似，可以得到各个子表达式的 truelist 和 falselist 如下：

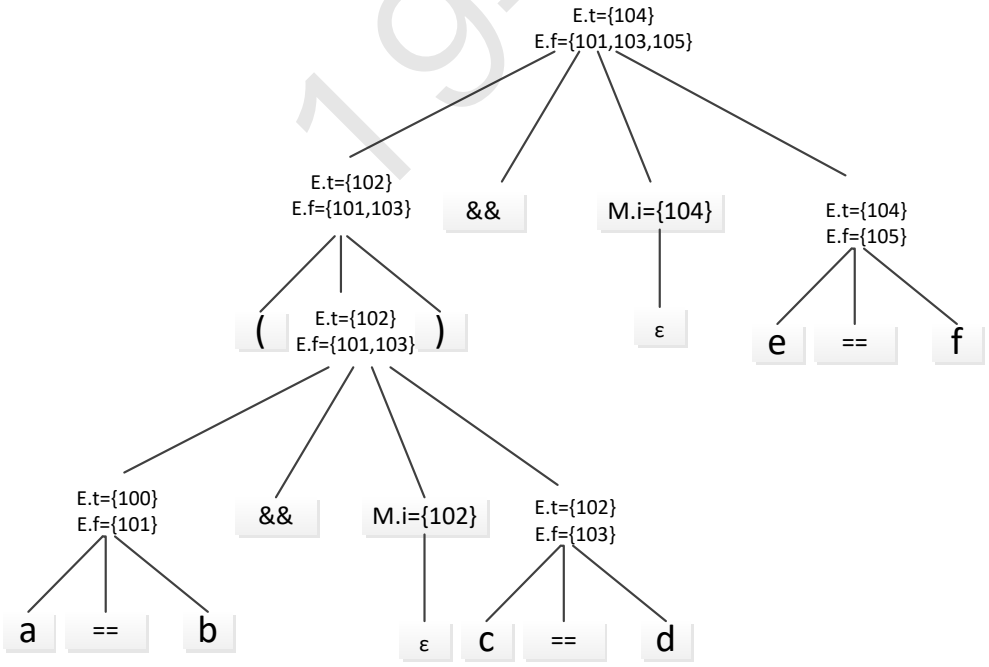


如果分别用 L1 和 L2 表示整个表达式的真、假两个出口，则最终生成的指令如下：

100: if a==b goto L1
101: goto 102
102: if c==d goto L1
103: goto 104
104: if e==f goto L1
105: goto L2

3) (a == b && c == d) && e == f

与 1)类似，可以得到各个子表达式的 truelist 和 falselist 如下：



如果分别用 L1 和 L2 表示整个表达式的真、假两个出口，则最终生成的指令如下：

```
100: if a==b goto 102
101: goto L2
102: if c==d goto 104
103: goto L2
104: if e==f goto L1
105: goto L2
```