第 1 章　编译概述

1.1　典型的编译程序划分为六个部分：词法分析、语法分析、语义分析、中间代码生成、代码优化、以及代码生成。各部分的主要作用如下：

词法分析：扫描字符串表示的源程序，根据词法规则识别出具有独立意义的单词符号，输出记号流。

语法分析：分析记号的逻辑结构，把记号流按语言的语法结构层次地分组，以形成语法短语。

语义分析：对语法成分的类型、含义进行检查，以保证程序各部分能够有机地结合在一起，并为以后生成目标代码收集如类型、目标地址等必要的信息。

中间代码生成：将源程序转换为一种中间表示形式，以便于进一步转换成目标代码。

代码生成：将中间形式代码转换成目标代码。

代码优化：对代码进行改进以获得高效的代码，即使之占用的空间少、运行速度快。

1.3　编译程序的伙伴工具主要有：预处理器、汇编程序、连接装配程序。它们的作用是：

预处理器：对源程序进行处理，产生编译程序的输入。预处理器主要完成宏处理、文件包含、语言扩充等功能。

汇编程序：有些编译程序产生汇编语言的目标代码，此时就需要由汇编程序作进一步的处理，生成可重定位的机器代码。

连接装配程序：完成连接和装入任务。所谓连接，即把几个可重定位的机器代码文件连接成一个可执行的程序，这些文件可以是分别编译或汇编得到的，也可能是由系统提供的、对任何需要它们的程序而言都可用的子程序组成的库文件。所谓装入，即读入可重定位的机器代码，修改需重定位的地址，把修改后的指令和数据放在内存中适当的地方或形成可执行文件。

1.4　翻译程序：把用某种语言书写的程序翻译成计算机能够执行的表示形式或直接执行这个程序的程序。

编译程序：源语言为高级语言、目标语言是某种机器的机器语言或汇编语言的翻译程序。

汇编程序：源语言是汇编语言、目标语言是机器语言的翻译程序。

解释程序：直接执行程序的翻译程序。

编译程序的遍：一“遍”是指对源程序或其中间表示形式从头到尾扫描一遍，并作相关的加工处理，生成新的中间表示形式或目标程序。

编译程序的前端：前端主要由与源语言有关而与目标机器无关的那些部分组成，通常包括词法分析、语法分析、语义分析和中间代码生成、符号表的建立、以及与机器无关的代码优化工作，当然，前端也包括相应的错误处理工作和符号表操作。

编译程序的后端：　后端由编译程序中与目标机器有关的部分组成，一般来讲，这些部分与源语言无关而仅仅依赖于中间语言。后端包括目标代码的生成、与机器有关的代码优化、以及相应的错误处理和符号表操作。

1.5　语法分析阶段可以发现关键字拼写错误、缺少运算对象、以及本应为常数，但却在数中出现了非数字字符的错误。

语义分析阶段可以发现实参与形参的类型不一致、所引用的变量没有定义的错误。

数组下标越界的错误可以在语义分析阶段、或执行时被发现。

第 3 章　词法分析

3.2　(1)　$0(0|1)^*0$ 表示的语言是：以 0 开头、以 0 结尾的，由 0、1 组成的符号串的全体。

(4)　$0^*10^*10^*10^*$表示的语言是：含有三个 1 的由 0、1 组成的符号串的全体。

3.4

(1)　num→ num2 | 0 num1 | 1 num1 | …… | 9 num1

num1→ num2 | 0 num1 | 1 num1 | …… | 9 num1

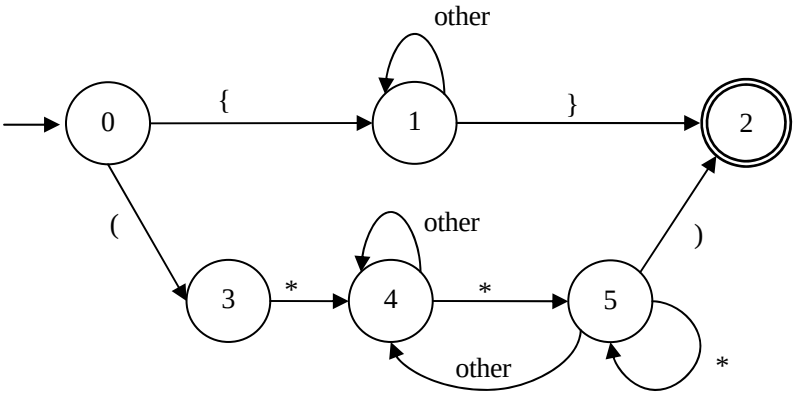
num2→ 2 | 4 | 6 | 8 | 0

(2)　num→ num2 | 1 num1 | 2 num1 | …… | 9 num1

num1→ num2 | 0 num1 | 1 num1 | …… | 9 num1

num2→ 2 | 4 | 6 | 8 | 0

3.8　识别这两种风格的注释的 DFA 如下：



3.9　(3)!=　(5)<=　不需要超前扫描

(1) =　(2) for　(4) +　需要超前扫描，因为==、+=、++、for_loop 等在 C 语言中都是合法的单词。

第 4 章　语法分析

4.1

首先，消除文法中的左递归，得到：

bexpr→bterm E’

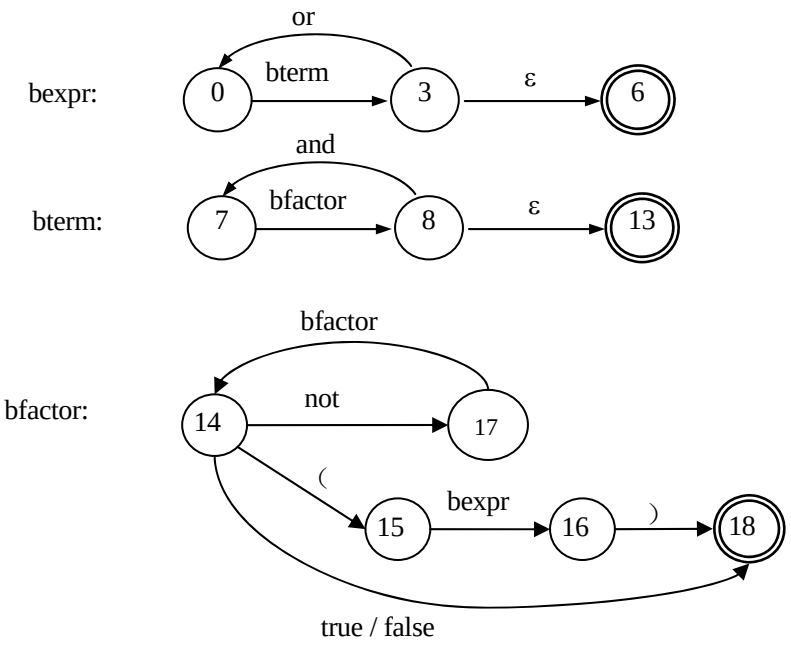
E’→or bterm E’ | ε

bterm→bfactor T’

T’→and bfactor T’ | ε

bfactor→not bfactor | (bexpr) | true | false

其次，构造状态转换图并化简，得到：



然后，根据状态转换图构造递归调用函数，代码主体结构描述如下：

bexpr 的过程：void proce_xpr(void)

```
{
   proc_term();
   if (char==‘or’) {
      forward pointer;
      proc_expr;
      }
}
```

bterm 的过程：void proc_term(void)

```
{
```

```
        proc_factor();
        if (char==‘and’) {
            forward pointer;
            proc_term();
        }
    }
```

bfactor 的过程：

```
void proc_factor(void)
{
  if (char==‘not’){
       forward pointer;
       proc_factor();
       }
   else if (char==‘(’) {
         forward pointer;
         proc_expr();
         if (char==‘)’)
              forward pointer;
          else    error();
      };
       else if (char==‘true’)||(char==‘false’)
                  forward pointer;
            else error();
}
```

4.3

(1) FIRST(A)={(, ε}, FOLLOW(A)={$,)}

(2) 由于 FIRST((A)A)∩FIRST(ε)=ϕ 并且 FIRST((A)A)∩FOLLOW(A)=ϕ

所以，该文法是 LL(1)文法。

4.5

(1) 消除文法中的左递归后，得到：

E→A | B

A→num | id

B→(L)

L→EL’

L’→EL’ | ε

(2) 为改写后的文法中的非终结符号构造 FIRST 和 FOLLOW 集合,如下：

	FIRST	FOLLOW
E	(, num, id	$, (,), num, id
A	num, id	$, (,), num, id
B	(	$, (,), num, id
L	(, num, id	)
L’	(, num, id, ε	)

(3) 改写后的文法是 LL(1)文法，因为：

对于产生式：

E→A | B　　FIRST(A)∩FIRST(B)=ϕ

A→num | id　　FIRST(num)∩FIRST(id)=ϕ

B→(L)

L→EL’

L’→EL’ | ε　　FIRST(EL’)∩FOLLOW(L’)=ϕ

构造其 LL(1)分析表，如下：

	（	)	,	num	id	$
E	E→B			E→A	E→A	
A				A→num	A→id	
B	B→(L)					
L	L→EL’			L→EL’	L→EL’	
L’	L’→EL’	L’→ε		L’→EL’	L’→EL’	

(4) 对于输入符号串(a(b(2))(c))的预测分析过程如下：

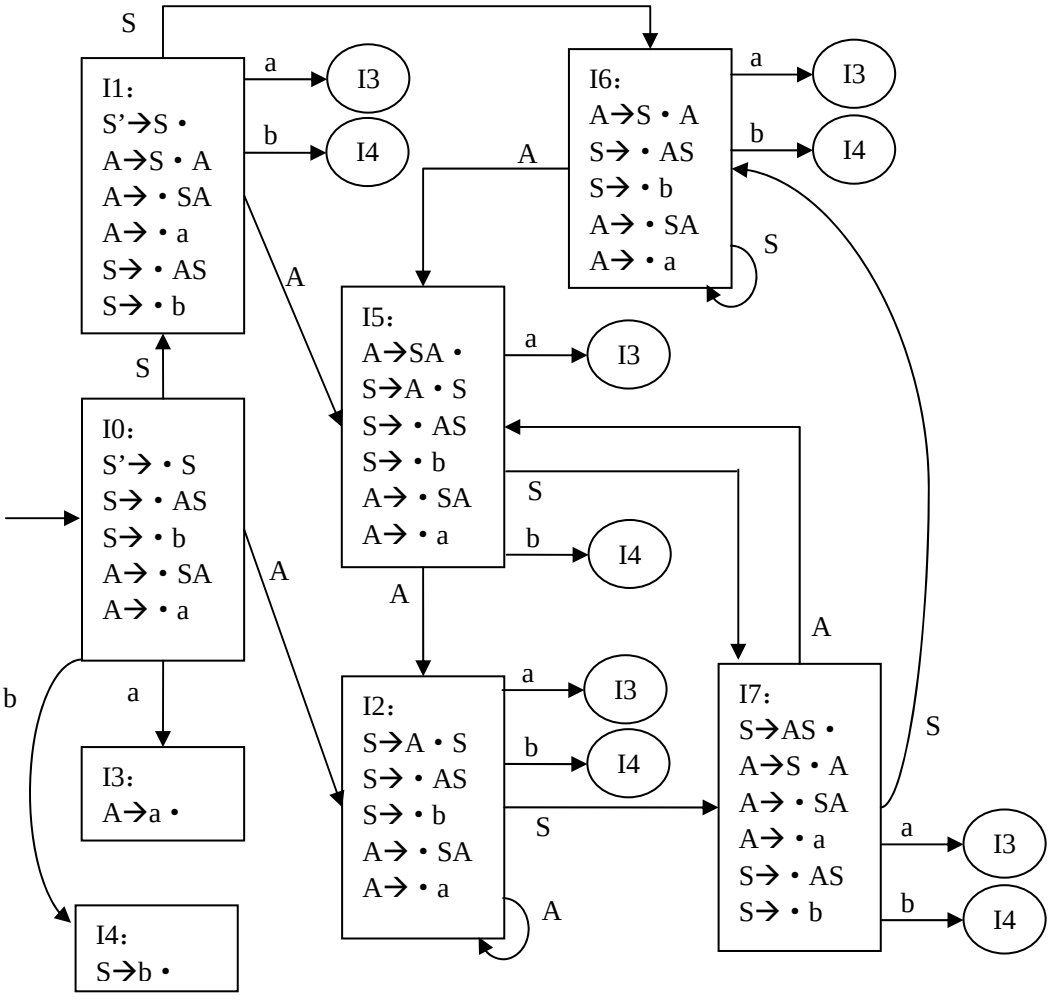
步骤	栈	输入	输出
(1)	$E	(a(b(2))(c))$	E→B
(2)	$B	(a(b(2))(c))$	B→(L)
(3)	$)L(	(a(b(2))(c))$	
(4)	$)L	a(b(2))(c))$	L→EL’
(5)	$)L’E	a(b(2))(c))$	E→A
(6)	$)L’A	a(b(2))(c))$	
(7)	$)L’id	a(b(2))(c))$	
(8)	$)L’	(b(2))(c))$	L’→EL’
(9)	$)L’E	(b(2))(c))$	E→B
(10)	$)L’B	(b(2))(c))$	B→(L)
(11)	$)L’)L(	(b(2))(c))$	
(12)	$)L’)L	b(2))(c))$	L→EL’
(13)	$)L’)L’E	b(2))(c))$	E→A
(14)	$)L’)L’A	b(2))(c))$	A→id
(15)	$)L’)L’id	b(2))(c))$	
(16)	$)L’)L’	(2))(c))$	L’→EL’
(17)	$)L’)L’E	(2))(c))$	E→B
(18)	$)L’)L’B	(2))(c))$	B→(L)
(19)	$)L’)L’)L(	(2))(c))$	
(20)	$)L’)L’)L	2))(c))$	L→EL’
(21)	$)L’)L’)L’E	2))(c))$	
(22)	$)L’)L’)L’A	2))(c))$	A→num
(23)	$)L’)L’)L’num	2))(c))$	
(24)	$)L’)L’)L’	))(c))$	L’→ε
(25)	$)L’)L’)	))(c))$	
(26)	$)L’)L’	)(c))$	L’→ε
(27)	$)L’)	)(c))$	

(28)	$)L’	(c))$	L’→EL’
(29)	$)L’E	(c))$	E→B
(30)	$)L’B	(c))$	B→(L)
(31)	$)L’)L(	(c))$	
(32)	$)L’)L	c))$	L→EL’
(33)	$)L’)L’E	c))$	E→A
(34)	$)L’)L’A	c))$	A→id
(35)	$)L’)L’id	c))$	
(36)	$)L’)L’	))$	L’→ε
(37)	$)L’)	))$	
(38)	$)L’	)$	L’→ε
(39)	$)	)$	
(40)	$	$	接受

4.9

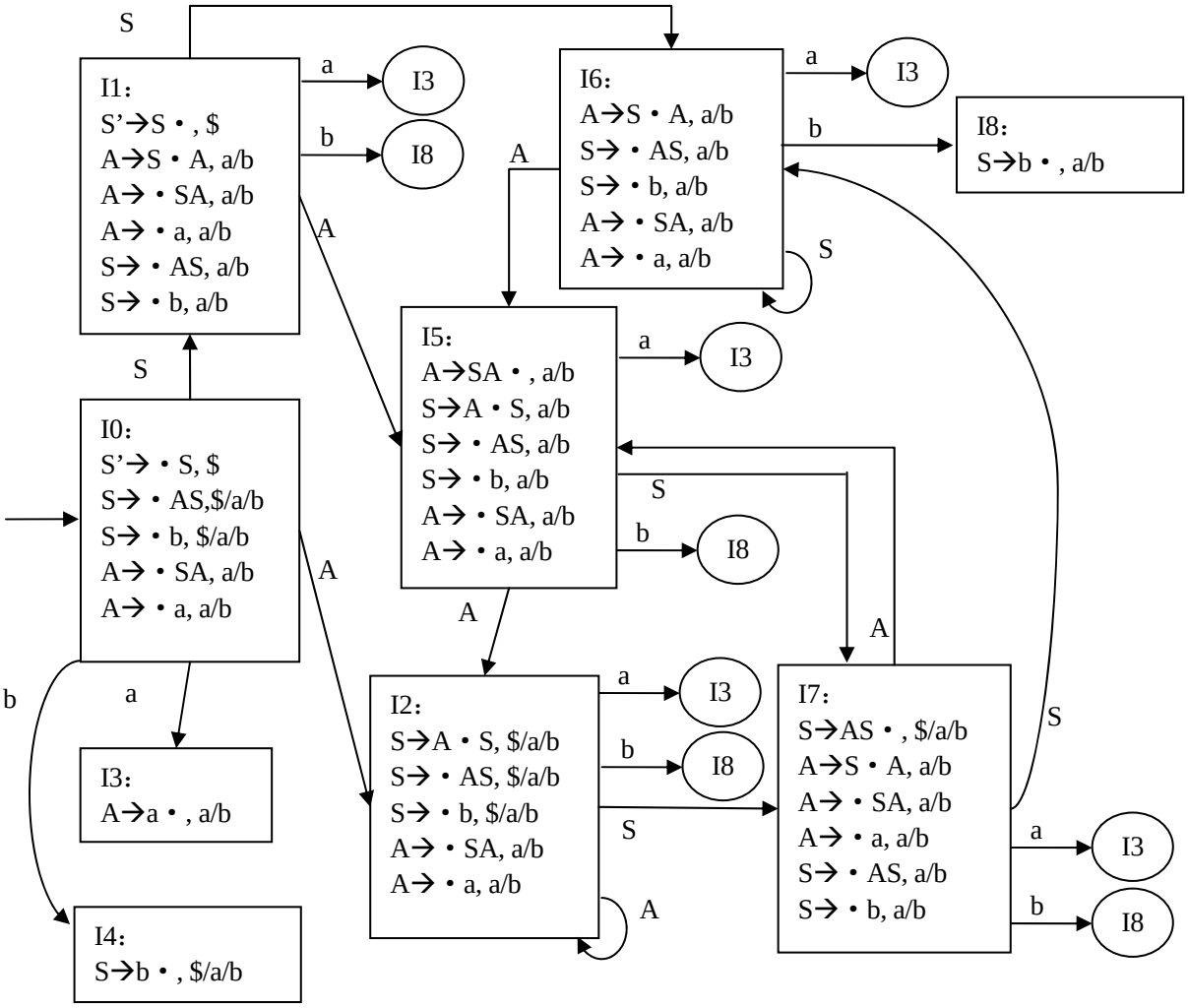
(1) 首先拓广文法，(0) S’→S (1) S→AS (2) S→b (3) A→SA (4) A→a

构造其 LR(0)项目集规范族及识别它所有活前缀的 DFA 如下：



(2) 该文法不是 SLR(1)文法，因为在状态 I5 中存在 SLR 方法无法解决的冲突。对于归约项目 A→SA・　由于 FOLLOW(A)={a,b}，M[5,b]=reduce A→SA
而对于移进项目 S→・b 则有 M[5,b]=shift 4
所以，该文法不是 SLR(1)文法。

(3) 首先，构造该文法的 LR(1)项目集规范族及识别它所有活前缀的 DFA 如下



在状态 I5 中，项目 [A→SA・, a/b] 和 [A→・a, a/b] 存在移进-归约冲突，所以该文法不是 LR(1)文法。

4.14
首先证明该文法是 LL(1)文法。
构造文法符号的 FIRST 和 FOLLOW 集合，如下

	FIRST	FOLLOW
S	a, b	$
A	ε	a, b
B	ε	a, b

对于产生式：S→AaAb|BbBa　有：FIRST(AaAb)∩ FOLLOW(BbBa)=ϕ

LL(1)分析表如下

	a	b	$
S	S→AaAb	S→BbBa	
A	A→ε	A→ε	
B	B→ε	B→ε	

分析表中无冲突，是 LL(1)分析表，所以，该文法是 LL(1)文法。

再证该文法不是 SLR(1)文法。

拓广文法如下：

(0) S'→S

(1) S→AaAb (2) S→BbBa

(3) A→ε (4) B→ε

构造文法的ε活前缀的 LR(0)有效项目集如下：

I0：S'→·S S→·AaAb S→·BbBa A→· B→·

由于 FOLLOW(A)=FOLLOW(B)={a,b}，故该集合中的项目 A→· 和 B→·存在归约-归约冲突。

所以，该文法不是 SLR(1)文法。

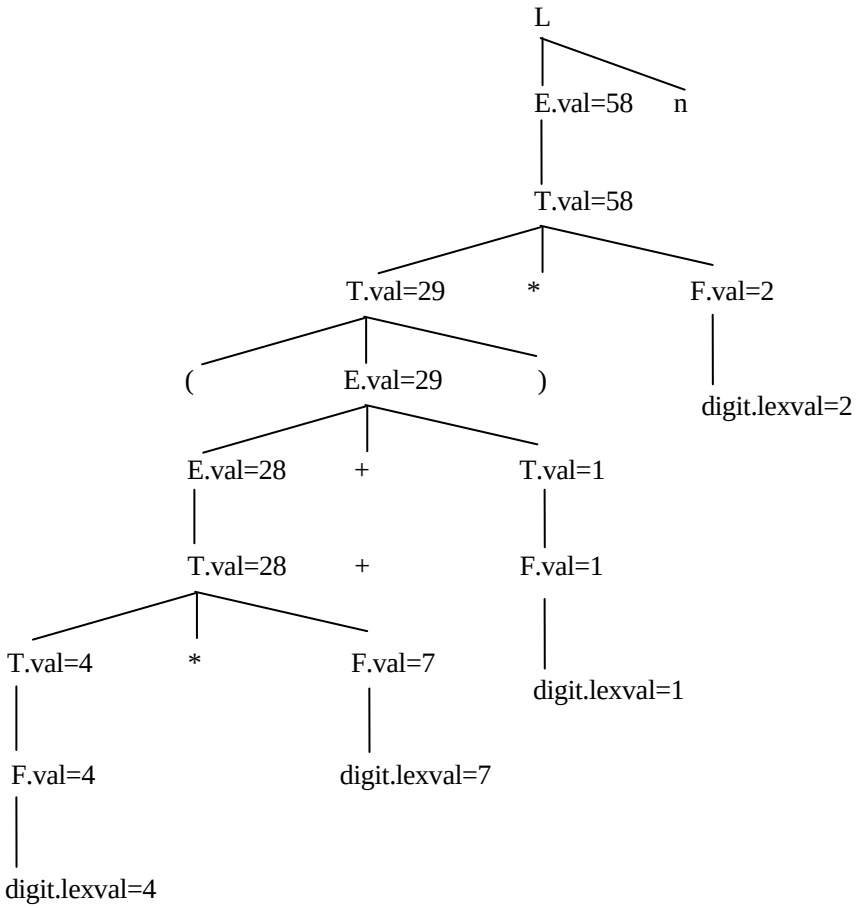
4.16

提示：

首先拓广文法，其次，构造文法的 LR(0)有效项目集规范族及识别活前缀的 DFA，可以看出，这些 LR(0)有效项目集中，要么只含有一个归约项目，要么只含有移进或待约项目，所以，文法是 LR(0)文法，同时也是 SLR(1)、LR(1)、LALR(1)文法。

第 5 章　语法制导翻译技术

5.1　表达式(4*7+1)*2 的注释分析树如下：



5.2　引入以下属性

E.val、T.val、F.val：分别表示 E、T、F 所代表的表达式的值。

E’.i、T’.i：分别记录在展开 E’和 T’之前所得到的表达式的值。

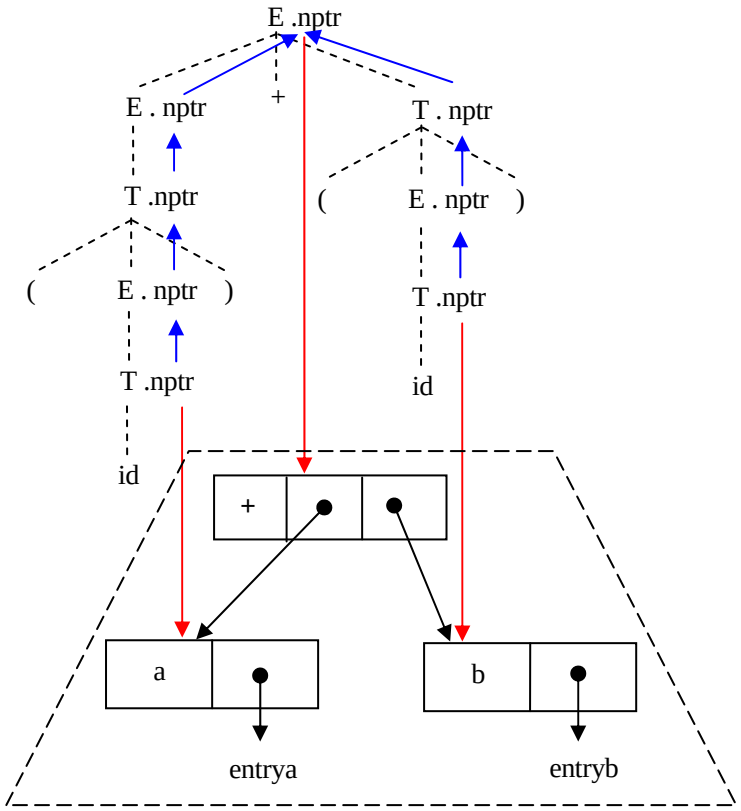
num.lexval：代表数 num 的词法值。

为表达式求值的语法制导定义

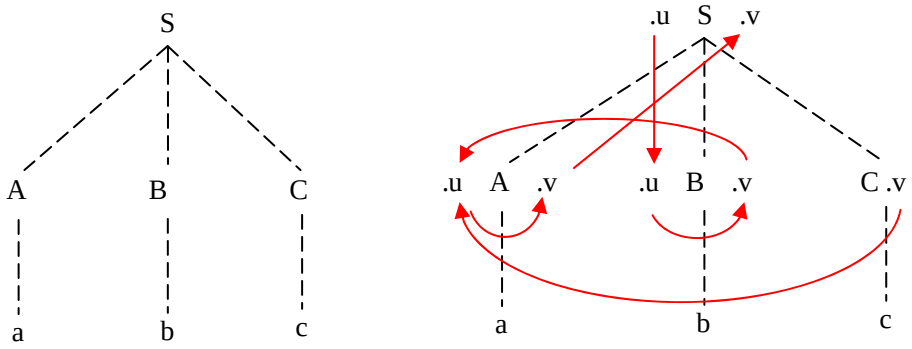
产生式	语义规则
E→TE’	E’.i=T.val E.val=E’.val
E’→+TE’$_1$	E’$_1$.i=E’.i+T.val E’.val=E’$_1$.val
E’→-TE’$_1$	E’$_1$.i=E’.i-T.val E’.val=E’$_1$.val
E’→ε	E’.val=E’.i

T→FT’	T’.i=F.val T.val=T’.val
T’→*FT’$_1$	T’$_1$.i= T’.i*F.val T’.val=T’$_1$.val
T’→ε	T’.val=T’.i
F→(E)	T.val=E.val
F→num	T.val=num.lexval

5.3 为表达式((a)+(b))建立的分析树和语法树如下，其中上半部分所示为分析树，图中虚线框起来的部分为语法树。



5.4 (1) 符串 abc 的分析树及相应的依赖图如下，右图中实线部分的是依赖图。



分析树　　依赖图

(2) 根据上述依赖图，可以写出如下的一个有效的语义规则执行顺序：

C.v=2

B.u=S.u

B.v=B.u

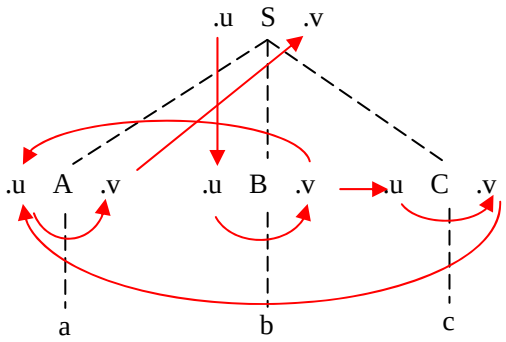
A.u=B.v+C.v

A.v=3*A.u

S.v=A.v

(3) 假设分析 abc 之前 S.u 的初值为 5，则翻译完成时 S.v 的值是：21

(4) 根据修改后的与法制导定义，可以构造出字符串 abc 的分析树和依赖图图下：



根据该依赖图，有如下的语义规则计算顺序：

B.u=S.u

B.v=B.u

C.u=B.v

C.v=C.u-2

A.u=B.v+C.v

A.v=3*A.u

S.v=A.v

若分析开始时，S.u 的初值为 5，则翻译完成时 S.v 的值是 24。

5.6

(1) 设计 E.type、T.type：表示相应表达式的类型。

确定每个子表达式类型的语法制导定义如下：

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) 设计以下属性：

E.type、T.type：表示相应表达式的类型。

E.code、T.code：表示相应表达式的中缀表示。

满足要求的语法制导定义如下：

产生式	语义规则
L→E	Print(E.code)

E→E_1+T	If (E_1.type==integer)&&(T.type==integer) {
	E.type=integer; E.code='+'|| E_1.code||T.code; }
	Else { E.type=real;
	if (E_1.type==integer) E_1.code=inttoreal(E_1.code);
	else if (T.type==integer) T.code=inttoreal(T.code);
	E.code='+'|| E_1.code||T.code; }

E→T	E.type=T.type E.code=T.code
T→num.num	T.type=real T.code=num.num.lexval
T→num	T.type=integer T.code=num.lexval

5.9

(1) 设计综合属性 T.type，记录声明语句中关键字确定的类型信息。

设计继承属性 L.in，用于将类型信息传递给声明语句中的名字。

设计过程 addtype（id.entry, type），实现将类型信息 type 写入符号表中由 id.entry 指向的记录中。

确定声明语句中变量类型的语法制导定义如下：

产生式	语义规则
D→LT	L.in=T.type
T→integer	T.type=integer
T→real	T.type=real
L→L_1,id	L_1.in=L.in addtype(id.entry, L.in)
L→id	addtype(id.entry,L.in)

(2) 该定义不是 L 属性定义，因为在 D→LT 的语义规则中 L.in 依赖它右边符号 T 的属性，不满足 L 属性定义对继承属性的约束。

5.11

(1) 该语法制导定义中定义了综合属性 B.ht 和继承属性 B.ps，B.ps 要么取值常数，要么依赖于产生式左部符号 B 的继承属性，满足 L 属性定义对语义规则的约束，所以该语法制导定义是 L 属性定义。

(2) 该语法制导定义的翻译方案如下：

S→{B.ps=10} B {S.ht=B.ht}

B→{B_1.ps=B.ps} B_1 {B_2.ps=B.ps} B_2 {B.ht=max(B_1.ht,B_2.ht)}

B→{B_1.ps=B.ps} B_1 sub {B_2.ps=shrink(B.ps)} B_2 {B.ht=disp(B_1.ht,B_2.ht)}

B→text {B.ht=text.h×B.ps}

(3) 将插入产生式中的语义动作去除之后，就可以用 LR 方法进行翻译。

为此，需要引入标记非终结符号 L、M 和 N，以及相应的产生式 L→ε、M→ε 和 N→ε。分别用 L、M 和 N 实现继承属性的初始化和产生式中第二个 B 的继承属性的计算。并设计继承属性 i 和综合属性 s。

相应的语法制导定义如下：

产生式	语义规则
S→LB	B.ps=L.s S.ht=B.ht
L→ε	L.s=10
B→B_1MB_2	B_1.ps=B.ps M.i=B.ps B_2.ps=M.s B.ht=max(B_1.ht,B_2.ht)
B→B_1subNB_2	B_1.ps=B.ps N.i=B.ps B_2.ps=N.s B.ht=disp(B_1.ht,B_2.ht)
B→text	B.ht=text.h×B.ps
M→ε	M.s=M.i
N→ε	N.s=shrink(N.i)

其中，L 的作用是完成对继承属性的初始化工作。从 S 的产生式 S→LB 可知，分析程序首先把 L 入栈，随之一起入栈的还有 L.s，即继承属性 B.ps 的初值。在以后的分析过程中，L 保留在栈中，这样当在分析 B 的过程中需要继承属性 B.ps 的值时，就可以直接从 L.s 处取得。

产生式 B→B_1MB_2 中 M 的作用是使 B.ps 的值在栈中的位置恰好在 B_2 的下面。

产生式 B→B_1subNB_2 中的 N 的用是，通过复制规则 N.i=B.ps 继承的属性值是 B_2.ps 要依赖的，并且通过规则 N.s=shrink(N.i)计算出了 B_2.ps 的值，并保存在 N.s 处。

引入这三个标记非终结符号后，在分析过程中，每当需要 B 的属性 B.ps 时，都可以知道它在栈中的位置，并直接从栈中取得。

相应的翻译方案如下：

S→L{B.ps=L.s} B {S.ht=B.ht}

B→{B_1.ps=B.ps} B_1 M{B_2.ps=M.s} B_2 {B.ht=max(B_1.ht,B_2.ht)}

B→{B_1.ps=B.ps} B_1 sub N{B_2.ps=N.s)} B_2 {B.ht=disp(B_1.ht,B_2.ht)}

B→text {B.ht=text.h×B.ps}

L→ε{L.s=10}

M→ε{M.s=M.i}

N→ε{N.s=shrink(N.i)}

(4) 由于上述翻译方案中的所有继承属性都由复制规则赋值，所以，语法制导定义的实现都是通过跟踪它们在 val 栈中的位置来获得属性值的。设 top 和 ntop 分别是归约前和归约后的栈顶指针。则实现上述翻译方案的代码如下：

产生式	语义规则
S→LB	val[ntop]=val[top]
L→ε	val[ntop]=10
$B\rightarrow B_1MB_2$	val[ntop]=max(val[top-2],val[top])
$B\rightarrow B_1subNB_2$	val[ntop]=disp(val[top-3],val[top])
B→text	val[ntop]=val[top]×val[top-1]
M→ε	val[ntop]=val[top-1]
N→ε	val[ntop]=shrink(val[top-2])

(5) 假定有输入符号串 Esub1.val，并且假定词法分析识别出 E、sub、1、.、和 val 的属性 h 的值均为 1，并且 Shrink(x)=0.3*x，disp(x,y)=x+y，则其分析过程如下：

步骤	栈	输入	分析动作
(1)	S: V:	Esub1.val$	归约 L→ε
(2)	S: L V:10	Esub1.val$	移进
(3)	S: L E V:10 1	sub1.val$	归约 B→text
(4)	S: L B V:10 10	sub1.val$	移进
(5)	S: L B sub V:10 10 -	1.val$	归约 N→ε
(6)	S: L B sub N V:10 10 - 3	1.val$	移进
(7)	S: L B sub N 1 V:10 10 - 3 1	.val$	归约 B→text
(8)	S: L B sub N B V:10 10 - 3 3	.val$	归约 B→B sub N B
(9)	S: L B V:10 13	.val$	归约 M→ε
(10)	S: L B M V:10 13 10	.val$	移进
(11)	S: L B M . V:10 13 10 1	val$	归约 B→text
(12)	S: L B M B V:10 13 10 10	val$	归约 B→B M B
(13)	S: L B V:10 13	val$	归约 M→ε
(14)	S: L B M V:10 13 10	val$	移进

(15)	S: L B M val V:10 13 10 1	$	归约 B→text
(16)	S: L B M B V:10 13 10 10	$	归约 B→B M B
(17)	S: L B V:10 13	$	归约 S→LB
(18)	S: S V:13	$	接受

从上述步骤(3)、(7)、(11)、(15)可以看出，每当把一个右部归约为 B 时，继承属性 B.ps 的值在栈中的位置总是恰好在该被归约右部的下面。

5.15

(1) 设计综合属性 S.num 和 L.num，分别记录由 S 和 L 产生的符号串中出现的配对括号个数。满足要求的语法制导定义如下：

产生式	语义规则
S'→S	Pring(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

(2) 设计继承属性 S.deep 和 L.deep，分别记录当前符号串的嵌套深度。满足要求的翻译方案如下：

翻译方案
S'→{S.deep=0} S
S→({L.deep=S.deep+1} L)
S→a { print S.deep}
L→ {L_1.deep=L.deep} L_1, {S.deep=L.deep} S
L→ {S.deep=L.deep} S

5.16 设计综合属性 L.val 和 L.length，其中 L.val 记录识别出的二进制数字串相应的十进制值，L.length 记录识别出的二进制数字串的长度。

确定 S.val 值得语法制导定义如下：

产生式	语义规则
S→L_1.L_2	S.val=L_1.val+L_2.val/$2^{L2.length}$
S→L	S.val=L.val
L→L_1B	L.val=L_1.val*2+B.val L.length=L_1.length+1
L→B	L.val=B.val L.length=1
B→0	B.val=0
B→1	B.val=1

第 6 章　语义分析

6.2 在符号表上的操作有四种：插入、检索、定位、重定位。
插入操作：当识别出一个新的名字时被调用。
检索操作：插入操作之前执行，目的是“查重”；
　遇到名字引用时执行，目的是从符号表取得名字的信息，以便进行类型检查或代码生成。
定位操作：识别到一个块的开始时执行，目的是建立该块的符号子表。
重定位操作：识别到一个块的结束时执行，目的是从逻辑上或物理上删除块的符号表，实现作用域规则。

6.3 允许在一个声明语句中声明多个名字时的翻译方案如下：

P→ { offset=0 } D
D→D;D
D→L
L→id : T { L.type=T.type; L.width=T.width ;
　　enter(id,name, T.type, offset);
　　offset=offset+T.width }
L→id, L_1 { L.type=L_1.type; L.width=L_1.width;
　　enter(id.name, L_1.type, L_1.width);
　　offset=offset+L_1.width}
T→integer { T.type=integer; T.width=4 }
T→real { T.type=real; T.width=8 }
T→array [num] of T_1 { T.type=array(num.val, T_1.type);
　　T.width=num.val×T_1.width }
T→↑T_1 { T.type=pointer(T_1.type); T.width=4 }

6.9 (1) array(1..100, pointer(real))
(2) array(0..9, array(-10..10, integer))
(3) (integer→pointer(integer)) → record((a×integer)×(b×char))

6.10 CELL 的类型表达式：record((a×integer)×(b×integer))
PCELL 的类型表达式：pointer(record((a×integer)×(b×integer)))
foo 的类型表达式：array(0..99, CELL)
bar 的类型表达式：int×CELL→PCELL

6.12 (1) 不会出现类型错误
(2) 会出现类型错误
原因：C 语言对于结构类型遵循名字等价，x 和 y 名字等价，但与 z 名字不等价。

6.13 (1) x 和 y 名字等价， z 和 w 名字等价
(2) x, y, z 和w 都结构等价
(3) 全部结构等价

第 7 章　运行环境

7.1　Pascal 语言遵循静态作用域规则。

第(8)行引用的 a 是在第(3)行声明的，第(9)行引用的 a 和 b 是在第(4)行声明的。

第(10)行引用的 b 是在第(5)行声明的，第(11)行引用的 a 和 b 是在第(6)行声明的。

第(12)行引用的 a.a 和 a.b 是在第(3)行声明记录变量 a 中的 a 域和 b 域，域类型在第(4)行声明。

第(12)行引用的 b.a 和 b.b 是在第(5)行声明记录变量 b 中的 a 域和 b 域，域类型在第(6)行声明。

第(15)行引用的 b 是在第(2)行声明中的过程名。

7.2　该程序在语法上是正确的，但却是错误的，无法实现加 1 的功能。

原因是：在 C 语言中，后缀一元算符的优先级高于前缀一元算符的优先级，所以*x++中的加 1 动作作用在指针变量 x 上，而不是*x 上，原来 x 所指向的存储单元的值没有发生变化。正确的代码应该是：

```
void inc_1 (int* x)
{   (*x)++;   }
```

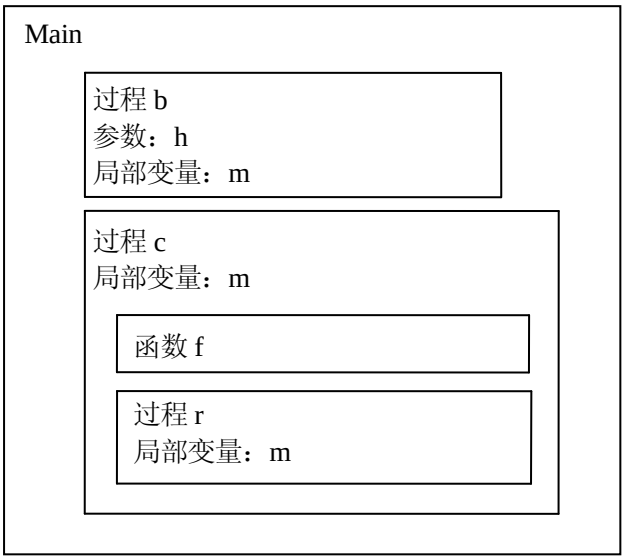
7.3　传值调用：1, 2, 1, 0
　　1, 2, 1, 0
　　引用调用：1, 2, 1, 0
　　1, 2, 0, 0
　　复制恢复：1, 2, 1, 0
　　1, 2, 1, 0
　　传名调用：0, 2, 1, 2
　　0, 0, 1, 2

7.4　(1) 2　(2) 8　(3) 7　(4) 9

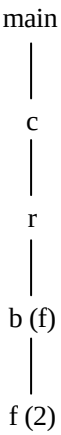
7.6　Pascal 语言采用静态作用域规则。

(1)　该程序的输出结果是：2

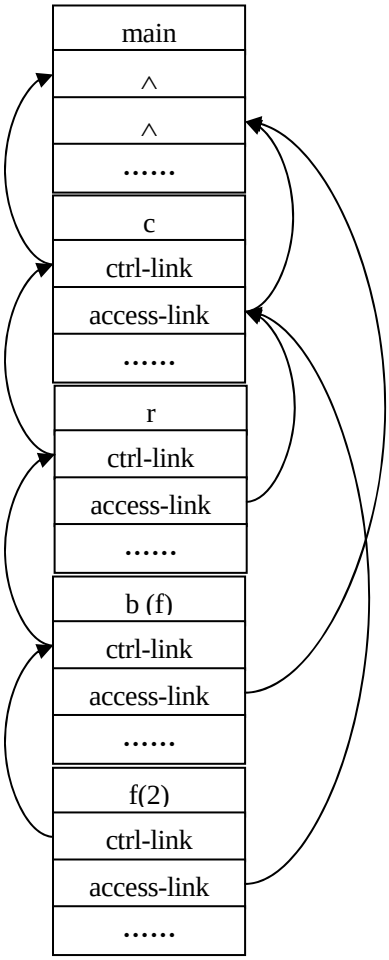
该程序中，各过程之间的嵌套关系如右：



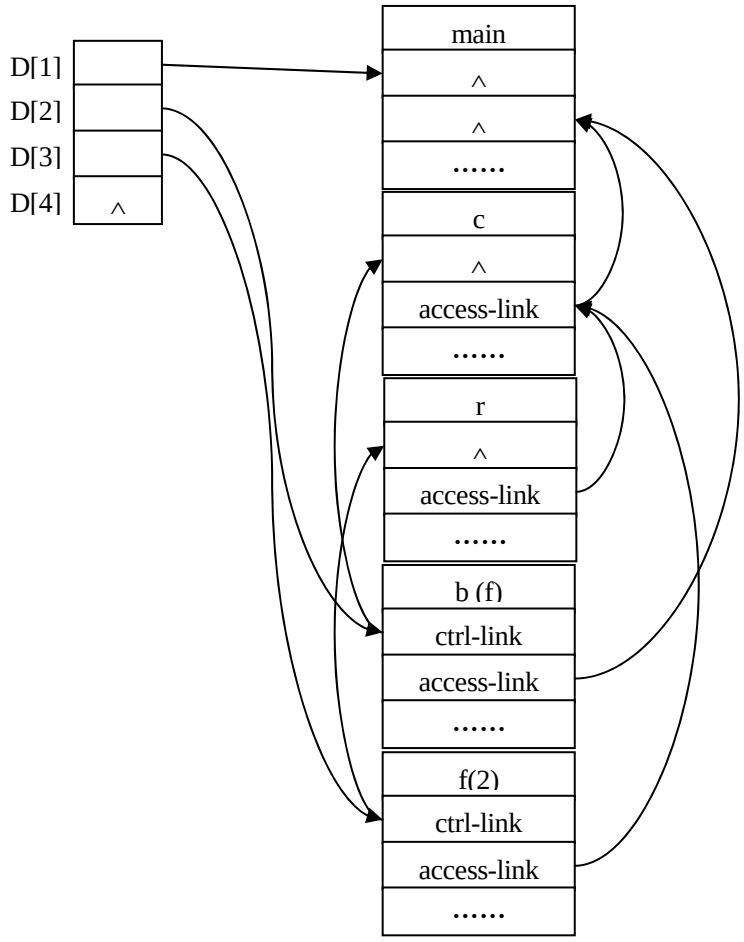
(2) 活动树如下：



(3) 控制栈状态如下：

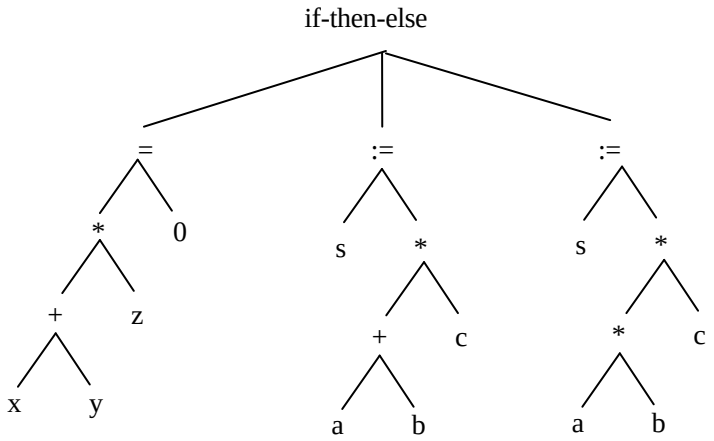


(4) 当控制处于函数 f 中时的控制栈状态和 d 表的内容如下：



第 8 章　中间代码生成

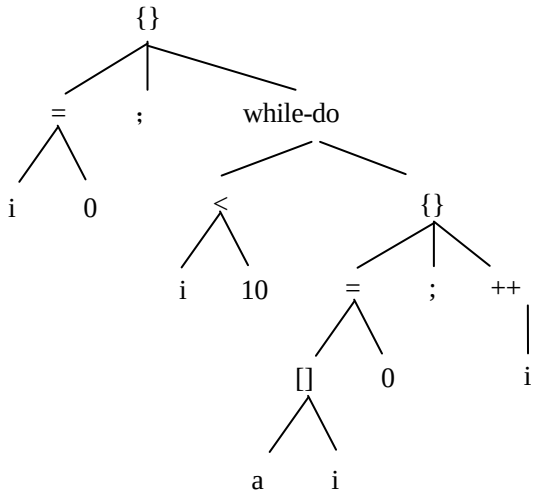
8.3　(1) 该语句的语法树如下：



(2) 该语句的三地址代码如下：

0: t_1:=x+y
1: t_2:=t1*x
2: if t_2=0 goto 4
3: goto 8
4: t_3:=a+b
5: t_4:=t_3*c
6: s:=t_4
7: goto　–　　// 以后回填为 goto 11
8: t_5:=a*b
9: t_6:=t_5*c
10: s:=t_6
11:

8.4　(1) 该程序的程序体是一个语句序列，包括一个赋值语句和一个循环语句，其语法树如下：



(2) 程序体对应的三地址代码如下：

0: i:=0
1: if i<10 goto 3
2: goto 9
3: t_1:=a
4: t_2:=2*i
5: $t_1[t_2]$:=0
6: t_3:=i+1
7: i:=t_3
8: goto 1
9: exit

8.6 文法 8.3 的产生式如下：

(1) S→L:=E　　(5) L→id
(2) E→E+E　　(6) L→Elist]
(3) E→(E)　　(7) Elist→id[E
(4) E→L　　(8) Elist→Elist,E

在设计翻译方案时，需要定义以下属性：

为非终结符号 Elist 设计以下属性及函数。

综合属性 Elist.array 用来保存指向符号表中相应数组名表项的指针。

综合属性 Elist.ndim 记录目前已经识别出的下标表达式的个数，即维数。

综合属性 Elist.entry 表示临时变量，存放由 Elist 中的下标表达式计算出来的值，即递归公式中 e_m 的值。

函数 limit(array,j)返回由 array 所指示的数组的第 j 维的长度 n_j。

函数 invariant(array)返回数组元素地址计算公式中的不变部分，即公式(8.7)的第二行中除 base 以外部分的值。

为非终结符号 L 设计两个综合属性 L.entry 和 L.offset。

如果 L 是一个简单变量，则 L.entry 是一个指针，指向该名字在符号表中的入口，而 L.offset 为 null，表示该左值是一个简单变量而不是数组元素引用。

如果 L 是数组元素引用，则 L.entry 是临时变量，保存公式(8.7)中第二项的值，即 base-C 的值，L.offset 也是临时变量，保存公式(8.7)中第一项的值，即 $e_m \times w$ 的值。

为非终结符号 E 设计综合属性 E.entry，其含义同翻译方案 8.1 中的 E.entry。

过程 emit 用于产生三地址语句。

所设计翻译方案如下：

(1) S→L:=E { if (L.offset==null) /* L 是简单变量 */
emit(L.entry ':=' E.entry);
else emit(L.entry '[' L.offset ']' ':=' E.entry) }

(2) E→E_1+E_2 { E.entry=newtemp;
emit(E.entry ':=' E_1.entry '+' E_2.entry); }

(3) E→(E_1) { E.entry=E_1.entry }

(4) E→L { if (L.offset==null) /* L 是简单变量 */
E.entry=L.entry;
else { E.entry=newtemp;
emit(E.entry ':=' L.entry '[' L.offset ']'); } }

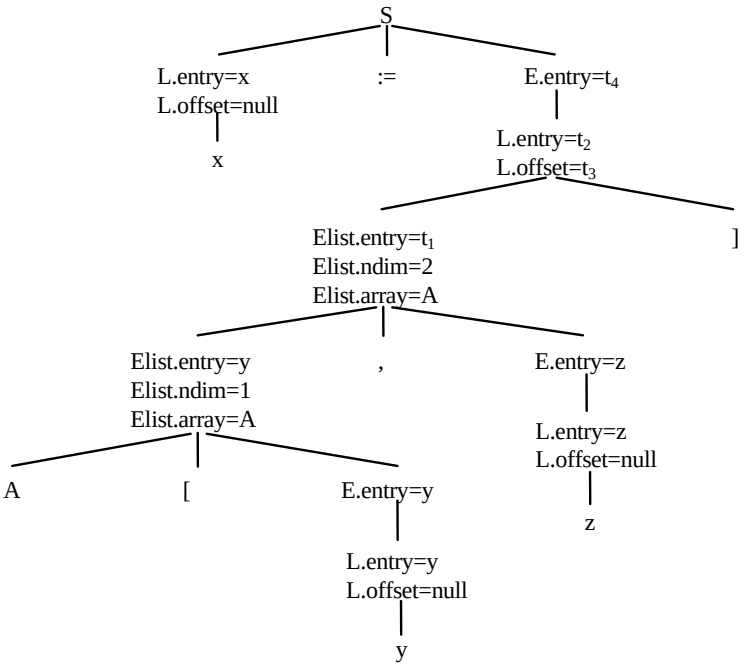
(5) L→id { L.entry=id.entry; L.offset=null }

(6) L→Elist] { L.entry=newtemp;
emit(L.entry ':=' Elist.array '-'invariant(Elist.array));
L.offset=newtemp;
emit(L.offset ':=' w '×' Elist.entry) }

(7) Elist→id[E { Elist.entry=E.entry;
Elist.ndim=1;
Elist.array=id.entry }

(8) Elist→$Elist_1$,E { t=newtemp;
m=Elist1.ndim+1;
emit(t ':=' $Elist_1$.entry '×' limit(Elist.array,m));
emit(t ':=' t '+' E.entry);
Elist.array=$Elist_1$.array;
Elist.entry=t;
Elist.ndim=m }

用该翻译方案翻译赋值语句 x:=A[y,z]。赋值语句 x:=A[y,z]的注释分析树如下：



翻译得到的三地址代码如下：

t_1:=y×20

t_1:=t_1+z

t_2:=A-84

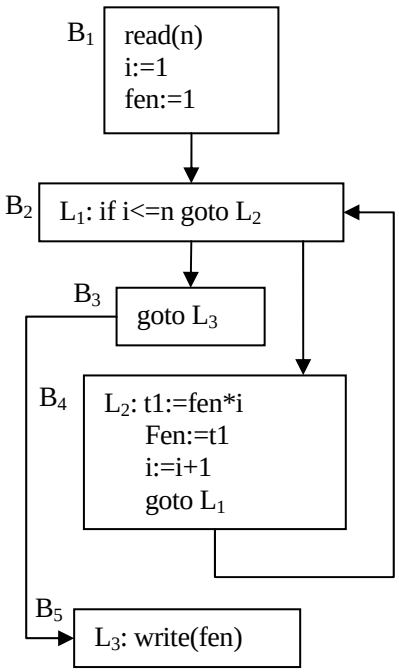
t_3:=4×t_1

t_4:=t_2[t_3]

x:=t_4

第 9 章　代码生成

9.1 这段代码共划分为 5 个基本块（B_1~B_5），基本块及相应的流图如下：



9.2　假设个名字在内存中的地址分别用各自的名字表示。

(4) 赋值语句 x:=(a-b)+(a-c)+(a-c)的三地址代码如下：

t_1:=a-b

t_2:=a-c

t_3:=t_1+t_2

t_4:=a-c

t_5:=t_3+t_4

x:=t_5

相应的目标代码如下：

MOV a, R_0

SUB b, R_0　　// a-b in R_0

MOV a, R_1

SUB c, R_1　　// a-c in R_1

ADD R_1, R_0　　// (a-b)+(a-c) in R_0

MOV a, R_2

SUB c, R_2　　// a-c in R_2

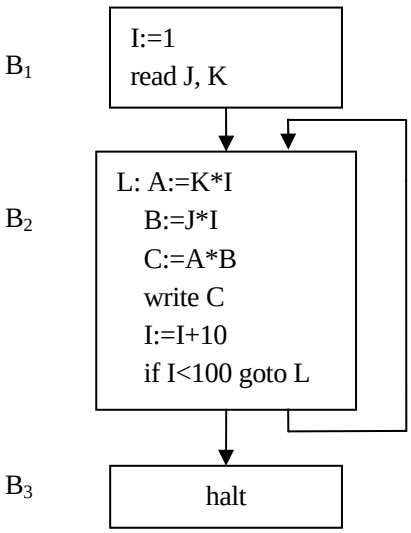
ADD R_2, R_0　　// (a-b)+(a-c)+(a-c) in R_0

MOV R_0, X

第 10 章　代码优化

10.2

(1) 该三地址代码的流图如下：

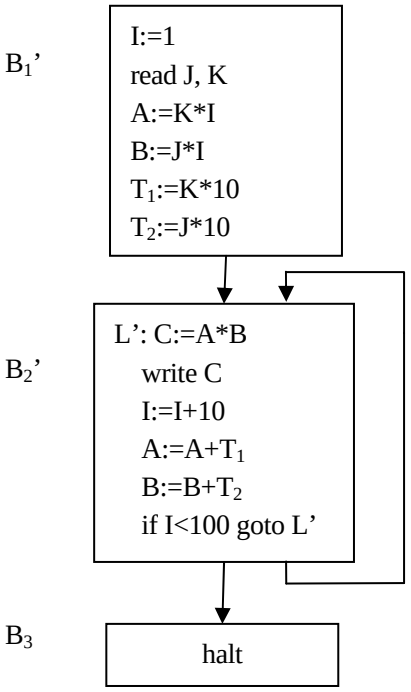


(2) 需要对 B_2 进行循环优化，即代码外提、强度削弱、删除归纳变量。

代码外提：目前没有

强度削弱：

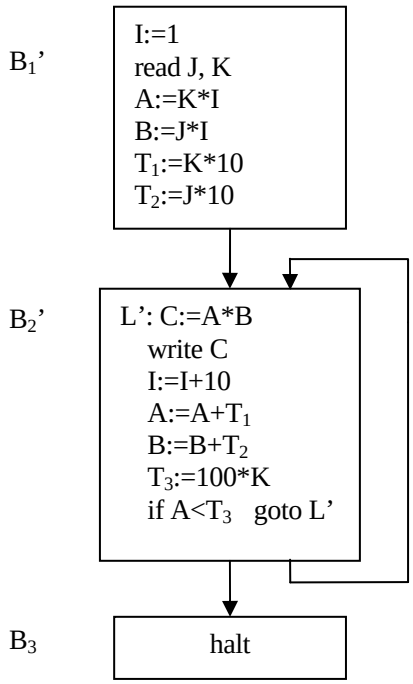
循环变量的变化步长为 10，所以：A:=K*I 可用 A:=A+10*K 代替，K 是循环不变量，故 T_1:=10*K 是常量，A 的初值为 A:=K*I，把这两条语句提到循环之外，放在 B_1 中。对于 B:=J*I，也作同样的处理，经过削弱计算强度优化后的结果是：



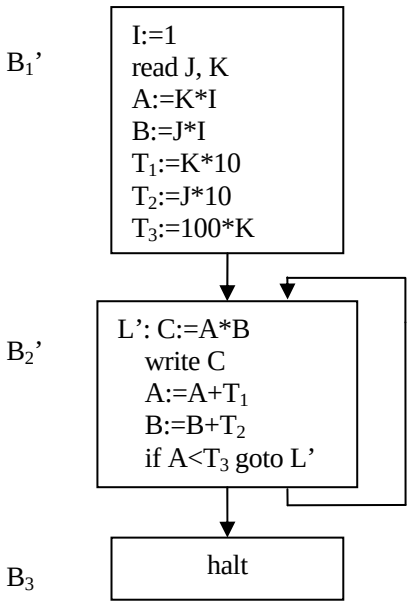
删除归纳变量：

A、B 和 I 是一组归纳变量，且有线性关系：A:=K*I　B:=J*I

所以，条件 I<100 可以用 A<100*K 或 B<100*J 代替。
这样，if 语句就可以变换为：T_3:=100*K
if A<T_3 goto L’
如下图所示：



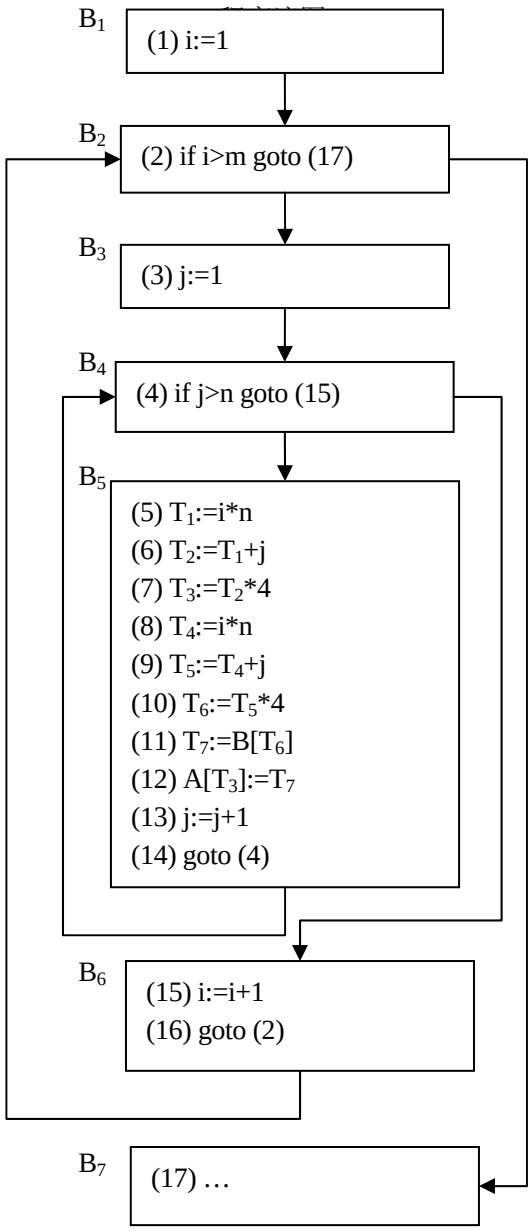
这样，I 就可以删除了，T_3 也可以提到 B_1 中，优化后的结果是：



10.3
(1) 该程序段中的可执行语句的三地址代码如下
(1) i:=1
(2) if i>m goto (17)
(3) j:=1
(4) if j>n goto (15)
(5) T_1:=i*n

(6) T_2:=T_1+j
(7) T_3:=T_2*4
(8) T_4:=i*n
(9) T_5:=T_4+j
(10) T_6:=T_5*4
(11) T_7:=B[T_6]
(12) A[T_3]:=T_7
(13) j:=j+1
(14) goto (4)
(15) i:=i+1
(16) goto (2)
(17) …

(2) 对于(1)中的三地址代码，基本块划分结果及流图如下：



(3) 程序中的循环有：

L_1={ B_4，B_5 }

L_2={ B_2，B_3，B_4，B_5，B_6 }

对于内循环 L_1 进行优化。

首先对 B_5 进行基本块优化。

由于 T_1:=i*n 和 T_4:=i*n 是公共表达式，可以删除后者，通过复写传播，用 T_1 代替 T_4，得到基本块 B_5'。

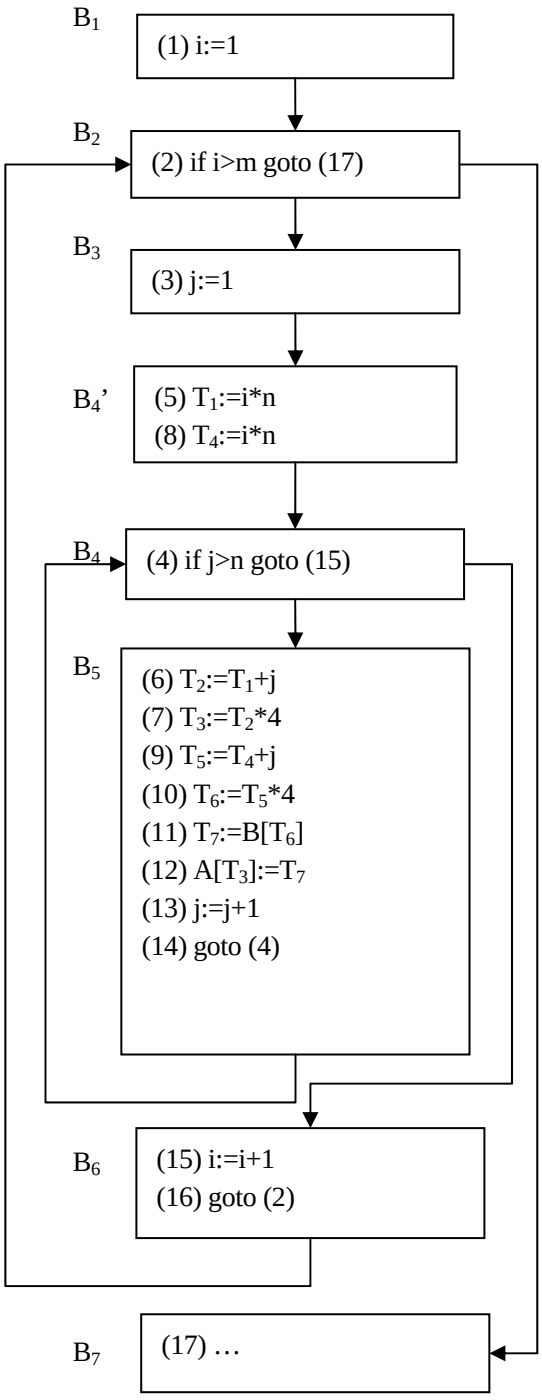
从 B_5'中可以看出，T_2:=T_1+j 和 T_5:=T_1+j 成为公共表达式，可以删除后者，通过复写传播，用 T_2 代替 T_5，得到基本块 B_5''。

从 B_5''中可以看出，T_3:=T_2*4 和 T_6:=T_2*4 成为公共表达式，可以删除后者，通过复写传播，用 T_3 代替 T_6，得到基本块 B_5'''。

B_5	B_5'	B_5''	B_5'''
(5) T_1:=i*n	(5) T_1:=i*n	(5) T_1:=i*n	(5) T_1:=i*n
(6) T_2:=T_1+j	(6) T_2:=T_1+j	(6) T_2:=T_1+j	(6) T_2:=T_1+j
(7) T_3:=T_2*4	(7) T_3:=T_2*4	(7) T_3:=T_2*4	(7) T_3:=T_2*4
(8) T_4:=i*n	(9) T_5:=T_1+j	(10) T_6:=T_2*4	(11) T_7:=B[T_3]
(9) T_5:=T_4+j	(10) T_6:=T_5*4	(11) T_7:=B[T_6]	(12) A[T_3]:=T_7
(10) T_6:=T_5*4	(11) T_7:=B[T_6]	(12) A[T_3]:=T_7	(13) j:=j+1
(11) T_7:=B[T_6]	(12) A[T_3]:=T_7	(13) j:=j+1	(14) goto (4)
(12) A[T_3]:=T_7	(13) j:=j+1	(14) goto (4)	
(13) j:=j+1	(14) goto (4)		
(14) goto (4)			

代码外提：

在 B_5 中，T_1:=i*n 和 T_4:=i*n 中涉及的 i、n、m 对循环 L_1 来说是不变的，可以把这两个运算提到循环 L_1 前面。得到下面的流图：

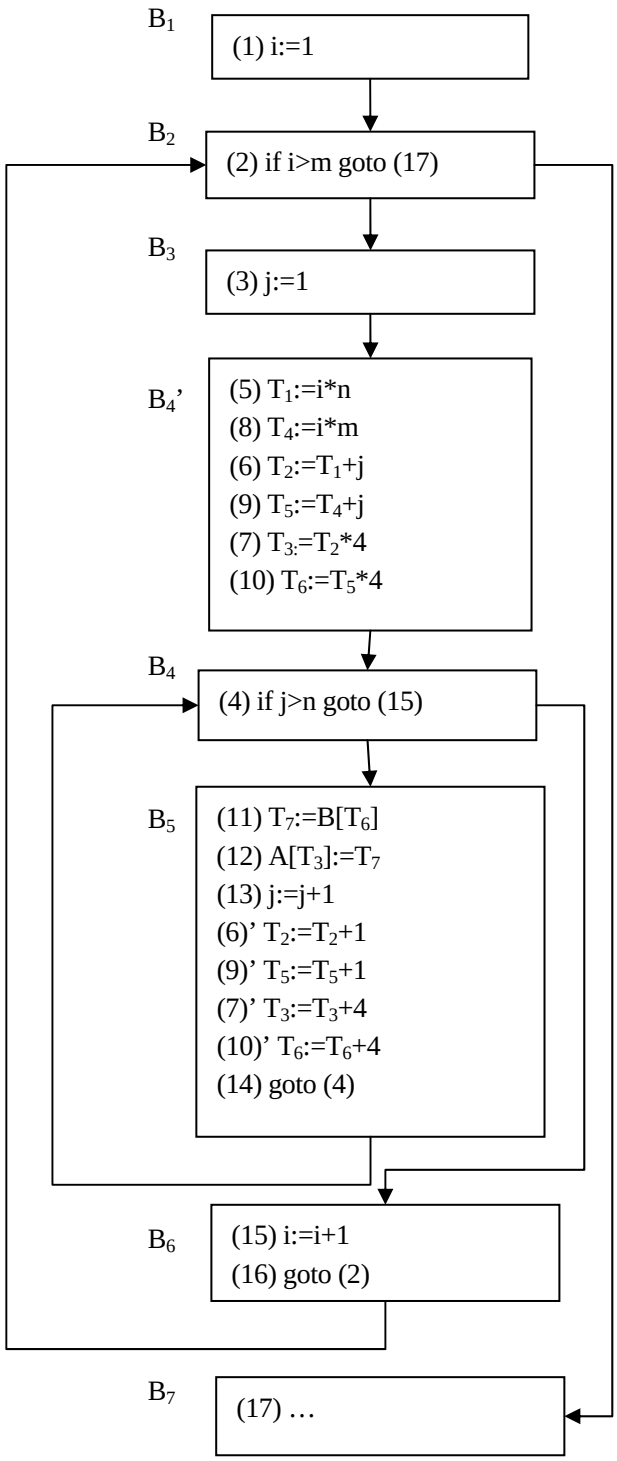


L_1 中不变代码外提后的流图

削弱计算强度：

在 B_5 中，由于 T_1 和 T_4 外提，$T_2:=T_1+j$ 和 $T_5:=T_4+j$ 中，T_2 和 T_5 随循环的进行递增，所以，可将其进行强度削弱，将这两条语句改为：$T_2:=T_2+1$ 和 $T_5:=T_5+1$，而将计算 T_2 和 T_5 初值的语句 $T_2:=T_1+j$ 和 $T_5:=T_4+j$ 外提到 B_4'中。

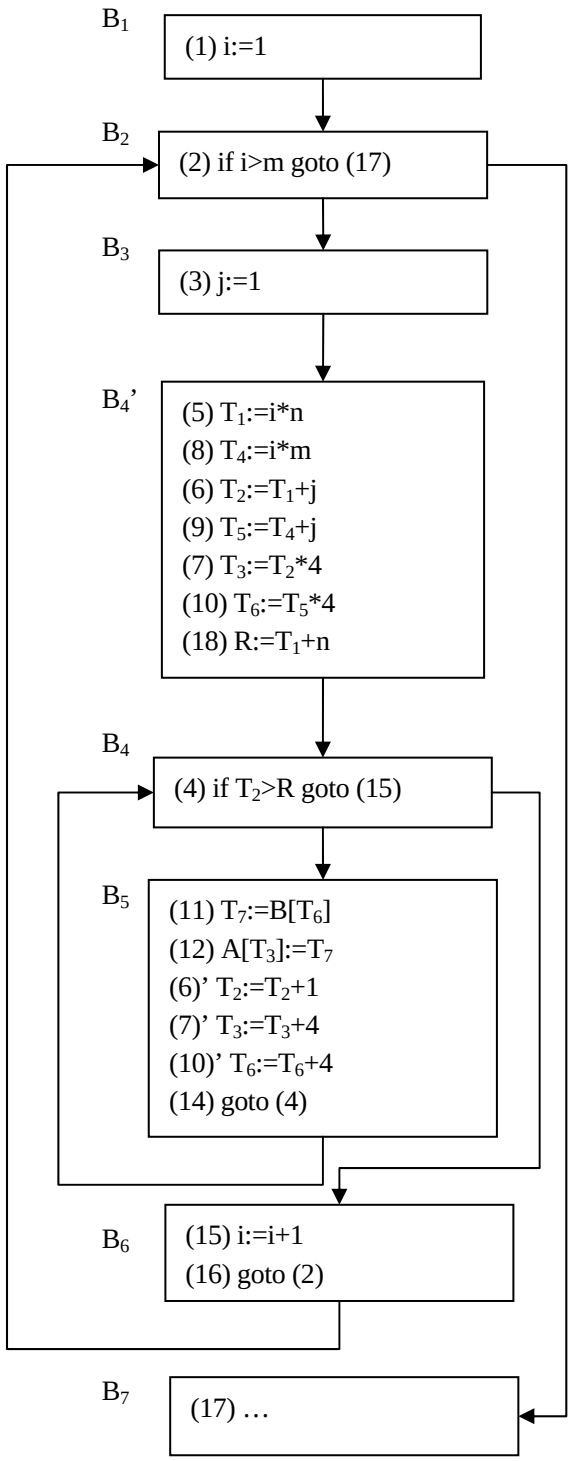
又由于 $T_3:=T_2*4$，$T_6:=T_5*4$，T_3 和 T_6 随着循环递增，所以，可将其计算强度削弱，将这两条语句变换为：$T_3:=T_3+4$ 和 $T_6:=T_6+4$，而把为 T_3 和 T_6 赋初值的语句 $T_3:=T_2*4$，$T_6:=T_5*4$ 外提到 B_4'中，得到如下流图：



L_1 削弱计算强度后的流图

删除归纳变量：

在 L_1 中，T_2、T_5 和 j 是一组归纳变量，可以将 j 删除，并用 T_2>R 替换循环控制条件，把 R 的初始化语句 R:=T_1+n 外提到 B_4’中，这样 B_5 中的语句 T_5:=T_5+1 将成为无用代码，可以删除。得到如下流图：



L_1 删除归纳变量和无用代码后的流图

下面对循环 $L_2=\{B_2, B_3, B_4, B_5, B_6\}$进行优化。

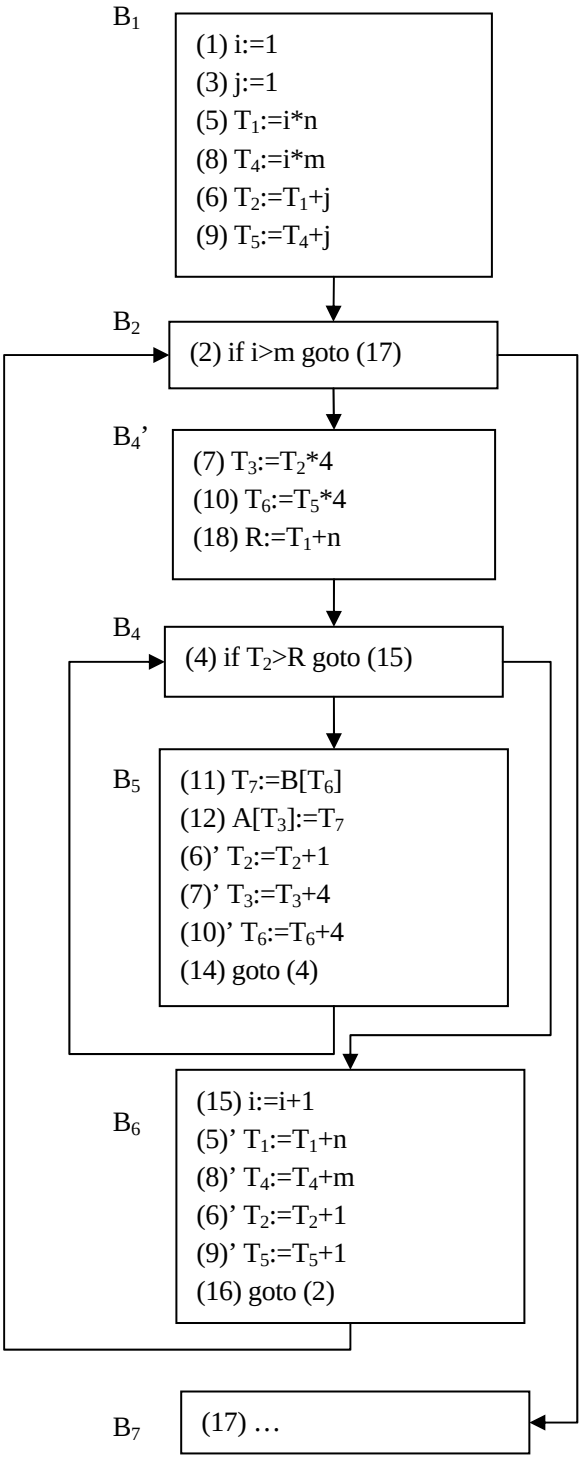
代码外提：

B_3 中的 j:=j+1 是循环不变量，可以提到 L_2 之外，放到 B_1 中。

削弱计算强度：

在 B_4’中，由于 T_1:=i*n 和 T_4:=i*m，T_1 和 T_4 随着循环递增，所以，可以削弱它们的计算强度，代之以语句 T_1:=T_1+n 和 T_4:=T_4+m，而把 T_1 和 T_4 初始化的语句 T_1:=i*n 和 T_4:=i*m 外提到 B_1 中。

B_4’中，T_2 和 T_5 随着循环 L_2 递增，可以将其进行强度削弱，代之以：T_2:=T_2+1 和 T_5:=T_5+1, 而把初始化语句外提到 B_1 中，得到如下流图：



L_2 代码外提和削弱计算强度后的流图

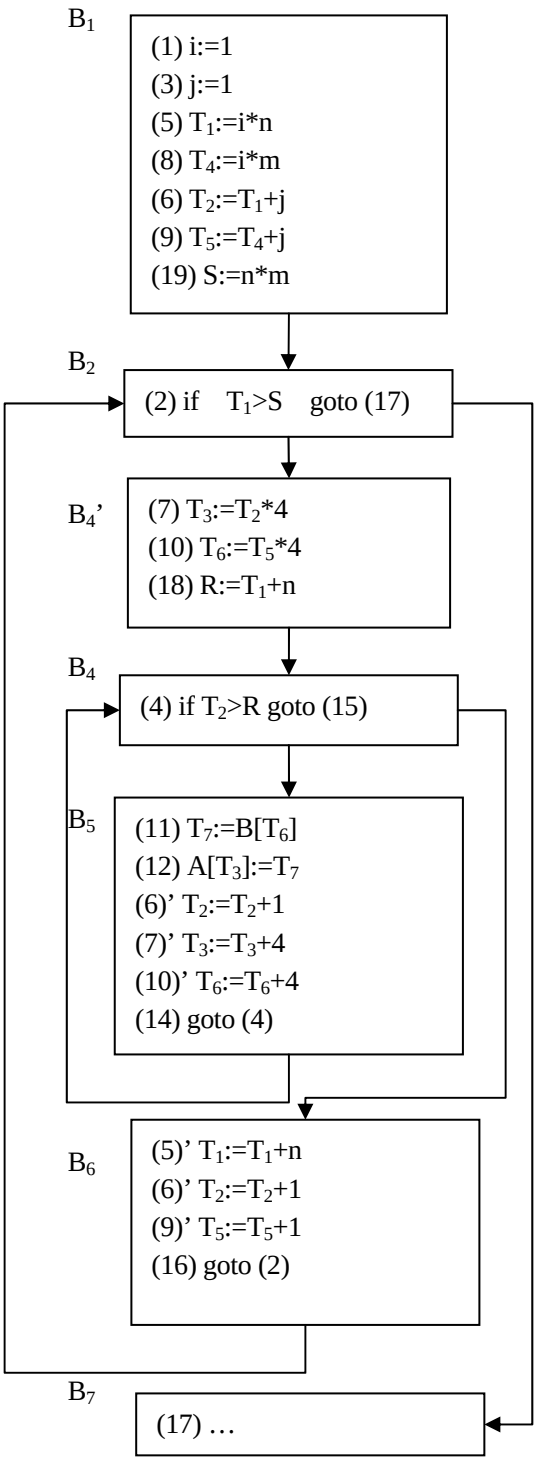
删除死代码：

B_6 中的 T_4:=T_4+m 已经成为无用代码，可以删除。

删除归纳变量：

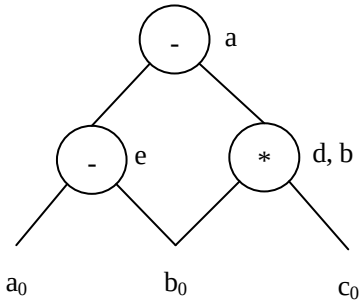
B_6 中的 T_1 与 i 为一组归纳变量，可以将循环控制条件 i>m 变换为 T_1>S，其中 S:=n*m，可将 S 的计算提到 B_1 中，从而可以删除 i

得到如下流图：



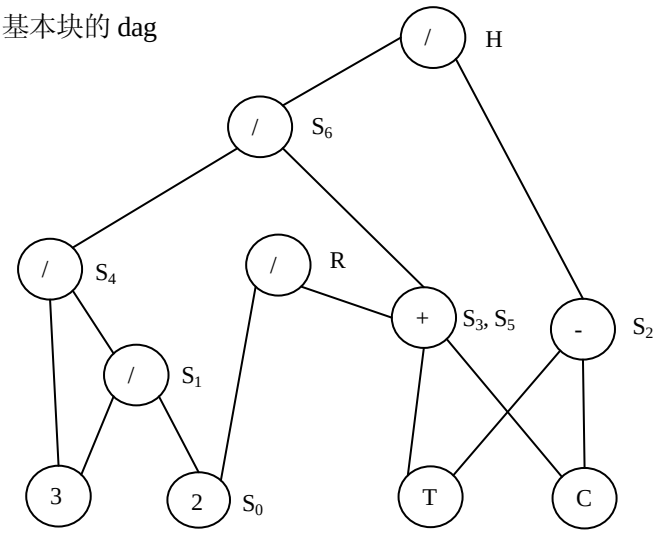
L_2 删除死代码和归纳变量后的流图

10.4



10.5

(1) 基本块的 dag



(2) 根据启发式排序，得到如下计算顺序：

S_0:=2
S_2:=T-C
S_3:=T+C
S_5:=S_3
S_1:=3/S_0
S_4:=3/S_1
S_6:=S_4/S_3
R:=S_0/S_3
H:=S_6*S_2

进行常数合并与传播，得到：

S_0:=2
S_2:=T-C
S_3:=T+C
S_5:=S_3
S_1:=1.5
S_4:=2
S_6:=2/S_3
R:=2/S_3
H:=S_6*S_2

若只有 R、H 在出口处是活跃的，则优化后的基本块如下：

S_2:=T-C
S_3:=T+C
R:=2/S_3
H:=S_6*S_2