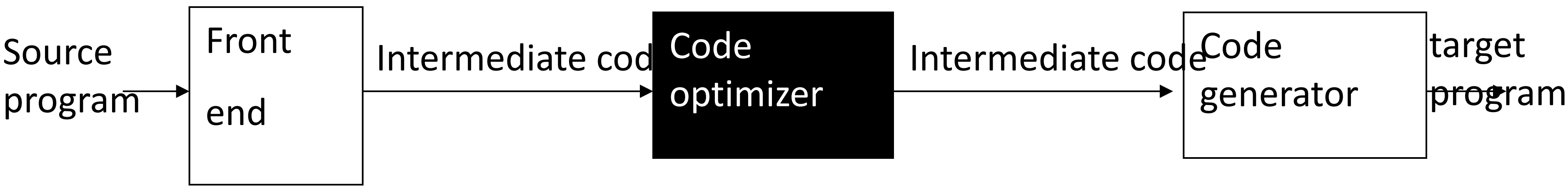
Compiler
--- Machine Independent Optimization

Zhang Zhizheng
seu_zzz@seu.edu.cn
School of Computer Science and Engineering,
Software College
Southeast University

8.4, 8.5, 9.1 in Textbook

Role

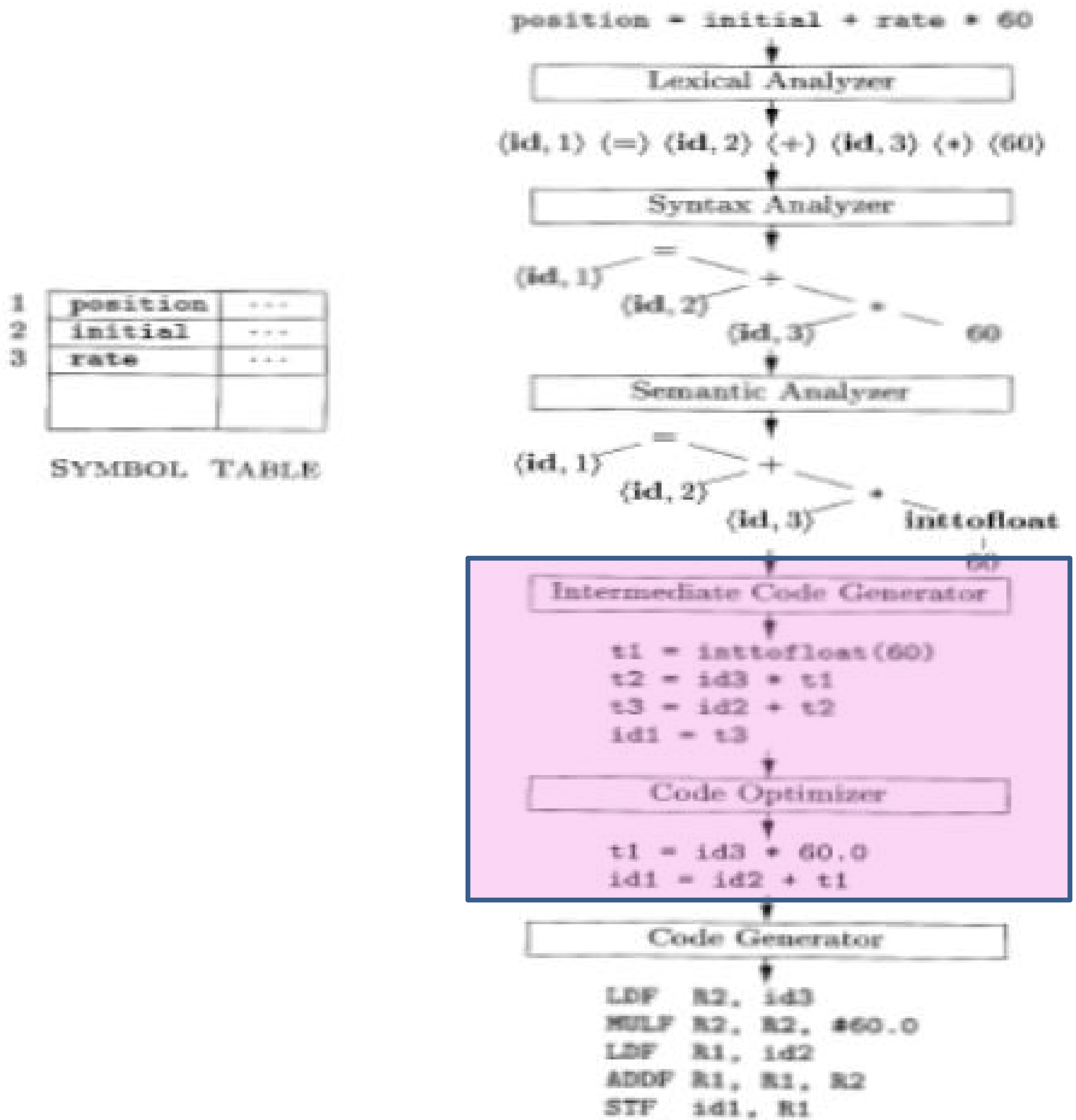
1.Position of code optimizer



2.Purpose of code optimizer

- to get better efficiency
  - Run faster
  - Take less space

Example

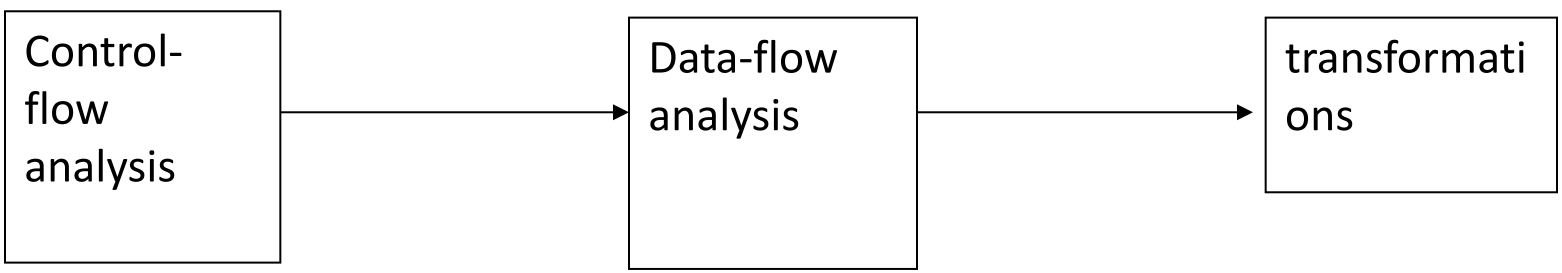


position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
60
Semantic Analyzer
=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
inttofloat
60
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
1 position ...
2 initial ...
3 rate ...
SYMBOL TABLE

Three Types of Optimizations

- Source code
  - User can profile the program, change algorithm or transform loops.
- Intermediate code (**Machine-independent**)
  - Compiler can improve loops, procedure calls or address calculations
- Target code
  - Compiler can use registers, select instructions or do peephole transformations

Machine-independent Optimization Procedure



Control-flow analysis

- Identify blocks and loops in the flow graph of a program

Data-flow analysis

- Collect information about the program as a whole and to distribute this information to each block in the flow graph.

Basic Blocks and flow graphs

1.Flow graph

- A directed graph that are composed of the set of basic blocks making up a program

2.Basic blocks

- A sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

- E.g. this is a basic block

T1=a*a

T2=a*b

T3=2*T2

T4=T1+T3

T5=b*b

T6=T4+T5

Notes: A name in a basic block is said to be live at a given point if its value is used after that point in the program, perhaps in another basic block.

3.Partition into basic blocks

- **Input.** A sequence of three-address statements
- **Output.** A list of basic blocks with each three-address statement in exactly one block.
- **Method.**

1) We first determine the set of leaders, the first statements of basic blocks.The rules we use are the following:

- (1) The first statement is a leader.
- (2) Any statement that is the target of a conditional or unconditional goto is a leader.
- (3) Any statement that immediately follows a goto or conditional goto statement is a leader.

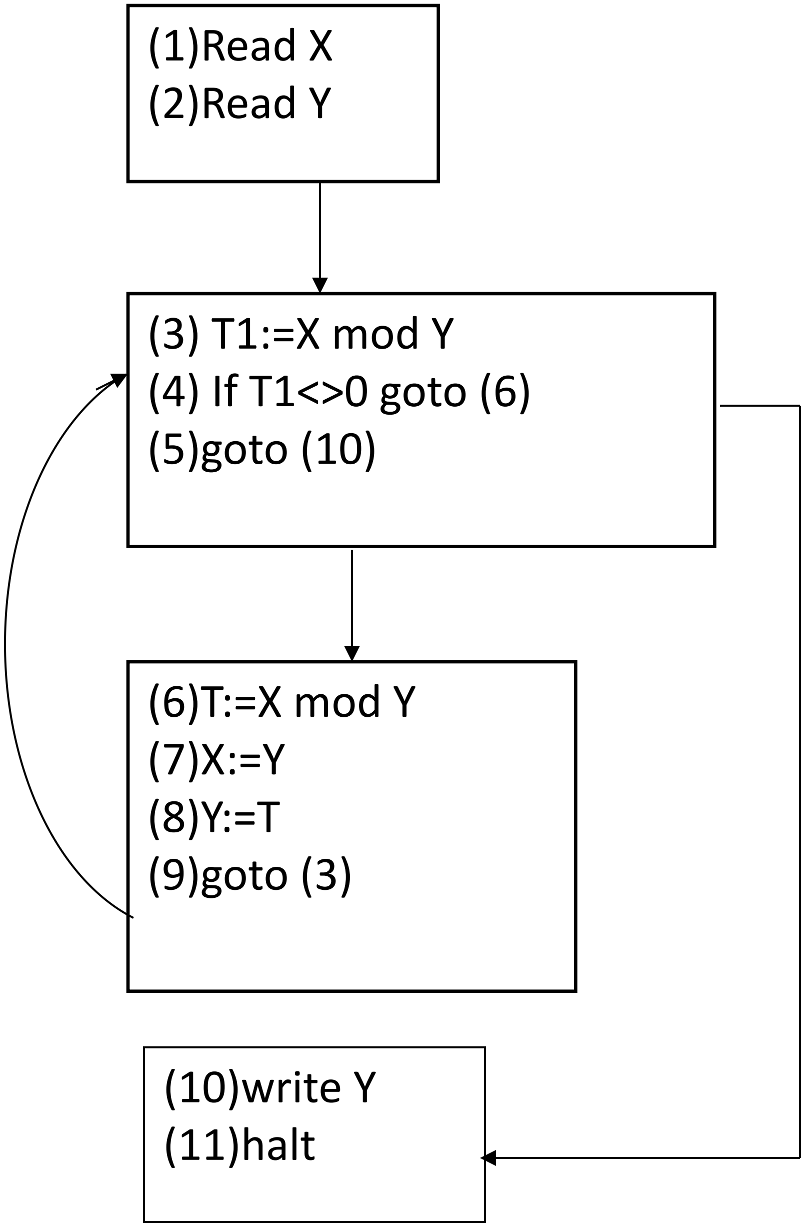
2) For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

EXAMPLE 1.

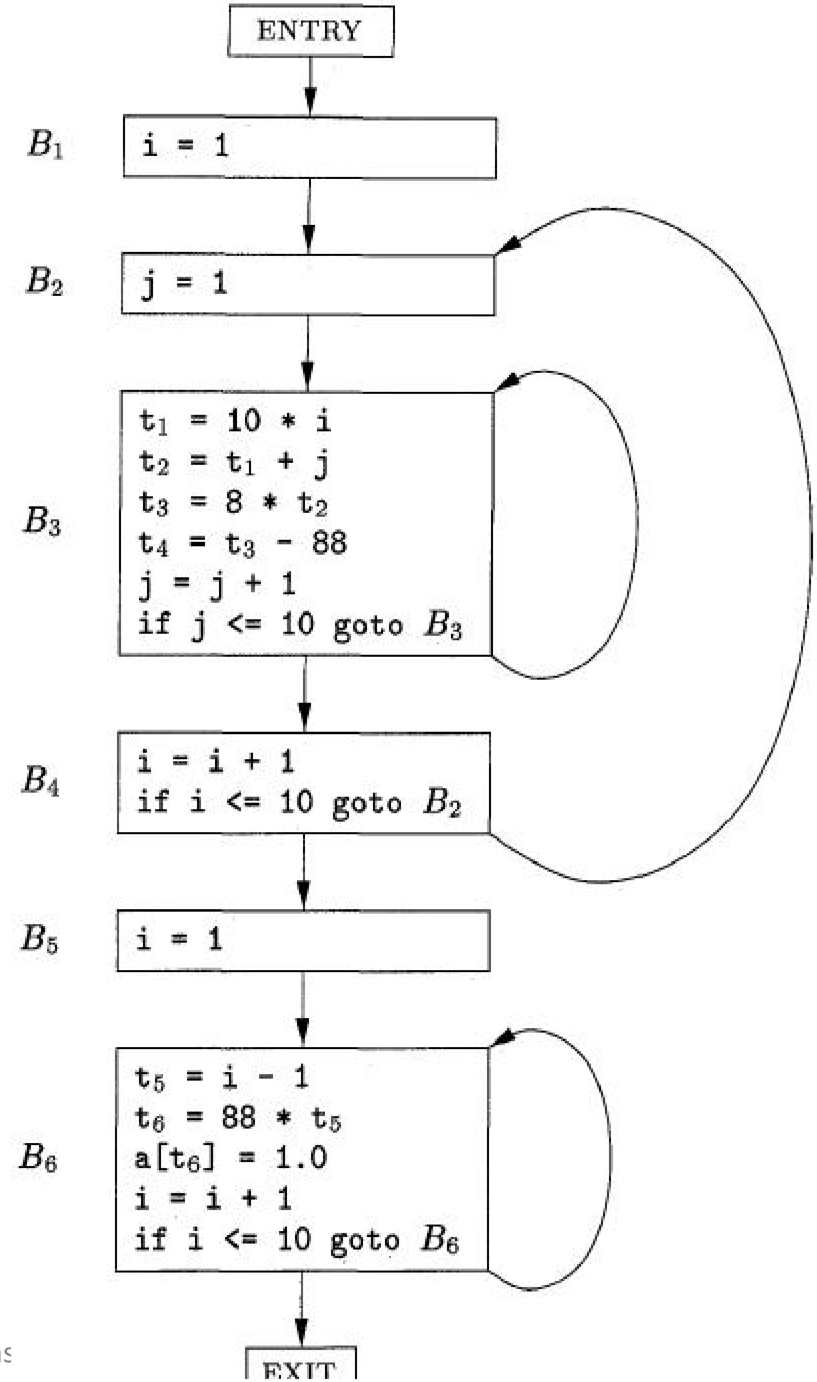
```
begin
    read X;
    read Y;
    while (X mod Y<>0) do
      begin
        T:=X mod Y;
        X:=Y;
        Y:=T
      end;
  write Y
end
```

```
(1)Read X
(2)Read Y
(3) T1:=X mod Y
(4) If T1<>0 goto (6)
(5)goto (10)
(6)T:=X mod Y
(7)X:=Y
(8)Y:=T
(9)goto (3)
(10)write Y
(11)halt
```

Blocks and Flow graph



```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```



Next-Use Information

Knowing when the value of a variable will be used next is essential for generating good code. If the value of a variable that is currently in a register will never be referenced subsequently, then that register can be assigned to another variable.

The *use* of a name in a three-address statement is defined as follows. Suppose three-address statement i assigns a value to x. If statement j has x as an operand, and control can flow from statement i to j along a path that has no intervening assignments to x, then we say statement j *uses* the value of x computed at statement i. We further say that x is *live* at statement i.

Optimization of basic blocks

1.Function-preserving transformations

1)Methods

- Constant folding
  - The evaluation at compile-time of expressions whose operands are known to be constant
- Common sub-expression elimination
- Copy propagation
- Dead-code elimination

2)Constant folding & Common sub-expression elimination

- An expression E is called a common sub-expression if an expression E was previously computed, and the values of variables in E have not changed since the previous computation.

Notes: We can avoid re-computing the expression if we can use the previously computed value e.g：A source code

```
Pi:=3.14
A:＝2*Pi*(R+r);
B:=A;
B:=2* Pi*(R+r)*(R-r)
```

```
(1)Pi:=3.14
(2)T1:=2*Pi
(3)T2:=R+r
(4)A:=T1*T2
(5)B:=A
(6)T3:=2*Pi
(7)T4:=R+r
(8)T5:=T3*T4
(9)T6:=R-r
(10)B:=T5*T6
```

(1)Pi:=3.14
(2)T1:= 2*Pi
(3)T2:=R+r
(4)A:=T1*T2
(5)B:=A
(6)T3:= 2*Pi
(7)T4:=R+r
(8)T5:=T3*T4
(9)T6:=R-r
(10)B:=T5*T6

(1)Pi:=3.14
(2)T1:=6.28
(3)T2:=R+r
(4)A:=T1*T2
(5)B:=A
(6)T3:=T1
(7)T4:=T2
(8)T5:=T3*T4
(9)T6:=R-r
(10)B:=T5*T6

3)Copy Propagation

- One concerns assignments of the form f:=g called copy statements

(1)Pi:=3.14
(2)T1:=6.28
(3)T2:=R+r
(4)A:=T1*T2
(5)B:=A
(6)T3:=T1
(7)T4:=T2
(8)T5:=T3*T4
(9)T6:=R-r
(10)B:=T5*T6

(1)Pi:=3.14
(2)T1:=6.28
(3)T2:=R+r
(4)A:=T1*T2
(5)B:=A
(6)T3:=T1
(7)T4:=T2
(8)T5:=T1*T2
(9)T6:=R-r
(10)B:=T5*T6

4) Dead-code elimination

- A variable is live at a point in a program if its value can be used subsequently; otherwise it is dead at that point.

Notes: A related idea is dead or useless code, statements that compute values that never get used.

```
(1)Pi:=3.14
(2)T1:=6.28
(3)T2:=R+r
(4)A:=T1*T2
(5)B:=A
(6)T3:=T1
(7)T4:=T2
(8)T5:=T1*T2
(9)T6:=R-r
(10)B:=T5*T6
```

```
(1)Pi:=3.14
(2)T1:=6.28
(3)T2:=R+r
(4)A:=T1*T2
(8)T5:=T1*T2
(9)T6:=R-r
(10)B:=T5*T6
```

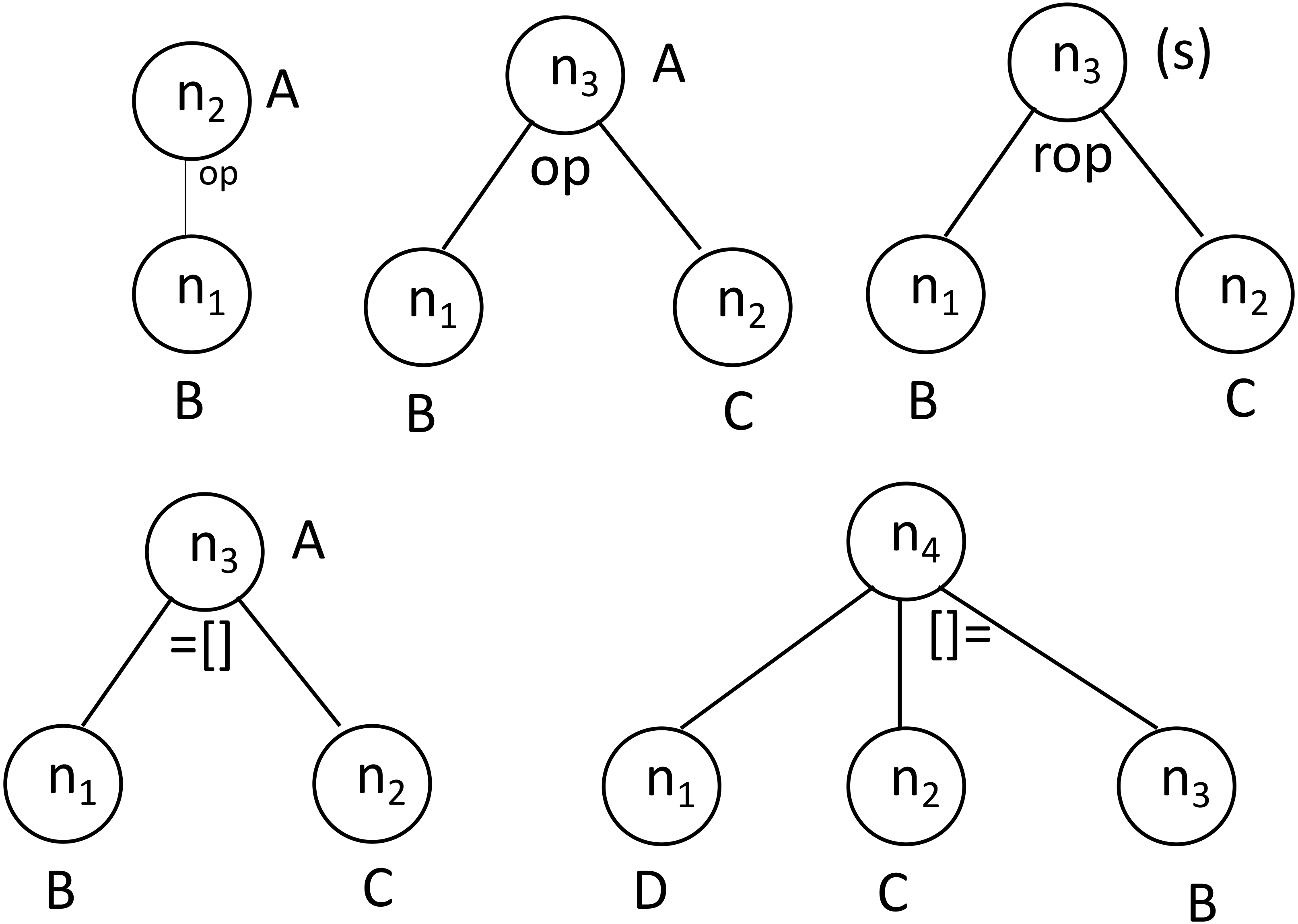
2.DAG Representation(**8.5.1 section**)

A Dag for a basic block is a directed acyclic graph with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants.
2. There is a node associated with each statement *s* within the block.
3. Interior nodes are labeled by an operator symbol.

- The children of *n* are those nodes corresponding to statements that are the last definitions prior to *s* of the operands used by *s*.

- Nodes are also optionally given a sequence of identifiers for labels. The intension is that interior nodes represent computed values, and the identifiers labeling a node are deemed to have the value.



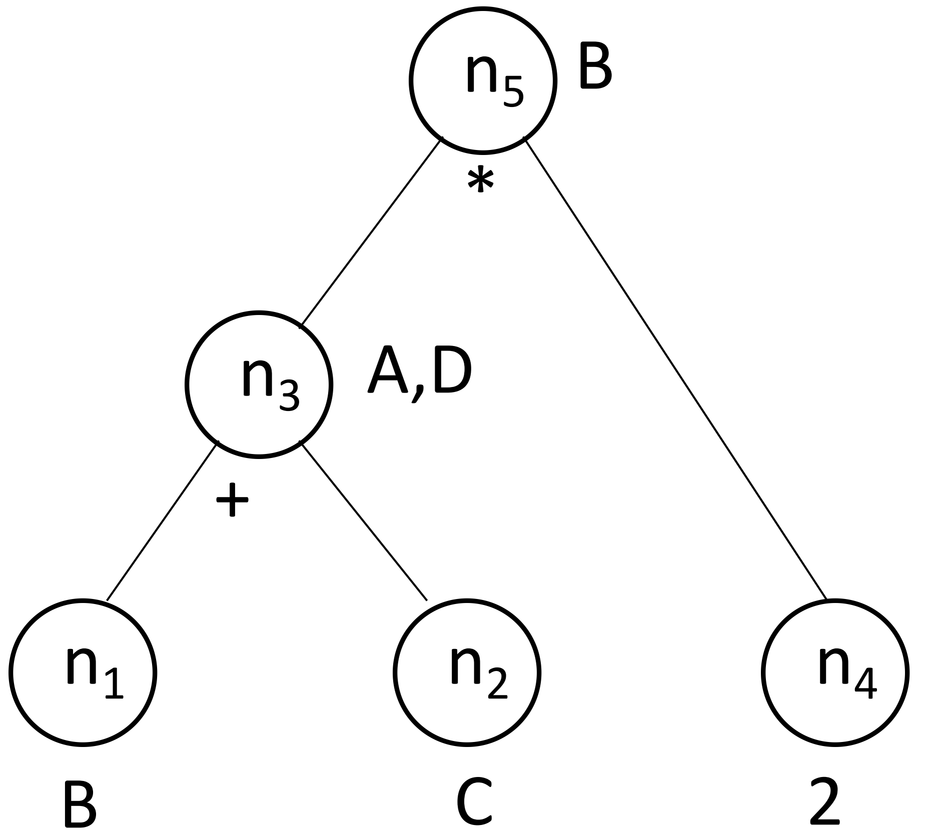
n2 A
op
n1
B
n3 A
op
n1 n2
B C
n3 (s)
rop
n1 n2
B C
n3 A
=[]
n1 n2
B C
n4
[]=
n1 n2 n3
D C B

e.g.

A:=B+C

B:=2*A

D:=A



(1)t1:=4*I

(2)t2:=a[t1]

(3)t3:=4*I

(4)t4:=b[t3]

(5)t5:=t2*t4

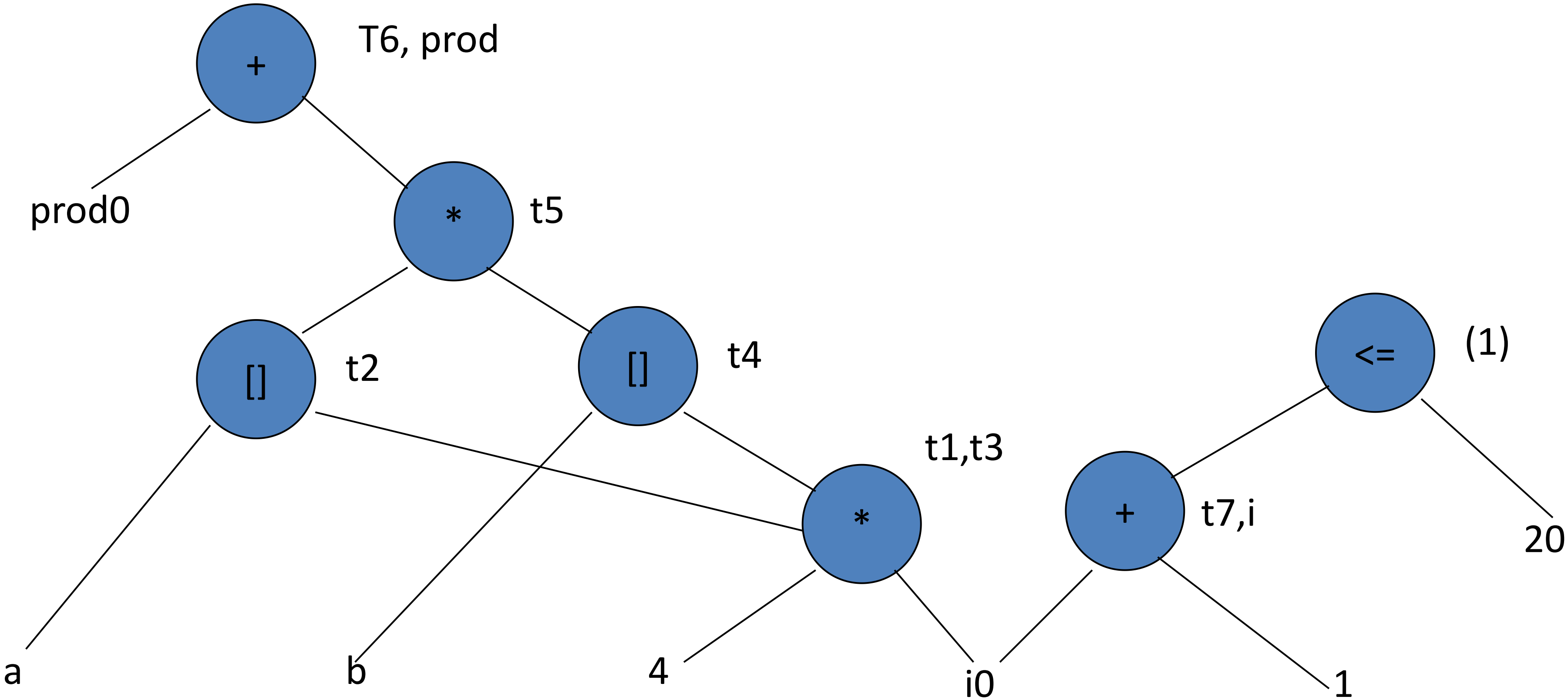
(6)t6:=prod+t5

(7)prod:=t6

(8)t7:=i+1

(9)i:=t7

(10)If i<=20 goto(1)



T6, prod
+
prod0
*
t5
[]
t2
[]
t4
<=
(1)
t1,t3
*
+
t7,i
20
a
b
4
i0
1

DAG based Optimization by Rewriting TAC

- a) a leave without additional label, no TAC
- b) a leave with label B, and additional label A, then we can get assignment statement A=B, if it has several additional labels, then we can get several assignment statements with the form X=B.
- c)for a Interior node with additional label A and label op, we can get A:=B op C, A:=op B, A:=B[C] or if B rop C goto (s)), if it has more than one additional labels, then we can also get assignment statements with the form X=A
- d)for a Interior node without additional label, add a new temp additional label S_i, and goto c)
- e) Dead code Elimination
- f) Using Algebraic Identities

Note.

Algebraic Identities

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

Expensive		Cheaper
x^2	$=$	$x \times x$
$2 \times x$	$=$	$x + x$
$x/2$	$=$	$x \times 0.5$

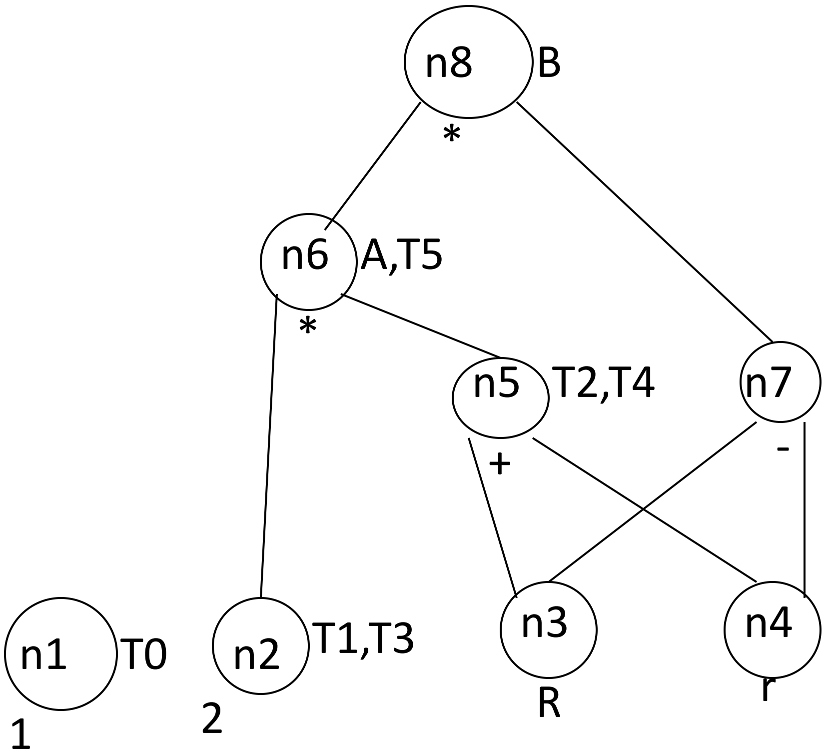
Example

(1)T0:=1 (2)T1:=2*T0 (3)T2:=R+r

(4)A:=T1*T2 (5)B:=A (6)T3:=2*T0

(7)T4:=R+r (8)T5:=T3*T4 (9)T6:=R-r

(10)B:=T5*T6



(1)T0:=1

(2)T1:=2

(3)T3:=2

(4)T2:=R+r

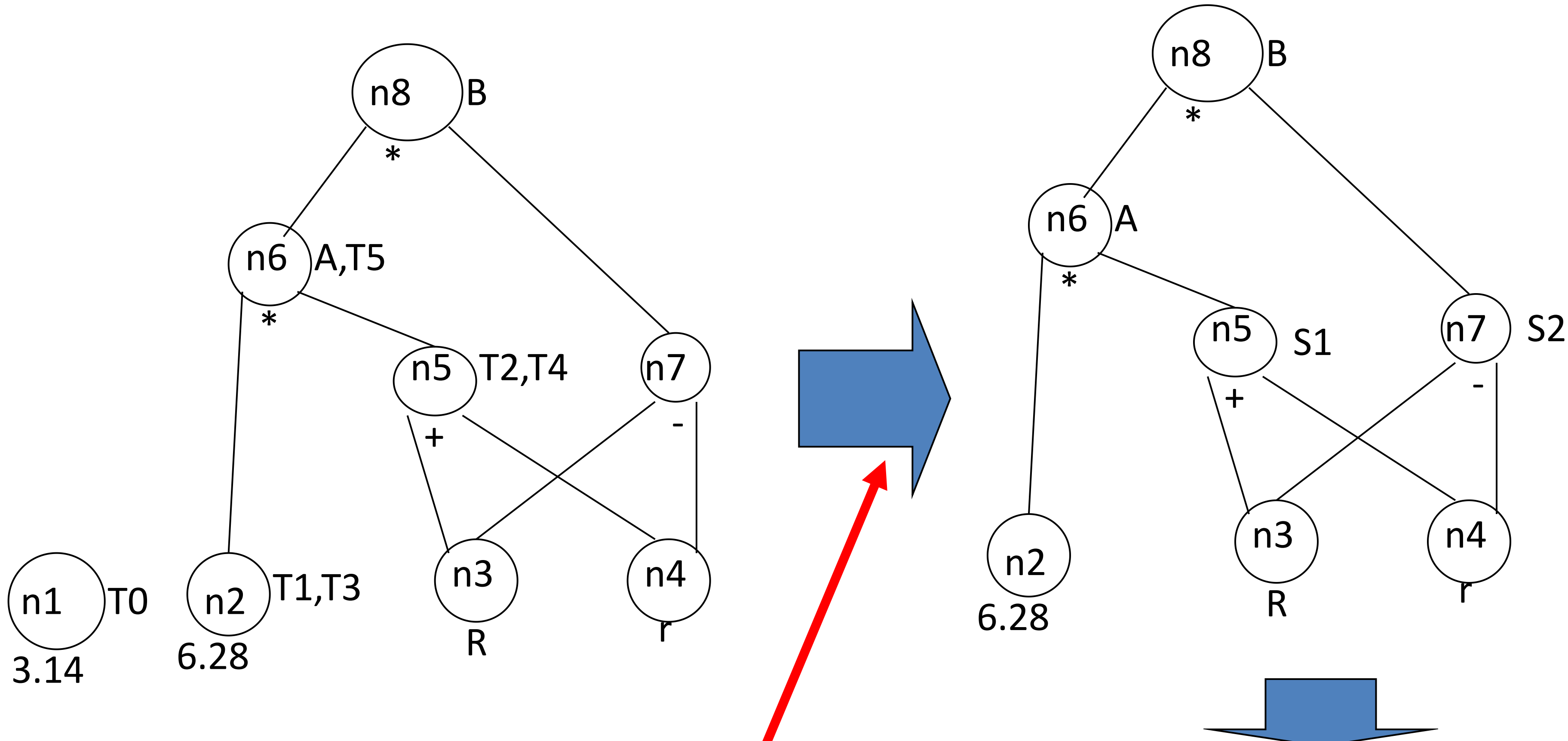
(5)T4:=T2

(6)A:=2*T2

(7)T5:=A

(8)T6:=R-r

(9)B:=A*T6



When T0,T1,T2,T3,T4,T5, T6 are dead

(1)S1:=R+r

(2)A:=2*s1

(3)s2:=R-r

(4)B:=A*s2

(1)S1:=R+r

(2)A:=2*s1

(3)s2:=R-r

(4)B:=A*s2

(1)S1:=R+r

(2)A:=s1+s1

(3)s2:=R-r

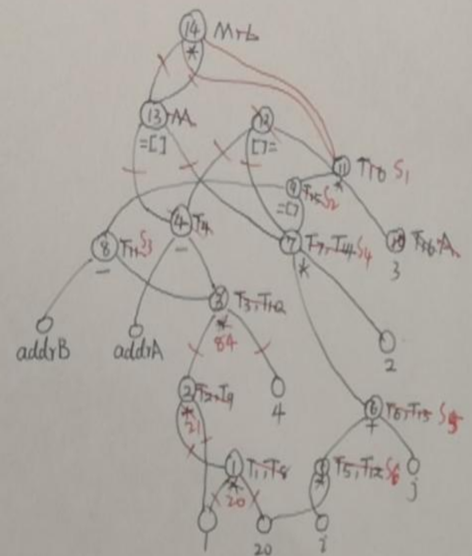
(4)B:=A*s2

Please **construct the DAG** for the following basic block, and we assume that only variable "M" would be used later, please optimize the block and **rewrite the block** in optimized code form.(10%)

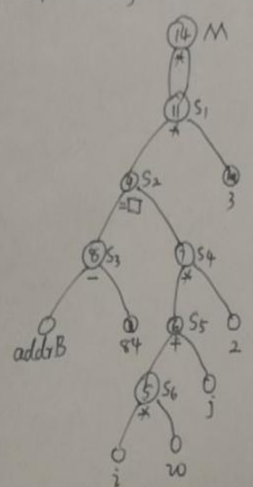
T1=1*20
T2=T1+1
T3=T2*4
T4=addrA-T3
T5=i*20
T6=T5+j
T7=T6*2
T8=1*20
T9=T8+1
T10=T9*4
T11=addrB-T10
T12=i*20
T13=T12+j
T14=T13*2
T15=T11[T14]
A=3
T16=T15*A
T4[T7]=T16
M=T4[T7]
M=M*M
L=M

8.

1) Construct the DAG



2) Optimizing the DAG



3) Rewriting the TAC

Toplogy order: ⑭ ⑪ ⑨ ⑧ ⑦ ⑥ ⑤ ←

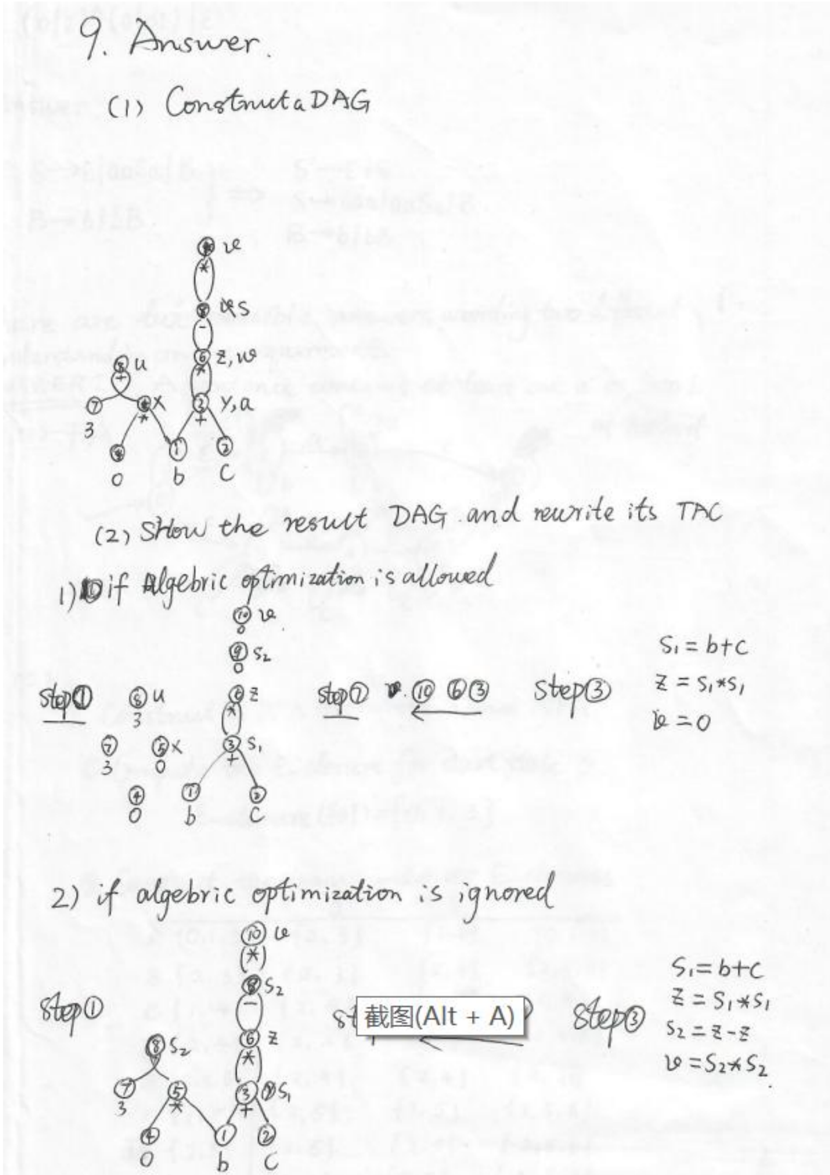
$S_6 = i * 20$
$S_5 = S_6 + j$
$S_4 = S_5 * 2$
$S_3 = addrB - 84$
$S_2 = S_3[S_4]$
$S_1 = S_2 * 3$
$M = S_1 * S_1$

6

Generate a quadruple sequence for the following expression, and optimize and rewrite it using DAG techniques.

y := b + c
x := 0 * b
a := b + c
z := a * a
w := y * y
u := x + 3
v := z - w
v:= v * v

Assume that the only variables that are live at the exit of this block are v and z.



Creating an Topology Order:
(1) Depth first traversal.
(2) Ancestor node first traversal.

Creating an Topology Order:

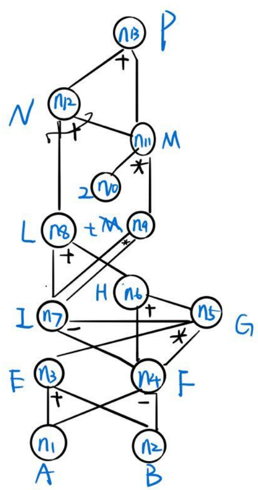
(1) Depth first traversal.

(2) Ancestor node first traversal.

从根节点按深度优先遍历，遇到有父节点尚未遍历的则先遍历其父节点。无附标的叶子节点不遍历。

12:28 2月24日周五 27%

1\.
$E = A + B$
$F = A - B$
$G = E * F$
$H = F + G$
$I = F - G$
$L = I + H$
$M = I * I$
$M = 2 * M$
$N = L + M$
$P = N + M$



n13 n12 n8 n11 n9 n7 n6 n5 n3 n4

$\longleftarrow$

n4 $F = A - B$
n3 $E = A + B$
n5 $G = E * F$
n6 $H = F + G$
n7 $I = F - G$
n9 $t = I * I$
n11 $M = 2 * t$
n8 $L = I + H$
n12 $N = L + M$
n13 $P = N + M$

3 / 4

Exercises

- Please **construct the DAG** for the following basic block, and we assume that only variable “M” is live on exit, please optimize the block and **rewrite the block** in optimized code form

E=A+B

F=E-C

G=F*F

H=E-C

I=H-G

J=I+G

M=J*J

Loops in flow graphs

1. Dominators

We say node d of a flow graph dominates node n, written d dom n, if every path from the initial node of the flow graph to n goes through d

2.Back edge

- The edges whose heads dominate their tails.

3.Find all the loops in a flow graph

- Input. A flow graph G and a back edge n-> d.
- Output. The set loop consisting of all nodes in the natural loop of n->d.

```
void insert (m){
 if m is not in loop {
      loop=loop ∪ {m};
      push m onto stack }}
Main( )
{  Stack=empty;
    Loop={d};
    Insert(n);
   While stack is not empty {
      pop m, the first element of stack, off stack;
      for each predecessor p of m do insert(p) }
}
```

4.Pre-Headers

- Move statements before the header
- Create a new block, preheader. It has only the header as successor,and all edges which formerly entered the header of L from outside L instead enter the preheader.

5.Code optimization in a loop

- Code motion
- Induction Variables and Reduction in Strength

e.g.

j:=1;

for i=1 to 100 do

A[i,j]:=B[i,j] +2

And the array is :

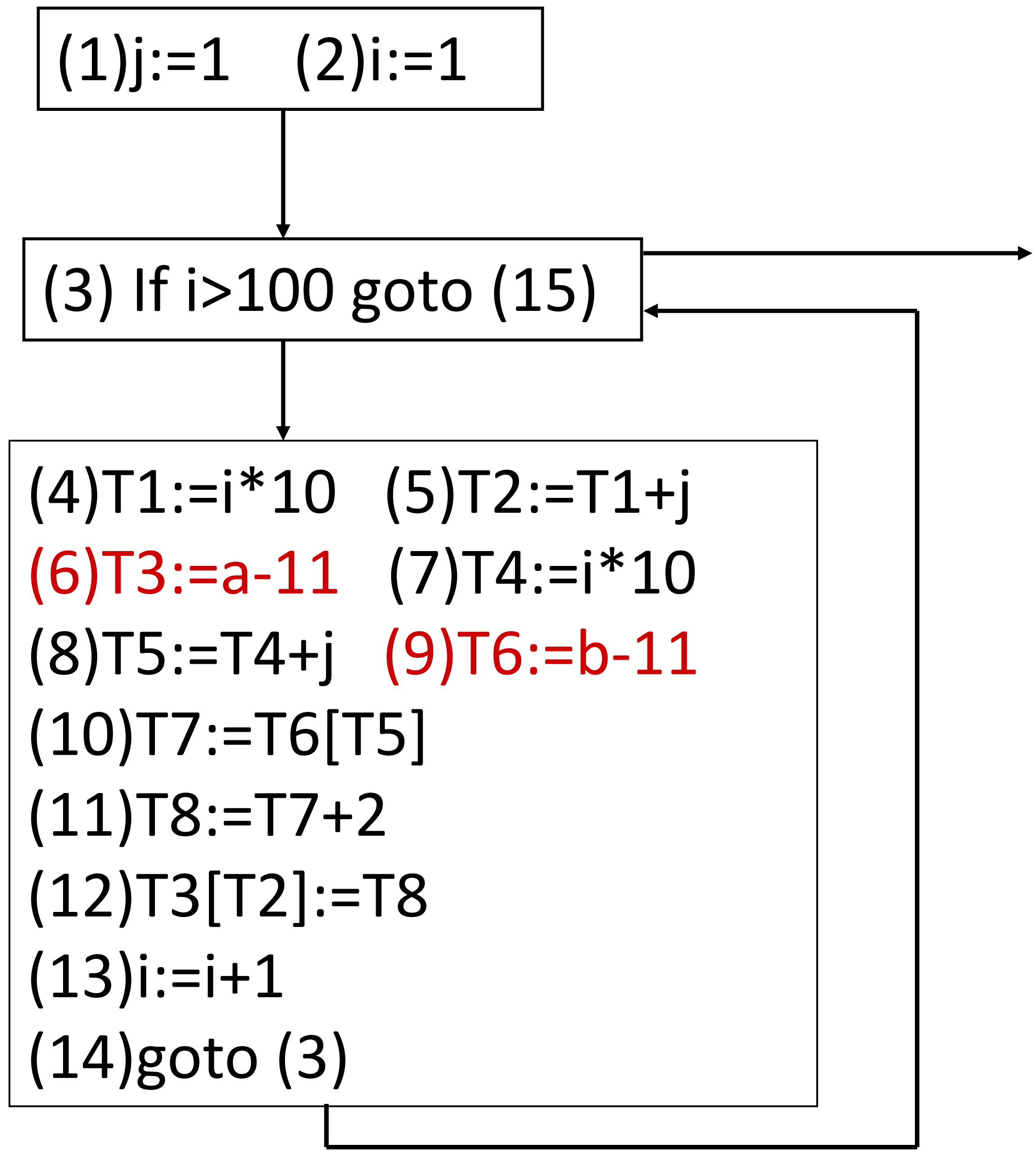
A,B:array[1:100,1:10]

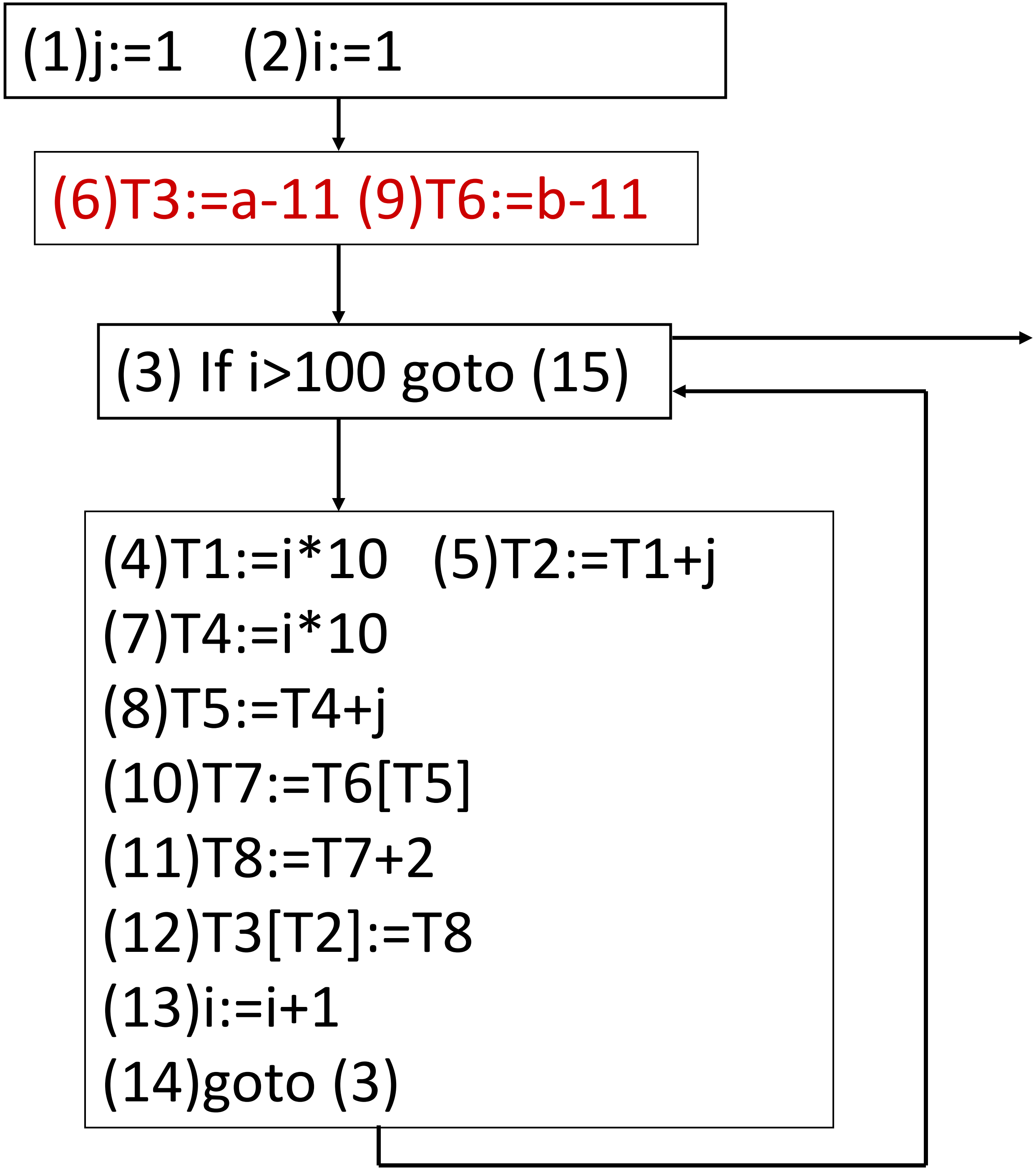
m=1

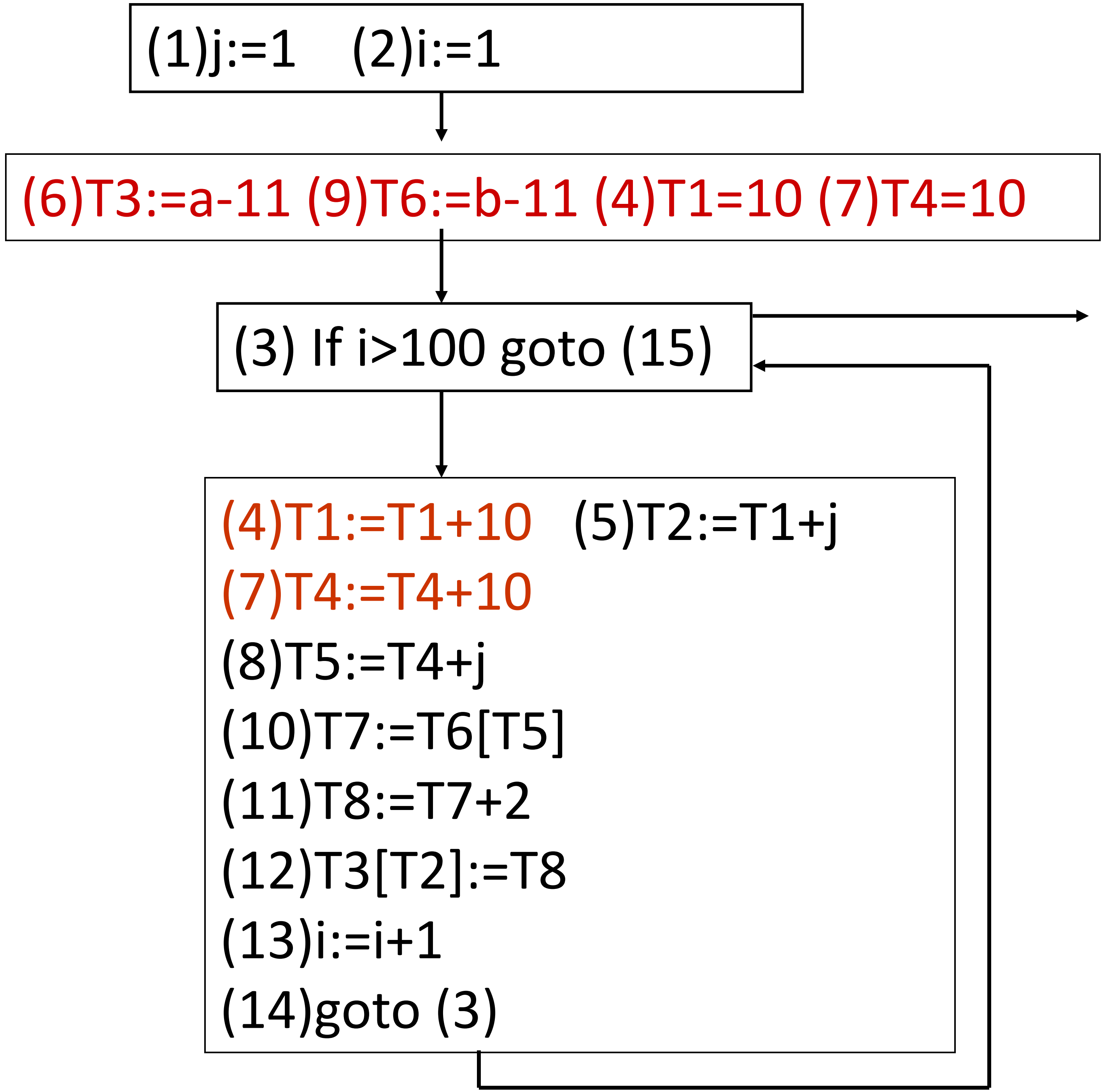
```
(1)j:=1    (2)i:=1
```

```
(3) If i>100 goto (15)
```

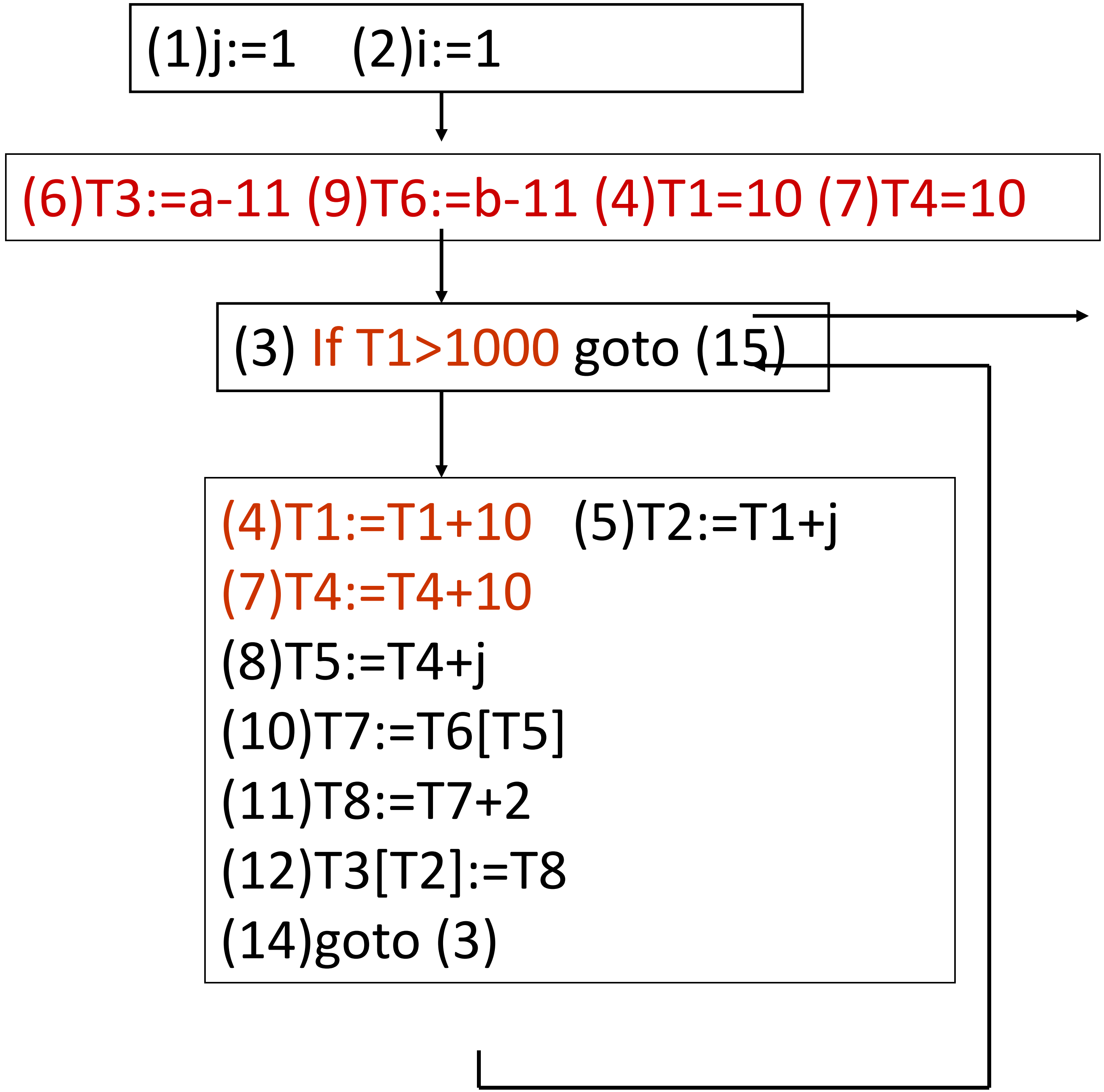
```
(4)T1:=i*10   (5)T2:=T1+j
(6)T3:=a-11   (7)T4:=i*10
(8)T5:=T4+j   (9)T6:=b-11
(10)T7:=T6[T5]
(11)T8:=T7+2
(12)T3[T2]:=T8
(13)i:=i+1
(14)goto (3)
```


(1)j:=1 (2)i:=1
(3) If i>100 goto (15)
(4)T1:=i*10 (5)T2:=T1+j
(6)T3:=a-11 (7)T4:=i*10
(8)T5:=T4+j (9)T6:=b-11
(10)T7:=T6[T5]
(11)T8:=T7+2
(12)T3[T2]:=T8
(13)i:=i+1
(14)goto (3)


(1)j:=1 (2)i:=1
(6)T3:=a-11 (9)T6:=b-11
(3) If i>100 goto (15)
(4)T1:=i*10 (5)T2:=T1+j
(7)T4:=i*10
(8)T5:=T4+j
(10)T7:=T6[T5]
(11)T8:=T7+2
(12)T3[T2]:=T8
(13)i:=i+1
(14)goto (3)



(1)j:=1 (2)i:=1
(6)T3:=a-11 (9)T6:=b-11 (4)T1=10 (7)T4=10
(3) If i>100 goto (15)
(4)T1:=T1+10 (5)T2:=T1+j
(7)T4:=T4+10
(8)T5:=T4+j
(10)T7:=T6[T5]
(11)T8:=T7+2
(12)T3[T2]:=T8
(13)i:=i+1
(14)goto (3)


(1)j:=1 (2)i:=1
(6)T3:=a-11 (9)T6:=b-11 (4)T1=10 (7)T4=10
(3) If T1>1000 goto (15)
(4)T1:=T1+10 (5)T2:=T1+j
(7)T4:=T4+10
(8)T5:=T4+j
(10)T7:=T6[T5]
(11)T8:=T7+2
(12)T3[T2]:=T8
(14)goto (3)

Data-flow analysis

1.Data-flow analysis

- Analyze global data-flow information to do code optimization and a good job of code generation

Notes: Data-flow information can be collected by setting up and solving systems of equations that relate data information at various points in a program flow.

2.Definitions

1)Point

The label of a statement in a program

2)Path

Path from p1 to pn is a sequence of points p1,p2,…, pn such that for each i between 1 and n-1,either

2.Definitions

2)Path

- (1) pi is the point immediately preceding a statement and pi+1 is the point immediately following that statement in the same block, or
- (2) pi is the end of some block and pi+1 is the beginning of a successor block.

2.Definitions

3) Reaching

- A definition of a variable x is a statement that assigns, or may assign, a value to x.
- We say a definition d reaches a point p is there is a path from the point immediately following d to p, such that d is not “killed” along that path.

2.Definitions

3) Reaching

Notes: (1)Intuitively, if a definition d of some variable a reaches point p, then d might be the place at which the value of a used at p might last have been defined.

(2)We kill a definition of a variable a if between two points along the path there is a definition of a.

3.Reaching-Definition Data-flow equations

OUT[B]=(IN[B]–KILL[B])$\cup$GEN[B]

IN[B]= $$\bigcup_{P \in P[B]} OUT[P]$$

P[B] is the predecessor of B

IN[B]:Set of definitions of each reachable variable at the beginning of Block B

KILL[B]:Set of re-defined definitions of variables reached at the beginning of Block B

GEN[B]:Set of definitions of new generated variables in Block B

OUT[B]:Set of definition of each reachable variables at the end of Block B

4. Reaching Algorithm

- Input . A flow graph for which kill[B] for which kill[B] and gen[B] have been computed for each block B.
- Output. In[B] and out[B] for each block B.
- Method

```
{for each block B  do
   { IN[Bi]=Φ;OUT[Bi]=GEN[Bi];  }
  change=TRUE;
  while change do {
     change=FALSE;
     for each block B do {
         IN[B]= ⋃_{P∈P[Bi]} OUT[P]
         oldout=out[B];
         OUT[Bi]=IN[Bi]-KILL[Bi]∪GEN[Bi]
       if out[B]<>oldout
         change=TRUE;
     }
   }
}
```

5.Live-Variable Analysis

- In live-variable analysis we wish to know for variable x and point p whether the value of x at p could be used along some path in the flow graph starting at p.
- If so, we say x is live at p; otherwise x is dead at p.

5.Live-Variable Analysis

L.IN[B]=(L.OUT[B]–L.DEF[B])∪L.USE[B]

$$L.OUT[B] = \bigcup_{S \in S[B]} L.IN[S]$$

S[B] is a successor of B

6.Live variable Algorithm

- Input. A flow graph with L.DEF and L.USE computed for each block.
- Output. L.Out[B], the set of variables live on exit from each block B of the flow graph.
- Method.

```
{for each block B do L.IN[Bi]=Φ;
while change do {
   for each block B do {
      L.OUT[B]= ⋃_{S∈S[Bi]} IN[S]
      IN[B]=(OUT[B]-DEF[B])∪USE[Bi]
   }
 }}
```

7.Definition-Use Chains

- We say a variable is used at statement s if its r-value may be required.
- The du-chaining problem is to compute for a point p the set of uses s of a variable, say x, such that there is a path from p to s that does not redefine x.