PUNE VIDYARTHI GRIHA's

COLLEGE OF ENGINEERING, NASHIK.

" CODE GENERATION "

PREPARED BY :

PROF. ANAND N. GHARU

ASSISTANT PROFESSOR

COMPUTER DEPARTMENT

SUBJECT – COMPILER (BE COMPUTER SPPU-2019)

CONTENTS :

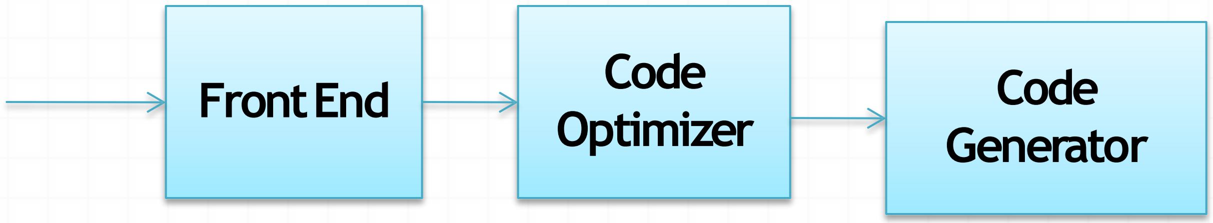
- Code Generation - Issues in code generation, basic blocks, flow graphs, DAG representation of basic blocks, Target machine description, peephole optimization, Register allocation and Assignment, Simple code generator, Code generation from labeled tree, Concept of code generator.

Outline

- Code Generation Issues
- Target Machine Description
- Basic Blocks and Flow Graphs
- Optimizations of Basic Blocks
- A Simple Code Generator
- Peephole optimization
- Register allocation and assignment
- Instruction selection by tree rewriting

INTRODUCTION

- The final phase of a compiler is code generator
- It receives an intermediate representation (IR) with supplementary information in symbol table
- Produces a semantically equivalent target program
- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering



Code Generation

- Code produced by compiler **must be correct**
  - Source-to-target program transformation should be *semantics preserving*
- Code produced by compiler **should be of high quality**
  - Effective use of target machine resources
  - Heuristic techniques should be used to generate good but suboptimal code, because generating optimal code is undecidable
- **code generator should run efficiently**

Issues in the design of codegenerator

- **Input:** Intermediate representation with symbol table assume that input has been validated by the front end

- **Target programs :**

  The back-end code generator of a compiler may generate different forms of code, depending on the requirements:

  – Absolute machine code (executable code)

  – Relocatable machine code (object files for linker)

  – Assembly language (facilitates debugging)

  – Byte code forms for interpreters (e.g. JVM)

Issues in the design of codegenerator

- **Target Machine:**

  Implementing code generation requires thorough understanding of the target machine architecture and its instruction set

Issues in the design of codegenerator

Instruction Selection:

- Instruction selection is important to obtain efficient code

```
a:=a+1  ⇒  MOV a,R0
           ADD #1,R0
           MOV R0,a
```

Cost = 6

Better ⇓

```
ADD #1,a
```

Cost = 3

Best ⇓

```
INC a
```

Cost = 2

Issues in the design of code generator Instruction Selection

- Suppose we translate `a:=b+c` into

```
MOV b,R0
ADD c,R0
MOV R0,a
```

- Assuming addresses of `a`, `b`, and `c` are stored in `R0`, `R1`, and `R2`

```
MOV *R1,*R0
ADD *R2,*R0
```

- Assuming `R1` and `R2` contain values of `b` and `c`

```
ADD R2,R1
    MOV R1,a
```

Issues in the design of code generator Instruction Selection

- Suppose we translate three-address code

```
        x:=y+z
to:     MOV y,R0
        ADD z,R0
        MOV R0,x
```

- Then, we translate

```
a:=b+c   d:=a+e

to:     MOV a,R0
        ADD b,R0
        MOV R0,a
        MOV a,R0    ← Redundant
        ADD e,R0
        MOV R0,d
```

Issues in the design of code generator Register Allocation and Assignment

- Efficient utilization of the limited set of registers is important to generate good code
- Registers are assigned by
  - *Register allocation* to select the set of variables that will reside in registers at a point in the code
  - *Register assignment* to pick the specific register that a variable will reside in
- Finding an optimal register assignment in general is NP-complete

Issues in the design of code generator : Register Allocation and Assignment : Example

```
t:=a*b
t:=t+a
t:=t/d
```

{R1=t}

```
MOV  a,R1
MUL  b,R1
ADD  a,R1
DIV  d,R1
MOV R1,t
```

```
t:=a*b
t:=t+a
t:=t/d
```

{R0=a, R1=t}

```
MOV  a,R0
MOV R0,R1
MUL  b,R1
ADD R0,R1
DIV  d,R1
MOV R1,t
```

Issues in the design of code generator Choice of Evaluation Order

- When instructions are independent, their evaluation order can be changed

```
a+b-(c+d)*e
```

```
t1:=a+b
t2:=c+d
t3:=e*t2
t4:=t1-t3
```

```
MOV a,R0
ADD b,R0
MOV R0,t1
MOV c,R1
ADD d,R1
MOV e,R0
MUL R1,R0
MOV t1,R1
SUB R0,R1
MOV R1,t4
```

reorder

```
t2:=c+d
t3:=e*t2
t1:=a+b
t4:=t1-t3
```

```
MOV c,R0
ADD d,R0
MOV e,R1
MUL R0,R1
MOV a,R0
ADD b,R0
SUB R1,R0
MOV R0,t4
```

A simple target machine model

- Load operations: LD r,x and LD r1, r2
- Store operations: ST x,r
- Computation operations: OP dst, src1, src2
- Unconditional jumps: BR L
- Conditional jumps: Bcond r, L like BLTZ r, L

Addressing Modes

- variable name: x
- indexed address: a(r) like LD R1, a(R2) means R1=contents(a+contents(R2))
- integer indexed by a register : like LD R1, 100(R2)
- Indirect addressing mode: *r and *100(r)
- immediate constant addressing mode: like LD R1, #100

b = a [i]

```
LD R1, i                  //R1 =i
MUL R1, R1, 8             //R1 =Rl * 8
LD R2, a(R1)
        //R2=contents(a+contents(R1))
ST b, R2                  //b =R2
```

a[j] =c

```
LD R1, c               //R1 =c
LD R2, j               // R2 =j
MUL R2, R2, 8          //R2 =R2 * 8
ST a(R2), R1
        //contents(a+contents(R2))=R1
```

x=*p

```
LD R1, p                //R1 =p

LD R2, 0(R1)            // R2=
   contents(0+contents(R1))

ST  x, R2               // x=R2
```

conditional-jump three-address instruction

If x<y goto L

```
LD R1, x          // R1=x
LD R2, y          // R2=y
SUB R1, R1, R2    // R1=R1-R2
BLTZ R1, M        // if R1<0 jump to M
```

costs associated with the addressingmodes

- LD R0, R1 cost = 1
- LD R0, M cost = 2
- LD R1, *100(R2) cost = 3

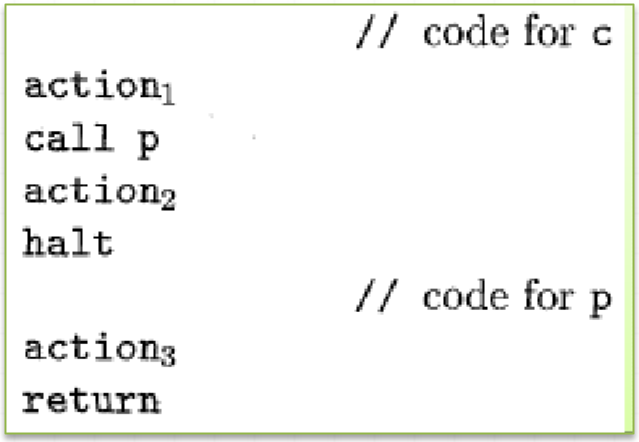
Addresses in the Target Code

- Astatically determined area Code
- Astatically determined data area Static
- Adynamically managed area Heap
- Adynamically managed area Stack

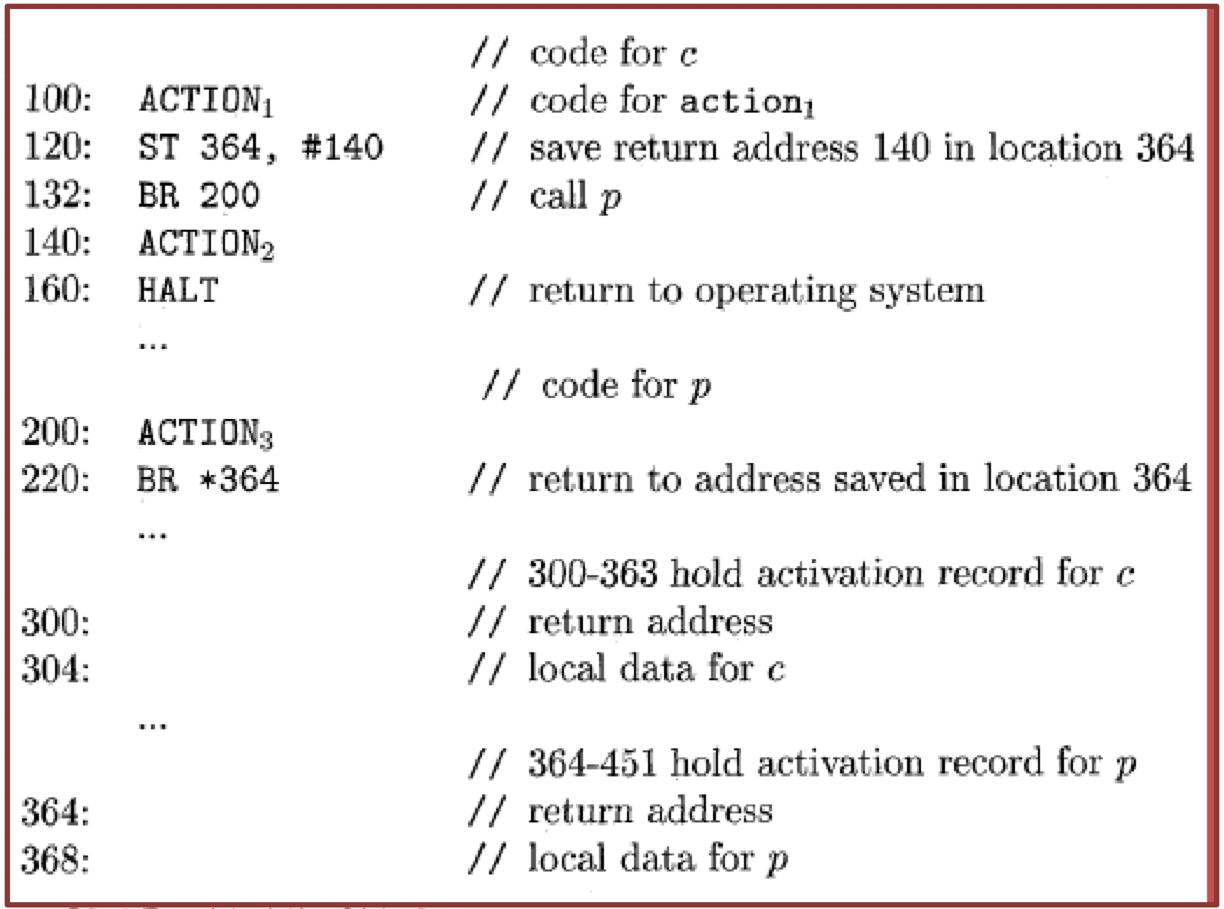
Three-address statements for procedure calls and returns

- call callee
- Return
- Halt
- action

Target program for a sample call and return

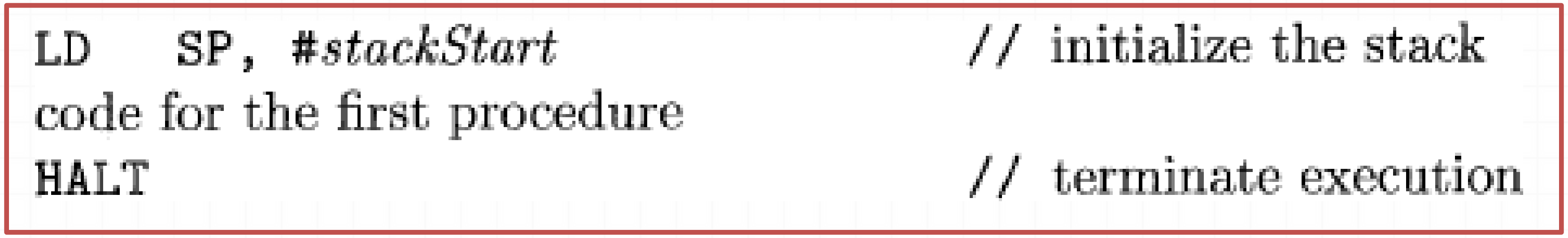


```
                        // code for c
action₁
call p
action₂
halt
                        // code for p
action₃
return
```

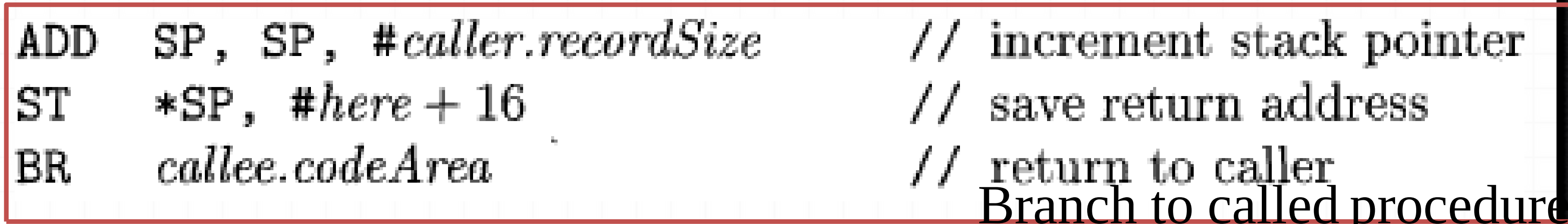


```
                             // code for c
100:  ACTION₁                // code for action₁
120:  ST 364, #140           // save return address 140 in location 364
132:  BR 200                 // call p
140:  ACTION₂
160:  HALT                   // return to operating system
      ...
                             // code for p
200:  ACTION₃
220:  BR *364                // return to address saved in location 364
      ...
                             // 300-363 hold activation record for c
300:                         // return address
304:                         // local data for c
      ...
                             // 364-451 hold activation record for p
364:                         // return address
368:                         // local data for p
```

Stack Allocation



```
LD    SP, #stackStart             // initialize the stack
code for the first procedure
HALT                              // terminate execution
```



```
ADD   SP, SP, #caller.recordSize  // increment stack pointer
ST    *SP, #here + 16             // save return address
BR    callee.codeArea             // return to caller
```

Branch to called procedure

Return to caller

in Callee: BR *0(SP)

in caller: SUB SP, SP, #caller.recordsize

Target code for stack allocation

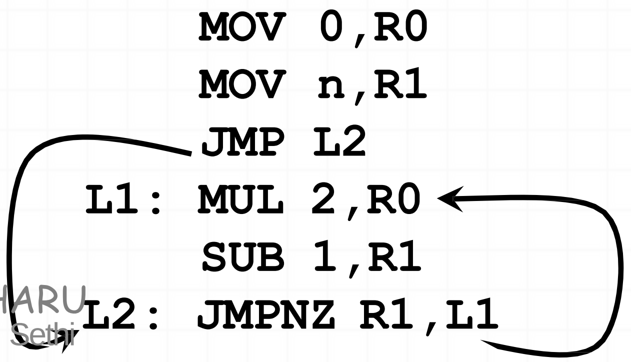
```
                    // code for m
action₁
call q
action₂
halt
                    // code for p
action₃
return
                    // code for q
action₄
call p
action₅
call q
action₆
call q
return
```

```
                                 // code for m
100:  LD SP, #600                // initialize the stack
108:  ACTION₁                    // code for action₁
128:  ADD SP, SP, #msize         // call sequence begins
136:  ST *SP, #152               // push return address
144:  BR 300                     // call q
152:  SUB SP, SP, #msize         // restore SP
160:  ACTION₁2
180:  HALT
      ...
                                 // code for p
200:  ACTION₃
220:  BR *0(SP)                  // return
      ...
                                 // code for q
300:  ACTION₄                    // contains a conditional jump to 456
320:  ADD SP, SP, #qsize
328:  ST *SP, #344               // push return address
336:  BR 200                     // call p
344:  SUB SP, SP, #qsize
352:  ACTION₅
372:  ADD SP, SP, #qsize
380:  BR *SP, #396               // push return address
388:  BR 300                     // call q
396:  SUB SP, SP, #qsize
404:  ACTION₆
424:  ADD SP, SP, #qsize
432:  ST *SP, #440               // push return address
440:  BR 300                     // call q
448:  SUB SP, SP, #qsize
456:  BR *0(SP)                  // return
      ...
600:                             // stack starts here
```

Flow Graphs

- A *flow graph* is a graphical depiction of a sequence of instructions with control flow edges
- A flow graph can be defined at the intermediate code level or target code level

```
    MOV 1,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```



```
    MOV 0,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```

Basic Blocks

- A *basic block* is a sequence of consecutive instructions with exactly one entry point and one exit point (with natural flow or a branch instruction)

```
    MOV 1,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```

```
    MOV 1,R0
    MOV n,R1
    JMP L2
```

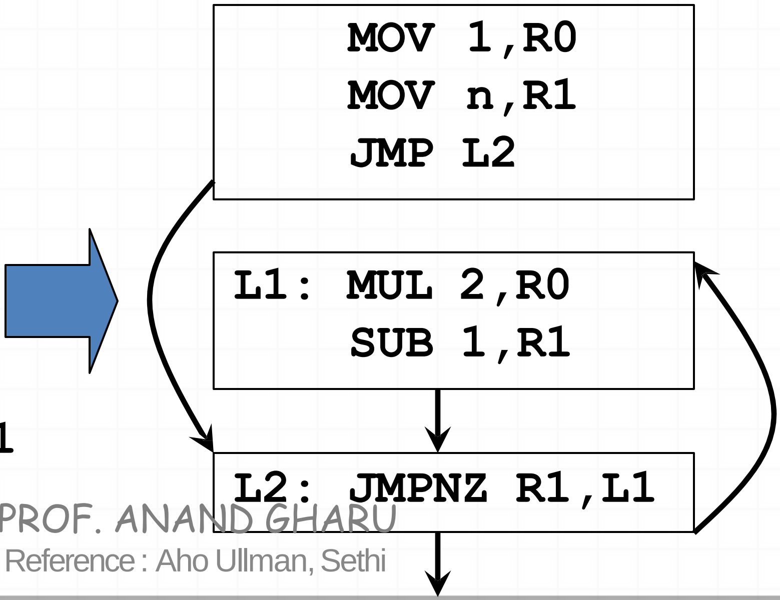
```
L1: MUL 2,R0
    SUB 1,R1
```

```
L2: JMPNZ R1,L1
```

Basic Blocks and Control Flow Graphs

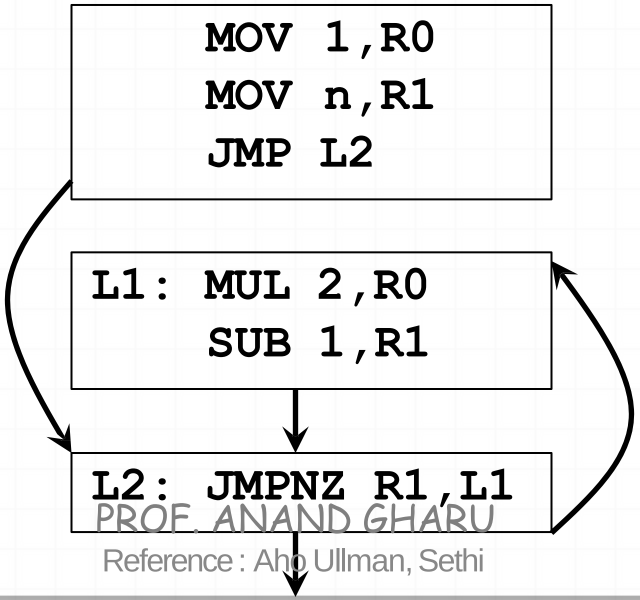
- A *control flow graph* (CFG) is a directed graph with basic blocks B_i as vertices and with edges $B_i \rightarrow B_j$ iff B_j can be executed immediately after B_i

```
     MOV 1,R0
     MOV n,R1
     JMP L2
L1:  MUL 2,R0
     SUB 1,R1
L2:  JMPNZ R1,L1
```



Successor and Predecessor Blocks

- Suppose the CFG has an edge $B_1 \rightarrow B_2$
  - Basic block B_1 is a *predecessor* of B_2
  - Basic block B_2 is a *successor* of B_1



```
      MOV  1,R0
      MOV  n,R1
      JMP  L2

L1:   MUL  2,R0
      SUB  1,R1

L2:   JMPNZ R1,L1
```

Partition Algorithm for Basic Blocks

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

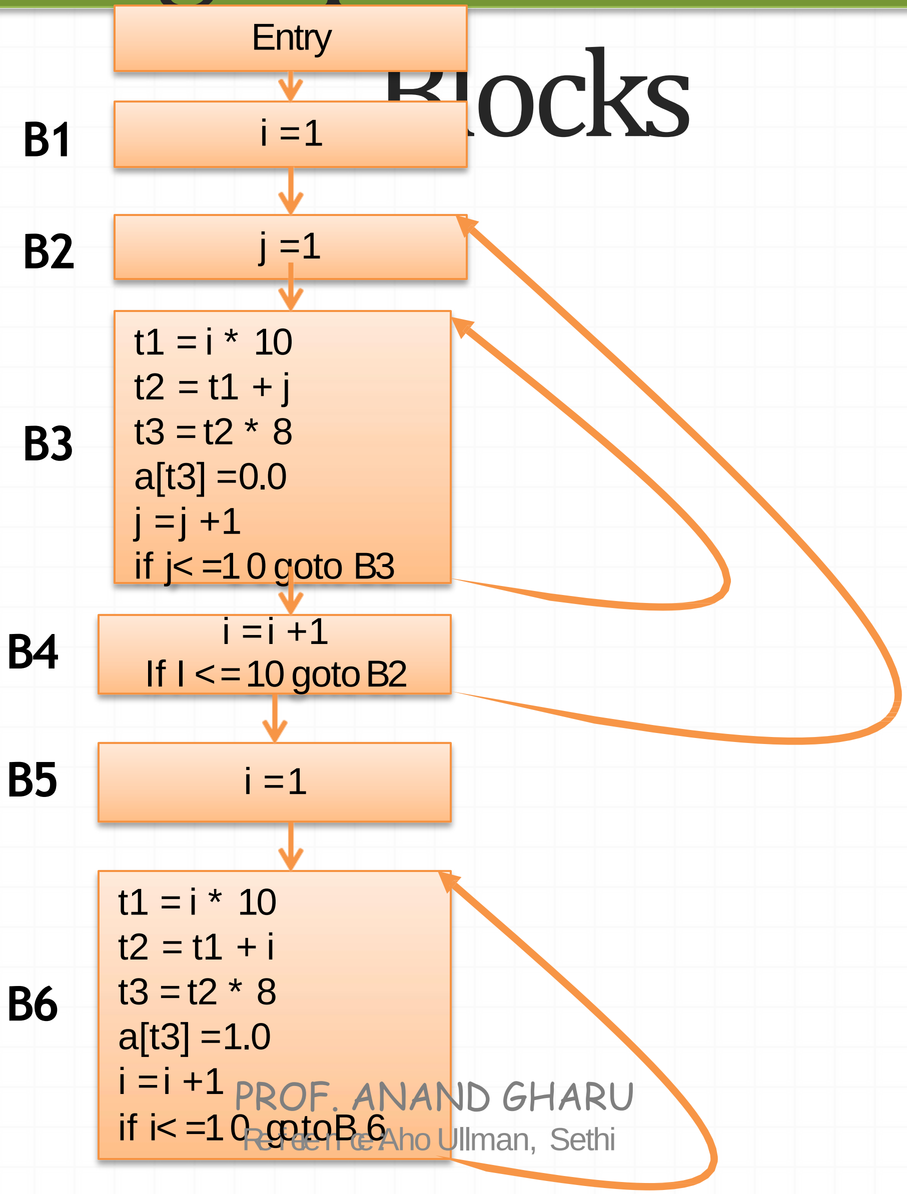
1. **Determine the set of *leaders*, the first statements of basic blocks**
   a) The **first statement** is the leader
   b) Any statement that is **the target of a goto** is a leader
   c) Any statement **that immediately follows a goto** is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Intermediate code to set a 10*10 matrix to an identity matrix

```
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i, j] = 0.0;
for i from 1 to 10 do
    a[i, i] = 1.0;
```

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

Flow graph based on Basic Blocks



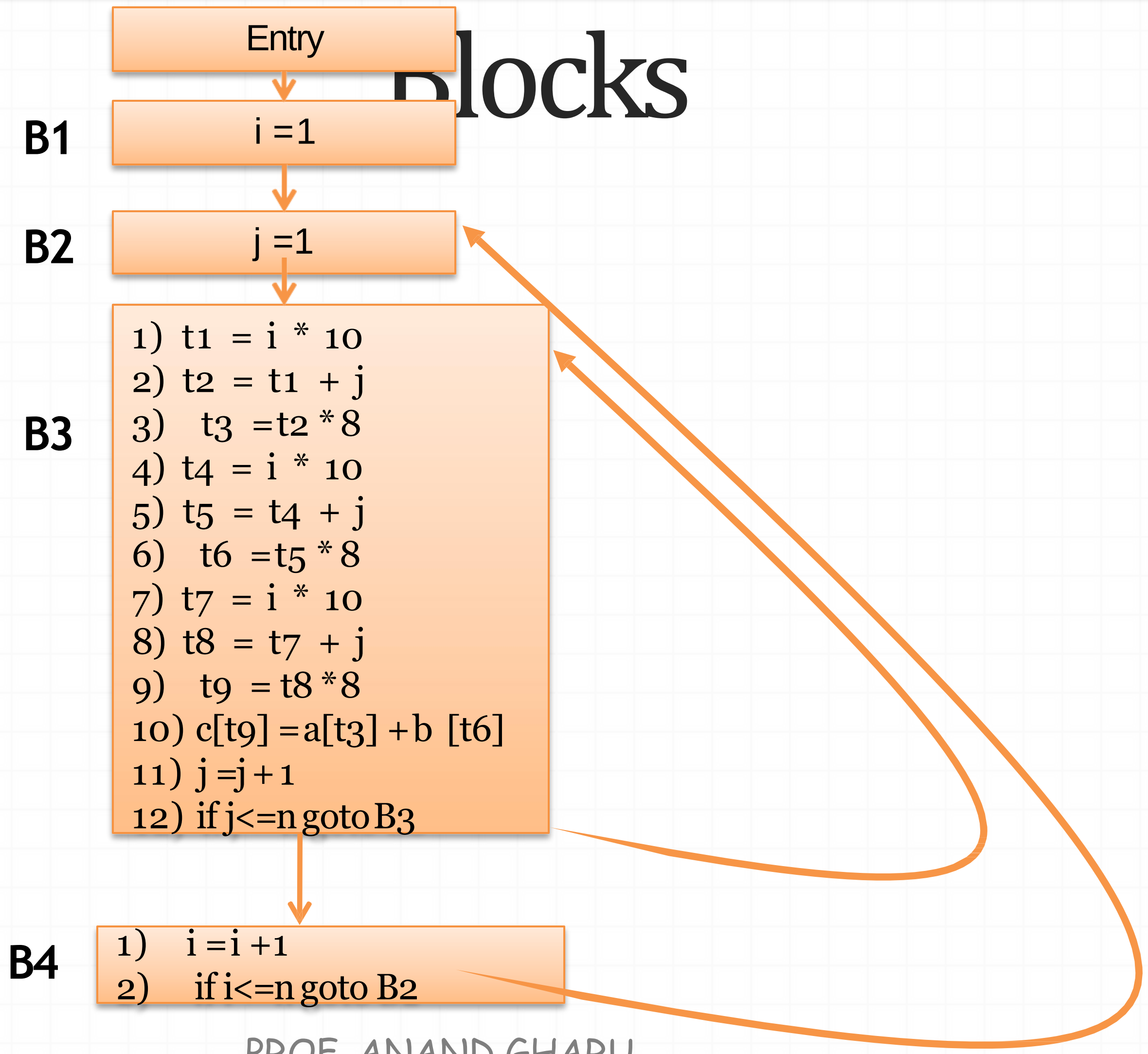
Intermediate code to set a 10*10 matrix to an identity matrix

Generate TAC and Partition below code into basic blocks

```
for i=1 to n
    for j=1 to n
        c[i,j] = a[i,j] +b[i,j]
```

```
1)  i = 1
2)  j = 1
3) t1  = i * 10
4) t2  = t1  + j
5)  t3  = t2 * 8
6) t4  = i  * 10
7) t5  = t4  + j
8)  t6  =t5 * 8
9) t7  = i  * 10
10)t8  = t7  + j
11)t9  =t8 *  8
12) c[t9] = a[t3] + b [t6]
13)j =j + 1
14)if j<=n goto (3)
15)i  =i +1
16) if i<=n goto (2)
```

Flow graph based on Basic Blocks

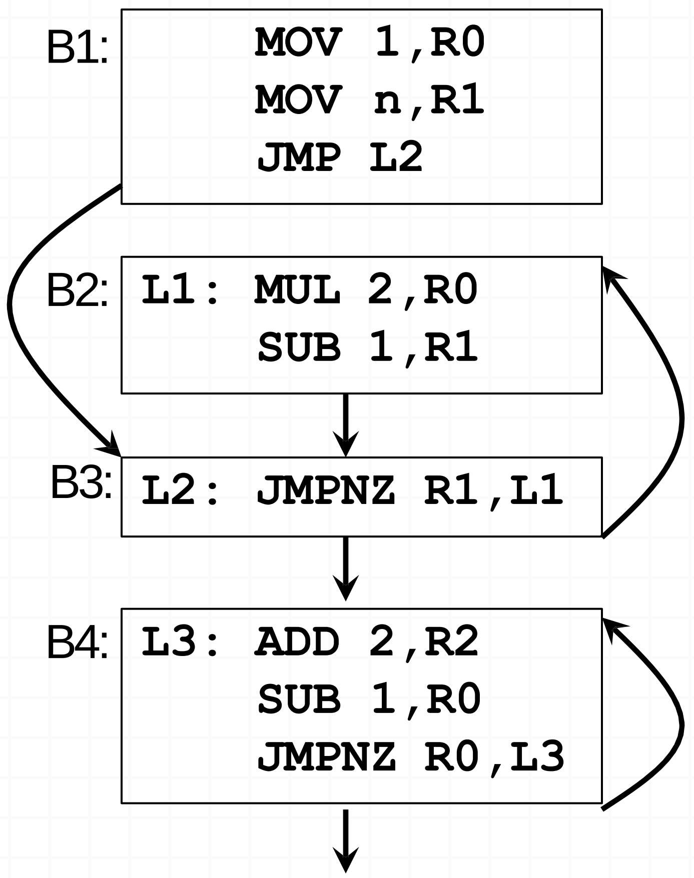


```
B1:
i =1

B2:
j =1

B3:
1) t1  = i  *  10
2) t2  = t1  + j
3)   t3  =t2 * 8
4) t4  = i  *  10
5) t5  = t4  + j
6)   t6  =t5 * 8
7) t7  = i  *  10
8) t8  = t7  + j
9)   t9  = t8 *8
10) c[t9] =a[t3] +b  [t6]
11) j =j +1
12) if j<=n goto B3

B4:
1)    i =i +1
2)     if i<=n goto B2
```

Loops

- A *loop* is a collection of basic blocks, such that
  - All blocks in the collection are *strongly connected*
  - The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry

Loops (Example)



```
B1:       MOV  1,R0
          MOV  n,R1
          JMP  L2

B2:  L1:  MUL  2,R0
          SUB  1,R1

B3:  L2:  JMPNZ R1,L1

B4:  L3:  ADD  2,R2
          SUB  1,R0
          JMPNZ R0,L3
```

Strongly connected components:

SCC={ {B2,B3}, {B4} }

Entries: B3, B4

Equivalence of Basic Blocks

- Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions

```
b   := 0
t1  := a + b
t2  := c * t1
a   := t2
```

```
a  := c*a
b  := 0
```

```
a   := c * a
b   := 0
```

```
a  := c*a
b  := 0
```

Blocks are equivalent, assuming **t1** and **t2** are *dead*: no longer used (no longer *live*)

Transformations on Basic Blocks

- A ***code-improving transformation*** is a code optimization to **improve speed or reduce code size**
- ***Global transformations*** are performed **across basic blocks**
- ***Local transformations*** are only performed **on single basic blocks**
- Transformations must be safe and preserve the meaning of the code
  - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form

Transformations on Basic Blocks

- Common Subexpression Elimination
- Dead Code Elimination
- Renaming Temporary Variables
- Interchange of Statements
- Algebraic Transformations
- Next-Use Computation

Common-Subexpression Elimination

- Remove redundant computations

```
a := b + c
b := a - d
c := b + c
d := a - d
```

$\Rightarrow$

```
a := b + c
b := a - d
c := b + c
d := b
```

```
t1 := b * c
t2 := a - t1
t3 := b * c
t4 := t2 + t3
```

$\Rightarrow$

```
t1 := b * c
t2 := a - t1
t4 := t2 + t1
```

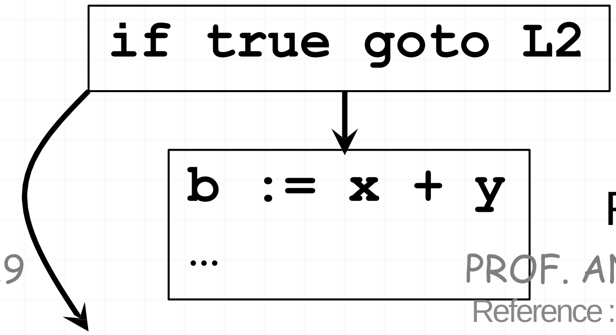
Dead Code Elimination

- Remove unused statements

```
b := a + 1
a := b + c
…
```

```
b := a + 1
…
```

Assuming **a** is *dead* (not used)



```
if true goto L2
```

```
b := x + y
…
```

Remove unreachable code

Renaming Temporary Variables

- Temporary variables that are dead at the end of a block can be safely renamed

```
t1 := b + c
t2 := a - t1
t1 := t1 * d
d := t2 + t1
```

$\Rightarrow$

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

Normal-form block

Interchange of Statements

- Independent statements can be reordered

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

```
t1 := b + c
t3 := t1 * d
t2 := a - t1
d := t2 + t3
```

Note that normal-form blocks permit all statement interchanges that are possible

Algebraic Transformations

- Change arithmetic operations to transform blocks to algebraic equivalent forms

```
t1 := a - a
t2 := b + t1
t3 := 2 * t2
```

$\Rightarrow$

```
t1 := 0
t2 := b
t3 := t2 << 1
```

Next-Use

- Next-use information is needed for dead-code elimination and register assignment
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement

  i: x := y op z

  - Add liveness/next-use info on x, y, and z to statement i
  - Set x to "not live" and "no next use"
  - Set y and z to "live" and the next uses of y and z to i

Example

$1: t_1 = a * a$

$2: t_2 = a * b$

$3: t_3 = 2 * t_2$

$4: t_4 = t_1 + t_3$

$5: t_5 = b * b$

$6: t_6 = t_4 + t_5$

$7: X = t_6$

Example

1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$

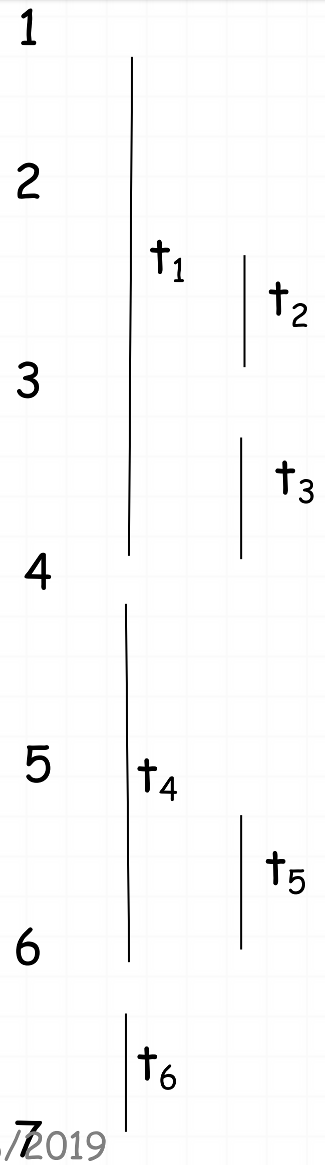
STATEMENT

7: no temporary is live
6: t_6:use(7), t_4 t_5 not live
5: t_5:use(6)
4: t_4:use(6), t_1 t_3 not live
3: t_3:use(4), t_2 not live
2: t_2:use(3)
1: t_1:use(4)

Symbol Table

t_1	dead	Use in 4
t_2	dead	Use in 3
t_3	dead	Use in 4
t_4	dead	Use in 6
t_5	dead	Use in 6
t_6	dead	Use in 7

Example ...



1: $t_1 = a * a$

2: $t_2 = a * b$

3: $t_2 = 2 * t_2$

4: $t_1 = t_1 + t_2$

5: $t_2 = b * b$

6: $t_1 = t_1 + t_2$

7: $X = t_1$

A Code Generator

- Generates target code for a sequence of three-address statements using next-use information
- Uses new function ***getreg*** to **assign registers to variables**
- Computed results are kept in registers as long as possible, which means:
  - Result is needed in another computation
  - Register is kept up to a procedure call or end of block
- Checks if operands to three-address code are available in registers

Code Generator : Register and Address Descriptors

- A ***register descriptor*** keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

```
MOV a,R0
```
"**R0** contains **a**"

- An ***address descriptor*** keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.

```
MOV a,R0
    MOV R0,R1
```
"**a** in **R0** and **R1**"

The Code Generation Algorithm

- For each statement *x* := *y* op *z*

1. Set location *L* = *getreg*(*y*, *z*)
2. If *y* $\notin$ *L* then generate

   MOV *y'*,*L*

   where *y'* denotes one of the locations where the value of *y* is available (choose register if possible)
3. Generate

   OP *z'*,*L*

   where *z'* is one of the locations of *z*;
   Update register/address descriptor of *x* to include *L*
4. If *y* and/or *z* has no next use and is stored in register, update register descriptors to remove *y* and/or *z*

The *getreg* Algorithm

- To compute *getreg*(*y*,*z*)
  1. If *y* is stored in a register *R* and *R* only holds the value *y*, and *y* has no next use, then return *R*;
     Update address descriptor: value *y* no longer in *R*
  2. Else, return a new empty register if available
  3. Else, find an occupied register *R*;
     Store contents (register spill) by generating
     `MOV` *R* , *M*
     for every *M* in address descriptor of *y*;
     Return register *R*
  4. Return a memory location

Example

Stmt	code	reg desc	addr desc
t_1=a-b	mov a,R_0	R_0contains t_1	t_1 in R_0
	sub b,R_0		
t_2=a-c	mov a,R_1	R_0contains t_1	t_1 in R_0
	sub c,R_1	R_1 contains t_2	t_2 in R_1
t_3=t_1+t_2	add R_1,R_0	R_0 contains t_3	t_3 in R_0
		R_1 contains t_2	t_2 in R_1
d=t_3+t_2	add R_1,R_0	R_0contains d	d in R_0
	mov R_0,d		d in $R_0$and memory

Register Allocation and Assignment

- Global Register Allocation
- Usage Counts
- Register Assignment for Outer Loops
- Register Allocation by Graph Coloring

Global register allocation

- Previously explained algorithm does local (block based) register allocation
- This resulted that all live variables be stored at the end of block
- To save some of these stores and their corresponding loads, **we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally)**
- Some options are:
  - **Keep values of variables used in loops inside registers**
  - **Use graph coloring approach for more globally allocation**

Usage counts

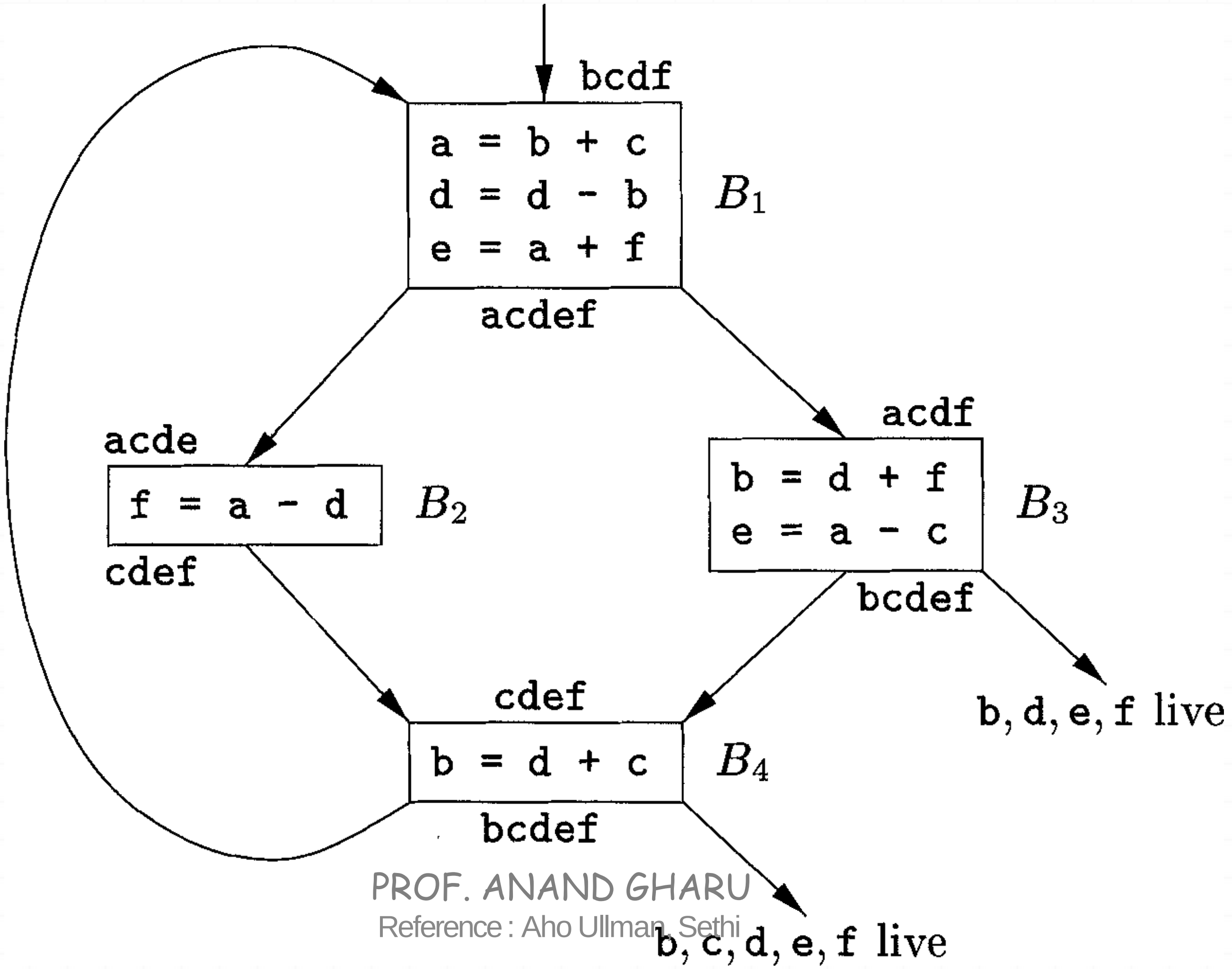
- Usage Count means to determine that after the definition of x , how many subsequent uses of x exists in that block
- For the loops we can approximate the saving by register allocation as: Sum over all blocks (B) in a loop (L)
  - For each uses of x before any definition in the block we add one unit of saving
  - If x is live on exit from B and is assigned a value in B, then we add 2 units of saving

i.e

$$\sum_{\substack{\text{For all} \\ \text{Blocks B in L}}} (use(x,B) + 2 * live(x,B))$$

- This is used for register assignment

Example : Flow graph of an inner loop



Usage counts

- Now, compute Usage Count for each variable considering each block.
- Use of a in B1 prior to its definition in B1 is 0

  ∴ use(a,B1) =0
- 'a' is live on exit from B1 and is assigned a value in B1 ∴ live(a,B1) = 1
- Ill larly, use(a,B2) = 1 , live(a,B2) = 0

  use(a,B3) = 1 , live(a,B3) = 0

  use(a,B4) = 0, live (a,B4) = 0

∴ usageCount(a) = (0+2*1) + (1+ 2* 0) + (1 +2* 0) +(0 + 2* 0)

2 +1 +1+ 0 = 4

∴ usageCount(a) = 4

Usage counts

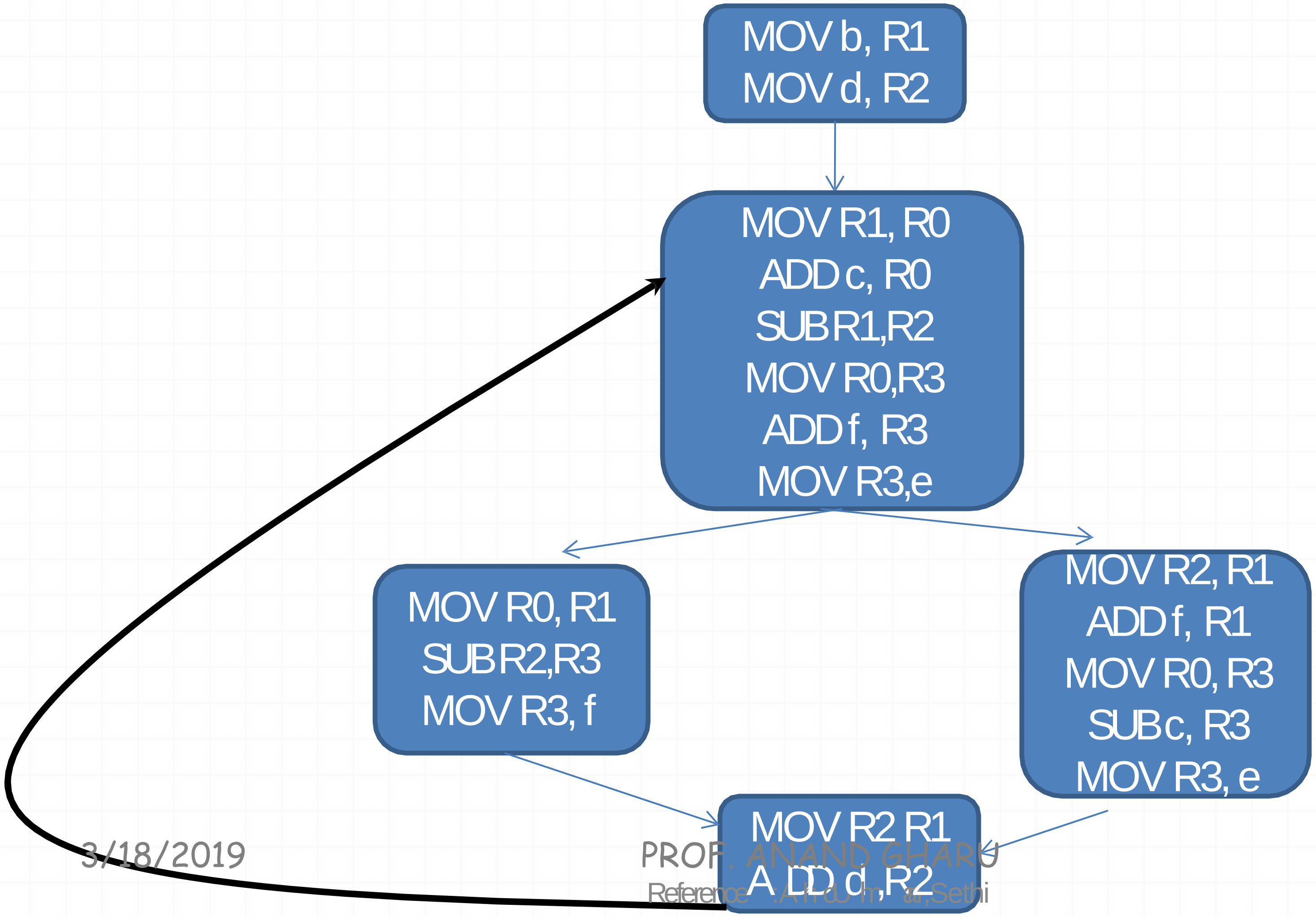
- Now, computing Usage Count for each variable, we get

Bloc k Variable	B1	B2	B3	B4	Total
a	0+2*1=2	1+2*0=1	1+2*0=1	0+2*0=0	4
b	2+2*0=2	0+2*0=0	0+2*1=2	0+2*1=2	6
c	1+2*0=1	0+2*0=0	1+2*0=1	1+2*0=1	3
d	1+2*1=3	1+2*0=1	1+2*0=1	1+2*0=1	6
e	0+2*1=2	0+2*0=0	0+2*1=2	0+2*0=0	4
f	1+2*0=1	0+2*1=2	1+2*0=1	0+2*0=0	4

Usage counts

- Thus, we may select ‘a’,’b’,’d’ for registers , since their usage count is largest.
- We can use ‘e’ or ‘f’ too instead of ‘a’

Code sequence using global register assignment



```
MOV b, R1
MOV d, R2
```

```
MOV R1, R0
ADD c, R0
SUB R1,R2
MOV R0,R3
ADD f, R3
MOV R3,e
```

```
MOV R0, R1
SUB R2,R3
MOV R3, f
```

```
MOV R2, R1
ADD f, R1
MOV R0, R3
SUB c, R3
MOV R3, e
```

```
MOV R2 R1
ADD d ,R2
```

Global Register Allocation with Graph Coloring

- When a register is needed but all available registers are in use, the content of one of the used registers must be stored (spilled) into a memory location to free a register
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a *conflict graph* (*interference graph*)
- Find a *k*-coloring for the graph, with *k* the number of registers

Register allocation by Graph coloring

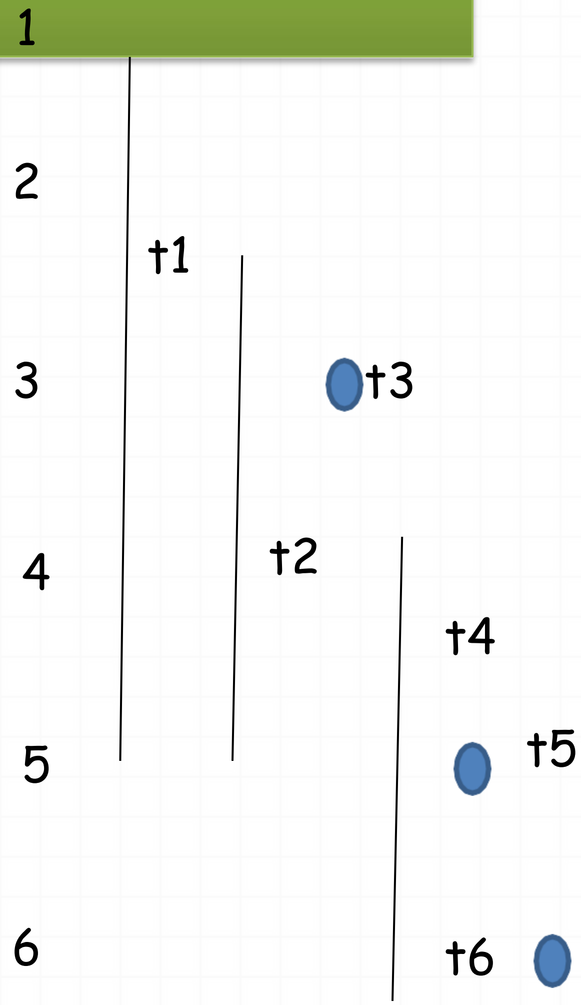
- Two passes are used
  - Target-machine instructions are selected as though there are an infinite number of symbolic registers
  - Assign physical registers to symbolic ones
    - Create a register-interference graph
    - Nodes are symbolic registers and edges connects two nodes if one is live at a point where the other is defined.
    - Use a graph coloring algorithm to assign registers.

Exampl
e

STATEMENT

1: x = 2
2: y = 4
3: w = x + y
4: z = x + 1
5: u = x * y
6: x = z * 2

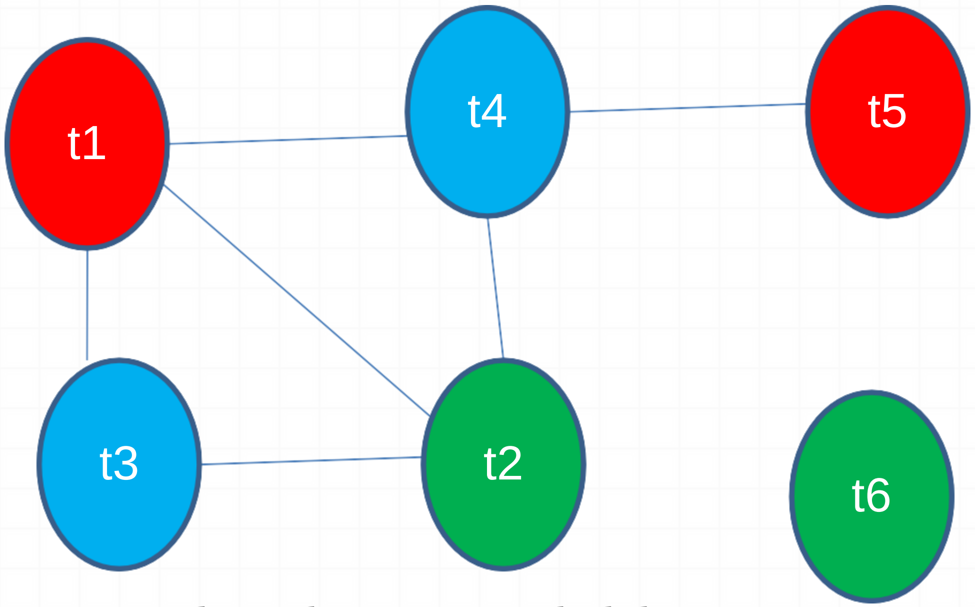
t1 = 2
t2 = 4
t3 = t1 + t2
t4 = t1 + 1
t5 = t1 * t2
t6 = t4 * 2



Register allocation by Graph coloring

- Construct Interference Graph
- Connect nodes with edges that interfere with other
- Then, Color the interference graph such that nodes connected to each other have different colors
- Then Allocate variables having the same color with same registers

Register allocation by Graph coloring



Now, consider three available registers R1, R2 and R3

t1 & t5 reside in R1 say red
t2 & t6 reside in R2 say green
t3 & t4 reside in R3 say blue

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs
- Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence
- Peephole is a small moving window on the target systems

Peephole optimization examples…

1. Redundant loads and stores

- Consider the code sequence

Move R_0, a

Move a, R_0

- Instruction 2 can always be removed if it does not have a label.

Peephole optimization examples…

2. Unreachable code

- Consider following code sequence

```
#define debug 0
if (debug) {
        print debugging info
}
```

this may be translated as

```
        if debug = 1  goto L1
        goto L2
L1: print debugging info
L2:
```

Eliminate jump over jumps

```
        if debug <>1  goto L2
                print debugging information
L2:
```

Unreachable code example …

constant propagation

```
        if 0 <> 1 goto L2
            print debugging information
L2:
```

Evaluate boolean expression. Since if condition is always true the code becomes

```
        goto L2
        print debugging information
L2:
```

The print statement is now unreachable. Therefore, the code becomes

```
L2:
```

Peephole optimization examples…

3. Flow of control: replace jump sequences

```
      goto L1                 by          goto L2
      …                                        …
      …                                        …
L1 : goto L2                             L1: goto L2
```

4. Simplify algebraic expressions

remove x := x+0 or   x:=x*1

Peephole optimization examples…

5. Strength reduction
   - Replace X^2 by X*X
   - Replace multiplication by left shift
   - Replace division by right shift

6. Use faster machine instructions

replace Add #1,R

by Inc R

Example

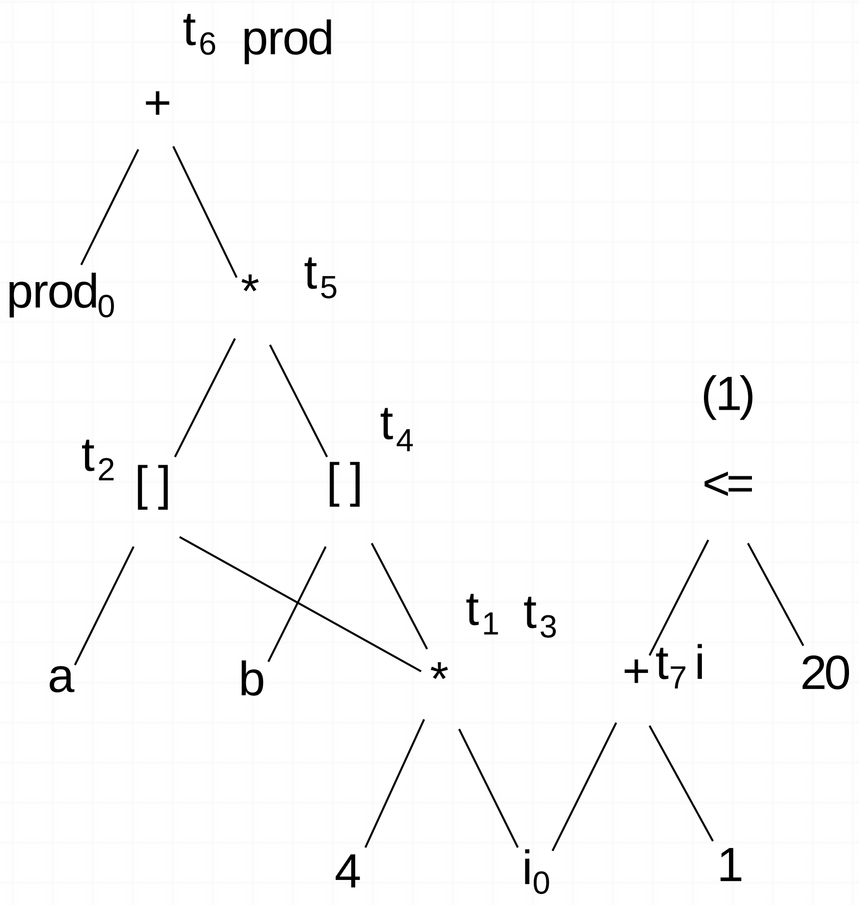
STATEMENT	Finally Code generated is
t1 =2	MOV #2,R1
t2 =4	MOV#4, R2
t3 =t1 +t2	ADD R1, R2 MOV R2, R3
t4 =t1 +1	ADD #1,R1 MOV R1, R3
t5 =t1 * t2	MUL R2,R1
t6 =t4 * 2	MUL#2, R3 MOV R3, R2

DAG representation of basic blocks

- Useful data structures for implementing transformations on basic blocks
- Gives a picture of how value computed by a statement is used in subsequent statements
- Good way of determining common sub-expressions
- A dag for a basic block has following labels on the nodes
  - **leaves are labeled by unique identifiers, either variable names or constants**
  - **interior nodes are labeled by an operator symbol**
  - **nodes are also optionally given a sequence of identifiers for labels**

DAG representation: example

1. $t_1 := 4 * i$
2. $t_2 := a[t_1]$
3. $t_3 := 4 * i$
4. $t_4 := b[t_3]$
5. $t_5 := t_2 * t_4$
6. $t_6 := prod + t_5$
7. $prod := t_6$
8. $t_7 := i + 1$
9. $i := t_7$
10. if $i <= 20$ goto (1)



DAGrepresentation in form of Intermediate code

$S_1 = 4 * i$
$S_2 = c(A)$
$S_3 = S_2[S_1]$
$S_4 = 4 * i$
$S_5 = c(B)$
$S_6 = S_5[S_4]$
$S_7 = S_3 * S_6$
$S_8 = prod + S_7$
$prod = S_8$
$S_9 = i + 1$
$i = S_9$
If i <= 20 goto (1)

Final I/c Code Generated

$S_1 = 4 * i$
$S_2 = c(A)$
$S_3 = S_2[S_1]$

$S_5 = c(B)$
$S_6 = S_5[S_4]$
$S_7 = S_3 * S_6$

$prod = prod + S_7$

$i = i + 1$
If i <= 20 goto (1)

Rearranging order of the code

- Consider following basic block

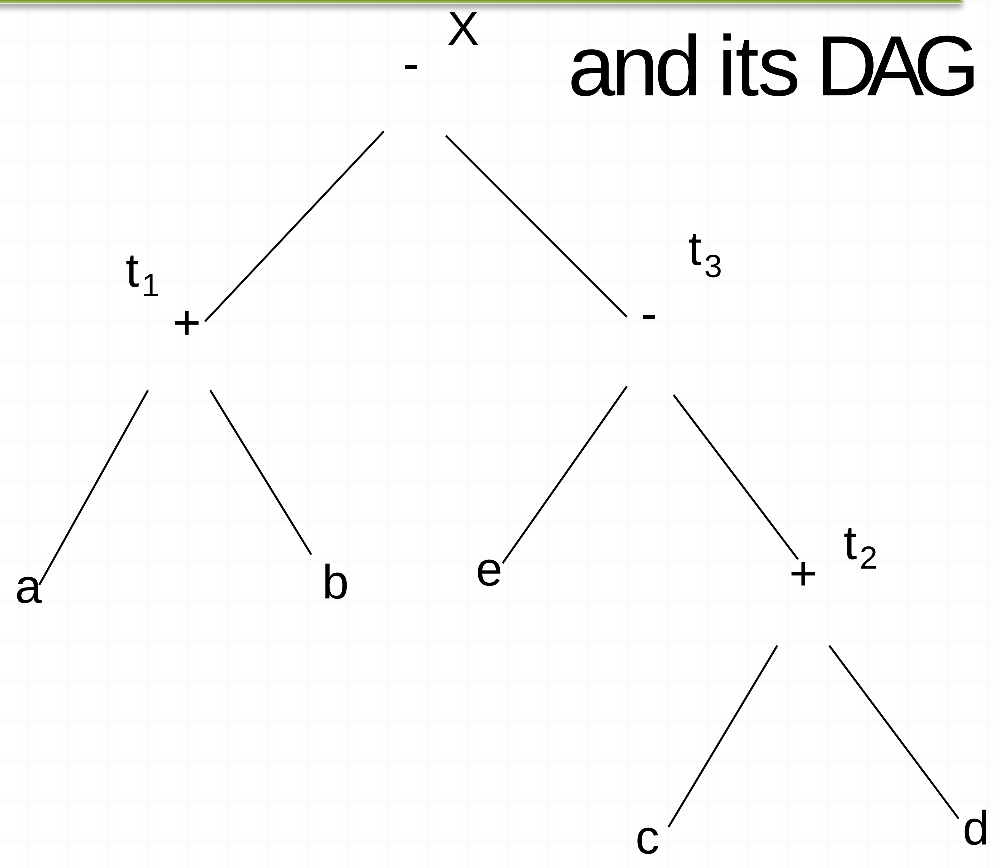
  $t_1 = a + b$

  $t_2 = c + d$

  $t_3 = e - t_2$

  $X = t_1 - t_3$

and its DAG



Rearranging order …

$t_1 = a + b$
$t_2 = c + d$
$t_3 = e - t_2$
$X = t_1 - t_3$

```
MOV  a, R0
ADD  b, R0
MOV  c, R1
ADD  d, R1
MOV R0, t1
MOV  e, R0
SUB  R1, R0
MOV  t1, R1
SUB  R0, R1
MOV R1, X
```

So. We need to decide the order

Register spilling

Register reloading

Rearranging the code as

$t_2 = c + d$
$t_3 = e - t_2$
$t_1 = a + b$
$X = t_1 - t_3$

gives

```
MOV  c, R0
ADD  d, R0
MOV  e, R1
SUB R0, R1
MOV  a, R0
ADD  b, R0
SUB R1, R0
MOV R1, X
```

Ordering of Trees

Ordering can be decided using

- **A) Heuristic Ordering**
- **B) Optimal Ordering(Labelling)**

Heuristic Ordering of Trees

- Heuristics attempts to order the nodes of a DAG so that, if possible, a node immediately follows the evaluation of its left-mostoperand.

Heuristic Ordering of Trees

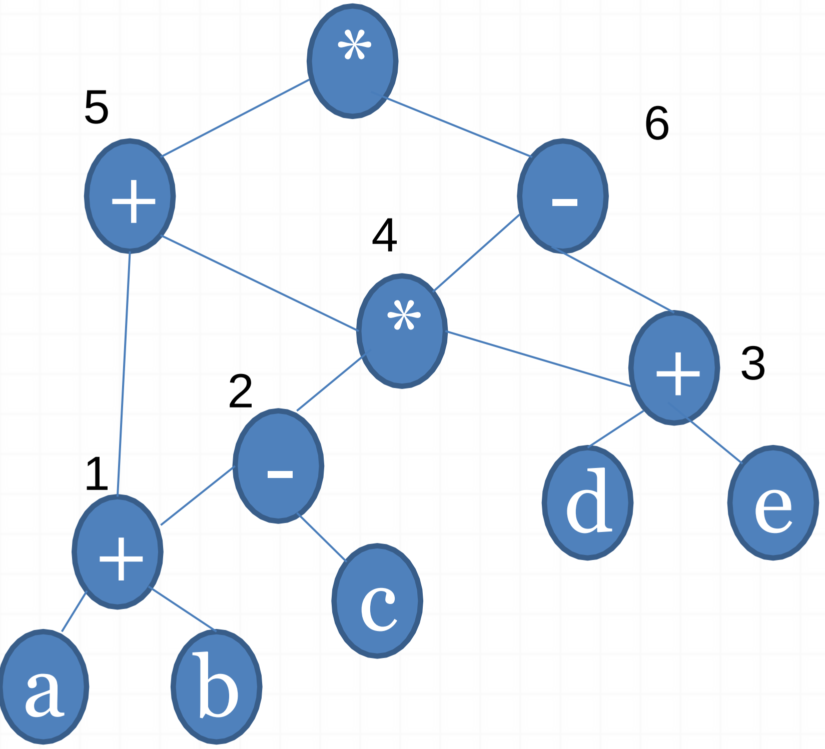
- The algorithm for heuristic ordering is given below. It lists the nodes of a DAG such that the node's reverse listing results in the computation order.

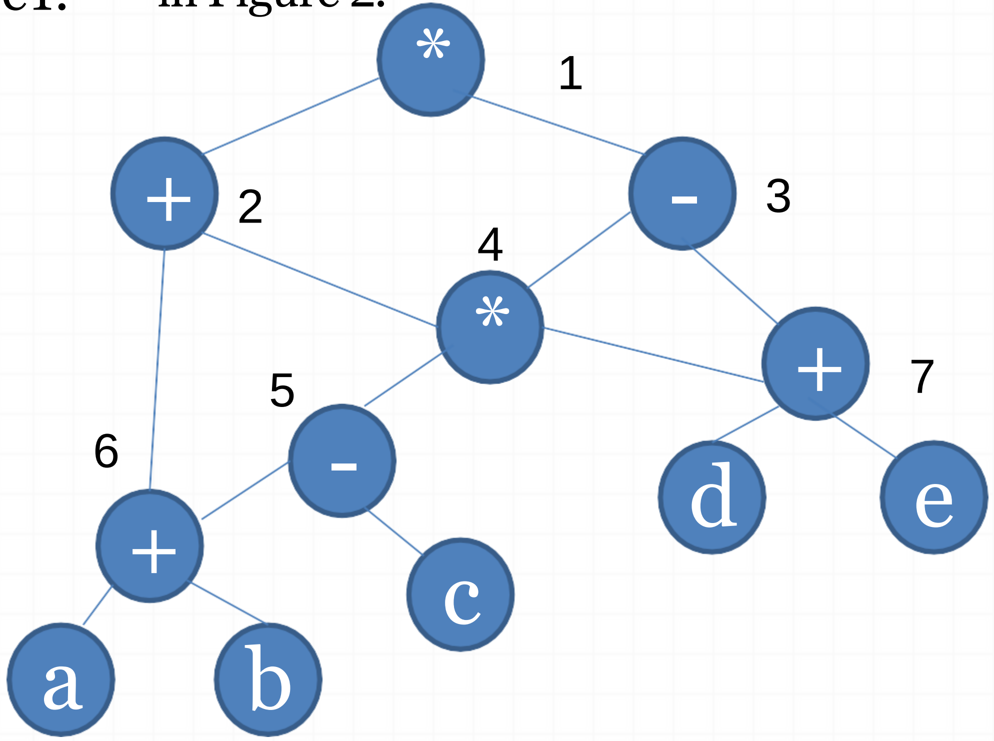
```
While there exists an unlisted interior node do
    {
      Select an unlisted node n whose parents have been listed
                  list n
    while there exists a left-most child m of n that has no
    unlisted parents and m is not a leaf do
    {
          list m
          n = m
    }
 }
order = reverse of the order of listing of nodes
 }
```

Heuristic Ordering of Trees

- Consider the DAG shown in Figure 1.

The order in which the nodes are listed by the heuristic ordering is shown in Figure 2.





Order : 7654321

Heuristic Ordering of Trees

The previous order :

```
t1 = a+ b
t2 = t1- c
t3= d + e
t4 = t2 * t3
t5 = t1 + t4
t6 = t4 – t3
t7 = t5 * t6
```

Final Order decided is :

```
t7 = d+ e
t6 = a+ b
t5= t6- c
t4 =t5* t8
t7 = t4- t8
t2 = t6 + t4
t1 = t2 * t3
```

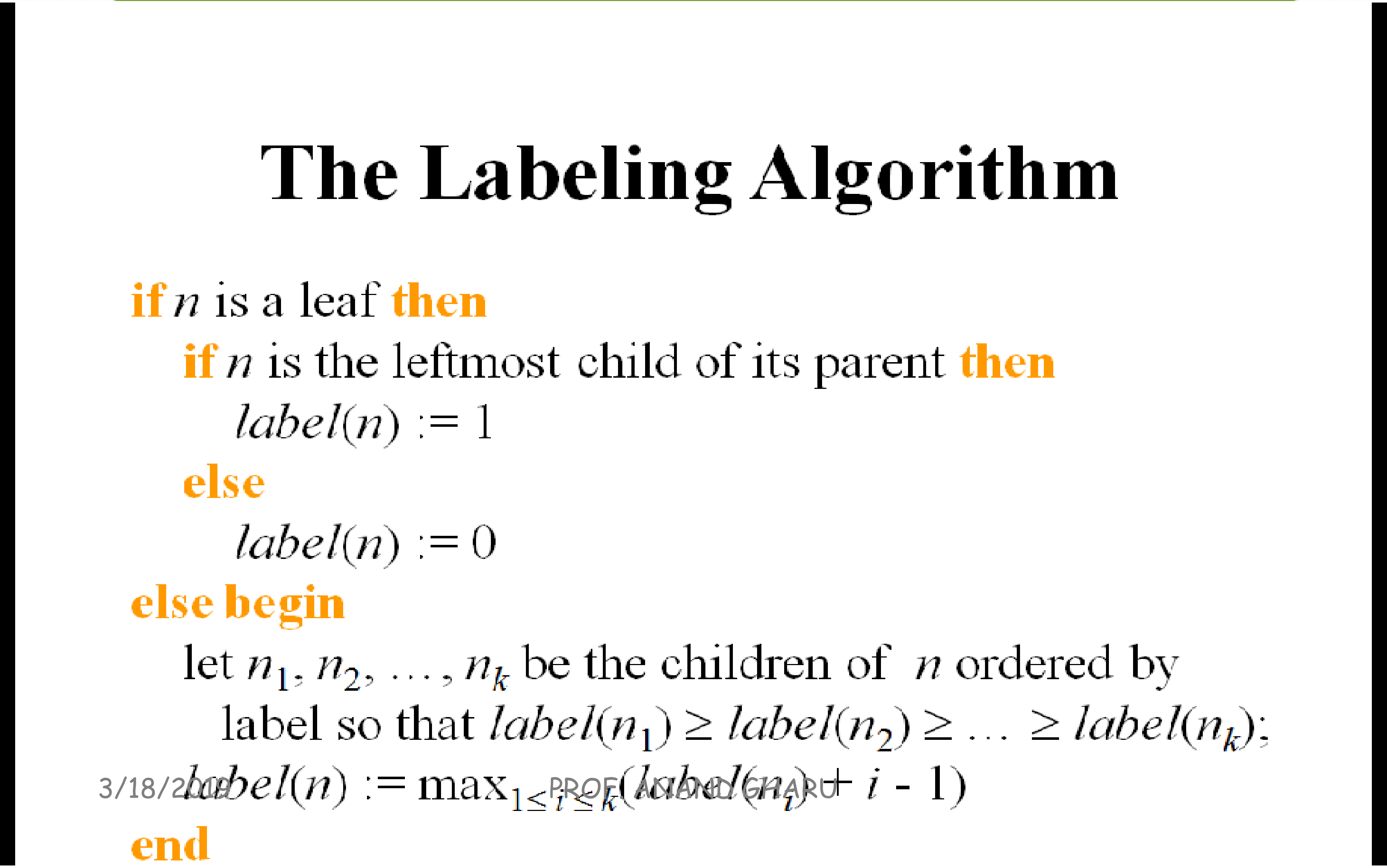
Optimal Ordering of Trees : Labelling Algorithm

•Optimal ordering means yielding the order that has shortest instruction sequence

Optimal Ordering of Trees : Labelling Algorithm

- Optimal ordering means yielding the order that has shortest instruction sequence
- Label each node of the tree bottom up with an integer denoting fewest number of registers required to evaluate the tree with no stores of immediate results.
- Generate code during tree traversal by first evaluating the operand requiring more registers

Optimal Ordering of Trees : Labeling Algorithm



The Labeling Algorithm

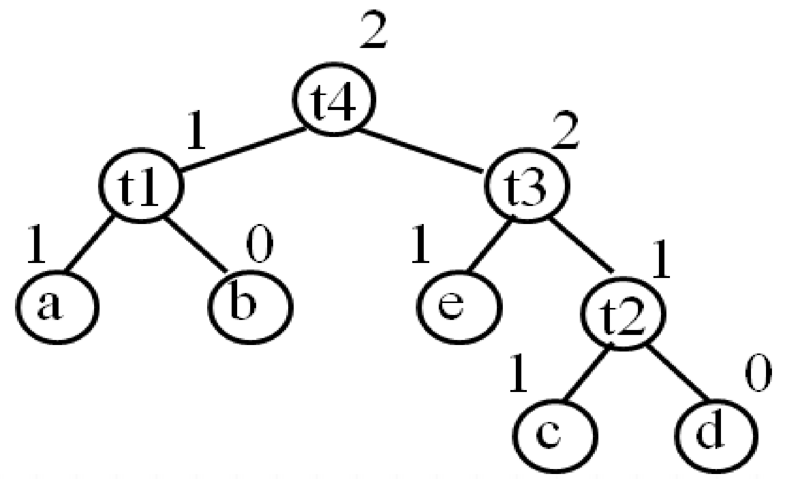
```
if $n$ is a leaf then
    if $n$ is the leftmost child of its parent then
        $label(n) := 1$
    else
        $label(n) := 0$
else begin
    let $n_1, n_2, \ldots, n_k$ be the children of $n$ ordered by
      label so that $label(n_1) \geq label(n_2) \geq \ldots \geq label(n_k)$;
    $label(n) := \max_{1 \leq i \leq k}(label(n_i) + i - 1)$
end
```

Optimal Ordering of Trees : Labeling Algorithm

An Example

For binary interior nodes:

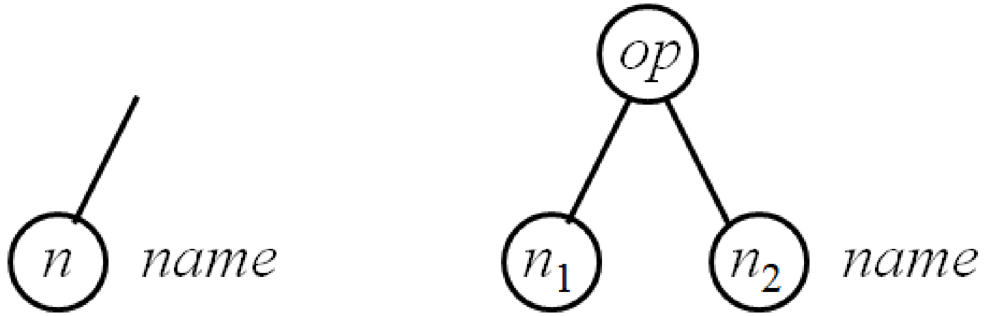
$$label(n) = \begin{cases} \max(l1, l2), & \text{if } l1 \neq l2 \\ l1 + 1, & \text{if } l1 = l2 \end{cases}$$

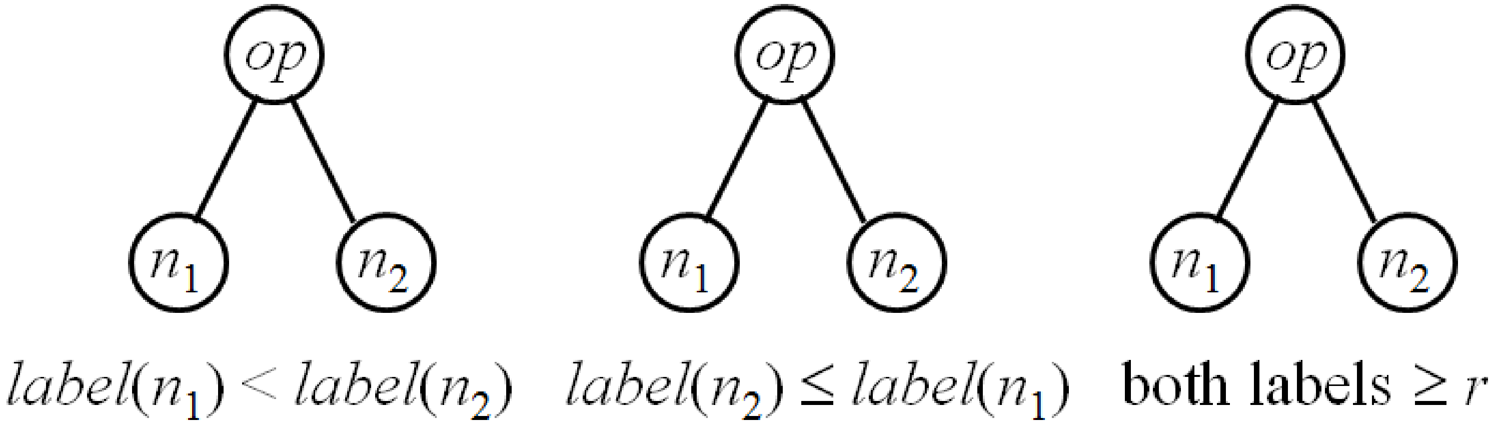


Code Generation From a Labeled Tree

- Use a stack *rstack* to allocate *registers* R0, R1, …, R(*r*-1)
- The value of a tree is always computed in the top register on *rstack*
- The function *swap*(*rstack*) interchanges the top two registers on *rstack*
- Use a stack *tstack* to allocate *temporary memory locations* T0, T1, …

Cases Analysis





The Function *gencode*

```
procedure gencode(n);
begin
  if n is a left leaf representing operand name
          and n is the leftmost child of its parent then
    print 'MOV' || name || ',' || top(rstack)
  else if n is an interior node with operator op,
          left child $n_1$, and right child $n_2$ then
    if label($n_2$) = 0 then /* case 1 */
    else if $1 \leq label(n_1) < label(n_2)$ and $label(n_1) < r$ then /* case 2 */
    else if $1 \leq label(n_2) \leq label(n_1)$ and $label(n_2) < r$ then /* case 3 */
    else /* case 4, both labels $\geq r$ */
end
```

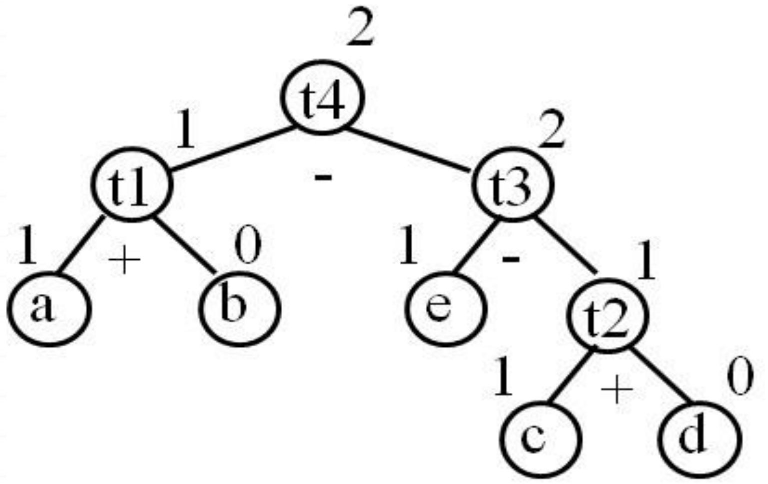
The Function *gencode*

```
/* case 1 */
begin
    let name be the operand represented by $n_2$;
    gencode($n_1$);
    print op || name || ',' || top(rstack)
end

/* case 2 */
begin
    swap(rstack);   gencode($n_2$);
    R := pop(rstack);   gencode($n_1$);
    print op || R || ',' || top(rstack);
    push(rstack, R);   swap(rstack);
end
```

The Function *gencode*

```
/* case 3 */
begin
    gencode($n_1$);
    R := pop(rstack);   gencode($n_2$);
    print op || R || ',' || top(rstack);
    push(rstack, R);
end

/* case 4 */
begin
    gencode($n_2$);   T := pop(tstack);
    print 'MOV' || top(rstack) || ',' || T;
    gencode($n_1$);   push(tstack, T);
    print op || T || ',' || top(rstack);
end
```

An Example



```
gencode(t4)  [R1, R0]      /* 2 */
 gencode(t3)  [R0, R1]     /* 3 */
  gencode(e)  [R0, R1]     /* 0 */
   print MOV e, R1
  gencode(t2)  [R0]        /* 1 */
   gencode(c)  [R0]        /* 0 */
    print MOV c, R0
   print ADD d, R0
  print SUB R0, R1
 gencode(t1)  [R0]         /* 1 */
  gencode(a)  [R0]         /* 0 */
   print MOV a, R0
```

Multiregister Operations

- Some operations like multiplication, division, or a function call normally require more than one register
- The labeling algorithm needs to ensure that *label*(*n*) is always at least the number of registers required by the operation

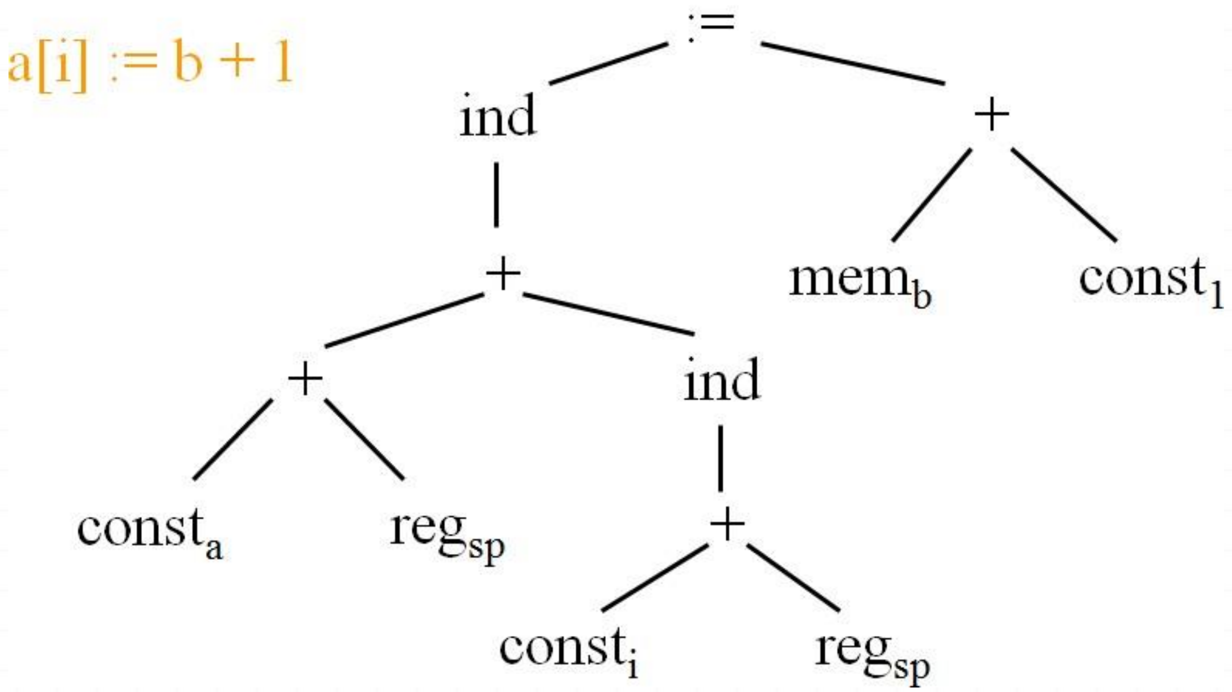
$$label(n) = \begin{cases} \max(2, l1, l2), & \text{if } l1 \neq l2 \\ l1 + 1, & \text{if } l1 = l2 \end{cases}$$

Code Generator Generators

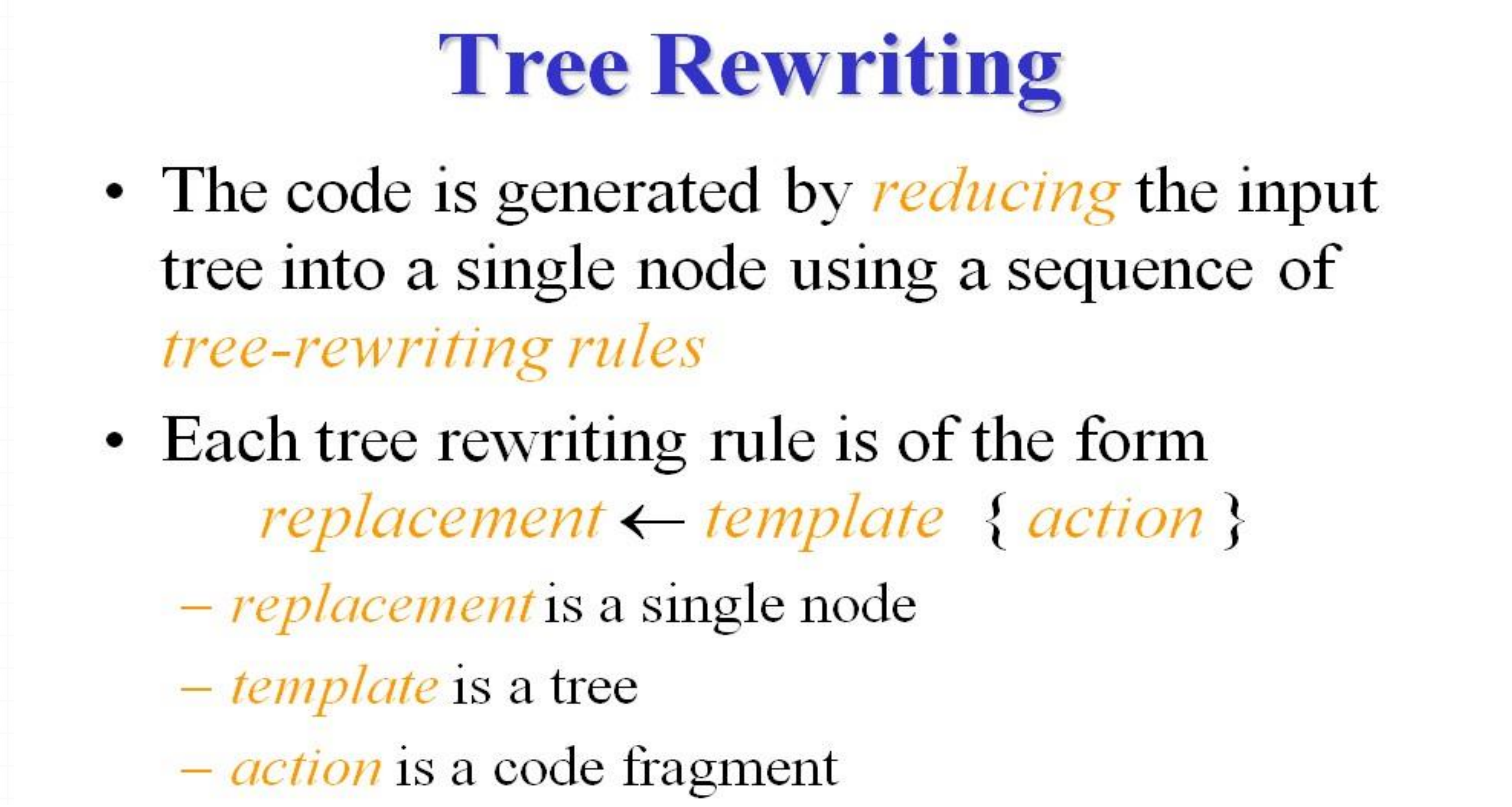
- A tool to automatically construct the *instruction selection* phrase of a code generator
- Such tools may use *tree grammars* or *context free grammars* to describe the *target machines*
- *Register allocation* will be implemented as a separate mechanism
- *Graph coloring* is one of the approaches for register allocation

Code Generator Generators

Tree Rewriting



Code Generator Generators



Tree Rewriting

- The code is generated by *reducing* the input tree into a single node using a sequence of *tree-rewriting rules*
- Each tree rewriting rule is of the form

  replacement ← *template* { *action* }

  – *replacement* is a single node

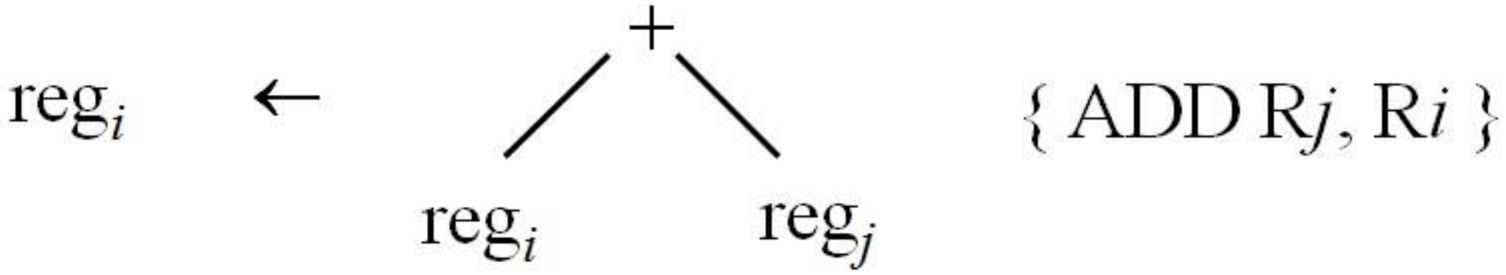
  – *template* is a tree

  – *action* is a code fragment

- A set of tree-rewriting rules is called a *tree-translation scheme*

ence : Aho Ullman, Sethi

An Example



Each tree template represents a computation performed by the sequence of machines instructions emitted by the associated action

Tree Rewriting Rules

(1) $reg_i \leftarrow const_c$ { MOV #c, R*i* }

(2) $reg_i \leftarrow mem_a$ { MOV a, R*i* }

(3) $mem \leftarrow$

$$\begin{tikzcd} & := \arrow[dl, dash] \arrow[dr, dash] & \\ mem_a & & reg_i \end{tikzcd}$$

{ MOV R*i*, a }

(4) $mem \leftarrow$

$$\begin{tikzcd} & := \arrow[dl, dash] \arrow[dr, dash] & \\ ind \arrow[d, dash] & & reg_j \\ reg_i & & \end{tikzcd}$$

{ MOV R*j*, *R*i* }

(5) $reg_i \leftarrow$

$$\begin{tikzcd} & ind \arrow[d, dash] & \\ & + \arrow[dl, dash] \arrow[dr, dash] & \\ const_c & & reg_j \end{tikzcd}$$

{ MOV c(R*j*), R*i* }

Tree Rewriting Rules

(6) $\text{reg}_i \leftarrow$

$$\begin{tikzcd}
 & + \arrow[dl, dash] \arrow[dr, dash] & & \\
\text{reg}_i & & \text{ind} \arrow[d, dash] & \\
 & & + \arrow[dl, dash] \arrow[dr, dash] & \\
 & \text{const}_c & & \text{reg}_j
\end{tikzcd}$$

{ ADD c(R*j*), R*i* }

(7) $\text{reg}_i \leftarrow$

$$\begin{tikzcd}
 & + \arrow[dl, dash] \arrow[dr, dash] & \\
\text{reg}_i & & \text{reg}_j
\end{tikzcd}$$

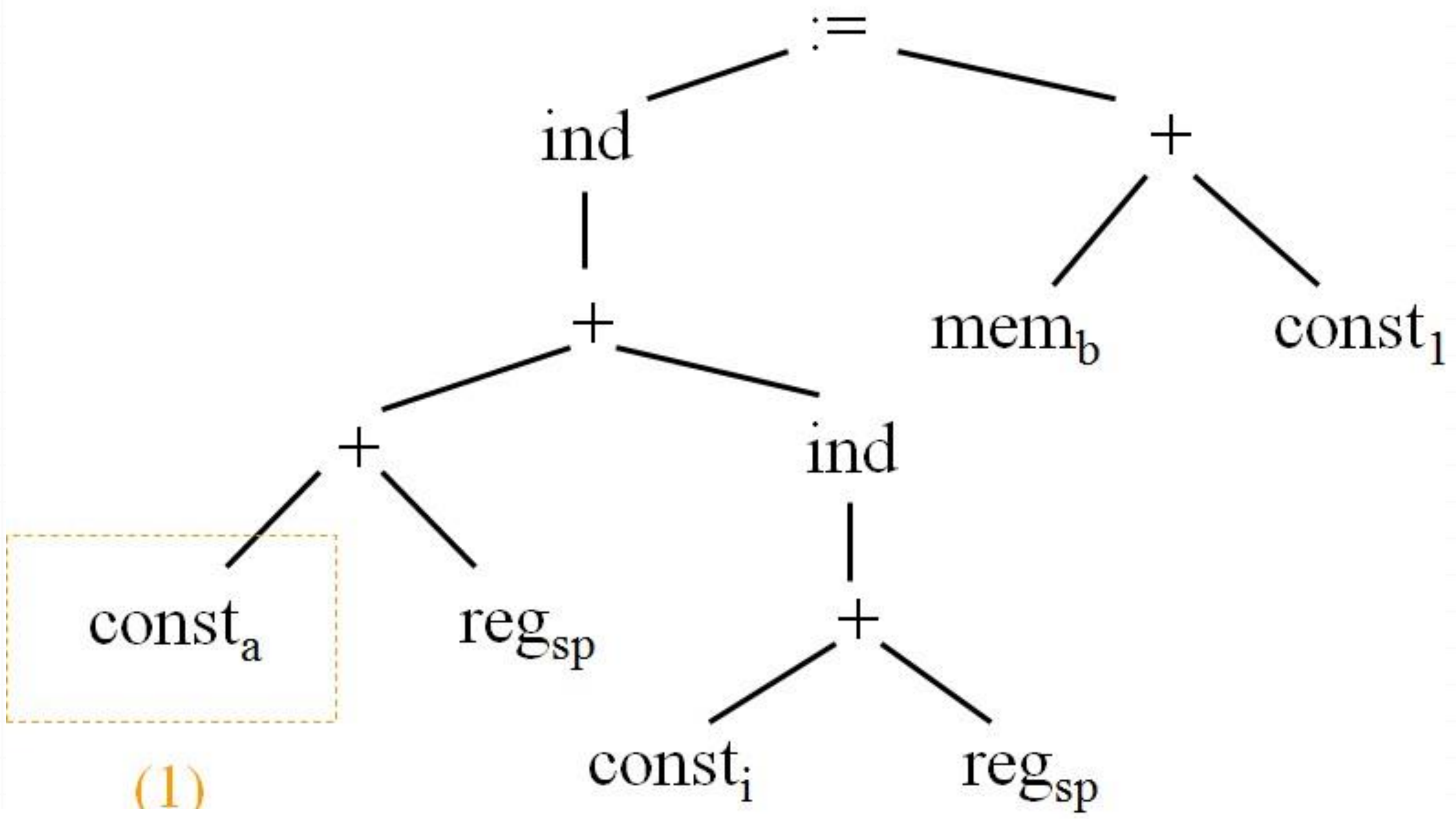
{ ADD R*j*, R*i* }

(8) $\text{reg}_i \leftarrow$

$$\begin{tikzcd}
 & + \arrow[dl, dash] \arrow[dr, dash] & \\
\text{reg}_i & & \text{const}_1
\end{tikzcd}$$

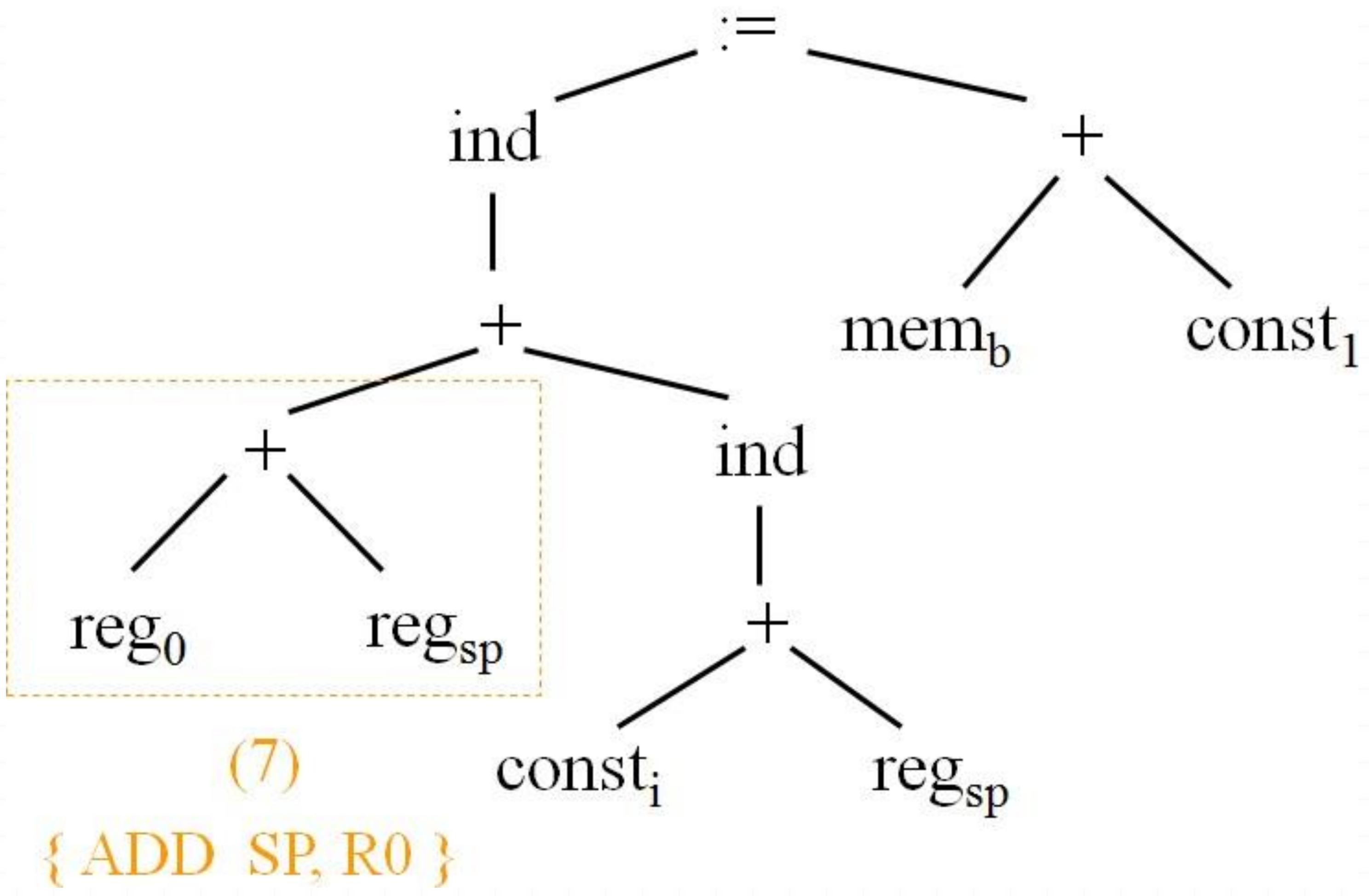
{ INC R*i* }

An Example

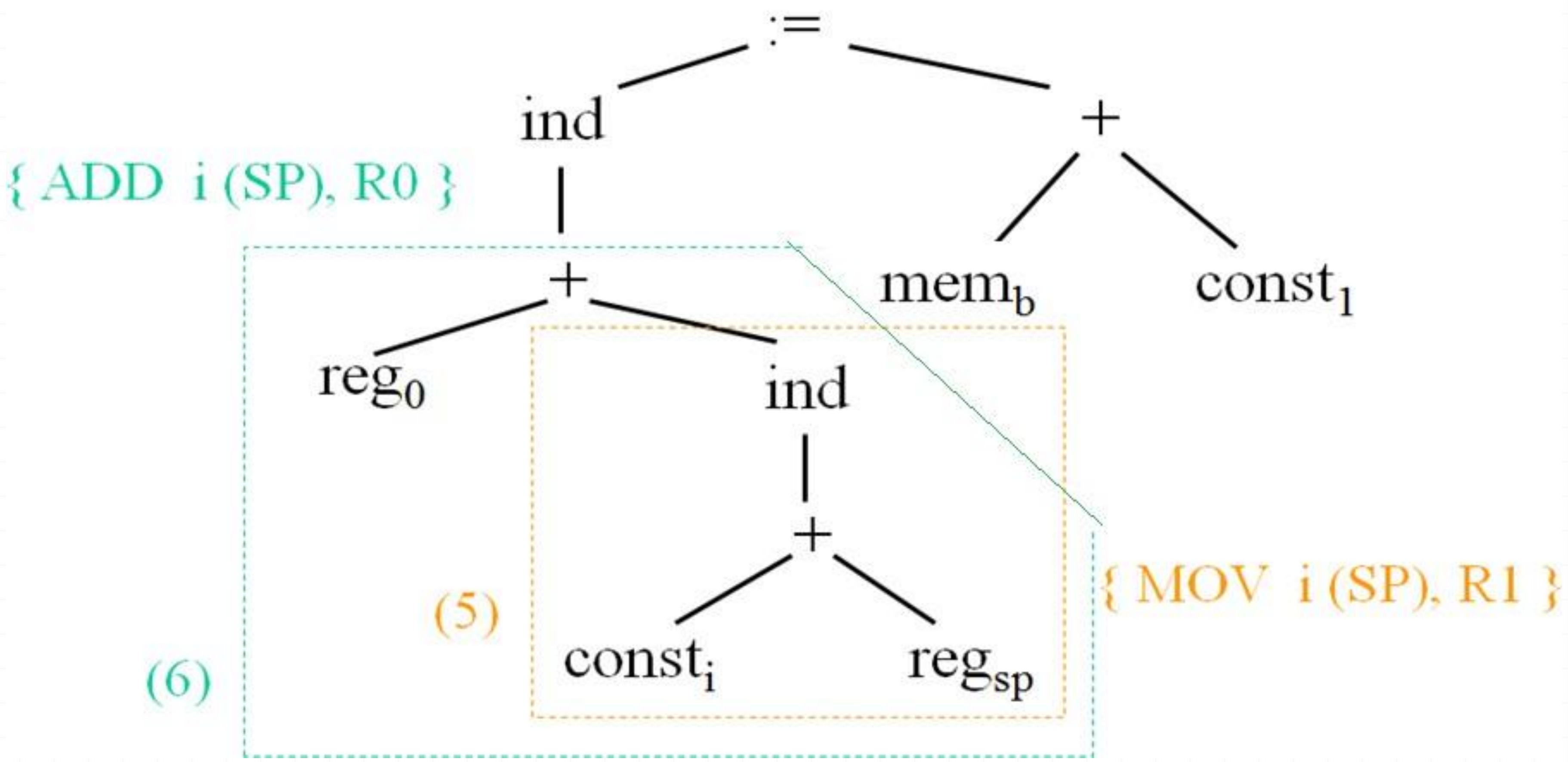


{ MOV #a, R0 }

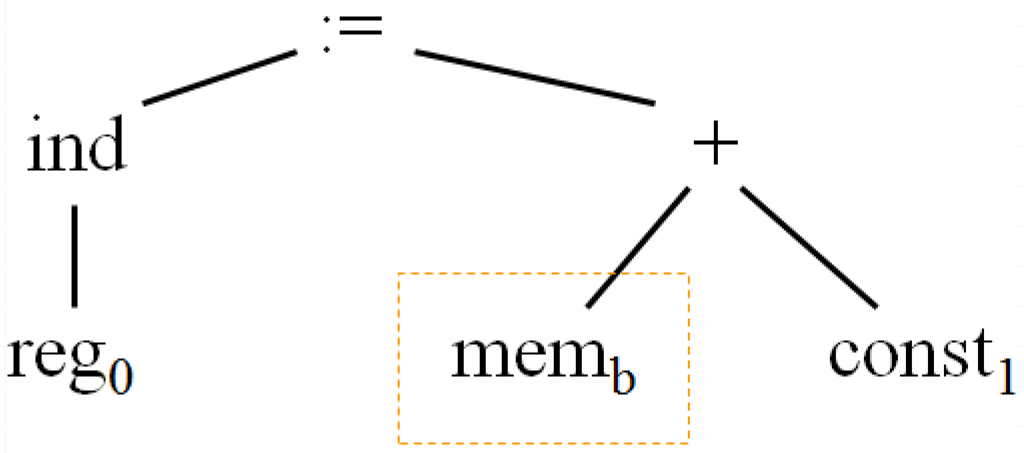
An Example



An Example


:=
ind
+
{ ADD i (SP), R0 }
+
mem_b
const_1
reg_0
ind
+
(5)
(6)
const_i
reg_{sp}
{ MOV i (SP), R1 }

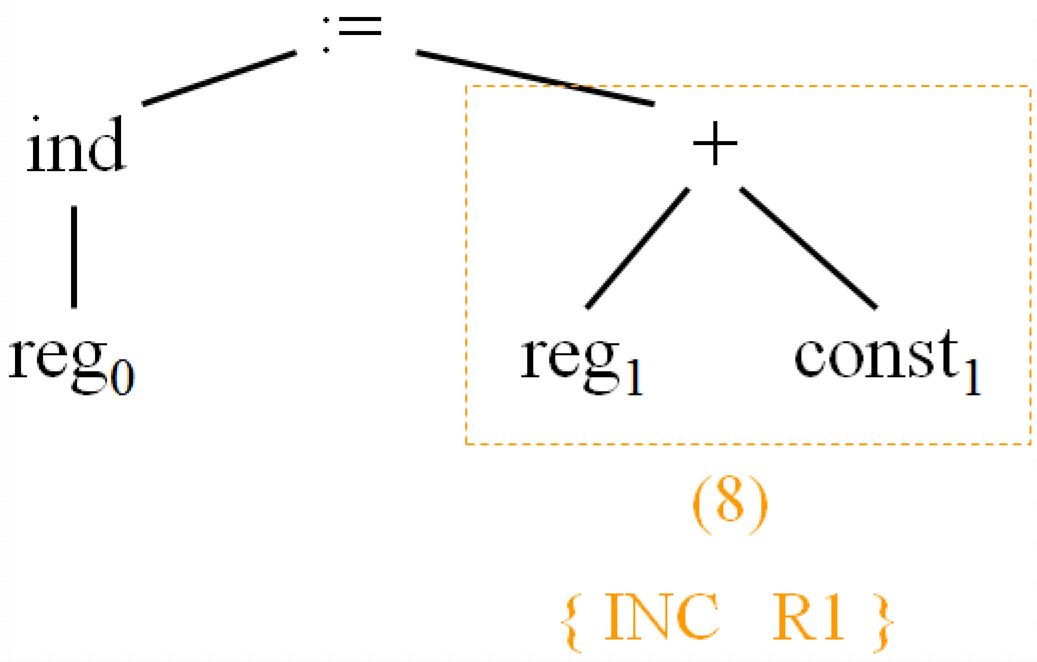
An Example



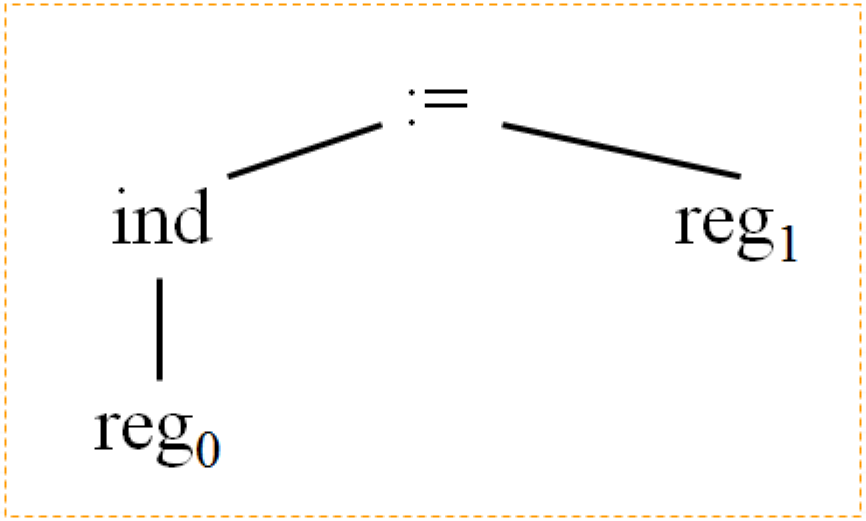
(2)

{ MOV b, R1 }

An Example



An Example



(4)

{ MOV R1, *R0 }

CODE GENERATOR GENERATOR: FINAL CODE GENERATED

```
MOV#a,R0
ADD Sp, R0
ADD i(SP), R0
MOV b, R1
INC R1
MOV R1, *R0
```

Not in syllabus…but may ask

Dynamic Programming Code Generation

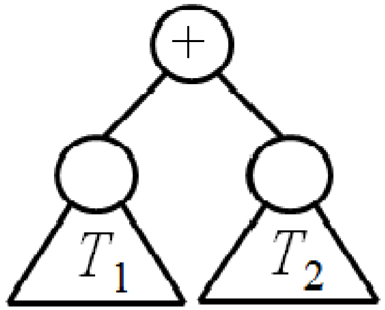
- The *dynamic programming* algorithm applies to a *broad* class of *register* machines with *complex* instruction sets
- Machines has *r* interchangeable registers
- Machines has instructions of the form

  $R_i = E$

  where E is any expression containing operators, registers, and memory locations. If E involves registers, then R_i must be one of them

Dynamic Programming

- The *dynamic programming* algorithm partitions the problem of generating optimal code for an expression into sub-problems of generating optimal code for the sub-expressions of the given expression



Contiguous Evaluation

- We say a program *P* evaluates a tree *T* *contiguously* if
- it first evaluates those subtrees of *T* that need to be computed into *memory*
- it then evaluates the subtrees of the root in *either* order
- it finally evaluates the root

Optimally Contiguous Program

- For the machines defined above, given any program *P* to evaluate an expression tree *T*, we can find an *equivalent* program *P*' such that
  - P' is of no higher cost than P
  - P' uses no more registers than P
  - P' evaluates the tree in a contiguous fashion
- This implies that every expression tree can be evaluated *optimally* by a *contiguous* program

Dynamic Programming Algorithm

- *Phase 1*: compute bottom-up for each node n of the expression tree T an array C of costs, in which the ith component $C[i]$ is the optimal cost of computing the subtree S rooted at n into *a register*, assuming i registers are available for the computation. $C[0]$ is the optimal cost of computing the subtree S into *memory*

Dynamic Programming Algorithm

- To compute $C[i]$ at node n, consider each machine instruction $R := E$ whose expression E *matches* the subexpression rooted at node n
- Determine the costs of evaluating the *operands* of E by examining the cost vectors at the corresponding descendants of n

Dynamic Programming Algorithm

- For those operands of *E* that are registers, consider all possible orders in which the corresponding subtrees of *T* can be evaluated into registers
- In each ordering, the first subtree corresponding to a register operand can be evaluated using *i* available registers, the second using *i*-1 registers, and so on

Dynamic Programming Algorithm

- For node n, add in the cost of the instruction $R := E$ that was used to match node n
- The value $C[i]$ is then the minimum cost over all possible orders
- At each node, store the instruction used to achieve the best cost for $C[i]$ for each i
- The smallest cost in the vector gives the minimum cost of evaluating T

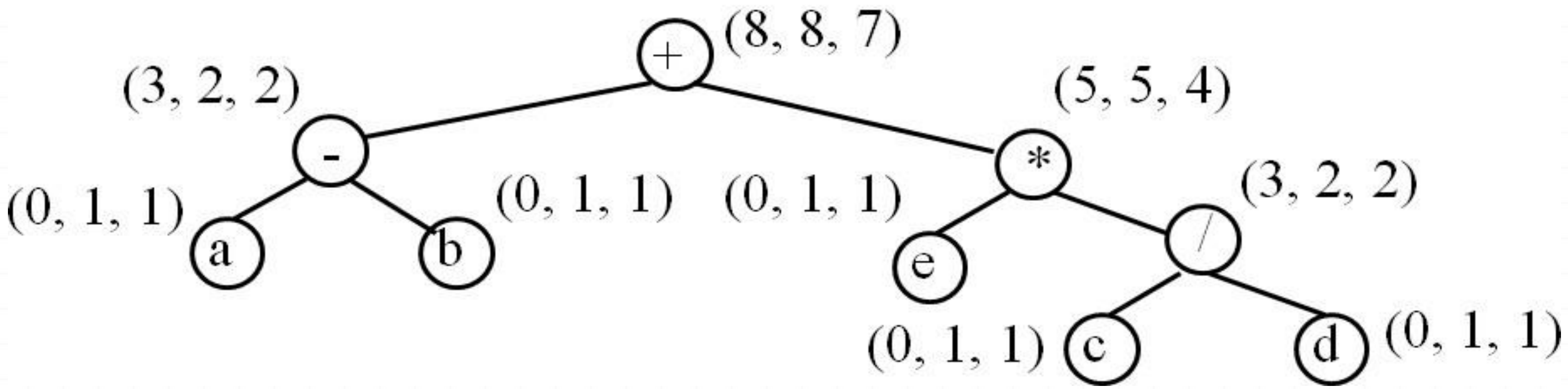
Dynamic Programming Algorithm

- *Phase 2*: traverse *T* and use the cost vectors to determine which subtrees of *T* must be computed into memory
- *Phase 3*: traverse *T* and use the cost vectors and associated instructions to generate the final target code

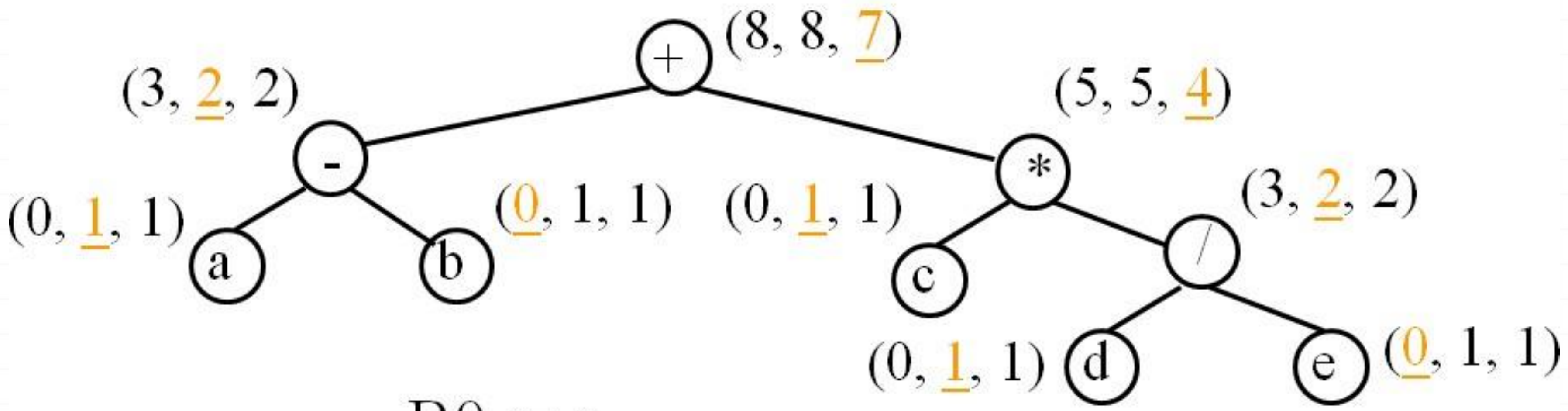
An Example

Consider a machine with two registers R0 and R1 and instructions

Ri := Mj	Mi := Ri	Ri := Rj
Ri := Ri *op* Rj	Ri := Ri *op* Mj	



An Example



R0 := c

R1 := d

R1 := R1 / e

R0 := R0 * R1

R1 := a

R1 := R1 - b

R1 := R1 + R0

0**THANK YOU!!!!!!!!!!!**

MY BLOG :

anandgharu.wordpress.comg : anandgharu.wordpress.com