Compiler course

Chapter 8

Code Generation

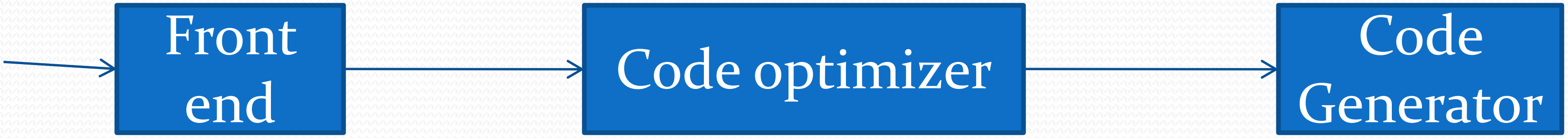
By Varun Arora

Outline

- Code Generation Issues
- Target language Issues
- Addresses in Target Code
- 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
  - Insrtuction ordering



Issues in the Design of Code Generator

- The most important criterion is that it produces correct code
- Input to the code generator
  - IR + Symbol table
  - We assume front end produces low-level IR, i.e. values of names in it can be directly manipulated by the machine instructions.
  - Syntactic and semantic errors have been already detected
- The target program
  - Common target architectures are: RISC, CISC and Stack based machines
  - In this chapter we use a very simple RISC-like computer with addition of some CISC-like addressing modes

complexity of mapping

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

x=y+z

```
LD      R0, y
ADD     R0, R0, z
ST      x, R0
```

a=b+c
d=a+e

```
LD      R0, b
ADD     R0, R0, c
ST      a, R0
LD      R0, a
ADD     R0, R0, e
ST      d, R0
```

Register allocation

- Two subproblems
  - Register allocation: selecting the set of variables that will reside in registers at each point in the program
  - Resister assignment: selecting specific register that a variable reside in
- Complications imposed by the hardware architecture
  - Example: register pairs for multiplication and division

```
t=a+b
t=t*c
T=t/d

L       R1, a
A       R1, b
M       R0, c
D       R0, d
ST      R1, t
```

```
t=a+b
t=t+c
T=t/d

L       R0, a
A       R0, b
M       R0, c
SRDA    R0, 32
D       R0, d
ST      R1, t
```

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 addressing modes

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

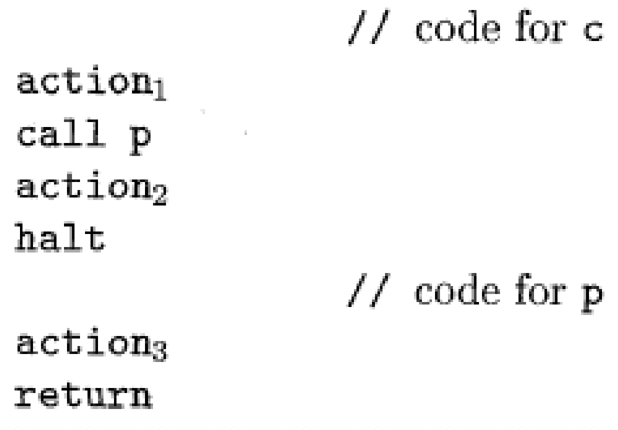
Addresses in the Target Code

- A statically determined area Code
- A statically determined data area Static
- A dynamically managed area Heap
- A dynamically managed area Stack

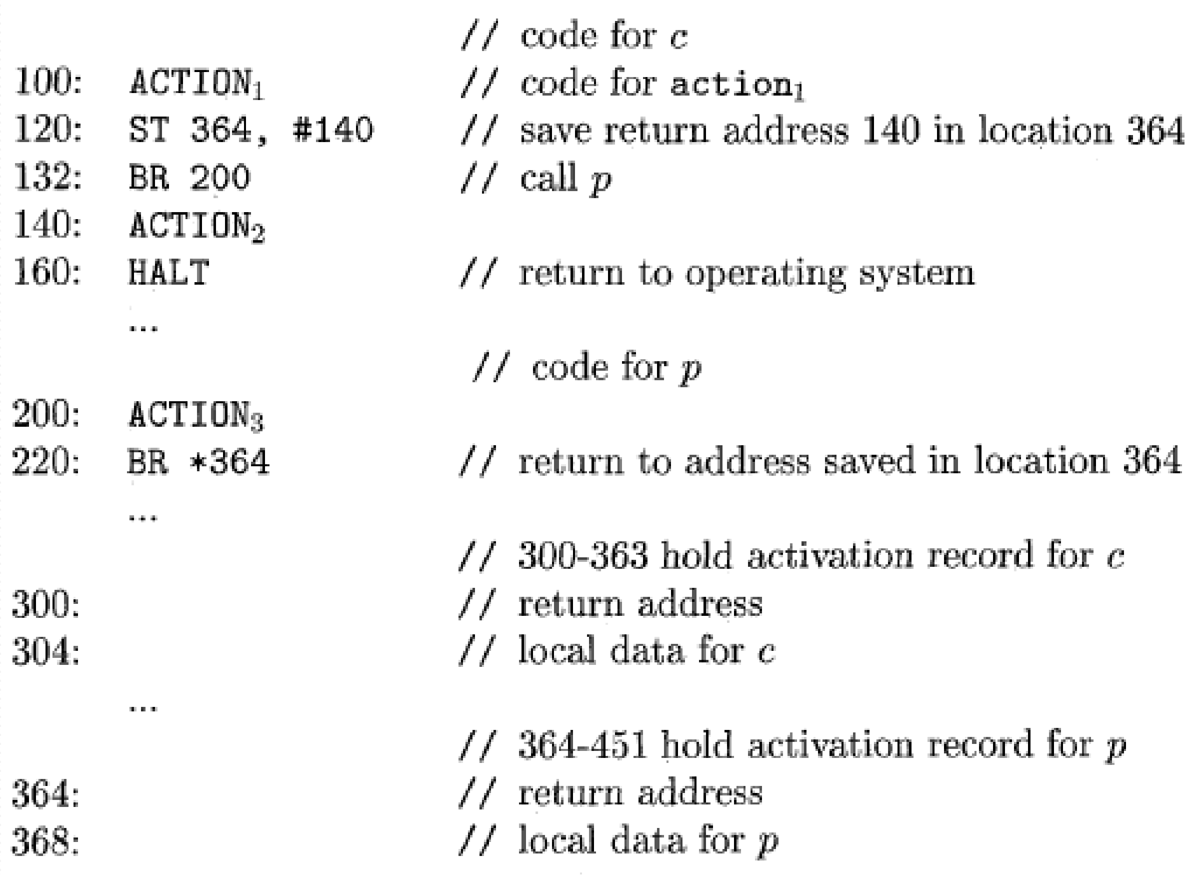
three-address statements for procedure calls and returns

- call callee
- Return
- Halt
- action

Target program for a sample call and return

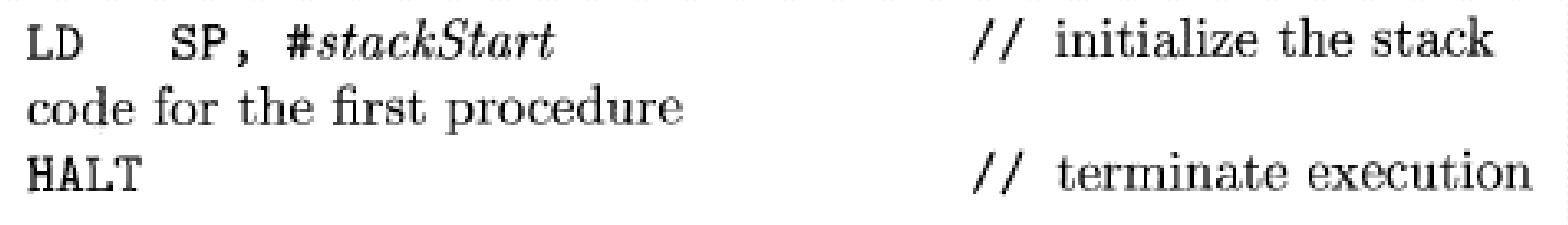


```
                      // code for c
action1
call p
action2
halt
                      // code for p
action3
return
```

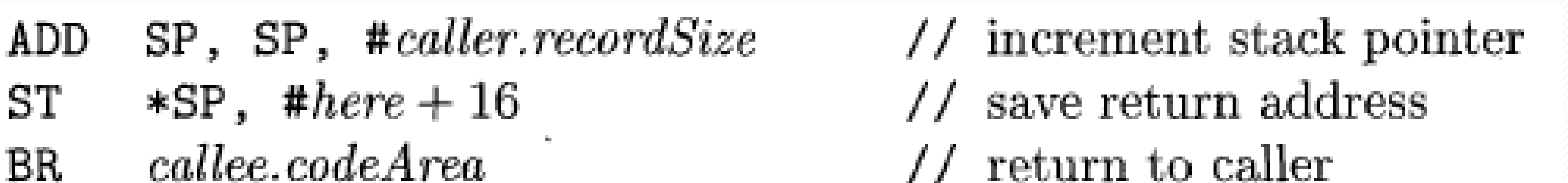


```
                                  // code for c
100:  ACTION1                     // code for action1
120:  ST 364, #140                // save return address 140 in location 364
132:  BR 200                      // call p
140:  ACTION2
160:  HALT                        // return to operating system
      ...
                                   // 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
```

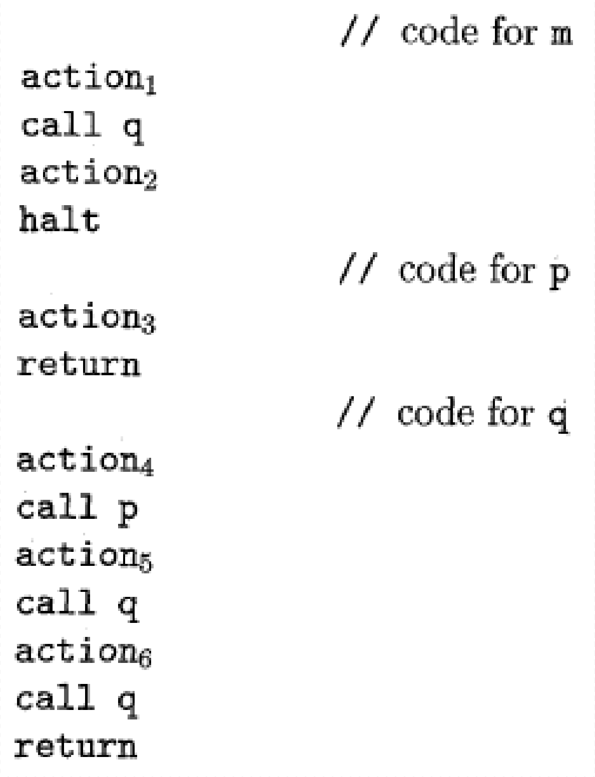
Branch to called procedure

Return to caller

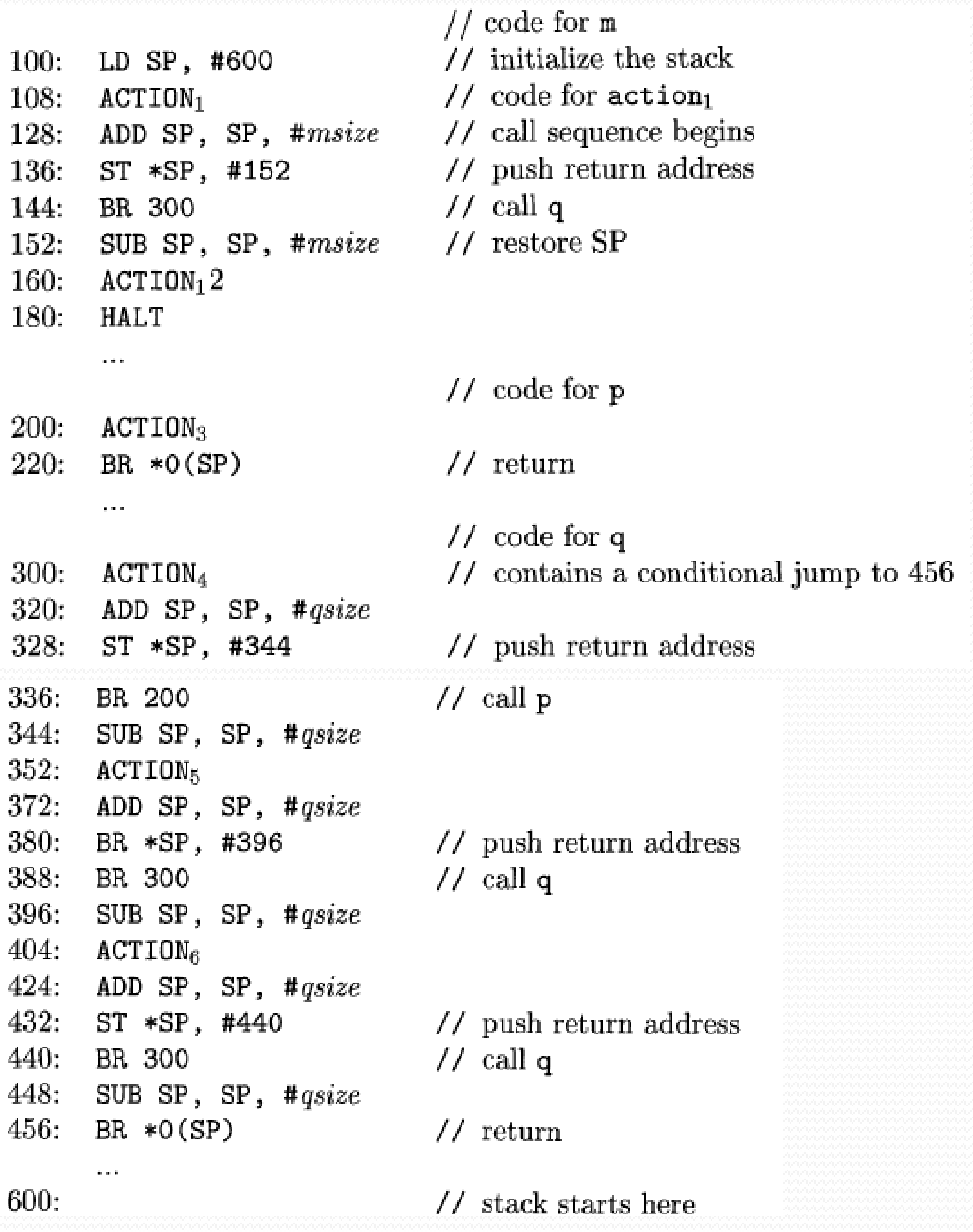
in Callee: BR *0(SP)

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
```

By Varun Arora

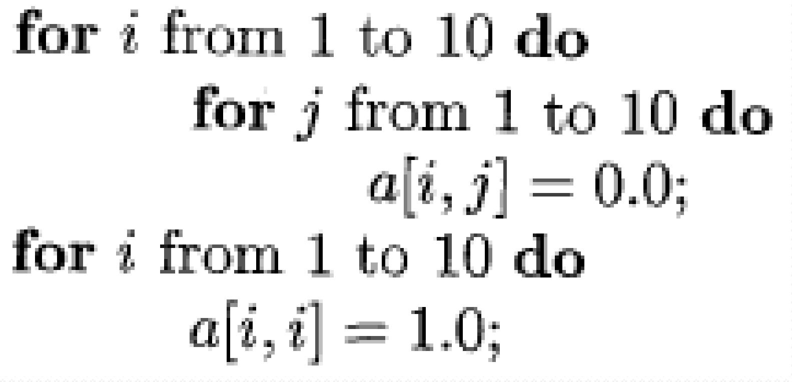
Basic blocks and flow graphs

- Partition the intermediate code into basic blocks
  - The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
  - Control will leave the block without halting or branching, except possibly at the last instruction in the block.
- The basic blocks become the nodes of a flow graph

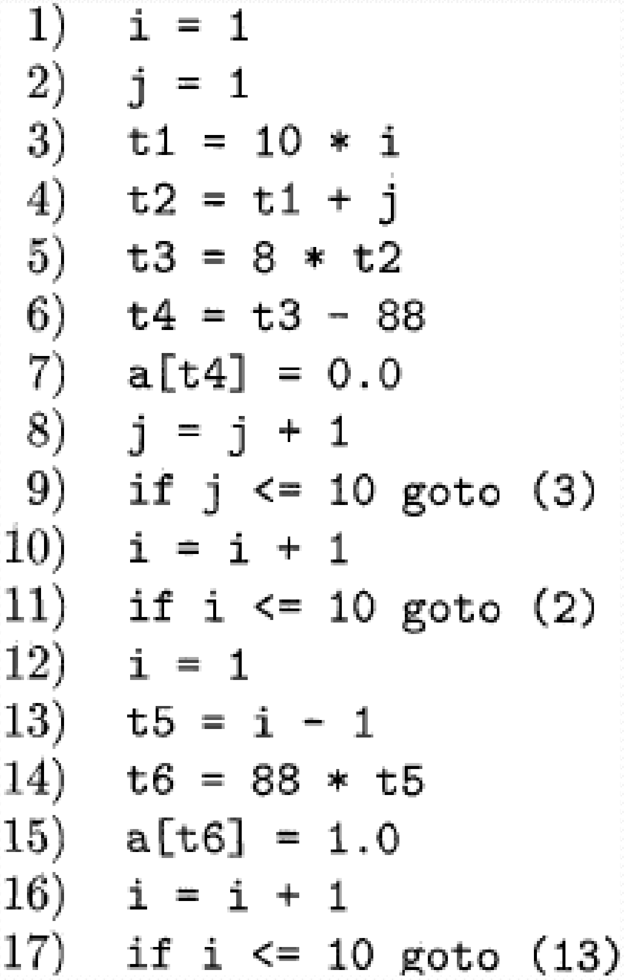
rules for finding leaders

- The first three-address instruction in the intermediate code is a leader.
- Any instruction that is the target of a conditional or unconditional jump is a leader.
- Any instruction that immediately follows a conditional or unconditional jump is a leader.

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

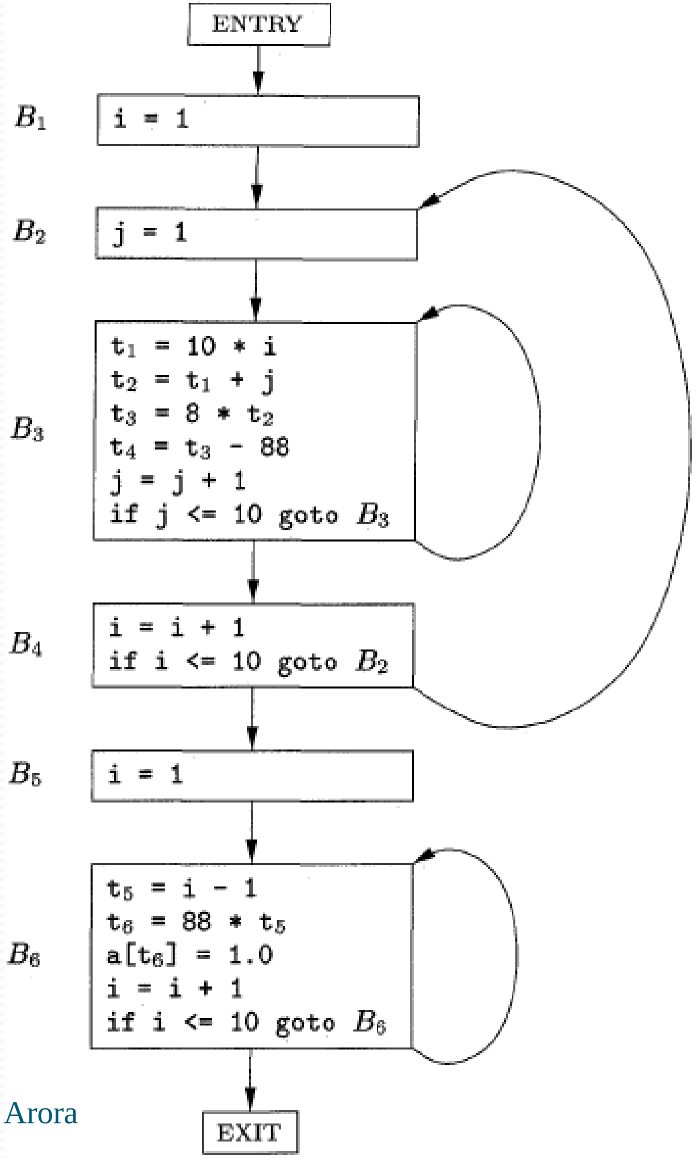


```
for $i$ from 1 to 10 do
    for $j$ from 1 to 10 do
        $a[i, j] = 0.0;$
for $i$ from 1 to 10 do
    $a[i, i] = 1.0;$
```



```
 1)   i = 1
 2)   j = 1
 3)   t1 = 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)
```

Flow graph based on Basic Blocks



ENTRY

B_1

i = 1

B_2

j = 1

B_3

t_1 = 10 * i
t_2 = t_1 + j
t_3 = 8 * t_2
t_4 = t_3 - 88
j = j + 1
if j <= 10 goto B_3

B_4

i = i + 1
if i <= 10 goto B_2

B_5

i = 1

B_6

t_5 = i - 1
t_6 = 88 * t_5
a[t_6] = 1.0
i = i + 1
if i <= 10 goto B_6

EXIT

liveness and next-use information

- We wish to determine for each three address statement x=y+z what the next uses of x, y and z are.
- Algorithm:
  - Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
  - In the symbol table, set x to "not live" and "no next use.“
  - In the symbol table, set y and z to "live" and the next uses of y and z to i.

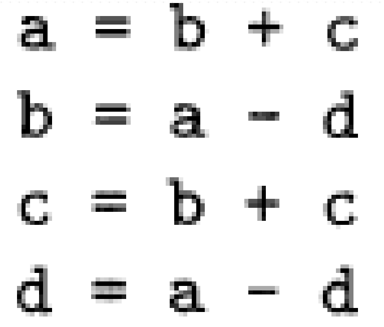
DAG representation of basic blocks

- There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
- There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
- Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
- Certain nodes are designated *output nodes*. These are the nodes whose variables are *live on exit* from the block.

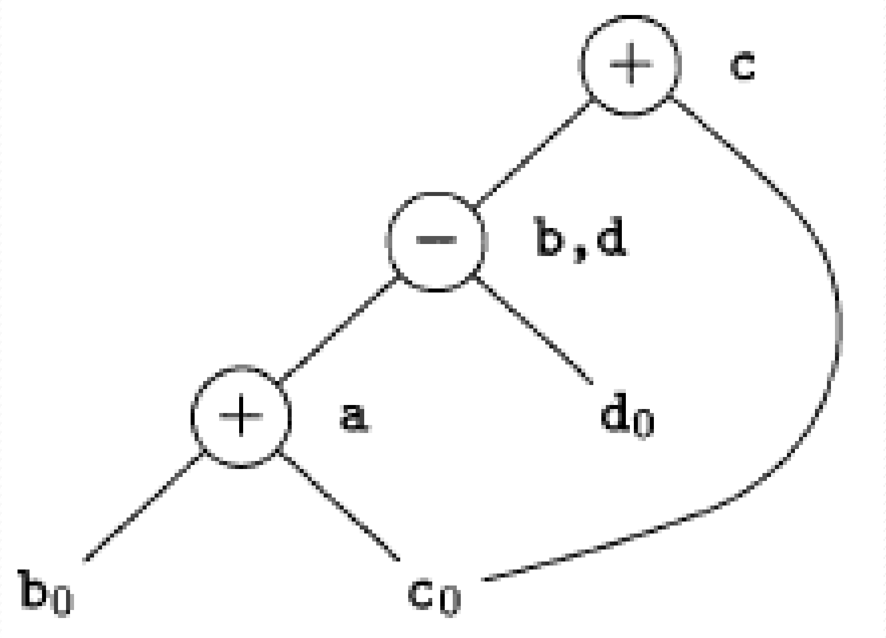
Code improving transformations

- We can eliminate *local common subexpressions,* that is, instructions that compute a value that has already been computed.
- We can eliminate *dead code,* that is, instructions that compute a value that is never used.
- We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
- We can apply algebraic laws to reorder operands of three-address instructions, and sometimes t hereby simplify t he computation.

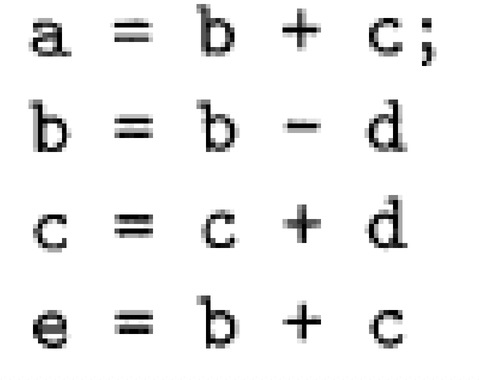
DAG for basic block



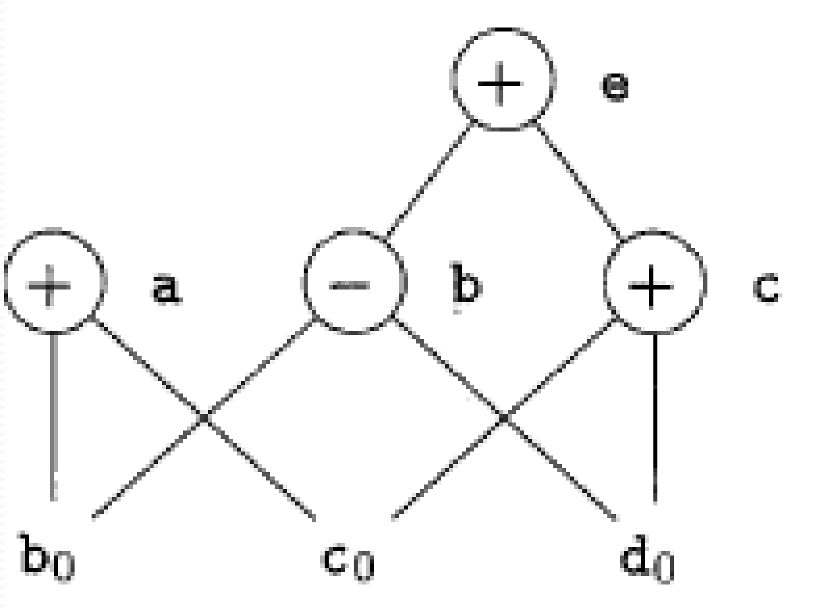
```
a = b + c
b = a - d
c = b + c
d = a - d
```



DAG for basic block



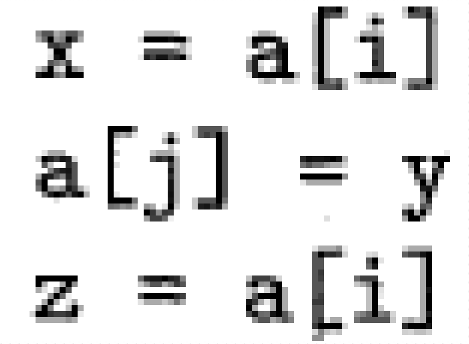
```
a = b + c;
b = b - d
c = c + d
e = b + c
```



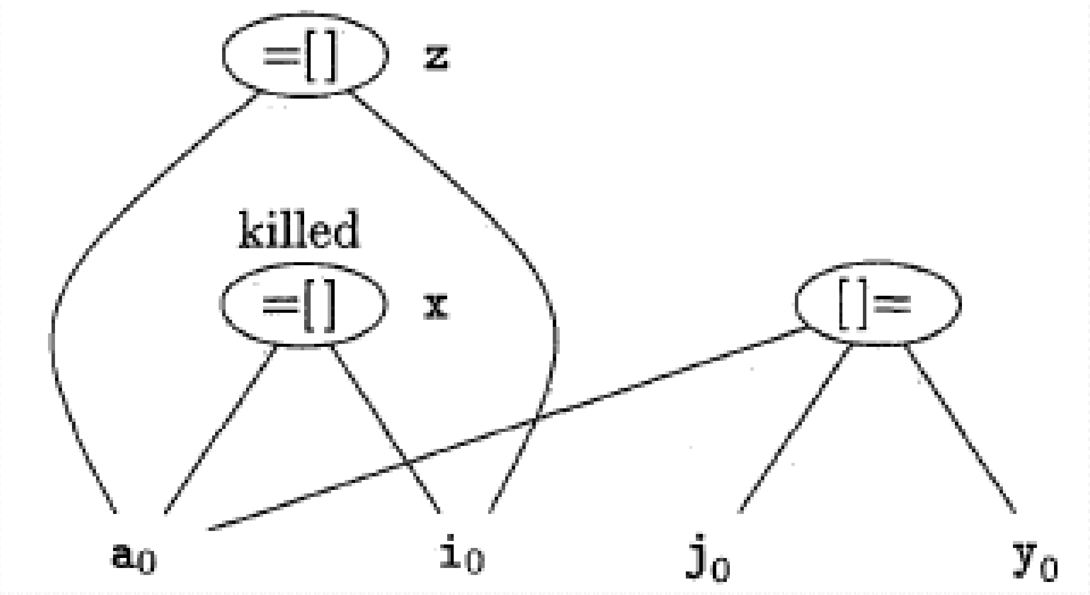
array accesses in a DAG

- An assignment from an array, like x = a [i], is represented by creating a node with operator =[] and two children representing the initial value of the array, ao in this case, and the index i. Variable x becomes a label of this new node.
- An assignment to an array, like a [j] = y, is represented by a new node with operator []= and three children representing ao, j and y. There is no variable labeling this node. What is different is that the creation of this node *kills* all currently constructed nodes whose value depends on ao. **A** node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

DAG for a sequence of array assignments



```
x = a[i]
a[j] = y
z = a[i]
```



Rules for reconstructing the basic block from a DAG

- The order of instructions must respect the order of nodes in the DAG. That is, we cannot compute a node's value until we have computed a value for each of its children.
- Assignments to an array must follow all previous assignments to, or evaluations from, the same array, according to the order of these instructions in the original basic block.
- Evaluations of array elements must follow any previous (according to the original block) assignments to the same array. The only permutation allowed is that two evaluations from the same array may be done in either order, as long as neither crosses over an assignment to that array.
- Any use of a variable must follow all previous (according to the original block) procedure calls or indirect assignments through a pointer.
- Any procedure call or indirect assignment through a pointer must follow all previous (according to the original block) evaluations of any variable.

principal uses of registers

- In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation.
- Registers make good temporaries - places to hold the result of a subexpression while a larger expression is being evaluated, or more generally, a place to hold a variable that is used only within a single basic block.
- Registers are often used to help with run-time storage management, for example, to manage the run-time stack, including the maintenance of stack pointers and possibly the top elements of the stack itself.

Descriptors for data structure

- For each available register, a register descriptor keeps track of the variable names whose current value is in that register. Since we shall use only those registers that are available for local use within a basic block, we assume that initially, all register descriptors are empty. As the code generation progresses, each register will hold the value of zero or more names.
- For each program variable, an address descriptor keeps track of the location or locations where the current value of that variable can be found. The location might be a register, a memory address, a stack location, or some set of more than one of these. The information can be stored in the symbol-table entry for that variable name.

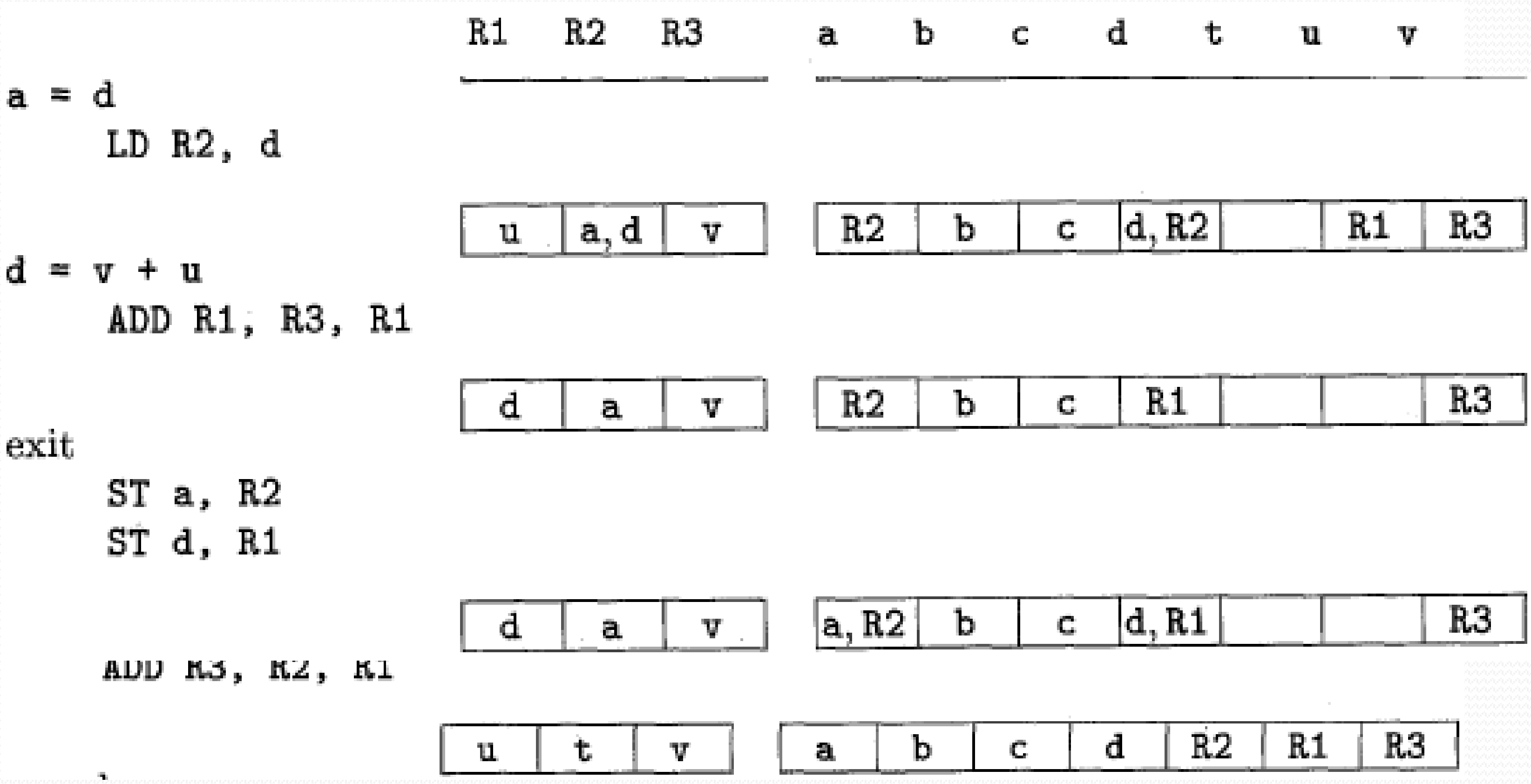
Machine Instructions for Operations

- Use getReg(x = y + z) to select registers for x, y, and z. Call these R_x, R_y and R_z.
- If y is not in R_y (according to the register descriptor for R_y), then issue an instruction LD R_y, y', where y' is one of the memory locations for y (according to the address descriptor for y).
- Similarly, if z is not in R_z, issue and instruction LD R_z, z', where z' is a location for *x* .
- Issue the instruction ADD R_x , R_y, R_z.

Rules for updating the register and address descriptors

- For the instruction LD R, x
  - Change the register descriptor for register R so it holds only x.
  - Change the address descriptor for x by adding register R as an additional location.
- For the instruction ST x, R, change the address descriptor for x to include its own memory location.
- For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction x = y + *x*
  - Change the register descriptor for R_x so that it holds only *x*.
  - Change the address descriptor for x so that its only location is R_x. Note that the memory location for x is *not* now in the address descriptor for x.
  - Remove R_x from the address descriptor of any variable other than x.
- When we process a copy statement x = y, after generating the load for y into register R_y, if needed, and after managing descriptors as for all load statements (per rule I):
  - Add x to the register descriptor for $R_{y.}$
  - Change the address descriptor for x so that its only location is R_y .

Instructions generated and the changes in the register and address descriptors



	R1	R2	R3	a	b	c	d	t	u	v
a = d										
LD R2, d										
	u	a, d	v	R2	b	c	d, R2		R1	R3
d = v + u										
ADD R1, R3, R1										
	d	a	v	R2	b	c	R1			R3
exit										
ST a, R2										
ST d, R1										
	d	a	v	a, R2	b	c	d, R1			R3
ADD R3, R2, R1										
	u	t	v	a	b	c	d	R2	R1	R3

Rules for picking register R_y for y

- If y is currently in a register, pick a register already containing y as R_y. Do not issue a machine instruction to load this register, as none is needed.
- If y is not in a register, but there is a register that is currently empty, pick one such register as R_y.
- The difficult case occurs when y is not in a register, and there is no register that is currently empty. We need to pick one of the allowable registers anyway, and we need to make it safe to reuse.

Possibilities for value of R

- If the address descriptor for v says that v is somewhere besides R, then we are OK.
- If v is x, the value being computed by instruction I, and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.
- Otherwise, if v is not used later (that is, after the instruction I, there are no further uses of v, and if v is live on exit from the block, then v is recomputed within the block), then we are OK.
- If we are not OK by one of the first two cases, then we need to generate the store instruction **ST** v, *R* to place a copy of v in its own memory location. This operation is called a spill.

Selection of the register Rx

1. Since a new value of x is being computed, a register that holds only x is always an acceptable choice for Rx.
2. If y is not used after instruction I, and Ry holds only y after being loaded, Ry can also be used as Rx. A similar option holds regarding z and Rx.

Possibilities for value of R

- If the address descriptor for v says that v is somewhere besides R, then we are OK.
- If v is x, the value being computed by instruction I, and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.
- Otherwise, if v is not used later (that is, after the instruction I, there are no further uses of v, and if v is live on exit from the block, then v is recomputed within the block), then we are OK.
- If we are not OK by one of the first two cases, then we need to generate the store instruction **ST** v, *R* to place a copy of v in its own memory location. This operation is called a spill.

Characteristic of peephole optimizations

- Redundant-instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Redundant-instruction elimination

- LD a, R0

  ST R0, a
- if debug == 1 goto **L1**

  goto L2

  L I : print debugging information

  L2:

Flow-of-control optimizations

```
goto L1
…
Ll: goto L2
```

Can be replaced by:

```
goto L2
…
Ll: goto L2
```

```
if a<b goto L1
…
Ll: goto L2
```

Can be replaced by:

```
if a<b goto L2
…
Ll: goto L2
```

Algebraic simplifications

- $x=x+0$
- $x=x*1$

Register Allocation and Assignment

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

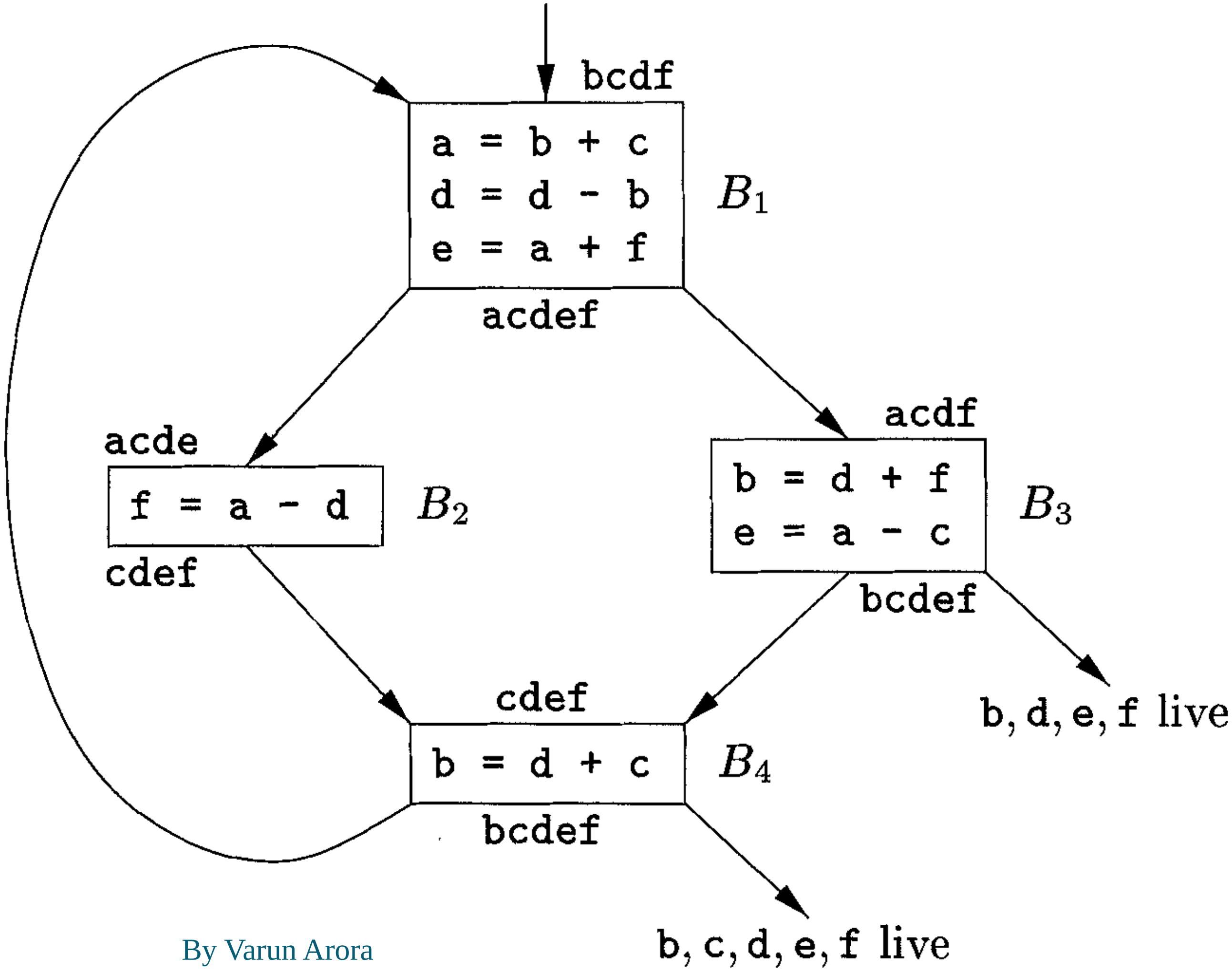
Global register allocation

- 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

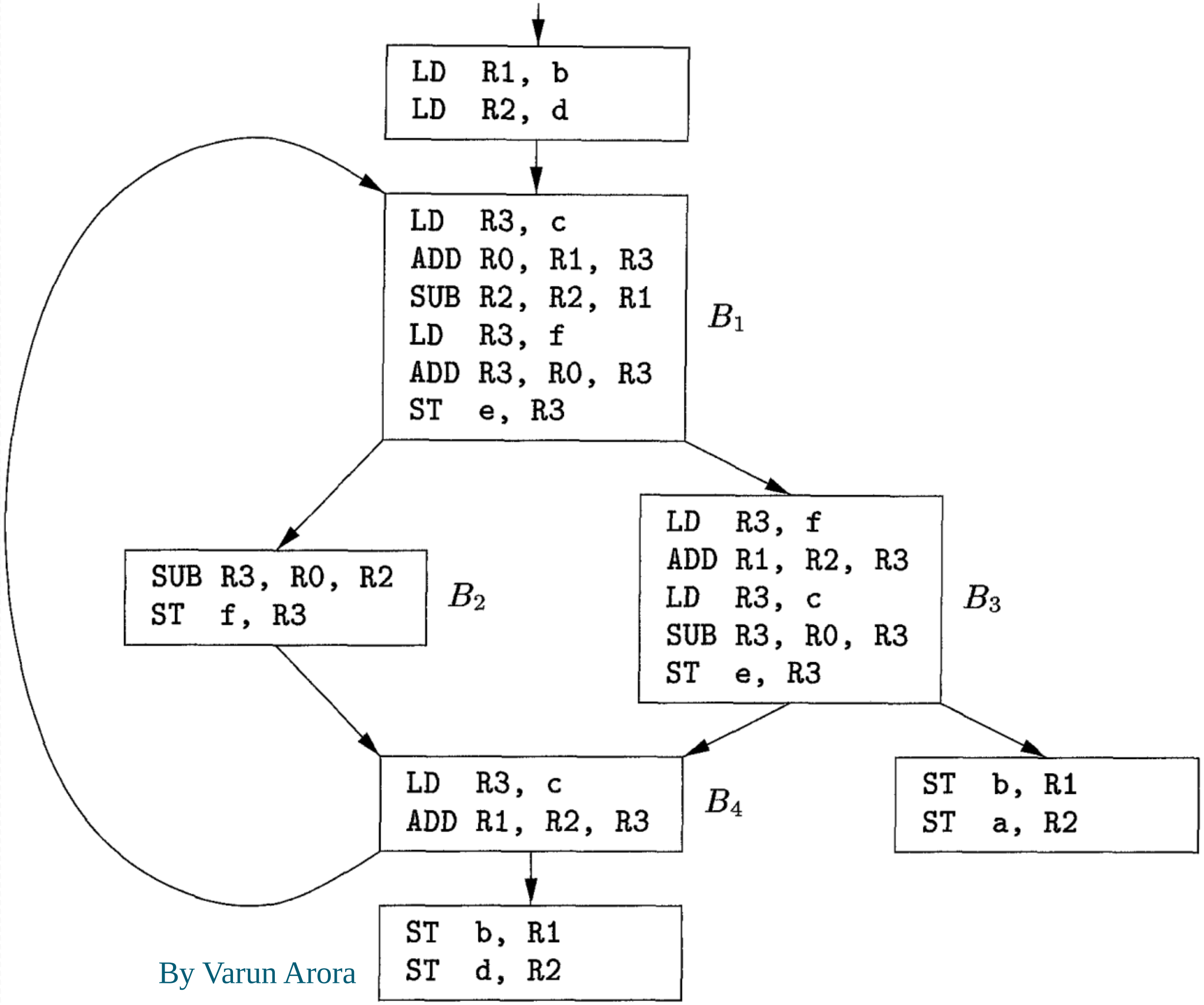
Usage counts

- 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 ass 2 units of saving

Flow graph of an inner loop



Code sequence using global register assignment



```
LD   R1, b
LD   R2, d
```

B_1

```
LD   R3, c
ADD  R0, R1, R3
SUB  R2, R2, R1
LD   R3, f
ADD  R3, R0, R3
ST   e, R3
```

B_2

```
SUB  R3, R0, R2
ST   f, R3
```

B_3

```
LD   R3, f
ADD  R1, R2, R3
LD   R3, c
SUB  R3, R0, R3
ST   e, R3
```

B_4

```
LD   R3, c
ADD  R1, R2, R3
```

```
ST   b, R1
ST   a, R2
```

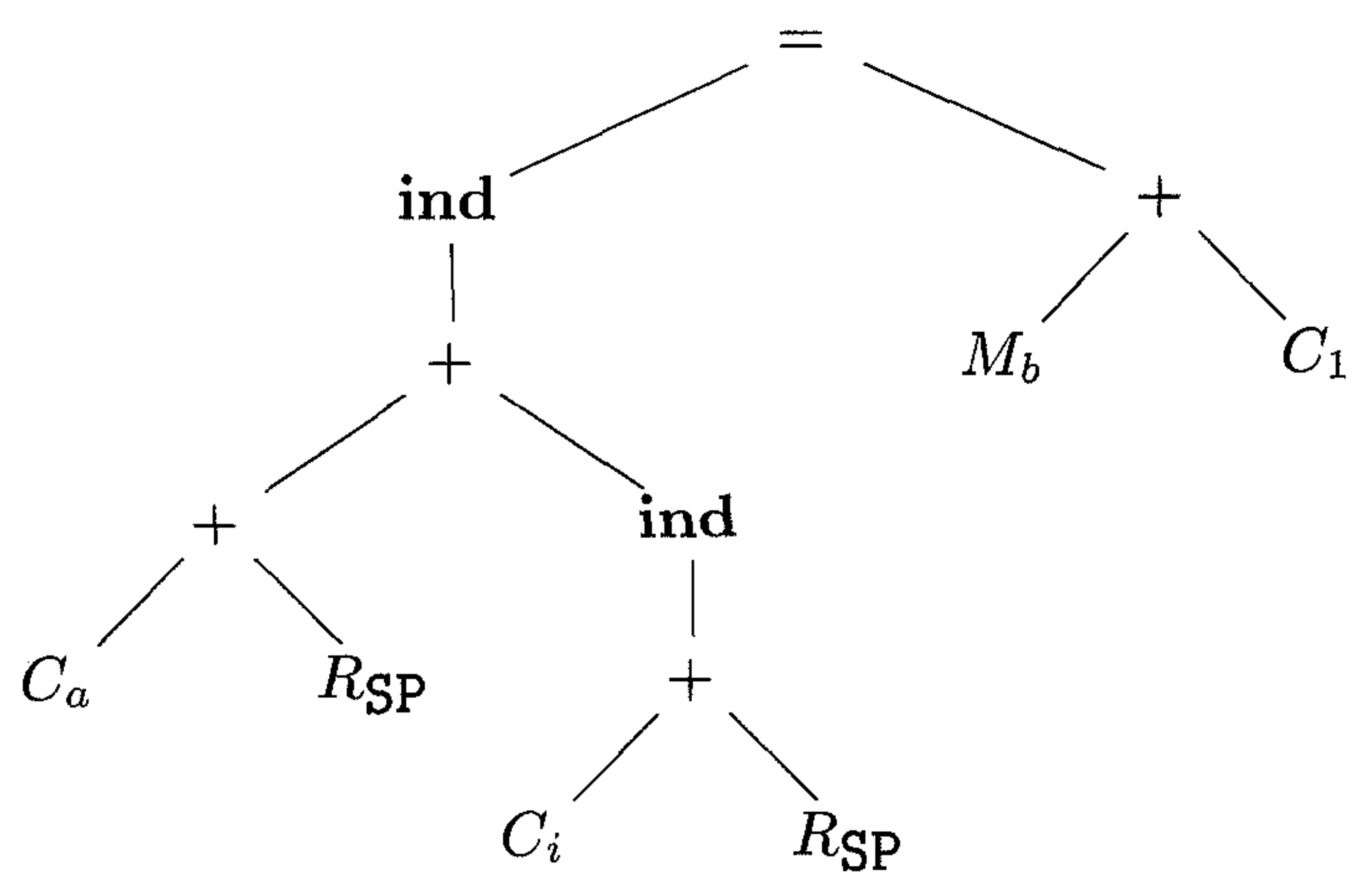
```
ST   b, R1
ST   d, R2
```

By Varun Arora

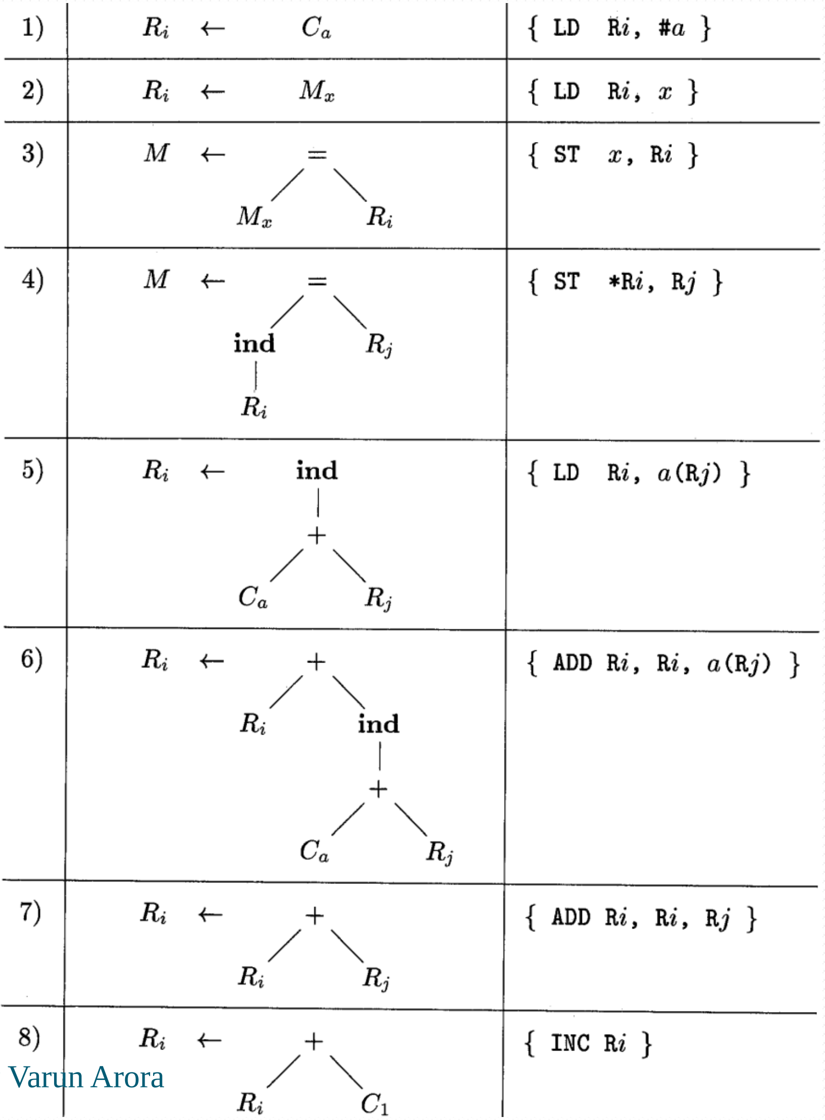
Register allocation by Graph coloring

- 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.
    - For example in the previous example an edge connects a and d in the graph
    - Use a graph coloring algorithm to assign registers.

Intermediate-code tree for a[i]=b+1

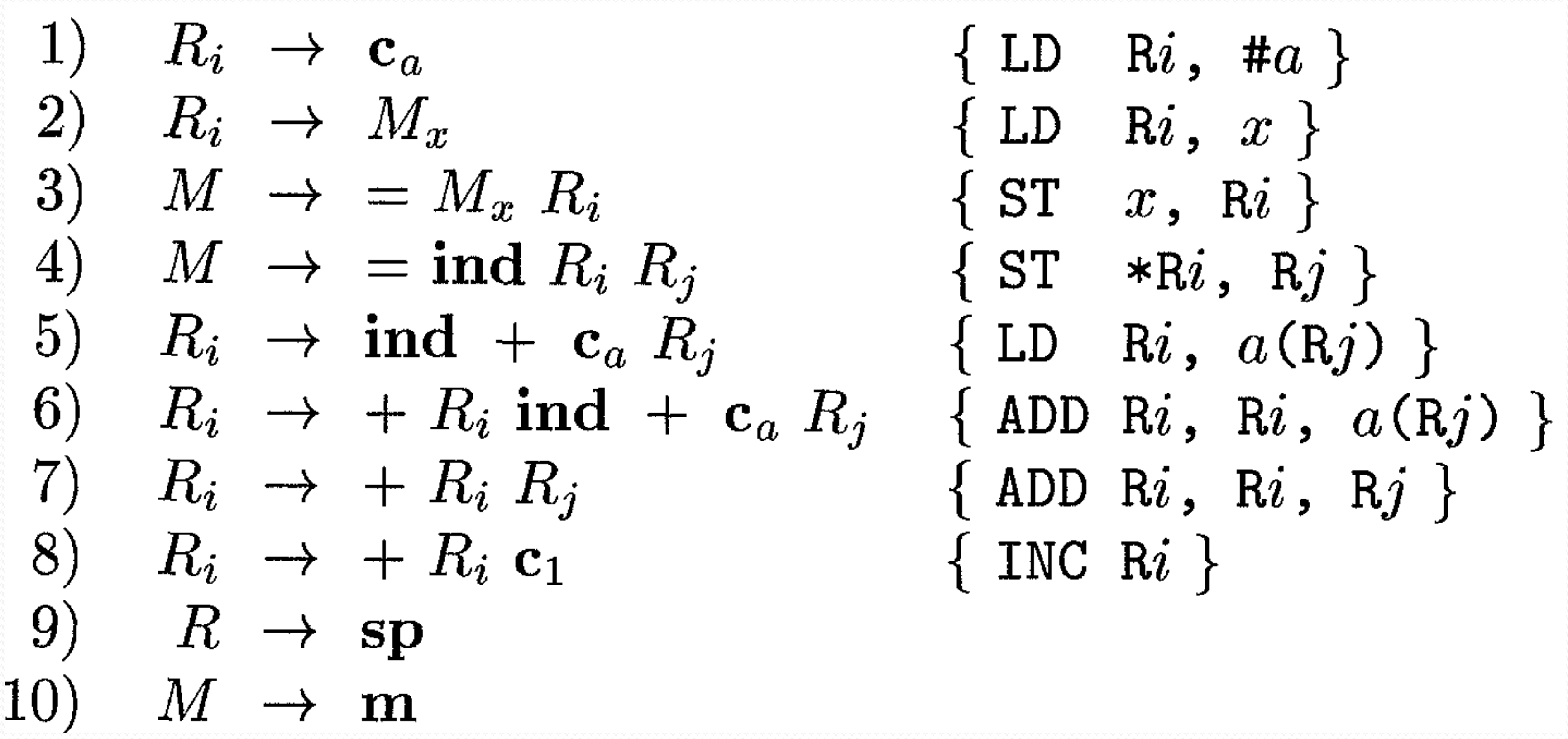


Tree-rewriting rules



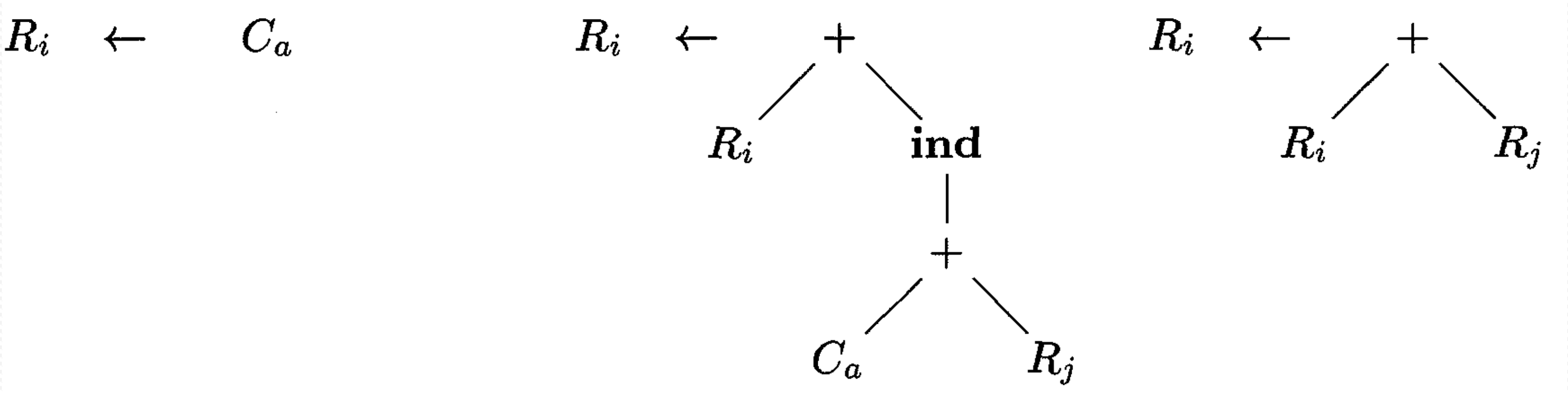
1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow$ = M_x R_i	{ ST x, Ri }
4)	$M \leftarrow$ = **ind** R_j R_i	{ ST *Ri, Rj }
5)	$R_i \leftarrow$ **ind** + C_a R_j	{ LD Ri, a(Rj) }
6)	$R_i \leftarrow$ + R_i **ind** + C_a R_j	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \leftarrow$ + R_i R_j	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow$ + R_i C_1	{ INC Ri }

Syntax-directed translation scheme



1)	$R_i \rightarrow \mathbf{c}_a$	{ LD Ri , #a }
2)	$R_i \rightarrow M_x$	{ LD Ri , x }
3)	$M \rightarrow = M_x \ R_i$	{ ST x , Ri }
4)	$M \rightarrow = \mathbf{ind} \ R_i \ R_j$	{ ST *Ri , Rj }
5)	$R_i \rightarrow \mathbf{ind} \ + \ \mathbf{c}_a \ R_j$	{ LD Ri , a(Rj) }
6)	$R_i \rightarrow + \ R_i \ \mathbf{ind} \ + \ \mathbf{c}_a \ R_j$	{ ADD Ri , Ri , a(Rj) }
7)	$R_i \rightarrow + \ R_i \ R_j$	{ ADD Ri , Ri , Rj }
8)	$R_i \rightarrow + \ R_i \ \mathbf{c}_1$	{ INC Ri }
9)	$R \rightarrow \mathbf{sp}$	
10)	$M \rightarrow \mathbf{m}$	

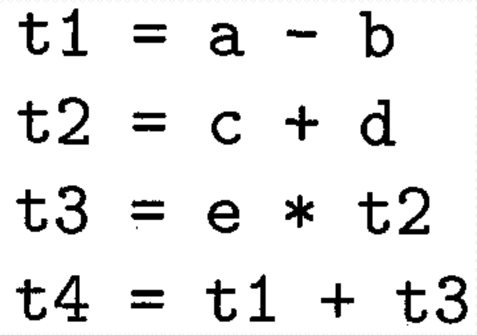
An instruction set for tree matching



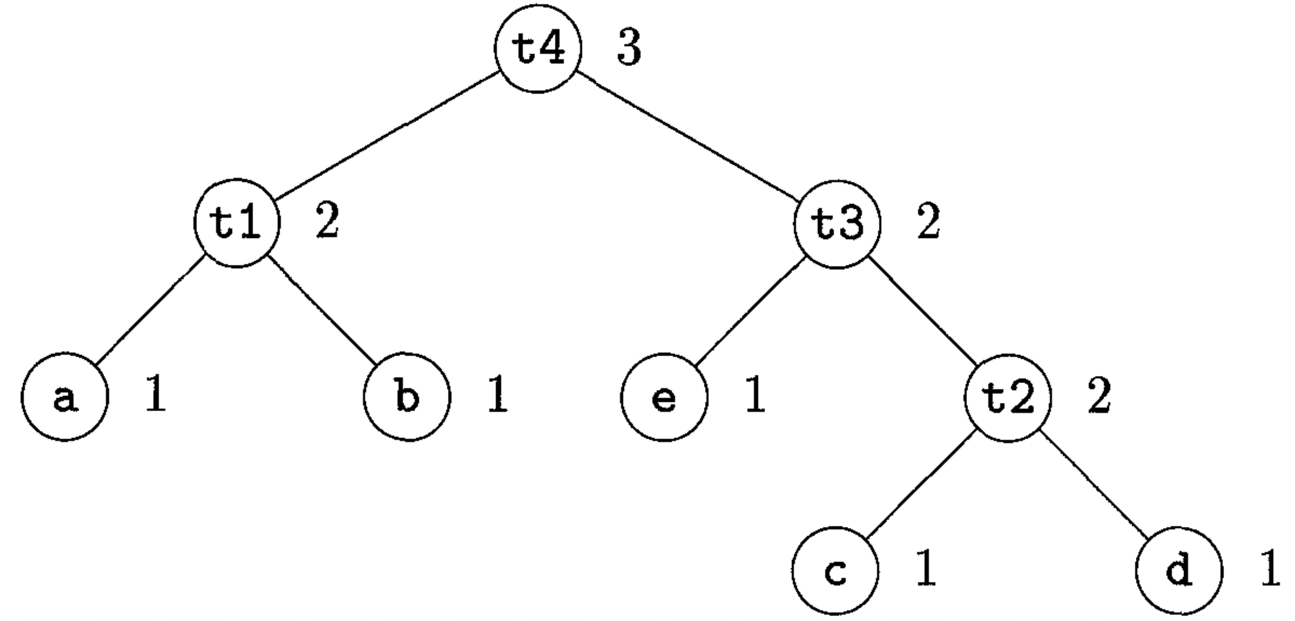
Ershov Numbers

- Label any leaf 1.
- The label of an interior node with one child is the label of its child.
- The label of an interior node with two children is
  - The larger of the labels of its children, if those labels are different.
  - One plus the label of its children if the labels are the same.

A tree labeled with Ershov numbers



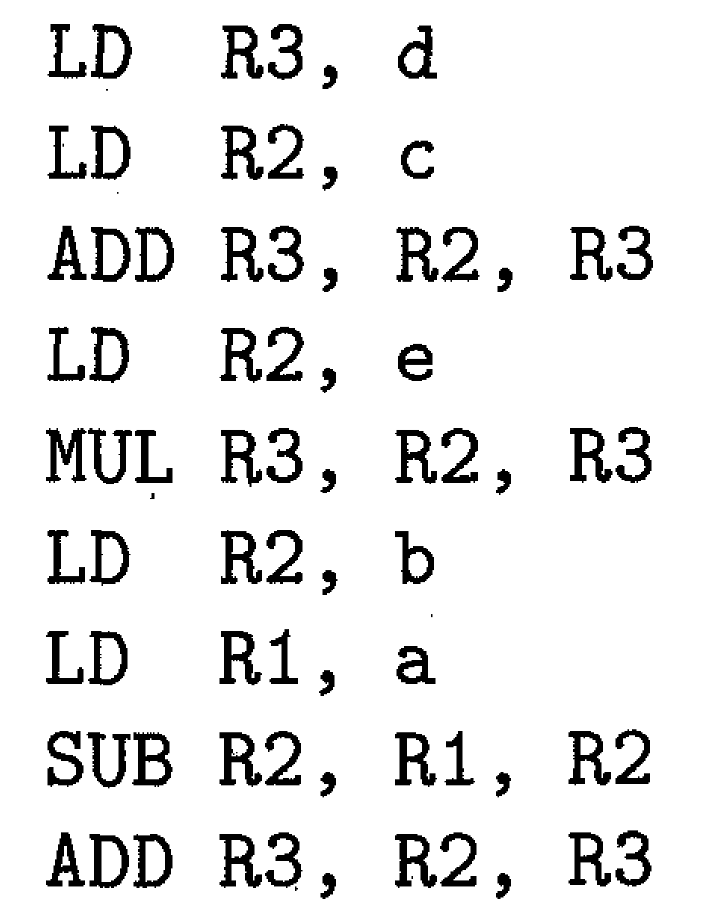
```
t1 = a - b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```



Generating code from a labeled expression tree

- To generate machine code for an interior node with label *k* and two children with equal labels (which must be *k* - l) do the following:
  - Recursively generate code for the right child, using base *b*+1. The result of the right child appears in register R_{b+k}.
  - Recursively generate code for the left child, using base b; the result appears in R_{b+k-1}.
  - Generate the instruction OP R_{b+k}, R_{b+k-1}, R_{b+k}, where OP is the appropriate operation for the interior node in question.
- Suppose we have an interior node with label *k* and children with unequal labels. Then one of the children, which we'll call the "big" child, has label *k* , and the other child, the "little" child, has some label m < *k*. Do the following to generate code for this interior node, using base *b:*
  - Recursively generate code for the big child, using base b; the result appears in register R_{b+k-l}.
  - Recursively generate code for the small child, using base *b;* the result appears in register R_{b+m-l}. Note that since m < k, neither R_{b+k-l} nor any higher-numbered register is used.
  - Generate the instruction OP R_{b+k-l}, R_{b+m-l}, R_{b+k-1} or the instruction OP R_{b+k-l}, R_{b+k-l}, R_{b+m+l}, depending on whether the big child is the right or left child, respectively.
- For a leaf representing operand x, if the base is *b* generate the instruction LD R_b, *x*.

Optimal three-register code

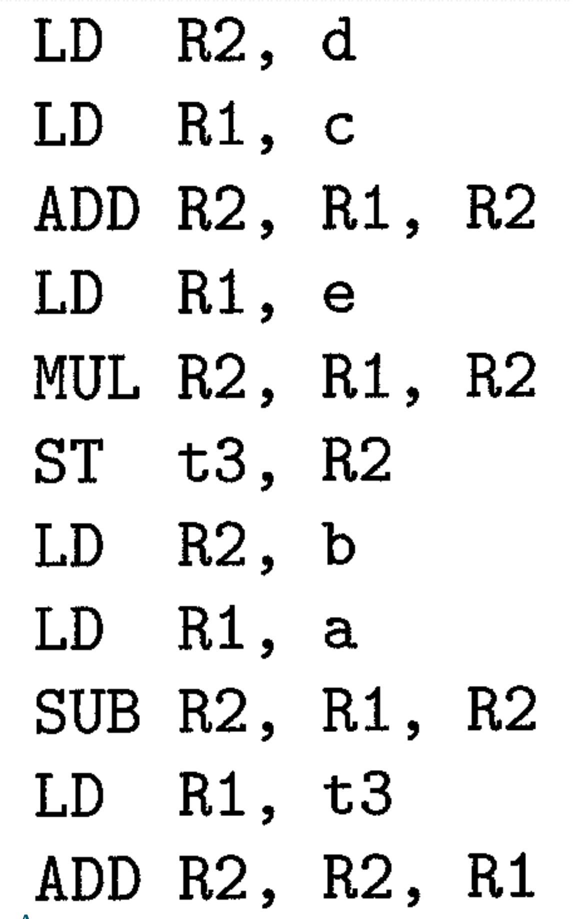


```
LD  R3, d
LD  R2, c
ADD R3, R2, R3
LD  R2, e
MUL R3, R2, R3
LD  R2, b
LD  R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

Evaluating Expressions with an Insufficient Supply of Registers

- Node N has at least one child with label r or greater. Pick the larger child (or either if their labels are the same) to be the "big" child and let the other child be the "little" child.
- Recursively generate code for the big child, using base b = 1. The result of this evaluation will appear in register R_r
- Generate the machine instruction ST t_k, R_r, where t_k is a temporary variable used for temporary results used to help evaluate nodes with label k.
- Generate code for the little child as follows. If the little child has label r or greater, pick base b=1. If the label of the little child is *j*<r, then pick b=r-*j*. Then recursively apply this algorithm to the little child; the result appears in R_r.
- Generate the instruction LD R_r-l, t_k.
- If the big child is the right child of N, then generate the instruction OP R_r, R_r, R_{r-1}. If the big child is the left child, generate OP R_r, R_{r-1}, R_r.

Optimal three-register code using only two registers

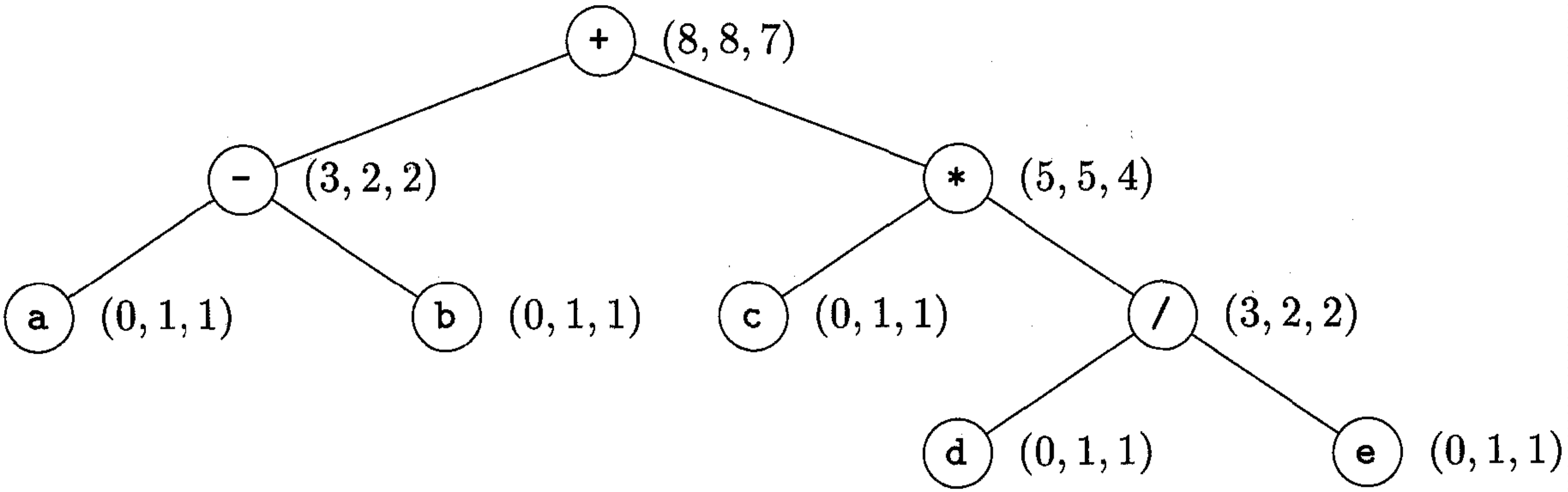


```
LD   R2, d
LD   R1, c
ADD  R2, R1, R2
LD   R1, e
MUL  R2, R1, R2
ST   t3, R2
LD   R2, b
LD   R1, a
SUB  R2, R1, R2
LD   R1, t3
ADD  R2, R2, R1
```

Dynamic Programming Algorithm

- 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, for $1 \le i \le r$
- Traverse T, using the cost vectors to determine which subtrees of T must be computed into memory.
- Traverse each tree using the cost vectors and associated instructions to generate the final target code. The code for the subtrees computed into memory locations is generated first.

Syntax tree for (a-b)+c*(d/e) with cost vector at each node



minimum cost of evaluating the root with two registers available

- Compute the left subtree with two registers available into register R0, compute the right subtree with one register available into register R1, and use the instruction ADD R0, R0, R1 to compute the root. This sequence has cost 2+5+1=8.
- Compute the right subtree with two registers available into R l , compute the left subtree with one register available into R0, and use the instruction ADD R0, R0, R1. This sequence has cost 4+2+1=7.
- Compute the right subtree into memory location M, compute the left subtree with two registers available into register RO, and use the instruction ADD R0, R0, M. This sequence has cost 5+2+1=8.