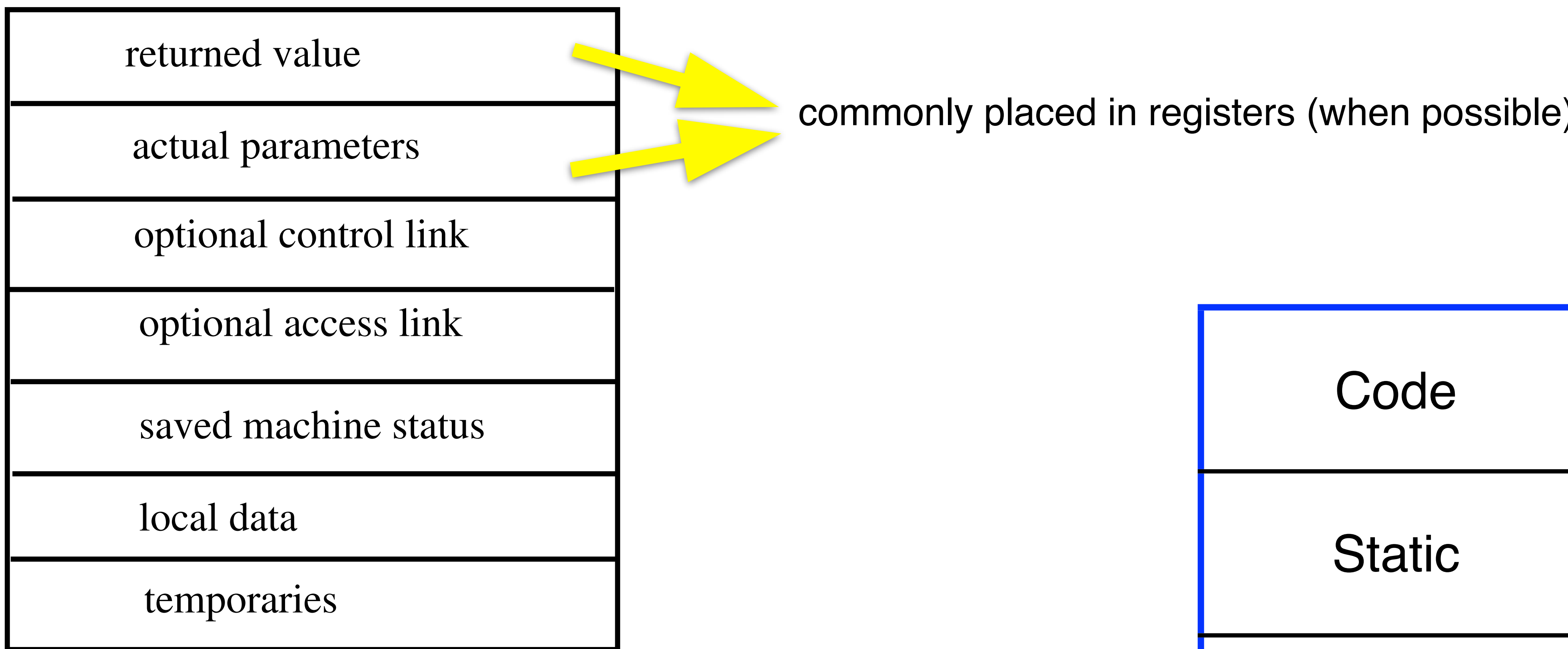
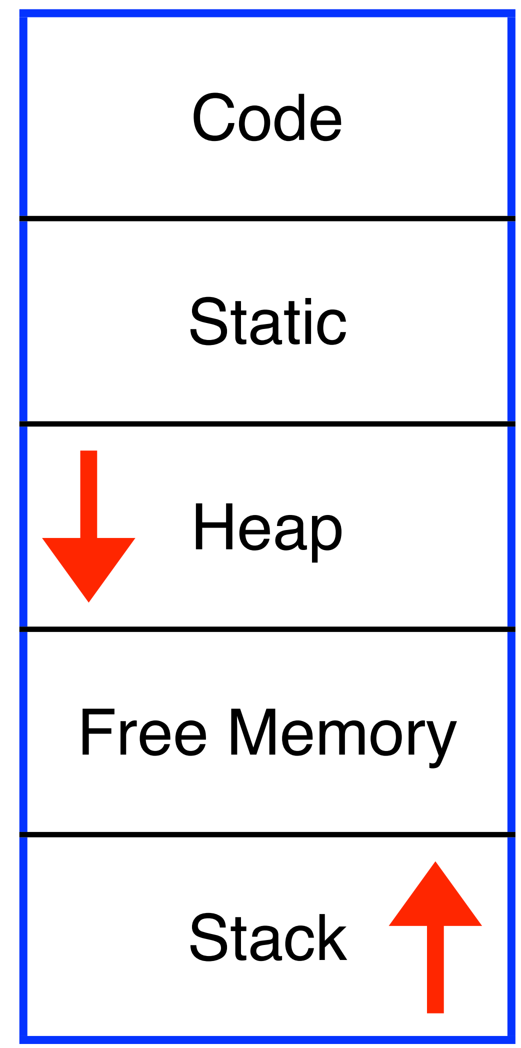
CODE GENERATION

memory management



A.R.



For some compiler, the intermediate code is a pseudo-code of a virtual machine.

- Interpreter of the virtual machine is invoked to execute the intermediate code.
- No machine-dependent code generation is needed.
- Usually with great overhead.

  Example:
  - Pascal: P-code for the virtual P machine.
  - JAVA: Byte code for the virtual JAVA machine.

Machine-dependent issues

- Input and output formats:
  - The formats of the intermediate code and the target program.
- Memory management:
  - Alignment, indirect addressing, paging, segment, . . .
  - Those you learned from your assembly language class.
- Instruction cost:
  - Special machine instructions to speed up execution.
  - Example:
    - Increment by 1.
    - Multiplying or dividing by 2.
    - Bit-wise manipulation.
    - Operators applied on a continuous block of memory space.
  - Pick a fastest instruction combination for a certain target machine.

Machine-dependent issues

- **Register allocation**: in-between machine dependent and independent issues.
  - C language allows the user to management a pool of registers.
  - Some language leaves the task to compiler.
  - Idea: save mostly used intermediate result in a register. However, finding an optimal solution for using a limited set of registers is NP-hard.

```
t=a+b
```

```
load R0,a
load R1,b
add R0,R1
store R0,T
load R0,a
add R0,b
store R0,T
```

Heuristic solutions: similar to the ones used for the swapping problem.

Machine-independent issues

Techniques.

- Analysis of dependence graphs.
- Analysis of basic blocks and flow graphs.
- Semantics-preserving transformations.
- Algebraic transformations.

Machine dependend issues

the target language: RISC (+ a little of CISC)

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

A Simple Target Machine Model

byte-addressable machine with *n* general-purpose registers, R_0, R_1, ... , R_{n-1}

OPERATIONS	FORMAT
Load	LD r,x (r=x)
Store	ST x,r (x=r)
Computation	OP r1, r2,r3 (SUB r1, r2,r3 // r1=r2-r3)
Unconditional jumps	BR L
Conditional jump	Bcond r, L (BLTZ r, L)

addressing modes

format	addr	examples
x //	Lval(x)	name
a(r) //	Lval(a)+ Rval(r)	LD R1 a(R2)
const(r) //	const+Rval(r)	LD R1, 100(R2)
*r //	Rval(Rval(r))	LD R1, *(R2)
*const(r) //	Rval(const+Rval(r))	LD R1, *100(R2)
#const // immediate op	nil	LD R1, #100

x = y-z

```
LD R1, y              //R1=y
LD R2, z              //R2=z
SUB R1,R1,R2          //R1=R1=R2
ST x, R1              //x=R1
```

b = a[i]

```
LD R1, i              //R1=i
MUL R1,R1,8           //R1= R1*8
LD R2,a(R)            //R2 = Rval(a+Rval(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                //Rval(a+Rval(R2)) = R1
```

x = *i

```
LD R1, i                 //R1=i
LD R2,0(R1)              //R2 = Rval(0 + Rval(R1))
ST b, R2                 //b=R2
```

```
*p = y
```

```
LD R1, p            //R1=p
LD R2, y            //R2=y
ST 0(R1),R2         //Rval(0 + Rval(R1)) = R2
```

```
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
```

M is the label that represents the first machine instruction generated from the three-address instruction that has label L

Program and Instruction Costs

we shall assume each target-language instruction has an associated cost

we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands

- cost(**`LD R0, R1`**)=1

  This instruction has a cost of one because no additional memory words are required.
- cost(**`LD R0, M`**)=2

  The cost is two since the address of memory location M is in the word following the instruction.
- cost(**`LD R1, *100(R2)`**)=3

  The cost is three because the constant 100 is stored in the word following the instruction.

Procedure call, return:

no parameters:

procedure call:

`call` *callee*

return:

```
return
halt (return of the main)
```

`action` `a generic sequence of three addr instructions`

```
here            :ST callee.static.area, #(here+20)   // save return address here+20
                                                     // in location with
                                                     // address callee.static.area
here+16         :BR callee.codeArea                  // call procedure
here+20         : ...

end             :HALT                                // return to operating system

callee.codeArea :action2

ret             :BR *callee.static.area              // return to address saved in location
                                                     // callee.static.area
```

```
// 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
```

```
452:
```

```
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
```

```
BR *0 (SP)   // return to caller
```

// 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:	ACTION1 2	
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

Register allocation: in-between machine dependent and independent issues.

- C language allows the user to management a pool of registers.
- Some language leaves the task to compiler.
- Idea: save mostly used intermediate result in a register.

finding an optimal solution for using a limited set of registers isNP-hard.

```
t=a+b
```

```
load R0,a
load R1,b
add R0,R1
store R0,T
```

```
load R0,a
add R0,b
store R0,T
```

Machine-independent issues

Techniques.

- Analysis of dependence graphs.
- Analysis of basic blocks and flow graphs.
- Semantics-preserving transformations.
- Algebraic transformations.

basic blocks:

maximal sequences of consecutive three-address instructions s.t.:

- The flow of control can only enter the basic block through the first instruction in the block (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.

- Partition the intermediate code into *basic blocks*
- The basic blocks become the nodes of a *flow graph,* whose edges indicate which blocks can follow which other blocks.

Partitioning three-address instructions into basic blocks.

Algorithm 8.5:

INPUT:

A sequence of three-address instructions.

OUTPUT:

A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD:

a) determine the *leaders:*

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

b) for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

```
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)
```

leaders ?

```
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)
```

```
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)
```

```
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)
```

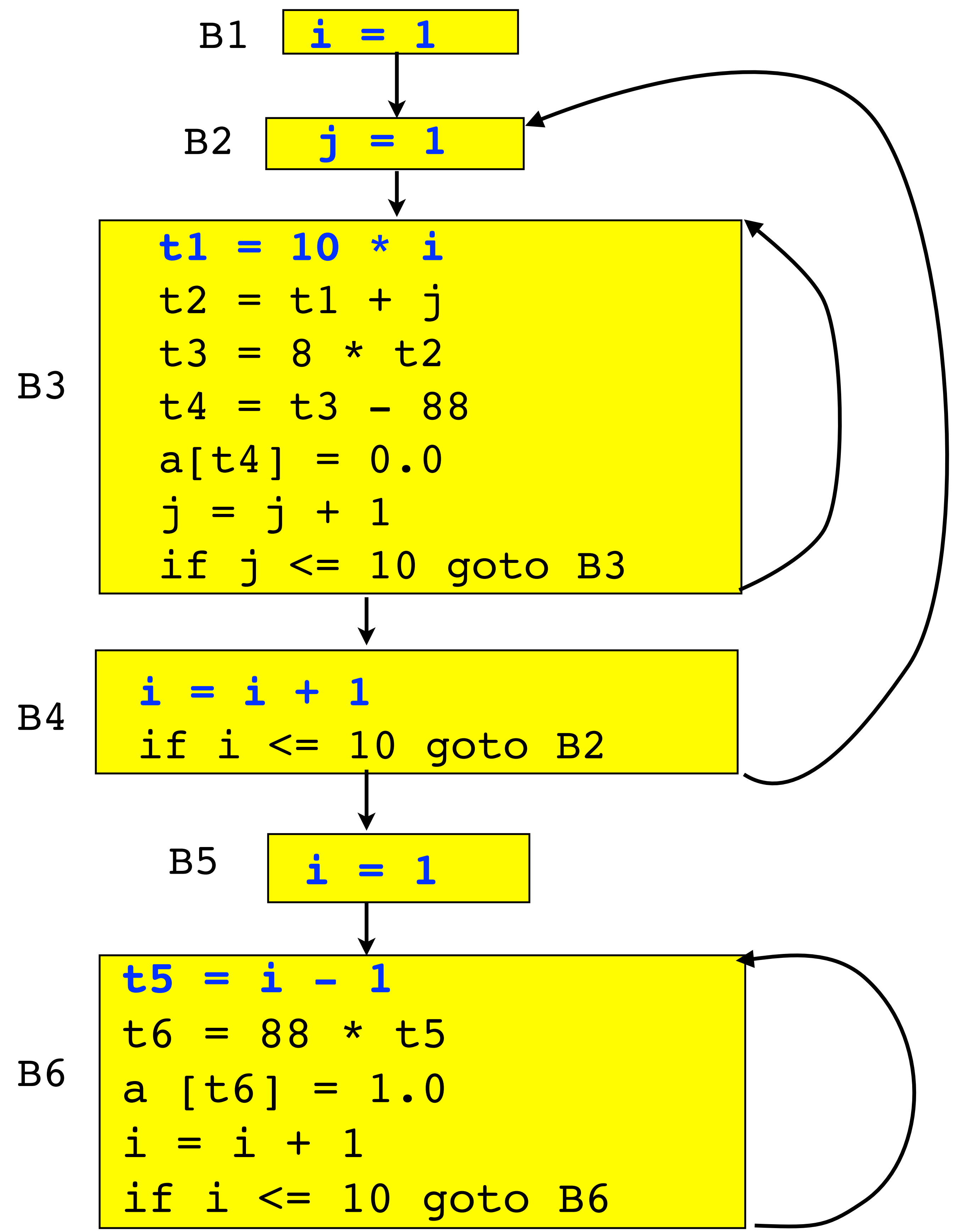
```
12) i = 1
```

```
13) t5 = i - 1
14) t6 = 88 * t5
15) a [t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

```
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)
```

```
B1:
    i = 1

B2:
    j = 1

B3:
    t1 = 10 * i
    t2 = t1 + j
    t3 = 8 * t2
    t4 = t3 - 88
    a[t4] = 0.0
    j = j + 1
    if j <= 10 goto B3

B4:
    i = i + 1
    if i <= 10 goto B2

B5:
    i = 1

B6:
    t5 = i - 1
    t6 = 88 * t5
    a [t6] = 1.0
    i = i + 1
    if i <= 10 goto B6
```

```
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
(inside a block)

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:

1) **statement j *uses* the value of `x` computed at statement i.**
2) **`x` is *live* at statement i.**

Our algorithm to determine liveness and next-use information makes a backward pass over each basic block. We store the information in the symbol table. Since procedures can have arbitrary side effects, we assume for convenience that each procedure call starts a new basic block

Determining the liveness and next-use

Algorithm 8.7:

INPUT:

A basic block ***B*** of three-address statements. We assume that the symbol table initially shows all nontemporary variables in *B* as being **live** on exit.

OUTPUT:

$\forall$ `x=y+z` $\in$ ***B***, we attach to `x=y+z` the liveness and next-use information of **x**, **y**, and **z**.

METHOD:

Starting with the last statement in ***B*** and scanning backwards

foreach `x = y+z` **do**

1. Attach to statement `x = y+z` the information currently found in the symbol table
   regarding the next use and liveness of **x**, **y**, and **y**.
2. In the symbol table, set **x** to "**not live**" and "**no next use**."
3. In the symbol table, set **y** and **z** to "**live**" and the next uses
   of **y** and **z** to `x=y+z`.

Here we have used + as a symbol representing any operator. If the three-address statement i is of the form *x* = + *y* or *x* = *y*, the steps are the same as above, ignoring *z*. Note that the order of steps (2) and (3) may not be interchanged because *x* may be *y* or *z*.

Optimization of Basic Blocks

- Local optimization within each basic block
- Global optimization

focuses on the local optimization

DAG Representation of Basic Blocks

Target: Construct a DAG for a basic block

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.

2. ∀ statement **s.** we associate a node N_s.
The **children** of N_s are those nodes corresponding to statements that are the last definitions, prior to ***s****,* of the operands used by ***s****.*

3. each node N_s is labeled by the operator applied at **s**.
Attached to N_s is the list of variables for which it is the last definition within the block.

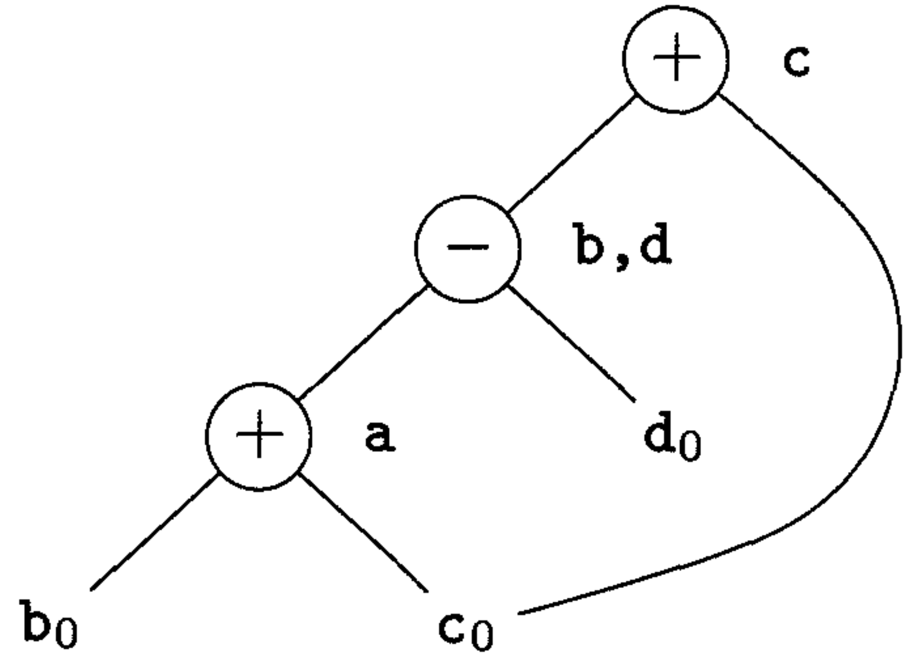
4. Certain nodes are designated *output nodes*. These are the nodes whose variables are live on exit from the block; that is, their values may be used later, in another block of the flow graph.

The DAG representation of a basic block lets us perform

- eliminating local common sub-expressions
- eliminating dead code
- reordering statements that do not depend on one another
- applying algebraic laws to reorder operands of three-address instructions

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

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

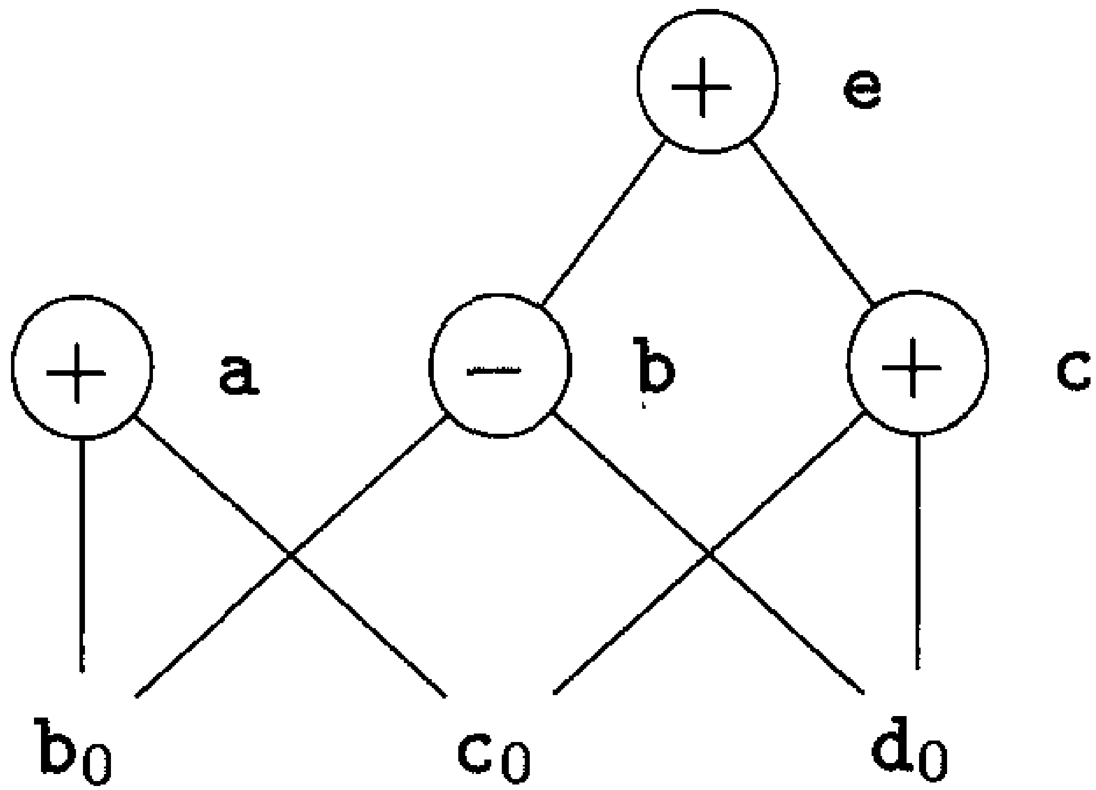


When we construct the node for the third statement **c=b+c**, we know that the use of **b** in **b+c** refers to the node labeled **-**, because that is the most recent definition of **b**. Thus, we do not confuse the values computed at statements one and three.

However, the node corresponding to the fourth statement **d=a-d** has the operator **-** and the nodes with attached variables **a** and $\mathbf{d_0}$ as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add **d** to the list of definitions for the node labeled

Dead Code Elimination

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

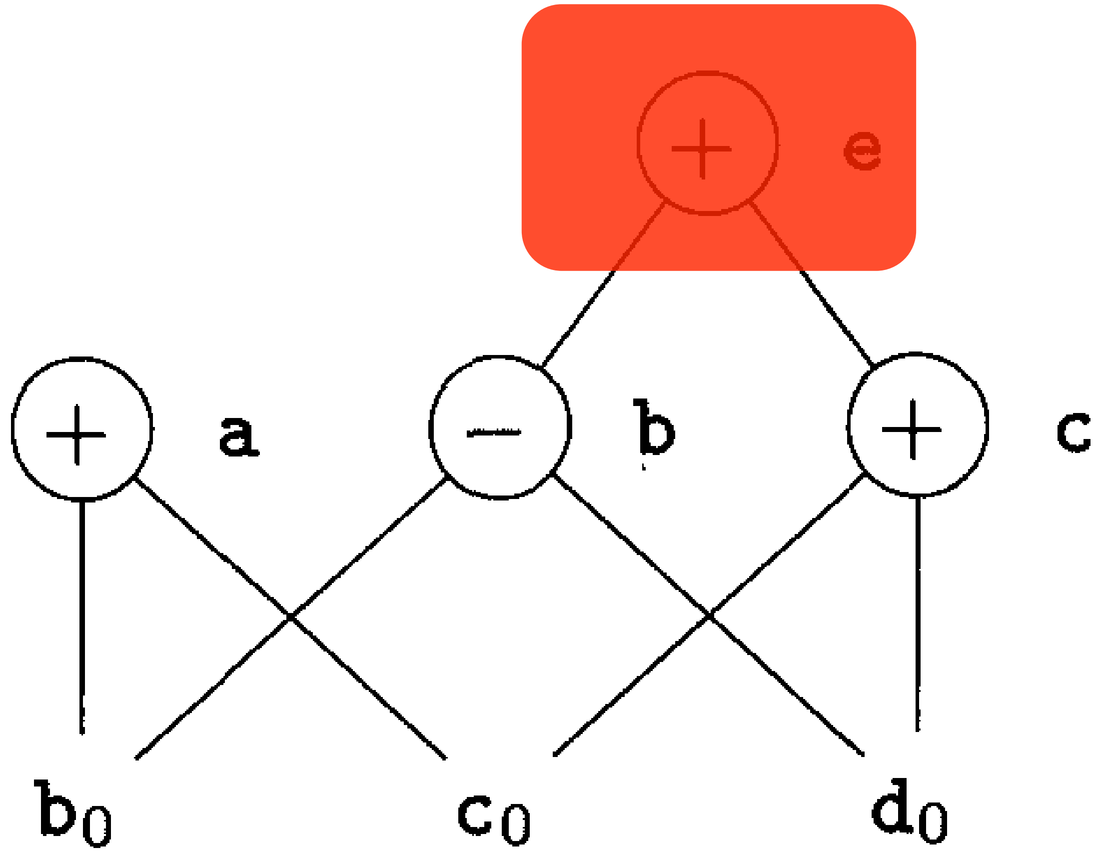


If a and b are live but c and e are not:

We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Dead Code Elimination

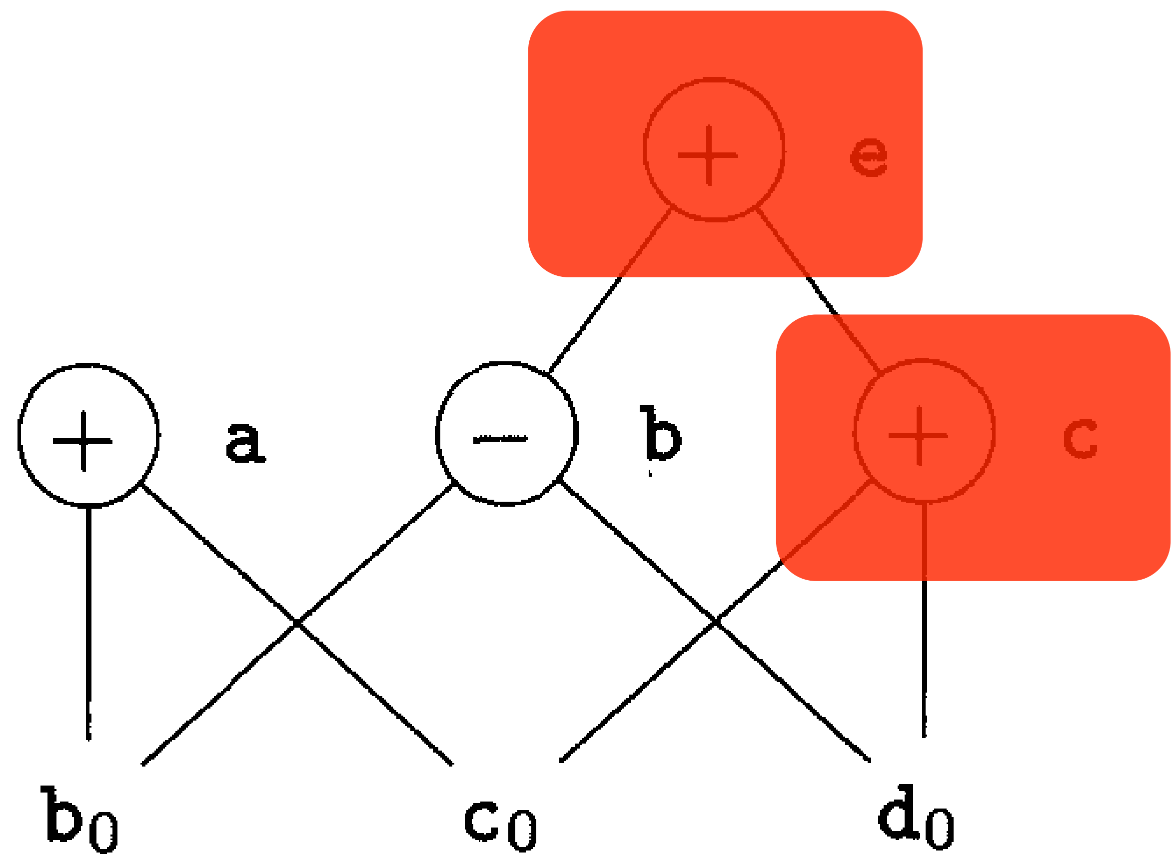
```
a = b + c;
b = b - d
c = c + d
e = b + c
```



We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Dead Code Elimination

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



We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Algebraic Simplifications

```
x+0=0+x=x  x*1=1*x=x  ...
```

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

intermediate code translation

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

associativity

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

The compiler writer should examine the language reference manual carefully to determine what rearrangements of computations are permitted, since (because of possible overflows or underflows) computer arithmetic does not always obey the algebraic identities of mathematics

Algebraic Simplifications

reduction in strength

replacing a more expensive operator by a cheaper one

EXPENSIVE		CHEAPER
x^2	=	`x * x`
`2*x`	=	`x+x`
`x/2`	=	`x * 0.5`

Constant Folding

- Operations on constants can be computed at compile time
  - If there is a statement `x = y op z`
  - And `y` and `z` are constants
  - Then `y` op `z` can be computed at compile time
- Example: `x = 2 + 2` ⇒ `x = 4`
- Example: `if 2 < 0 goto L` can be deleted

Flow of Control Optimizations

- **Eliminate unreachable basic blocks:**
  Code that is unreachable from the initial block

- **E.g., basic blocks that are not the target of any jump or "fall through" from a conditional**

- **Why would such basic blocks occur?**

- **Removing unreachable code makes the program smaller ... and sometimes also faster**

Single Assignment Form

- **Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment**
- **Rewrite intermediate code in single assignment form**

```
x = z + y
a = x
x = 2 * x
```

```
b = z + y
a = b
x = 2 * b
```

(`b` is a fresh register)

More complicated in general, due to loops

Common Subexpression Elimination

- If
  - Basic block is in single assignment form
  - A definition `x :=` is the first use of `x` in a block
- Then
  - When two assignments have the same rhs, they compute the same value
- Example:

```
x = y + z
w = y + z
```

```
x = y + z
w = x
```

(the values of x, y, and z do not change in the … code)

Copy Propagation

- If `w = x` appears in a block, replace subsequent uses of `w` with uses of `x`
  – Assumes single assignment form
- Example:

```
b = z + y
a = b
x = 2 * a
```

```
b = z + y
a = b
x = 2 * b
```

- Only useful for enabling other optimizations
  – Constant folding
  – Dead code elimination

Copy Propagation and Constant Folding

```
a = 5
x = 2 * a
y = x + 6
t = x * y
```

```
a = 5
x = 10
y = 16
t = x << 4
```

Copy Propagation and Dead Code Elimination

If `w = rhs` appears in a basic block
`w` does not appear anywhere else in the program
then
the statement `w = rhs` is dead and can be eliminated

– Dead = does not contribute to the program's result
Example: (`a` is not used anywhere else)

```
x = z + y
a = x
x = 2 * a
```

```
b = z + y
a = b
x = 2 * b
```

```
b = z + y
x = 2 * b
```