

Compilerconstructie

najaar 2017

<http://www.liacs.leidenuniv.nl/~vlietrvan1/coco/>

Rudy van Vliet

kamer 140 Snellius, tel. 071-527 2876

rvvliet(at)liacs(dot)nl

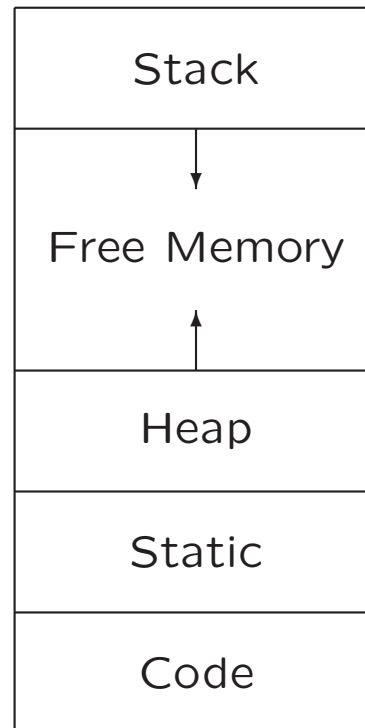
college 8, vrijdag 17 november 2017

+ 'werkcollege'

Storage Organization

Code Generation

7.1 Storage Organization



Typical subdivision of run-time memory into code and data areas

7.1.1 Static Versus Dynamic Storage Allocation

- Static: compile time
- Dynamic: run time

Dynamic storage allocation:

- Stack storage: for data local to procedure
- Heap storage: for data that outlives procedure

Garbage collection to support heap management

7.2 Stack Allocation of Space

Possible because procedure calls are **nested**

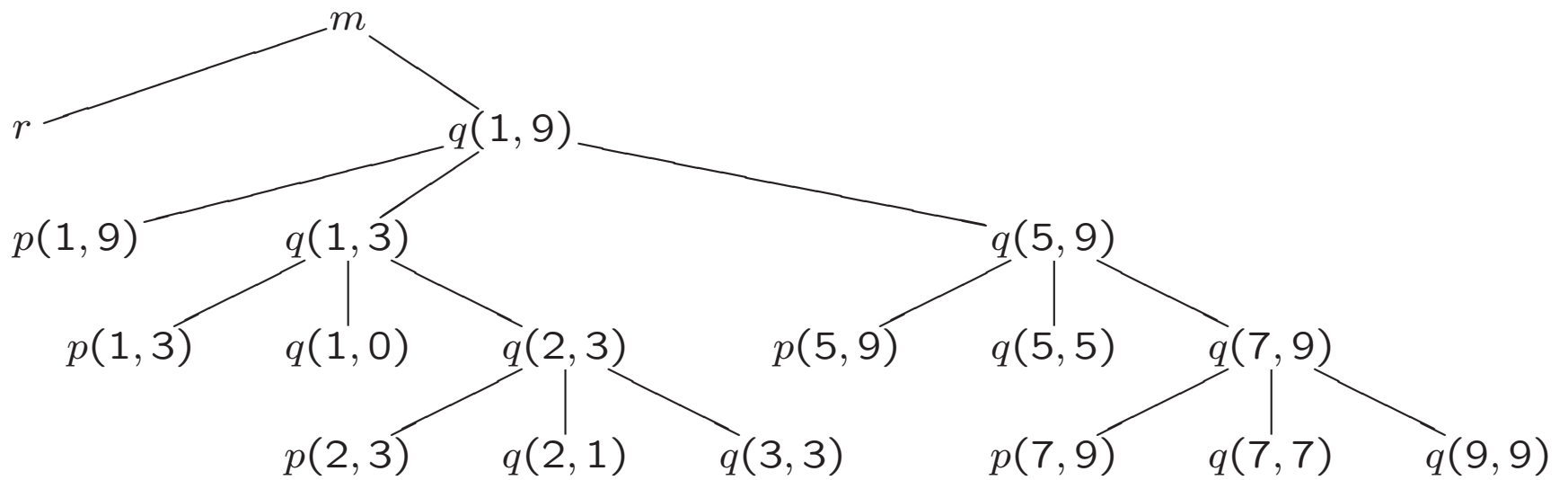
7.2 Stack Allocation of Space

```
int a[11];
void readArray() /* Reads 9 integers into a[1],...a[9]. */
{ int i;
  ...
}
int partition (int m, int n)
{ /* Picks a separator value v, and partitions a[m..n] so that
   a[m..p-1] are less than v, a[p]=v, and a[p+1..n} are
   equal to or greater than v. Returns p. */
  ...
}
void quicksort (int m, int n)
{ int i;
  if (n > m)
  { i = partition(m, n);
    quicksort(m, i-1);
    quicksort(i+1, n);
  }
}
main ()
{ readArray();
  a[0] = -9999;
  a[10] = 9999;
  quicksort(1,9);
}
```

Possible Activations

```
enter main()
  enter readArray()
  leave readArray()
  enter quicksort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quicksort(1,3)
      ...
    leave quicksort(1,3)
    enter quicksort(5,9)
      ...
    leave quicksort(5,9)
  leave quicksort(1,9)
leave main()
```

7.2.1 Activation Trees



Traversal of Activation Tree

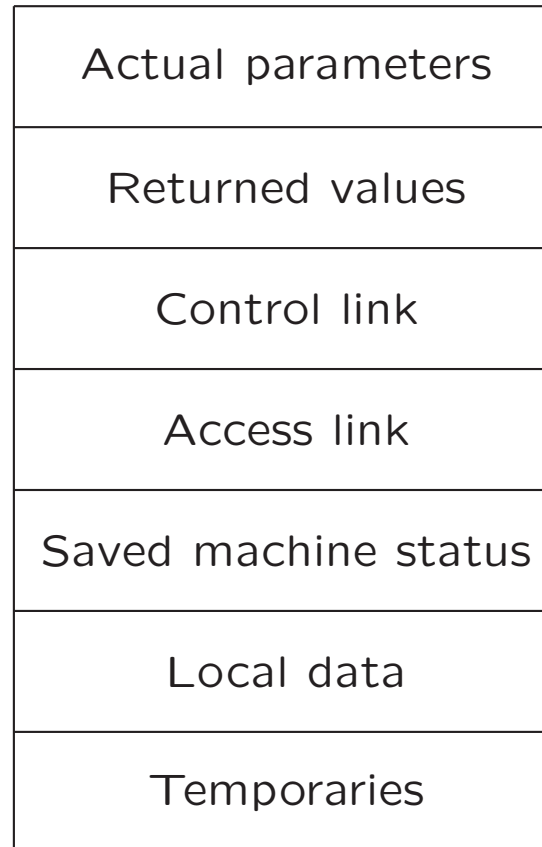
1. Sequence of procedure *calls* \approx ... traversal
2. Sequence of procedure *returns* \approx ... traversal
3. When control lies at particular node (\approx activation), the 'open' (*live*) activations are ...

Traversal of Activation Tree

1. Sequence of procedure *calls* \approx preorder traversal
2. Sequence of procedure *returns* \approx postorder traversal
3. When control lies at particular node (\approx activation), the 'open' (*live*) activations are on path from root

7.2.2. Activation Records

(= stack frames)

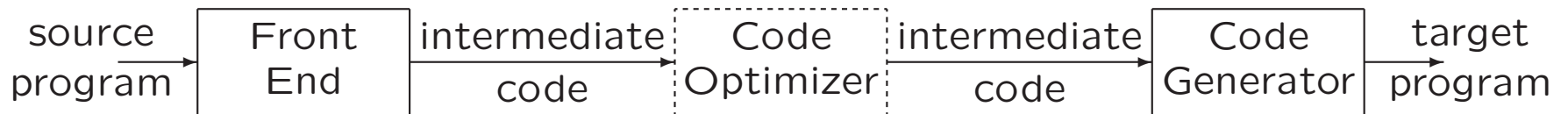


Possible (order of) elements of activation record

7.2.3 Calling Sequences

- Code to allocate (and fill) activation record on stack
- Divided between caller (at every location) and callee
- Return sequences analogous

8 Code Generation



- Output code must
 - be correct
 - use resources of target machine effectively
- Code generator must run efficiently

Generating optimal code is undecidable problem

Heuristics are available

8.1 Issues in Design of Code Generator

- Input to the code generator
- The target program
- Instruction selection
- Register allocation and assignment
- Evaluation order

8.1.1 Input to the Code Generator

- Intermediate representation of source program
 - Three-address representations (e.g., quadruples)
 - Virtual machine representations (e.g., bytecodes)
 - Postfix notation
 - Graphical representations (e.g., syntax trees and DAGs)
- Information from symbol table to determine run-time addresses
- Input is free of errors
 - Type checking and conversions have been done

8.1.2 The Target Program

- Common target-machine architectures
 - **RISC**: reduced instruction set computer
 - **CISC**: complex instruction set computer
 - **Stack-based**
- Possible output
 - Absolute machine code (executable code)
 - Relocatable machine code (object files for linker)
 - **Assembly-language**

8.1.3 Instruction Selection

- Given IR program can be implemented by many different code sequences
- Different machine instruction speeds
- Naive approach: statement-by-statement translation, with a code template for each IR statement

Example: $x = y + z$

Now, $a = b + c$ $d = a + e$

```
LD  R0, y
LD  R1, z
ADD R0, R0, R1
ST  x, R0
```

```
LD  R0, b
LD  R1, c
ADD R0, R0, R1
ST  a, R0
LD  R0, a
LD  R1, e
ADD R0, R0, R1
ST  d, R0
```


8.2 The Target Language

- Designing code generator requires understanding of target machine and its instruction set
- Our machine model
 - byte-addressable
 - has n general purpose registers $R0, R1, \dots, Rn - 1$
 - assumes operands are integers

Instructions of Target Machine

- Load operations: $LD\ dst, addr$
e.g., $LD\ r, x$ or $LD\ r_1, r_2$
- Store operations: $ST\ x, r$
- Computation operations: $OP\ dst, src_1, src_2$
e.g., $SUB\ r_1, r_2, r_3$
- Unconditional jumps: $BR\ L$
- Conditional jumps: $Bcond\ r, L$
e.g., $BLTZ\ r, L$

Addressing Modes of Target Machine

Form	Address	Example
r	r	LD R1, R2
x	x	LD R1, x
$a(r)$	$a + contents(r)$	LD R1, a(R2)
$c(r)$	$c + contents(r)$	LD R1, 100(R2)
$*r$	$contents(contents(r))$	LD R1, *R2
$*c(r)$	$contents(c + contents(r))$	LD R1, *100(R2)
$\#c$		LD R1, #100

Addressing Modes (Examples)

b = a[i]:

```
LD R1, i
MUL R1, R1, #8
LD R2, a(R1)
ST b, R2
```

x = *p

```
LD R1, p
LD R2, 0(R1)
ST x, R2
```

a[j] = c

```
LD R1, c
LD R2, j
MUL R2, R2, #8
ST a(R2), R1
```

if x < y goto L

```
LD R1, x
LD R2, y
SUB R1, R1, R2
BLTZ R1, M
```

8.2.2 Program and Instruction Costs

- Costs associated with compiling / running a program
 - Compilation time
 - Size, running time, power consumption of target program
- Finding optimal target problem: undecidable
- (Simple) cost per target-language instruction:
 - 1 + cost for addressing modes of operands
≈ length (in words) of instruction

Examples:

instruction	cost
LD R0, R1	1
LD R0, x	2
LD R1, *100(R2)	2

8.4 Basic Blocks and Flow Graphs

1. **Basic block:** maximal sequence of consecutive three-address instructions, such that
 - (a) Flow of control can only enter through first instruction of block
 - (b) Control leaves block without halting or branching
2. **Flow graph:** graph with
 - nodes: basic blocks
 - edges: indicate flow between blocks

8.4.1 Determining Basic Blocks

- Determine **leaders**
 1. First three-address instruction is leader
 2. Any instruction that is target of goto is leader
 3. Any instruction that immediately follows goto is leader
- For each leader, its basic block consists of leader and all instructions up to next leader (or end of program)

Determining Basic Blocks (Example)

Determine **leaders**

Pseudo code

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

Three-address code

```
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  $j \leq 10$  goto (3)
10)  $i = i + 1$ 
11) if  $i \leq 10$  goto (2)
12)  $i = 1$ 
13)  $t5 = i - 1$ 
14)  $t6 = 88 * t5$ 
15)  $a[t6] = 1.0$ 
16)  $i = i + 1$ 
17) if  $i \leq 10$  goto (13)
```


Determining Basic Blocks (Example)

Determine **leaders**

Pseudo code

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

Three-address code

```
→ 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  $j \leq 10$  goto (3)  
→ 10)  $i = i + 1$   
11) if  $i \leq 10$  goto (2)  
→ 12)  $i = 1$   
→ 13)  $t5 = i - 1$   
14)  $t6 = 88 * t5$   
15)  $a[t6] = 1.0$   
16)  $i = i + 1$   
17) if  $i \leq 10$  goto (13)
```

8.4.3 Flow Graphs

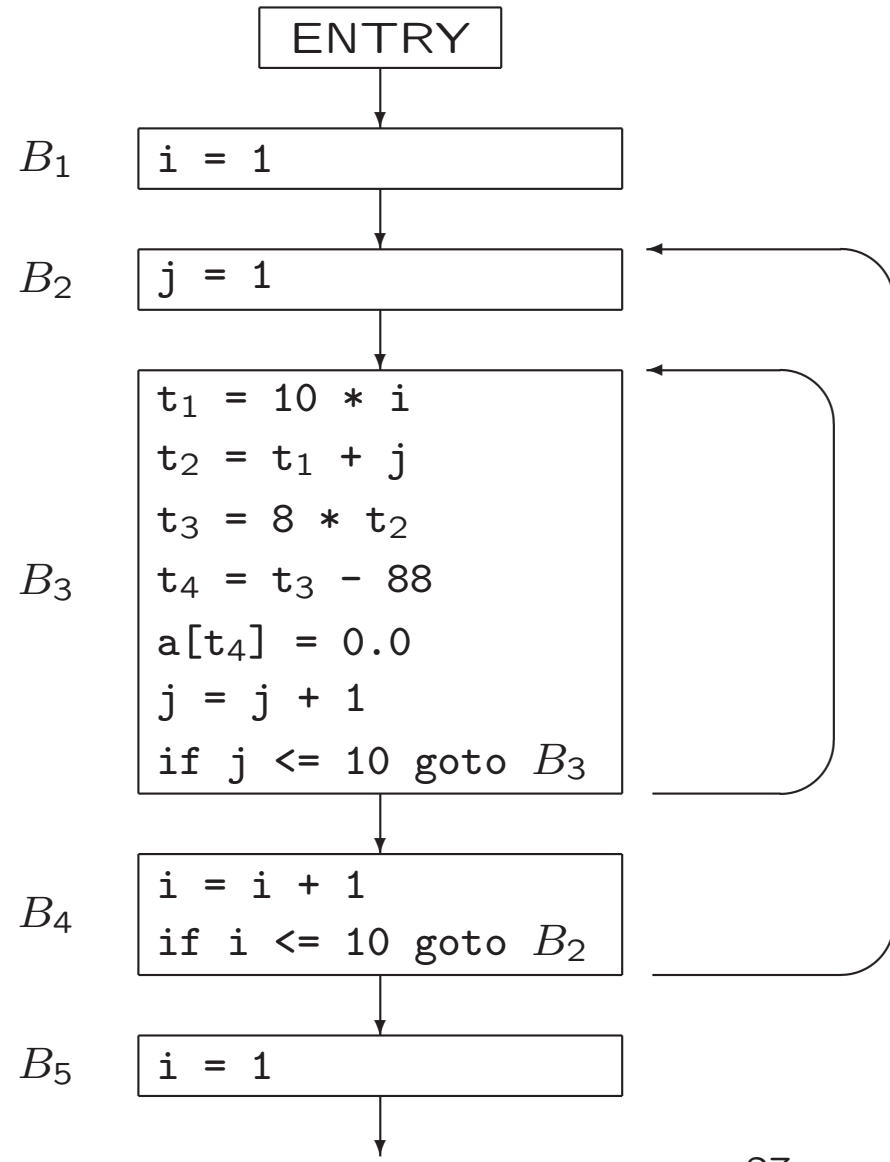
Edge from block B to block C

- if there is (un)conditional jump from end of B to beginning of C
- if C immediately follows B in original order, and B does not end in unconditional jump

Flow Graph (Example)

Three-address code

```
→ 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)
```



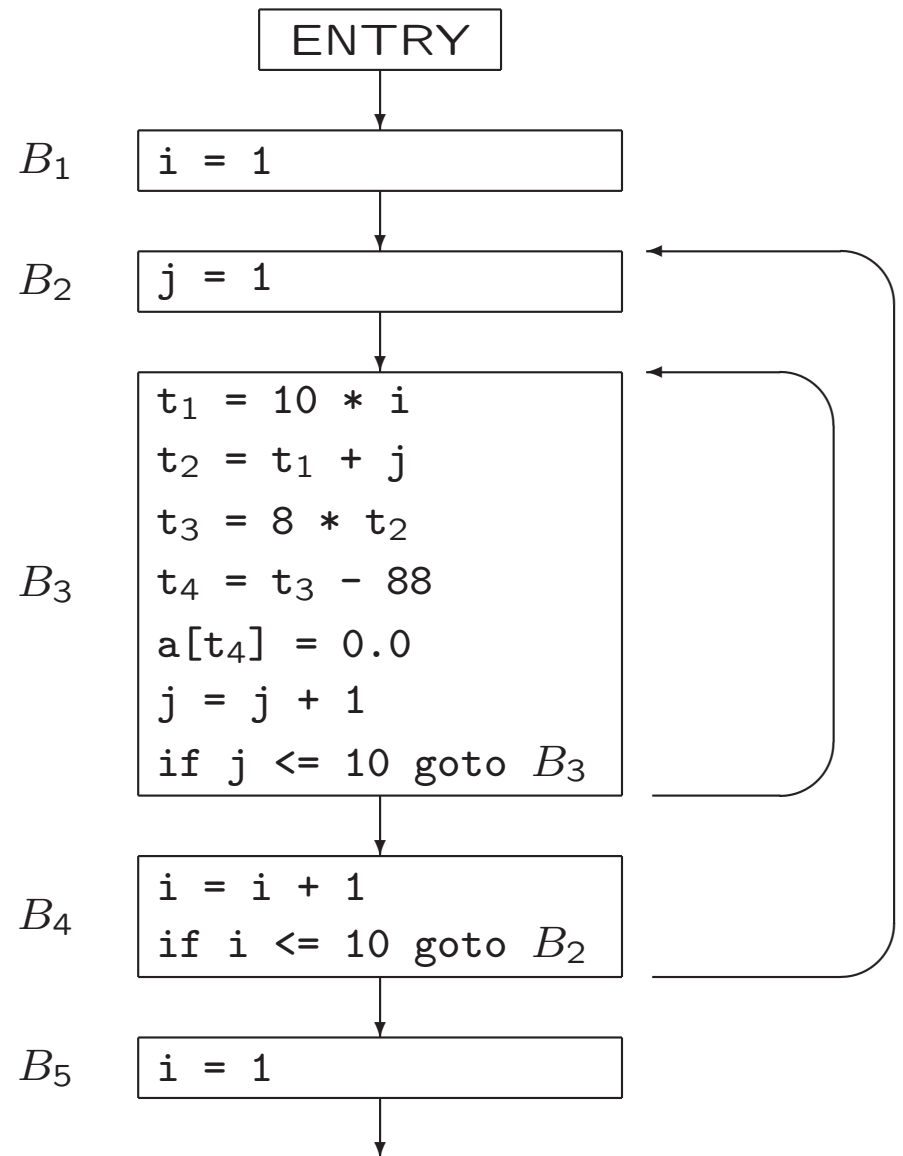
8.4.5 Loops

Loop is set of nodes in flow graph

- With unique loop entry e
- Every node in L has nonempty path in L to e

Example

- $\{B_3\}$, with loop entry B_3
- $\{B_2, B_3, B_4\}$, with loop entry B_2
- $\{B_6\}$, with loop entry B_6



8.4.2 Next-Use Information

- Next-use information is needed for **dead-code elimination** and **register assignment**

(i) $x = a * b$

...

(j) $z = c + x$

Instruction j **uses** value of x computed at i
 x is **live** at i ,
i.e., we need value of x later

- For each three-address statement $x = y \text{ op } z$ in block, record next-uses of x, y, z

Determining Next-Use Information

For **single** basic block

- Assume all non-temporary variables are **live on exit** (stored in symbol table)
- Make backward scan of instructions in block
- For each instruction $i: x = y \text{ op } z$
 1. Attach to i current next-use- and liveness information of x, y, z
 2. Set x to 'not live' and 'no next use'
 3. Set y and z to 'live'
Set 'next uses' of y and z to i

Determining Next-Use Information (Example)

1) $t = a - b$		NU(t) = ...	NU(a) = ...	NU(b) = ...
2) $u = a - c$		NU(u) = ...	NU(a) = ...	NU(c) = ...
3) $v = t + v$		NU(v) = ...	NU(t) = ...	
4) $a = d$		NU(a) = ...	NU(d) = ...	
5) $d = v + u$		NU(d) = ...	NU(v) = ...	NU(u) = ...

Assume all variables are non-temporary, and thus are live on exit

Next-Use information in symbol table:

	a	b	c	d	t	u	v
after line 5 (on exit)
before line 5				...			

. = live, but next use is not known

— = not live

i = next use in line i

Determining Next-Use Information (Example)

1) $t = a - b$		NU(t) = 3	NU(a) = 2	NU(b) = .
2) $u = a - c$		NU(u) = 5	NU(a) = -	NU(c) = .
3) $v = t + v$		NU(v) = 5	NU(t) = .	
4) $a = d$		NU(a) = .	NU(d) = -	
5) $d = v + u$		NU(d) = .	NU(v) = .	NU(u) = .

	a	b	c	d	t	u	v
after line 5 (on exit)
before line 5	.	.	.	-	.	5	5
before line 4	-	.	.	4	.	5	5
before line 3	-	.	.	4	3	5	3
before line 2	2	.	2	4	3	-	3
before line 1 (on entry)	1	1	2	4	-	-	3

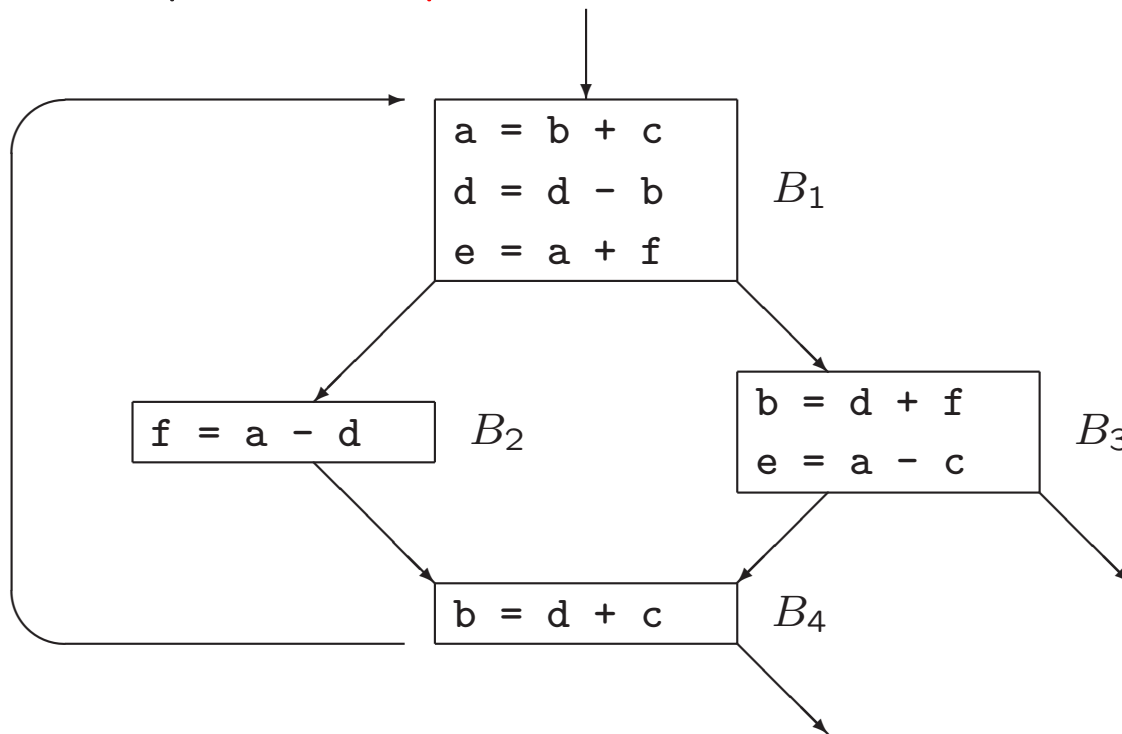
. = live, but next use is not known

- = not live

i = next use in line i

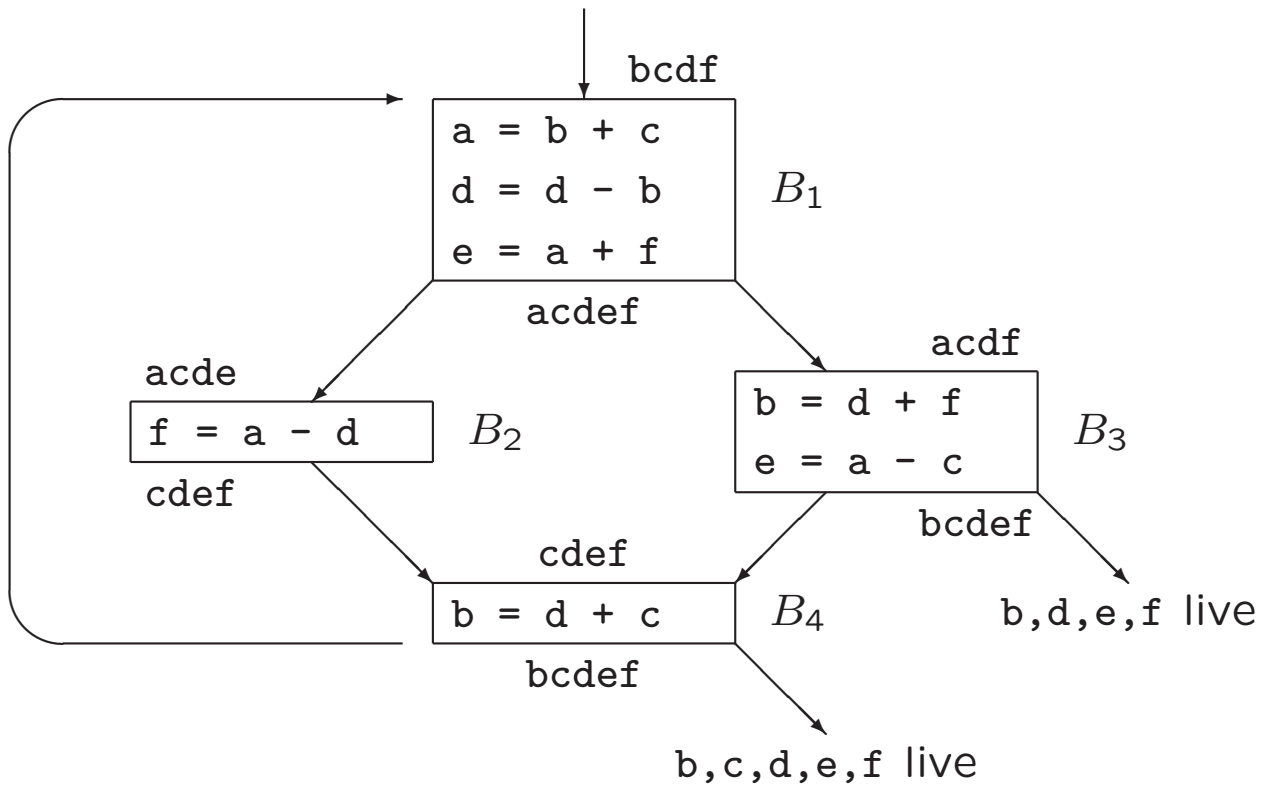
8.8.2 Passing Liveness Information over Blocks

Example of **loop**



Passing Liveness Information over Blocks

Example of **loop**



8.6 A Simple Code Generator

Use of registers

- Operands of operation must be in registers
- To hold values of temporary variables
- To hold (global) values that are used in several blocks
- To manage run-time stack

Assumption: subset of registers available for block

Machine instructions of form

- LD *reg, mem*
- ST *mem, reg*
- OP *reg, reg, reg*

8.6.1 Register and Address Descriptors

- **Register descriptor** keeps track of what is currently in register
 - Example:

$LD\ R, x \quad \rightarrow$ register R contains x

- Initially, all registers are empty

- **Address descriptor** keeps track of locations where current value of a variable can be found
 - Example:

$LD\ R, x \quad \rightarrow x$ is (also) in R

- Information stored in symbol table

8.6.2 The Code-Generation Algorithm

For each three-address instruction $x = y \text{ op } z$

1. Use $getReg(x = y \text{ op } z)$ to select registers R_x, R_y, R_z
2. If y is not in R_y , then issue instruction LD R_y, y' , where y' is a memory location for y (according to address descriptor)
3. If z is not in R_z , ...
4. Issue instruction $OP R_x, R_y, R_z$

Special case: $x = y \dots$

At end of block: store all variables that are live-on-exit and not in their memory locations (according to address descriptor)

Managing Register / Address Descriptors

1. For the instruction LD R, x, \dots
2. For the instruction ST x, R, \dots
3. For an operation like ADD R_x, R_y, R_z , implementing $x = y + z$,
 - (c) Remove R_x from addr. descr. of other variables
 - (d) Remove x from reg. descr. of other registers
 - (a) Change reg. descr. for R_x : only x
 - (b) Change addr. descr. for x : only in R_x (not in x itself!)
4. For the copy statement $x = y, \dots$

Managing Register / Address Descriptors

Example: $d = (a - b) + (a - c) + (a - c)$ $a = \dots$ old value of d

```
t = a - b
    LD R1, a
    LD R2, b
    SUB R2, R1, R2
u = a - c
    LD R3, c
    SUB R1, R1, R3
v = t + u
    ADD R3, R2, R1
a = d
    LD R2, d
d = v + u
    ADD R1, R3, R1

exit
    ST a, R2
    ST d, R1
```

R1	R2	R3	a	b	c	d	t	u	v
			a	b	c	d			

Managing Register / Address Descriptors

Example: $d = (a - b) + (a - c) + (a - c)$ $a = \dots$ old value of d

```

t = a - b
    LD R1, a
    LD R2, b
    SUB R2, R1, R2
u = a - c
    LD R3, c
    SUB R1, R1, R3
v = t + u
    ADD R3, R2, R1
a = d
    LD R2, d
d = v + u
    ADD R1, R3, R1

exit
    ST a, R2
    ST d, R1
    
```

R1	R2	R3	a	b	c	d	t	u	v
d	a	v	a,R2	b	c	d,R1			R3

8.6.3 Design of Function *getReg*

For each instruction $x = y \text{ op } z$

- To compute R_y
 1. If y is in register, $\rightarrow R_y$
 2. Else, if empty register available, $\rightarrow R_y$
 3. Else, select occupied register
 - For each register R and variable v in R
 - (a) If v is also somewhere else, then OK
 - (b) If v is x , and x is not z , then OK
 - (c) Else, if v is not used later, then OK
 - (d) Else, $\text{ST } v, R$ is required

Take R with smallest number of stores

In fact, . . .

Alternative Function *getReg*

For each instruction $x = y \text{ op } z$

- To compute R_y
 1. If y is in register, $\rightarrow R_y$
 2. Else, if empty register available, $\rightarrow R_y$
 3. Else, select occupied register
 - For each register R and variable v in R
 - (a) If v is also in other register, then OK
 - (b) Else, **if v is z , then not OK** (i.e., do not take R)
 - (c) Else, if v is x , then OK
 - (d) Else, if v is not used later, then OK
 - (e) Else, if v is also in own memory location, then add 1 to score of R (for future LD)
 - (f) Else, add 2 to score of R (for ST v, R and future LD)

Take R with smallest score

8.6.3 Design of Function *getReg*

For each instruction $x = y \text{ op } z$

- To compute R_y
 1. If y is in register, $\rightarrow R_y$
 2. Else, if empty register available, $\rightarrow R_y$
 3. Else, select occupied register
 - For each register R and variable v in R
 - (a) If v is also somewhere else, then OK
 - (b) If v is x , and x is not z , then OK
 - (c) Else, if v is not used later, then OK
 - (d) Else, $\text{ST } v, R$ is required

Take R with smallest number of stores

- To compute R_x , similar with few differences (which?)

8.6.3 Design of Function *getReg*

For each instruction $x = y \text{ op } z$

- To compute R_x
 1. If x is **only value** in register, $\longrightarrow R_x$
(also if x is y or z)
 2. Else, if empty register available, $\longrightarrow R_x$
 3. Else, select occupied register
 - For each register R and variable v in R
 - (a) If v is also somewhere else, then OK
(e.g., if v is y or z , just loaded)
 - (b) If v is x (also if x is y or z), then OK
 - (c) Else, if v is not used later, then OK
(v might also be y or z)
 - (d) Else, $\text{ST } v, R$ is required

Take R with smallest number of stores

Design of Function *getReg*

For each instruction $x = y$, choose $R_x = R_y$

Exercise 1

Addressing Modes of Target Machine

Form	Address	Example
x	x	LD R1, x
$a(r)$	$a + contents(r)$	LD R1, a(R2)
$\#c$		LD R1, #100

8.8 Register Allocation and Assignment

So far, live variables in registers are stored at end of block

Use of registers

- Operands of operation must be in registers
- To hold values of temporary variables
- To hold (global) values that are used in several blocks
- To manage run-time stack

8.8.2 Usage Counts

With x in register during loop L

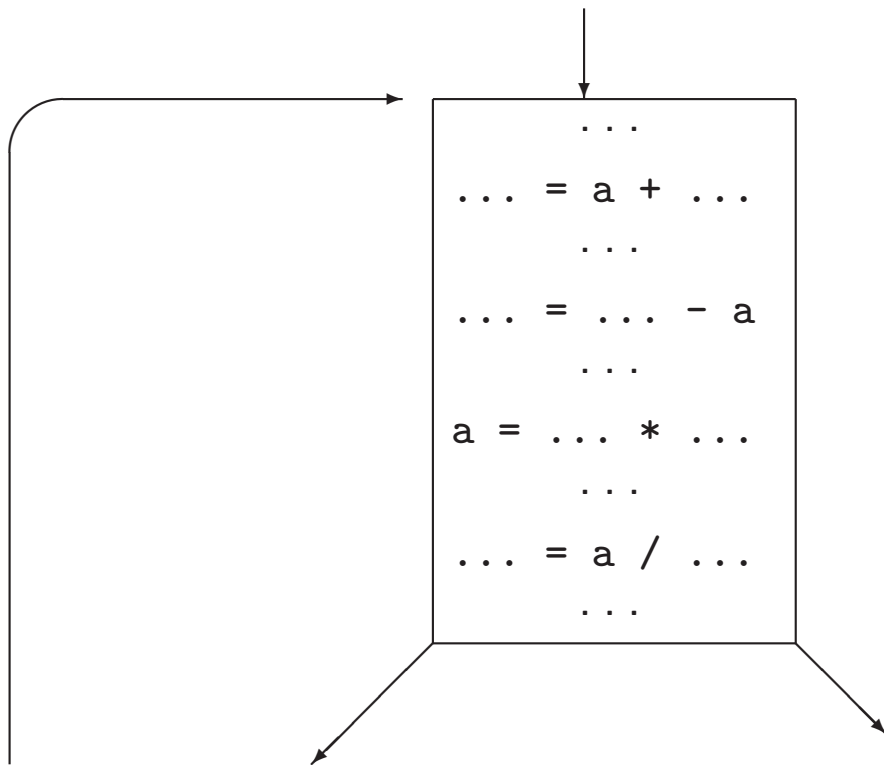
- Save ... for ... use of x that is not preceded by assignment in same block
- Save ... for each block, where x is assigned a value and x is live on exit

-

$$\text{Total savings} \approx \sum_{\text{blocks } B \in L} \dots$$

Choose variables x with largest savings

Savings for One Block



Usage Counts

With x in register during loop L

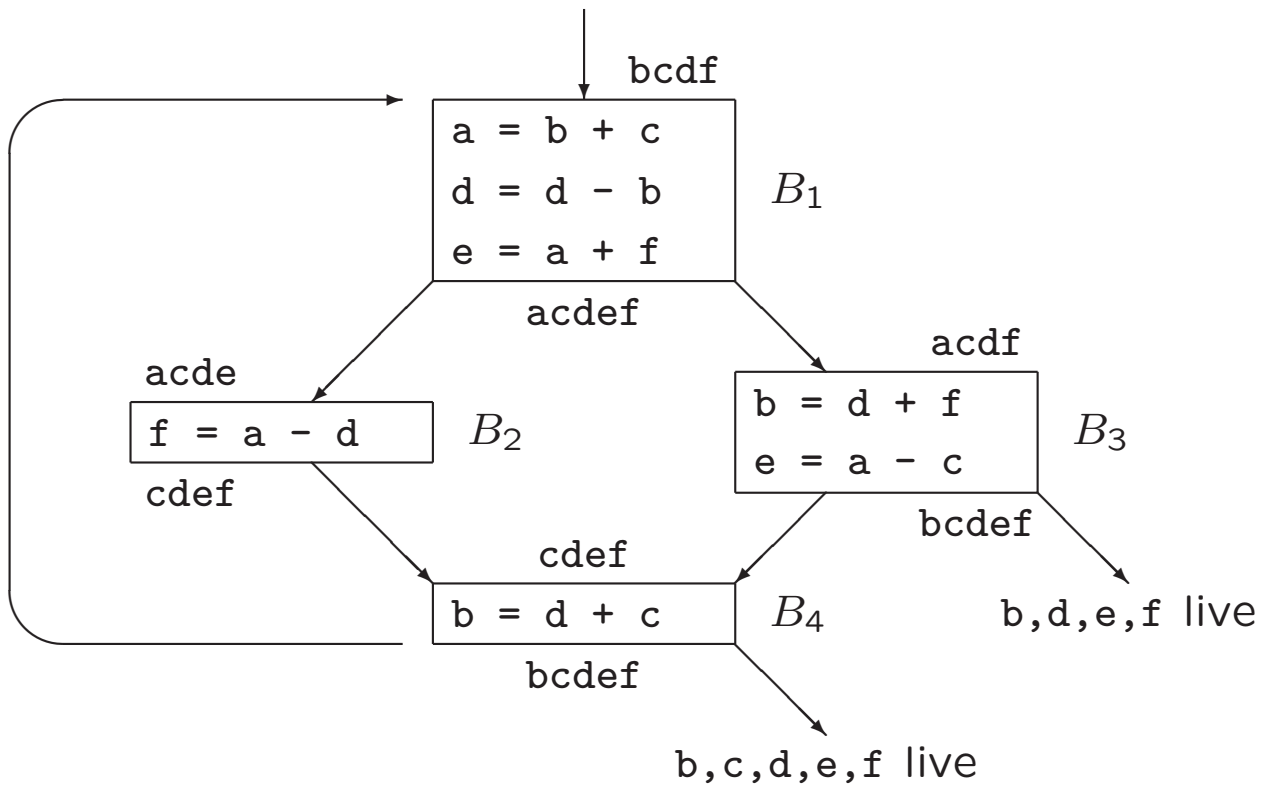
- Save 1 for each use of x that is not preceded by assignment in same block
- Save 2 for each block, where x is assigned a value and x is live on exit

-

$$\text{Total savings} \approx \sum_{\text{blocks } B \in L} \text{use}(x, B) + 2 * \text{live}(x, B)$$

Choose variables x with largest savings

Usage Counts (Example)



Savings for a are $1 + 1 + 1 * 2 = 4$

Komende week

- Woensdag 22 november, 11.00–12.45: practicum
- Donderdag 23 november: inleveren opdracht 3
- Vrijdag 24 november, 11.00–12.45: hoorcollege
+ introductie opdracht 4 (inleveren 14 december)
- Vrijdag 24 november, 13:30–...: werkcollege

Compiler constructie

college 8

Storage Organization

Code Generation

Chapters for reading:

7.1, 7.2–7.2.3

8.intro, 8.1, 8.2, 8.4, 8.6, 8.8–8.8.2