Compilerconstructie

najaar 2012

`http://www.liacs.nl/home/rvvliet/coco/`

Rudy van Vliet
kamer 124 Snellius, tel. 071-527 5777
rvvliet(at)liacs.nl

college 7, dinsdag 6 november 2012

Code Generation

Code Generator Position in a Compiler

- 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

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

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

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, $a = b + c \quad d = a + e$

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

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

Target Machine

- Designing code generator requires understanding of target machine and its instruction set

- Our machine model
  - byte-addressable
  - has n general purpose registers $\texttt{R0}, \texttt{R1}, \ldots, \texttt{R}n-1$
  - 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: `B`*cond* 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)`
$*r$	$contents(r)$	`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
```

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

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 * 1
 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)
```

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 * 1
    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

Edge from block B to block C

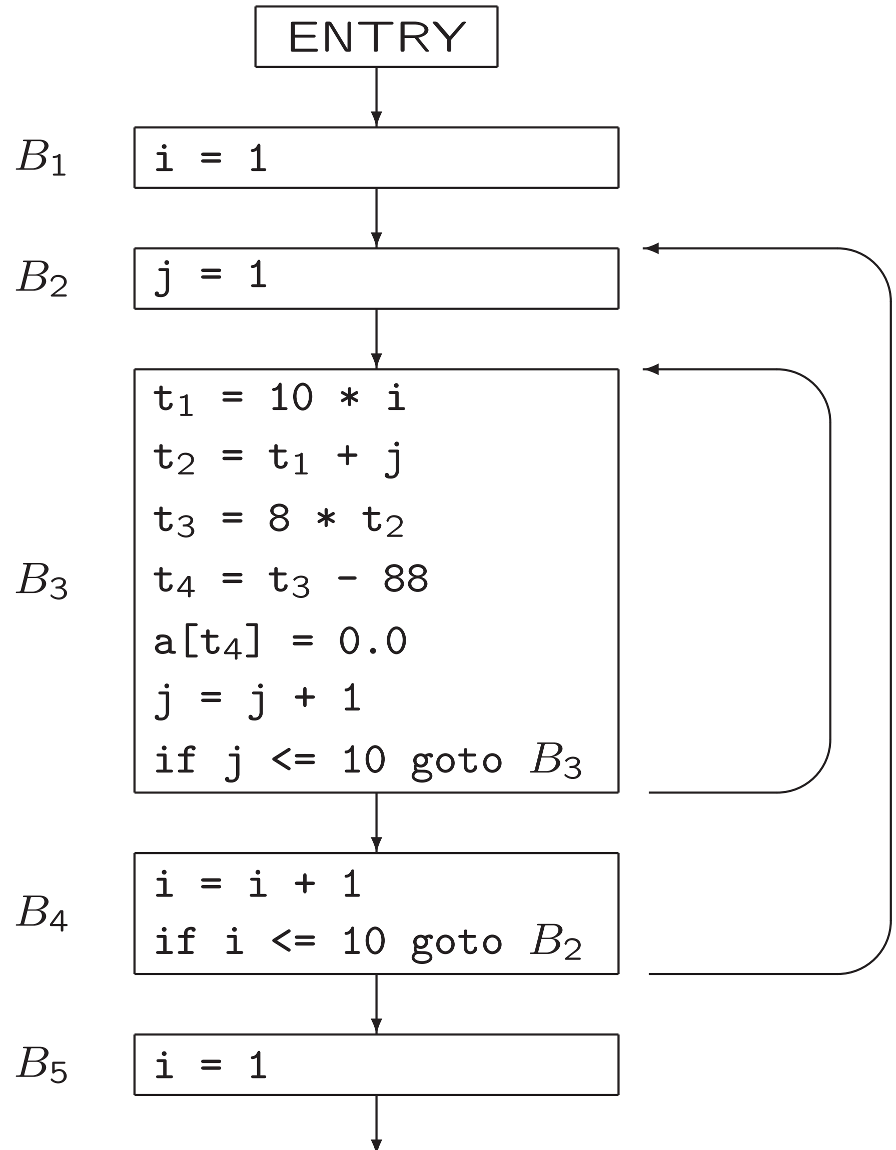
- if there is (un)conditional jump from end of B to beginning of C

- 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 * 1
    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)
```



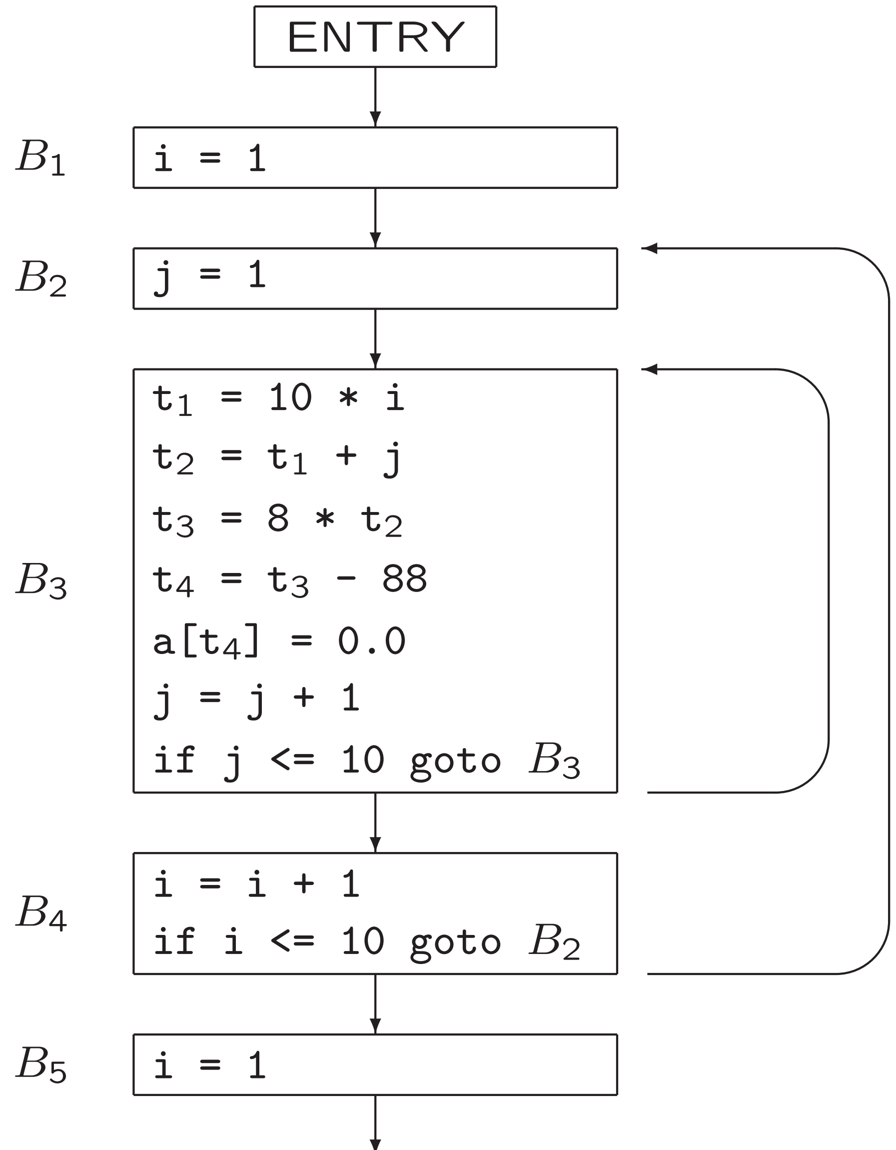
Loops in Flow Graph

Loop is set of nodes

- 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



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 $x = y \ op \ z$ in block, record next-uses of x, y, z

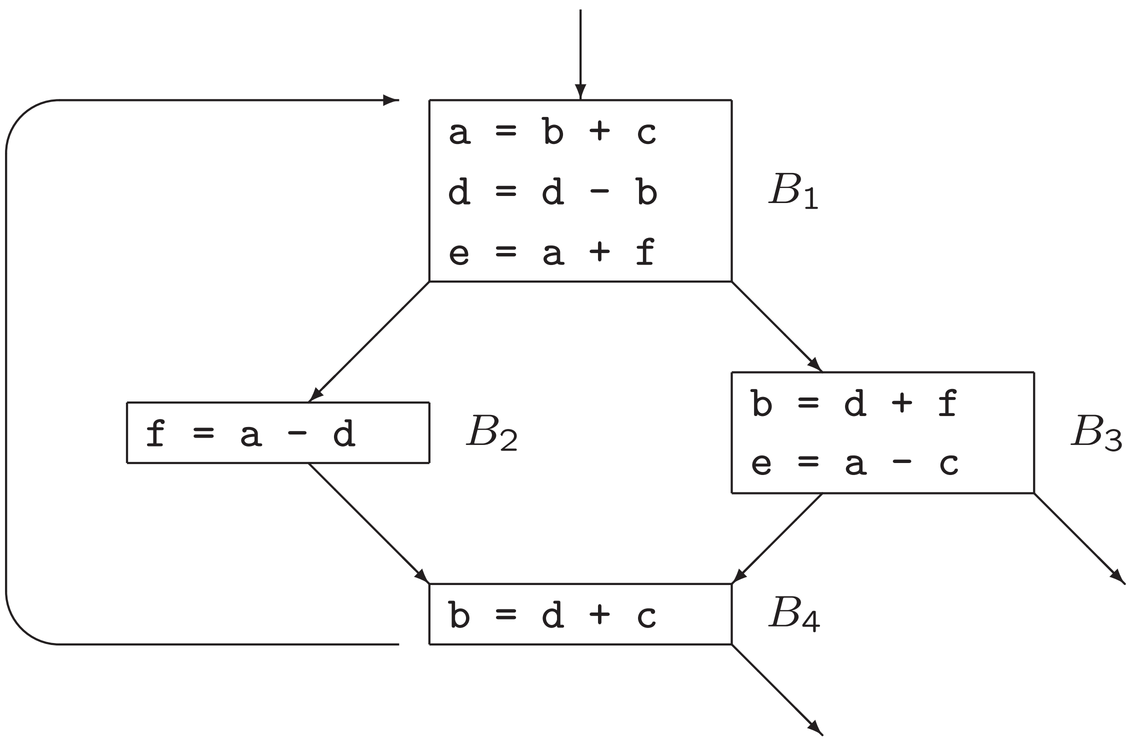
Determining Next-Use Information

For single basic block

- Assume all non-temporary variables are live on exit
- Make backward scan of instructions in block
- For each instruction i: $x = y$ *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

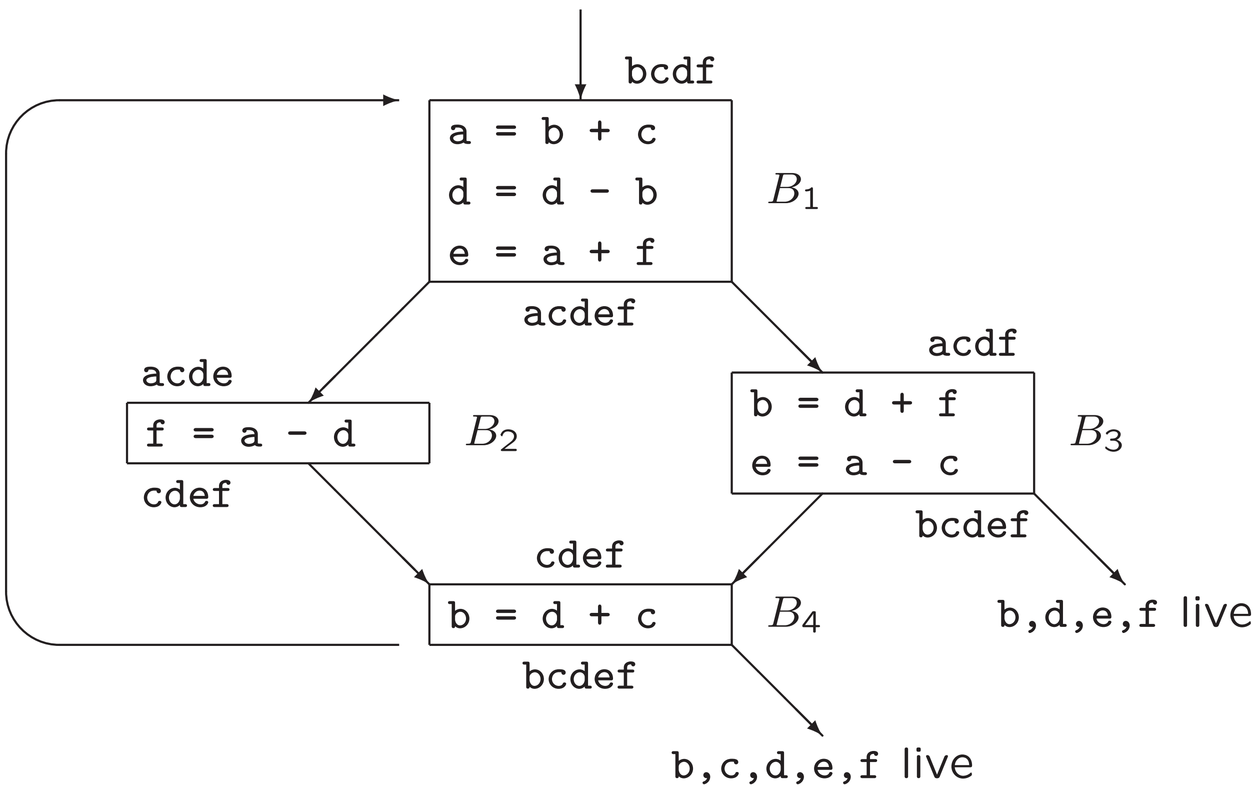
Passing Liveness Information over Blocks

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*

Register and Address Descriptors

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

    $$\texttt{LD}\ R, x \qquad \rightarrow \text{register } R \text{ contains } x$$

  - Initially, all registers are empty

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

    $$\texttt{LD}\ R, x \qquad \rightarrow x \text{ is (also) in } R$$

  - Information stored in symbol table

The Code-Generation Algorithm

For each three-address instruction $x = y$ *op* z

1. Use *getReg*($x = y$ *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

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

Description in book

Example: $d = (a - b) + (a - c) + (a - c) \quad a = \ldots$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
```

Function *getReg*

For each instruction $x = y$ *op* z

- To compute R_y
  1. If y is in register, $\longrightarrow R_y$
  2. Else, if empty register available, $\longrightarrow R_y$
  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, `ST` v, R is required

     Take R with smallest number of stores

Function *getReg*

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

- To compute R_x, similar with few differences

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

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

Usage counts

With x in register during loop L

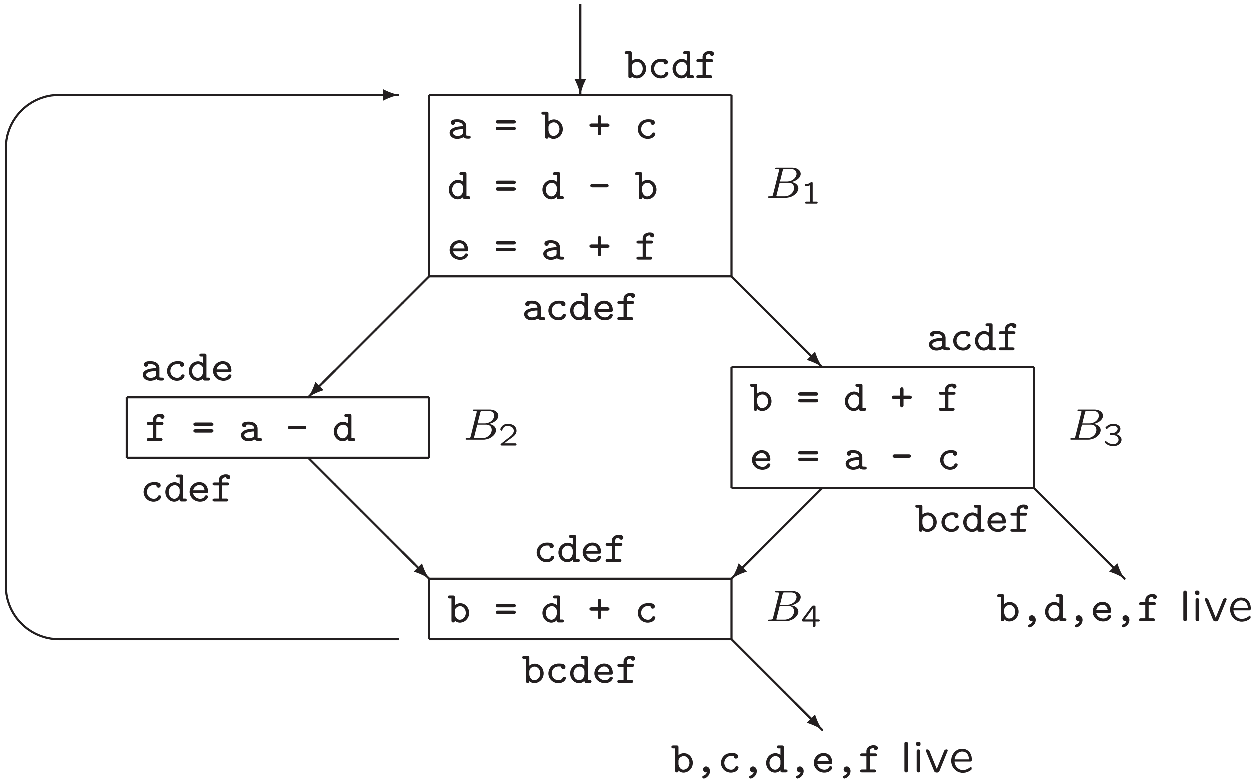
- Save 1 for each use of x that is not preceded by assignment in same block

- Save 2 for each block, where x is assigned a value and x is live on exit

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

Choose variables x with largest savings

Usage counts (Example)



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

8.5 Optimization of Basic Blocks

To improve running time of code

- Local optimization: within block

- Global optimization: across blocks

Local optimization benefits from DAG representation of basic block

DAG Representation of Basic Blocks

1. A node for initial value of each variable appearing in block

2. A node N for each statement s in block
   Children of N are nodes corresponding to last definitions of operands used by s

3. Node N is labeled by operator applied at s
   N has list of variables for which s is last definition in block

Example:

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

Local Common Subexpression Elimination

- Use value-number method to detect common subexpressions
- Remove redundant computations

Example:

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

Local Common Subexpression Elimination

- Use value-number method to detect common subexpressions

- Remove redundant computations

Example:

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

Dead Code Elimination

- Remove roots with no live variables attached

- If possible, repeat

Example:

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

No common subexpression

If c and e are not live. . .

Dead Code Elimination

- Remove roots with no live variables attached

- If possible, repeat

Example:

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

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

No common subexpression

If c and e are not live. . .

Algebraic Transformations

(see assignment 3)

Algebraic identities:

$$\begin{aligned} x + 0 &= 0 + x &= x \\ x * 1 &= 1 * x &= x \end{aligned}$$

Reduction in strength:

$$\begin{aligned} x^2 &= x * x & \text{(cheaper)} \\ 2 * x &= x + x & \text{(cheaper)} \\ x/2 &= x * 0.5 & \text{(cheaper)} \end{aligned}$$

Constant folding:

$$2 * 3.14 = 6.28$$

Algebraic Transformations

Common subexpressions resulting from commutativity / associativity of operators:

$$\begin{aligned} x * y &= y * x \\ c + d + b &= (b + c) + d \end{aligned}$$

Common subexpressions generated by relational operators:

$$x > y \Leftrightarrow x - y > 0$$

8.7 Peephole Optimization

- Examines short sequence of instructions in a window (peep-hole) and replace them by faster/shorter sequence

- Applied to intermediate code or target code

- Typical optimizations
  - Redundant instruction elimination
  - Eliminating unreachable code
  - Flow-of-control optimization
  - Algebraic simplification
  - Use of machine idioms

Redundant Instruction Elimination

Example:

```
ST  a, R0
LD  R0, a
```

Eliminating Unreachable Code

Example:

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

Eliminating Unreachable Code

Example:

```
      if debug != 1 goto L2
  L1: print debugging information
  L2:
```

If `debug` is set to 0 at beginning of program, . . .

Flow-of-Control Optimizations

Example:

```
     goto L1
     ...
L1:  goto L2
```

Compiler constructie

college 7
Code Generation

Chapters for reading:
8.intro, 8.1, 8.2, 8.4, 8.5–8.5.4, 8.6–8.8