Lecture 11 – Code Generation

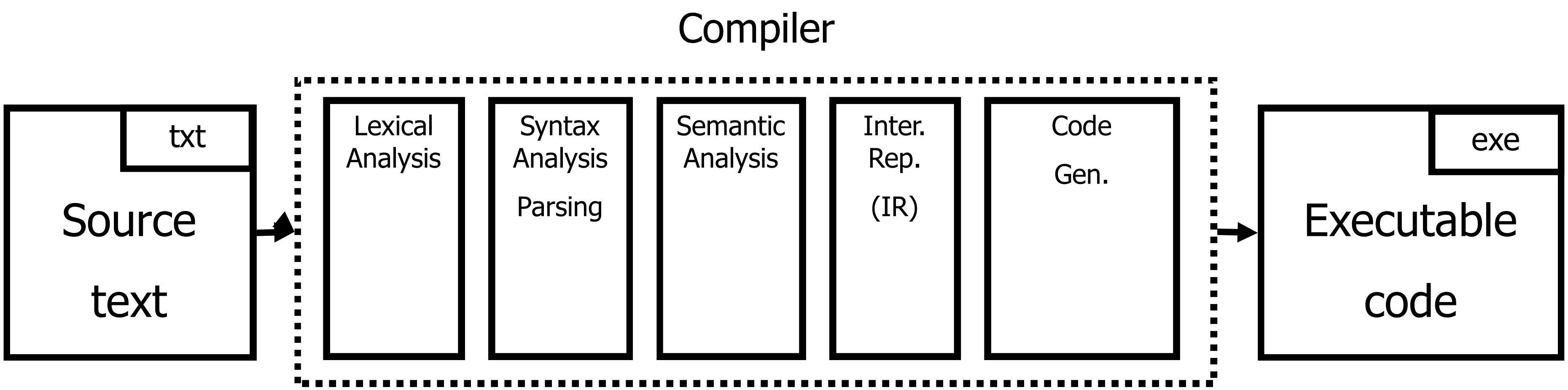
THEORY OF COMPILATION

Eran Yahav

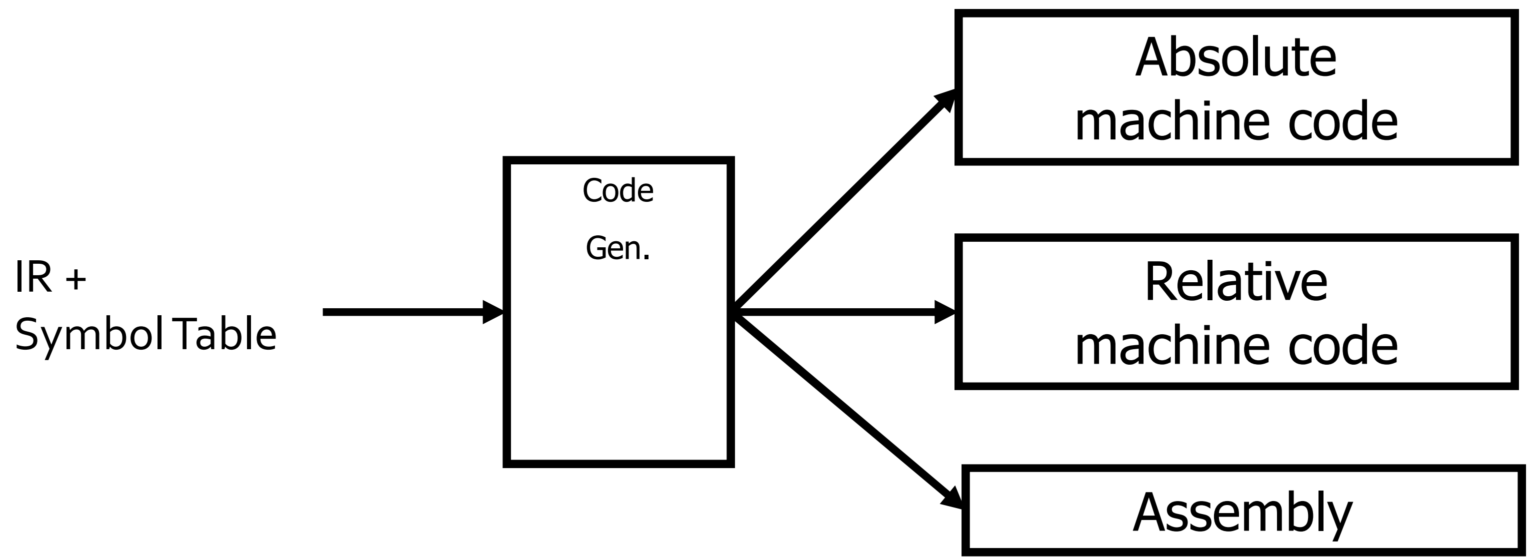
www.cs.technion.ac.il/~yahave/tocs2011/compilers-lec11.pptx

Reference: Dragon 8. MCD 4.2.4

You are here


Compiler
txt
Source
text
Lexical
Analysis
Syntax
Analysis
Parsing
Semantic
Analysis
Inter.
Rep.
(IR)
Code
Gen.
exe
Executable
code

target languages



From IR to ASM: Challenges

- mapping IR to ASM operations
  - what instruction(s) should be used to implement an IR operation?
  - how do we translate code sequences
- call/return of routines
  - managing activation records
- memory allocation
- register allocation
- optimizations

Intel IA-32 Assembly

- Going from Assembly to Binary…
  - Assembling
  - Linking

- AT&T syntax vs. Intel syntax
- We will use AT&T syntax
  - matches GNU assembler (GAS)

IA-32 Registers

- Eight 32-bit general-purpose registers
  - EAX – accumulator for operands and result data.
    Used to return value from function calls.
  - EBX – pointer to data. Often use as array-base address
  - ECX – counter for string and loop operations
  - EDX – I/O pointer (GP for us)
  - ESI – GP and source pointer for string operations
  - EDI – GP and destination pointer for string operations
  - EBP – stack frame (base) pointer
  - ESP – stack pointer
- EFLAGS register
- EIP (instruction pointer) register
- Six 16-bit segment registers
- … (ignore the rest for our purposes)

Not all registers are born equal

- EAX
  - Required operand of MUL,IMUL,DIV and IDIV instructions
  - Contains the result of these operations
- EDX
  - Stores remainder of a DIV or IDIV instruction
    (EAX stores quotient)
- ESI, EDI
  - ESI – required source pointer for string instructions
  - EDI – required destination pointer for string instructions
- Destination Registers of Arithmetic operations
  - EAX, EBX, ECX, EDX
- EBP – stack frame (base) pointer
- ESP – stack pointer

IA-32 Addressing Modes

- Machine-instructions take zero or more operands
- Source operand
  - Immediate
  - Register
  - Memory location
  - (I/O port)
- Destination operand
  - Register
  - Memory location
  - (I/O port)

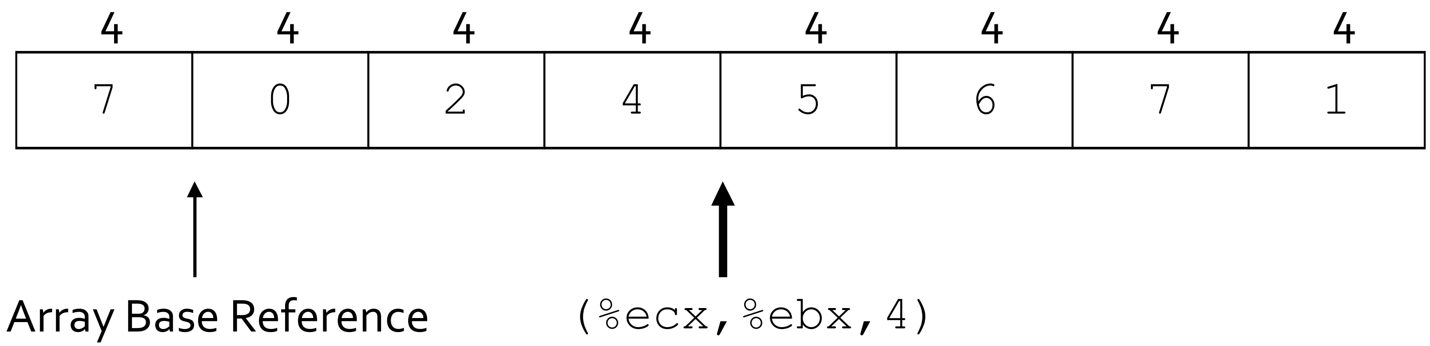
Immediate and Register Operands

- Immediate
  - Value specified in the instruction itself
  - GAS syntax – immediate values preceded by `$`
  - add `$4`, %esp
- Register
  - Register name is used
  - GAS syntax – register names preceded with %
  - mov %esp,%ebp

Memory and Base Displacement Operands

- Memory operands
  - Value at given address
  - GAS syntax - parentheses
  - mov (%eax), %eax
- Base displacement
  - Value at computed address
  - Address computed out of
    - base register, index register, scale factor, displacement
  - offset = base + (index*scale) + displacement
  - Syntax: disp(base,index,scale)
  - movl $42, $2(%eax)
  - movl $42, $1(%eax,%ecx,4)

Base Displacement Addressing




```
Mov (%ecx,%ebx,4), %eax
```

```
%ecx = base
%ebx = 3
```

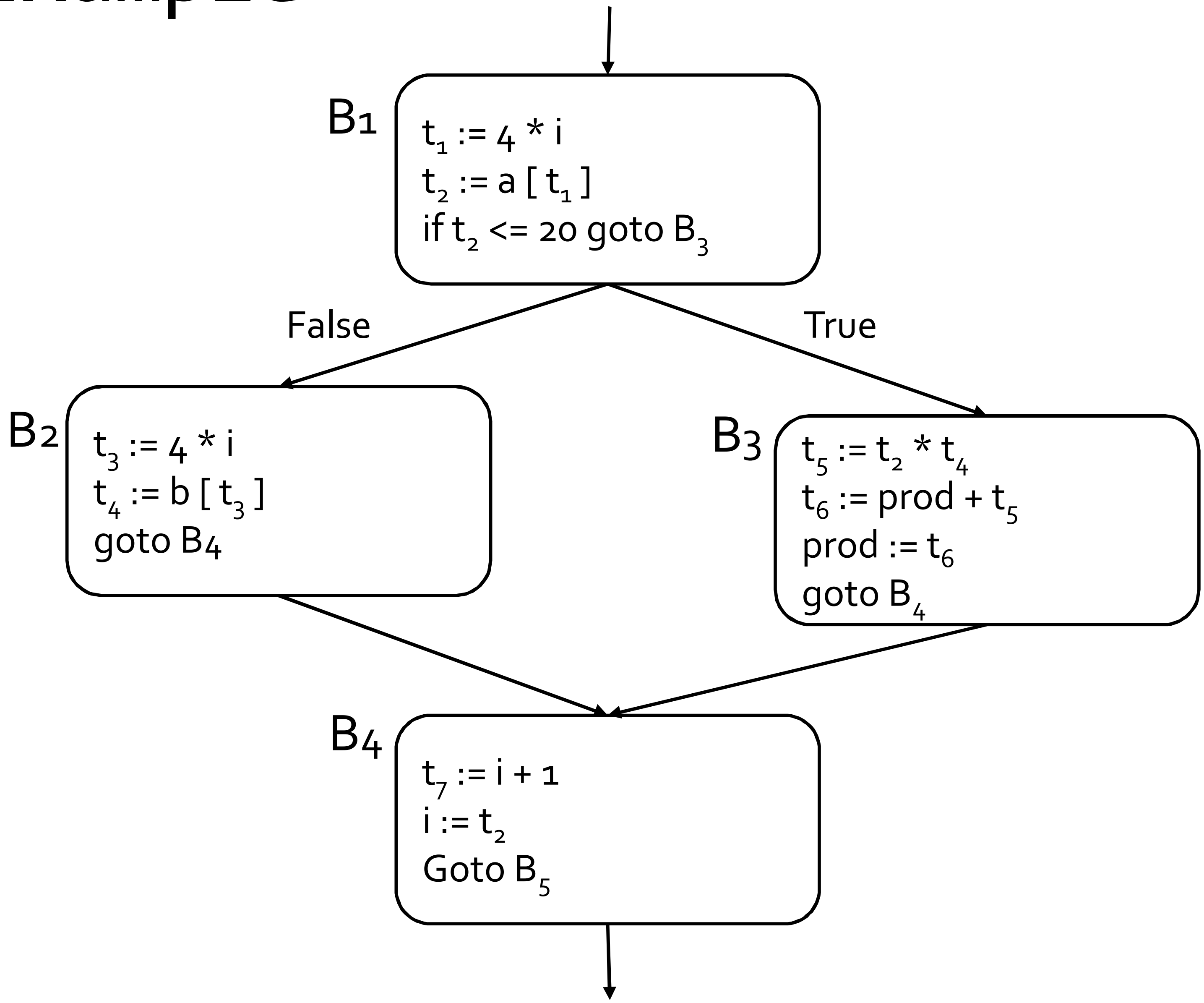
offset = base + (index*scale) + displacement

offset = base + (3*4) + 0 = base + 12

How do we generate the code?

- break the IR into basic blocks
- basic block is a sequence of instructions with
  - single entry (to first instruction), no jumps to the middle of the block
  - single exit (last instruction)
  - code execute as a sequence from first instruction to last instruction without any jumps
- edge from one basic block B1 to another block B2 when the last statement of B1 may jump to B2

Example



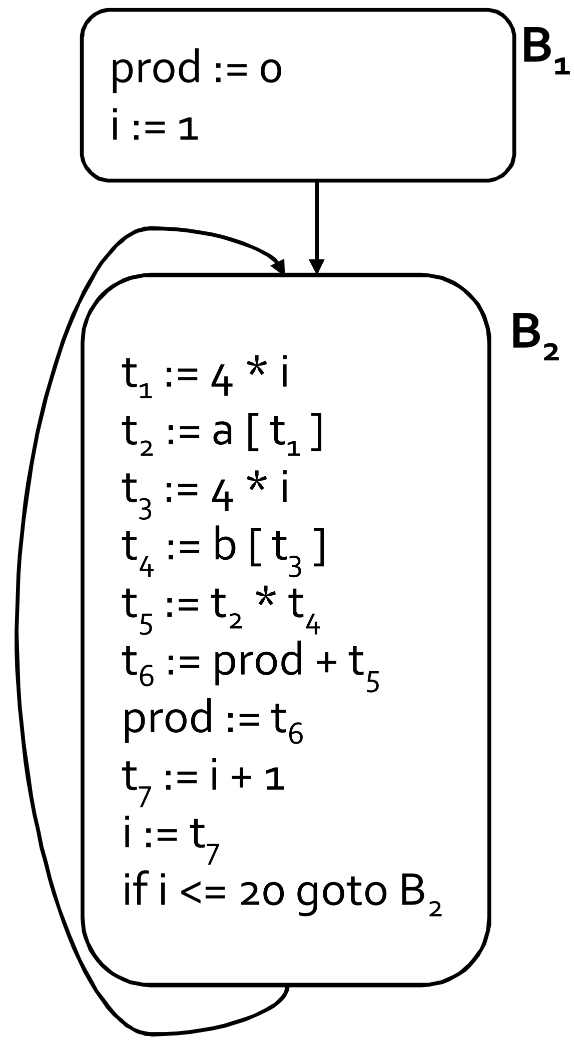
B1
t1 := 4 * i
t2 := a [t1]
if t2 <= 20 goto B3
False
True
B2
t3 := 4 * i
t4 := b [t3]
goto B4
B3
t5 := t2 * t4
t6 := prod + t5
prod := t6
goto B4
B4
t7 := i + 1
i := t2
Goto B5

creating basic blocks

- Input: A sequence of three-address statements
- Output: A list of basic blocks with each three-address statement in exactly one block
- Method
  - Determine the set of leaders (first statement of a block)
    - The first statement is a leader
    - Any statement that is the target of a conditional or unconditional jump is a leader
    - Any statement that immediately follows a goto or conditional jump statement is a leader
  - For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

control flow graph

- A directed graph G=(V,E)
- nodes V = basic blocks
- edges E = control flow
  - (B1,B2) $\in$E when control from B1 flows to B2



example

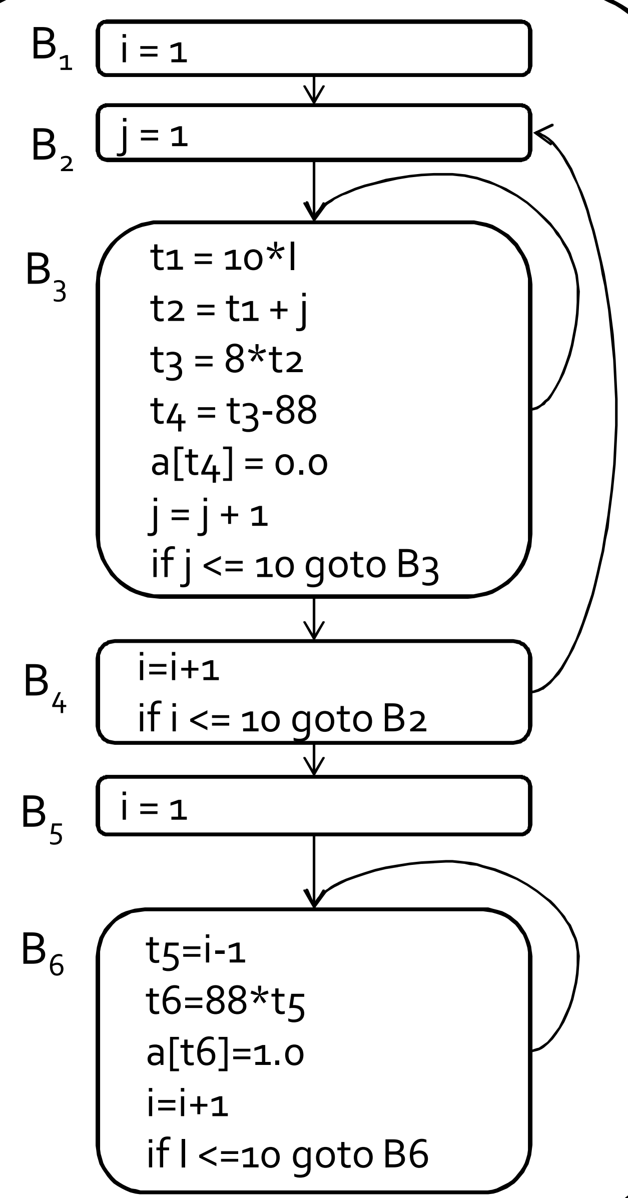
source

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;

IR

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)

CFG



Variable Liveness

- A statement x = y + z
  - defines x
  - uses y and z
- A variable x is live at a program point if its value is used at a later point

```
y = 42
z = 73
x = y + z
print(x);
```

x undef, y live, z undef

x undef, y live, z live

x is live, y dead, z dead

x is dead, y dead, z dead

(showing state after the statement)

Computing Liveness Information

- between basic blocks – dataflow analysis (next lecture)

- within a single basic block?
- idea
  - use symbol table to record next-use information
  - scan basic block backwards
  - update next-use for each variable

Computing Liveness Information

- INPUT: A basic block B of three-address statements. symbol table initially shows all non-temporary variables in B as being live on exit.

- OUTPUT: At each statement i: x = y + z in B, liveness and next-use information of x, y, and z at i.

- Start at the last statement in B and scan backwards
  - At each statement i: x = y + z in B, we do the following:

1. Attach to i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
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 i

Computing Liveness Information

- Start at the last statement in B and scan backwards
  - At each statement i: x = y + z in B, we do the following:

1. Attach to i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
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 i

```
x = 1
y = x + 3
z = x * 3
x = x * z
```

can we change the order between 2 and 3?

common-subexpression elimination

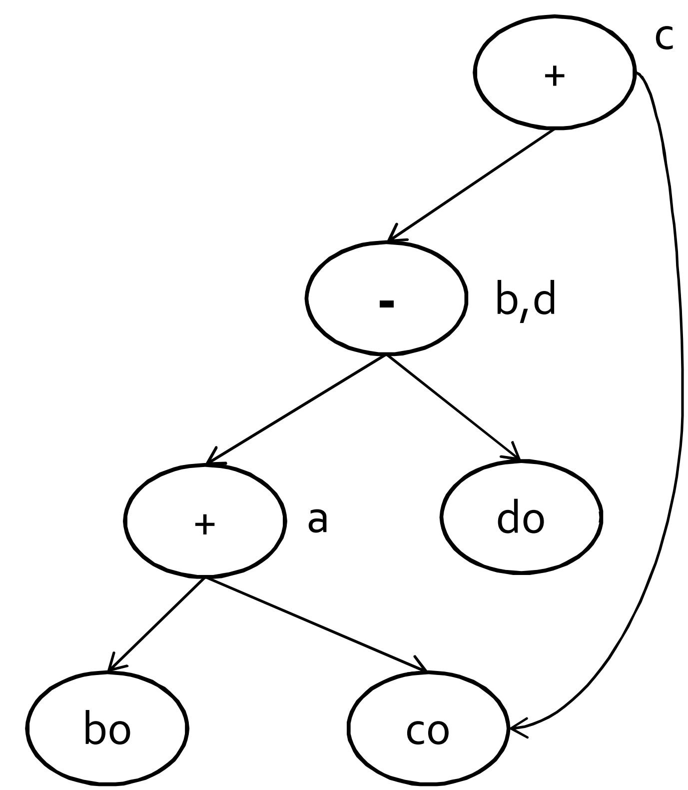
- common-subexpression elimination

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

```
a = b + c
b = a – d
c = b + c
d = b
```

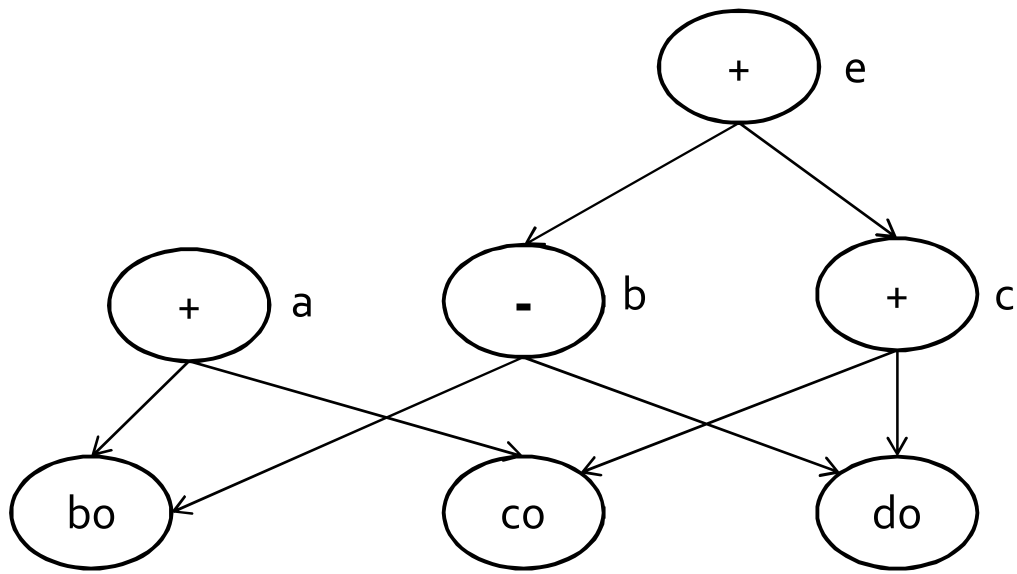
DAG Representation of Basic Blocks

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



DAG Representation of Basic Blocks

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



algebraic identities

```
a = x^2
b = x*2
c = x/2
d = 1*x
```

```
a = x*x
b = x+x
c = x*0.5
d = x
```

coming up next

- register allocation

The End