Compiler construction in4303 – lecture 9

Code generation

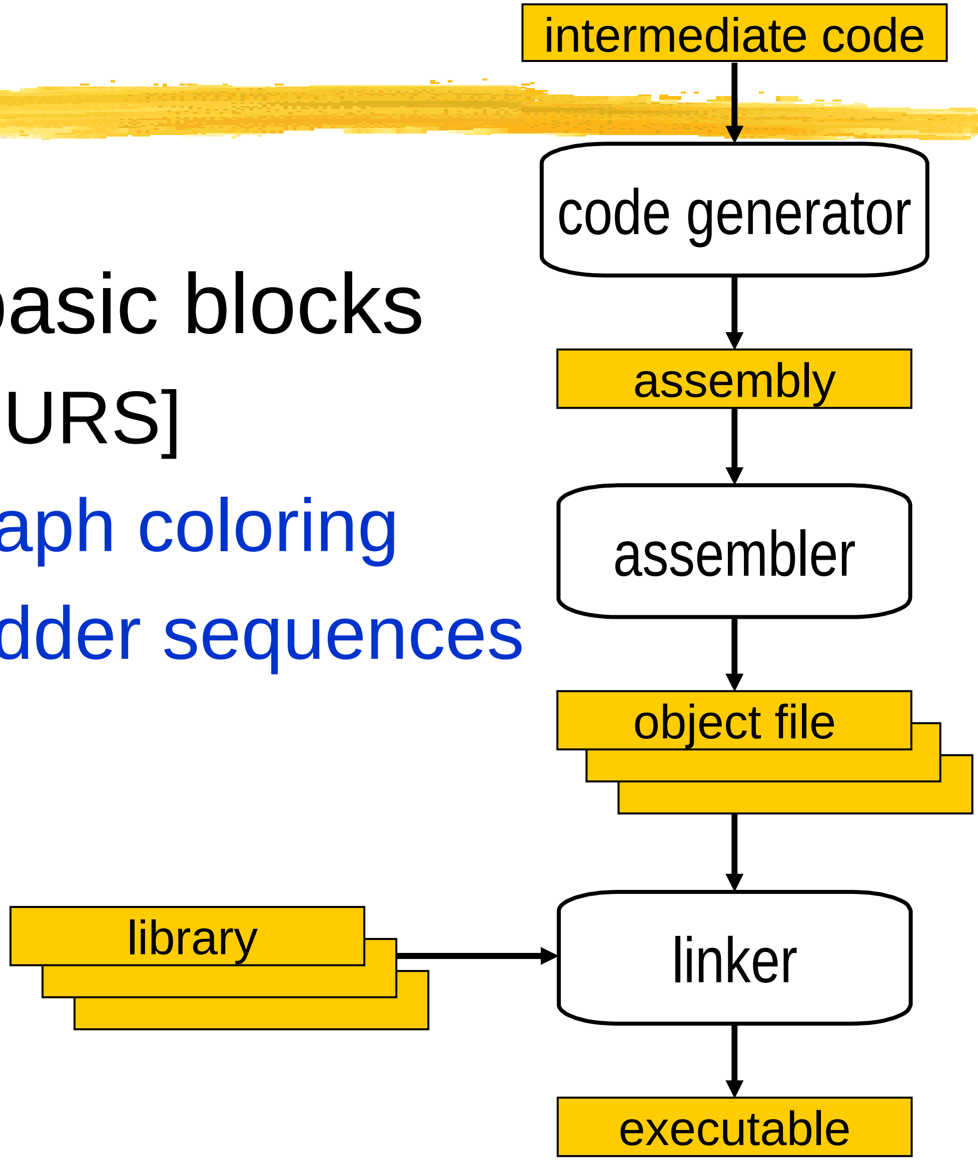
Chapter 4.2.5, 4.2.7, 4.2.11 – 4.3

Overview

Code generation for basic blocks

- instruction selection:[BURS]
- register allocation: graph coloring
- instruction ordering: ladder sequences

Assemblers & linkers



Code generation for basic blocks

- improve quality of code emitted by simple code generation
- consider multiple AST nodes at a time

> basic block: a part of the control graph that contains no splits (jumps) or combines (labels)

- generate code for maximal basic blocks that cannot be extended by including adjacent AST nodes

Code generation for basic blocks

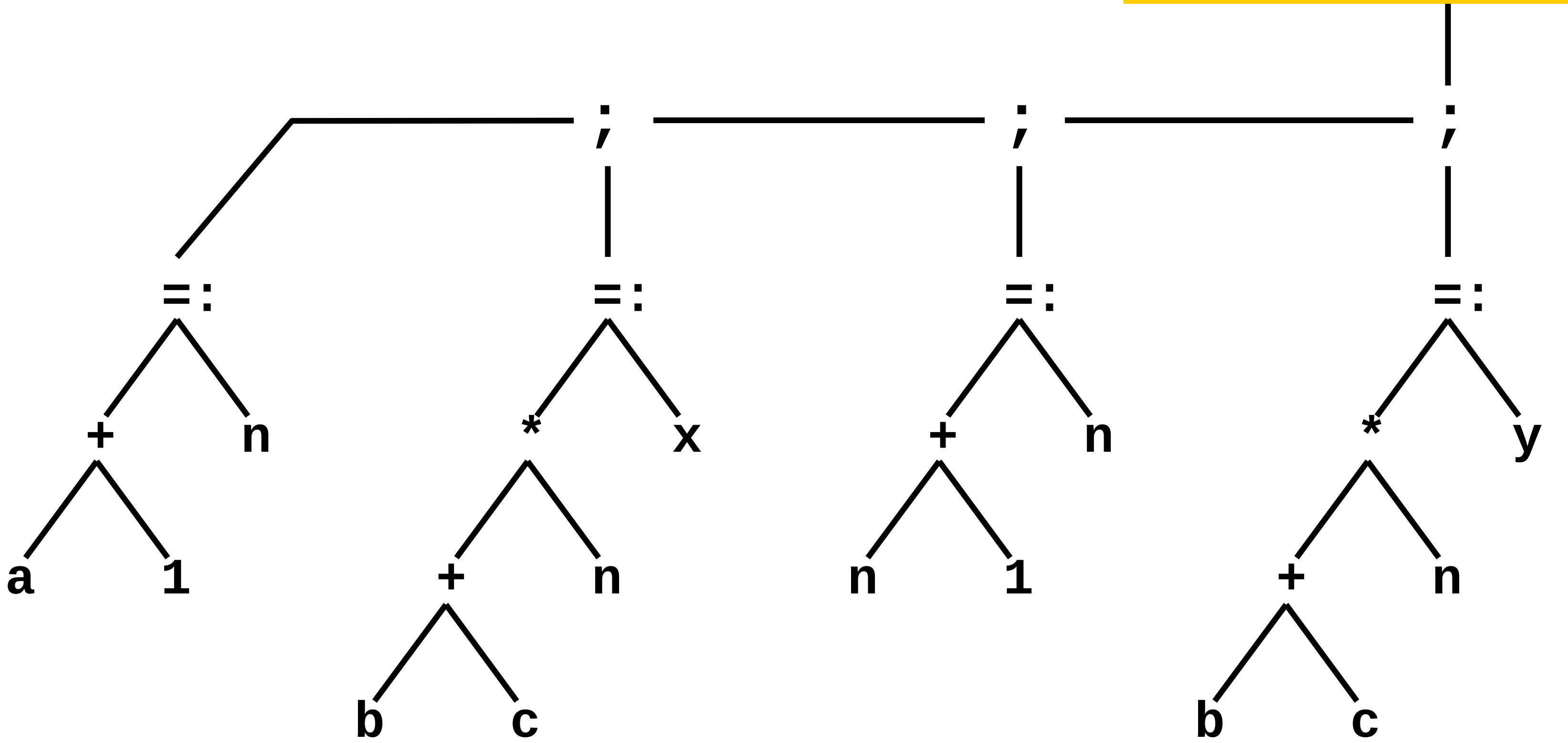
- a basic block consists of expressions and assignments

```
{  int n;

   n = a+1;
   x = (b+c) * n;
   n = n+1;
   y = (b+c) * n;
}
```

- fixed sequence (;) limits code generation
- an AST is too restrictive

Example AST

```
{  int n;

   n = a+1;
   x = (b+c) * n;
   n = n+1;
   y = (b+c) * n;
}
```



Dependency graph

- convert AST to a directed acyclic graph (dag) capturing essential data dependencies
  - data flow inside expressions:

    operands must be evaluated before operator is applied
  - data flow from a value assigned to variable V to the use of V:

    the usage of V is not affected by other assignments

AST to dependency graph

AST

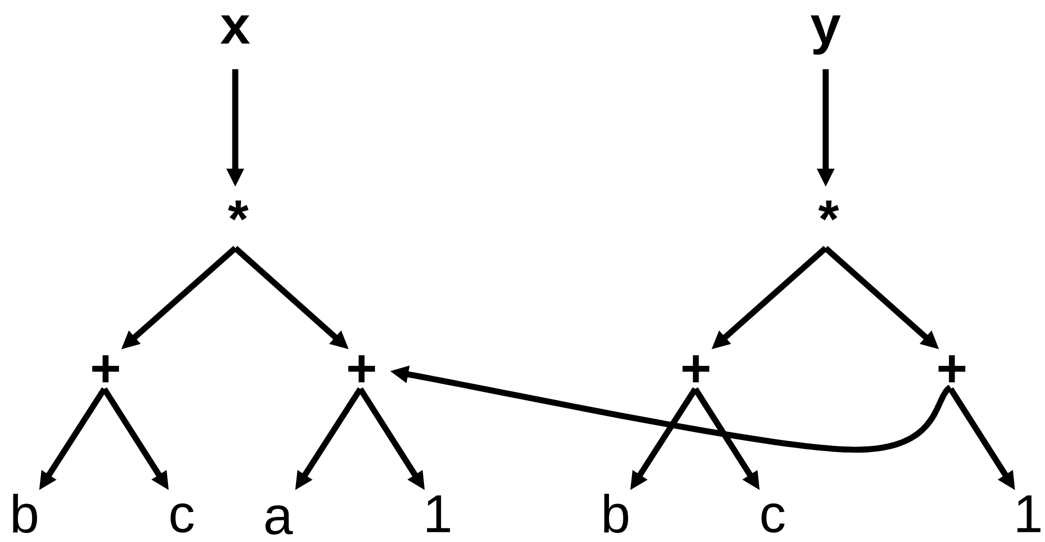
- replace arcs by downwards arrows (upwards for destination under assignment)
- insert data dependencies from use of V to preceding assignment to V
- insert data dependencies from each assignment to V to all previous **uses**
- add roots to the graph (output variables)
- remove **;**-nodes and connecting arrows

check errata

dependency graph

Example dependency graph

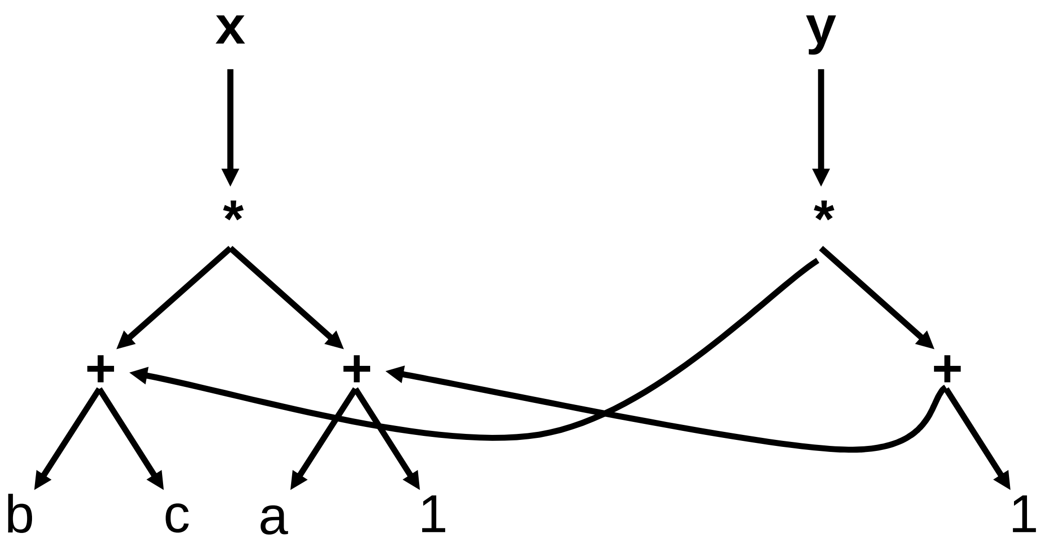
```
{  int n;

   n = a+1;
   x = (b+c) * n;
   n = n+1;
   y = (b+c) * n;
}
```



Common subexpression elimination

- common subexpressions occur multiple times and evaluate to the same value

```
{  int n;

   n = a+1;
   x = (b+c) * n;
   n = n+1;
   y = (b+c) * n;
}
```



Exercise (5 min.)

- given the code fragment

```
x := a*a + 2*a*b;
y := a*a – 2*a*b;
```

draw the dependency graph before and after common subexpression elimination.

Answers

Recognizing CSEs

Problem: how do we identify nodes with the same type (operator) and dependencies (operands)?

$O(n^2)$

Solution: hash table with a hash function based on operator and operands.

$O(n)$

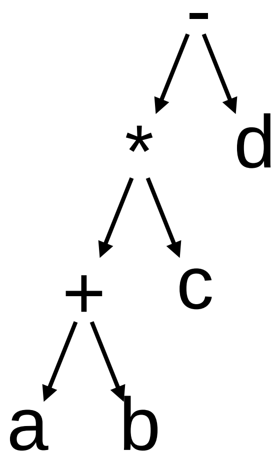
From dependency graph to code

- target: register machine (lecture 8) with additional operations on memory
  - reg op:= reg — Add_Reg R2, R1
  - reg op:= mem — Add_Mem x, R1

- rewrite nodes with machine instruction templates, and linearize the result
  - instruction ordering: — ladder sequences
  - register allocation: — graph coloring

Linearization of the data dependency graph

- example:

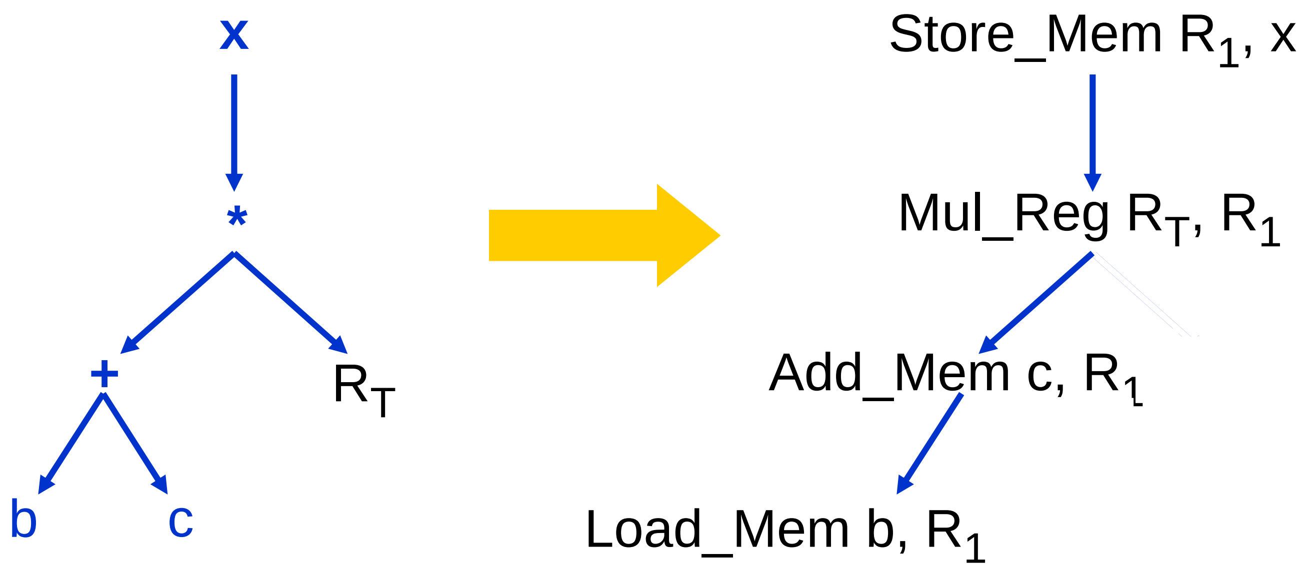



```
Load_Mem a, R1
Add_Mem b, R1
Mul_Mem, c, R1
Sub_Mem d, R1
```

- definition of a ladder sequence
  - each root node is a ladder sequence
  - a ladder sequence S ending in operator node N can be extended with the left operand of N
  - if operator N is communitative then S may also extended with the right operand of N

Linearization of the data dependency graph

- code generation for a ladder sequence

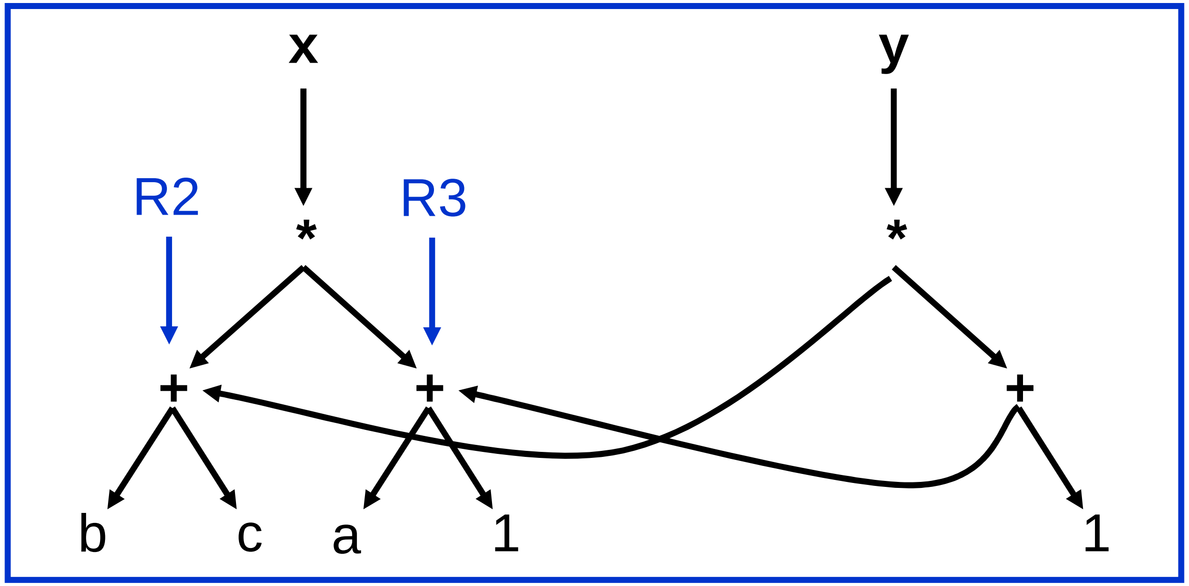


- instructions from bottom to top, one register

Linearization of the data dependency graph

- late evaluation – don't occupy registers
  - select ladder sequence S without additional incoming dependencies
  - introduce temporary registers for non-leaf operands, which become additional roots
  - generate code for S, using R1 as the ladder register
  - remove S from the graph
- note: code blocks produced in reverse order

Example code generation



1) ladder: y, *, +

```
Load_Const 1, R1
Add_Reg R3, R1
Mul_Reg, R2, R1
Store_Mem R1, y
```

2) ladder: x, *

```
Load_Reg R2, R1
Mul_Reg R3, R1
Store_Mem R1, x
```

3) ladder: R2, +

```
Load_Mem b, R1
Add_Mem c, R1
Load_Reg R1, R2
```

4) ladder: R3, +

```
Load_Const 1, R1
Add_Mem a, R1
Load_Reg R1, R3
```

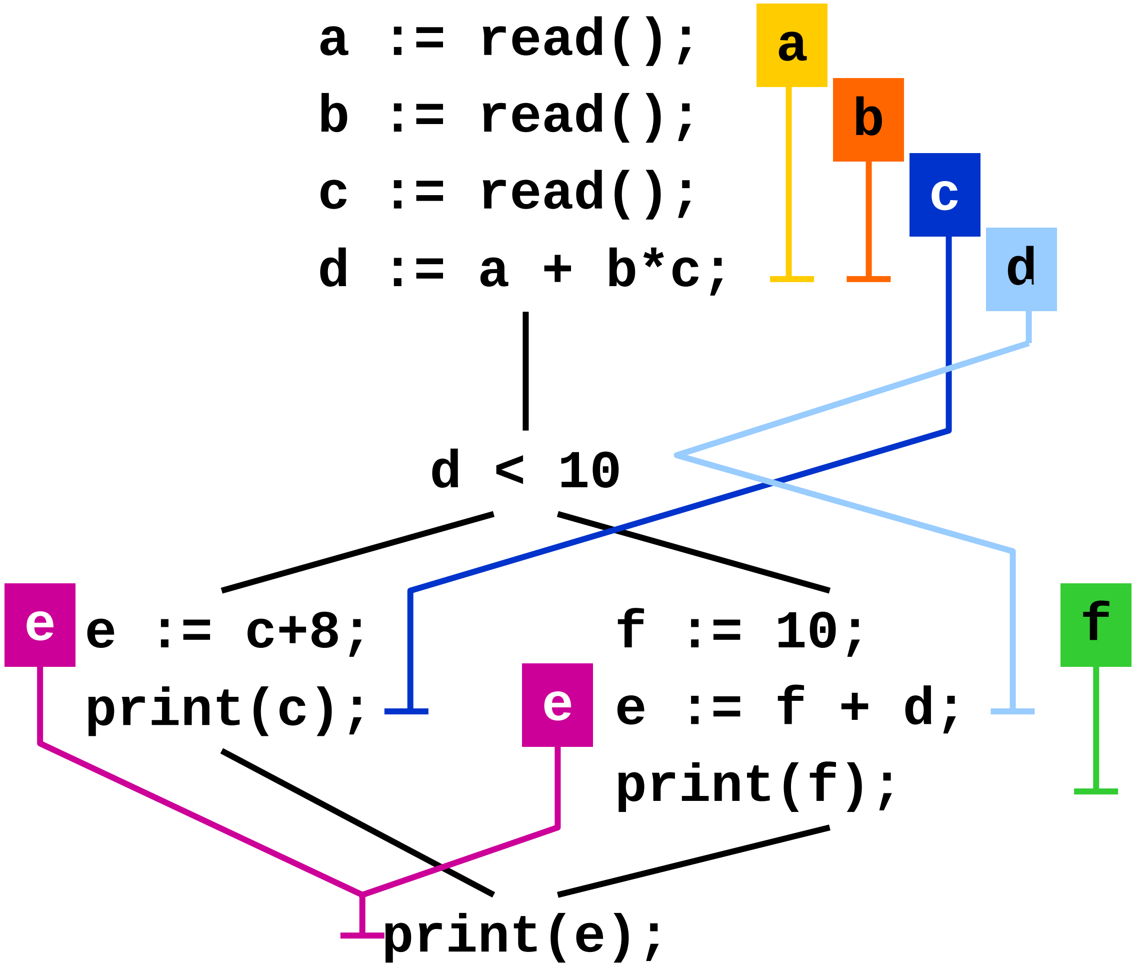
Register allocation by graph coloring

- procedure-wide register allocation
- only live variables require register storage

> **dataflow analysis**: a variable is live at node N if the value it holds is used on some path further down the control-flow graph; otherwise it is dead

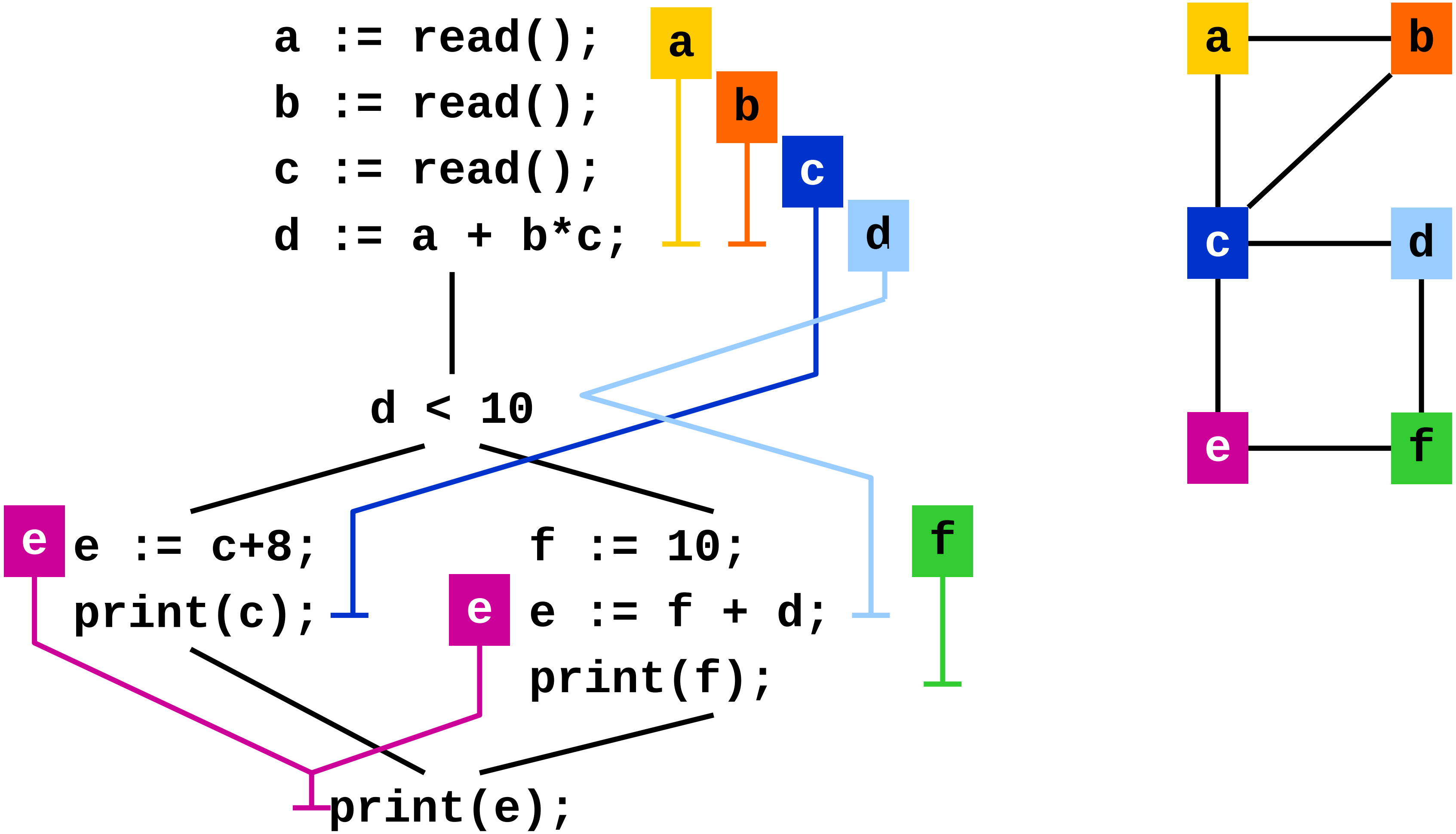
- two variables(values) interfere when their live ranges overlap

Live analysis



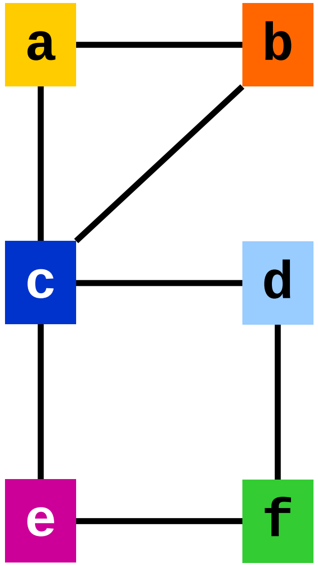

```
a := read();
b := read();
c := read();
d := a + b*c;
if (d < 10 ) then
   e := c+8;
   print(c);
else
   f := 10;
   e := f + d;
   print(f);
fi
print(e);
```

Register interference graph



Graph coloring

- NP complete problem
- heuristic: color easy nodes last
  - find node *N* with lowest degree
  - remove *N* from the graph
  - color the simplified graph
  - set color of *N* to the first color that is not used by any of *N*'s neighbors



Exercise (7 min.)

```
{   int tmp_2ab = 2*a*b;
    int tmp_aa = a*a;
    int tmp_bb = b*b;

    x := tmp_aa + tmp_2ab + tmp_bb;
    y := tmp_aa - tmp_2ab + tmp_bb;
}
```

given that a and b are live on entry and dead on exit, and that x and y are live on exit:

(a) construct the register interference graph

(b) color the graph; how many register are needed?

Answers

Code optimization

- preprocessing
  - constant folding `a[1]` ≡ `*(a+4*1)` ⇒ `*(a+4)`
  - strength reduction `4*i` ⇒ `i<<2`
  - in-lining
  - ...
- postprocessing
  - peephole optimization: replace inefficient patterns

```
Load_Reg R2, R1
Load_Reg R1, R2
```

⇒

```
Load_Reg R2, R1
```

Assemblers, linkers & loaders

- assembler

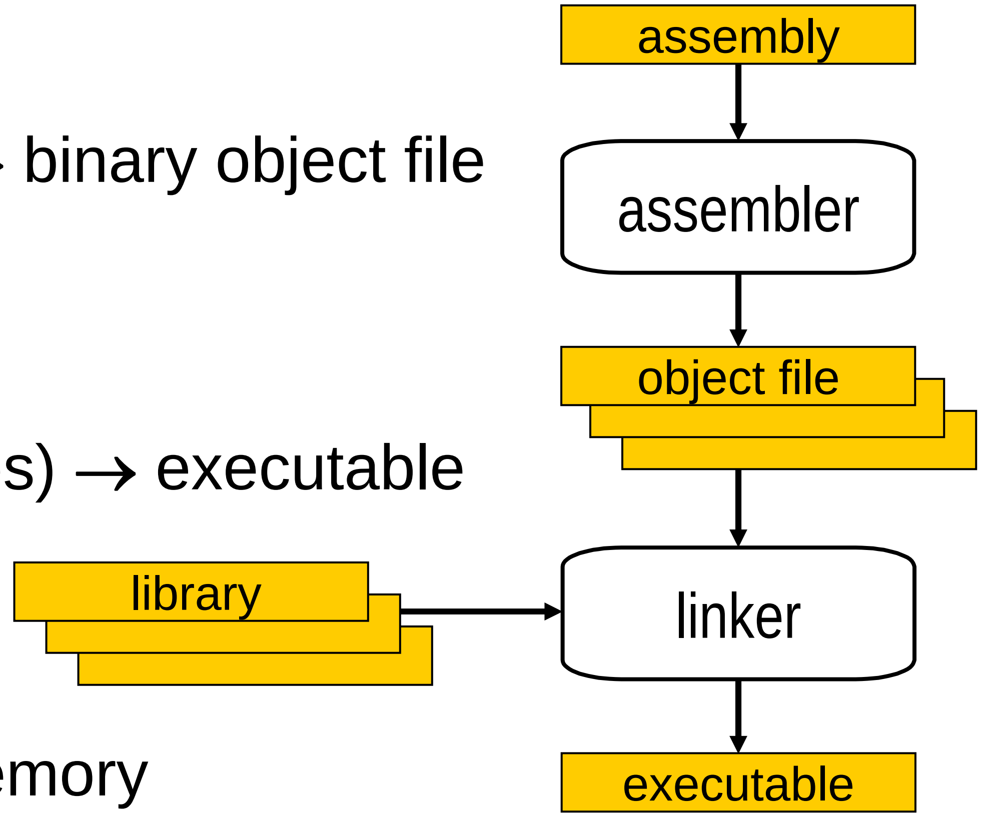
  symbolic object code → binary object file

- linker

  multiple objects (libraries) → executable

- loader

  load executable into memory



Assemblers

- symbolic code → binary machine code

`addl %edx, %ecx`   **0000 0001 11 010 001**

opcode **addl** reg-reg **edx ecx**

- handle addresses
  - internal: resolve
  - external: store in relocation table

Handling internal addresses

Two passes

- compute addresses
- generate object file

One pass

- backpatch list

disclaimer: only works for internal text labels and pc relative addressing

```
clear_vect:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %ecx
        movl    12(%ebp), %edx
        testl   %edx, %edx
        jle     .L5
        movl    $0, %eax
.L4:    movl    $0, (%ecx,%eax,4)
        incl    %eax
        cmpl    %eax, %edx
        jne     .L4
.L5:    popl    %ebp
        ret
```

Handling external addresses

- relocation information
  - entry points (text + data)
  - unresolved references

```
int **A;
int N;

void foo()
{
    int i;

    for (i=0; i<N; i++)
        clear_vect(A[i], N);
}
```

symbol	type	address	
A	entry	000000	data
N	entry	000004	data
foo	entry	000000	text
N	ref	000005	text
A	ref	000014	text
clear_vect	ref	00001c	text