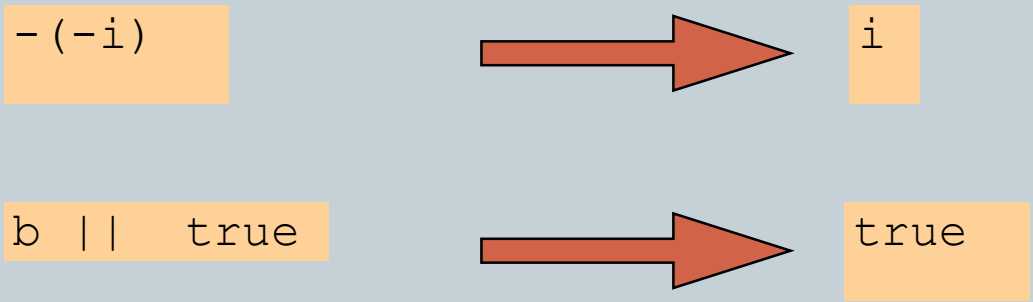
Compiler Construction

ARNOLD MEIJSTER
A.MEIJSTER@RUG.NL

Algebraic Simplifications

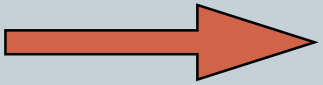
- Use algebraic properties to simplify expressions



Constant Folding

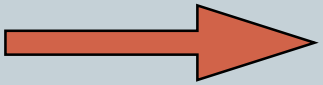
- Evaluate constant expressions at compile time

```
c = 1 + 3
```



```
c = 4
```

```
not true
```



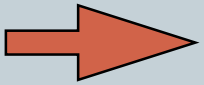
```
false
```

Constant Propagation + constant folding

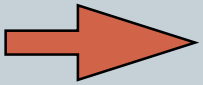
- Given an assignment **`x = c`**, where **`c`** is a constant, replace later uses of **`x`** with uses of **`c`**, provided there are no intervening assignments to **`x`**.

- Can be used together with constant folding

```
b = 3
c = 1 + b
d = b + c
```



```
b = 3
c = 1 + 3
d = 3 + c
```



```
b = 3
c = 4
d = 7
```

Order of Statements Matters

(A+B) – (C – X*Z)

```
1.T1 = A + B;
2.T2 = X * Z;
3.T3 = C - T2;
4.T4 = T1 – T3;
```

```
1.Load A, R1
1.Add B, R1
2.Load X, R2
2.Mult Z, R2
Store R1, T1
3.Load C, R1
3.Sub R2, R1
Load T1, R2
4.Sub R1, R2
4.Store R2, T4
```

Spill Code

```
1.T2 = X * Z;
2.T3 = C - T2;
3.T1 = A + B;
4.T4 = T1 – T3;
```

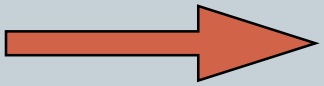
```
1.Load X, R1
1.Mult Z, R1
2.Load C, R2
2.Sub R1, R2
3.Load A, R1
3.Add B, R1
4.Sub R2, R1
4.Store R1, T4
```

Assumption: we have only two registers R1 and R2 available.

Peephole Optimization

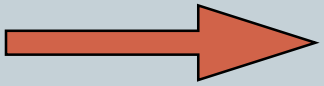
- Simple local optimization performed on assembly
- Look at code "through a hole"
  - replace sequences by known shorter ones
  - table pre-computed

```
store R,a
load a,R
```



```
store R,a
```

```
imul 2,R
```

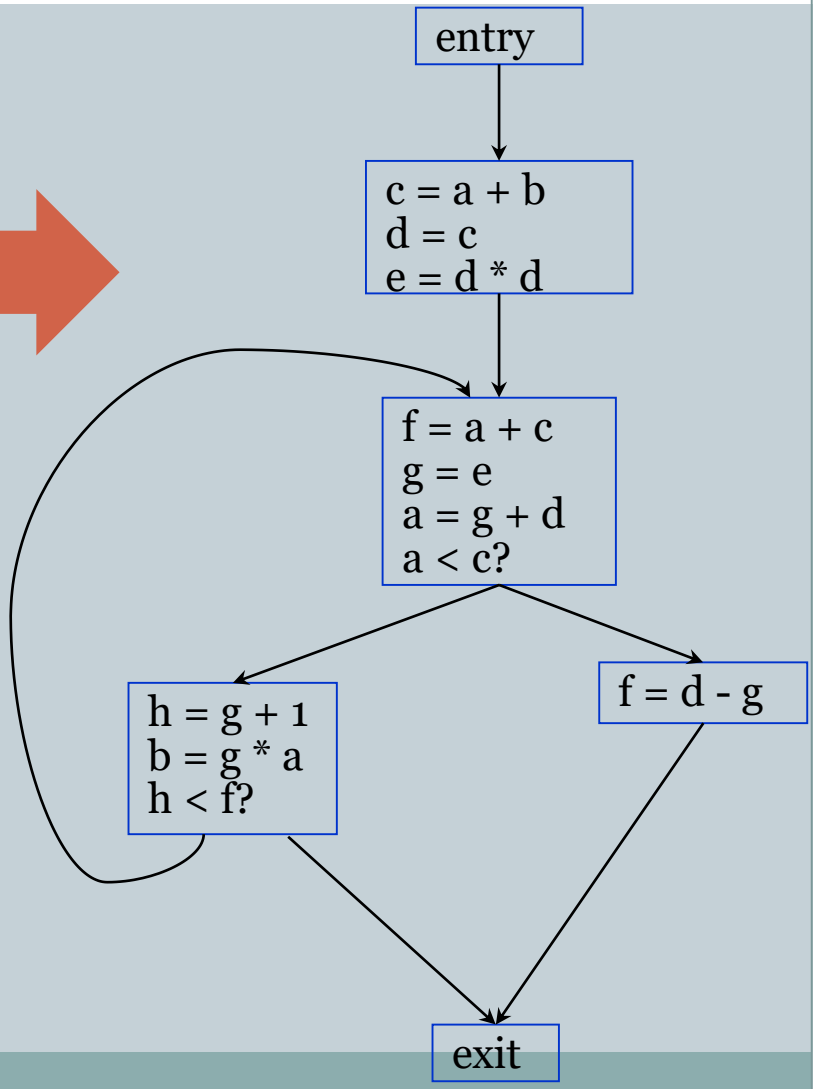


```
shl 1,R
```

Global Optimizations

Flow graph

```
        c = a + b;
        d = c;
        e = d*d;
lab1:   f = a + c;
        g = e;
        a = g + d;
        if (a < c) goto lab2;
        f = d - g;
        goto lab3:
lab2:   h = g + 1;
        b = g*a;
        if (h < f) goto lab1;
lab3:
```



Basic blocks

- For control flow analysis, we need to detect basic blocks.

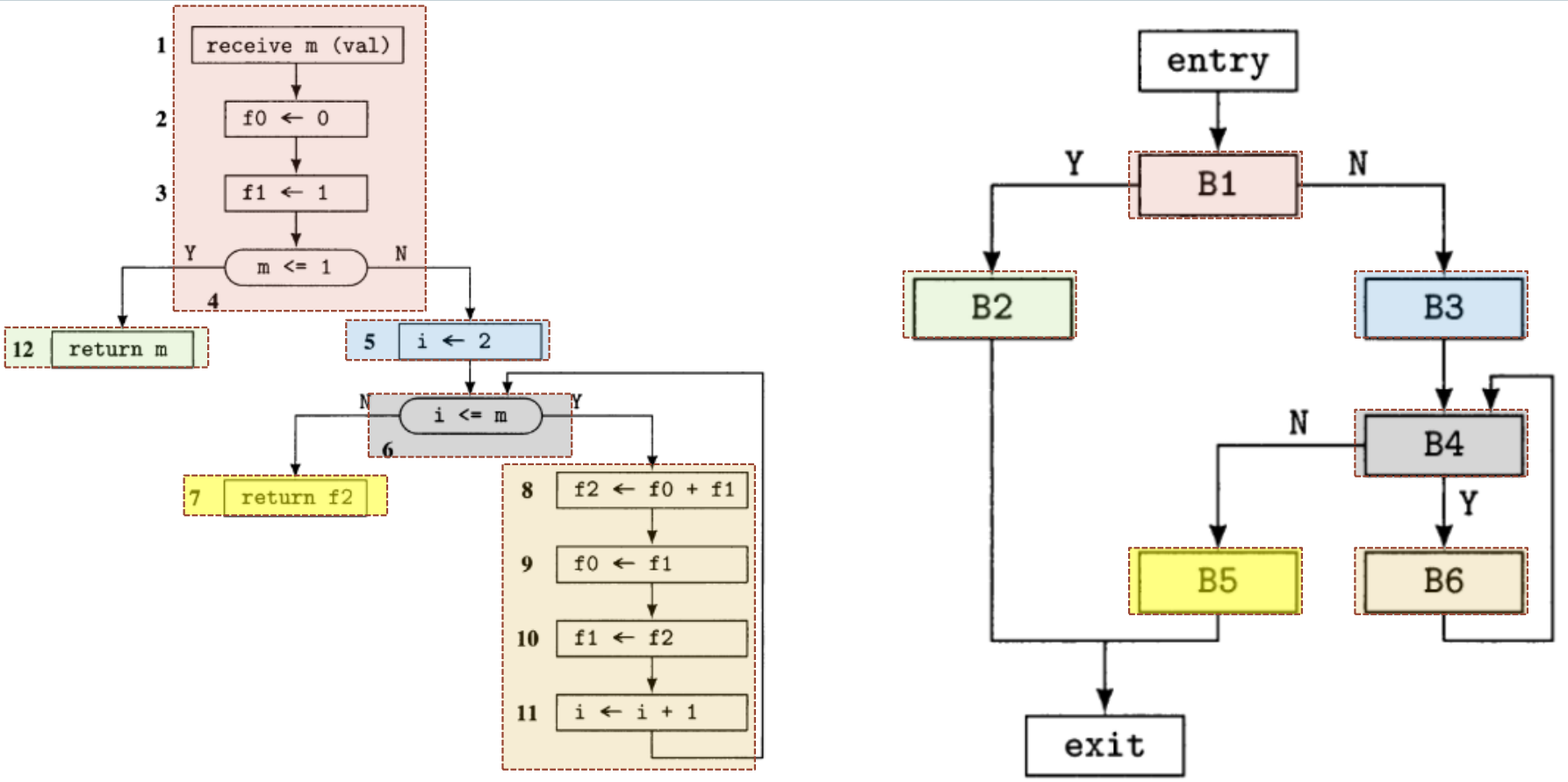
```
   y = 12;
   x = y * 2;      // here x = 24
lab1:
   x = y * 2;      // x = 24? Can't tell, y may be different
```

- A basic block is a single-entry, single-exit code fragment: values that are computed within a basic block have a single origin

Finding basic blocks

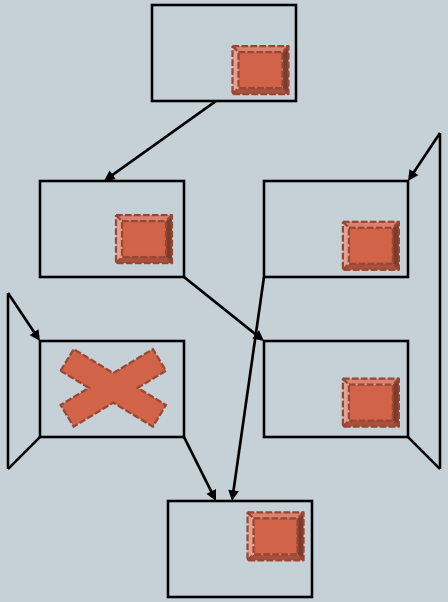
- To partition a program into basic blocks:
  - The first instruction (quadruple) in a basic block is called the leader of the block
  - The first instruction in the program is therefore a leader
  - Any instruction that is the target of a jump is a leader
  - Any instruction that follows a jump is also a leader
  - In the presence of procedures with side-effects, every procedure call ends a basic block
  - A basic block includes the leader and all instructions that follow, up to but not including the next leader

Finding basic blocks



Unreachable Code Elimination

```
Initially all BB's are unmarked
Mark leader BB
to_visit = { leader BB }
while (to_visit not empty)
     current = to_visit.pop()
     for each unmarked successor of current
        Mark successor;
        to_visit.push(successor)
     endfor
endwhile
Eliminate all unmarked blocks
```

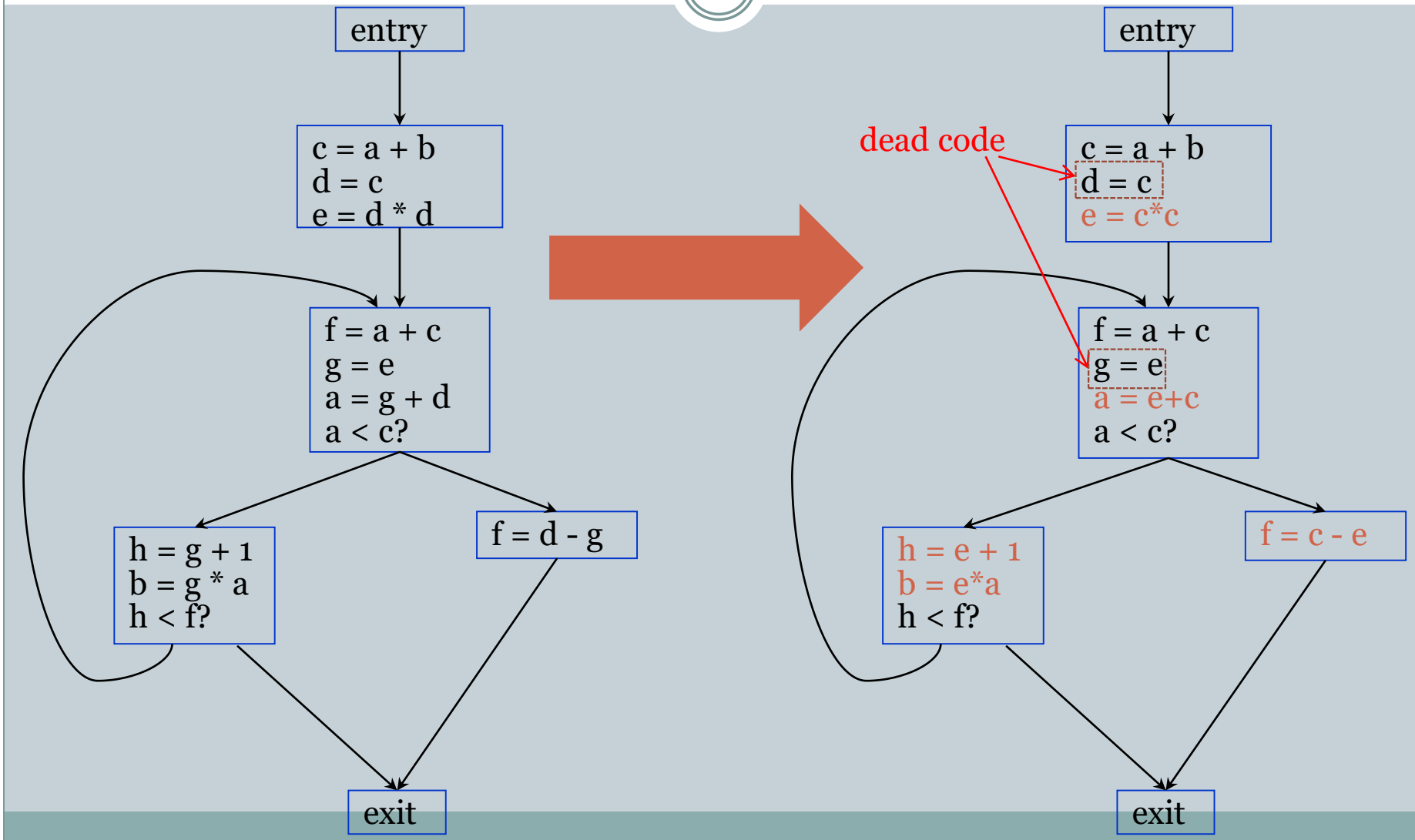


Which BB(s) can be deleted?

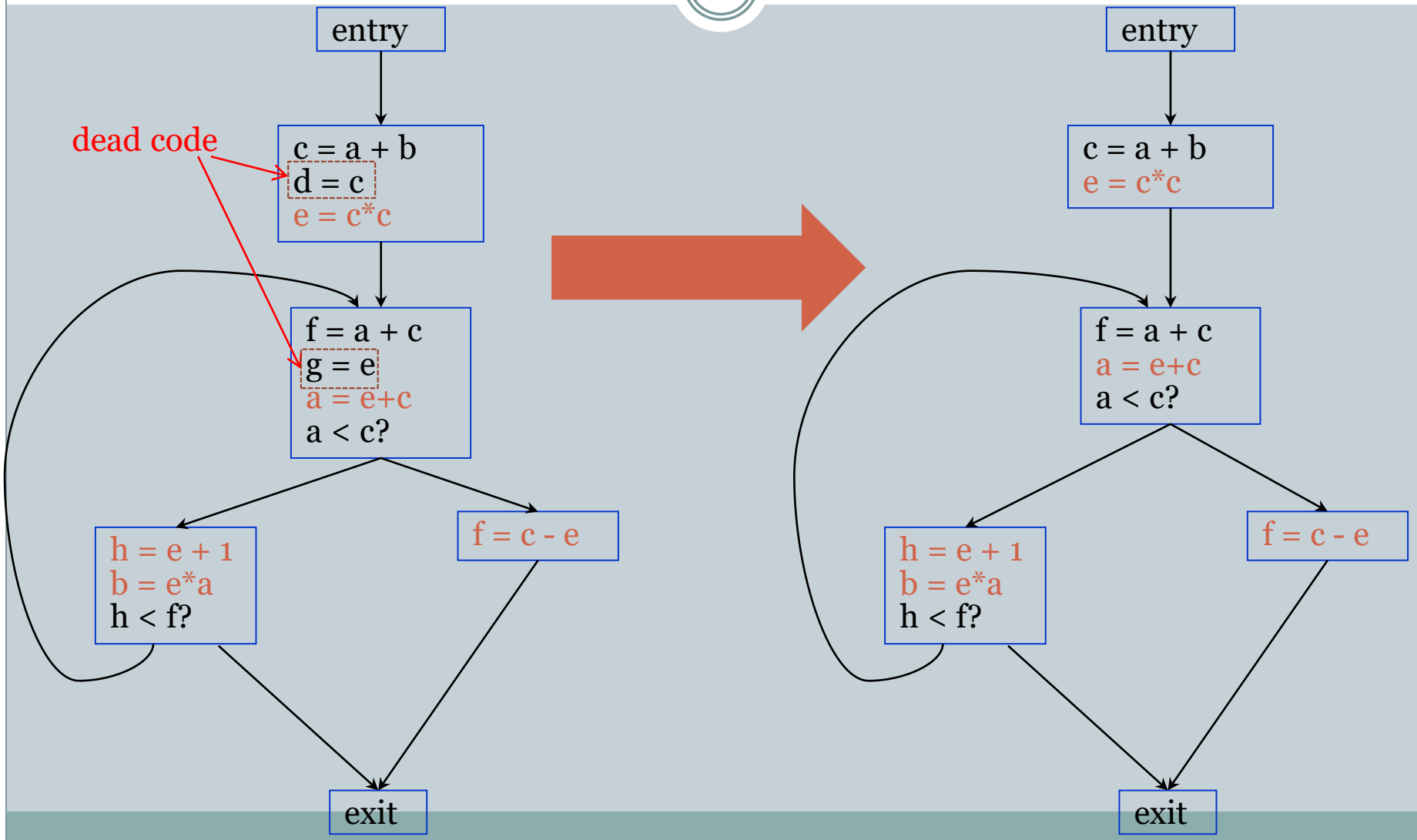
Global Copy Propagation

- Global copy propagation
  - Performed on flow graph.
  - Given *definition* statement **x=y** and *use* **w=x**, we can replace **w=x** with **w=y** only if the following conditions are met:
    - **x=y** must be the only definition of **x** reaching **w**
    - there may be no definitions of **y** on any path from **x=y** to **w=x**.
  - Use data flow analysis to check this.

Global Copy Propagation



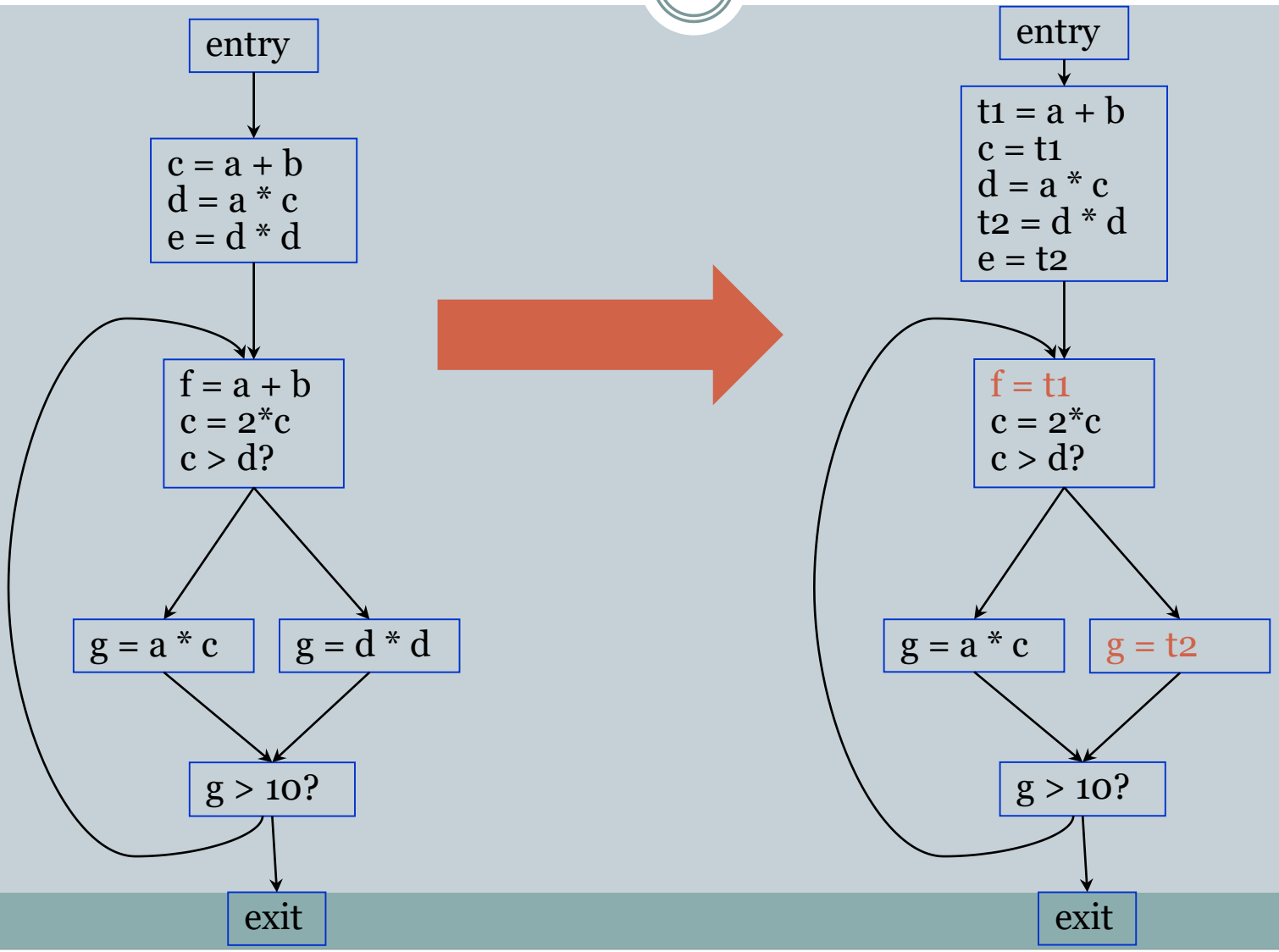
Dead code eliminiation (after propagation)



Global Common Subexpression Elimination

- Global common subexpression elimination
  - Performed on flow graph
  - Requires available expression information
    - We need to know which expressions are available at the endpoints of basic blocks
    - And, we need to know where each of those expressions was most recently evaluated.

Global Common Subexpression Elimination



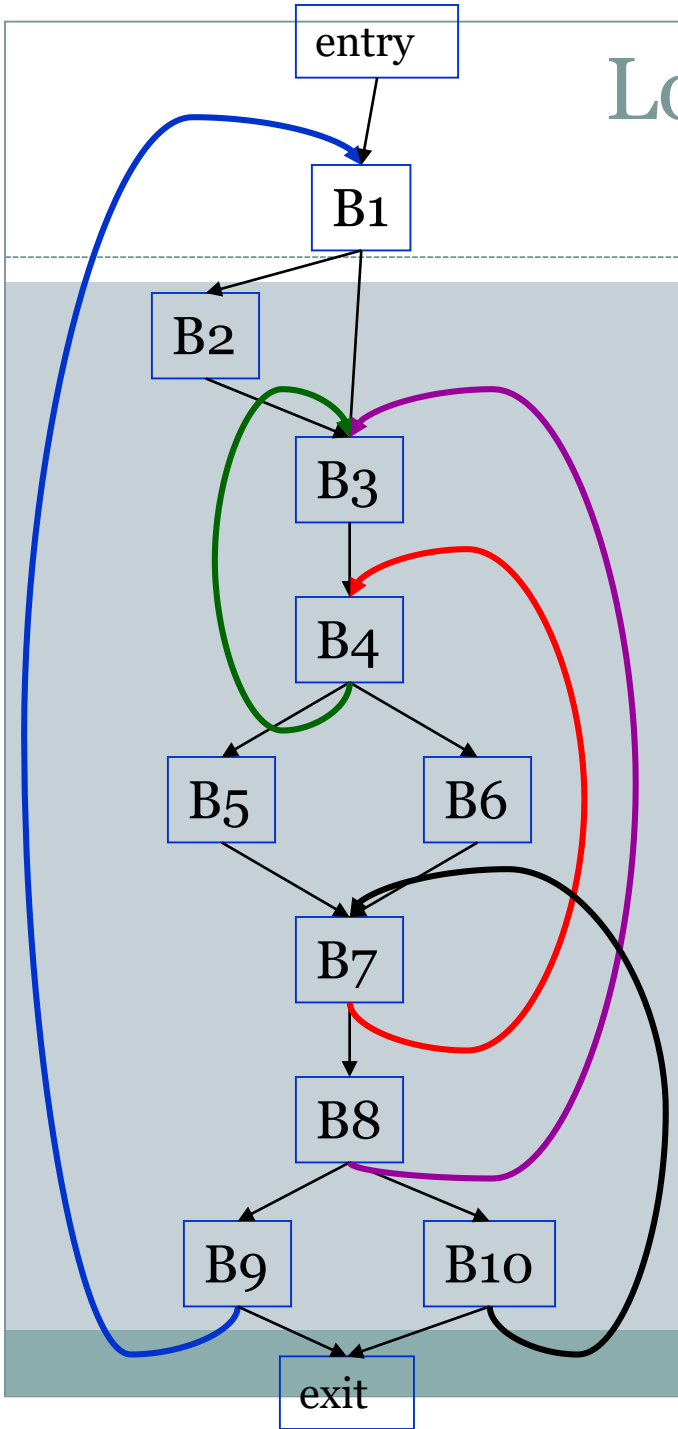
What is a loop (in a flow graph)?

- Block B_i dominates block B_j if every path from the flow graph entry to B_j goes through B_i

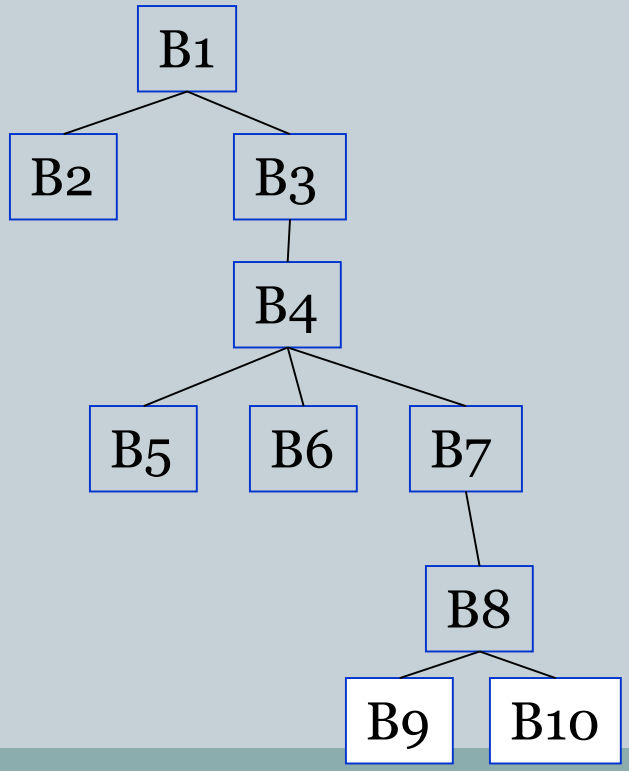
- Loop = A set of basic blocks with
  - a single entry point called the header, which dominates all the other blocks in the set, and
  - at least one way to iterate (i.e. go back to the header)

- A loop can be identified by finding a flow graph edge $B_j \rightarrow B_i$ (called a back edge) such that B_i dominates B_j and then finding all blocks that can reach B_j without going through B_i

Loops: dominator tree

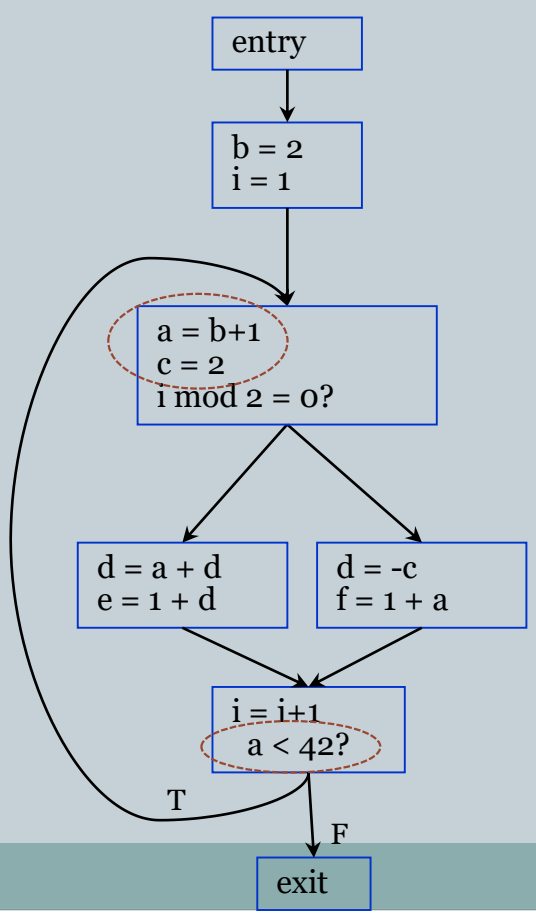


The *dominator tree* shows the dominator relation: each node in the tree is the *immediate* dominator of its children.

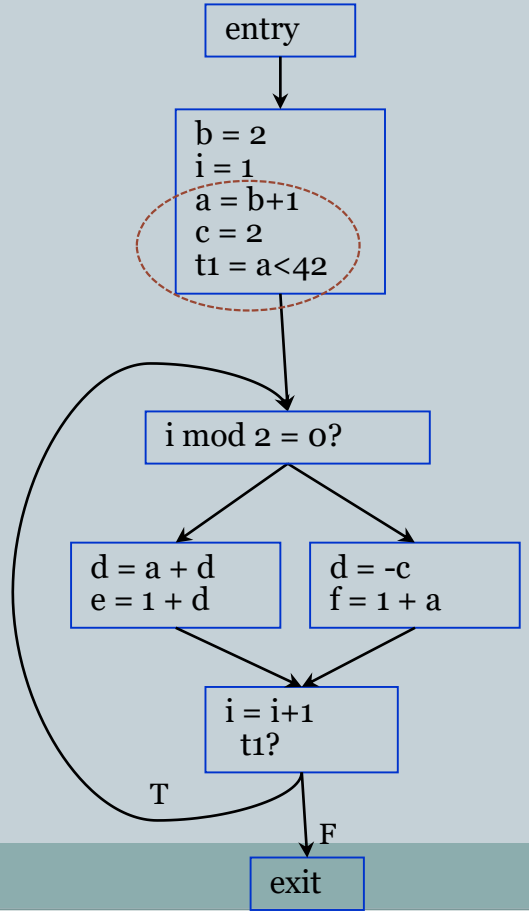


Loop-Invariant Code Motion/Hoisting

- Computations that are performed in a loop and have the same value at every iteration can be moved outside the loop.



Note that a good compiler will detect that this loop is actually an infinite loop!



Loop-Invariant Code Motion

- How do we identify loop-invariant computations?
  - If a computation i depends solely on a loop-invariant computation j, then i is also loop-invariant.
  - This gives rise to an inductive definition of loop-invariant computations

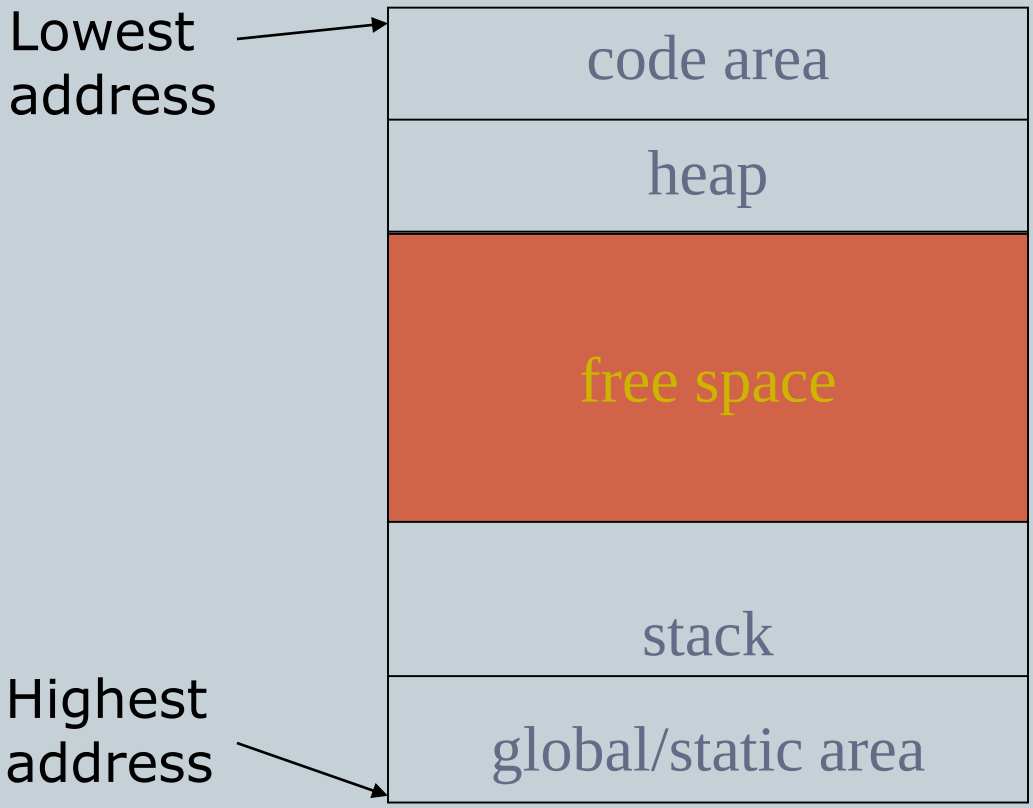
- An instruction is *loop-invariant* if, for each operand:
  1. The operand is constant, OR
  2. All definitions of the operand that reach the instruction are outside the loop, OR
  3. There is exactly one in-loop definition of the operand that reaches the instruction, and that definition is loop invariant

Activation records

Stack-based Runtime Environment

- In programming languages that allow (recursive) function/procedure calls, *activation records* are allocated in a stack-based fashion.
- This stack is called the stack of activation records (runtime stack, call stack).
- A (mutual) recursive procedure/function may have several different activation records at the same time.

Memory Organization

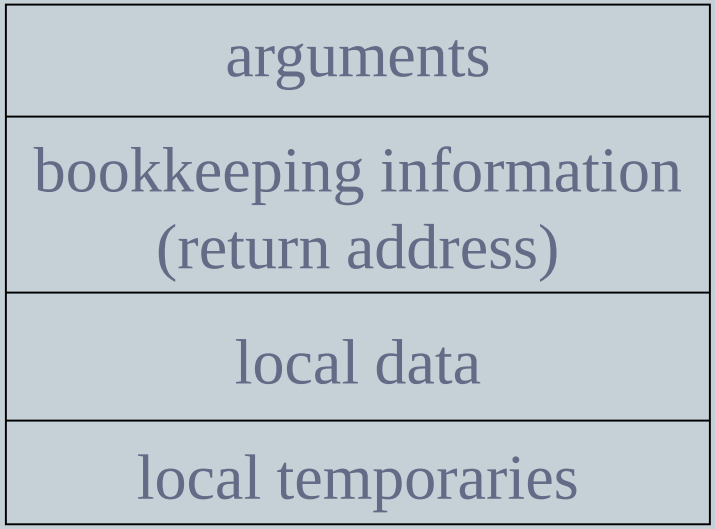


Runtime stack

- Call = push new activation record
- Return = pop activation record
- Only one “active” activation record – top of stack
  - Naturally handles recursion

Activation Record

- Function/procedure activation record (AR) contains memory allocated for the local data, arguments, local variables, possibly local temporaries that are introduced by the compiler, and bookkeeping information.



- General organization of an activation record.

- Details depend on the architecture of target machine and properties of the language.

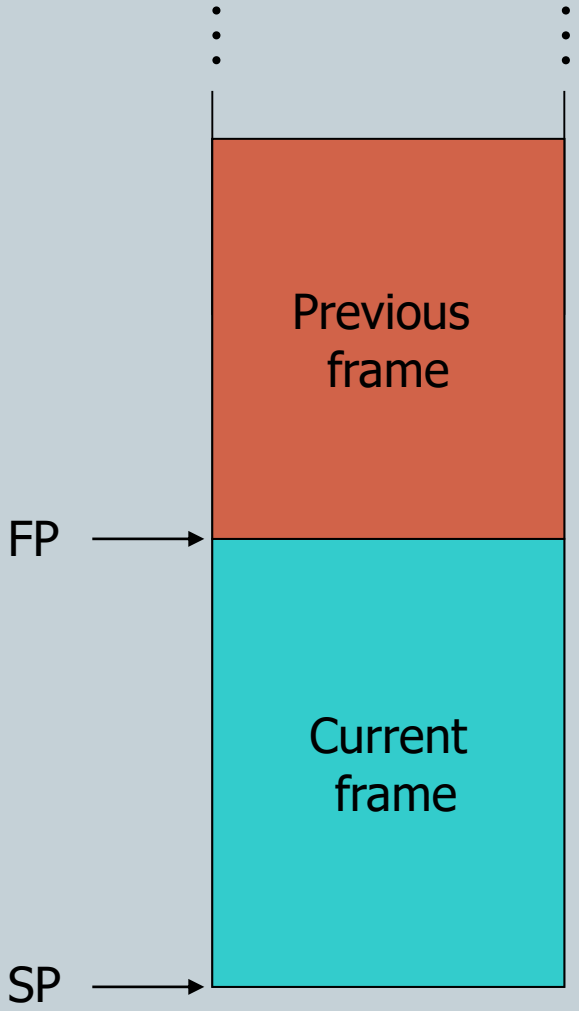
Global Procedures

- In a language where all procedures are global (like in C), a stack-based environment requires two things:

1. A pointer to the current activation record to allow access to local variables.
   - This pointer is called the frame pointer (fp) and is usually kept in a register.
2. The position of the caller's activation record
   - This information is kept in the current activation record as a pointer to the previous activation record and is referred to as the dynamic link.
3. Of course, there is also a stack pointer (sp)
   - It always points to the top of the stack

Runtime stack

- Stack grows downwards (towards smaller addresses)
- SP – stack pointer
  – Top of current frame
- FP – frame pointer
  – Base of current frame
  - Sometimes called BP (base pointer)



Stack maintenance

- A typical calling sequence :
  1. Caller assembles arguments and transfers control
     - evaluate arguments
     - push arguments on stack (and/or in registers)
     - Optionally: reserve space for return value callee
     - push return address
     - jump to callee's first instruction

Stack maintenance

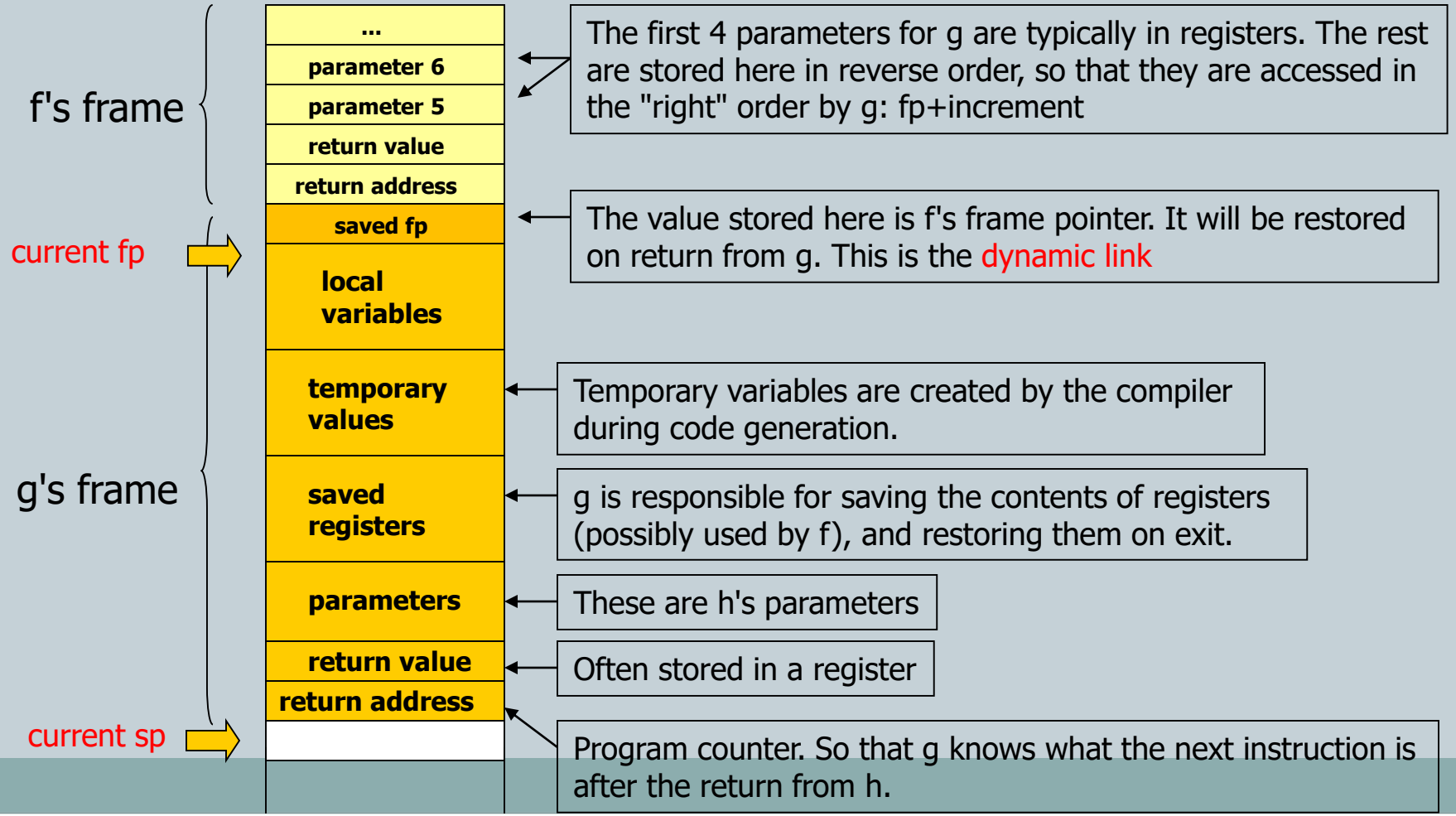
- A typical calling sequence :
  2. Callee saves info on entry
     - allocate memory for stack frame, update stack pointer
     - save registers
     - save old frame pointer
     - update frame pointer
  3. Callee executes code

Stack maintenance

- A typical calling sequence :
  4. Callee restores info on exit and returns control
     - place return value in appropriate location
     - restore frame pointer
     - restore registers
     - pop the stack frame
     - jump to return address
  5. Caller continues

Storage organization

- Typical stack layout. Assume function f called g, and g is about to call h



x86 runtime stack support

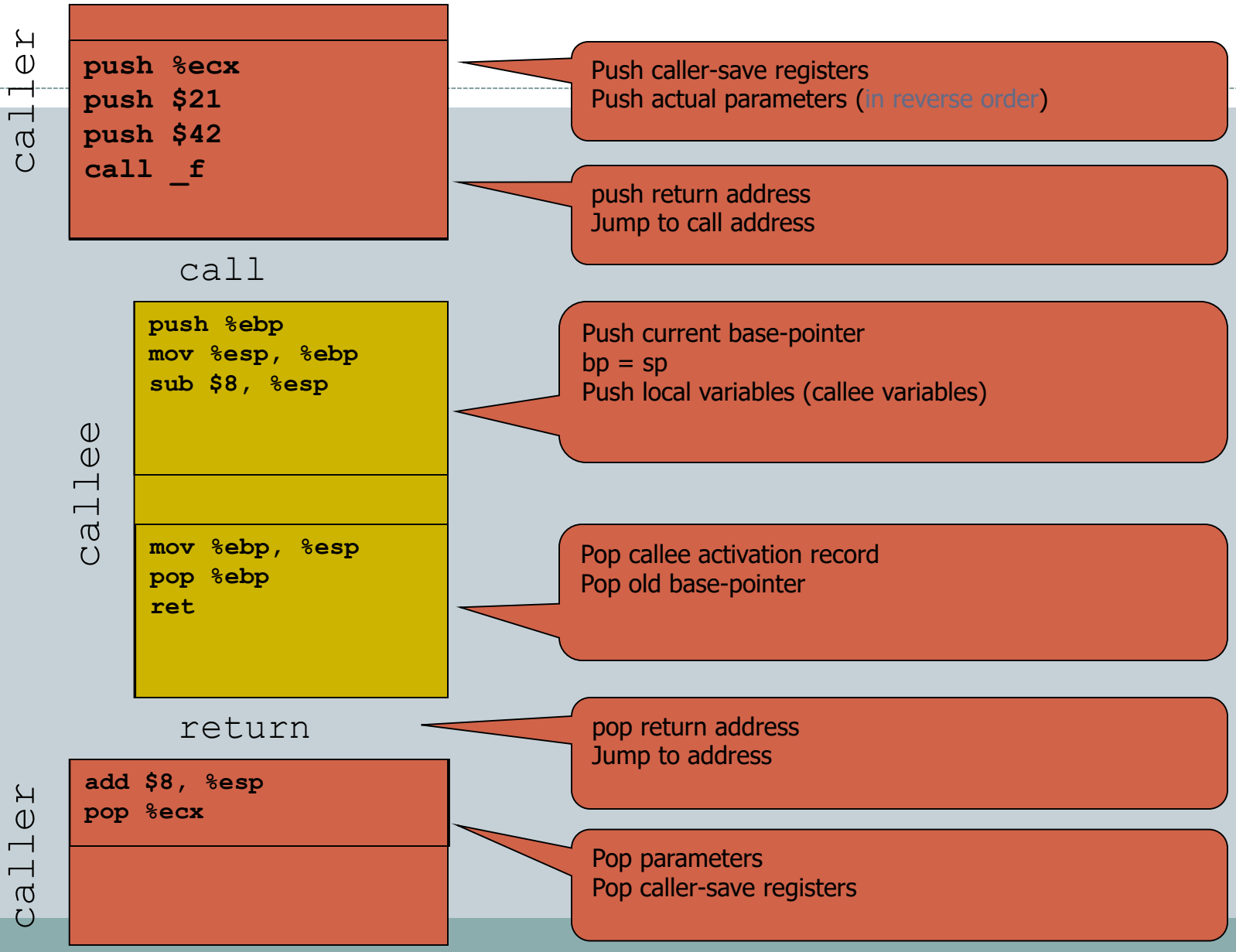
Register	Usage
ESP	Stack pointer
EBP	Base pointer

Pentium stack registers

Instruction	Usage
push, pusha,…	Push on runtime stack
pop, popa,…	Pop from runtime stack
call	Transfer control to called routine
ret	Transfer control back to caller

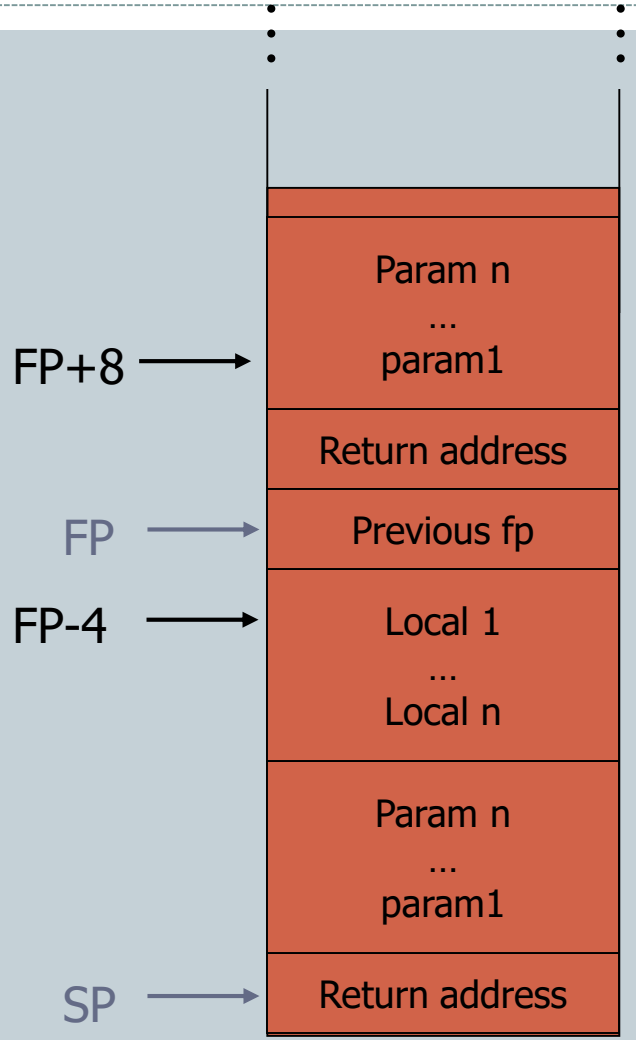
x86 stack and call/ret instructions

Call sequences – `f(42,21)`



Accessing stack variables

- Use offset from EBP
- Remember –
  stack grows downwards
- Above EBP = parameters
- Below EBP = locals
- Examples
  - %ebp + 4 = return address
  - %ebp + 8 = first parameter
  - %ebp – 4 = first local



Summary: routine `old` calls routine `new`

- Push the arguments for **`new`** in reverse order.
- Reserve stack space for the return value of **`new`**.
- Push the return address (i.e. program counter in **`old`**).
- Push the frame pointer value of **`old`** (to save it).
- Use the stack pointer as frame pointer of **`new`**.
- Reserve stack space for the local variables of **`new`**.
- Execute the code of **`new`**.
- Place the return value at the reserved stack location.
- Restore the old frame pointer value.
- Jump to the saved return address

Amen

Exam: Friday January 26th, 14:00-17:00