

## 4 Register Allocation

### Use of registers:

1. intermediate **results of expression evaluation**
2. reused results of expression evaluation (CSE)
3. contents of frequently used **variables**
4. **parameters** of functions, **function result**  
(cf. register windowing)
5. stack pointer, **frame pointer**, heap pointer, ...

**Number of registers is limited** - for each register class: address, integer, floating point

### Specific allocation methods for different context ranges:

- 4.1 expression trees (Sethi, Ullman)
- 4.2 basic blocks (Belady)
- 4.3 control flow graphs (graph coloring)

### Register allocation aims at reduction of

- number of memory accesses
- spill code, i. e. instructions that store and reload the contents of registers

**Symbolic registers:** allocate a new symbolic register to each value assignment (single assignment, no re-writing); defer allocation of real registers to a later phase.

# Register Windowing

Register windowing:

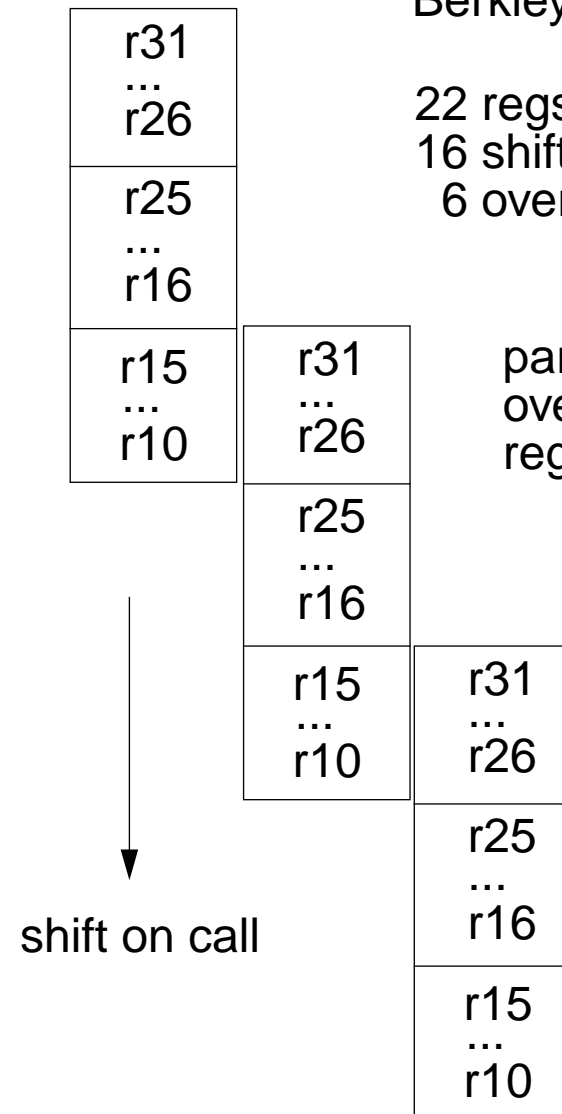
- Fast storage of the processor is accessed through a window.
- The  $n$  elements of the window are used as registers in instructions.
- On a call the window is shifted by  $m < n$  registers.
- Overlapping registers can be used under different names from both the caller and the callee.
- Parameters are passed without copying.
- Storage is organized in a ring;  
4-8 windows; saved and restored as needed

Typical for Risc processors,  
e.g. Berkley RISC, SPARC

Berkley Risc:

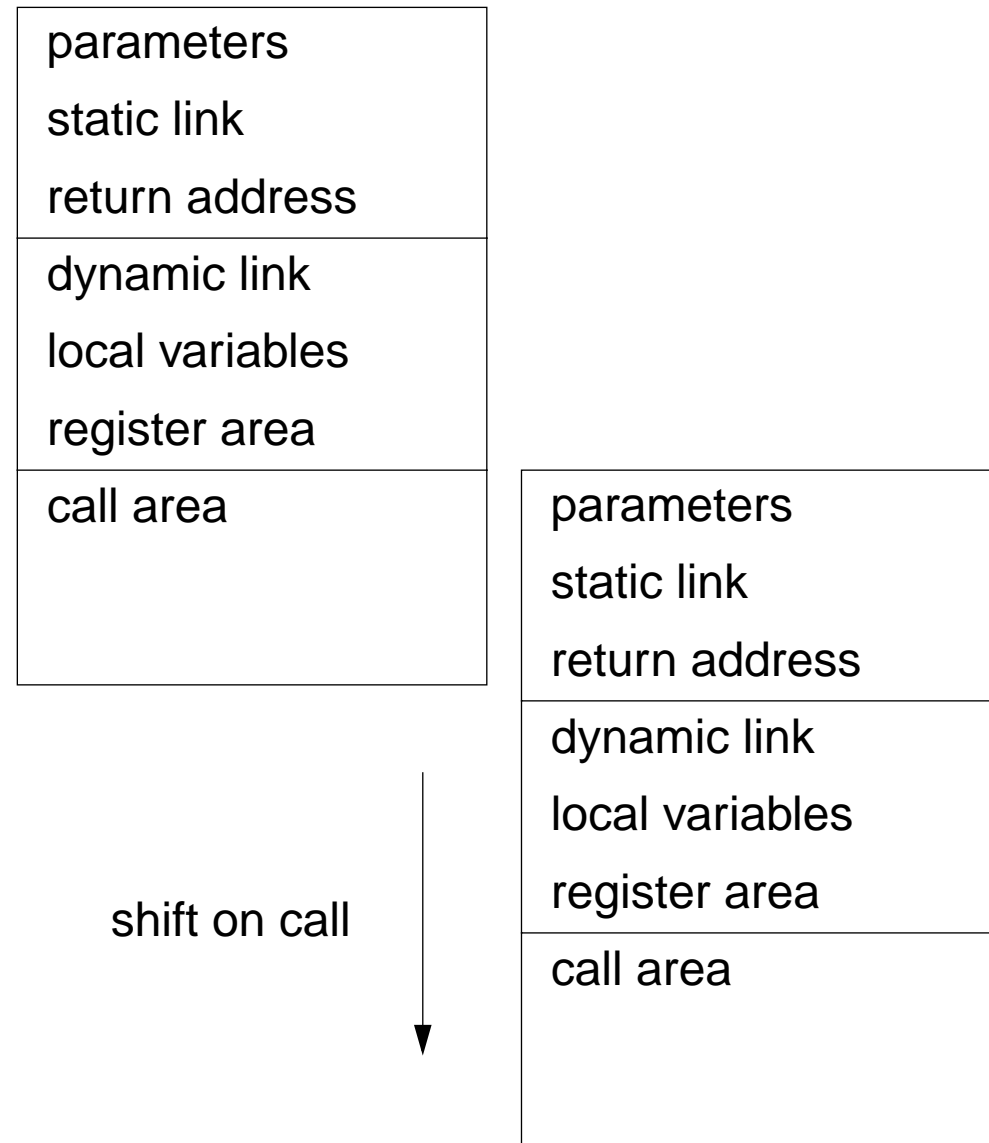
22 regs in window  
16 shifted  
6 overlapped

parameters in  
overlapping  
registers



# Activation Records in Register Windows

- **Parameters** are passed in overlap area **without copying**.
- **Registers need not be saved** explicitly.
- If **window is too small** for an activation record, the remainder is allocated on the **run-time stack**; pointer to it in window.



## 4.1 Register Allocation for Expression Trees

### Problem:

Generate code for **expression** evaluation.

**Intermediate results** are stored in registers.

Not enough registers:

**spill code** saves and restores.

### Goal:

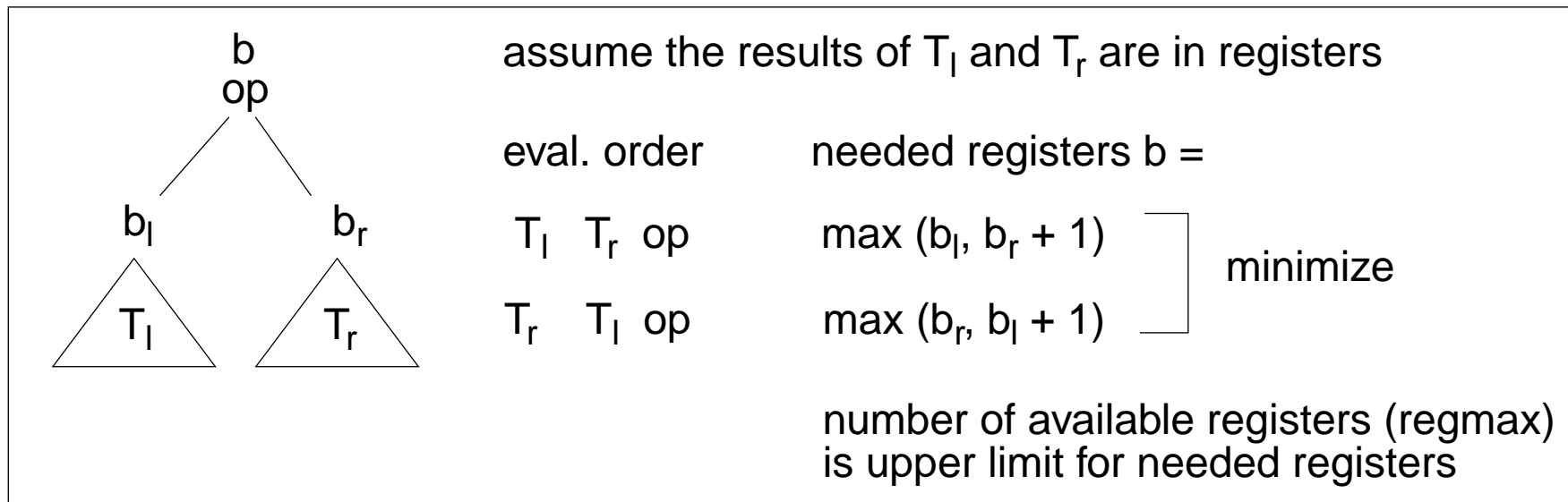
Minimize amount of spillcode.

see C-4.5a for optimality condition

### Basic idea (Sethi, Ullman):

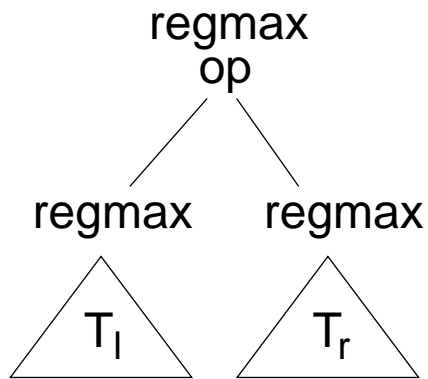
For each subtree minimize the  
**number of needed registers:**

evaluate **first the subtree that  
needs most** registers



# Expression Tree Attribution

Spill code needed:



Code ( $T_r$ )  
**store**  $R_r, h$   
 Code ( $T_l$ )  
**load**  $h, R_r$   
 op  $R_r, R_l$

load  $h, R_r$  is not needed if  $h$  can be a memory operand in op  $h, R_l$

**Implementation by attribution of trees:**

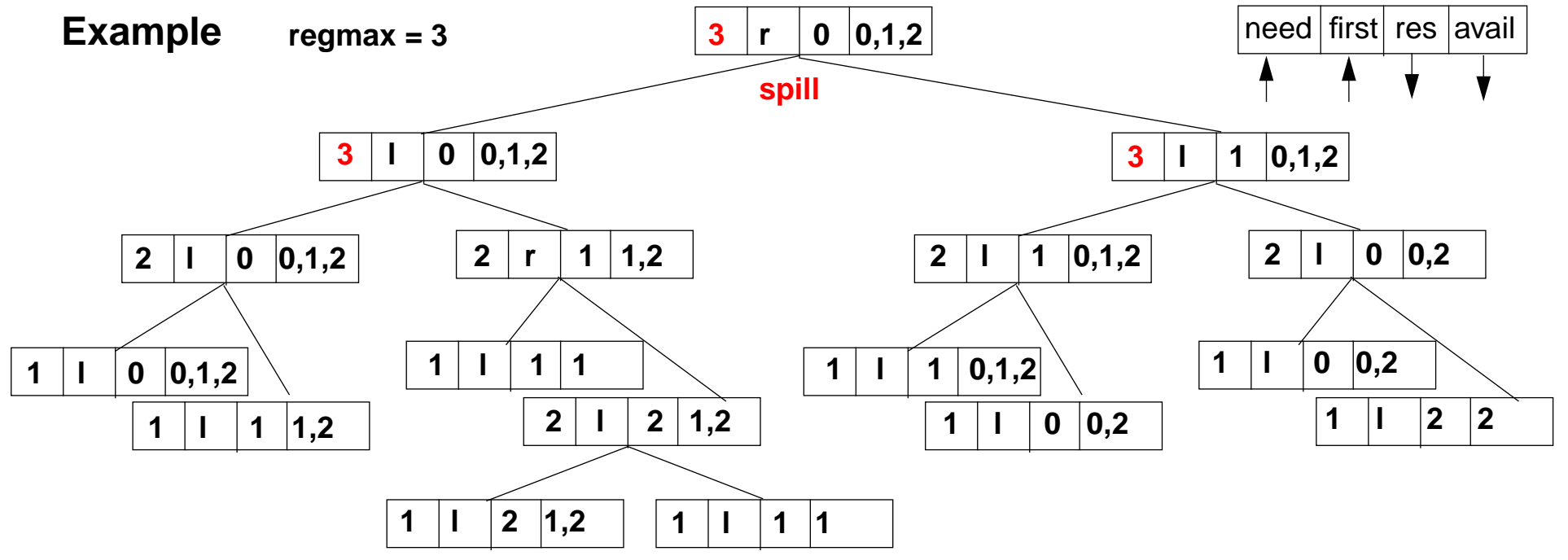
**Phase 1** bottom-up:  
 needed registers, evaluation order

**Phase 2** top-down:  
 allocate registers

**Phase 3** bottom-up:  
 compose code in evaluation order

**Example**

regmax = 3



## Contiguous code vs. optimal code

The method assumes that the **code for every subtree is contiguous**.  
(I.e. there is no interleaving between the code of any two disjoint subtrees.)

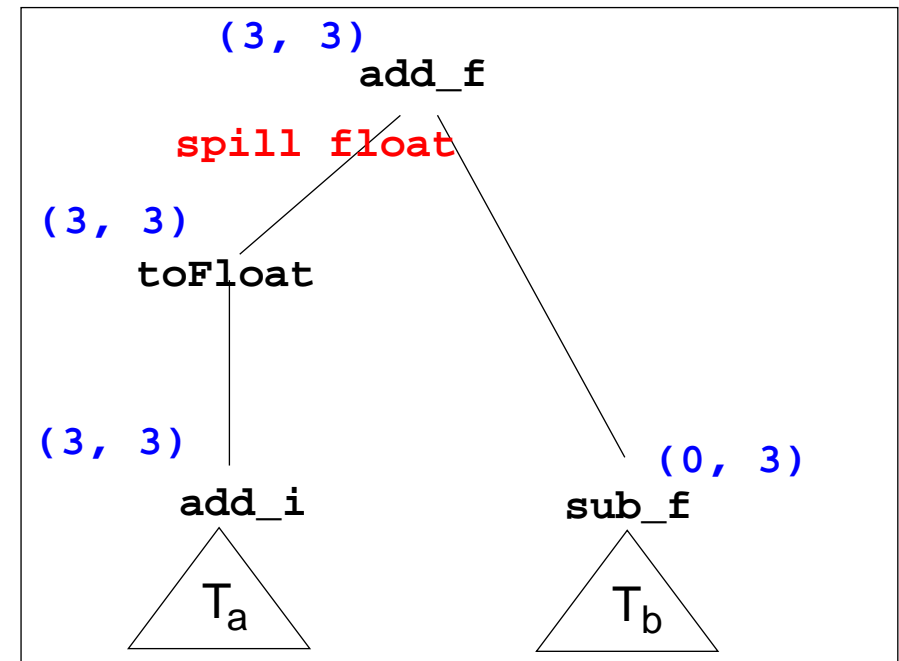
The **method is optimal** for a certain **configuration of registers and operations**, iff every **optimal evaluation code** can be arranged to be **contiguous**.

### Counter example:

Registers: 3 `int` and 3 `float`

Register need: (i, f) from (0, 0) to (3, 3)

Operations: `int`- and `float`- arithmetic,  
`toFloat` (widening)



register use: (3, 3) (1, 0) (0, 1) (0, 0) (0, 3) (0, 1) (0, 2) (0, 1)

**contiguous:** T<sub>a</sub> add\_i toFloat **store\_f** T<sub>b</sub> sub\_f **load\_f** add\_f

**optimal:** T<sub>a</sub> add\_i T<sub>b</sub> sub\_f toFloat add\_f

register use: (3, 3) (1, 0) (1, 3) (1, 1) (1, 2) (0, 1)

## 4.2 Register Allocation for Basic Blocks by Life-Time Analysis

**Lifetimes of values in a basic block** are used to minimize the number of registers needed.

**1st Pass:** Determine the **life-times** of values: from the definition to the last use (there may be several uses!).

**Life-times are represented by intervals in a graph**

**cut of the graph** = number of **registers needed** at that point

**at the end of 1st pass:**

maximal cut = number of register needed for the basic block

allocate registers **in the graph:**

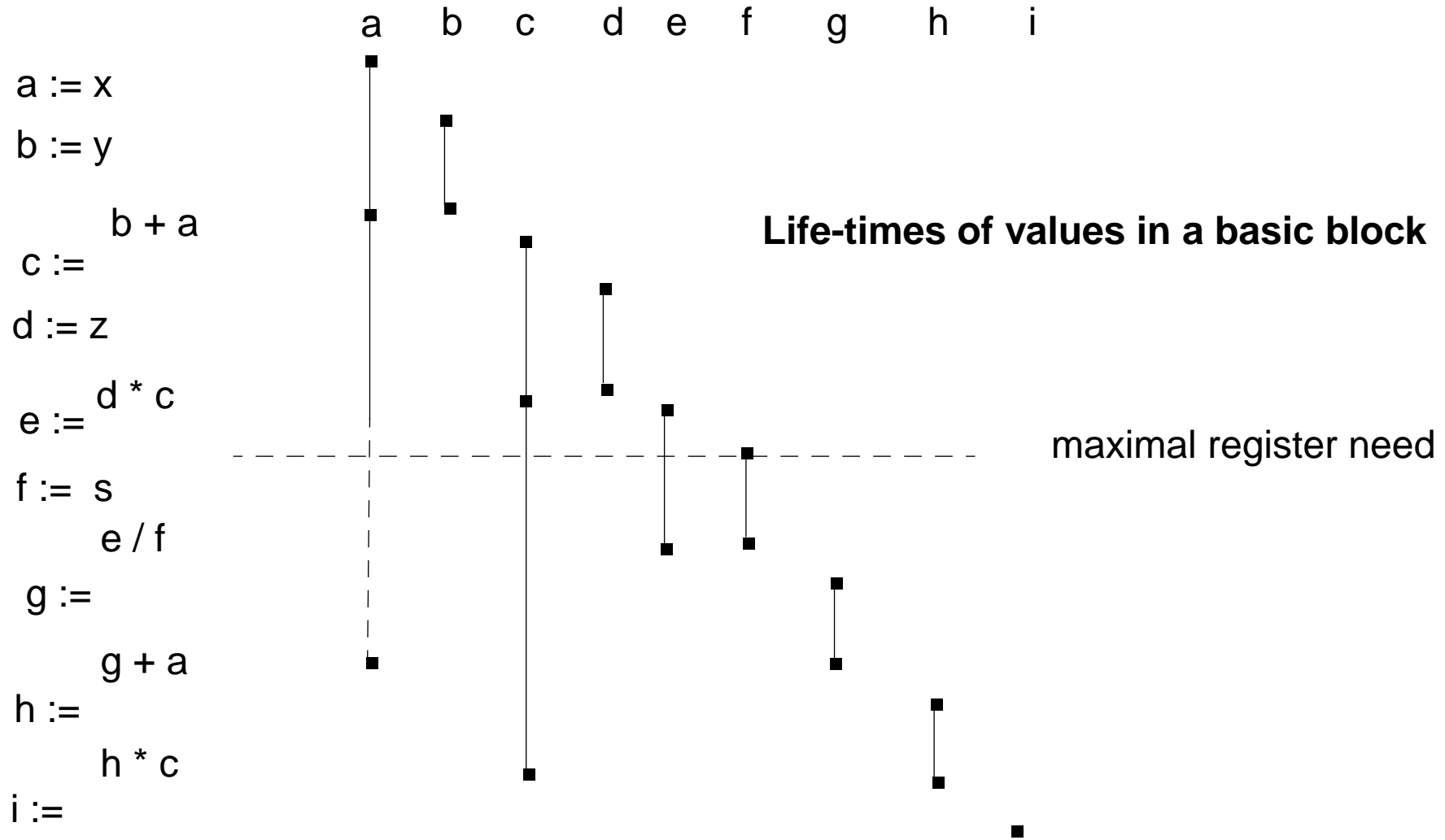
In case of shortage of registers: select values to be **spilled**; **criteria:**

- a **value that is already in memory** - store instruction is saved
- the **value that is latest used again**

**2nd Pass:** allocate registers **in the instructions**; evaluation order remains unchanged

The technique has been presented originally 1966 by **Belady** as a **paging technique for storage allocation**.

# Example for Belady's Technique



## register allocations

4 regs	(a)	d1	d2	d2	d3	d3	d4	d3	d3	d3
3 regs	(b)	d1	d2	d2*	d3	d3	d2	d3	d3	d3
3 regs	(c)	d1*	d2	d2	d3	d3	d1	d3	d3	d3

\* spilled: store;...; load

\* spilled: reloaded from x



## 4.3 Register Allocation by Graph Coloring

Definitions and uses of variables in control-flow graphs for **function bodies** are analyzed (DFA). Conflicting life-times are modelled. Presented by **Chaitin**.

### Construct an interference graph:

**Nodes:** Variables that are candidates for being kept in registers

**Edge {a, b}:** **Life-times** of variables a and b overlap  
=> a, b have to be kept in different registers

Life-times for CFGs are determined by **data-flow analysis**.

**Graph is „colored“ with register numbers.**

NP complete problem; **heuristic technique** for coloring with k colors (registers):

eliminate nodes of degree  $< k$  (and its edges)

if the graph is finally empty:

graph can be colored with k colors

assign colors to nodes in reverse order of elimination

else

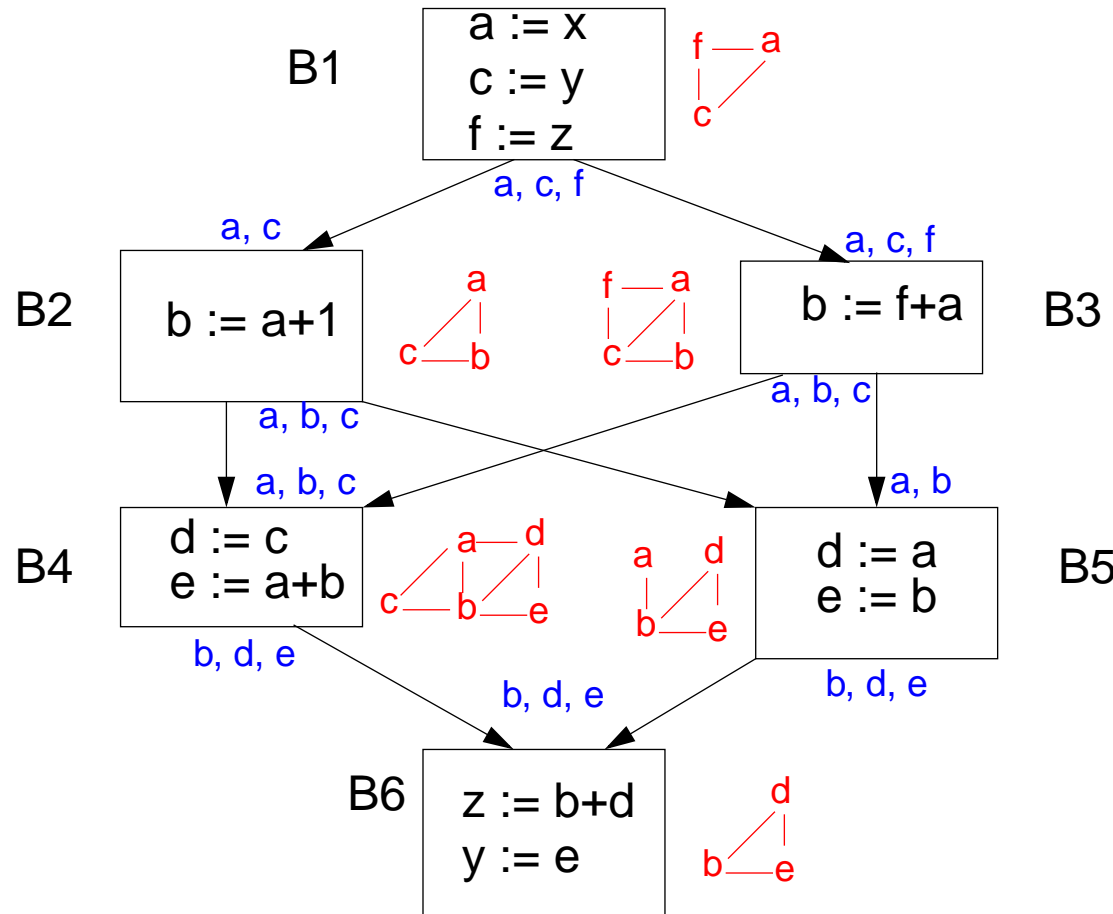
graph can not be colored this way

select a node for spilling

repeat the algorithm without that node

# Example for Graph Coloring

## CFG with definitions and uses of variables



variables in memory:  $x, y, z$

variables considered for register alloc.:  
 $a, b, c, d, e, f$

results of live variable analysis:  
 $b, d, e$

