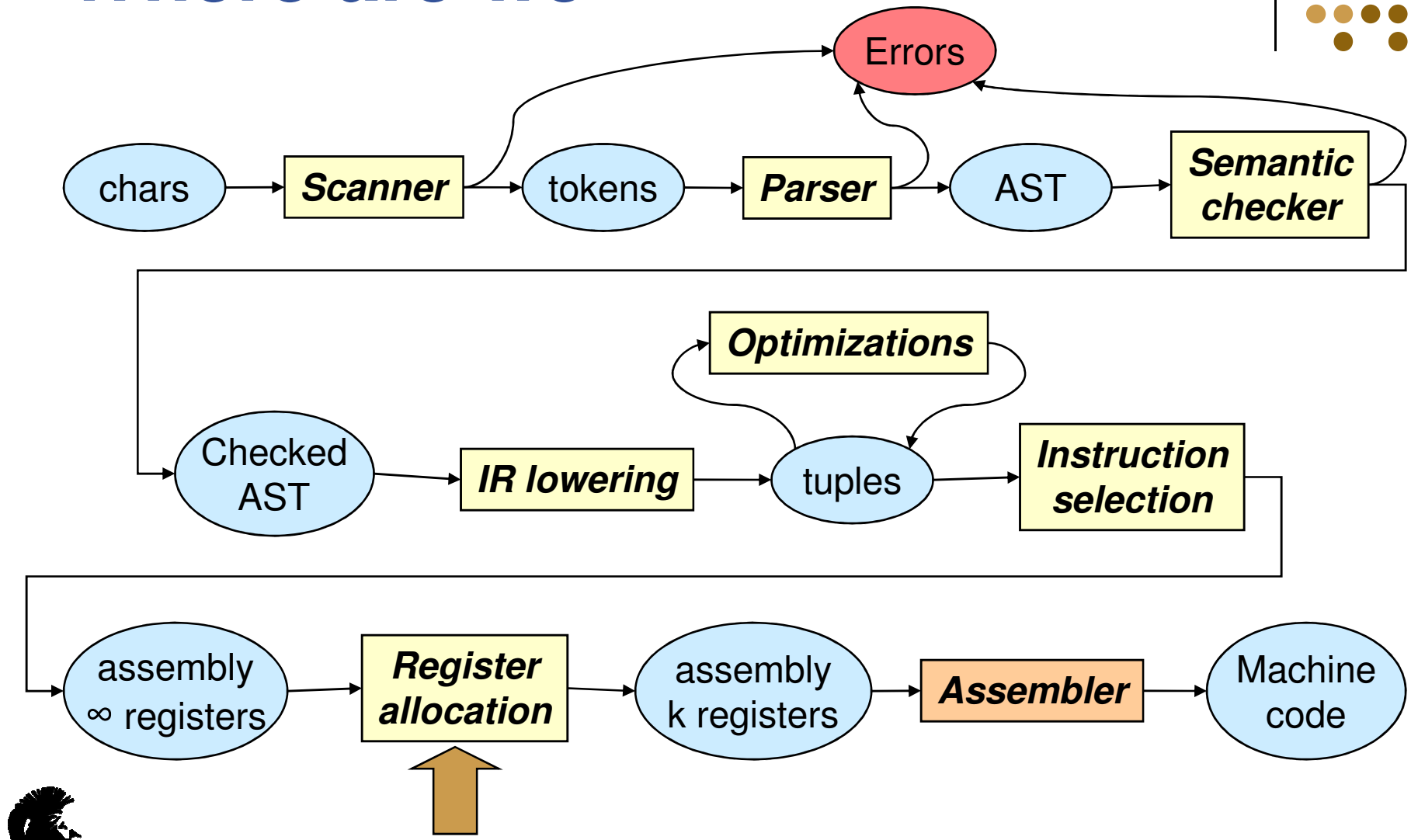
Compilers

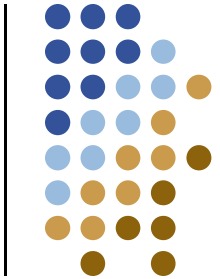
Register allocation

Yannis Smaragdakis, U. Athens
(original slides by Sam Guyer@Tufts)

Where are we

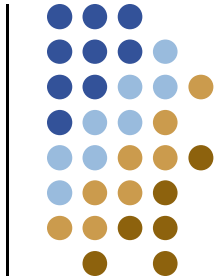

Errors
chars
Scanner
tokens
Parser
AST
Semantic checker
Optimizations
Checked AST
IR lowering
tuples
Instruction selection
assembly ∞ registers
Register allocation
assembly k registers
Assembler
Machine code

Register allocation

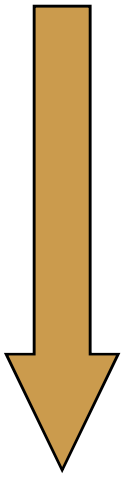


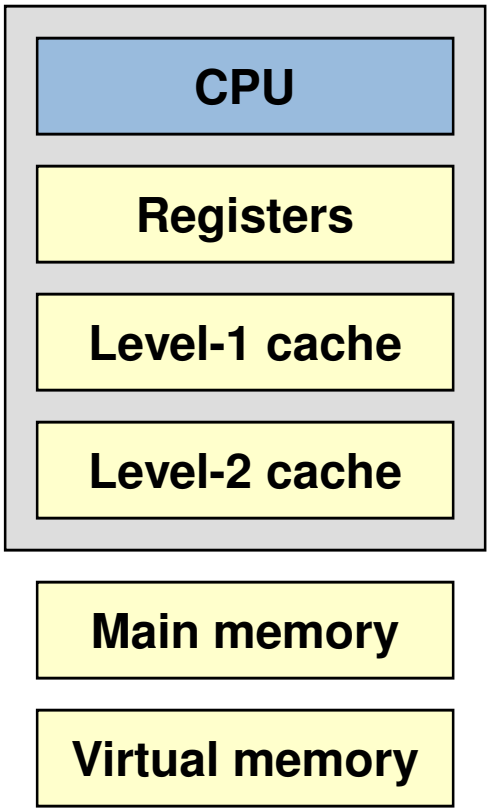
- What are registers?
  - Memory
  - Very close to the processor – very fast to access
  - On many architectures, required by ISA
    - RISC – all computations use registers
    - Pentium – many instructions register + memory
- Part of the memory hierarchy
  - Top: close to CPU, fast, small
  - Bottom: far from CPU, slow, large

Memory hierarchy



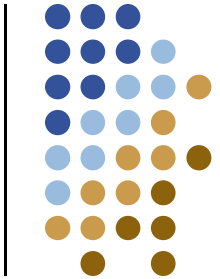
Farther away, larger, slower



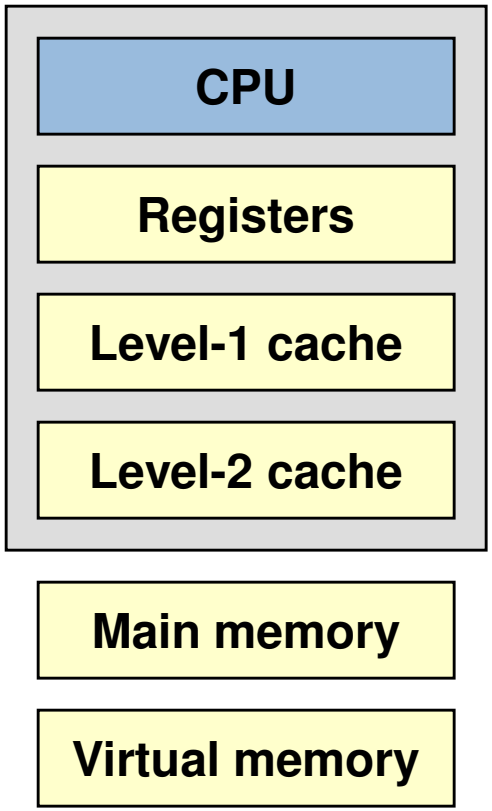


Pentium 4 3.2 Ghz	*Core 2 Duo*	*Athlon 64*
1 cycle	**1 cycle**	**1 cycle**
2 cycles (16 KB*)	**3 cycles** (64 KB)	**3 cycles** (128 KB)
19 cycles (2 MB)	**14 cycles** (2 MB)	**13 cycles** (1 MB)
204 cycles	**180 cycles**	**125 cycles**
millions of cycles	**millions of cycles**	**millions of cycles**

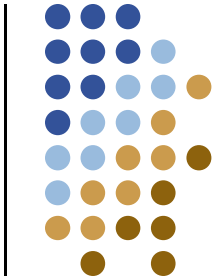
Memory hierarchy



- What is the compiler's role in the memory hierarchy?
- Virtual memory?
- Main memory?
  - Heap layout
  - Prefetching
- Level-1 and level-2 cache?
  - Many ***locality*** optimizations
  - Loop transforms, tiling, strip mining
- Registers
  - Compiler has direct control

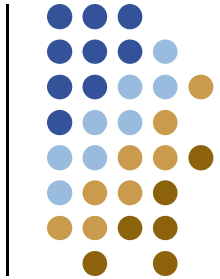


Registers



- How important is register allocation?
  - Widely recognized as one of the most important "optimizations" performed by the compiler
  - An order of magnitude compared to poor or no register allocation
  - Most other optimizations: at most ~ 10% to 20%
- Varies somewhat depending on machine
  - Number of registers
  - Architecture constraints on register use
  - Speed of memory hierarchy

Using registers



- Software view

  ISA: Assembly code and machine code

  - Register names are explicit
  - Like variables, names represent data dependences
  - Hard to change this view – why?

- Machine view

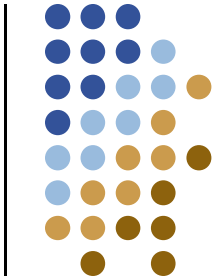
  - Actual machine may have more registers

	At ISA level	***Physical***
DEC Alpha	**64 int+float**	**80 int, 72 float**
IA-32	**8 int**	**128 int**

  - Why have more physical registers than ISA?

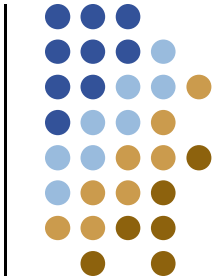
    Renaming may occur inside the processor

Register allocation



- What are we trying to do?
  - Register allocation
    - Decide which values will be kept in registers
  - Register assignment
    - Select specific registers for each value
- Constraints
  - Primary: limited number of registers
  - Different kinds of registers -- integer vs floating point
  - Special-purpose registers – SP
  - Instruction requirements – x86 mul must use eax, adx
  - Some values cannot go in registers

Register allocation

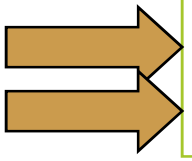


- What values can go in registers?

  What does it mean to "allocate a variable in a register"?

  - Most cases: variable ***becomes*** a register
  - All uses and defs replaced with the register
  - It has no storage on the stack

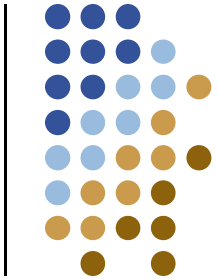
- What is the implication of that decision?
  - The compiler must be able to see all accesses
  - For example:



```
int x;
int * p = &x;
(*p) = 7;
foo(p);
```

Might be able to handle (*p) = 7 case

Register allocation

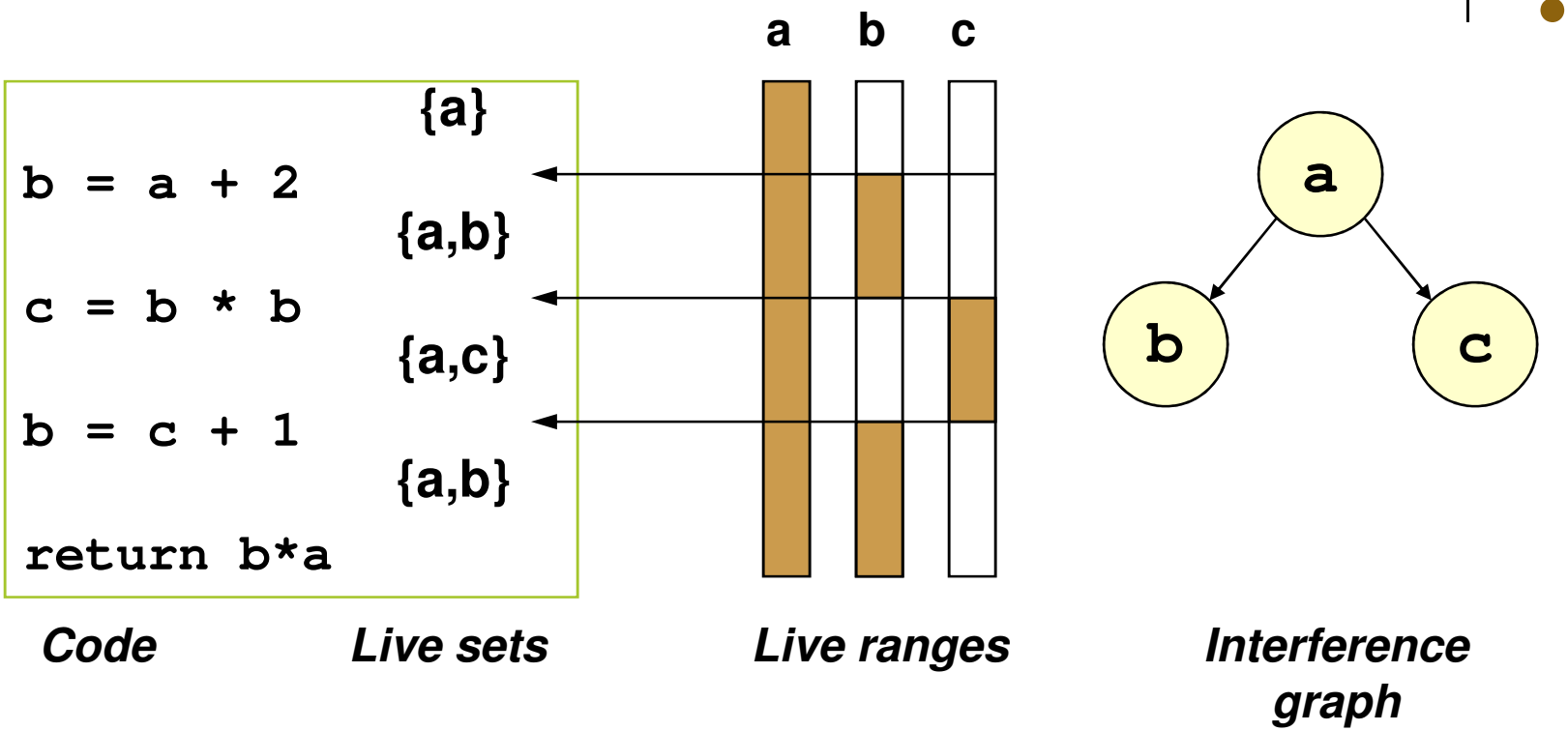


- Primary problems to be solved:
  - Usually more variables than registers
  - Can't use the same register for two variables that are live at the same time
- **Key insight**:

  We can cast this as a graph coloring problem ***(Lavrov, Chaitin)***
  - Nodes = program variables
  - Edges = connect variables that are live at the same time
  - "Interference graph" or "conflict graph"

  ⇨ Colors represent registers

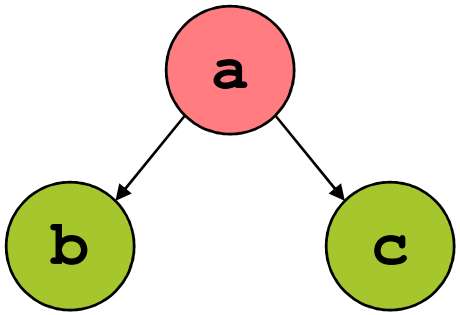
Example

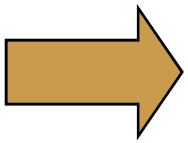


- **Key idea**: if we can color the graph with K colors, then we can allocate the variables to K registers

Example

- Graph is 2-colorable



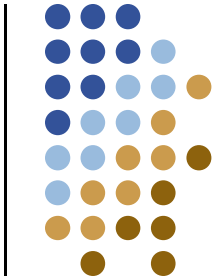


```
R2 = R1 + 2

R2 = R2 * R2

R2 = R2 + 1

return R2*R1
```

= Register 1 (R1)

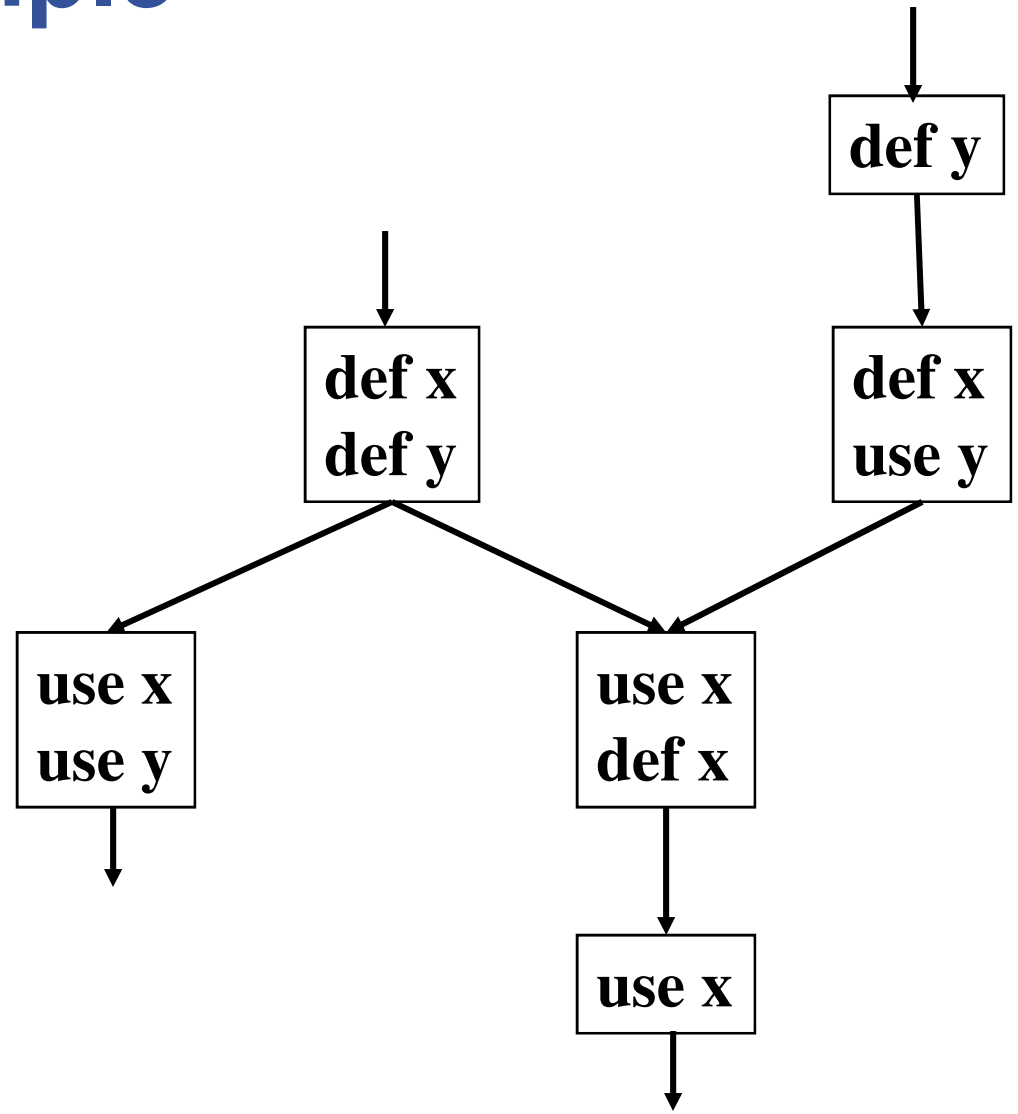
= Register 2 (R2)

Scope

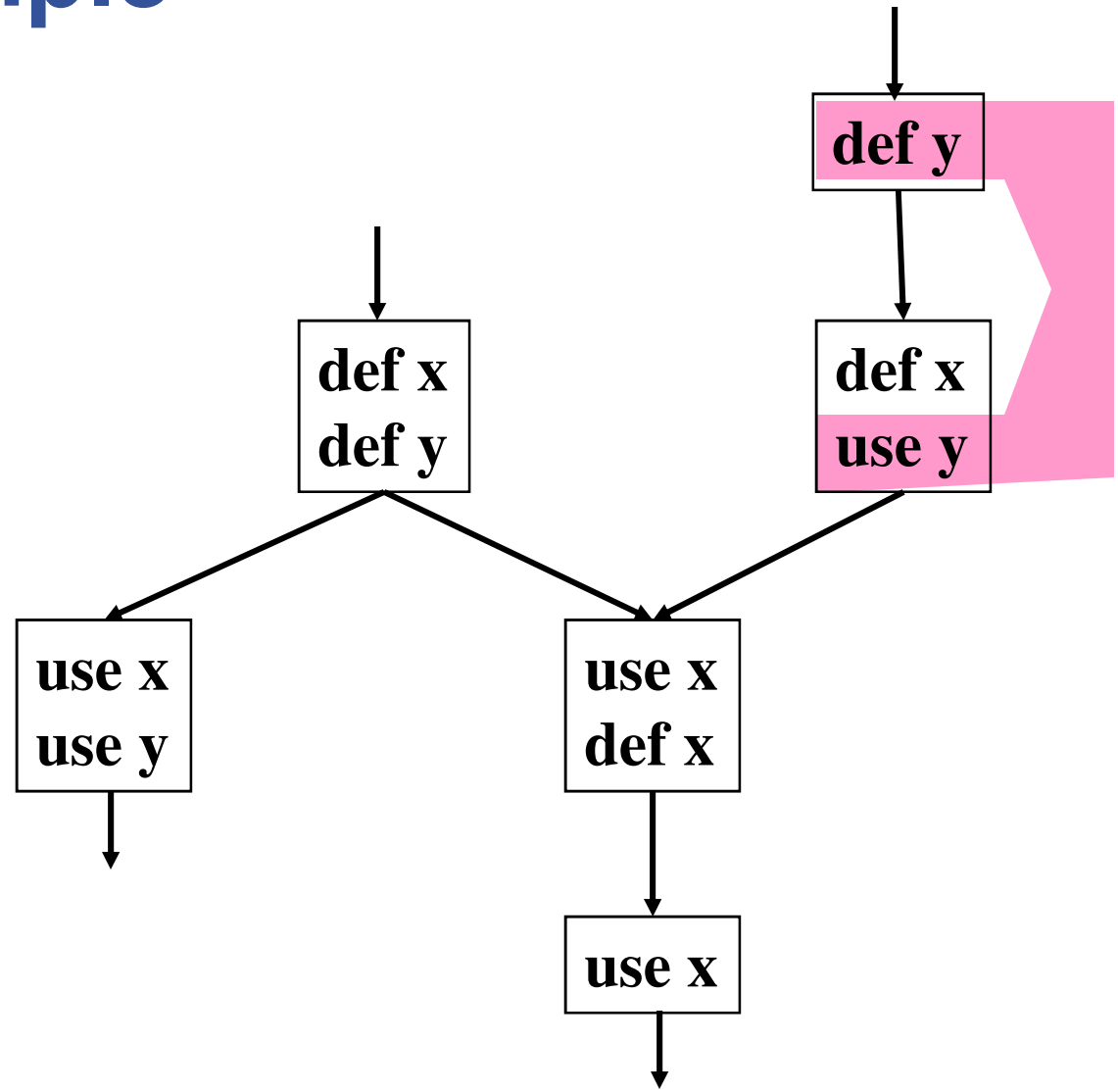


- Simple formulation:
  - Within a basic block – called ***local***
  - Live ranges are linear – just look at how they overlap
  - What to do at basic block boundaries?
    - Load all live vars into registers on entry
    - Store all live vars to memory on exit
- More sophisticated:
  - Across the control-flow graph – called ***global***
  - Consider live ranges as “webs” of dependences
  - **Key**: use the same graph coloring algorithm

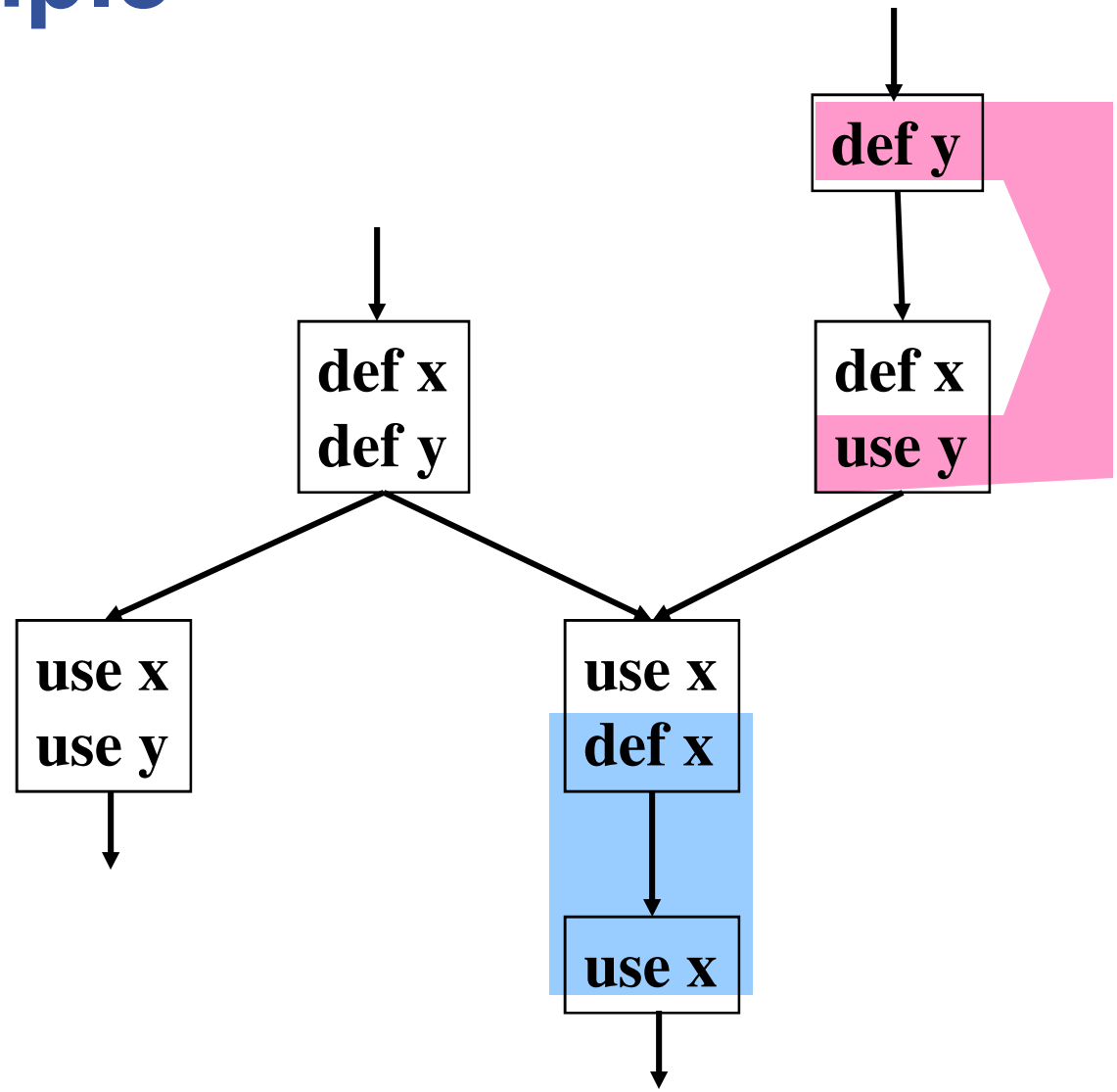
Example



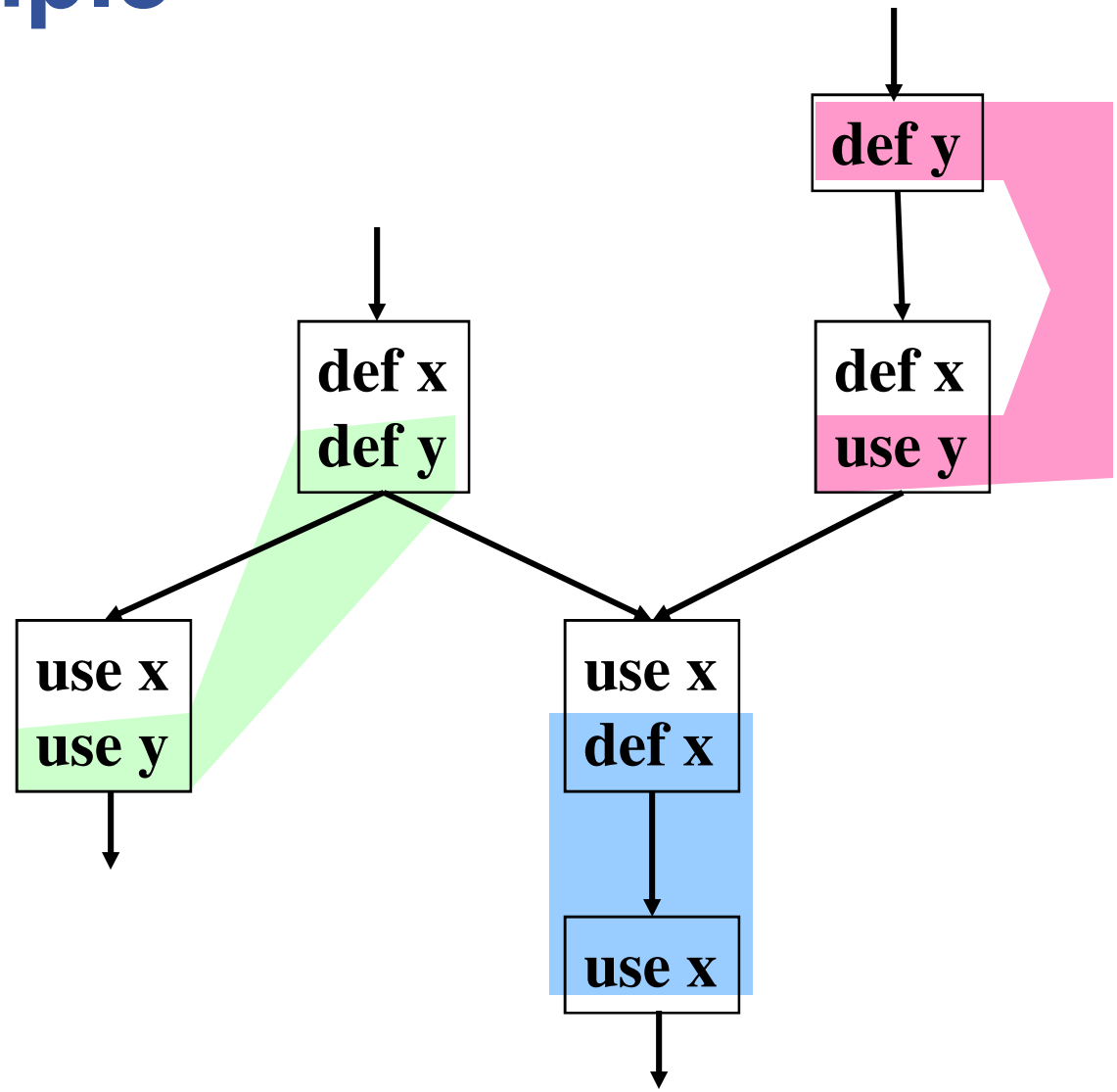
Example



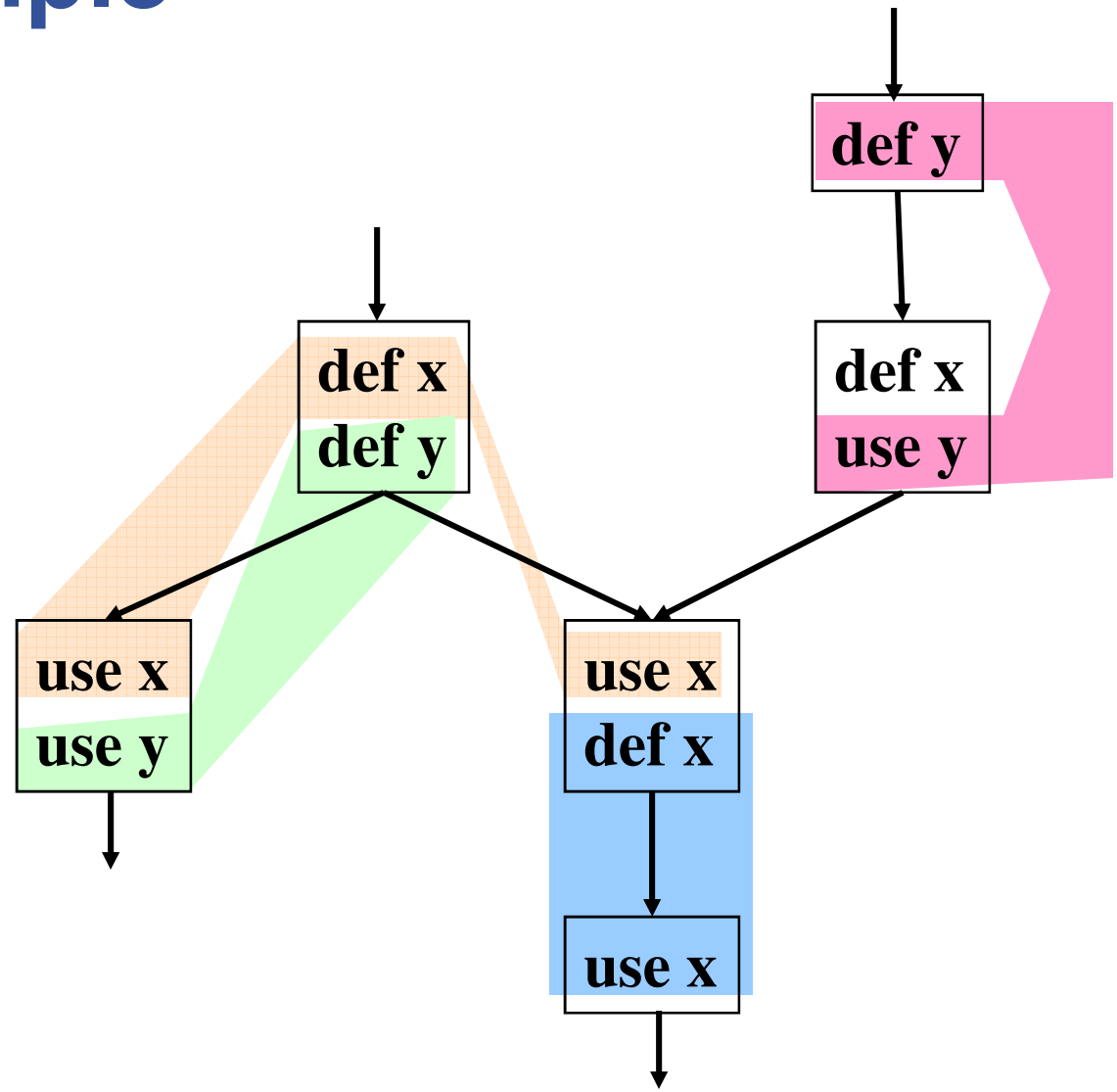
Example



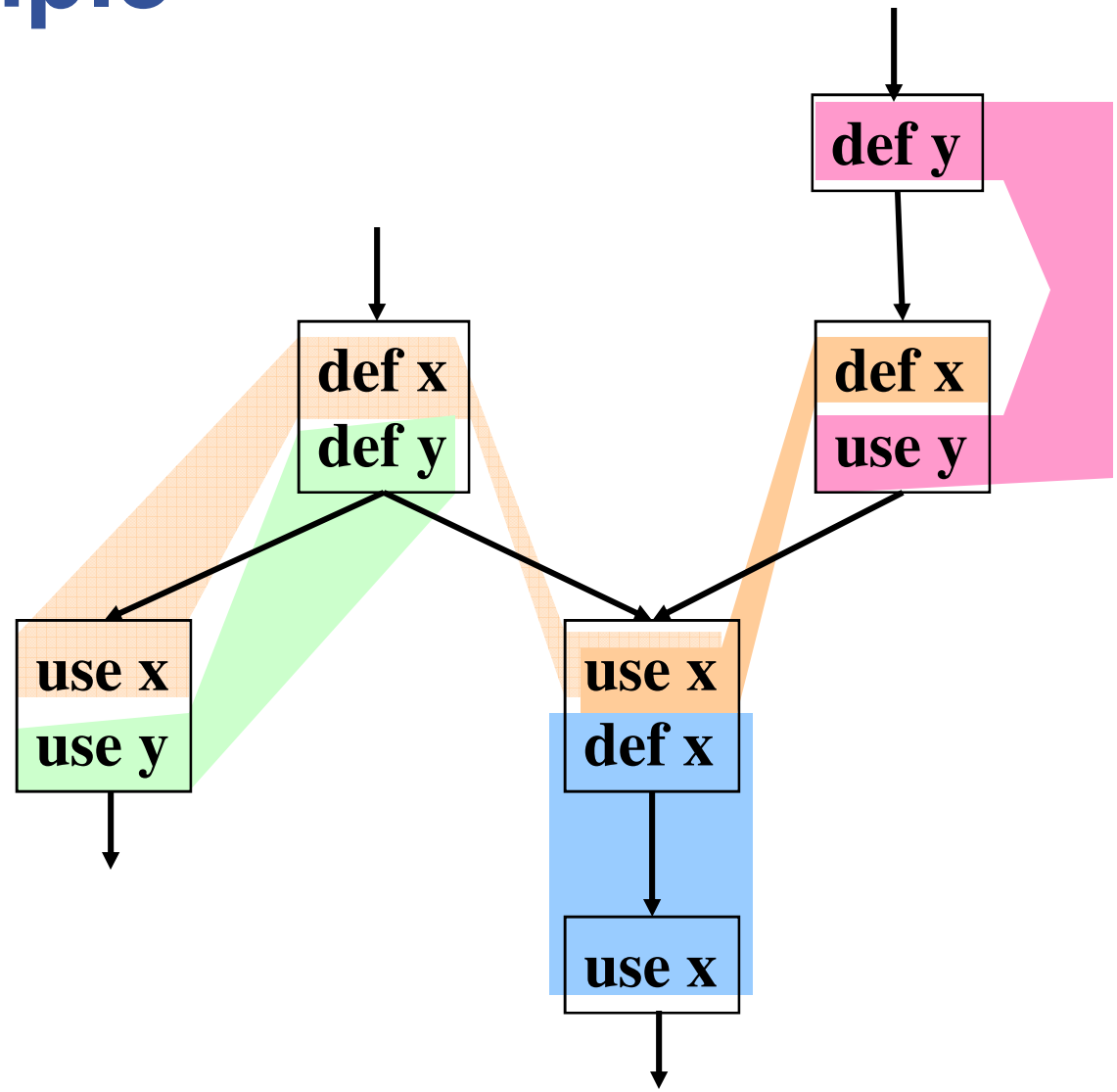
Example



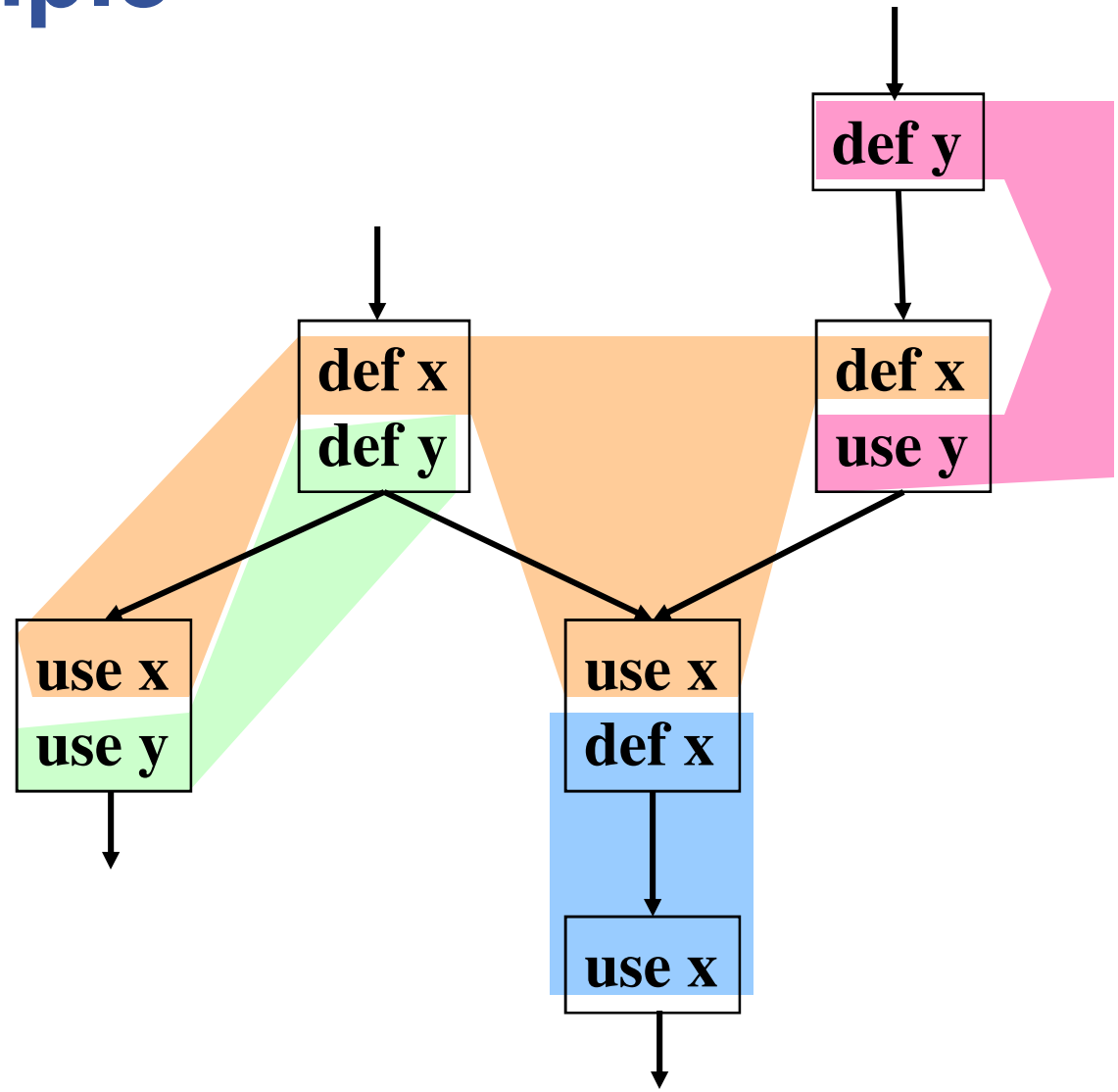
Example



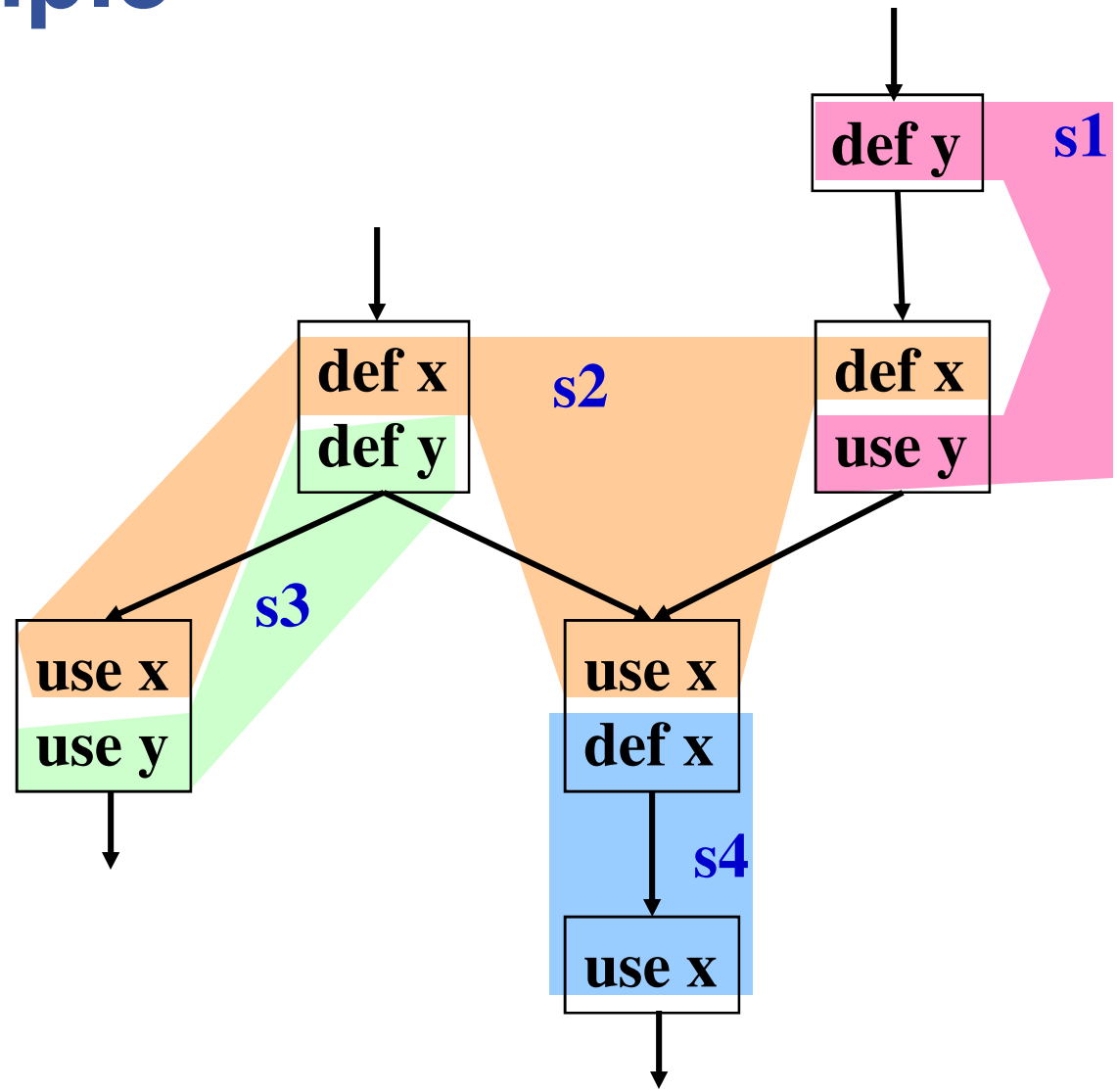
Example



Example

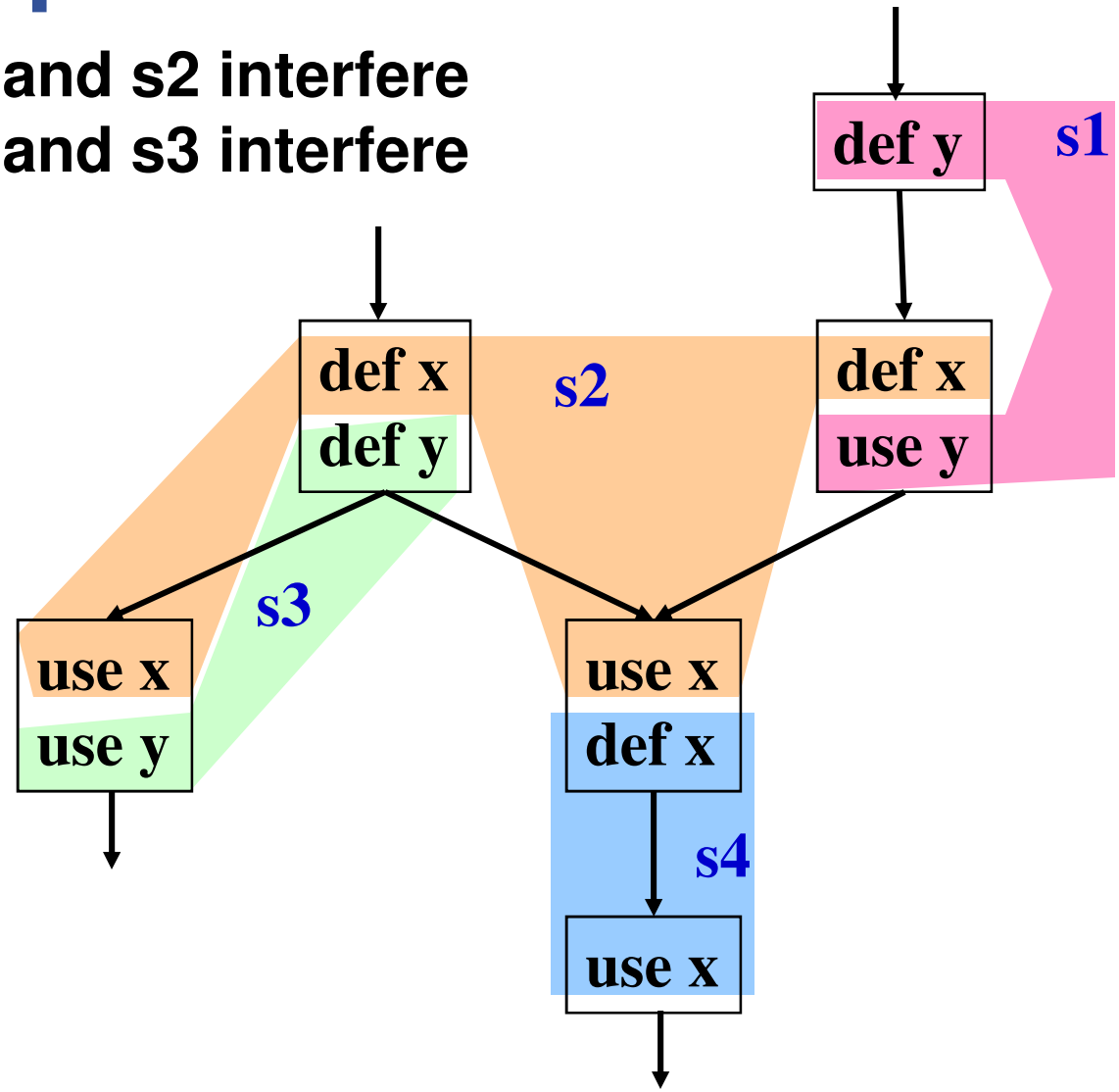


Example

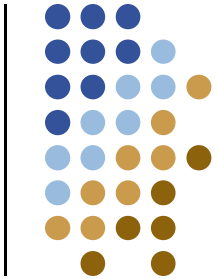


Example

Webs s1 and s2 interfere
Webs s2 and s3 interfere



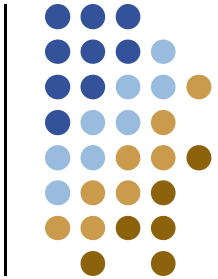
Graph coloring



The big questions:

- Can we efficiently find a ***K-coloring*** of the graph?
- Can we efficiently find the ***optimal coloring*** of the graph (i.e., using the least number of colors)?
- What do we do when there aren't enough colors (registers) to color the graph?

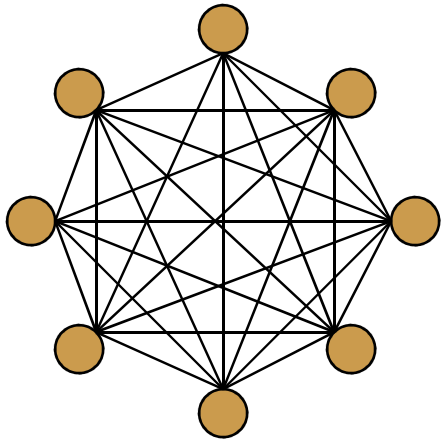
Graph coloring



- The bad news:

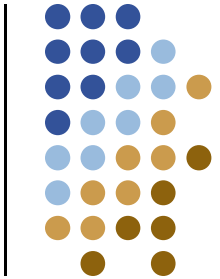
  Graph coloring is NP-complete

- Do we need the optimal algorithm?
  - Works on any graph
  - Tells us *for certain* if a graph is K-colorable



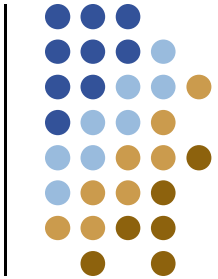
- Observations
  - We'll never see the worst-case graph
  - We don't necessarily need the perfect coloring
- Compute an approximation with heuristics

Spilling



- What if the graph is not K-colorable?
  - There aren't enough registers to hold all variables
  - Sadly: this happens a lot
- Pick a variable, ***spill*** it back to the stack
  - Value lives on the stack
  - Must generate extra code to load and store it
- Need registers to hold value temporarily
  - Simple approach: keep a few registers just for this purpose
  - Better approach:
    - Rewrite the code introducing a new temporary
    - Use the temporary to "load" and "store" the spilled variable
    - Rerun the liveness analysis and register allocation

Rewriting the code



- Example:

```
add     v1, v2
```

  - Suppose v2 is selected for spilling and assigned to stack location [SP-12]
  - Add a new variable t23 just for this instruction:

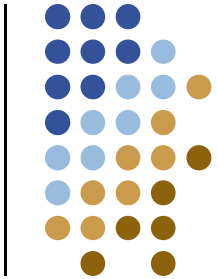
```
mov     [SP-12], t23
add      v1, t23
```

  - Rerun the whole algorithm

- **Idea**:

  t23 has a short live range and (hopefully) doesn't interfere with other variables as much as v2

Graph coloring



- Assume you have K registers

  Looking for K-coloring of interference graph

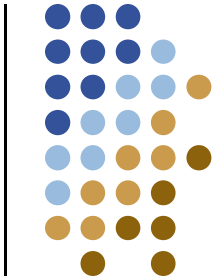
- **Observation**:

  Any node with less than K neighbors (***degree*** < K) must be colorable

  - Why?
  - Pick the color *not* used by any neighbor
  - There must be one!

- This is the basis for Chaitin's algorithm ***(Chaitin, 1981)***

Chaitin's algorithm



Idea:

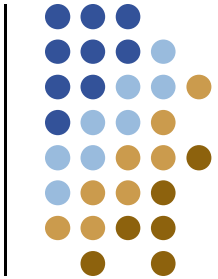
- Pick any vertex *n* with fewer than k neighbors

  This is a k-colorable vertex

- Remove that vertex from the graph
  - Also: remove incident edges
  - **Key**: this may result in some other nodes now having fewer than k neighbors
  - Now choose one of those vertices, continue…
- What if we get stuck?

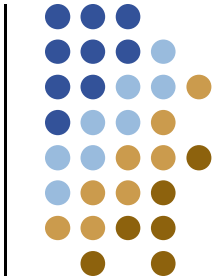
  Spill the variable whose node has more that k neighbors, and continue

Chaitin's Algorithm

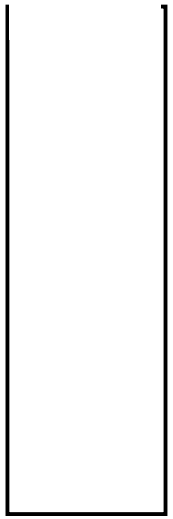


1. While ∃ vertices with < *k* neighbors in G_I
   - Pick any vertex *n* such that *n*°< *k* and put it on the stack
   - Remove that vertex and all edges incident to it from G_I
     - This will lower the degree of n's neighbors
2. If G_I *is non-empty* (all vertices have *k* or more neighbors) then:
   - Pick a vertex *n* (using some heuristic) and spill the live range associated with *n*
   - Remove vertex *n* from G_I, along with all edges incident to it
   - If this causes some vertex in G_I to have fewer than *k* neighbors, then go to step 1; otherwise, repeat step 2
3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbor

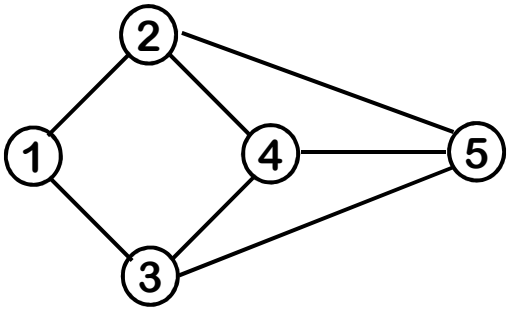
Chaitin's Algorithm in Practice



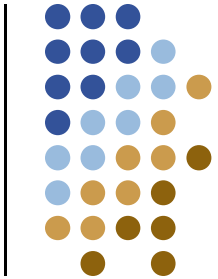
3 Registers



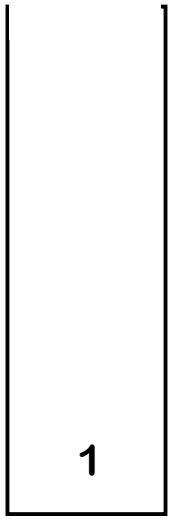
Stack



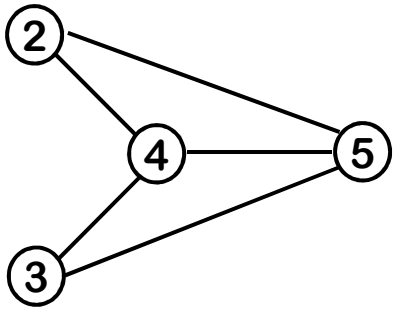
Chaitin's Algorithm in Practice



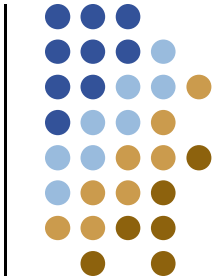
3 Registers



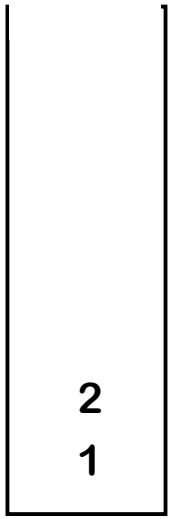
Stack



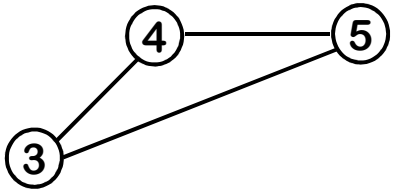
Chaitin's Algorithm in Practice



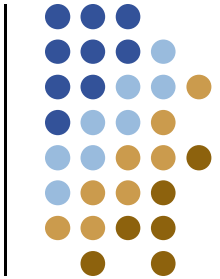
3 Registers



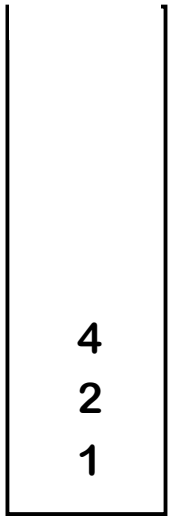
Stack



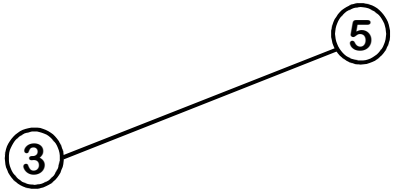
Chaitin's Algorithm in Practice



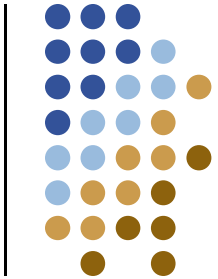
3 Registers



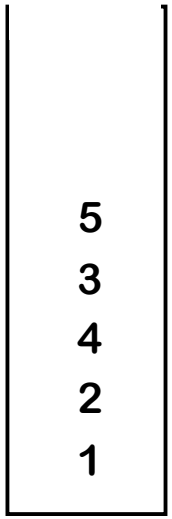
Stack



Chaitin's Algorithm in Practice



3 Registers



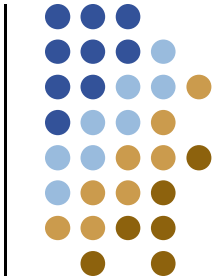
Colors:

1:

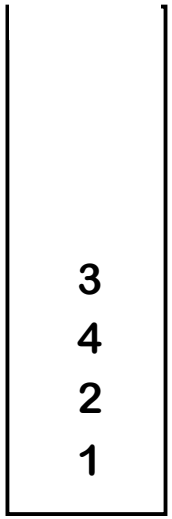
2:

3:

Chaitin's Algorithm in Practice



3 Registers



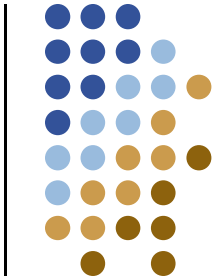
Colors:

1: 

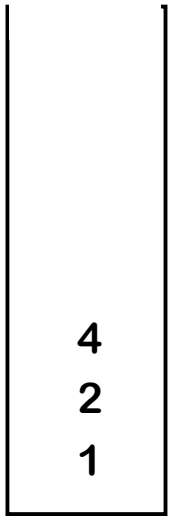
2: 

3: 

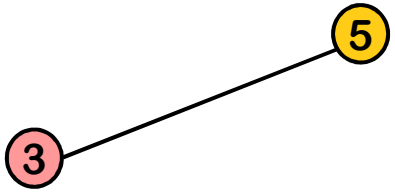
Chaitin's Algorithm in Practice



3 Registers



Stack



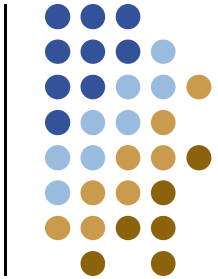
Colors:

1:

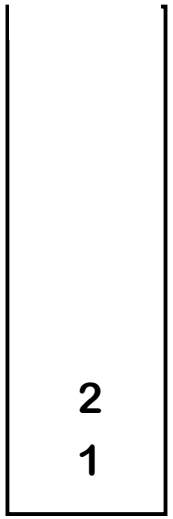
2:

3:

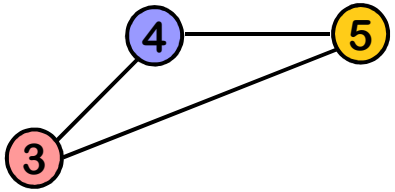
Chaitin's Algorithm in Practice



3 Registers



Stack



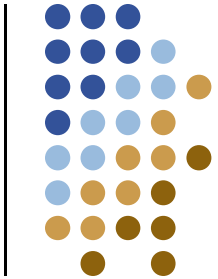
Colors:

1:

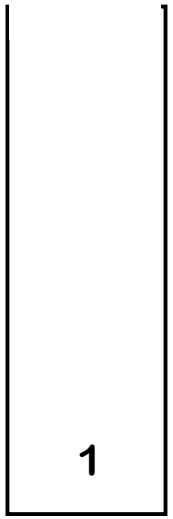
2:

3:

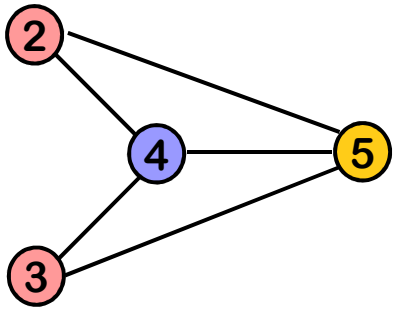
Chaitin's Algorithm in Practice



3 Registers



Stack



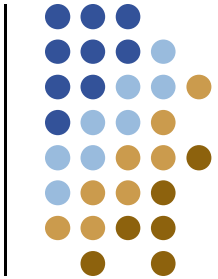
Colors:

1:

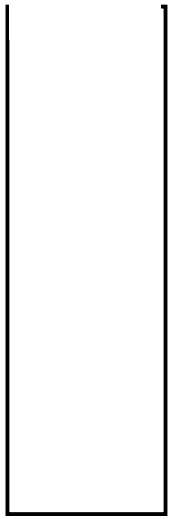
2:

3:

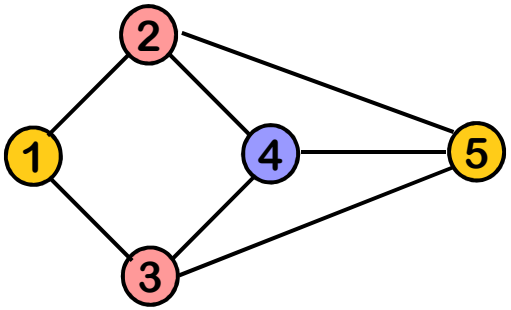
Chaitin's Algorithm in Practice



3 Registers



Stack



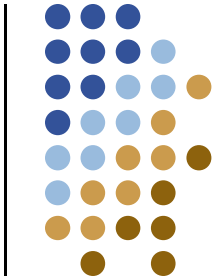
Colors:

1:

2:

3:

Improvements

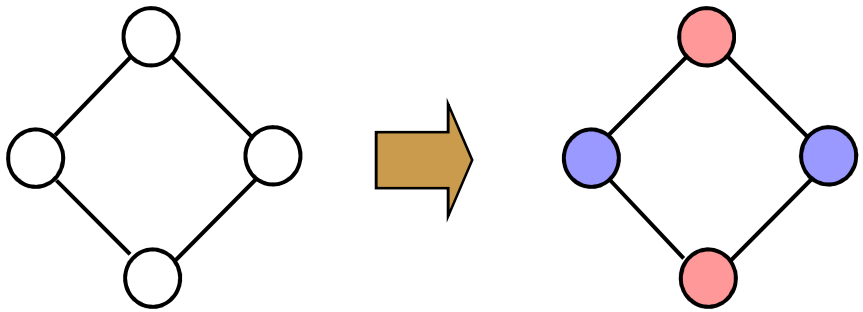


Optimistic Coloring

(*Briggs, Cooper, Kennedy, and Torczon*)

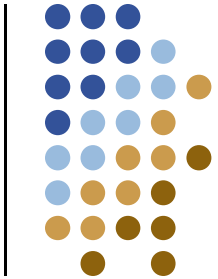
- Observation:
  - Some graphs may be k-colorable, even though all vertices have k neighbors
  - Example:

2 Registers:



2-colorable

Improvements

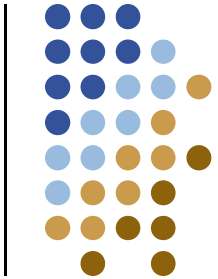


Optimistic Coloring

(*Briggs, Cooper, Kennedy, and Torczon*)

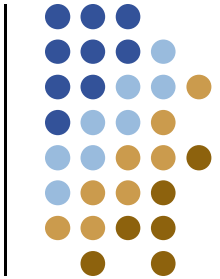
- Idea:
  - Don't spill when we get stuck
  - Remove k-neighbor vertices, as usual
  - Push on stack in some priority order
  - If popping and coloring fails, then spill and start over

Chaitin-Briggs Algorithm



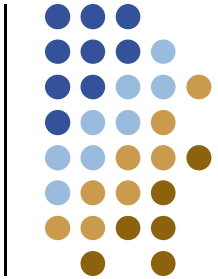
1. While ∃ vertices with < k neighbors in G_I
   - Pick any vertex n such that $n°$< k and put it on the stack
   - Remove that vertex and all edges incident to it from G_I
2. If G_I is non-empty (all vertices have k or more neighbors) then:
   - Pick a vertex n (using some heuristic condition), **push *n* on the stack** and remove n from G_I, along with all edges incident to it
   - If this causes some vertex in G_I to have fewer than k neighbors, then go to step 1; otherwise, repeat step 2
3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbor
   - If some **vertex cannot be colored**, then pick an uncolored vertex to spill, **spill it**, and restart at step 1

Picking a spill candidate

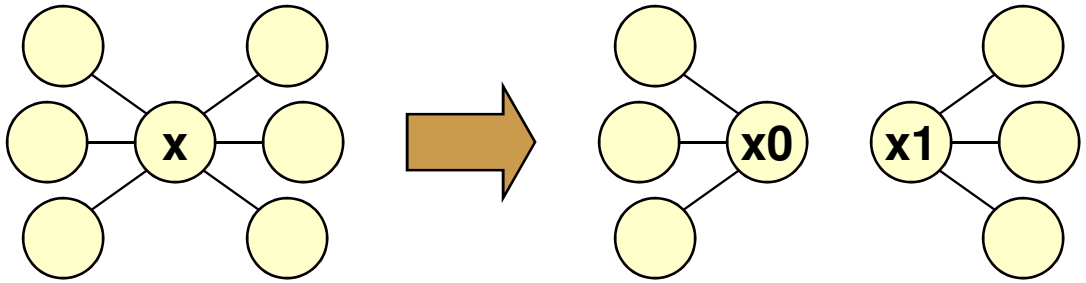


- How important is choosing a spill candidate?
- **Goal**: minimize the performance impact
  - Spilled variable is stored at each def, loaded at each use
  - Higher degree nodes interfere with more variables
  - Chaitin: minimize spill cost ÷ current degree
- Many subtle variations
  - Live range splitting
  - More sophisticated spill cost estimation
  - Impact on rest of the coloring problem
  - Interaction with other optimizations – scheduling, copy propagation

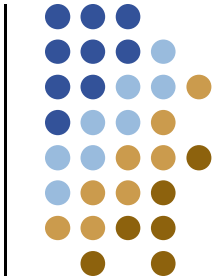
More spilling



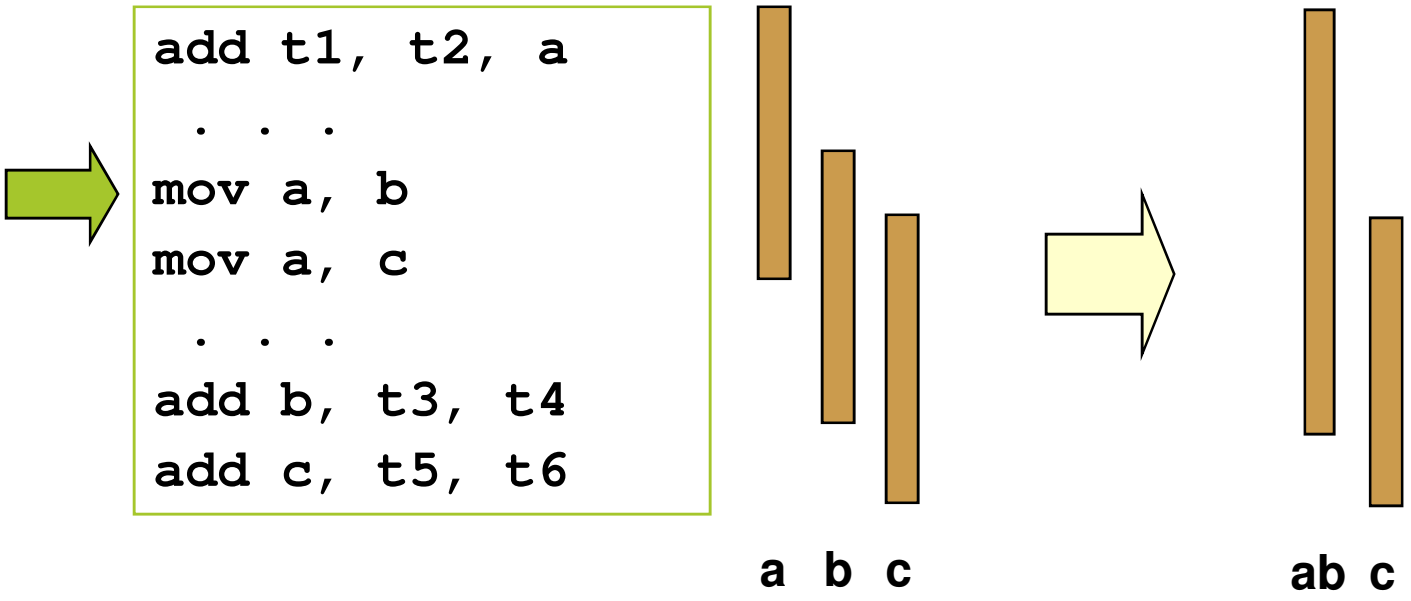
- **Problem**:
  - This approach turns a single large live range into many small live ranges with many loads and stores
  - Can we do better?
- ***Live range splitting***
  - Choose a point in the live range -- insert a store followed by a load
  - Divides the live range into two (or more pieces)
  - **Key**: choose carefully to reduce the degree of nodes



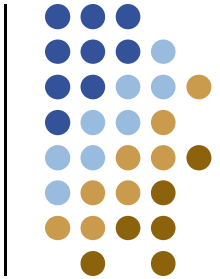
Another improvement



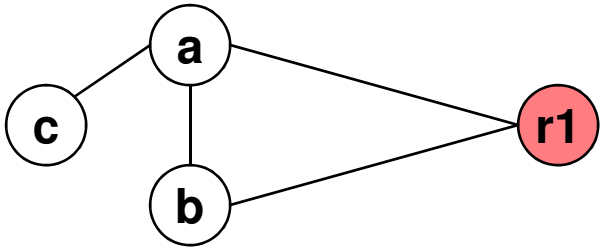
- Register ***coalescing***
  - We may be able to reduce the degree of vertices by merging live ranges that are connected only by a copy
  - **Idea**:
    - Find a register copy “tb = ta”
    - If ta and tb do not interfere, combine their live ranges



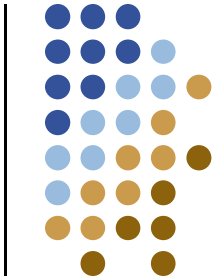
Allocation constraints



- How do we deal with architectural constraints?
  - Register types (floating point versus integer)
  - Reserved registers – the stack pointer
  - Instruction-level constraints
    - Instruction requirements – x86 mul must use eax, adx
- We can encode constraints in the graph
  - Precolored nodes (for required registers)
  - Additional nodes and edges for constraints
  - Example: explicit nodes for physical registers



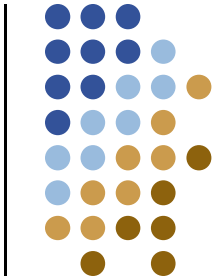
Bin Packing



Different approach

- What is the bin packing problem?
  - Some number of objects of different “weights” or “volumes”
  - Series of bins of fixed size
  - Pack objects using the fewest number of bins
  - How hard is this problem?
- Mapping to register allocation?
  - “Objects” are live ranges
  - “Bins” are registers
- Can use existing bin packing approximations

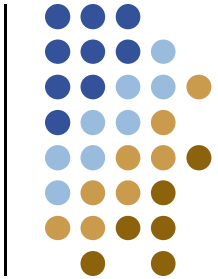
Another approach



- What if graph coloring and bin packing are still too expensive?
- How big can interference graph get?
  - Worst-case quadratic size (edges)

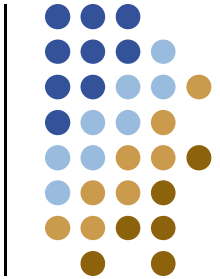
  Example: in a just-in-time compiler
  - Compilation time is critical
  - Compiler needs to be simple and fast
- **Alternative**: ***Linear scan*** *register allocation* ***(Poletto, 1999)***
  - Make one pass over the list of variables
  - Spill variables with longest lifetimes – those that would tie up a register for the longest time

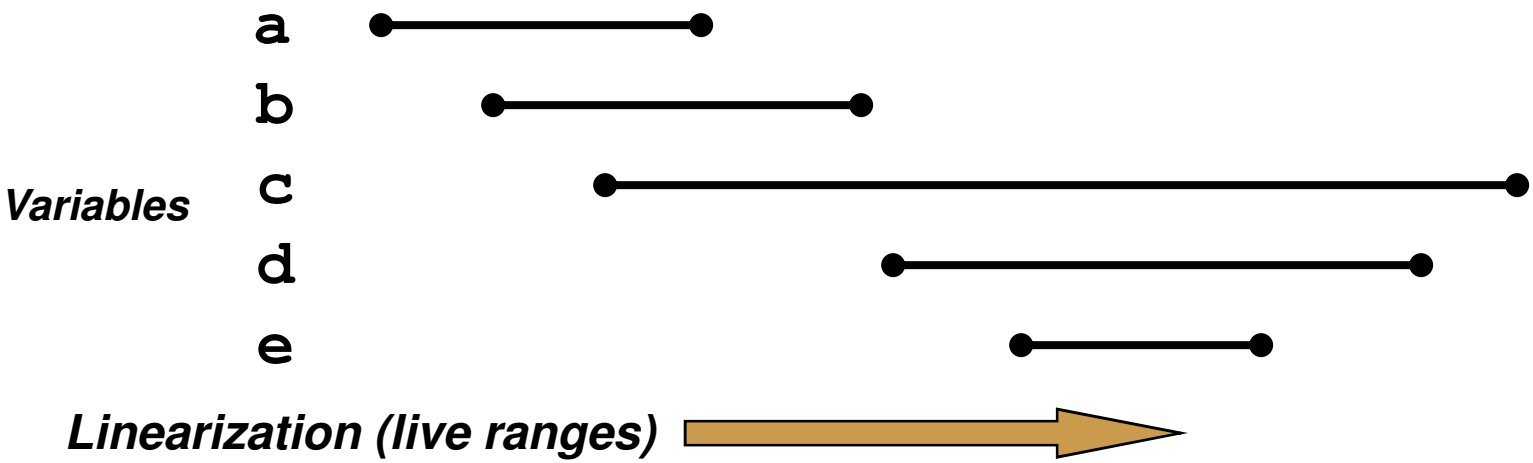
Linear scan



- First: Compute live intervals
  - Linearize the IR – usually just a list of tuples/instructions
  - A ***live interval*** for a variable is a range [i,j]
    - The variable is not live before instruction i
    - The variable is not live after instruction j
- **Idea**: overlapping live intervals imply interference
  - Given R registers and N overlapping intervals
    - R intervals allocated to registers
    - N-R intervals spilled to the stack
  - What does this mean about the linearization?
- **Key**: choosing the right intervals to spill

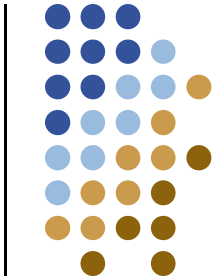
Example





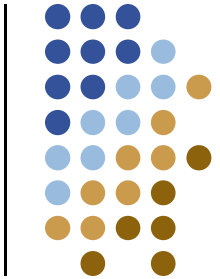
- How many registers do we need?
- What would the interference graph look like?
- What if we only have two registers?

Algorithm



- Sort live intervals
  - In order of increasing start points
  - Quickly find the next live interval in order
- Maintain a sorted list of ***active*** intervals
  - In order of increasing end points
  - Quickly find expired intervals
- At each step, update ***active*** as follows
  - Add the next interval from the sorted list
  - Remove any expired intervals *(those whose end points are earlier that the start point of the new interval)*

Algorithm



- Extra restriction:

  Never allow ***active*** to have more than R elements

- Spill scenario:

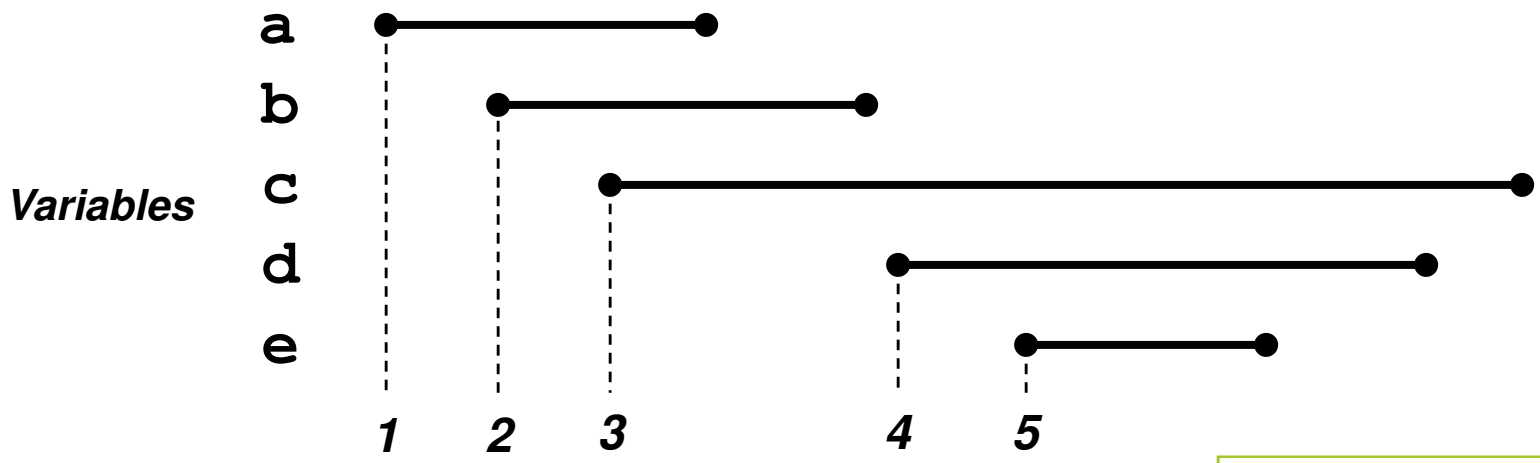
  active has R elements, new interval doesn't cause any existing intervals to expire

- Heuristic:

  Spill the interval that ends last (furthest from current position)

  - Has optimal behavior for straight-line code
  - Appears to work well even in linearized code

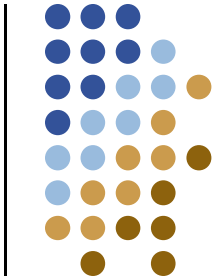
Example (2 registers)



Allows this code to use 2 registers, with one spill

- Step 1: *active* = {a}
- Step 2: *active* = {a,b}
- Step 3: *active* = {a,b,c} → spill c → *active* = {a,b}
- Step 4: a and b expire, *active* = {d}
- Step 5: *active* = {d,e}

Linear scan



- Register allocation
  - Each new interval added to active gets the next register
  - Registers freed as intervals are removed
- Resulting code:

  within 10% of graph coloring
- Compilation time:

  2 – 3 times faster than graph coloring
- Architectural considerations
  - How sensitive is architecture to register allocation?
  - Many registers (Alpha, PowerPC): use linear scan
  - Few registers (x86): use graph coloring