



**NTNU – Trondheim**  
Norwegian University of  
Science and Technology

## Register allocation

# Variables vs. registers

- TAC has any number of variables
- Assembly code has to deal with memory and registers
- Compiler back end must decide how to juggle the contents of the memory and registers to fit every variable as necessary



# Straightforward solution

- We know how to do this, just
  - Put everything in the activation record
  - For each instruction, shuttle variables into registers
  - Combine registers
  - Put variables back into activation record
- That's fine and dandy, but it creates
  - Redundant copy instructions
  - Constant memory traffic



# Register allocation

- Goal: keep variables in registers as long as possible
- In the best of cases, a variable can be in a register throughout its lifetime
- If it can't, it'll need a place in the activation record

# What can go in registers?

- That depends on the number of registers
- It also depends on how variables are being used  
(You can't make a pointer to a register)

- Main idea:

*Two variables can't share the same register if they are live simultaneously*



# The basic approach

- Do live variable analysis
- Go through the sets of live variables
- When two variables appear in the same set, they *interfere*
  - Conversely, when two variables don't interfere, their live ranges are disjoint
  - Pairs like that can share the same register, because they won't need it at the same time

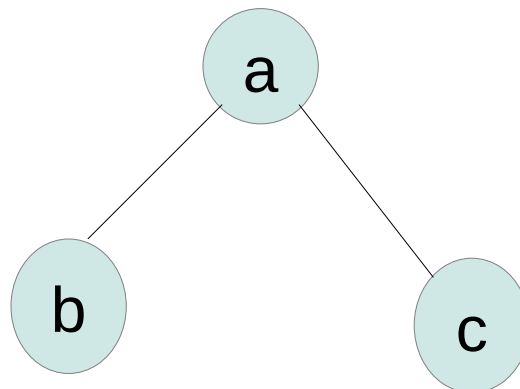


# Interference graphs

- An interference graph is a graph where
  - every variable is a node
  - edges connect interfering variables

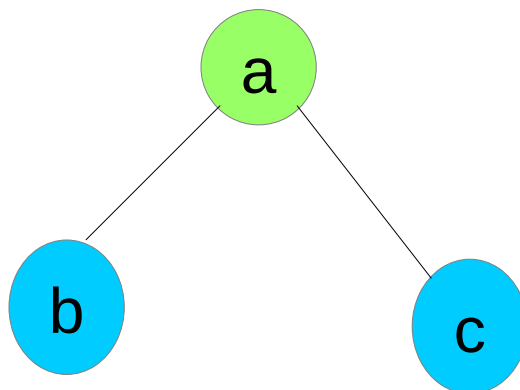
as in this one:

```
    {a}
b = a + 2
    {a, b}
c = b * b
    {c, a}
b = c + 1
    {b, a}
return b*a
    {}
```



# Graph coloring

- If every register has a color, a register assignment of the interference graph is a mapping where no two neighbors have the same color



RAX  
RBX



# Um... “colors”?

- Graph “coloring” is one of the classic problems of computer science
- If we have  $k$  registers, the question of whether each variable can have one is the same as whether the interference graph is *k-colorable*
- K-colorability is an NP complete problem, finding an optimal solution takes exponential time  
(as far as we know today)
- We can still approximate it with an imperfect heuristic



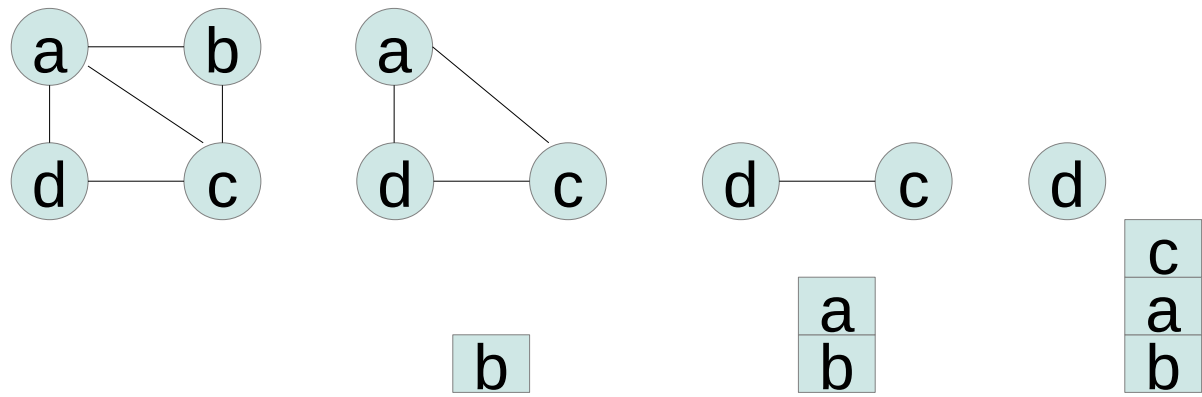
# Practical k-coloring

- Simplify the graph
  - Find a node with at most  $k-1$  edges
  - Remove it from the graph, put it on a stack
  - Repeat until simplified graph is trivially k-colorable (or, until there are no nodes left, if you prefer)
- Reintroduce the nodes
  - Add nodes back (in reverse order of the simplification)
  - Color them with colors that they don't interfere with
  - Hope that total number of colors is  $k$  or smaller

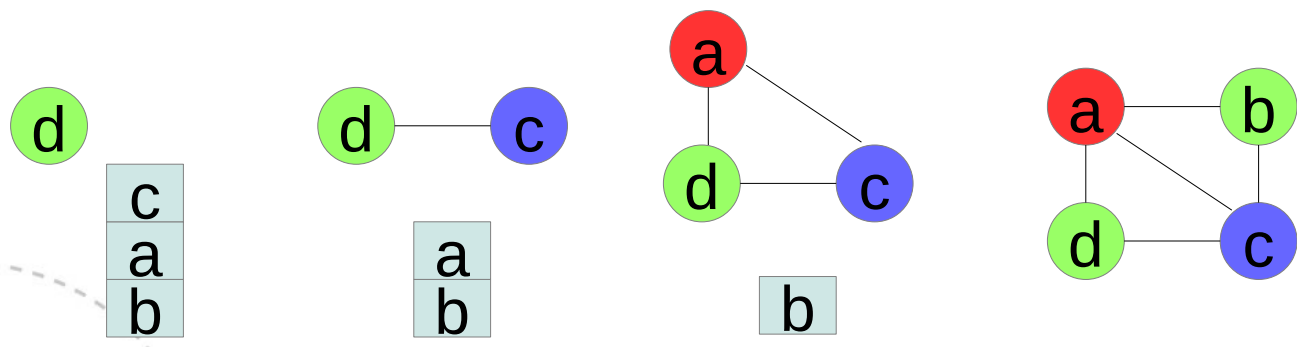


# Sometimes it works

- Is this graph 3-colorable?



(simplify)



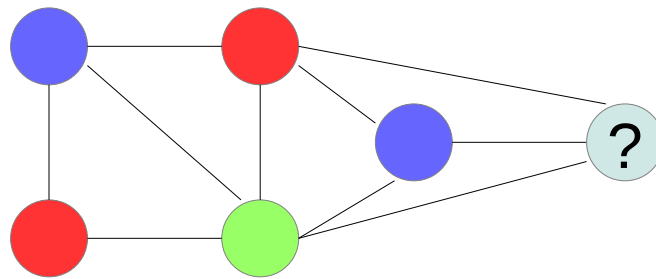
(reconstruct)



NTNU - Trondheim  
 Norwegian University of  
 Science and Technology

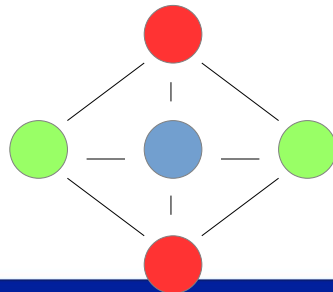
# Sometimes it doesn't

- If the graph can't be colored, it'll find a form where every node has  $k$  or more neighbors  
(otherwise, there'd be a color to spare for them)



3 neighbors  
spent all 3 colors

- $K$  or more neighbors doesn't imply uncolorability



# Spilling

- When all nodes have  $k$  or more neighbors, pick one and mark it for *spilling*  
(a place in the activation record)
- Remove from graph, push on stack
- Aim for little-used nodes



# Access to spilled variables

- Some additional instructions will be needed to move spilled variables back and forth to the activation record
- Simple: keep a few extra registers for shuttling data in the load-modify-store way
- Better: rewrite low-IR code with new temporary, redo liveness and register allocation

# Precolored nodes

- Some variables need designated registers  
(e.g. “return value goes in RAX”)
- Treat their temporaries as special, and set their colors in the interference graph
- Simplification: Never remove pre-colored nodes  
(They don't need to be reintroduced to get a color anyway)
- Coloring: Use the pre-colored nodes as starting point when reintroducing the rest



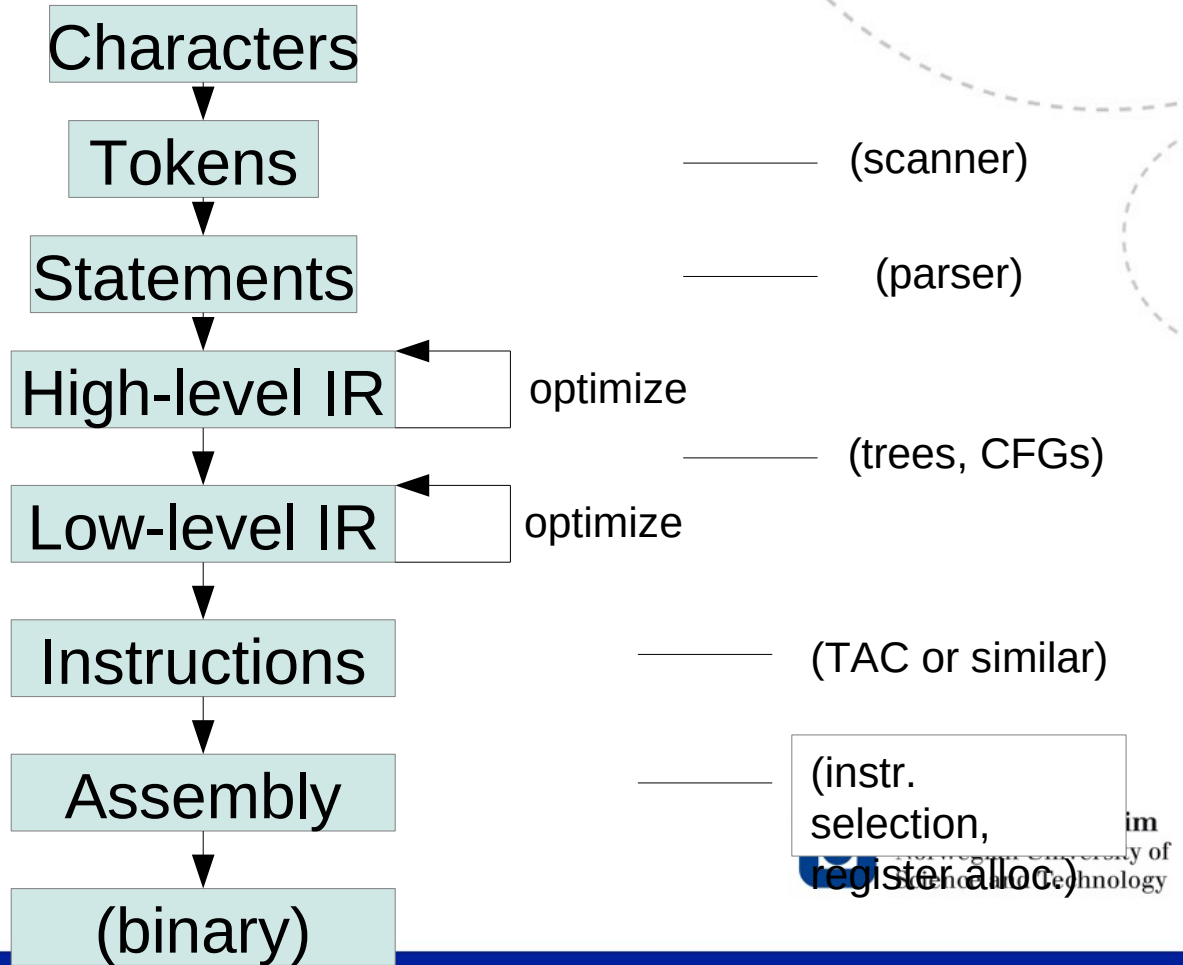
# Big picture of code generation

- Start from low-level IR
- Build DAG of the computation
  - Global variables = static addresses
  - Arguments taken from frame pointer
  - Assume all locals and temporaries in (infinite number of) registers
- Tile the DAG, obtaining abstract assembly
- Allocate registers
  - Liveness analysis of abstract assembly
  - Assign registers and generate assembly





# The whole process



# The final sessions

- At the end, we'll revisit all those topics, and speed-review key elements of the techniques we've covered