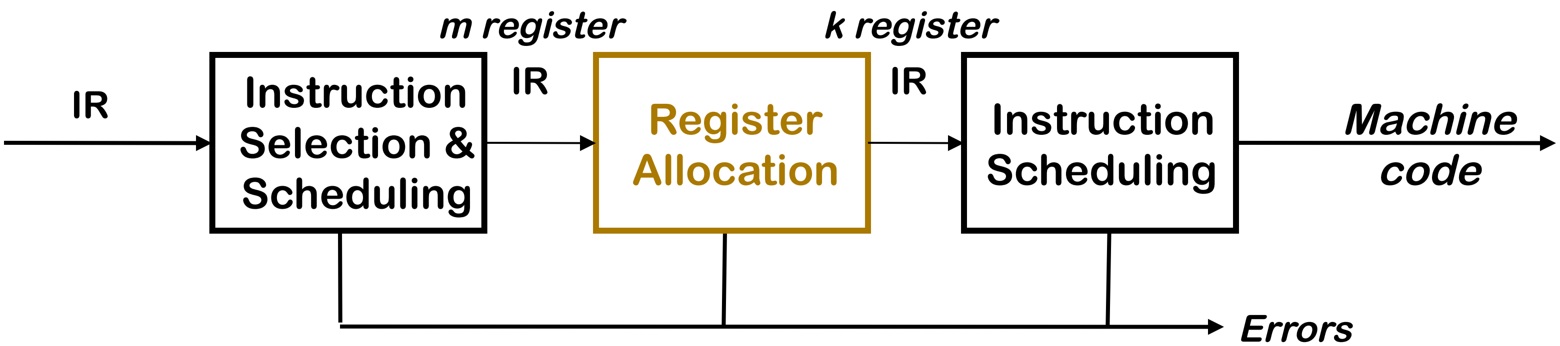
Global Register Allocation via Graph Coloring and Wrap Up

Register Allocation

Part of the compiler's back end



Critical properties

- Produce correct code that uses *k* (or fewer) registers
- Minimize added loads and stores
- Minimize space used to hold *spilled values*
- Operate efficiently

  $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$

Global Register Allocation

The big picture

Optimal global allocation is NP-Complete, under almost any assumptions.

At each point in the code

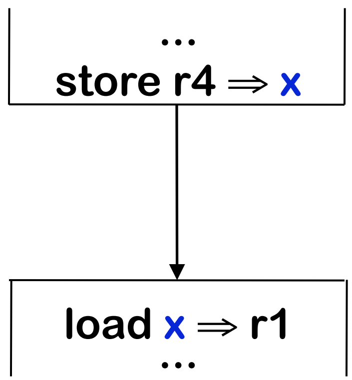
1. Determine which values will reside in registers
2. Select a register for each such value

The goal is an allocation that "minimizes" running time

Most modern, global allocators use a graph-coloring paradigm

- Build a "conflict graph" or "interference graph"
- Find a k-coloring for the graph, or change the code to a nearby problem that it can k-color

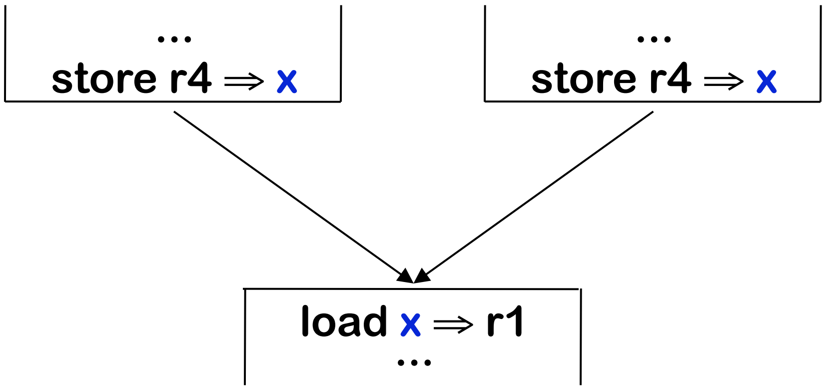
Global Register Allocation



What's harder across multiple blocks?

- Could replace a load with a move
- Good assignment would obviate the move
- Must build a control-flow graph to understand inter-block flow
- Can spend an inordinate amount of time adjusting the allocation

Global Register Allocation



What if one block has x in a register, but the other does not?

A more complex scenario

- Block with multiple predecessors in the control-flow graph
- Must get the "right" values in the "right" registers in each predecessor
- In a loop, a block can be its own predecessors

This adds tremendous complications

Global Register Allocation

Taking a global approach

- Make systematic use of registers or memory
- Adopt a general scheme to approximate a good allocation

Graph coloring paradigm (*Lavrov & (later) Chaitin*)

1 Build an interference graph G_I for the procedure
   - → Computing LIVE is harder than in the local case
   - → G_I is not an interval graph

2 (try to) construct a k-coloring
   - → Minimal coloring is NP-Complete
   - → Spill placement becomes a critical issue

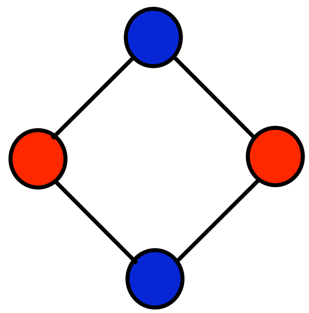
3 Map colors onto physical registers

Graph Coloring (A Background Digression)

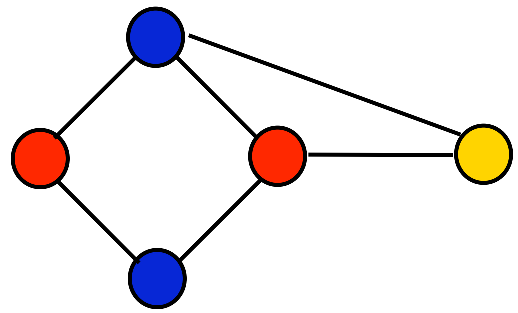
The problem

> *A graph G is said to be k-colorable iff the nodes can be labeled with integers 1… k so that no edge in G connects two nodes with the same label*

Examples



2-colorable



3-colorable

Each color can be mapped to a distinct physical register

Observation on Coloring for Register Allocation

- Suppose you have k registers—look for a k coloring
- Any vertex n that has fewer than k neighbors in the interference graph ($n^\circ < k$) can always be colored!
  - Pick any color not used by its neighbors — there must be one
- Ideas behind Chaitin's algorithm:
  - Pick any vertex n such that $n^\circ < k$ and put it on the stack
  - Remove that vertex and all edges incident from the interference graph
    - This may make some new nodes have fewer than k neighbors
  - At the end, if some vertex n still has k or more neighbors, then spill the live range associated with n
  - Otherwise successively pop vertices off the stack and color them in the lowest color not used by some neighbor

Chaitin's Algorithm

1. While ∃ vertices with < k neighbors in G_I
   - Pick any vertex n such that $n^\circ < k$ 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 and put it on the stack
   - 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 Allocator (Bottom-up Coloring)

renumber — Build SSA, build live ranges, rename

build — Build the interference graph

coalesce — Fold unneeded copies

$LR_x \rightarrow LR_y$, and $< LR_x, LR_y > \notin G_I \Rightarrow$ combine LR_x & LR_y

spill costs — Estimate cost for spilling each live range

simplify — Remove nodes from the graph

```
while N is non-empty
    if ∃ n with n° < k then
        push n onto stack
    else pick n to spill
        push n onto stack
    remove n from G_I
```

select — While stack is non-empty

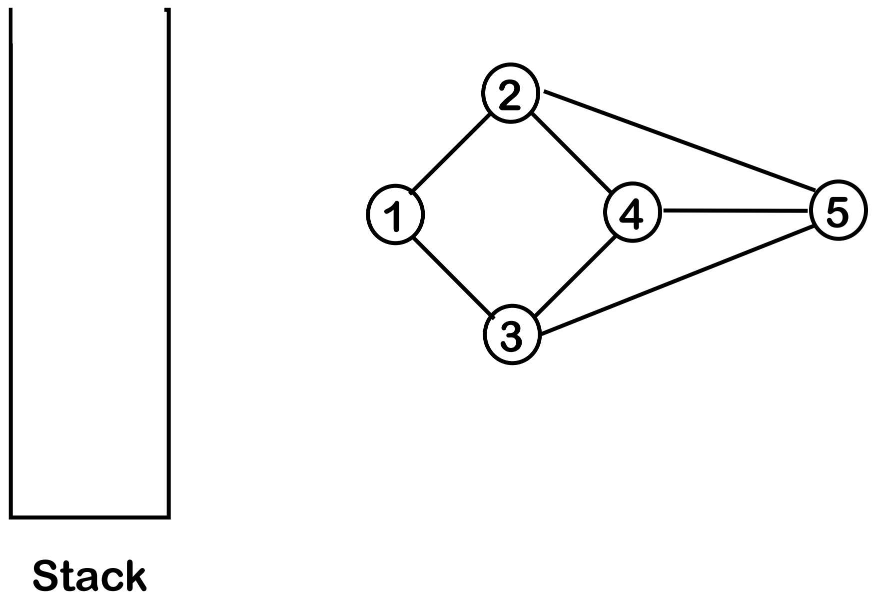
pop n, insert n into G_I, & try to color it

spill — Spill uncolored definitions & uses

Chaitin's algorithm

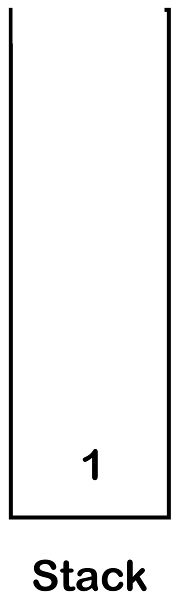
Chaitin's Algorithm in Practice

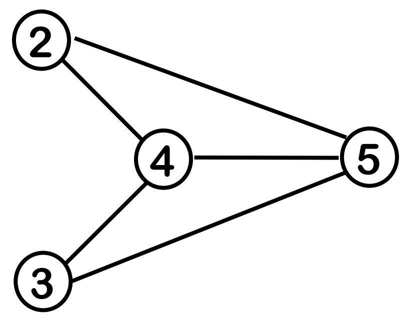
3 Registers



Chaitin's Algorithm in Practice

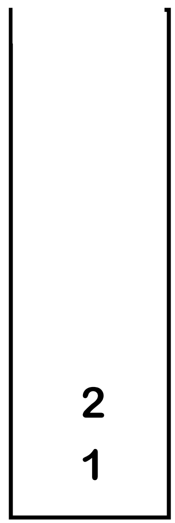
3 Registers



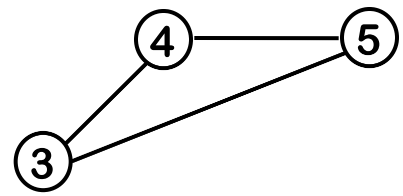


Chaitin's Algorithm in Practice

3 Registers

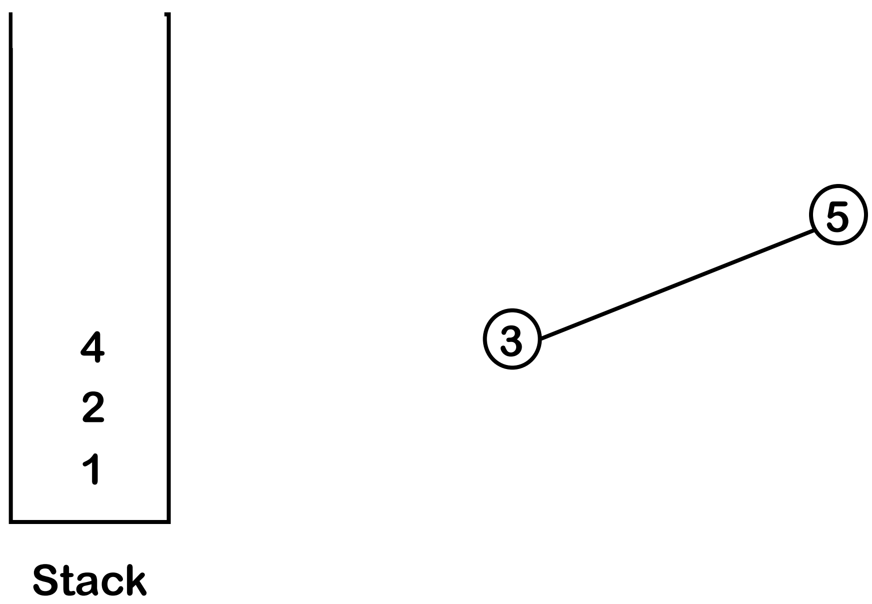


Stack



Chaitin's Algorithm in Practice

3 Registers



Chaitin's Algorithm in Practice

3 Registers

5
3
4
2
1

Stack

Colors:

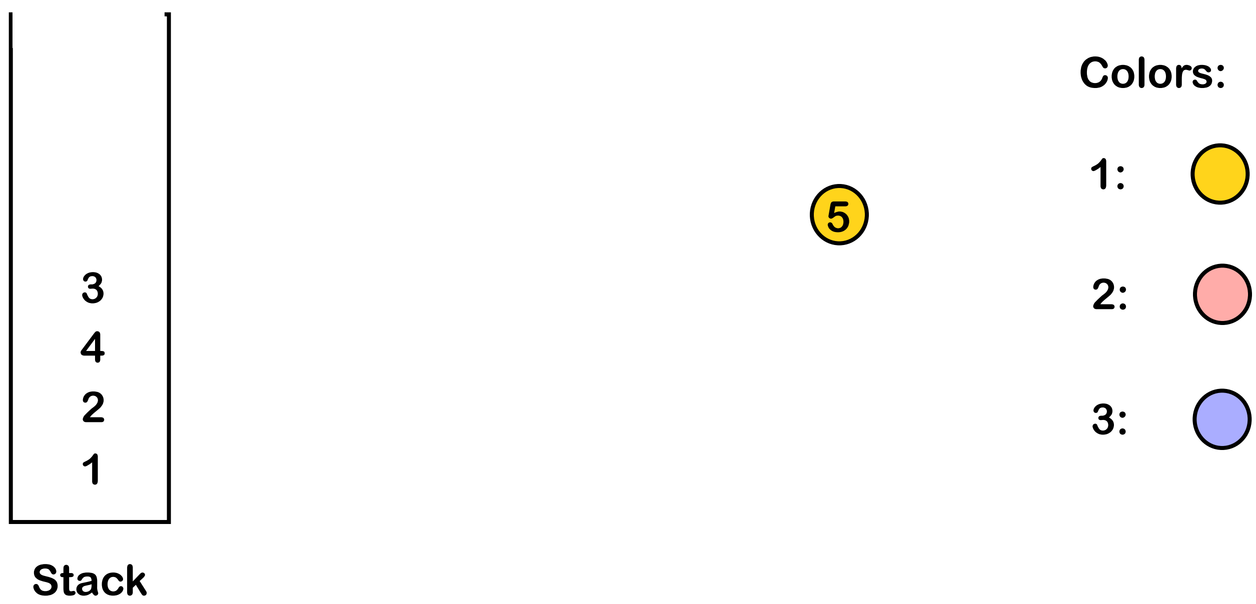
1:

2:

3:

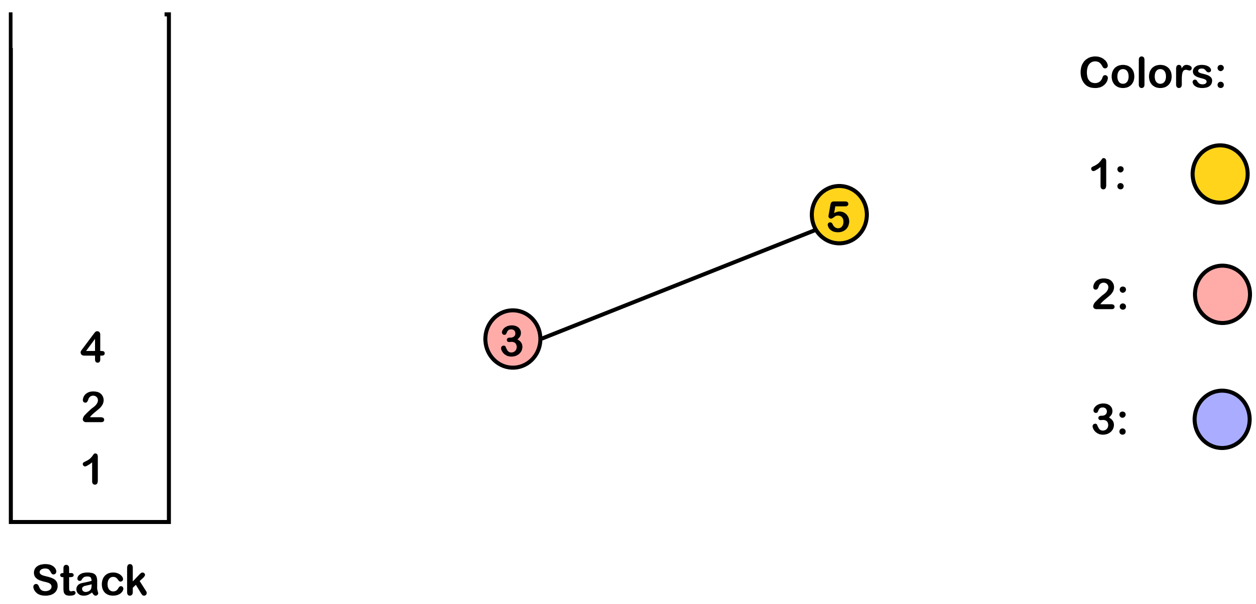
Chaitin's Algorithm in Practice

3 Registers



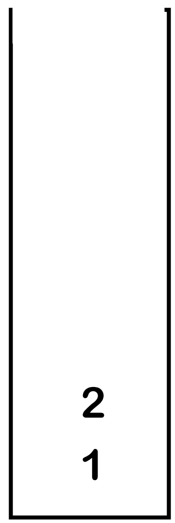
Chaitin's Algorithm in Practice

3 Registers

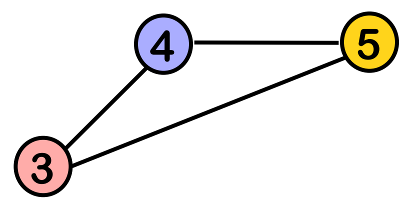


Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

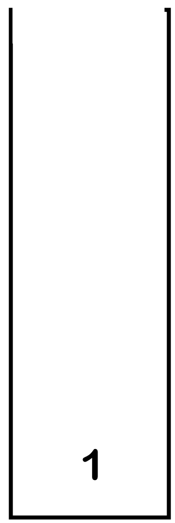
1:

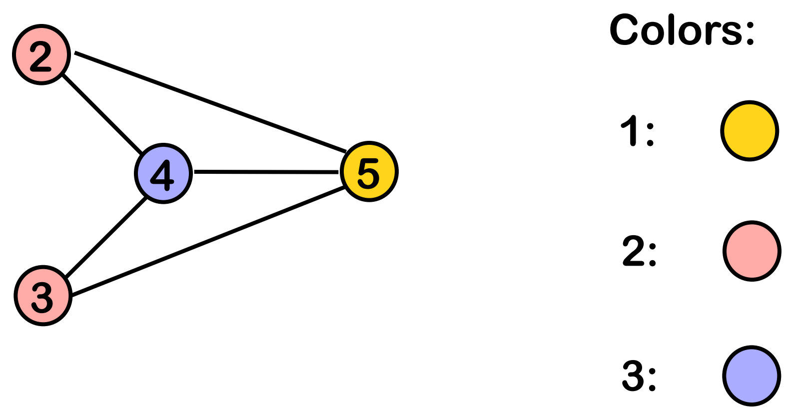
2:

3:

Chaitin's Algorithm in Practice

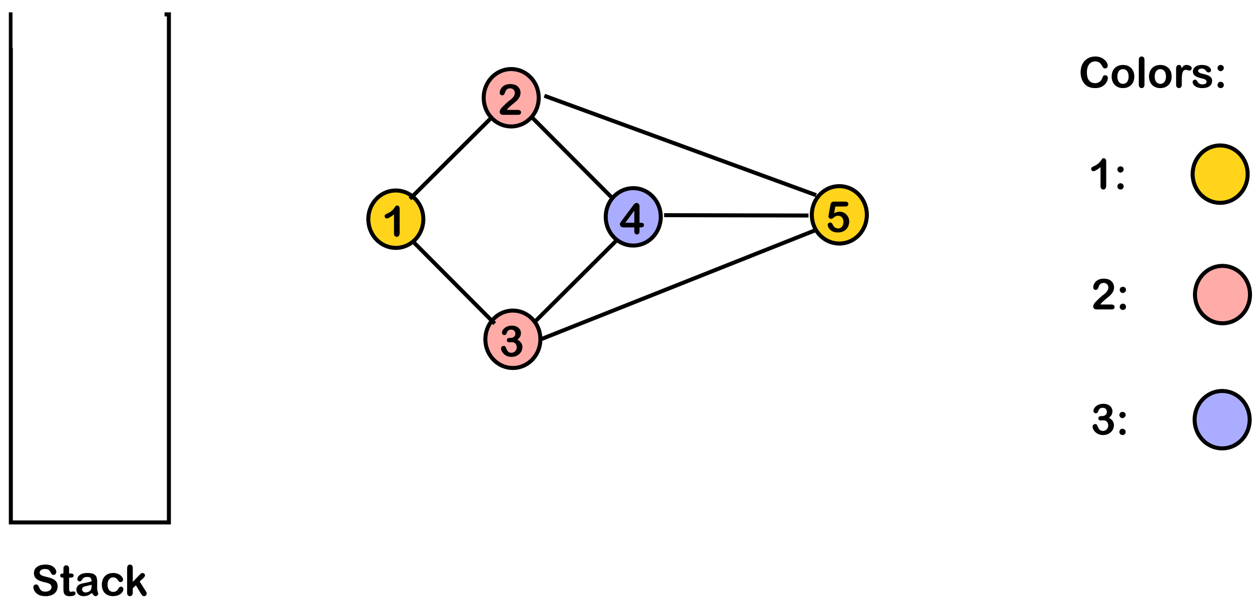
3 Registers





Chaitin's Algorithm in Practice

3 Registers



Picking a Spill Candidate

When $\forall\ n \in G_I,\ n^\circ \geq k$, simplify must pick a spill candidate

Chaitin's heuristic

- Minimize spill cost ÷ current degree
- If LR_x has a negative spill cost, spill it pre-emptively
  - → Cheaper to spill it than to keep it in a register
- If LR_x has an infinite spill cost, it cannot be spilled
  - → No value dies between its definition & its use
  - → No more than *k* definitions since last value died (***safety valve***)

Spill cost is weighted cost of loads & stores needed to spill x

Bernstein *et al.* Suggest repeating simplify, select, & spill
wit
h several different spill choice heuristics & keeping the best