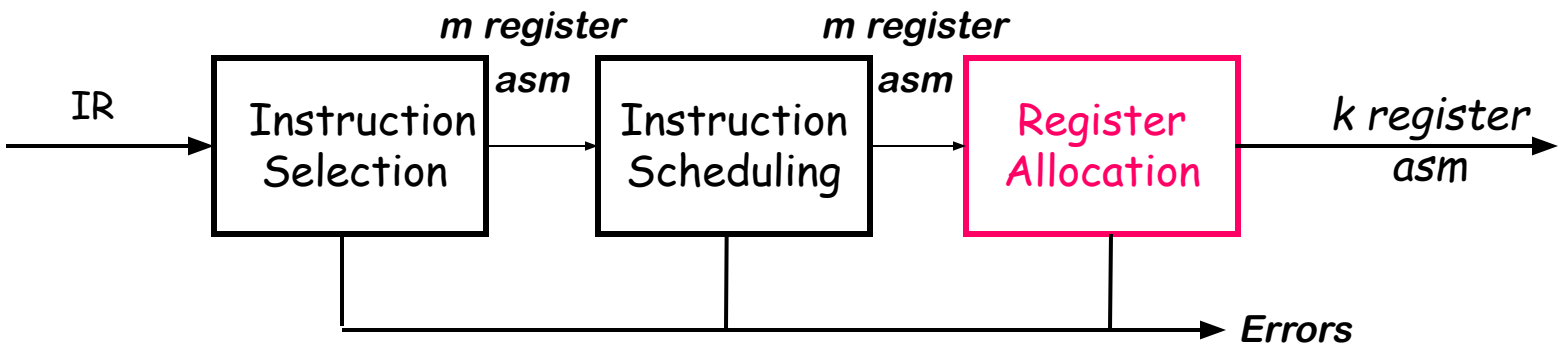
CENG513 Compiler Design and Construction
Register Allocation

Note by Işıl ÖZ:
Our slides are adapted from Cooper and Torczon's slides that are prepared for COMP 412 at Rice.

Register Allocation

Part of the compiler's back end



Critical properties

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

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

Register Allocation

Consider a fragment of assembly code (or ILOC)

```
loadI 2          ⇒ r1    // r1 ← 2
loadAI r0, @b    ⇒ r2    // r2 ← b
mult r1, r2      ⇒ r3    // r3 ← 2 · b
loadAI r0, @a    ⇒ r4    // r4 ← a
sub r4, r3       ⇒ r5    // r5 ← a – (2 · b)
```

From the allocation perspective, these registers are **virtual** or pseudo-registers

The Problem

- At each instruction, decide which *values* to keep in registers
  - Note: each virtual register in the example is a value
- Simple if $|values| \leq |registers|$
- Harder if $|values| > |registers|$
- The compiler must automate this process

Register Allocation

The Task

- At each point in the code, pick the values to keep in registers
- Insert code to move values between registers & memory
  - No transformations (leave that to optimization & scheduling)
- Minimize inserted code — both dynamic & static measures
- Make good use of any *extra* registers

Allocation versus *assignment*

- *Allocation* is deciding which values to keep in registers
- *Assignment* is choosing specific registers for values

The compiler must perform both allocation & assignment

Register Allocation

Optimal register allocation is hard

<u>Local Allocation</u>

- Simplified cases ⇒ O(n)
- Real cases ⇒ NP-Complete

<u>Local Assignment</u>

- Single size, no spilling ⇒ O(n)
- Two sizes ⇒ NP-Complete

<u>Global Allocation</u>

- NP-Complete for 1 register
- NP-Complete for *k* registers

(most sub-problems are NPC, too)

Real compilers face real problems

<u>Global Assignment</u>

- NP-Complete

<u>Recent Results:</u>

Optimal allocation on a procedure in SSA Form can be done in low-order polynomial time.

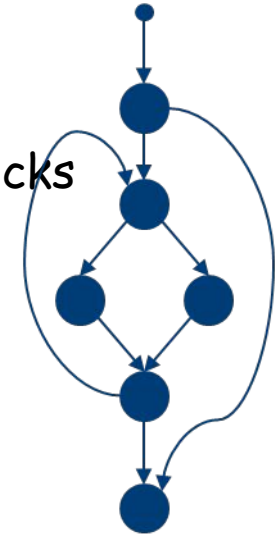
This result does not solve the entire problem, but it does offer insight into the structure of the problem & where the complexity lies.

Local Register Allocation

- What is "local" ? (different from "regional" or "global")
  - A local transformation operates on basic blocks
  - Many optimizations are done on a local scale or scope

- Does local allocation solve the problem?
  - It produces good register use inside a block
  - Inefficiencies can arise at boundaries between blocks

- How many passes can the allocator make?
  - This is an off-line problem
  - As many passes as it takes, within reason
    - → *You can do a fine job in a couple of passes*



Blocks in a Control-flow Graph (CFG)

Top-down Allocator

The idea

- Keep busiest (most heavily used) values in a register
- Reserve registers for use in spills, say *r* registers

Algorithm

- Rank values by number of occurrences
  - may or may not use explicit live ranges
- Allocate first *k* – *r* values to physical registers
- Rewrite code to reflect these choices

Move values with no register into memory
→spill
(add LOADs & STOREs)

Top-down Allocation Algorithm

- Compute a priority for each virtual register
  - number of occurrences of each virtual register
  - priority → a virtual register's count
- Sort the virtual registers into priority order
- Assign registers in priority order
  - first k – feasible virtual registers are assigned physical registers
- Rewrite the code
  - References to virtual registers with assigned physical registers are rewritten with the physical register names
  - Any reference to a virtual register with no physical register is replaced with a reference to a reserved temporary register; a load or store operation is inserted

+ Keeps heavily used virtual registers in physical registers
- Dedicates a physical register to a virtual register for the entire basic block

Top-down Allocation Example

											Counts
loadI 1028	⇒ r_a	//	r_a								r_a=4
load r_a	⇒ r_b	//	r_a	r_b							r_b=3
mult r_a, r_b	⇒ r_c	//	r_a	r_b	r_c						r_c=2
load x	⇒ r_d	//	r_a	r_b	r_c	r_d					r_d=2
sub r_d, r_b	⇒ r_e	//	r_a		r_c		r_e				r_e=2
load z	⇒ r_f	//	r_a		r_c		r_e	r_f			r_f=2
mult r_e, r_f	⇒ r_g	//	r_a		r_c				r_g		r_g=2
sub r_g, r_c	⇒ r_h	//	r_a							r_h	r_h=2
store r_h	→ r_a	//									

											Counts
loadI 1028	⇒ r_a	//	r_a								r_a=4
load r_a	⇒ r_b	//	r_a	r_b							r_b=3
mult r_a, r_b	⇒ r_c	//	r_a	r_b	r_c						r_c=2
load x	⇒ r_d	//	r_a	r_b	r_c	r_d					r_d=2
sub r_d, r_b	⇒ r_e	//	r_a		r_c		r_e				r_e=2
load z	⇒ r_f	//	r_a		r_c		r_e	r_f			r_f=2
mult r_e, r_f	⇒ r_g	//	r_a		r_c				r_g		r_g=2
sub r_g, r_c	⇒ r_h	//	r_a							r_h	r_h=2
store r_h	→ r_a	//									

Must have r_d

r_c < r_a, r_b

Spill r_c

Restore r_c

Top-down Allocation Example

```
loadI 1028                 ra
load  ra                   rb
mult  ra, rb          ⇒    rc
store rc              →    rarp, spillc
load  x                    rd
sub   rd, rb               re
load  z                    rf
mult  re, rf               rg
load  rarp, spillc         rc
sub   rf, rc               rh
store rh              →    ra
```

- ➔ Store after definition
- ➔ Load before use

```
loadI 1028                 r1
load  r1                   r2
mult  r1, r2          ⇒    r3
store r3              →    rarp, spillc
load  x                    r3
sub   r3, r2               r2
load  z                    r3
mult  r2, r3               r2
load  rarp, spillc         r3
sub   r2, r3               r2
store r2              →    r1
```

- ➔ Register assignment

Bottom-up Allocator

The idea:

- Focus on replacement rather than allocation
- Keep values used "soon" in registers

Algorithm:

- Start with empty register set
- Iterate over the operations in the block, load on demand
- When no register is available, free one (spill)

Replacement:

- Spill the value whose next use is farthest in the future
- Prefer clean value to dirty value

Live Ranges in a Basic Block

1	loadI	...	⇒	r_{arp}
2	loadAI	r_{arp}, 0	⇒	r_w
3	loadI	2	⇒	r_2
4	loadAI	r_{arp}, @x	⇒	r_x
5	loadAI	r_{arp}, @y	⇒	r_y
6	loadAI	r_{arp}, @z	⇒	r_z
7	mult	r_w, r_2	⇒	r_w
8	mult	r_w, r_x	⇒	r_w
9	mult	r_w, r_y	⇒	r_w
10	mult	r_w, r_z	⇒	r_w
11	storeAI	r_w	⇒	r_{arp}, 0

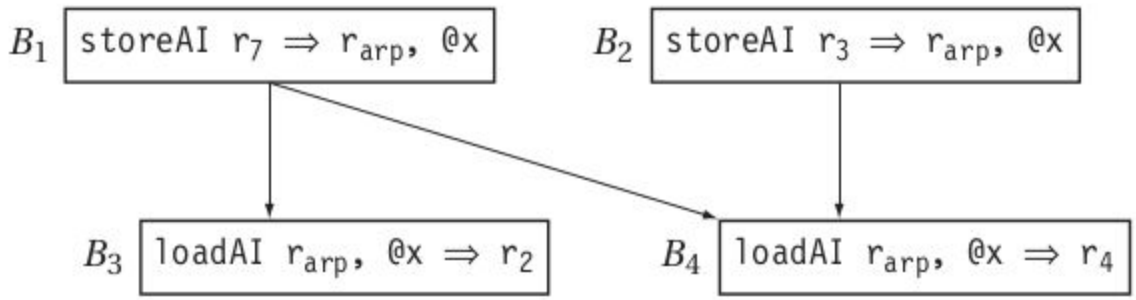
Defines Register	Interval
r_{arp}	[1,11]
r_w	[2,7]
r_w	[7,8]
r_w	[8,9]
r_w	[9,10]
r_w	[10,11]
r_2	[3,7]
r_x	[4,8]
r_y	[5,9]
r_z	[6,10]

Problems with Multiple Blocks

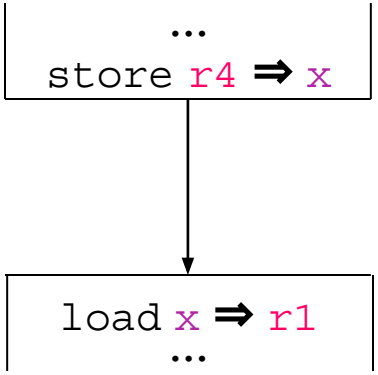
Live variable analysis: LIVEIN and LIVEOUT

Store LIVEOUT (at the end of each block)

Load LIVEIN (at the start of each block)



What Makes Global Register Allocation Hard?



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

Taking a global approach

- Abandon the distinction between local & global
- Make systematic use of registers or memory
- Adopt a general scheme to approximate a good allocation

Graph coloring paradigm

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

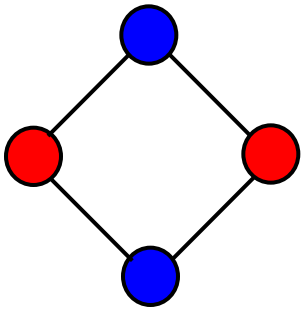
Graph Coloring (A Background Digression)

The problem

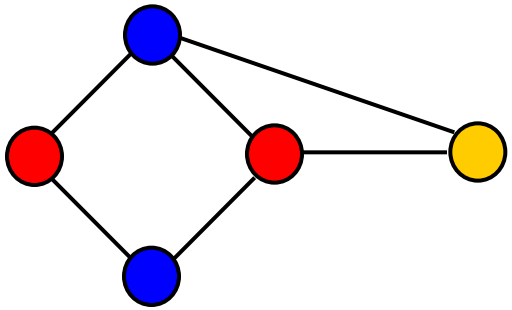
A graph G is said to be *k-colorable* iff the nodes can be labeled with integers 1 … k so that no edge in G connects two nodes with the same label

NP-complete

Examples



2-colorable



3-colorable

Each color can be mapped to a distinct physical register

Global Register Allocation

Discovering live ranges

- Relationships among definitions and uses

Estimating spill costs

- Save the value and restore it from memory when a later operation needs it
- Annotate each block with an estimated execution count

Building an interference graph

- If there is an operation in the program during which two values are live, they cannot reside in the same register, they *interfere* (*conflict*)

Building the Interference Graph

What is an "interference" ? (or conflict)

- Two values *interfere* if there exists an operation where both are simultaneously live
- If x and y interfere, they cannot occupy the same register

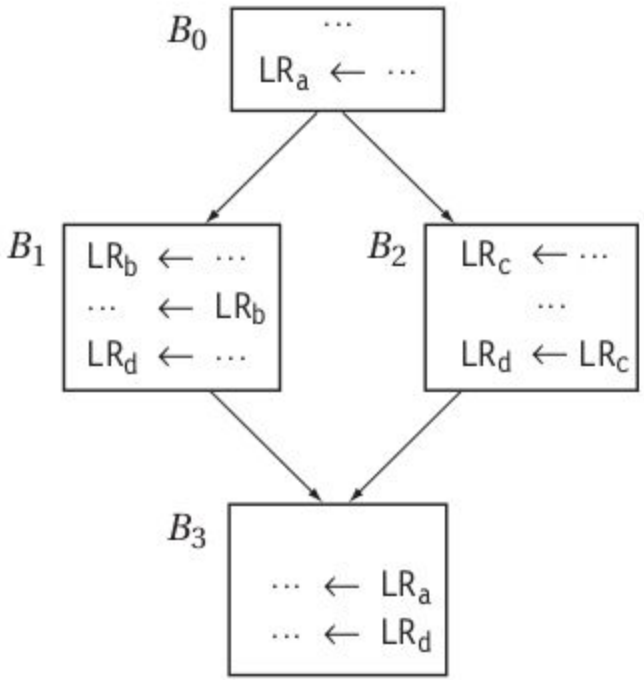
To compute interferences, we must know where values are "live"

The interference graph, $G_I = (N_I, E_I)$

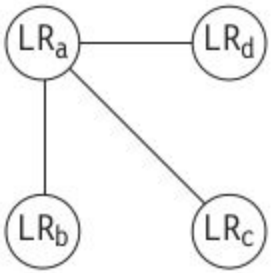
- Nodes in G_I represent values, or live ranges
- Edges in G_I represent individual interferences
  - For $x, y \in N_I$, $\langle x,y \rangle \in E_I$ iff x and y interfere
- A k-coloring of G_I can be mapped into an allocation to k registers

Building the Interference Graph

- LR_a cannot receive the same color as LR_b , LR_c , or LR_d because it interferes with each of them
- The other three live ranges can all share a single color
- 2-colorable, and the code can be rewritten to use just two registers

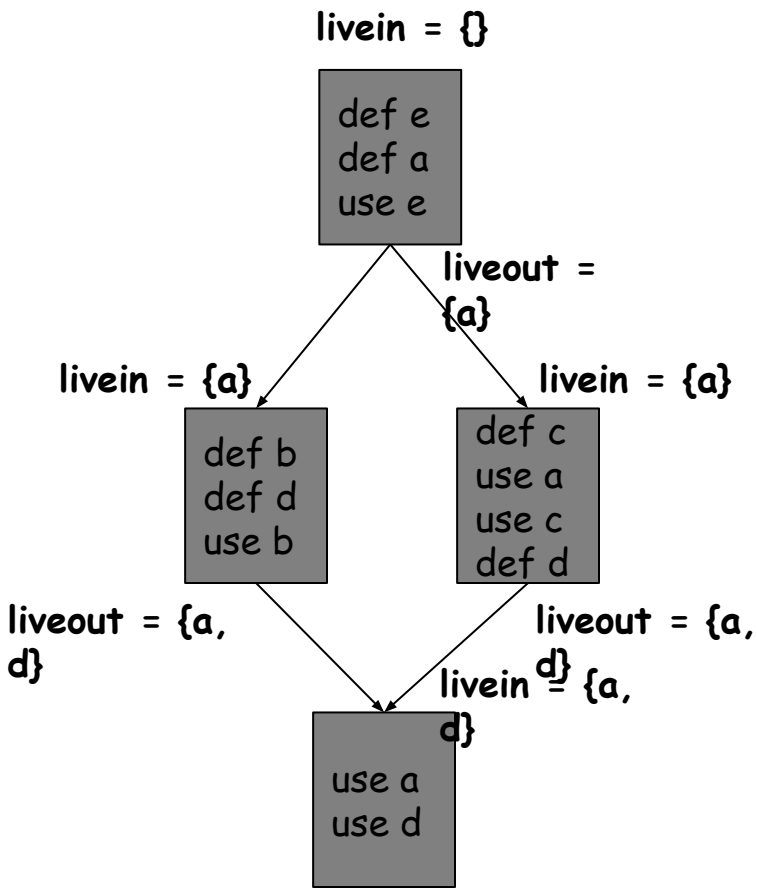


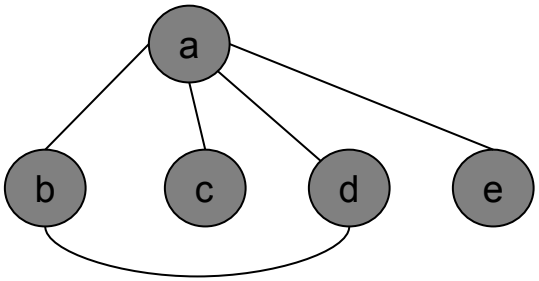
Code Fragment with Live-Range Names



Interference Graph

Building the Interference Graph





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 <u>*must*</u> 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 additional 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

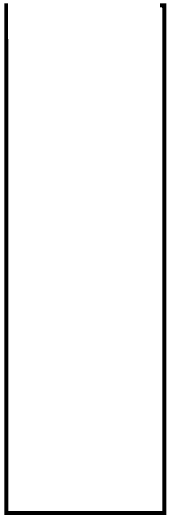
1. While ∃ vertices with < k neighbors in G_I
   - Pick any vertex n such that $n^\circ < k$ and put it on the stack
   - Remove that vertex and all edges incident to it from G_I

Lowers degree of n's neighbors

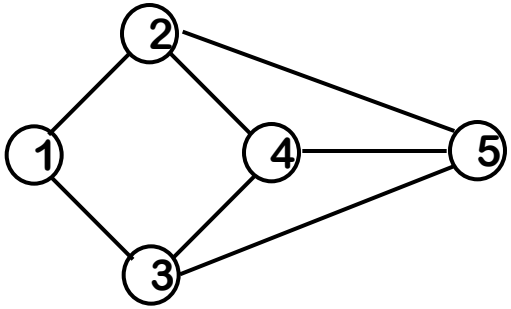
2. If G_I *is non-empty* (all vertices have k or more neighbors) then:
   - Pick a vertex n (using some heuristic) and spill the live range associated with n
   - Remove vertex n from G_I, along with all edges incident to it and put it on the "spill list"
   - If this causes some vertex in G_I to have fewer than k neighbors, then go to step 1; otherwise, repeat step 2
3. If the spill list is not empty, insert spill code, then rebuild the interference graph and try to allocate, again
4. Otherwise, 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



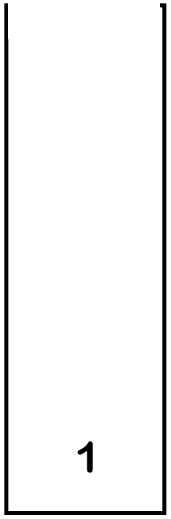
Stack



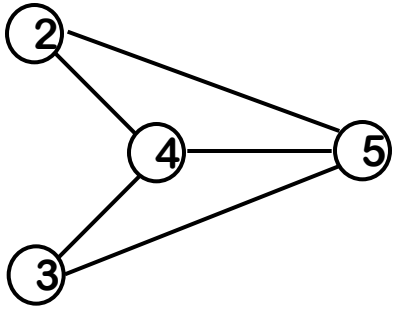
1 is the only node with degree < 3

Chaitin's Algorithm in Practice

3 Registers



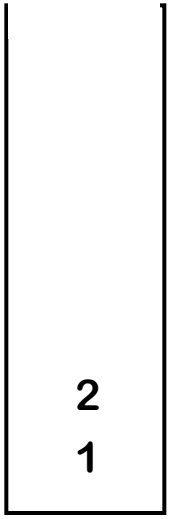
Stack



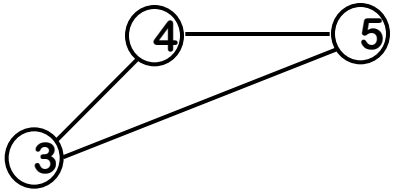
Now, 2 & 3 have degree < 3

Chaitin's Algorithm in Practice

3 Registers



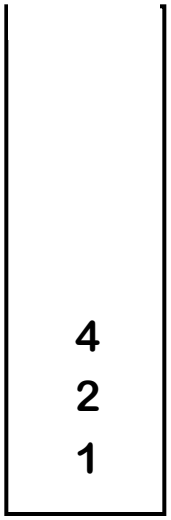
Stack



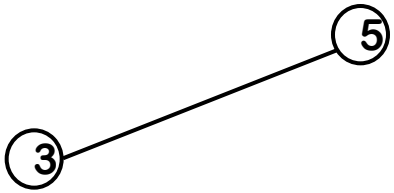
Now all nodes have degree < 3

Chaitin's Algorithm in Practice

3 Registers

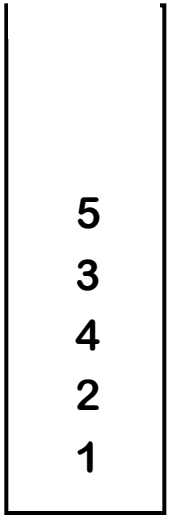


Stack



Chaitin's Algorithm in Practice

3 Registers



Stack

Colors:

1:

2:

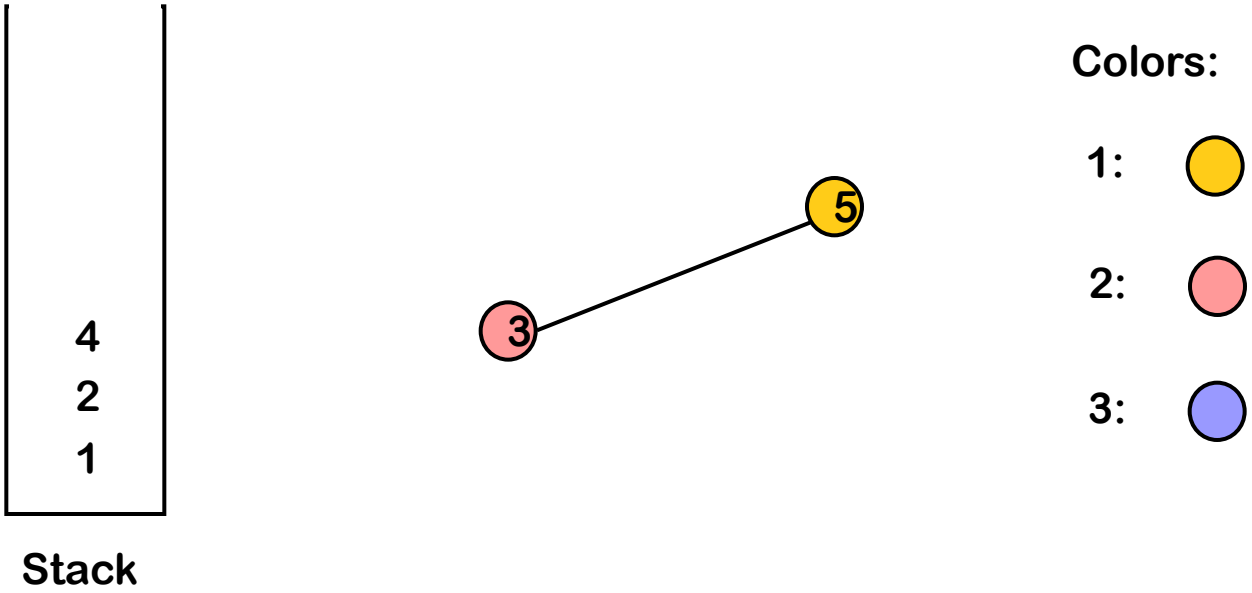
3:

Chaitin's Algorithm in Practice

3 Registers

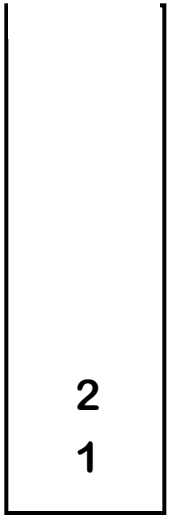
Chaitin's Algorithm in Practice

3 Registers

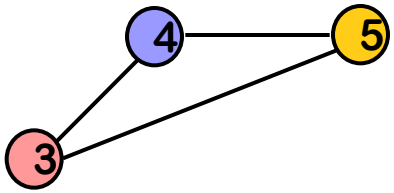


Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

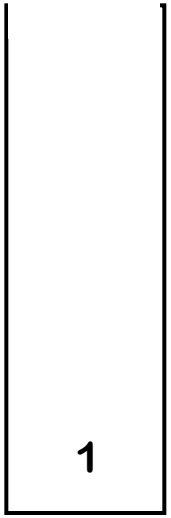
1:

2:

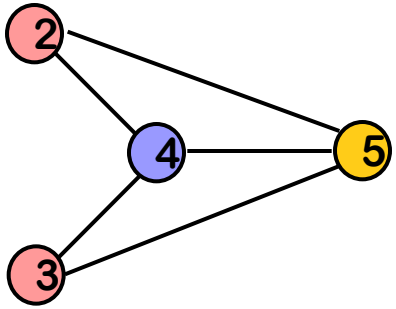
3:

Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

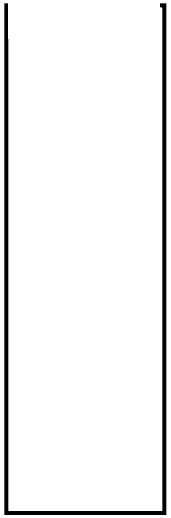
1:

2:

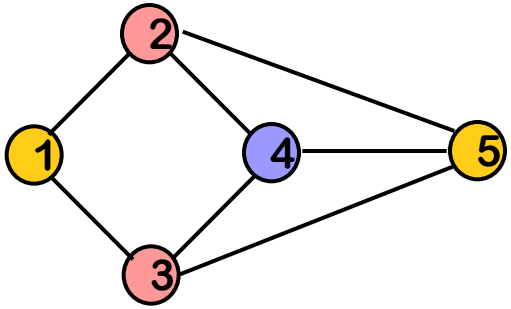
3:

Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

1:

2:

3:

References

Chapter sections from the book:

- 13.1-13.5

Selected videos from compiler course from California State University:

- https://www.youtube.com/watch?v=8x057jVBCVM&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=28
- https://www.youtube.com/watch?v=e_Mn0SI_lI8&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=30
- https://www.youtube.com/watch?v=K6YIXlM1Wj8&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=31