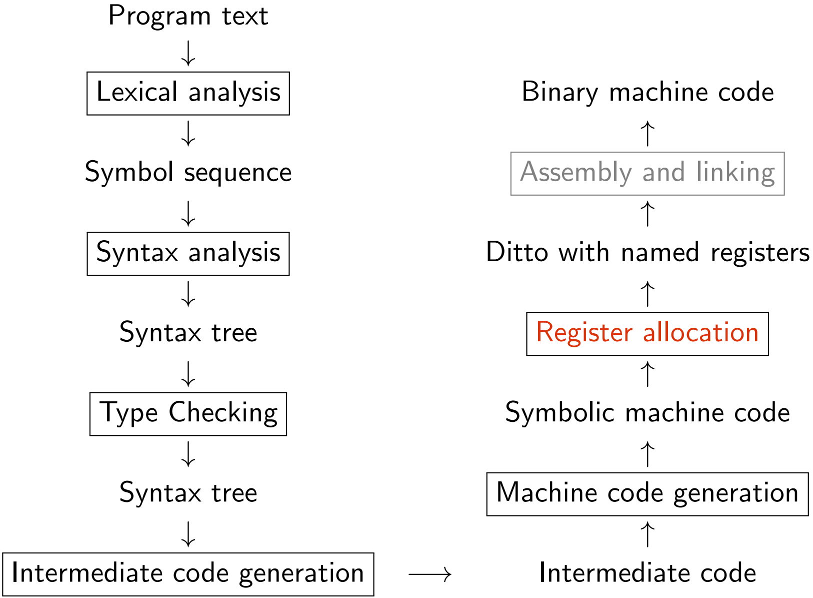
Liveness Analysis and Register Allocation

Cosmin E. Oancea
cosmin.oancea@diku.dk
Modified by Marco Valtorta (UofSC) for CSCE 531 Spring 2021

Department of Computer Science (DIKU)
University of Copenhagen

February 2018 IPS Lecture Slides

Structure of a Compiler



Problem Statement

Processors have a limited number of registers:

X86: 8 (integer) registers,

ARM: 16 (integer) registers,

MIPS: 31 (integer) registers.

In addition, 3 – 4 special-purpose registers (can't hold variables).

Solution:

- Whenever possible, let several variables share the same register,
- If there are still variables that cannot be mapped to a register, store them in memory.

Where to Implement Register Allocation?

Two possibilities: at IL or at machine-language level. Pro/Cons?

Where to Implement Register Allocation?

Two possibilities: at IL or at machine-language level. Pro/Cons?

- IL Level:
  - \+ Can be shared between multiple architectures (parameterized on the number of registers).
  - \- Translation to machine code can introduce/remove intermediate results.
- Machine-Code Level:
  - \+ Accurate, near-optimal mapping.
  - \- Implemented for every architecture, no code reuse.

We show register allocation at IL level. Similar for machine code.

Register-Allocation Scope

- Code Sequence Without Jumps:
  - + Simple.
  - - A variable is saved to memory when jumps occur.
- Procedure/Function Level:
  - + Variables can still be in registers even across jumps.
  - - A bit more complicated.
  - - Variables saved to memory before function calls.
- Module/Program Level:
  - + Sometimes variables can still be hold in registers across function calls (but not always: recursion).
  - - More complicated alg of higher time complexity.

Most compilers implement register allocation at function level.

When Can Two Variables Share a Register?

Intuition: Two vars can share a register if the two variables do not have overlapping periods of use.

Period of Use: From var's assignment to the last use of the assigned value. A variable can have several periods of use (*live ranges*).

Liveness: If a variable's value may be used on the continuation of an execution path passing through program point PP, then the variable is *live* at PP. Otherwise: *dead* at PP.

When Can Two Variables Share a Register?

With the code below, can variables a and c share the same register?

```
a := 1
c := a + 1
a := c + 3
a := a + 2
RETURN a
```

(a) TRUE (b) FALSE

When Can Two Variables Share a Register?

With the code below, can variables a and c share the same register?

```
a := 1
c := a + 1
a := c + 3
a := a + 2
RETURN c
```

(a) TRUE (b) FALSE

Prioritized Rules for Liveness

1) If a variable, VAR, is used, i.e., its value, in an instruction, I, then VAR is *live* at the entry of I.

2) If VAR is assigned a value in instruction I (and 1) does not apply) then VAR is *dead* at the entry of I.

3) If VAR is *live* at the end of instruction I then it is live at the entry of I (unless 2) applies).

4) A VAR is *live* at the end of instruction I $\Leftrightarrow$ VAR is *live* at the entry of any instructions that may be executed immediately after I, i.e., immediate successors of I.

Liveness-Analysis Concepts

We number program instructions from 1 to n.

For each instruction we define the following sets:

- *succ*[i]: The instructions (numbers) that can possibly be executed immediately after instruction (numbered) i.
- *gen*[i]: The set of variables whose values are read by instruct i.
- *kill*[i]: The set of variables that are overwritten by instruction i.
- *in*[i]: The set of variables that are live at the entry of instrct i.
- *out*[i]: The set of variables that are live at the end of instruct i.

In the end, what we need is *out*[i] for all instructions.

Immediate Successors

- $succ[i] = \{i + 1\}$ unless instruction i is a GOTO, an IF-THEN-ELSE, or the last instruction of the program.
- succ[i] = {j}, if instruction i is: `GOTO` l
  and instruction j is: `LABEL` l.
- succ[i] = {j, k}, if instruction i is `IF c THEN` l_1 `ELSE` l_2, instruction j is `LABEL` l_1 , and instruction k is `LABEL` l_2.
- If n denotes the last instruction of the program, and n is not a GOTO or an IF-THEN-ELSE instruction, then $succ[n] = \emptyset$.

Note: Programs always exit by executing a RETURN instruction.

Rules for Constructing *gen* and *kill* Sets

Below k denotes a constant (value), $M[...]$ denotes memory access.

Instruction i	$gen[i]$	$kill[i]$
LABEL l	$\emptyset$	$\emptyset$
$x := y$	$\{y\}$	$\{x\}$
$x := k$	$\emptyset$	$\{x\}$
$x :=$ **unop** y	$\{y\}$	$\{x\}$
$x :=$ **unop** k	$\emptyset$	$\{x\}$
$x := y$ **binop** z	$\{y, z\}$	$\{x\}$
$x := y$ **binop** k	$\{y\}$	$\{x\}$
$x := M[y]$	$\{y\}$	$\{x\}$
$x := M[k]$	$\emptyset$	$\{x\}$
$M[x] :=$ y	$\{x, y\}$	$\emptyset$
$M[k] :=$ y	$\{y\}$	$\emptyset$
GOTO l	$\emptyset$	$\emptyset$
IF x **relop** y THEN l_t ELSE l_f	$\{x, y\}$	$\emptyset$
$x :=$ CALL $f(args)$	$args$	$\{x\}$
RETURN x	$\{x\}$	$\emptyset$

Gen & Kill Sets Multiple-Choice

The *kill* and *gen* sets of instruction `x := a + x` are:

(A) $kill = \emptyset, \quad gen = \{a\}$

(B) $kill = \{x\}, \quad gen = \{a, x\}$

(C) $kill = \{a, x\}, \quad gen = \{x\}$

(D) $kill = \{a\}, \quad gen = \emptyset$

(E) $kill = \{x\}, \quad gen = \{x\}$

Gen & Kill Sets Multiple-Choice

The *kill* and *gen* sets of instruction `M[i] := i + a` are:

(A) $kill = \emptyset, \quad gen = \{a\}$

(B) $kill = \{i\}, \quad gen = \{a, i\}$

(C) $kill = \{a, i\}, \quad gen = \{i\}$

(D) $kill = \{i\}, \quad gen = \{a\}$

(E) $kill = \emptyset, \quad gen = \{i, a\}$

Successors Multiple-Choice Question

With the code below, which of the following statements is TRUE?

```
1.        x := 0
2.        IF x = 0 THEN lab1 ELSE lab2
3.  Label lab1:
4.        x := 3
5.        GOTO lab3
6.  Label lab2:
7.        x := 4
8.  Label lab3:
```

(A) `succ[2] = {3,6,8}` and `succ[4] = {5}`

(B) `succ[2] = {3}` and `succ[5] = {6}`

(C) `succ[2] = {3}` and `succ[1] = {2}`

(D) `succ[2] = {3,6}` and `succ[5] = {8}`

(E) `succ[3] = {2}` and `succ[6] = {2}`

Data-Flow Equation for Liveness Analysis

4) A VAR is *live* at the end of instruction I $\Leftrightarrow$ VAR is *live* at the entry of any instructions that may be executed immediately after I, i.e., immediate successors of I.

The CORRECT Equation for $out[i]$ is:

(A) $out[i] = in[i]$

(B) $out[i] = \bigcup_{j \in succ[i]} in[j]$

(C) $out[i] = gen[i] \cup (\bigcup_{j \in succ[i]} in[j])$

(D) $out[i] = in[i] \setminus (\bigcup_{j \in succ[i]} in[j])$

(E) $out[i] = (gen[i] \cup out[i]) \setminus kill[i]$

Recall that $\setminus$ indicates set difference, i.e. $A \setminus B$ is the set of elements of A that are not in B.

Data-Flow Equation for Liveness Analysis

1) If a variable, VAR, is used, i.e., its value, in an instruction, I, then VAR is *live* at the entry of I.
2) If VAR is assigned a value in instruction I (and 1) does not apply) then VAR is *dead* at the entry of I.
3) If VAR is *live* at the end of instruction I then it is live at the entry of I (unless 2) applies).

The CORRECT Equation for $in[i]$ is:

(A) $in[i] = gen[i] \cup out[i]$

(B) $in[i] = gen[i] \setminus kill[i]$

(C) $in[i] = gen[i] \cup (out[i] \setminus kill[i])$

(D) $in[i] = (gen[i] \setminus kill[i]) \cup out[i]$

(E) $in[i] = (gen[i] \cup out[i]) \setminus kill[i]$

Data-Flow Equations for Liveness Analysis

$$
\begin{aligned}
in[i] &= gen[i] \cup (out[i] \setminus kill[i]) && (1)\\
out[i] &= \bigcup_{j \in succ[i]} in[j] && (2)
\end{aligned}
$$

The (recursive) equations are solved by iterating to a fix point:
$in[i]$ and $out[i]$ are initialized to $\emptyset$, and iterate until no changes occur.

Why does it converge?

For fast(er) convergence: compute $out[i]$ before $in[i]$ and $in[i+1]$ before $out[i]$, respectively (i.e., backward flow analysis).

Imperative-Fibonacci Example

fibo(*n*)
1: $a := 0$
2: $b := 1$
3: $z := 0$
4: `LABEL` *loop*
5: `IF` $n = z$ `THEN` *end* `ELSE` *body*
6: `LABEL` *body*
7: $t := a + b$
8: $a := b$
9: $b := t$
10: $n := n - 1$
11: $z := 0$
12: `GOTO` *loop*
13: `LABEL` *end*
14: `RETURN` *a*

i	$succ[i]$	$gen[i]$	$kill[i]$
1	2		a
2	3		b
3	4		z
4	5		
5	6, 13	n, z	
6	7		
7	8	a, b	t
8	9	b	a
9	10	t	b
10	11	n	n
11	12		z
12	4		
13	14		
14		a	

Fix-Point Iteration for the Fibonacci Example

Use backwards evaluation order: out[14], in[14], ..., out[1], in[1].

	Initial		Iteration 1		Iteration 2		Iteration 3	
i	*out*[*i*]	*in*[*i*]	*out*[*i*]	*in*[*i*]	*out*[*i*]	*in*[*i*]	*out*[*i*]	*in*[*i*]
1			n, a	n	n, a	n	n, a	n
2			n, a, b	n, a	n, a, b	n, a	n, a, b	n, a
3			n, z, a, b	n, a, b	n, z, a, b	n, a, b	n, z, a, b	n, a, b
4			n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b
5			a, b, n	n, z, a, b	a, b, n	n, z, a, b	a, b, n	n, z, a, b
6			a, b, n	a, b, n	a, b, n	a, b, n	a, b, n	a, b, n
7			b, t, n	a, b, n	b, t, n	a, b, n	b, t, n	a, b, n
8			t, n	b, t, n	t, n, a	b, t, n	t, n, a	b, t, n
9			n	t, n	n, a, b	t, n, a	n, a, b	t, n, a
10				n	n, a, b	n, a, b	n, a, b	n, a, b
11					n, z, a, b	n, a, b	n, z, a, b	n, a, b
12					n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b
13			a	a	a	a	a	a
14				a		a		a

Usually less than 5 iterations.

More Multiple Choice Questions

If a formal parameter `p` is NOT LIVE at the entry point of a function, then it means that:

(A) the original value of p is never used, e.g., p is redefined before being used or is never used.

(B) parameter `p` may be used before being written/updated.

(C) parameter `p` is only written inside the function and never read

(D) parameter `p` is only read inside the function and never written

(E) parameter `p` is read-and-written in all instructions in which it appears, i.e., `p:=p+x`

More Multiple Choice Questions

If a variable a, which is NOT a formal argument, is LIVE at the entry point of a function, then it means that:

(A) a is only read inside the function

(B) a may be used without being initialized

(C) a is only written inside the function

(D) a will certainly be used before being initialized

(E) a is read-and-written in all instructions in which it appears, i.e., `a:=a+x`

Interference

Definition: Variable x interferes with variable y, if there is an instruction numbered i such that:

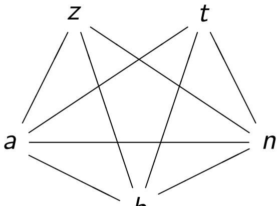
1. $x \in kill[i]$ and
2. $y \in out[i]$ and
3. $x \neq y$ and
4. If instruction i is $x := y$ then x does not interfere with y (but it interferes with any other variable in $out[i]$)

Two variables can share the same register iff they do not interfere with each other!

Interference for the Fibonacci Example

Instruction	Left-hand side	Interferes with	$out(i)$
$1 : a := 0$	a	n	n, a
$2 : b := 1$	b	n, a	n, a, b
$3 : z := 0$	z	n, a, b	n, z, a, b
$7 : t := a + b$	t	b, n	a, b, n
$8 : a := b$	a	t, n	t, b, n
$9 : b := t$	b	n, a	t, n, a
$10 : n := n - 1$	n	a, b	n, a, b
$11 : z := 0$	z	n, a, b	n, z, a, b

Since interference is a symmetric and non-reflexive relation, we can draw interference as a (undirected) graph:



Register Allocation By Graph Coloring

Two variables connected by an edge in the interference graph cannot share a register!

Idea: Associate variables with register numbers such that:

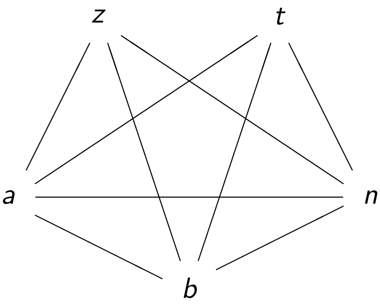
1. Two variables connected by an edge receive different numbers.
2. Numbers represent the (limited number of) hardware registers.

Equivalent to *graph-coloring problem*: color each node with one of n (available) colors, such that any two neighbors are colored differently.

Since graph coloring is NP complete, we use a heuristic method that gives good results in most cases.

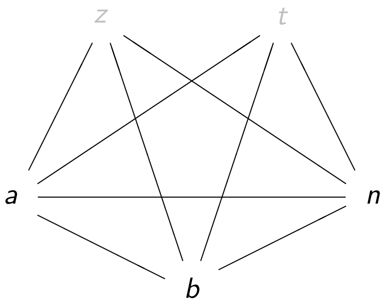
Idea: a node with less-than-n neighbors can always be colored. Eliminate such nodes from the graph and solve recursively!

Coloring The Graph With Four Colors



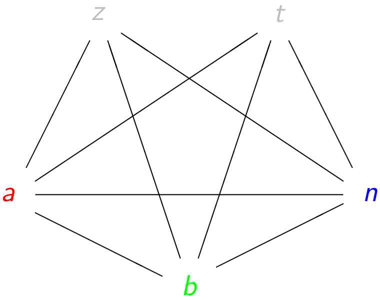
z and t have only three neighbors so they can wait.

Coloring The Graph With Four Colors



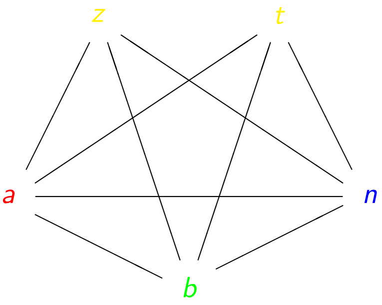
The remaining three nodes can now be given different colors!

Coloring The Graph With Four Colors



z and t can now be given a different color!

Coloring The Graph With Four Colors



But what if we only have three colors (registers) available?

Improved Algorithm

Initialization: Start with an empty stack.

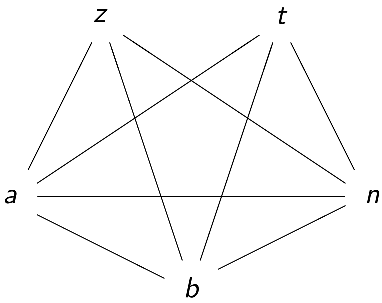
Simplify: 1) If there is a node with less than *n* edges (neighbors): (i) place it on the stack together with the list of edges, and (ii) remove it and its edges from the graph.

2. If there is no node with less than *n* neighbors, pick any node and do as above.
3. Continue until the graph is empty. If so go to *select*.

Select:
1. Take a node and its neighbor list from the stack.
2. If possible, color it differently than its neighbor's.
3. If not possible, select the node for *spilling* (fails).
4. Repeat until stack is empty.

The quality of the result depends on (i) how to chose a node in *simplify*, and (ii) how to chose a color in *select*.

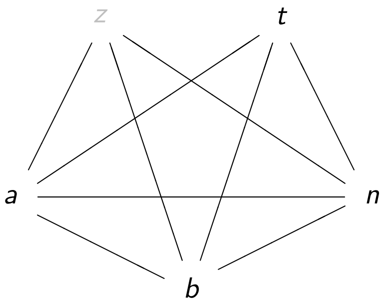
Example: Coloring the Graph with Three Colors



No node has < 3 neighbors, hence choose arbitrarily, say z.

Node	Neighbours	Color
z	a, b, n	

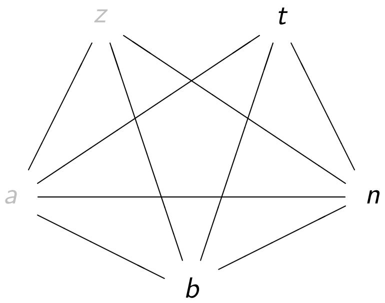
Example: Coloring the Graph with Three Colors



There are still no nodes with < 3 neighbors, hence we chose a.

Node	Neighbours	Color
a	b, n, t	
z	a, b, n	

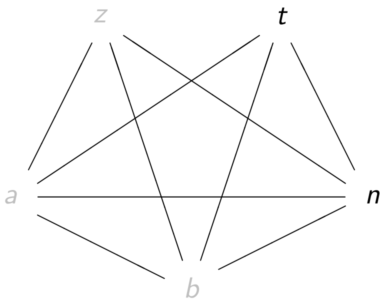
Example: Coloring the Graph with Three Colors



b has two neighbors, so we choose it.

Node	Neighbours	Color
b	*t*, *n*	
a	*b*, *n*, *t*	
z	*a*, *b*, *n*	

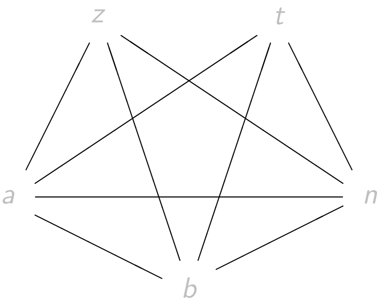
Example: Coloring the Graph with Three Colors



Finally, choose t and n.

Node	Neighbours	Color
n		
t	n	
b	t, n	
a	b, n, t	
z	a, b, n	

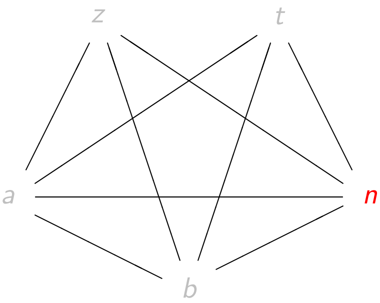
Example: Coloring the Graph with Three Colors



n has no neighbors so we can choose 1.

Node	Neighbours	Color
n		1
t	n	
b	t, n	
a	b, n, t	
z	a, b, n	

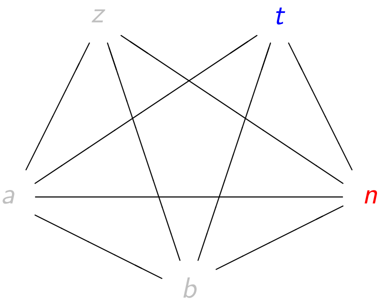
Example: Coloring the Graph with Three Colors



t only has *n* as neighbor, so we can color it with 2.

Node	Neighbours	Color
n		1
t	*n*	2
b	*t*, *n*	
a	*b*, *n*, *t*	
z	*a*, *b*, *n*	

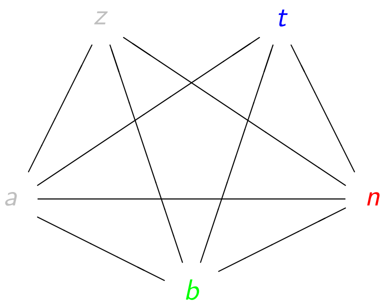
Example: Coloring the Graph with Three Colors



b has *t* and *n* as neighbors, hence we can color it with 3.

Node	Neighbours	Color
n		1
t	*n*	2
b	*t*, *n*	3
a	*b*, *n*, *t*	
z	*a*, *b*, *n*	

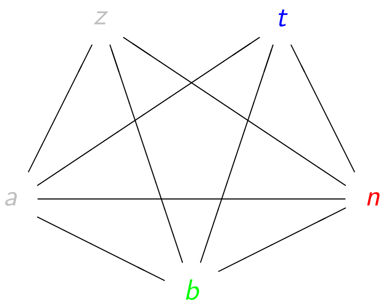
Example: Coloring the Graph with Three Colors



a has three differently-colored neighbors, so it is marked as *spill*.

Node	Neighbours	Color
n		1
t	*n*	2
b	*t*, *n*	3
a	*b*, *n*, *t*	*spill*
z	*a*, *b*, *n*	

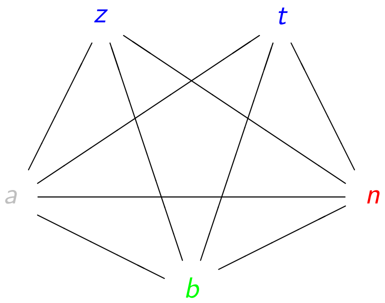
Example: Coloring the Graph with Three Colors



z has colors 1 and 3 as neighbors, hence we can color it with 2.

Node	Neighbours	Color
n		1
t	n	2
b	t, n	3
a	b, n, t	*spill*
z	a, b, n	2

Example: Coloring the Graph with Three Colors



We are now finished, but we need to *spill* a.

Node	Neighbours	Color
n		1
t	n	2
b	t, n	3
a	b, n, t	*spill*
z	a, b, n	2

Spilling

Spilling means that some variables will reside in memory (except for brief periods). For each spilled variable:

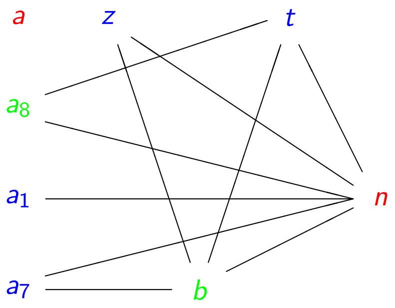
1) Select a memory address $addr_x$, where the value of x will reside.
2) If instruction i uses x, then rename it locally to x_i.
3) Before an instruction i, which reads x_i, insert $x_i := M[addr_x]$.
4) After an instruction i, which updates x_i, insert $M[addr_x] := x_i$.
5) If x is alive at the beginning of the function/program, insert $M[addr_x] := x$ before the first instruction of the function.
6) If x is live at the end of the program/function, insert $x := M[addr_x]$ after the last instruction of the function.

Finally, perform liveness analysis and register allocation again.

Spilling Example

1: $a_1 := 0$
   $M[address_a] := a_1$
2: $b := 1$
3: $z := 0$
4: LABEL *loop*
5: IF $n = z$ THEN *end* ELSE *body*
6: LABEL *body*
   $a_7 := M[address_a]$
7: $t := a_7 + b$
8: $a_8 := b$
   $M[address_a] := a_8$
9: $b := t$
10: $n := n - 1$
11: $z := 0$
12: GOTO *loop*
13: LABEL *end*
   $a := M[address_a]$

After Spilling, Coloring Succeeds!



Heuristics

For **Simplify**: when choosing a node with $\geq n$ neighbors:

- Choose the node with fewest neighbors, which is more likely to be colorable, or
- Choose a node with many neighbors, each of them having close to n neighbors, i.e., spilling this node would allow the coloring of its neighbors.

For **Select**: when choosing a color:

- Choose colors that have already been used.
- If instructions such as $x := y$ exist, color x and y with the same color, i.e., eliminate this instruction.