Reading Assignment

- Read George and Appel's paper, "Iterated Register Coalescing." (Linked from Class Web page)
- Read Larus and Hilfinger's paper, "Register Allocation in the SPUR Lisp Compiler."

Liveness Analysis

Just because a definition reaches a Basic Block, b, *does not* mean it must be allocated to a register at b.

We also require that the definition be *Live* at b. If the definition is dead, then it will no longer be used, and register allocation is unnecessary.

For a Basic Block b and Variable V:

LiveIn(b) = true if V is Live (will be used before it is redefined) at the beginning of b.

LiveOut(b) = true if V is Live (will be used before it is redefined) at the end of b.

LiveIn and LiveOut are computed, using the following rules:

1. If Basic Block b has no successors then
   LiveOut(b) = false
2. For all Other Basic Blocks

$$\text{LiveOut}(b) = \bigvee_{s \in \text{Succ}(b)} \text{LiveIn}(s)$$

3. LiveIn(b) =
   If V is used before it is defined in Basic Block b
   Then true
   Elsif V is defined before it is used in Basic Block b
   Then false
   Else LiveOut(b)

Merging Live Ranges

It is possible that each Basic Block that contains a definition of v creates a *distinct* Live Range of V.

∀ Basic Blocks, b, that contain a definition of V:

$$\text{Range}(b) = \{b\} \cup \{k \mid b \in \text{DefsIn}(k) \ \&\ \text{LiveIn}(k)\}$$

This rule states that the Live Range of a definition to V in Basic Block b is b plus all other Basic Blocks that the definition of V reaches and in which V is live.

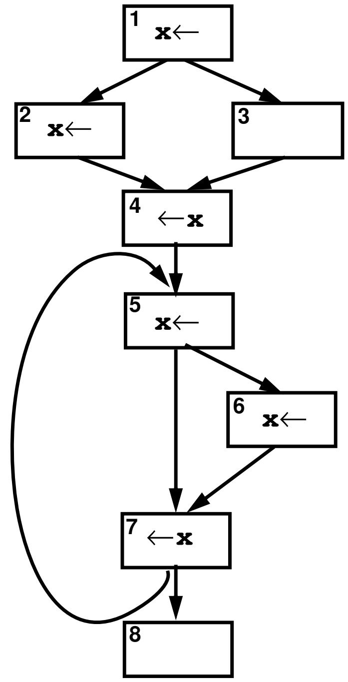
If two Live Ranges overlap (have one of more Basic Blocks in common), they *must* share the same register too. (Why?)

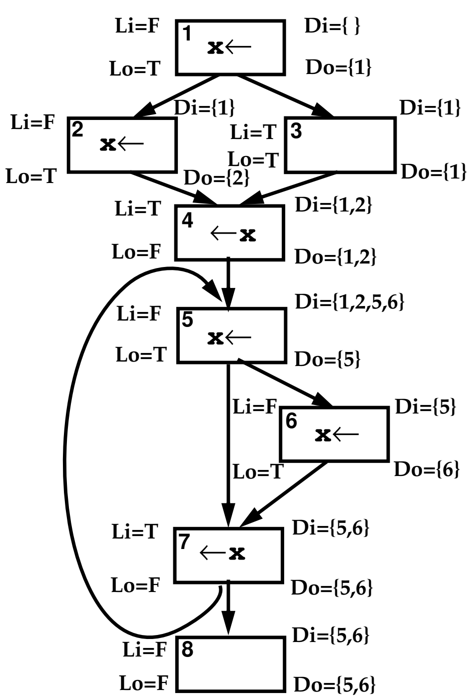
Therefore,

If $\text{Range}(b_1) \cap \text{Range}(b_2) \neq \phi$

Then replace
$\text{Range}(b_1)$ and $\text{Range}(b_2)$
with $\text{Range}(b_1) \cup \text{Range}(b_2)$

Example





The Live Ranges we Compute are

Range(1) = {1} U {3,4} = {1,3,4}

Range(2) = {2} U {4} = {2,4}

Range(5) = {5} U {7} = {5,7}

Range(6) = {6} U {7} = {6,7}

Ranges 1 and 2 overlap, so

Range(1) = Range(2) = {1,2,3,4}

Ranges 5 and 6 overlap, so

Range(5) = Range(6) = {5,6,7}

Interference Graph

An *Interference Graph* represents interferences between Live Ranges.

Two Live Ranges *interfere* if they share one or more Basic Blocks in common.

Live Ranges that interfere *must* be allocated different registers.

In an Interference Graph:

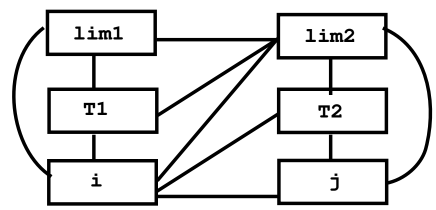
- Nodes are Live Ranges
- An undirected arc connects two Live Ranges if and only if they interfere

Example

```
int p(int lim1, int lim2) {
  for (i=0; i<lim1 && A[i]>0;i++){}
  for (j=0; j<lim2 && B[j]>0;j++){}
  return i+j;
}
```

We optimize array accesses by placing `&A[0]` and `&B[0]` in temporaries:

```
int p(int lim1, int lim2) {
  int *T1 = &A[0];
  for (i=0; i<lim1 && *(T1+i)>0;i++){}
  int *T2 = &B[0];
  for (j=0; j<lim2 && *(T2+j)>0;j++){}
  return i+j;
}
```



Register Allocation via Graph Coloring

We model global register allocation as a Coloring Problem on the Interference Graph

We wish to use the fewest possible colors (registers) subject to the rule that two connected nodes can't share the same color.

Optimal Graph Coloring is NP-Complete

Reference:

"Computers and Intractability," M. Garey and D. Johnson, W.H. Freeman, 1979.

We'll use a Heuristic Algorithm originally suggested by Chaitin et. al. and improved by Briggs et. al.

References:

"Register Allocation Via Coloring," G. Chaitin et. al., Computer Languages, 1981.

"Improvement to Graph Coloring Register Allocation," P. Briggs et. al., PLDI, 1989.

Coloring Heuristic

To R-Color a Graph (where R is the number of registers available)

1. While any node, n, has < R neighbors:
   Remove n from the Graph.
   Push n onto a Stack.
2. If the remaining Graph is non-empty:
   Compute the Cost of each node.
   The Cost of a Node (a Live Range) is the number of extra instructions needed if the Node isn't assigned a register, scaled by 10^{loop_depth}.
   Let NB(n) =
   Number of Neighbors of n.
   Remove that node n that has the smallest Cost(n)/NB(n) value.
   Push n onto a Stack.
   Return to Step 1.

3. While Stack is non-empty:
   Pop n from the Stack.
   If n's neighbors are assigned fewer than R colors
   Then assign n any unassigned color
   Else leave n uncolored.