Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue 10:00-11:50 @UTS 203, Labs: Tue 12:00-13:50 & Wed 12:00-13:50 @FAB 88-10

Lab 7: Register Allocation via Graph Coloring

(Adapted from Prof. Andrew Tolmach's earlier version)

Recall that the goal of register allocation is to assign a *unique* physical machine register to each IR variable and temporary. With liveness information available, we can perform precise register assignment.

As described in lecture, one way to do this is to translate the assignment problem into a *graph coloring* problem. This has the following steps (for each procedure):

- Construct a *register interference graph*. This graph has a node for each IR variable used in the procedure, and an edge between every two nodes that are simultaneously live (more precisely, that appear together in some live-out set).
- We try to assign a physical register to each node, such that no two nodes connected by an edge are assigned the same register.
- If we succeed, we get a register assignment that avoids any need to spill registers to memory. If we fail, spilling will be necessary.

If we think of the different physical registers as colors, then the assignment process is like coloring the nodes of a graph such that no two connected nodes have the same color. This is closely related to the practical problem of coloring the countries on a map such that no two countries with a common border share the same color. (There is a famous mathematical result that says four colors suffice to color any map. Unfortunately, this doesn't imply that four registers are sufficient for any procedure. Why not?)

Register Interference Graphs

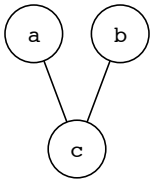
As examples, consider the following three programs (`example1.ir`, `example2.ir`, and `example3.ir`):

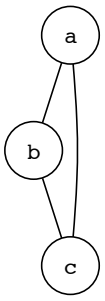
```
_f ()
(a,b,c)
{
 1.    a = 0
 2. L:
 3.    b = a + 1
 4.    c = c + b
 5.    a = b * 2
 6.    if a < 1000 goto L
 7.    return c
}
```

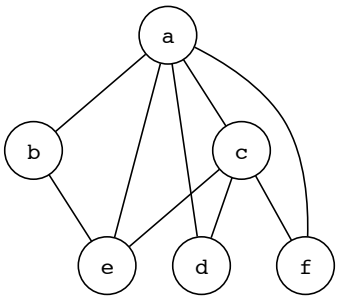
```
_f ()
(a,b,c)
{
 1.    a = 2
 2. L:
 3.    c = 1
 4.    b = a + 1
 5.    a = c * b
 6.    if a < b goto L
 7.    return b
}
```

```
_main (a)
(b,c,d,e,f)
{
 1.     e = 42
 2.     b = a * 4
 3.     goto L3
 4. L0:
 5.     c = a + b
 6.     if c < 100 goto L1
 7.     d = 10 + c
 8.     e = d + d
 9.     goto L2
10. L1:
11.     f = c / 10
12.     e = f - 40
13. L2:
14.     b = e - c
15. L3:
16.     if e > 0 goto L0
17.     return e
}
```

Here are their interference graphs:







Example 1 It is trivial to assign registers for this program: only two are needed, with `c` living in one register, say `%r0` and both `a` and `b` living in the other, say `%r1`. Substituting these assignments in for the variables in the original code gives:

```
1.    %r1 = 0
2. L:
3.    %r1 = %r1 + 1
4.    %r0 = %r0 + %r1
5.    %r1 = %r1 * 2
6.    if %r1 < 1000 goto L
7.    return %r0
```

Example 2 This proram obviously requires three registers. If we assign, say `a` to `%r0`, `b` to `%r1` and `c` to `%r2`, the resulting code after substitution is:

```
1.    %r0 = 2
2. L:
3.    %r2 = 1
4.    %r1 = %r0 + 1
5.    %r0 = %r2 * %r1
6.    if %r0 < %r1 goto L
7.    return %r1
```

Note that this program requires three registers even though no more than two variables are ever live-out at any point! This may seem paradoxical at first. The reason is that we are insisting that each variable be assigned a *unique* register for the entirety of the procedure. It is a worthwhile exercise to try to use just two registers for this example, e.g., using the greedy algorithm described in lecture. Where do things go wrong?

Also note that the inteference graph for this program contains 3-clique. A *clique* is a subgraph in which every node is connected to every other node. It should be easy to see that a graph containing k-clique always requires at least k distinct colors.

Example 3 It is a little less obvious how many registers are needed for this program. At least three must be needed (why?) but will they be enough? The answer is yes: if we assign `a` to `%r0`, then `b` and `c` can be assigned to `%r1`, and `d`,`e`, and `f` to `%r2`.

Notice that node `a` is connected to each of the other five nodes; we say that the *degree*(`a`) = 5. But this doesn't make it particularly hard to color the graph.

Exercise Write an example IR program whose interference graph has a node of degree 5, but can be colored with just two colors. Design your program so that it can be easily extended to induce a graph node with arbitrarily high degree, still requiring just two colors.

(Solution: `example4.ir`)

Exercise Write an example IR program whose interference graph requires five colors. Design your program so that it can be easily extended to induce a graph requiring an arbitrarily large number of colors.

(Solution: `example5.ir`)

Finding a Coloring

So far we have been finding colorings by inspection, but again, of course, we really want to write an algorithm that a compiler can implement to perform this task. If our machine architecture gives us k registers to play with, we want to find a k-coloring, i.e., one that uses no more than k colors. More generally, we might want to find a coloring that uses the minimum number of colors.

Unfortunately, even determining whether a graph has a k-coloring or not is an NP-complete problem, meaning that, as far as anyone currently knows, it requires $O(2^n)$ time, where n is the number of nodes in the graph. Exponential-time algorithms are usually not considered practical inside compilers!

However, there turn out to be simple and fast *heuristic* methods that will usually find a k-coloring if one exists. These methods aren't guaranteed to work, but they usually do a good job on graphs produced from real programs. One simple heuristic procedure to obtain a k-coloring works like this:

1. If the graph is empty, it is trivial to color (with zero colors!)
2. Otherwise, choose a node n from the graph such that $\mathit{degree}(n) < k$. If there are no such nodes, the procedure fails.
3. Remove node n and all its incident edges from the graph.
4. Recursively invoke the coloring procedure on the remaining graph. If it succeeds, there must be a k-coloring for that graph.
5. Now add back node n and its edges, and assign it a color. We are guaranteed to be able to do this, because it has fewer than k neighbors, so even if they all have different colors, there will still be at least one color left we can use.

In practice, this procedure is usually implemented using an explicit stack data structure and two loops, rather than via a recursive function. When a node is removed from the graph in step 2, it is pushed onto the stack, together with a list of its neighbors. Steps 1–4 become a loop that terminates when the graph is empty. Step 5 becomes a loop that iterates, popping each node from the stack in turn and assigning it color that doesn't conflict with those of its neighbors (which are guaranteed to have been colored already).

Also, rather than failing at step 2 if we can't find a node with $\mathit{degree}(n) < k$, we can just pick (any) node of lowest degree and push it on the stack. When it is popped off, we might get lucky and be able to color it even though it has k or more neighbors, as long as they don't all have distinct colors.

Exercise Walk through how this procedure can be used to find a 3-coloring for the graph of Example 3.