10 Liveness Analysis

live: of continuing or current interest

Webster's Dictionary

The front end of the compiler translates programs into an intermediate language with an unbounded number of temporaries. This program must run on a machine with a bounded number of registers. Two temporaries a and b can fit into the same register, if a and b are never "in use" at the same time. Thus, many temporaries can fit in few registers; if they don't all fit, the excess temporaries can be kept in memory.

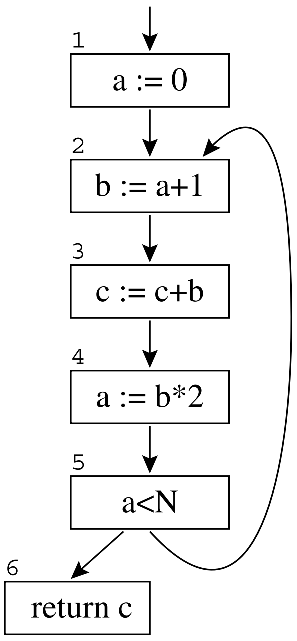
Therefore, the compiler needs to analyze the intermediate-representation program to determine which temporaries are in use at the same time. We say a variable is *live* if it holds a value that may be needed in the future, so this analysis is called *liveness* analysis.

To perform analyses on a program, it is often useful to make a *control-flow graph*. Each statement in the program is a node in the flow graph; if statement x can be followed by statement y, there is an edge from x to y. Graph 10.1 shows the flow graph for a simple loop.

Let us consider the liveness of each variable (Figure 10.2). A variable is live if its current value will be used in the future, so we analyze liveness by working from the future to the past. Variable b is used in statement 4, so b is live on the $3 \rightarrow 4$ edge. Since statement 3 does not assign into b, then b is also live on the $2 \rightarrow 3$ edge. Statement 2 assigns into b. That means that the contents of b on the $1 \rightarrow 2$ edge are not needed by anyone; b is dead on this edge. So the *live range* of b is $\{2 \rightarrow 3,\ 3 \rightarrow 4\}$.

The variable a is an interesting case. It's live from $1 \rightarrow 2$, and again from $4 \rightarrow 5 \rightarrow 2$, but not from $2 \rightarrow 3 \rightarrow 4$. Although a has a perfectly

```
      $a \leftarrow 0$
$L_1$ : $b \leftarrow a + 1$
      $c \leftarrow c + b$
      $a \leftarrow b * 2$
      if $a < N$ goto $L_1$
      return $c$
```



GRAPH 10.1. Control-flow graph of a program.

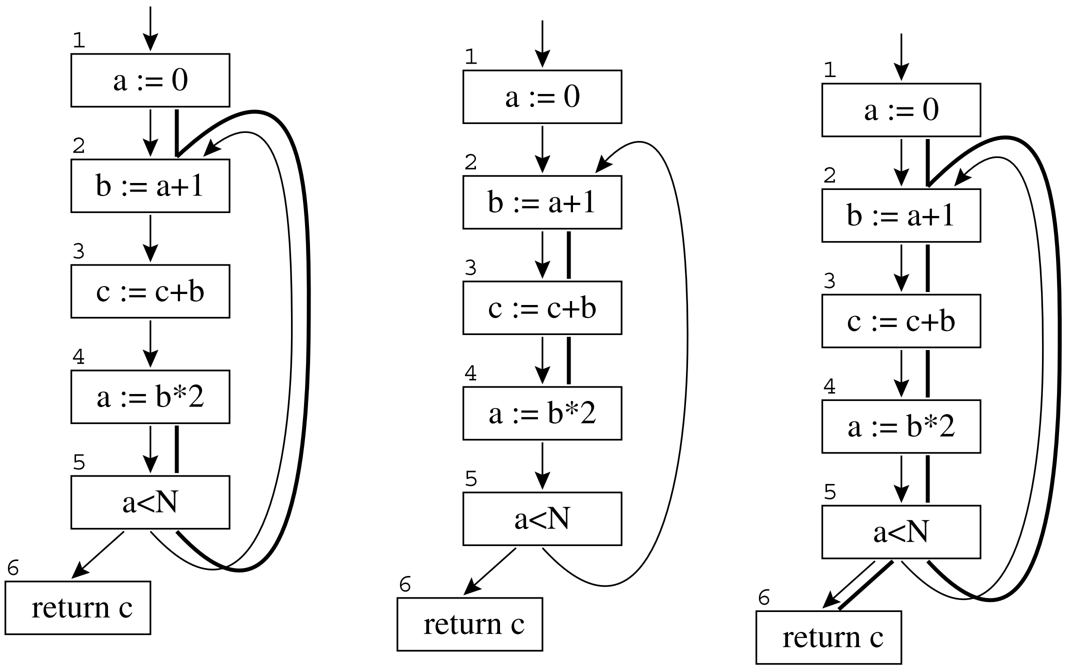


FIGURE 10.2. Liveness of variables a, b, c.

well-defined value at node 3, that value will not be needed again before a is assigned a new value.

The variable c is live on entry to this program. Perhaps it is a formal parameter. If it is a local variable, then liveness analysis has detected an uninitialized variable; the compiler could print a warning message for the programmer.

Once all the live ranges are computed, we can see that only two registers are needed to hold a, b, and c, since a and b are never live at the same time. Register 1 can hold both a and b, and register 2 can hold c.

10.1 SOLUTION OF DATAFLOW EQUATIONS

Liveness of variables "flows" around the edges of the control-flow graph; determining the live range of each variable is an example of a *dataflow* problem. Chapter 17 will discuss several other kinds of dataflow problems.

Flow-graph terminology. A flow-graph node has *out-edges* that lead to *successor* nodes, and *in-edges* that come from *predecessor* nodes. The set $pred[n]$ is all the predecessors of node n, and $succ[n]$ is the set of successors.

In Graph 10.1 the out-edges of node 5 are $5 \rightarrow 6$ and $5 \rightarrow 2$, and $succ[5] = \{2, 6\}$. The in-edges of 2 are $5 \rightarrow 2$ and $1 \rightarrow 2$, and $pred[2] = \{1, 5\}$.

Uses and defs. An assignment to a variable or temporary *defines* that variable. An occurrence of a variable on the right-hand side of an assignment (or in other expressions) *uses* the variable. We can speak of the *def* of a variable as the set of graph nodes that define it; or the *def* of a graph node as the set of variables that it defines; and similarly for the *use* of a variable or graph node. In Graph 10.1, $def(3) = \{c\}$, $use(3) = \{b, c\}$.

Liveness. A variable is *live* on an edge if there is a directed path from that edge to a *use* of the variable that does not go through any *def*. A variable is *live-in* at a node if it is live on any of the in-edges of that node; it is *live-out* at a node if it is live on any of the out-edges of the node.

CALCULATION OF LIVENESS

Liveness information (*live-in* and *live-out*) can be calculated from *use* and *def* as follows:

$$in[n] = use[n] \cup (out[n] - def[n])$$
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

EQUATIONS 10.3. Dataflow equations for liveness analysis.

```
**for each** $n$
    $in[n] \leftarrow \{\};\ out[n] \leftarrow \{\}$
**repeat**
    **for each** $n$
        $in'[n] \leftarrow in[n];\ out'[n] \leftarrow out[n]$
        $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
        $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
**until** $in'[n] = in[n]$ and $out'[n] = out[n]$ for all $n$
```

ALGORITHM 10.4. Computation of liveness by iteration.

1. If a variable is in *use*[n], then it is *live-in* at node n. That is, if a statement uses a variable, the variable is live on entry to that statement.
2. If a variable is *live-in* at a node n, then it is *live-out* at all nodes m in *pred*[n].
3. If a variable is *live-out* at node n, and not in *def*[n], then the variable is also *live-in* at n. That is, if someone needs the value of a at the end of statement n, and n does not provide that value, then a's value is needed even on entry to n.

These three statements can be written as Equations 10.3 on sets of variables. The live-in sets are an array *in*[n] indexed by node, and the live-out sets are an array *out*[n]. That is, *in*[n] is all the variables in *use*[n], plus all the variables in *out*[n] and not in *def*[n]. And *out*[n] is the union of the live-in sets of all successors of n.

Algorithm 10.4 finds a solution to these equations by iteration. As usual, we initialize *in*[n] and *out*[n] to the the empty set { }, for all n, then repeatedly treat the equations as assignment statements until a fixed point is reached.

Table 10.5 shows the results of running the algorithm on Graph 10.1. The columns 1st, 2nd, etc., are the values of in and out on successive iterations of the **repeat** loop. Since the 7th column is the same as the 6th, the algorithm terminates.

We can speed the convergence of this algorithm significantly by ordering the nodes properly. Suppose there is an edge $3 \rightarrow 4$ in the graph. Since *in*[4]

			1st		2nd		3rd		4th		5th		6th		7th	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	b	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c		c	

TABLE 10.5. Liveness calculation following forward control-flow edges.

			1st		2nd		3rd	
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

TABLE 10.6. Liveness calculation following reverse control-flow edges.

is computed from *out*[4], and *out*[3] is computed from *in*[4], and so on, we should compute the in and out sets in the order *out*[4] → *in*[4] → *out*[3] → *in*[3]. But in Table 10.5, just the opposite order is used in each iteration! We have waited as long as possible (in each iteration) to make use of information gained from the previous iteration.

Table 10.6 shows the computation, in which each **for** loop iterates from 6 to 1 (approximately following the *reversed* direction of the flow-graph arrows), and in each iteration the out sets are computed before the in sets. By the end of the second iteration, the fixed point has been found; the third iteration just confirms this.

When solving dataflow equations by iteration, the order of computation should follow the "flow." Since liveness flows *backward* along control-flow arrows, and from "out" to "in," so should the computation.

Ordering the nodes can be done easily by depth-first search, as shown in Section 17.4.

Basic blocks. Flow-graph nodes that have only one predecessor and one successor are not very interesting. Such nodes can be merged with their predecessors and successors; what results is a graph with many fewer nodes, where each node represents a basic block. The algorithms that operate on flow graphs, such as liveness analysis, go much faster on the smaller graphs. Chapter 17 explains how to adjust the dataflow equations to use basic blocks. In this chapter we keep things simple.

One variable at a time. Instead of doing dataflow "in parallel" using set equations, it can be just as practical to compute dataflow for one variable at a time as information about that variable is needed. For liveness, this would mean repeating the dataflow traversal once for each temporary. Starting from each *use* site of a temporary t, and tracing backward (following *predecessor* edges of the flow graph) using depth-first search, we note the liveness of t at each flow-graph node. The search stops at any definition of the temporary. Although this might seem expensive, many temporaries have very short live ranges, so the searches terminate quickly and do not traverse the entire flow graph for most variables.

REPRESENTATION OF SETS

There are at least two good ways to represent sets for dataflow equations: as arrays of bits or as sorted lists of variables.

If there are N variables in the program, the bit-array representation uses N bits for each set. Calculating the union of two sets is done by *or*-ing the corresponding bits at each position. Since computers can represent K bits per word (with $K = 32$ typical), one set-union operation takes N/K operations.

A set can also be represented as a linked list of its members, sorted by any totally ordered key (such as variable name). Calculating the union is done by merging the lists (discarding duplicates). This takes time proportional to the size of the sets being unioned.

Clearly, when the sets are sparse (fewer than N/K elements, on the average), the sorted-list representation is asymptotically faster; when the sets are dense, the bit-array representation is better.

TIME COMPLEXITY

How fast is iterative dataflow analysis?

A program of size N has at most N nodes in the flow graph, and at most N variables. Thus, each live-in set (or live-out set) has at most N elements;

			X		Y		Z	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a	c	ac	cd	acd	c	ac
2	a	b	ac	bc	acd	bcd	ac	b
3	bc	c	bc	bc	bcd	bcd	b	b
4	b	a	bc	ac	bcd	acd	b	ac
5	a		ac	ac	acd	acd	ac	ac
6	c		c		c		c	

TABLE 10.7. X and Y are solutions to the liveness equations; Z is not a solution.

each set-union operation to compute live-in (or live-out) takes $O(N)$ time.

The **for** loop computes a constant number of set operations per flow-graph node; there are $O(N)$ nodes; thus, the **for** loop takes $O(N^2)$ time.

Each iteration of the **repeat** loop can only make each in or out set larger, never smaller. This is because the in and out sets are *monotonic* with respect to each other. That is, in the equation $in[n] = use[n] \cup (out[n] - def[n])$, a larger $out[n]$ can only make $in[n]$ larger. Similarly, in $out[n] = \bigcup_{s \in succ[n]} in[s]$, a larger $in[s]$ can only make $out[n]$ larger.

Each iteration must add something to the sets; but the sets cannot keep growing infinitely; at most every set can contain every variable. Thus, the sum of the sizes of all in and out sets is $2N^2$, which is the most that the repeat loop can iterate.

Thus, the worst-case run time of this algorithm is $O(N^4)$. Ordering the nodes using depth-first search (Algorithm 17.5, page 363) usually brings the number of **repeat**-loop iterations to two or three, and the live sets are often sparse, so the algorithm runs between $O(N)$ and $O(N^2)$ in practice.

Section 17.4 discusses more sophisticated ways of solving dataflow equations quickly.

LEAST FIXED POINTS

Table 10.7 illustrates two solutions (and a nonsolution!) to the Equations 10.3; assume there is another program variable d not used in this fragment of the program.

In solution Y, the variable d is carried uselessly around the loop. But in fact, Y satisfies Equations 10.3 just as X does. What does this mean? Is d live or not?

The answer is that any solution to the dataflow equations is a *conservative approximation*. If the value of variable a will truly be needed in some execution of the program when execution reaches node n of the flow graph, then we can be assured that a is live-out at node n in any solution of the equations. But the converse is not true; we might calculate that d is live-out, but that doesn't mean that its value will really be used.

Is this acceptable? We can answer that question by asking what use will be made of the dataflow information. In the case of liveness analysis, if a variable is *thought to be live,* then we will make sure to have its value in a register. A conservative approximation of liveness is one that may erroneously believe a variable is live, but will never erroneously believe it is dead. The consequence of a conservative approximation is that the compiled code might use more registers than it really needs; but it will compute the right answer.

Consider instead the live-in sets Z, which fail to satisfy the dataflow equations. Using this Z we think that b and c are never live at the same time, and we would assign them to the same register. The resulting program would use an optimal number of registers but *compute the wrong answer*.

A dataflow equation used for compiler optimization should be set up so that any solution to it provides conservative information to the optimizer; imprecise information may lead to suboptimal but never incorrect programs.

Theorem. Equations 10.3 have more than one solution.

Proof. X and Y are both solutions.

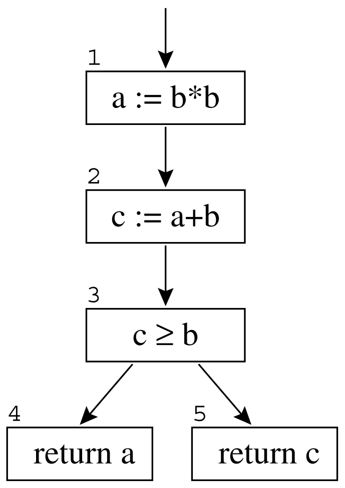
Theorem. All solutions to Equations 10.3 contain solution X. That is, if $in_X[n]$ and $in_Y[n]$ are the live-in sets for some node n in solutions X and Y, then $in_X[n] \subseteq in_Y[n]$.

Proof. See Exercise 10.2.

We say that X is the *least solution* to Equations 10.3. Clearly, since a bigger solution will lead to using more registers (producing suboptimal code), we want to use the least solution. Fortunately, Algorithm 10.4 always computes the least fixed point.

STATIC VS. DYNAMIC LIVENESS

A variable is live "if its value will be used in the future." In Graph 10.8, we know that $b \times b$ must be nonnegative, so that the test $c \geq b$ will be true. Thus,



GRAPH 10.8. Standard static dataflow analysis will not take advantage of the fact that node 4 can never be reached.

node 4 will never be reached, and a's value will not be used after node 2; a is not live-out of node 2.

But Equations 10.3 say that a is live-in to node 4, and therefore live-out of nodes 3 and 2. The equations are ignorant of which way the conditional branch will go. "Smarter" equations would permit a and c to be assigned the same register.

Although we can prove here that $b * b \geq 0$, and we could have the compiler look for arithmetic identities, no compiler can ever fully understand how all the control flow in every program will work. This is a fundamental mathematical theorem, derivable from the halting problem.

Theorem. There is no program H that takes as input any program P and input X and (without infinite-looping) returns true if $P(X)$ halts and false if $P(X)$ infinite-loops.

Proof. Suppose that there were such a program H; then we could arrive at a contradiction as follows. From the program H, construct the function F,

$$F(Y) = \textbf{if } H(Y, Y) \textbf{ then } (\textbf{while} \text{ true } \textbf{do} \text{ ()}) \textbf{ else} \text{ true}$$

By the definition of H, if $F(F)$ halts, then $H(F, F)$ is true; so the **then** clause is taken; so the **while** loop executes forever; so $F(F)$ does not halt. But if $F(F)$ loops forever, then $H(F, F)$ is false; so the **else** clause is taken; so $F(F)$ halts. The program $F(F)$ halts if it doesn't halt, and doesn't halt if it halts: a contradiction. Thus there can be no program H that tests whether another program halts (and always halts itself).

Corollary. No program $H'(X, L)$ can tell, for any program X and label L within X, whether the label L is ever reached on an execution of X.

Proof. From H' we could construct H. In some program that we want to test for halting, just let L be the end of the program, and replace all instances of the **halt** command with **goto** L.

Conservative approximation. This theorem does not mean that we can *never* tell if a given label is reached or not, just that there is not a general algorithm that can *always* tell. We could improve our liveness analysis with some special-case algorithms that, in some cases, calculate more information about run-time control flow. But any such algorithm will come up against many cases where it simply cannot tell exactly what will happen at run time.

Because of this inherent limitation of program analysis, no compiler can really tell if a variable's value is truly needed – whether the variable is truly live. Instead, we have to make do with a conservative approximation. We assume that any conditional branch goes both ways. Thus, we have a dynamic condition and its static approximation:

Dynamic liveness A variable a is dynamically live at node n if some execution of the program goes from n to a use of a without going through any definition of a.

Static liveness A variable a is statically live at node n if there is some path of control-flow edges from n to some use of a that does not go through a definition of a.

Clearly, if a is dynamically live, it is also statically live. An optimizing compiler must allocate registers, and do other optimizations, on the basis of static liveness, because (in general) dynamic liveness cannot be computed.

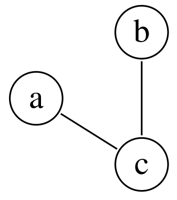
INTERFERENCE GRAPHS

Liveness information is used for several kinds of optimizations in a compiler. For some optimizations, we need to know exactly which variables are live at each node in the flow graph.

One of the most important applications of liveness analysis is for register allocation: We have a set of temporaries $a, b, c, \ldots$ that must be allocated to

	a	b	c
a			**x**
b			**x**
c	**x**	**x**	

(a) Matrix



(b) Graph

FIGURE 10.9. Representations of interference.

registers $r_1, \ldots, r_k$. A condition that prevents a and b from being allocated to the same register is called an *interference*.

The most common kind of interference is caused by overlapping live ranges: When a and b are both live at the same program point, then they cannot be put in the same register. But there are some other causes of interference: for example, when a must be generated by an instruction that cannot address register r_1, then a and r_1 interfere.

Interference information can be expressed as a matrix; Figure 10.9a has an **x** marking interferences of the variables in Graph 10.1. The interference matrix can also be expressed as an undirected graph (Figure 10.9b), with a node for each variable, and edges connecting variables that interfere.

Special treatment of MOVE instructions. In static liveness analysis, we can give MOVE instructions special consideration. It is important not to create artifical interferences between the source and destination of a move. Consider the program:

$$
\begin{array}{ll}
t \leftarrow s & (\textit{copy}) \\
\quad\vdots & \\
x \leftarrow \ldots s \ldots & (\textit{use of } s) \\
\quad\vdots & \\
y \leftarrow \ldots t \ldots & (\textit{use of } t)
\end{array}
$$

After the copy instruction both s and t are live, and normally we would make an interference edge (s, t) since t is being defined at a point where s is live. But we do not need separate registers for s and t, since they contain the same value. The solution is just not to add an interference edge (t, s) in this case. Of course, if there is a later (nonmove) definition of t while s is still live, that will create the interference edge (t, s).

Therefore, the way to add interference edges for each new definition is

```
package Graph;

public class Graph {
  public Graph();
  public NodeList nodes();
  public Node newNode();
  public void addEdge(Node from, Node to);
  public void rmEdge(Node from, Node to);
  public void show(java.io.PrintStream out);
}

public class Node {
    public Node(Graph g);
    public NodeList succ();
    public NodeList pred();
    public NodeList adj();
    public int outDegree();
    public int inDegree();
    public int degree();
    public boolean goesTo(Node n);
    public boolean comesFrom(Node n);
    public boolean adj(Node n);
    public String toString();
}
```

PROGRAM 10.10. The `Graph` abstract data type.

1. At any nonmove instruction that *defines* a variable a, where the *live-out* variables are $b_1, \ldots, b_j$, add interference edges $(a, b_1), \ldots, (a, b_j)$.
2. At a move instruction $a \leftarrow c$, where variables $b_1, \ldots, b_j$ are *live-out*, add interference edges $(a, b_1), \ldots, (a, b_j)$ for any b_i that is *not* the same as c.

10.2 LIVENESS IN THE MiniJava COMPILER

The flow analysis for the MiniJava compiler is done in two stages: First, the control flow of the `Assem` program is analyzed, producing a control-flow graph; then, the liveness of variables in the control-flow graph is analyzed, producing an interference graph.

GRAPHS

To represent both kinds of graphs, let's make a `Graph` abstract data type (Program 10.10).

The constructor `Graph()` creates an empty directed graph; `g.newNode()` makes a new node within a graph g. A directed edge from n to m is created by `g.addEdge(n,m)`; after that, m will be found in the list `n.succ()` and n will be in `m.pred()`. When working with undirected graphs, the function `adj` is useful: $m.\texttt{adj}() = m.\texttt{succ}() \cup m.\texttt{pred}()$.

To delete an edge, use `rmEdge`. To test whether m and n are the same node, use `m==n`.

When using a graph in an algorithm, we want each node to represent something (an instruction in a program, for example). To make mappings from nodes to the things they are supposed to represent, we use a `Hashtable`. The following idiom associates information x with node n in a mapping `mytable`.

```
java.util.Dictionary mytable = new java.util.Hashtable();
  ···  mytable.put(n,x);
```

CONTROL-FLOW GRAPHS

The `FlowGraph` package manages control-flow graphs. Each instruction (or basic block) is represented by a node in the flow graph. If instruction m can be followed by instruction n (either by a jump or by falling through), then there will be an edge (m, n) in the graph.

```
public abstract class FlowGraph extends Graph.Graph {
   public abstract TempList def(Node node);
   public abstract TempList use(Node node);
   public abstract boolean isMove(Node node);
   public void show(java.io.PrintStream out);
}
```

Each `Node` of the flow graph represents an instruction (or, perhaps, a basic block). The `def()` method tells what temporaries are defined at this node (destination registers of the instruction). `use()` tells what temporaries are used at this node (source registers of the instruction). `isMove` tells whether this instruction is a `MOVE` instruction, one that could be deleted if the `def` and `use` were identical.

The `AssemFlowGraph` class provides an implementation of `FlowGraph` for `Assem` instructions.

```
package FlowGraph;
public class AssemFlowGraph extends FlowGraph {
     public Instr instr(Node n);
     public AssemFlowGraph(Assem.InstrList instrs);
}
```

The constructor `AssemFlowGraph` takes a list of instructions and returns a flow graph. In making the flow graph, the `jump` fields of the `instrs` are used in creating control-flow edges, and the `use` and `def` information (obtained from the `src` and `dst` fields of the `instrs`) is attached to the nodes by means of the `use` and `def` methods of the `flowgraph`.

Information associated with the nodes. For a flow graph, we want to associate some *use* and *def* information with each node in the graph. Then the liveness-analysis algorithm will also want to remember *live-in* and *live-out* information at each node. We could make room in the `Node` class to store all of this information. This would work well and would be quite efficient. However, it may not be very modular. Eventually we may want to do other analyses on flow graphs, which remember other kinds of information about each node. We may not want to modify the data structure (which is a widely used interface) for each new analysis.

Instead of storing the information *in* the nodes, a more modular approach is to say that a graph is a graph, and that a flow graph is a graph along with separately packaged auxiliary information (tables, or functions mapping nodes to whatever). Similarly, a dataflow algorithm on a graph does not need to modify dataflow information *in* the nodes, but modifies its own privately held mappings.

There may be a trade-off here between efficiency and modularity, since it may be faster to keep the information *in* the nodes, accessible by a simple pointer-traversal instead of a hash-table or search-tree lookup.

LIVENESS ANALYSIS

The `RegAlloc` package has an abstract class `InterferenceGraph` to indicate which pairs of temporaries cannot share a register:

```
package RegAlloc;
abstract public class InterferenceGraph extends Graph.Graph{
   abstract public Graph.Node tnode(Temp.Temp temp);
   abstract public Temp.Temp gtemp(Node node);
   abstract public MoveList moves();
   public int spillCost(Node node);
}
```

The method `tnode` relates a `Temp` to a `Node`, and `gtemp` is the inverse map. The method `moves` tells what `MOVE` instructions are associated with this graph (this is a hint about what pairs of temporaries to try to allocate to the same register). The `spillCost(n)` is an estimate of how many extra instructions would be executed if n were kept in memory instead of in registers; for a naive spiller, it suffices to return 1 for every n.

The class `Liveness` produces an interference graph from a flow graph:

```
package RegAlloc;
public class Liveness extends InterferenceGraph {
   public Liveness(FlowGraph flow);
}
```

In the implementation of the `Liveness` module, it is useful to maintain a data structure that remembers what is live at the exit of each flow-graph node:

```
private java.util.Dictionary liveMap =
                           new java.util.Hashtable();
```

where the keys are nodes and objects are `TempLists`. Given a flow-graph node n, the set of live temporaries at that node can be looked up in a global `liveMap`.

Having calculated a complete `liveMap`, we can now construct an interference graph. At each flow node n where there is a newly defined temporary $d \in \mathit{def}(n)$, and where temporaries $\{t_1, t_2, \ldots\}$ are in the `liveMap`, we just add interference edges (d, t_1), (d, t_2), ... For MOVEs, these edges will be safe but suboptimal; pages 213–214 describe a better treatment.

What if a newly defined temporary is not live just after its definition? This would be the case if a variable is defined but never used. It would seem that there's no need to put it in a register at all; thus it would not interfere with any other temporaries. But if the defining instruction is going to execute (perhaps it is necessary for some other side effect of the instruction), then it *will* write to some register, and that register had better not contain any other live variable. Thus, zero-length live ranges *do* interfere with any live ranges that overlap them.

PROGRAM CONSTRUCTING FLOW GRAPHS

Implement the `AssemFlowGraph` class that turns a list of `Assem` instructions into a flow graph. Use the abstract classes `Graph.Graph` and `FlowGraph.FlowGraph` provided in `$MINIJAVA/chap10`.

PROGRAM LIVENESS

Implement the `Liveness` module. Use either the set-equation algorithm with the array-of-boolean or sorted-list-of-temporaries representation of sets, or the one-variable-at-a-time method.

EXERCISES

10.1 Perform flow analysis on the program of Exercise 8.6:

a. Draw the control-flow graph.

b. Calculate live-in and live-out at each statement.

c. Construct the register interference graph.

****10.2** Prove that Equations 10.3 have a least fixed point and that Algorithm 10.4 always computes it.

Hint: We know the algorithm refuses to terminate until it has a fixed point. The questions are whether (a) it must eventually terminate, and (b) the fixed point it computes is smaller than all other fixed points. For (a) show that the sets can only get bigger. For (b) show by induction that at any time the *in* and *out* sets are subsets of those in any possible fixed point. This is clearly true initially, when *in* and *out* are both empty; show that each step of the algorithm preserves the invariant.

***10.3** Analyze the asymptotic complexity of the one-variable-at-a-time method of computing dataflow information.

***10.4** Analyze the worst-case asymptotic complexity of making an interference graph, for a program of size N (with at most N variables and at most N control-flow nodes). Assume the dataflow analysis is already done and that *use*, *def*, and *live-out* information for each node can be queried in constant time. What representation of graph adjacency matrices should be used for efficiency?

10.5 The DEC Alpha architecture places the following restrictions on floating-point instructions, for programs that wish to recover from arithmetic exceptions:

1. Within a basic block (actually, in any sequence of instructions not separated by a *trap-barrier* instruction), no two instructions should write to the same destination register.
2. A source register of an instruction cannot be the same as the destination register of that instruction or any later instruction in the basic block.

$r_1 + r_5 \rightarrow r_4$	$r_1 + r_5 \rightarrow r_4$	$r_1 + r_5 \rightarrow r_3$	$r_1 + r_5 \rightarrow r_4$
$r_3 \times r_2 \rightarrow r_4$	$r_4 \times r_2 \rightarrow r_1$	$r_4 \times r_2 \rightarrow r_4$	$r_4 \times r_2 \rightarrow r_6$
violates rule 1.	*violates rule 2.*	*violates rule 2.*	*OK*

Show how to express these restrictions in the register interference graph.

11 Register Allocation

reg-is-ter: a device for storing small amounts of data
al-lo-cate: to apportion for a specific purpose

Webster's Dictionary

The `Translate`, `Canon`, and `Codegen` phases of the compiler assume that there are an infinite number of registers to hold temporary values and that MOVE instructions cost nothing. The job of the register allocator is to assign the many temporaries to a small number of machine registers, and, where possible, to assign the source and destination of a MOVE to the same register so that the MOVE can be deleted.

From an examination of the control and dataflow graph, we derive an *interference graph*. Each node in the interference graph represents a temporary value; each edge (t_1, t_2) indicates a pair of temporaries that cannot be assigned to the same register. The most common reason for an interference edge is that t_1 and t_2 are live at the same time. Interference edges can also express other constraints; for example, if a certain instruction $a \leftarrow b \oplus c$ cannot produce results in register r_{12} on our machine, we can make a interfere with r_{12}.

Next we *color* the interference graph. We want to use as few colors as possible, but no pair of nodes connected by an edge may be assigned the same color. Graph coloring problems derive from the old mapmakers' rule that adjacent countries on a map should be colored with different colors. Our "colors" correspond to registers: If our target machine has K registers, and we can K-color the graph (color the graph with K colors), then the coloring is a valid register assignment for the interference graph. If there is no K-coloring, we will have to keep some of our variables and temporaries in memory instead of registers; this is called *spilling*.

11.1 COLORING BY SIMPLIFICATION

Register allocation is an NP-complete problem (except in special cases, such as expression trees); graph coloring is also NP-complete. Fortunately there is a linear-time approximation algorithm that gives good results; its principal phases are **Build**, **Simplify**, **Spill**, and **Select**.

Build: Construct the interference graph. We use dataflow analysis to compute the set of temporaries that are simultaneously live at each program point, and we add an edge to the graph for each pair of temporaries in the set. We repeat this for all program points.

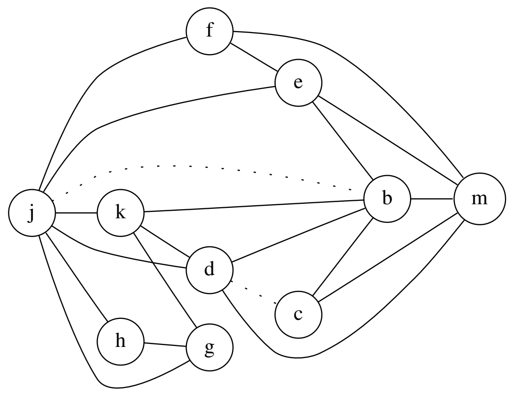
Simplify: We color the graph using a simple heuristic. Suppose the graph G contains a node m with fewer than K neighbors, where K is the number of registers on the machine. Let G' be the graph $G - \{m\}$ obtained by removing m. If G' can be colored, then so can G, for when m is added to the colored graph G', the neighbors of m have at most $K - 1$ colors among them, so a free color can always be found for m. This leads naturally to a stack-based (or recursive) algorithm for coloring: We repeatedly remove (and push on a stack) nodes of degree less than K. Each such simplification will decrease the degrees of other nodes, leading to more opportunity for simplification.

Spill: Suppose at some point during simplification the graph G has nodes only of *significant degree*, that is, nodes of degree $\geq K$. Then the *simplify* heuristic fails, and we mark some node for spilling. That is, we choose some node in the graph (standing for a temporary variable in the program) and decide to represent it in memory, not registers, during program execution. An optimistic approximation to the effect of spilling is that the spilled node does not interfere with any of the other nodes remaining in the graph. It can therefore be removed and pushed on the stack, and the simplify process continued.

Select: We assign colors to nodes in the graph. Starting with the empty graph, we rebuild the original graph by repeatedly adding a node from the top of the stack. When we add a node to the graph, there must be a color for it, as the premise for removing it in the simplify phase was that it could always be assigned a color provided the remaining nodes in the graph could be successfully colored.

When *potential spill* node n that was pushed using the *Spill* heuristic is

```
live-in:  k j
      g := mem[j+12]
      h := k - 1
      f := g * h
      e := mem[j+8]
      m := mem[j+16]
      b := mem[f]
      c := e + 8
      d := c
      k := m + 4
      j := b
live-out:  d k j
```



GRAPH 11.1. Interference graph for a program. Dotted lines are not interference edges but indicate move instructions.

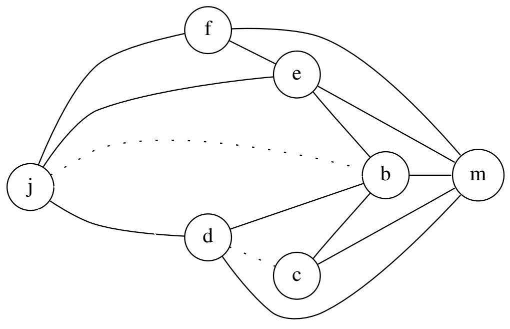
popped, there is no guarantee that it will be colorable: Its neighbors in the graph may be colored with K different colors already. In this case, we have an *actual spill*. We do not assign any color, but we continue the *Select* phase to identify other actual spills.

But perhaps some of the neighbors are the same color, so that among them there are fewer than K colors. Then we can color n, and it does not become an actual spill. This technique is known as *optimistic coloring*.

Start over: If the **Select** phase is unable to find a color for some node(s), then the program must be rewritten to fetch them from memory just before each use, and store them back after each definition. Thus, a spilled temporary will turn into several new temporaries with tiny live ranges. These will interfere with other temporaries in the graph. So the algorithm is repeated on this rewritten program. This process iterates until *simplify* succeeds with no spills; in practice, one or two iterations almost always suffice.

EXAMPLE

Graph 11.1 shows the interferences for a simple program. The nodes are labeled with the temporaries they represent, and there is an edge between two nodes if they are simultaneously live. For example, nodes `d`, `k`, and `j` are all connected since they are live simultaneously at the end of the block. Assuming that there are four registers available on the machine, then the simplify phase can start with the nodes `g`, `h`, `c`, and `f` in its working set, since they



GRAPH 11.2. After removal of h, g, k.

(a) stack	(b) assignment
m	1
c	3
b	2
f	2
e	4
j	3
d	4
k	1
h	2
g	4

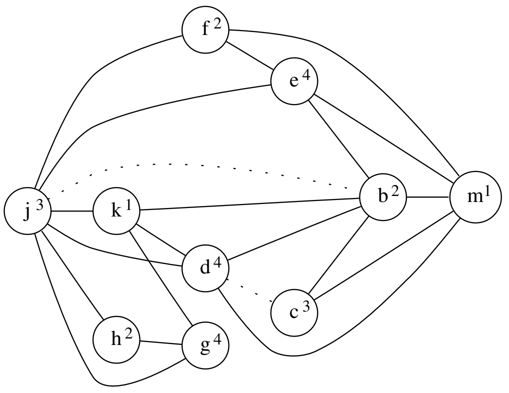


FIGURE 11.3. Simplification stack, and a possible coloring.

have less than four neighbors each. A color can always be found for them if the remaining graph can be successfully colored. If the algorithm starts by removing h and g and all their edges, then node k becomes a candidate for removal and can be added to the work list. Graph 11.2 remains after nodes g, h, and k have been removed. Continuing in this fashion a possible order in which nodes are removed is represented by the stack shown in Figure 11.3a, where the stack grows upward.

The nodes are now popped off the stack and the original graph reconstructed and colored simultaneously. Starting with m, a color is chosen arbitrarily since the graph at this point consists of a singleton node. The next node to be put into the graph is c. The only constraint is that it be given a color different from m, since there is an edge from m to c. A possible assignment of colors for the reconstructed original graph is shown in Figure 11.3b.

11.2 COALESCING

It is easy to eliminate redundant move instructions with an interference graph. If there is no edge in the interference graph between the source and destination of a move instruction, then the move can be eliminated. The source and destination nodes are *coalesced* into a new node whose edges are the union of those of the nodes being replaced.

In principle, any pair of nodes not connected by an interference edge could be coalesced. This aggressive form of copy propagation is very successful at eliminating move instructions. Unfortunately, the node being introduced is more constrained than those being removed, as it contains a union of edges. Thus, it is quite possible that a graph, colorable with K colors before coalescing, may no longer be K-colorable after reckless coalescing. We wish to coalesce only where it is *safe* to do so, that is, where the coalescing will not render the graph uncolorable. Both of the following strategies are safe:

Briggs: Nodes a and b can be coalesced if the resulting node ab will have fewer than K neighbors of significant degree (i.e., having $\geq K$ edges). The coalescing is guaranteed not to turn a K-colorable graph into a non-K-colorable graph, because after the simplify phase has removed all the insignificant-degree nodes from the graph, the coalesced node will be adjacent only to those neighbors that were of significant degree. Since there are fewer than K of these, *simplify* can then remove the coalesced node from the graph. Thus if the original graph was colorable, the conservative coalescing strategy does not alter the colorability of the graph.

George: Nodes a and b can be coalesced if, for every neighbor t of a, either t already interferes with b or t is of insignificant degree. This coalescing is safe, by the following reasoning. Let S be the set of insignificant-degree neighbors of a in the original graph. If the coalescing were not done, *simplify* could remove all the nodes in S, leaving a reduced graph G_1. If the coalescing is done, then *simplify* can remove all the nodes in S, leaving a graph G_2. But G_2 is a subgraph of G_1 (the node ab in G_2 corresponds to the node b in G_1), and thus must be at least as easy to color.

These strategies are *conservative*, because there are still safe situations in which they will fail to coalesce. This means that the program may perform some unnecessary MOVE instructions – but this is better than spilling!

Interleaving simplification steps with conservative coalescing eliminates most move instructions, while still guaranteeing not to introduce spills. The coalesce, simplify, and spill procedures should be alternated until the graph is empty, as shown in Figure 11.4.

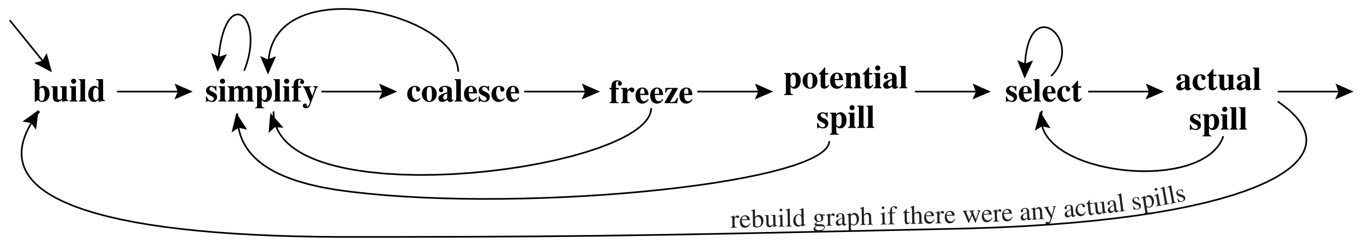


FIGURE 11.4. Graph coloring with coalescing.

These are the phases of a register allocator with coalescing:

Build: Construct the interference graph, and categorize each node as either *move-related* or *non-move-related*. A move-related node is one that is either the source or destination of a move instruction.

Simplify: One at a time, remove non-move-related nodes of low ($< K$) degree from the graph.

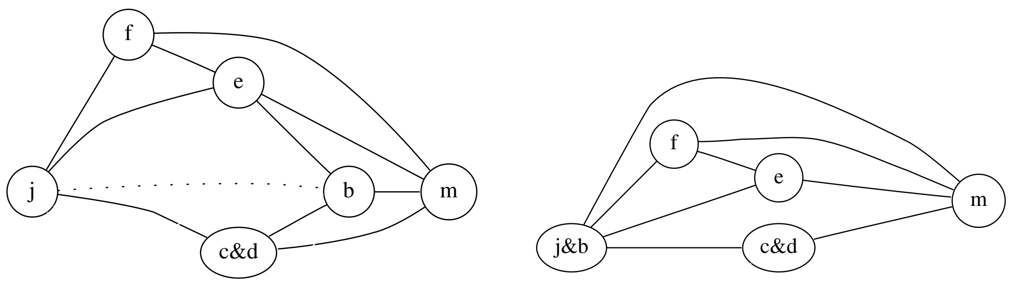
Coalesce: Perform conservative coalescing on the reduced graph obtained in the simplification phase. Since the degrees of many nodes have been reduced by *simplify*, the conservative strategy is likely to find many more moves to coalesce than it would have in the initial interference graph. After two nodes have been coalesced (and the move instruction deleted), if the resulting node is no longer move-related, it will be available for the next round of simplification. *Simplify* and *coalesce* are repeated until only significant-degree or move-related nodes remain.

Freeze: If neither *simplify* nor *coalesce* applies, we look for a move-related node of low degree. We *freeze* the moves in which this node is involved: That is, we give up hope of coalescing those moves. This causes the node (and perhaps other nodes related to the frozen moves) to be considered non-move-related, which should enable more simplification. Now, *simplify* and *coalesce* are resumed.

Spill: If there are no low-degree nodes, we select a significant-degree node for potential spilling and push it on the stack.

Select: Pop the entire stack, assigning colors.

Consider Graph 11.1; nodes `b`, `c`, `d`, and `j` are the only move-related nodes. The initial work list used in the simplify phase must contain only non-move-



GRAPH 11.5. (a) after coalescing `c` and `d`; (b) after coalescing `b` and `j`.

stack	coloring
e	1
m	2
f	3
j&b	4
c&d	1
k	2
h	2
g	1

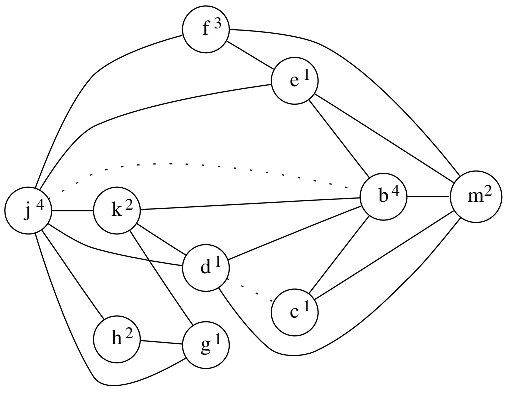


FIGURE 11.6. A coloring, with coalescing, for Graph 11.1.

related nodes and consists of nodes `g`, `h`, and `f`. Once again, after removal of `g`, `h`, and `k` we obtain Graph 11.2.

We could continue the simplification phase further; however, if we invoke a round of coalescing at this point, we discover that `c` and `d` are indeed coalesceable as the coalesced node has only two neighbors of significant degree: `m` and `b`. The resulting graph is shown in Graph 11.5a, with the coalesced node labeled as `c&d`.

From Graph 11.5a we see that it is possible to coalesce `b` and `j` as well. Nodes `b` and `j` are adjacent to two neighbors of significant degree, namely `m` and `e`. The result of coalescing `b` and `j` is shown in Graph 11.5b.

After coalescing these two moves, there are no more move-related nodes, and therefore no more coalescing is possible. The simplify phase can be invoked one more time to remove all the remaining nodes. A possible assignment of colors is shown in Figure 11.6.

Some moves are neither coalesced nor frozen. Instead, they are *constrained*. Consider the graph x, y, z, where (x, z) is the only interference edge and there are two moves $x \leftarrow y$ and $y \leftarrow z$. Either move is a candidate for coalescing. But after x and y are coalesced, the remaining move $xy \leftarrow z$ cannot be coalesced because of the interference edge (xy, z). We say this move is *constrained*, and we remove it from further consideration: It no longer causes nodes to be treated as move-related.

SPILLING

If spilling is necessary, *build* and *simplify* must be repeated on the whole program. The simplest version of the algorithm discards any coalescences found if *build* must be repeated. Then it is easy to see that coalescing does not increase the number of spills in any future round of *build*. A more efficient algorithm preserves any coalescences done *before the first potential spill was discovered*, but discards (uncoalesces) any coalescences done after that point.

Coalescing of spills. On a machine with many registers (> 20), there will usually be few spilled nodes. But on a six-register machine (such as the Intel Pentium), there will be many spills. The front end may have generated many temporaries, and transformations such as SSA (described in Chapter 19) may split them into many more temporaries. If each spilled temporary lives in its own stack-frame location, then the frame may be quite large.

Even worse, there may be many move instructions involving pairs of spilled nodes. But to implement $a \leftarrow b$ when a and b are both spilled temporaries requires a fetch-store sequence, $t \leftarrow M[b_{\text{loc}}];\ M[a_{\text{loc}}] \leftarrow t$. This is expensive, and also defines a temporary t that itself may cause other nodes to spill.

But many of the spill pairs are never live simultaneously. Thus, they may be graph-colored, with coalescing! In fact, because there is no fixed limit to the number of stack-frame locations, we can coalesce aggressively, without worrying about how many high-degree neighbors the spill nodes have. The algorithm is thus:

1. Use liveness information to construct the interference graph for spilled nodes.
2. While there is any pair of noninterfering spilled nodes connected by a move instruction, coalesce them.
3. Use *simplify* and *select* to color the graph. There is no (further) spilling in this coloring; instead, *simplify* just picks the lowest-degree node, and *select* picks the first available color, without any predetermined limit on the number of colors.
4. The colors correspond to activation-record locations for the spilled variables.

This should be done *before* generating the spill instructions and regenerating the register-temporary interference graph, so as to avoid creating fetch-store sequences for coalesced moves of spilled nodes.

11.3 PRECOLORED NODES

Some temporaries are *precolored* – they represent machine registers. The front end generates these when interfacing to standard calling conventions across module boundaries, for example. For each actual register that is used for some specific purpose, such as the frame pointer, standard-argument-1-register, standard-argument-2-register, and so on, the `Codegen` or `Frame` module should use the particular temporary that is permanently bound to that register (see also page 251). For any given color (that is, for any given machine register) there should be only one precolored node of that color.

The *select* and *coalesce* operations can give an ordinary temporary the same color as a precolored register, as long as they don't interfere, and in fact this is quite common. Thus, a standard calling-convention register can be reused inside a procedure as a temporary variable. Precolored nodes may be coalesced with other (non-precolored) nodes using conservative coalescing.

For a K-register machine, there will be K precolored nodes that all interfere with each other. Those of the precolored nodes that are not used explicitly (in a parameter-passing convention, for example) will not interfere with any ordinary (non-precolored) nodes; but a machine register used explicitly will have a live range that interferes with any other variables that happen to be live at the same time.

We cannot *simplify* a precolored node – this would mean pulling it from the graph in the hope that we can assign it a color later, but in fact we have no freedom about what color to assign it. And we should not spill precolored nodes to memory, because the machine registers are by definition *registers*. Thus, we should treat them as having "infinite" degree.

TEMPORARY COPIES OF MACHINE REGISTERS

The coloring algorithm works by calling *simplify, coalesce,* and *spill* until only the precolored nodes remain, and then the *select* phase can start adding the other nodes (and coloring them).

Because precolored nodes do not spill, the front end must be careful to keep their live ranges short. It can do this by generating MOVE instructions to move values to and from precolored nodes. For example, suppose r_7 is a callee-save register; it is "defined" at procedure entry and "used" at procedure exit. Instead of being kept in a precolored register throughout the procedure (Figure 11.7a), it can be moved into a fresh temporary and then moved back

$$
(a)\quad \begin{array}{ll}
\text{enter:} & \text{def}(r_7) \\
 & \vdots \\
\text{exit:} & \text{use}(r_7)
\end{array}
\qquad\qquad
(b)\quad \begin{array}{ll}
\text{enter:} & \text{def}(r_7) \\
 & t_{231} \leftarrow r_7 \\
 & \vdots \\
 & r_7 \leftarrow t_{231} \\
\text{exit:} & \text{use}(r_7)
\end{array}
$$

FIGURE 11.7. Moving a callee-save register to a fresh temporary.

(Figure 11.7b). If there is *register pressure* (a high demand for registers) in this function, t_{231} will spill; otherwise t_{231} will be coalesced with r_7 and the MOVE instructions will be eliminated.

CALLER-SAVE AND CALLEE-SAVE REGISTERS

A local variable or compiler temporary that is not live across any procedure call should usually be allocated to a caller-save register, because in this case no saving and restoring of the register will be necessary at all. On the other hand, any variable that is live across several procedure calls should be kept in a callee-save register, since then only one save/restore will be necessary (on entry/exit from the calling procedure).

How can the register allocator allocate variables to registers using this criterion? Fortunately, a graph-coloring allocator can do this very naturally, as a byproduct of ordinary coalescing and spilling. All the callee-save registers are considered live on entry to the procedure, and are *used* by the return instruction. The CALL instructions in the `Assem` language have been annotated to *define* (interfere with) all the caller-save registers. If a variable is not live across a procedure call, it will tend to be allocated to a caller-save register.

If a variable x is live across a procedure call, then it interferes with all the caller-save (precolored) registers, *and* it interferes with all the new temporaries (such as t_{231} in Figure 11.7) created for callee-save registers. Thus, a spill will occur. Using the common spill-cost heuristic that spills a node with high degree but few uses, the node chosen for spilling will not be x but t_{231}. Since t_{231} is spilled, r_7 will be available for coloring x (or some other variable). Essentially, the callee saves the callee-save register by spilling t_{231}.

EXAMPLE WITH PRECOLORED NODES

A worked example will illustrate the issues of register allocation with precolored nodes, callee-save registers, and spilling.

```
int f(int a, int b) {
   int d=0;
   int e=a;
   do {d = d+b;
       e = e-1;
   } while (e>0);
   return d;
}
```

(a)

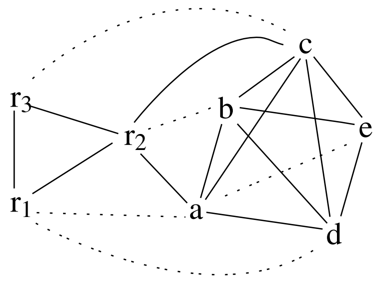
```
enter:  $c \leftarrow r_3$
        $a \leftarrow r_1$
        $b \leftarrow r_2$
        $d \leftarrow 0$
        $e \leftarrow a$
loop:   $d \leftarrow d + b$
        $e \leftarrow e - 1$
        if $e > 0$ goto loop
        $r_1 \leftarrow d$
        $r_3 \leftarrow c$
        return          ($r_1, r_3$ *live out*)
```

(b)

PROGRAM 11.8. A C function and its translation into instructions

A C compiler is compiling Program 11.8a for a target machine with three registers; r_1 and r_2 are caller-save, and r_3 is callee-save. The code generator has therefore made arrangements to preserve the value of r_3 explicitly, by copying it into the temporary c and back again.

The instruction-selection phase has produced the instruction list of Program 11.8b. The interference graph for this function is shown at right.

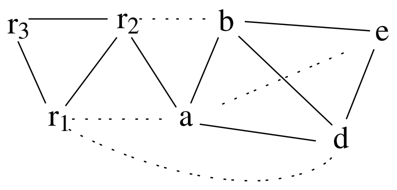


The register allocation proceeds as follows (with $K = 3$):

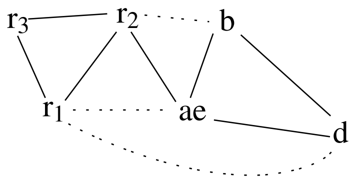
1. In this graph, there is no opportunity for *simplify* or *freeze* (because all the non-precolored nodes have degree $\geq K$). Any attempt to *coalesce* would produce a coalesced node adjacent to K or more significant-degree nodes. Therefore we must *spill* some node. We calculate spill priorities as follows:

Node		Uses+Defs outside loop		Uses+Defs within loop			Degree		Spill priority
a	(	2	$+ 10 \times$	0	)	/	4	=	0.50
b	(	1	$+ 10 \times$	1	)	/	4	=	2.75
c	(	2	$+ 10 \times$	0	)	/	6	=	0.33
d	(	2	$+ 10 \times$	2	)	/	4	=	5.50
e	(	1	$+ 10 \times$	3	)	/	3	=	10.33

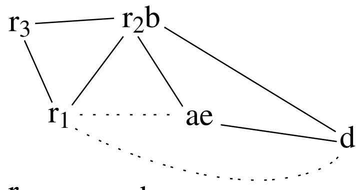
Node c has the lowest priority – it interferes with many other temporaries but is rarely used – so it should be spilled first. Spilling c, we obtain the graph at right.



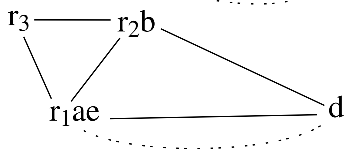
2. We can now coalesce a and e, since the resulting node will be adjacent to fewer than K significant-degree nodes (after coalescing, node d will be low-degree, though it is significant-degree right now). No other *simplify* or *coalesce* is possible now.



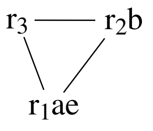
3. Now we could coalesce $ae\&r_1$ or coalesce $b\&r_2$. Let us do the latter.



4. We can now coalesce either $ae\&r_1$ or coalesce $d\&r_1$. Let us do the former.



5. We cannot now coalesce $r_1ae\&d$ because the move is *constrained:* The nodes r_1ae and d interfere. We must *simplify* d.

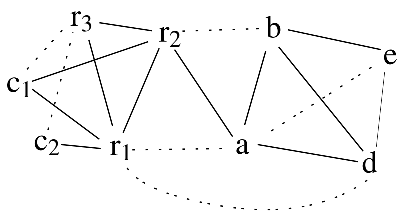


6. Now we have reached a graph with only precolored nodes, so we pop nodes from the stack and assign colors to them. First we pick d, which can be assigned color r_3. Nodes a, b, and e have already been assigned colors by coalescing. But node c, which was a *potential spill,* turns into an *actual spill* when it is popped from the stack, since no color can be found for it.

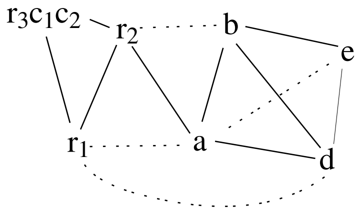
7. Since there was spilling in this round, we must rewrite the program to include spill instructions. For each use (or definition) of c, we make up a new temporary, and fetch (or store) it immediately beforehand (or afterward).

```
enter:  $c_1 \leftarrow r_3$
        $M[c_{loc}] \leftarrow c_1$
        $a \leftarrow r_1$
        $b \leftarrow r_2$
        $d \leftarrow 0$
        $e \leftarrow a$
loop:   $d \leftarrow d + b$
        $e \leftarrow e - 1$
        if $e > 0$ goto loop
        $r_1 \leftarrow d$
        $c_2 \leftarrow M[c_{loc}]$
        $r_3 \leftarrow c_2$
        return
```

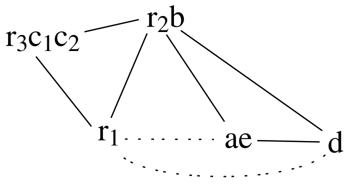
8. Now we build a new interference graph:



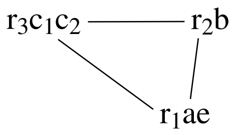
9. Graph-coloring proceeds as follows. We can immediately coalesce c_1&r_3 and then c_2&r_3.



10. Then, as before, we can coalesce a&e and then b&r_2.



11. As before, we can coalesce ae&r_1 and then simplify d.



12. Now we start popping from the stack: We select color r_3 for d, and this was the only node on the stack – all other nodes were coalesced or precolored. The coloring is shown at right.

Node	Color
a	r_1
b	r_2
c	r_3
d	r_3
e	r_1

13. Now we can rewrite the program using the register assignment.

```
enter: $r_3 \leftarrow r_3$
       $M[c_{\text{loc}}] \leftarrow r_3$
       $r_1 \leftarrow r_1$
       $r_2 \leftarrow r_2$
       $r_3 \leftarrow 0$
       $r_1 \leftarrow r_1$
loop:  $r_3 \leftarrow r_3 + r_2$
       $r_1 \leftarrow r_1 - 1$
       if $r_1 > 0$ goto loop
       $r_1 \leftarrow r_3$
       $r_3 \leftarrow M[c_{\text{loc}}]$
       $r_3 \leftarrow r_3$
       return
```

14. Finally, we can delete any move instruction whose source and destination are the same; these are the result of coalescing.

```
enter:  $M[c_{loc}] \leftarrow r_3$
        $r_3 \leftarrow 0$
loop:   $r_3 \leftarrow r_3 + r_2$
        $r_1 \leftarrow r_1 - 1$
        if $r_1 > 0$ goto loop
        $r_1 \leftarrow r_3$
        $r_3 \leftarrow M[c_{loc}]$
        return
```

The final program has only one uncoalesced move instruction.

11.4 GRAPH-COLORING IMPLEMENTATION

The graph-coloring algorithm needs to query the interference-graph data structure frequently. There are two kinds of queries:

1. Get all the nodes adjacent to node X; and
2. Tell if X and Y are adjacent.

An adjacency list (per node) can answer query 1 quickly, but not query 2 if the lists are long. A two-dimensional bit matrix indexed by node numbers can answer query 2 quickly, but not query 1. Therefore, we need both data structures to (redundantly) represent the interference graph. If the graph is very sparse, a hash table of integer pairs may be better than a bit matrix.

The adjacency lists of machine registers (precolored nodes) can be very large; because they're used in standard calling conventions, they interfere with any temporaries that happen to be live near *any* of the procedure-calls in the program. But we don't need to represent the adjacency list for a precolored node, because adjacency lists are used only in the *select* phase (which does not apply to precolored nodes) and in the Briggs coalescing test. To save space and time, we do not explicitly represent the adjacency lists of the machine registers. We coalesce an ordinary node a with a machine register r using the George coalescing test, which needs the adjacency list of a but not of r.

To test whether two ordinary (non-precolored) nodes can be coalesced, the algorithm shown here uses the Briggs coalescing test.

Associated with each move-related node is a count of the moves it is involved in. This count is easy to maintain and is used to test if a node is no longer move-related. Associated with all nodes is a count of the number of neighbors currently in the graph. This is used to determine whether a node is of significant degree during coalescing, and whether a node can be removed from the graph during simplification.

It is important to be able to quickly perform each *simplify* step (removing a low-degree non-move-related node), each *coalesce* step, and each *freeze* step. To do this, we maintain four work lists:

- Low-degree non-move-related nodes *(simplifyWorklist)*;
- Move instructions that might be coalesceable *(worklistMoves)*;
- Low-degree move-related nodes *(freezeWorklist)*;
- High-degree nodes *(spillWorklist)*.

Using these work lists, we avoid quadratic time blowup in finding coalesceable nodes.

DATA STRUCTURES

The algorithm maintains these data structures to keep track of graph nodes and move edges:

Node work lists, sets, and stacks. The following lists and sets are always *mutually disjoint* and every node is always in exactly one of the sets or lists.

precolored: machine registers, preassigned a color.
initial: temporary registers, not precolored and not yet processed.
simplifyWorklist: list of low-degree non-move-related nodes.
freezeWorklist: low-degree move-related nodes.
spillWorklist: high-degree nodes.
spilledNodes: nodes marked for spilling during this round; initially empty.
coalescedNodes: registers that have been coalesced; when $u \leftarrow v$ is coalesced, v is added to this set and u put back on some work list (or vice versa).
coloredNodes: nodes successfully colored.
selectStack: stack containing temporaries removed from the graph.

Since membership in these sets is often tested, the representation of each node should contain an enumeration value telling which set it is in. Since nodes must frequently be added to and removed from these sets, each set can be represented by a doubly linked list of nodes. Initially (on entry to Main), and on exiting RewriteProgram, only the sets *precolored* and *initial* are nonempty.

Move sets. There are five sets of move instructions, and every move is in exactly one of these sets (after Build through the end of Main).

coalescedMoves: moves that have been coalesced.
constrainedMoves: moves whose source and target interfere.
frozenMoves: moves that will no longer be considered for coalescing.
worklistMoves: moves enabled for possible coalescing.
activeMoves: moves not yet ready for coalescing.

Like the node work lists, the move sets should be implemented as doubly linked lists, with each move containing an enumeration value identifying which set it belongs to.

When a node x changes from significant to low-degree, the moves associated with its neighbors must be added to the move work list. Moves that were blocked with too many significant neighbors might now be enabled for coalescing.

Other data structures.

adjSet: the set of interference edges (u, v) in the graph; if $(u, v) \in$ adjSet, then $(v, u) \in$ adjSet.
adjList: adjacency list representation of the graph; for each non-precolored temporary u, adjList[u] is the set of nodes that interfere with u.
degree: an array containing the current degree of each node.
moveList: a mapping from a node to the list of moves it is associated with.
alias: when a move (u, v) has been coalesced, and v put in coalescedNodes, then alias$(v) = u$.
color: the color chosen by the algorithm for a node; for precolored nodes this is initialized to the given color.

INVARIANTS

After Build, the following invariants always hold:

Degree invariant.

$$\begin{aligned}&(u \in \text{simplifyWorklist} \cup \text{freezeWorklist} \cup \text{spillWorklist}) \quad \Rightarrow\\ &\quad \text{degree}(u) = |\text{adjList}(u) \cap (\text{precolored} \cup \text{simplifyWorklist}\\ &\qquad\qquad\qquad\qquad \cup \text{freezeWorklist} \cup \text{spillWorklist})|\end{aligned}$$

Simplify worklist invariant. Either u has been selected for spilling, or

$$\begin{aligned}&(u \in \text{simplifyWorklist}) \quad \Rightarrow\\ &\quad \text{degree}(u) < K \;\wedge\; \text{moveList}[u] \cap (\text{activeMoves} \cup \text{worklistMoves}) = \{\}\end{aligned}$$

Freeze worklist invariant.

$$\begin{aligned}&(u \in \text{freezeWorklist}) \quad \Rightarrow\\ &\quad \text{degree}(u) < K \;\wedge\; \text{moveList}[u] \cap (\text{activeMoves} \cup \text{worklistMoves}) \neq \{\}\end{aligned}$$

Spill worklist invariant.

$$(u \in \text{spillWorklist}) \quad \Rightarrow \quad \text{degree}(u) \geq K$$

PROGRAM CODE

The algorithm is invoked using the procedure *Main*, which loops (via tail recursion) until no spills are generated.

```
procedure Main()
    LivenessAnalysis()
    Build()
    MakeWorklist()
    repeat
        if simplifyWorklist $\neq$ {} then Simplify()
        else if worklistMoves $\neq$ {} then Coalesce()
        else if freezeWorklist $\neq$ {}  then Freeze()
        else if spillWorklist $\neq$ {} then SelectSpill()
    until simplifyWorklist = {}  $\wedge$ worklistMoves = {}
           $\wedge$ freezeWorklist = {}  $\wedge$ spillWorklist = {}
    AssignColors()
    if spilledNodes $\neq$ {} then
        RewriteProgram(spilledNodes)
        Main()
```

If *AssignColors* spills, then *RewriteProgram* allocates memory locations for the spilled temporaries and inserts store and fetch instructions to access them. These stores and fetches are to newly created temporaries (with tiny live ranges), so the main loop must be performed on the altered graph.

```
procedure Build ()
    forall $b \in$ blocks in program
        let live = liveOut($b$)
        forall $I \in$ instructions($b$) in reverse order
            if isMoveInstruction($I$) then
                live $\leftarrow$ live\use($I$)
                forall $n \in$ def($I$) $\cup$ use($I$)
                    moveList[$n$] $\leftarrow$ moveList[$n$] $\cup$ {$I$}
                worklistMoves $\leftarrow$ worklistMoves $\cup$ {$I$}
            live $\leftarrow$ live $\cup$ def($I$)
            forall $d \in$ def($I$)
                forall $l \in$ live
                    AddEdge($l$, $d$)
            live $\leftarrow$ use($I$) $\cup$ (live\def($I$))
```

Procedure *Build* constructs the interference graph (and bit matrix) using the results of static liveness analysis, and also initializes the *worklistMoves* to contain all the moves in the program.

```
procedure AddEdge($u$, $v$)
    if (($u$, $v$) $\notin$ adjSet) $\wedge$ ($u \neq v$) then
        adjSet $\leftarrow$ adjSet $\cup$ {($u$, $v$), ($v$, $u$)}
        if $u \notin$ precolored then
            adjList[$u$] $\leftarrow$ adjList[$u$] $\cup$ {$v$}
            degree[$u$] $\leftarrow$ degree[$u$] + 1
        if $v \notin$ precolored then
            adjList[$v$] $\leftarrow$ adjList[$v$] $\cup$ {$u$}
            degree[$v$] $\leftarrow$ degree[$v$] + 1

procedure MakeWorklist()
    forall $n \in$ initial
        initial $\leftarrow$ initial \ {$n$}
        if degree[$n$] $\geq K$ then
            spillWorklist $\leftarrow$ spillWorklist $\cup$ {n}
        else if MoveRelated($n$) then
            freezeWorklist $\leftarrow$ freezeWorklist $\cup$ {n}
        else
            simplifyWorklist $\leftarrow$ simplifyWorklist $\cup$ {n}

function Adjacent($n$)
    adjList[$n$] \ (selectStack $\cup$ coalescedNodes)

function NodeMoves ($n$)
    moveList[$n$] $\cap$ (activeMoves $\cup$ worklistMoves)

function MoveRelated($n$)
    NodeMoves($n$) $\neq$ {}

procedure Simplify()
    let $n \in$ simplifyWorklist
    simplifyWorklist $\leftarrow$ simplifyWorklist \ {$n$}
    push($n$, selectStack)
    forall m $\in$ Adjacent($n$)
        DecrementDegree($m$)
```

Removing a node from the graph involves decrementing the degree of its *current* neighbors. If the *degree* of a neighbor is already less than $K - 1$, then the neighbor must be move-related, and is not added to the `simplifyWork-list`. When the degree of a neighbor transitions from K to $K - 1$, moves associated with *its* neighbors may be enabled.

```
procedure DecrementDegree($m$)
    let $d$ = degree[$m$]
    degree[$m$] $\leftarrow$ d-1
    if $d = K$ then
        EnableMoves($\{m\} \cup$ Adjacent($m$))
        spillWorklist $\leftarrow$ spillWorklist $\setminus \{m\}$
        if MoveRelated($m$) then
            freezeWorklist $\leftarrow$ freezeWorklist $\cup \{m\}$
        else
            simplifyWorklist $\leftarrow$ simplifyWorklist $\cup \{m\}$

procedure EnableMoves(nodes)
    forall $n \in$ nodes
        forall $m \in$ NodeMoves($n$)
            if $m \in$ activeMoves then
                activeMoves $\leftarrow$ activeMoves $\setminus \{m\}$
                worklistMoves $\leftarrow$ worklistMoves $\cup \{m\}$
```

Only moves in the *worklistMoves* are considered in the coalesce phase. When a move is coalesced, it may no longer be move-related and can be added to the simplify work list by the procedure *AddWorkList*. *OK* implements the heuristic used for coalescing a precolored register. *Conservative* implements the conservative coalescing heuristic.

```
procedure AddWorkList(u)
    if ($u \notin$ precolored $\wedge$ not(MoveRelated($u$)) $\wedge$ degree[$u$] < K) then
        freezeWorklist $\leftarrow$ freezeWorklist $\setminus \{u\}$
        simplifyWorklist $\leftarrow$ simplifyWorklist $\cup \{u\}$

function OK($t$,$r$)
    degree[$t$] $< K \vee t \in$ precolored $\vee (t, r) \in$ adjSet

function Conservative(nodes)
    let $k = 0$
    forall $n \in$ nodes
        if degree[$n$] $\geq K$ then $k \leftarrow k + 1$
    return $(k < K)$
```

```
procedure Coalesce()
    let $m_{(=copy(x,y))} \in$ worklistMoves
    $x \leftarrow$ GetAlias($x$)
    $y \leftarrow$ GetAlias($y$)
    if $y \in$ precolored then
        let $(u, v) = (y, x)$
    else
        let $(u, v) = (x, y)$
    worklistMoves $\leftarrow$ worklistMoves $\setminus \{m\}$
    if $(u = v)$ then
        coalescedMoves $\leftarrow$ coalescedMoves $\cup \{m\}$
        AddWorkList($u$)
    else if $v \in$ precolored $\vee$ $(u, v) \in$ adjSet then
        constrainedMoves $\leftarrow$ constrainedMoves $\cup \{m\}$
        AddWorkList($u$)
        AddWorkList($v$)
    else if $u \in$ precolored $\wedge$ ($\forall t \in$ Adjacent($v$), OK($t$, $u$))
                $\vee$    $u \notin$ precolored $\wedge$
                    Conservative(Adjacent($u$) $\cup$ Adjacent($v$)) then
        coalescedMoves $\leftarrow$ coalescedMoves $\cup \{m\}$
        Combine($u$,$v$)
        AddWorkList($u$)
    else
        activeMoves $\leftarrow$ activeMoves $\cup \{m\}$

procedure Combine(u,v)
    if $v \in$ freezeWorklist then
        freezeWorklist $\leftarrow$ freezeWorklist $\setminus \{v\}$
    else
        spillWorklist $\leftarrow$ spillWorklist $\setminus \{v\}$
    coalescedNodes $\leftarrow$ coalescedNodes $\cup \{v\}$
    alias[$v$] $\leftarrow u$
    moveList[$u$] $\leftarrow$ moveList[$u$] $\cup$ moveList[$v$]
    EnableMoves($v$)
    forall $t \in$ Adjacent($v$)
        AddEdge($t$,$u$)
        DecrementDegree($t$)
    if degree[$u$] $\geq K \wedge u \in$ freezeWorkList
        freezeWorkList $\leftarrow$ freezeWorkList $\setminus \{u\}$
        spillWorkList $\leftarrow$ spillWorkList $\cup \{u\}$
```

```
**function** GetAlias ($n$)
    **if** $n \in$ coalescedNodes **then**
        GetAlias(alias[$n$])
    **else** $n$

**procedure** Freeze()
    **let** $u \in$ freezeWorklist
    freezeWorklist $\leftarrow$ freezeWorklist $\setminus \{u\}$
    simplifyWorklist $\leftarrow$ simplifyWorklist $\cup \{u\}$
    FreezeMoves($u$)

**procedure** FreezeMoves($u$)
    **forall** $m_{(=copy(x,y))} \in$ NodeMoves($u$)
        **if** GetAlias($y$)=GetAlias($u$) **then**
            $v \leftarrow$ GetAlias($x$)
        **else**
            $v \leftarrow$ GetAlias($y$)
        activeMoves $\leftarrow$ activeMoves $\setminus \{m\}$
        frozenMoves $\leftarrow$ frozenMoves $\cup \{m\}$
        **if** $v \in$ freezeWorklist $\wedge$ NodeMoves($v$) $= \{\}$ **then**
            freezeWorklist $\leftarrow$ freezeWorklist $\setminus \{v\}$
            simplifyWorklist $\leftarrow$ simplifyWorklist $\cup \{v\}$

**procedure** SelectSpill()
    **let** $m \in$ spillWorklist *selected using favorite heuristic*
        *Note: avoid choosing nodes that are the tiny live ranges*
        *resulting from the fetches of previously spilled registers*
    spillWorklist $\leftarrow$ spillWorklist $\setminus \{m\}$
    simplifyWorklist $\leftarrow$ simplifyWorklist $\cup \{m\}$
    FreezeMoves($m$)
```

```
procedure AssignColors()
    while SelectStack not empty
        let $n$ = pop(SelectStack)
        okColors $\leftarrow$ {0, ..., K-1}
        forall $w \in$ adjList[$n$]
            if GetAlias($w$) $\in$ (coloredNodes $\cup$ precolored) then
                okColors $\leftarrow$ okColors $\setminus$ {color[GetAlias($w$)]}
        if okColors = {}  then
            spilledNodes $\leftarrow$ spilledNodes $\cup$ {$n$}
        else
            coloredNodes $\leftarrow$ coloredNodes $\cup$ {$n$}
            let $c \in$ okColors
            color[$n$] $\leftarrow c$
    forall $n \in$ coalescedNodes
        color[$n$] $\leftarrow$ color[GetAlias($n$)]

procedure RewriteProgram()
    Allocate memory locations for each $v \in$ spilledNodes,
    Create a new temporary $v_i$ for each definition and each use,
    In the program (instructions), insert a store after each
    definition of a $v_i$, a fetch before each use of a $v_i$.
    Put all the $v_i$ into a set newTemps.
    spilledNodes $\leftarrow$ {}
    initial $\leftarrow$ coloredNodes $\cup$ coalescedNodes $\cup$ newTemps
    coloredNodes $\leftarrow$ {}
    coalescedNodes $\leftarrow$ {}
```

We show a variant of the algorithm in which all coalesces are discarded if the program must be rewritten to incorporate spill fetches and stores. For a faster algorithm, keep all the coalesces found before the first call to `Select-Spill` and rewrite the program to eliminate the coalesced move instructions and temporaries.

In principle, a heuristic could be used to select the freeze node; the *Freeze* shown above picks an arbitrary node from the freeze work list. But freezes are not common, and a selection heuristic is unlikely to make a significant difference.

```
function SimpleAlloc($t$)
    for each nontrivial tile $u$ that is a child of $t$
        SimpleAlloc($u$)
    for each nontrivial tile $u$ that is a child of $t$
        $n \leftarrow n - 1$
    $n \leftarrow n + 1$
    assign $r_n$ to hold the value at the root of $t$
```

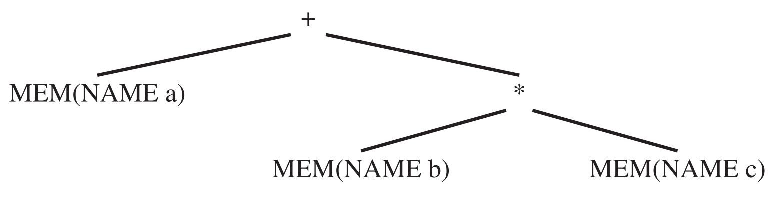
ALGORITHM 11.9. Simple register allocation on trees.

11.5 REGISTER ALLOCATION FOR TREES

Register allocation for expression trees is much simpler than for arbitrary flow graphs. We do not need global dataflow analysis or interference graphs. Suppose we have a tiled tree such as in Figure 9.2a. This tree has two *trivial* tiles, the TEMP nodes *fp* and *i*, which we assume are already in registers r_{fp} and r_i. We wish to label the roots of the nontrivial tiles (the ones corresponding to instructions, i.e., 2, 4, 5, 6, 8) with registers from the list $r_1, r_2, \ldots, r_k$.

Algorithm 11.9 traverses the tree in postorder, assigning a register to the root of each tile. With n initialized to zero, this algorithm applied to the root (tile 9) produces the allocation $\{\text{tile2} \mapsto r_1, \text{tile4} \mapsto r_2, \text{tile5} \mapsto r_2, \text{tile6} \mapsto r_1, \text{tile8} \mapsto r_2, \text{tile9} \mapsto r_1\}$. The algorithm can be combined with Maximal Munch, since both algorithms are doing the same bottom-up traversal.

But this algorithm will not always lead to an optimal allocation. Consider the following tree, where each tile is shown as a single node:



The SimpleAlloc function will use three registers for this expression (as shown at left on the next page), but by reordering the instructions we can do the computation using only two registers (as shown at right):

```
function Label($t$)
    for each tile $u$ that is a child of $t$
        Label($u$)
    if $t$ is trivial
        then $need[t] \leftarrow 0$
    else if $t$ has two children, $u_{\text{left}}$ and $u_{\text{right}}$
        then if $need[u_{\text{left}}] = need[u_{\text{right}}]$
                then $need[t] \leftarrow 1 + need[u_{\text{left}}]$
                else $need[t] \leftarrow \max(1, need[u_{\text{left}}], need[u_{\text{right}}])$
    else if $t$ has one child, $u$
        then $need[t] \leftarrow \max(1, need[u])$
    else if $t$ has no children
        then $need[t] \leftarrow 1$
```

ALGORITHM 11.10. Sethi-Ullman labeling algorithm.

$r_1 \leftarrow M[a]$	$r_1 \leftarrow M[b]$
$r_2 \leftarrow M[b]$	$r_2 \leftarrow M[c]$
$r_3 \leftarrow M[c]$	$r_1 \leftarrow r_1 \times r_2$
$r_2 \leftarrow r_2 \times r_3$	$r_2 \leftarrow M[a]$
$r_1 \leftarrow r_1 + r_2$	$r_1 \leftarrow r_2 + r_1$

Using dynamic programming, we can find the optimal ordering for the instructions. The idea is to label each tile with the number of registers it needs during its evaluation. Suppose a tile t has two nontrivial children u_{left} and u_{right} that require n and m registers, respectively, for their evaluation. If we evaluate u_{left} first, and hold its result in one register while we evaluate u_{right}, then we have needed $\max(n, 1+m)$ registers for the whole expression rooted at t. Conversely, if we evaluate u_{right} first, then we need $\max(1+n, m)$ registers. Clearly, if $n > m$, we should evaluate u_{left} first, and if $n < m$, we should evaluate u_{right} first. If $n = m$, we will need $n + 1$ registers no matter which subexpression is evaluated first.

Algorithm 11.10 labels each tile t with $need[t]$, the number of registers needed to evaluate the subtree rooted at t. It can be generalized to handle tiles with more than two children. Maximal Munch should identify – but not emit – the tiles, simultaneously with the labeling of Algorithm 11.10. The next pass emits *Assem* instructions for the tiles; wherever a tile has more than one

```
function SethiUllman($t, n$)
   if $t$ has two children, $u_{\text{left}}$ and $u_{\text{right}}$
      if $need[u_{\text{left}}] \geq K \;\wedge need[u_{\text{right}}] \geq K$
           SethiUllman($u_{\text{right}}$, 0)
           $n \leftarrow n - 1$
           spill: emit instruction to store $reg[u_{\text{right}}]$
           SethiUllman($u_{\text{left}}$, 0)
           unspill: $reg[u_{\text{right}}] \leftarrow$ "$r_1$"; emit instruction to fetch $reg[u_{\text{right}}]$
      else if $need[u_{\text{left}}] \geq need[u_{\text{right}}]$
           SethiUllman($u_{\text{left}}, n$)
           SethiUllman($u_{\text{right}}, n + 1$)
      else  $need[u_{\text{left}}] < need[u_{\text{right}}]$
           SethiUllman($u_{\text{right}}, n$)
           SethiUllman($u_{\text{left}}, n$)
      $reg[t] \leftarrow$ "$r_n$"
      emit OPER($instruction[t],\ reg[t]$, [ $reg[u_{\text{left}}],\ reg[u_{\text{right}}]$])
   else if $t$ has one child, $u$
      SethiUllman($u, n$)
      $reg[t] \leftarrow$ "$r_n$"
      emit OPER($instruction[t],\ reg[t],\ [reg[u]]$)
   else if $t$ is nontrivial but has no children
      $reg[t] \leftarrow$ "$r_n$"
      emit OPER($instruction[t],\ reg[t]$, [ ])
   else if $t$ is a trivial node TEMP($r_i$)
      $reg[t] \leftarrow$ "$r_i$"
```

ALGORITHM 11.11. Sethi-Ullman register allocation for trees.

child, the subtrees must be emitted in decreasing order of register *need.*

Algorithm 11.10 can profitably be used in a compiler that uses graph-coloring register allocation. Emitting the subtrees in decreasing order of *need* will minimize the number of simultaneously live temporaries and reduce the number of spills.

In a compiler without graph-coloring register allocation, Algorithm 11.10 is used as a pre-pass to Algorithm 11.11, which assigns registers as the trees are emitted and also handles spilling cleanly. This takes care of register allocation for the internal nodes of expression trees; allocating registers for explicit TEMPs of the *Tree* language would have to be done in some other way. In general, such a compiler would keep almost all program variables in the stack frame, so there would not be many of these explicit TEMPs to allocate.

PROGRAM GRAPH COLORING

Implement graph-coloring register allocation as two modules: `Color`, which does just the graph coloring itself, and `RegAlloc`, which manages spilling and calls upon `Color` as a subroutine. To keep things simple, do not implement spilling or coalescing; this simplifies the algorithm considerably.

```
package RegAlloc;

public class RegAlloc implements Temp.TempMap {
  public Assem.InstrList instrs;
  public String tempMap(Temp temp);
  public RegAlloc(Frame.Frame f, Assem.InstrList il);
}

class Color implements TempMap {
  public TempList spills();
  public String tempMap(Temp t);
  public Color(InterferenceGraph ig,
               TempMap initial,
               TempList registers);
}
```

Given an interference graph, an `initial` allocation (precoloring) of some temporaries imposed by calling conventions, and a list of colors (`registers`), `color` produces an extension of the `initial` allocation. The resulting allocation assigns all temps used in the flow graph, making use of registers from the `registers` list.

The `initial` allocation is the `frame` (which implements a `TempMap` describing precolored temporaries); the `registers` argument is just the list of all machine registers, `Frame.registers` (see page 251). The registers in the `initial` allocation can also appear in the `registers` argument to `Color`, since it's OK to use them to color other nodes as well.

The result of `Color` is a `TempMap` (that is, `Color implements TempMap`) describing the register allocation, along with a list of spills. The result of `RegAlloc` – if there were no spills – is an identical `TempMap`, which can be used in final assembly-code emission as an argument to `Assem.format`.

A better `Color` interface would have a `spillCost` argument that specifies the spilling cost of each temporary. This can be just the number of uses and defs, or better yet, uses and defs weighted by occurrence in loops and nested loops. A naive `spillCost` that just returns 1 for every temporary will also work.

A simple implementation of the coloring algorithm without coalescing requires only one work list: the `simplifyWorklist`, which contains all non-precolored, nonsimplified nodes of degree less than K. Obviously, no `freezeWorklist` is necessary. No `spillWorklist` is necessary either, if we are willing to look through all the nodes in the original graph for a spill candidate every time the `simplifyWorklist` becomes empty.

With only a `simplifyWorklist`, the doubly linked representation is not necessary: This work list can be implemented as a singly linked list or a stack, since it is never accessed "in the middle."

ADVANCED PROJECT: SPILLING

Implement spilling, so that no matter how many parameters and locals a Mini-Java program has, you can still compile it.

ADVANCED PROJECT: COALESCING

Implement coalescing, to eliminate practically all the MOVE instructions from the program.

FURTHER READING

Kempe [1879] invented the simplification algorithm that colors graphs by removing vertices of degree $< K$. Chaitin [1982] formulated register allocation as a graph-coloring problem – using Kempe's algorithm to color the graph – and performed copy propagation by (nonconservatively) coalescing noninterfering move-related nodes before coloring the graph. Briggs et al. [1994] improved the algorithm with the idea of optimistic spilling, and also avoided introducing spills by using the conservative coalescing heuristic before coloring the graph. George and Appel [1996] found that there are more opportunities for coalescing if conservative coalescing is done during simplification instead of beforehand, and developed the work-list algorithm presented in this chapter.

Ershov [1958] developed the algorithm for optimal register allocation on expression trees; Sethi and Ullman [1970] generalized this algorithm and showed how it should handle spills.

EXERCISES

11.1 The following program has been compiled for a machine with three registers r_1, r_2, r_3; r_1 and r_2 are (caller-save) argument registers and r_3 is a callee-save register. Construct the interference graph and show the steps of the register allocation process in detail, as on pages 229–232. When you coalesce two nodes, say whether you are using the Briggs or George criterion.

Hint: When two nodes are connected by an interference edge *and* a move edge, you may delete the move edge; this is called *constrain* and is accomplished by the first **else if** clause of procedure *Coalesce*.

```
$f$ :   $c \leftarrow r_3$
        $p \leftarrow r_1$
        if $p = 0$ goto $L_1$
        $r_1 \leftarrow M[p]$
        call $f$              (uses $r_1$, defines $r_1, r_2$)
        $s \leftarrow r_1$
        $r_1 \leftarrow M[p+4]$
        call $f$              (uses $r_1$, defines $r_1, r_2$)
        $t \leftarrow r_1$
        $u \leftarrow s + t$
        goto $L_2$
$L_1$ : $u \leftarrow 1$
$L_2$ : $r_1 \leftarrow u$
        $r_3 \leftarrow c$
        return                (uses $r_1, r_3$)
```

11.2 The table below represents a register-interference graph. Nodes 1–6 are precolored (with colors 1–6), and nodes A–H are ordinary (non-precolored). Every pair of precolored nodes interferes, and each ordinary node interferes with nodes where there is an x in the table.

	1	2	3	4	5	6	A	B	C	D	E	F	G	H
A	x	x	x	x	x	x								
B	x		x	x	x	x								
C			x	x	x	x				x	x	x	x	x
D	x		x	x	x				x		x	x	x	x
E	x		x		x	x			x	x		x	x	x
F	x		x	x		x			x	x	x		x	x
G									x	x	x	x		
H	x			x	x	x			x	x	x	x		

The following pairs of nodes are related by MOVE instructions:

$(A, 3)$ $(H, 3)$ $(G, 3)$ $(B, 2)$ $(C, 1)$ $(D, 6)$ $(E, 4)$ $(F, 5)$

Assume that register allocation must be done for an 8-register machine.

a. Ignoring the MOVE instructions, and without using the *coalesce* heuristic, color this graph using *simplify* and *spill*. Record the sequence (stack) of *simplify* and *potential-spill* decisions, show which potential spills become actual spills, and show the coloring that results.

b. Color this graph using coalescing. Record the sequence of *simplify*, *coalesce*, *freeze*, and *spill* decisions. Identify each *coalesce* as Briggs- or George-style. Show how many MOVE instructions remain.

*c. Another coalescing heuristic is *biased coloring*. Instead of using a *conservative coalescing* heuristic during simplification, run the *simplify-spill* part of the algorithm as in part (a), but in the *select* part of the algorithm,

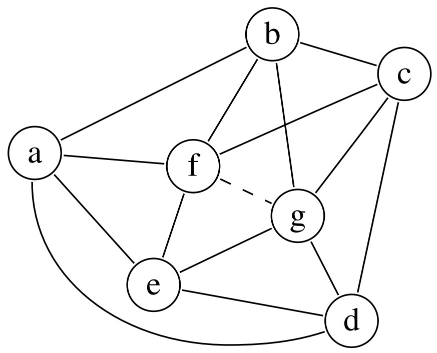
i. When selecting a color for node X that is move-related to node Y, when a color for Y has already been selected, use the same color if possible (to eliminate the MOVE).

ii. When selecting a color for node X that is move-related to node Y, when a color for Y has not yet been selected, use a color that is *not* the same as the color of any of Y's neighbors (to increase the chance of heuristic (i) working when Y is colored).

Conservative coalescing (in the *simplify* phase) has been found to be more effective than biased coloring, in general; but it might not be on this particular graph. Since the two coalescing algorithms are used in different phases, they can both be used in the same register allocator.

*d. Use both conservative coalescing and biased coloring in allocating registers. Show where biased coloring helps make the right decisions.

11.3 *Conservative coalescing* is so called because it will not introduce any (potential) spills. But can it avoid spills? Consider this graph, where the solid edges represent interferences and the dashed edge represents a MOVE:



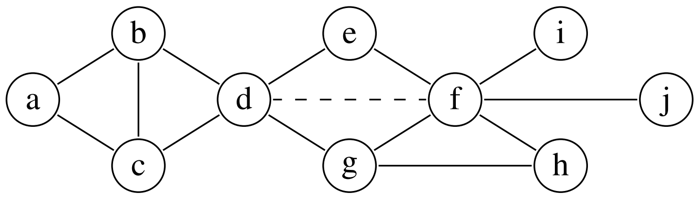
a. 4-color the graph without coalescing. Show the *select*-stack, indicating the order in which you removed nodes. Is there a potential spill? Is there an actual spill?

b. 4-color the graph with conservative coalescing. Did you use the Briggs or George criterion? Is there a potential spill? Is there an actual spill?

11.4 It has been proposed that the conservative coalescing heuristic could be simplified. In testing whether MOVE(a, b) can be coalesced, instead of asking whether the combined node ab is adjacent to $< K$ nodes of significant degree, we could simply test whether ab is adjacent to $< K$ nodes of any degree. The theory is that if ab is adjacent to many low-degree nodes, they will be removed by simplification anyway.

a. Show that this kind of coalescing cannot create any new potential spills.

b. Demonstrate the algorithm on this graph (with $K = 3$):



*c. Show that this test is less effective than standard conservative coalescing. **Hint:** Use the graph of Exercise 11.3, with $K = 4$.