Contents of Lecture 4

We will continue with SSA Form when you have done Lab 2

- Live Variables Analysis
- Graph Coloring Register Allocation
- Interprocedural Register Allocation
- Research from IBM Research Tokyo 2010: Coloring-based coalescing

Live Variables Analysis

```
int h(int a, int b)
{
        int     c;

S1:     c = a + b;

S2:     if (c < 0)
                return c * 44;

S3:     a = b - 14;

        return -a;
}
```

- A variable x is **live** at a point p (instruction) if it may be used in the future without being assigned to.
- a is live from the function start and up to and including the add, and then after S_3 and up to and including the negation.
- b is live from the start and up to and including the subtraction.
- c is live from S_1 and up to and including the multiplication.

Live Variables Analysis for Register Allocation

- Live Variables Analysis is used for different purposes.
- For example an assignment to a local variable which is not used in the future can be removed.
- This is called dead code elimination (DCE) and DCE based on live variables analysis was used before SSA Form, which introduced a better form of DCE (which you will implement in a project).
- We will use live variables analysis for register allocation.
- Two variables live at the same point in the program are said to **interfere** and cannot be allocated the same register.

Uses and Kills

- Live variables analysis is performed in a local and a global analysis.
- In the local analysis, each basic block (vertex) is inspected with the purpose of finding which variables are first used or first defined (assigned to).
- The information that a variable is live propagates backwards in the control flow graph (CFG) from a **use** and to its definition.
- The propagation of a use stops at a definition. The use in `a + 13` is **killed** by the definition `a = 14`.

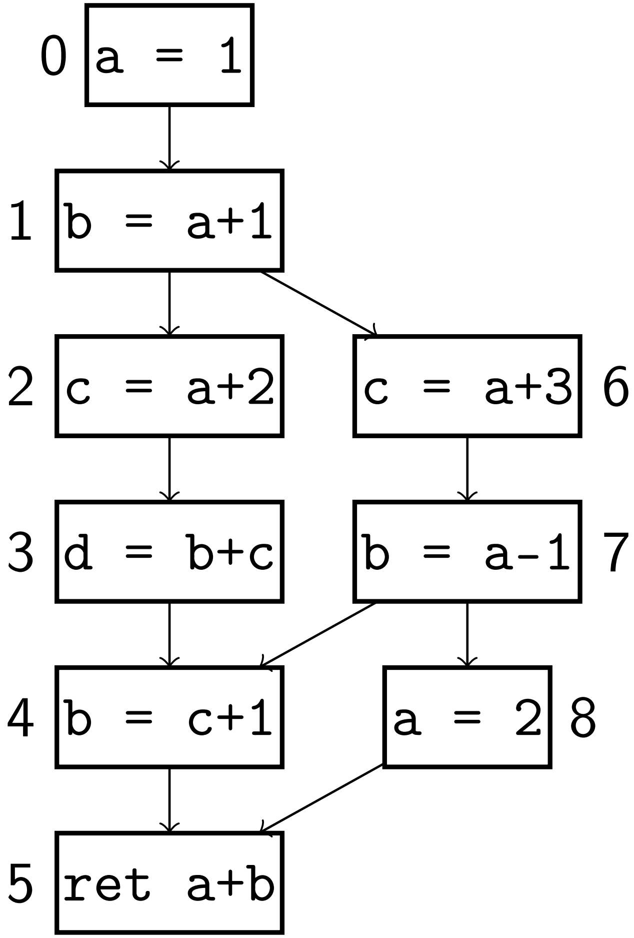
  ```
  a = 44;
  b = a + 11;
  a = 14;
  b = a + 13;
  ```

- In the global analysis the local information is combined to produce the complete view.
- Sometimes gen/kill is used instead of use/def.

Local Analysis

```
procedure local_live_analysis
    for each vertex w do
        for each stmt s do /* forward direction */
            for each used variable x of s do
                if (x ∉ def(w))
                    add x to use(w)
            for each defined variable x of s do
                if (x ∉ use(w))
                    add x to def(w)
end
```

Local Analysis Example



vertex	use	def
0	$\emptyset$	$\{a\}$
1	$\{a\}$	$\{b\}$
2	$\{a\}$	$\{c\}$
3	$\{b, c\}$	$\{d\}$
4	$\{c\}$	$\{b\}$
5	$\{a, b\}$	$\emptyset$
6	$\{a\}$	$\{c\}$
7	$\{a\}$	$\{b\}$
8	$\emptyset$	$\{a\}$

Global Analysis

procedure *global_live_analysis*
    $change \leftarrow$ true
    while ($change$) **do**
        $change \leftarrow$ false
        for each vertex w **do**

$$out(w) \leftarrow \bigcup_{s \in succ(w)} in(s)$$

$$old \leftarrow in(w)$$

$$in(w) \leftarrow use(w) \cup (out(w) - def(w))$$

            if ($old \neq in(w)$)
                $change \leftarrow$ true

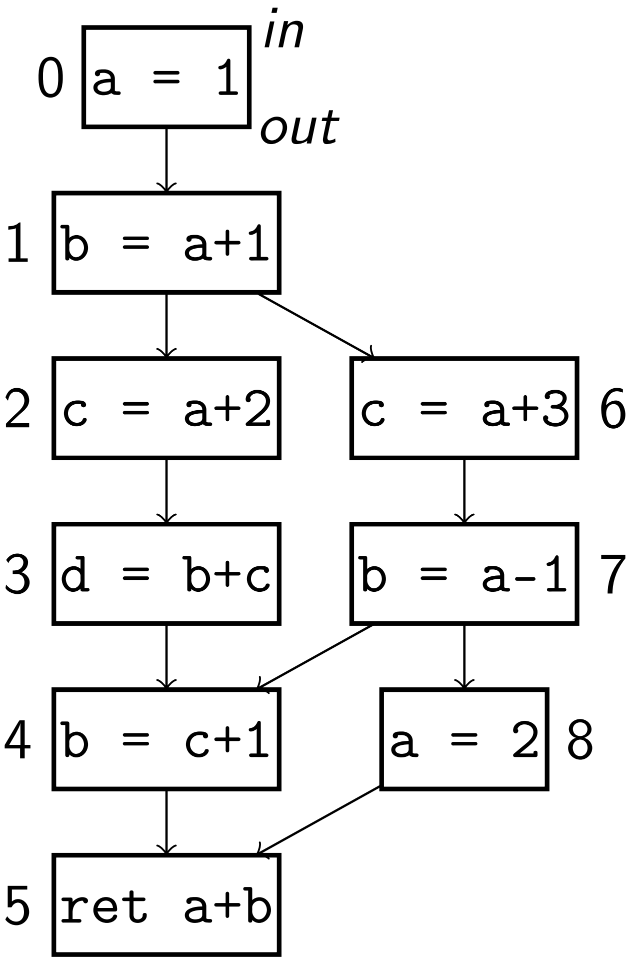
end

Dataflow Backwards: Postorder

- Since data flows backward we want to have processed the successors of a vertex w before we process w.

procedure *find_post_order(w)*
    $visited$(w) $\leftarrow$ true
    for each $s \in succ(w)$ **do**
        if (**not** $visited$(s))
            find_post_order(s)
    $array[num] \leftarrow$ w
    $num \leftarrow num + 1$
end

Global Analysis Example: Iteration 1

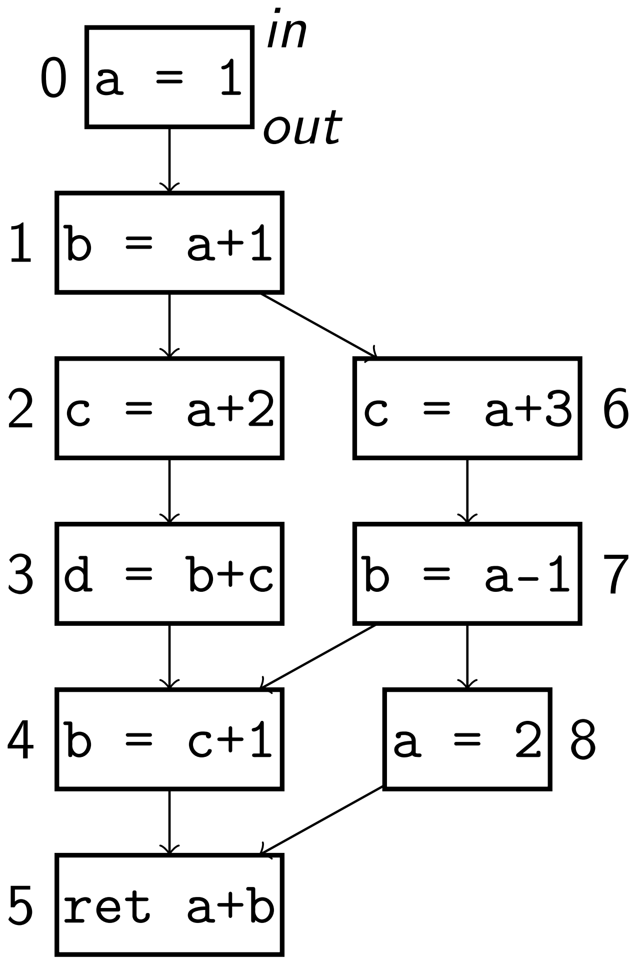


$$out(w) \leftarrow \bigcup_{s \in succ(w)} in(s)$$

$$in(w) \leftarrow use(w) \cup (out(w) - def(w))$$

vertex	**use**	**def**	**out**	**in**
5	$\{a, b\}$	$\emptyset$	$\emptyset$	$\{a, b\}$
4	$\emptyset$	$\{b\}$	$\{a, b\}$	$\{a, c\}$
3	$\{b, c\}$	$\{d\}$	$\{a, c\}$	$\{a, b, c\}$
2	$\{a\}$	$\{c\}$	$\{a, b, c\}$	$\{a, b\}$
8	$\emptyset$	$\{a\}$	$\{a, b\}$	$\{b\}$
7	$\{a\}$	$\{b\}$	$\{a, b, c\}$	$\{a, c\}$
6	$\{a\}$	$\{c\}$	$\{a, c\}$	$\{a\}$
1	$\{a\}$	$\{b\}$	$\{a, b\}$	$\{a\}$
0	$\emptyset$	$\{a\}$	$\{a\}$	$\emptyset$

Global Analysis Example: Iteration 2



$$out(w) \leftarrow \bigcup_{s \in succ(w)} in(s)$$

$$in(w) \leftarrow use(w) \cup (out(w) - def(w))$$

vertex	use	def	out	in
5	$\{a, b\}$	$\emptyset$	$\emptyset$	$\{a, b\}$
4	$\emptyset$	$\{b\}$	$\{a, b\}$	$\{a, c\}$
3	$\{b, c\}$	$\{d\}$	$\{a, c\}$	$\{a, b, c\}$
2	$\{a\}$	$\{c\}$	$\{a, b, c\}$	$\{a, b\}$
8	$\emptyset$	$\{a\}$	$\{a, b\}$	$\{b\}$
7	$\{a\}$	$\{b\}$	$\{a, b, c\}$	$\{a, c\}$
6	$\{a\}$	$\{c\}$	$\{a, c\}$	$\{a\}$
1	$\{a\}$	$\{b\}$	$\{a, b\}$	$\{a\}$
0	$\emptyset$	$\{a\}$	$\{a\}$	$\emptyset$

Constructing the Interference Graph

- Each vertex is analyzed again and the set of **live** variables in a vertex is maintained.
- The **live** set is initialized to $w(out)$ when vertex w is inspected.
- When a variable x is defined, an edge (x, y), $\forall y \in live - \{x\}$ is added to the interference graph (if it's not already there).
- The instructions in w are inspected in reverse order.
- After an instruction i has been inspected, the live set becomes:
$$live = use(i) \cup (live - \{def(i)\})$$
- Our description assumes there is at most one destination operand in an instruction.

An Example

```
a = 1
b = a + 2
c = a - b
d = c
e = d + 1
f = d - e
```

↓

```
ret c + f
```

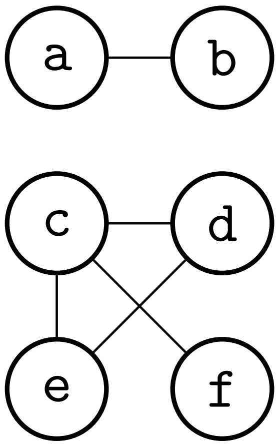
- Which variables cannot use the same register?
- How many registers are needed?

The Interference Graph

```
a = 1
b = a + 2
c = a - b
d = c
e = d + 1
f = d - e
```

↓

```
ret c + f
```



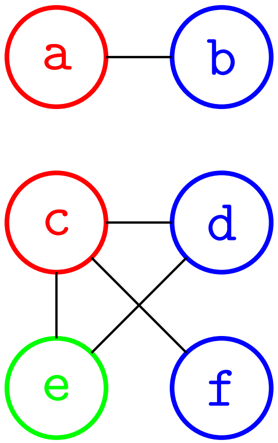
$live = use(i) \cup (live - \{def(i)\})$

1. Initially $live = out = \{c, f\}$.
2. $def(f)$: add edge (c, f).
   $live = \{c, d, e\}$.
3. $def(e)$: add edges $(e, c), (e, d)$.
   $live = \{c, d\}$.
4. $def(d)$: add edge (d, c).
   $live = \{c\}$.
5. $def(c)$: no new edge.
   $live = \{a, b\}$.
6. $def(b)$: add edge (a, b).
   $live = \{a\}$.
7. $def(a)$: no new edge. $live = \emptyset$.

Coloring the Interference Graph

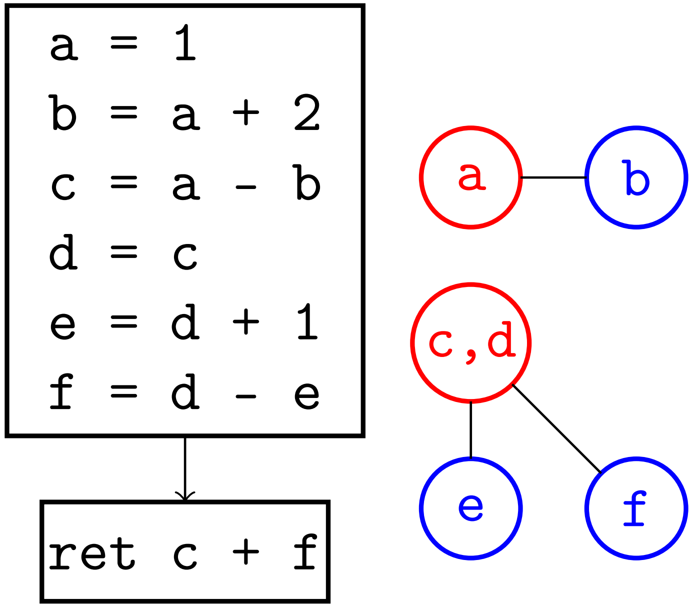
```
a = 1
b = a + 2
c = a - b
d = c
e = d + 1
f = d - e
```

```
ret c + f
```



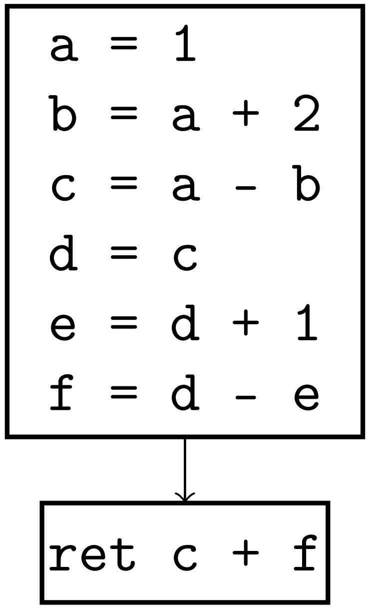
- This interference graph needs three colors.
- Can we use fewer colors?

Register Coalescing



- *c* and *d* have the same value so they can use the same register!
- It is done using a technique called **register coalescing**.
- Register coalescing is an example of **node merging**.
- Register coalescing needs a minor modification to the construction of the interference graph.

Constructing the Interference Graph for Register Coalescing



- Consider a copy instruction $x = y$.
- The interference graph is called the IG.
- Recall: an edge (x, y) is added to the IG between the defined variable x and each $y \in live, x \neq y, (x, y) \notin IG$.
- When $y \in live$ we will add (x, y) to IG.
- By temporarily removing y from *live* and noting that these variables might be merged to a single variable we prepare for register coalescing.
- The removed variable is added back after the instruction is processed:

$$live = use(i) \cup (live - \{def(i)\})$$

Summary so far

- Copy instructions are treated in a special way.
- Variables live at the same time cannot be allocated the same register and an edge in the interference graph *IG* is added between them.
- Given an interference graph, we want to color it with as few colors as possible.
- However, we are not always looking for the optimal solution with fewest colors since that solution may use more colors than there are registers.
- Furthermore, since graph coloring is NP-complete we use an approximation.
- The algorithm described next was invented by Greg Chaitin in 1980 for the IBM 801 project.
- A variable is called a **live range**.

Simplifying the Interference Graph

- Consider an interference graph IG and a number of available colors K.
- Assume the IG can be colored with K colors and there is a node $v \in IG$ with fewer than K neighbors.
- Since v has fewer than K neighbors there must be at least one unused color left for v.
- Therefore we can remove v from the IG without affecting the colorability of IG.
- We remove v from IG and push v on a stack.
- Then we proceed looking for a new node with fewer than K neighbors.
- Assume the original IG was colorable and all it's nodes have been pushed on the stack.
- Then each node is popped and re-inserted into IG and given a color which no neighbor has.

Spilling

- The number of neighbors of a node v is denoted its **degree**, or $deg(v)$.
- When there is no node with $deg(v) < K$ a variable is selected for spilling.
- Spilling means that a variable will reside in memory instead of being allocated a register.
- Through spilling the IG eventually will become empty, obviously.
- Heuristics are used to decide which variable (i.e. node) to spill.
- The expected number of memory accesses removed by allocating a variable is calculated, and this count is typically divided by a ”size” of the node.
- By size is meant the number of vertices or instructions that the register would be reserved in for that variable, and hence cannot be used for any other variable.

Rewriting the Program after Spills

```
a = b + c;

...

d = a + c;

------------

t1 = b + c;
a = t1;

...

t2 = a;
d = t2 + c;
```

- On a RISC machine where operands cannot be in memory a new tiny live range is created at each original memory access of the spilled variable.
- These tiny live ranges should never be spilled.
- The rewriting is done after all nodes have been removed from the interference graph.
- If there was spilling the algorithm is re-executed.
- Eventually it will terminate and three iteration almost always suffice.

Overview of the Algorithm

1. Perform live variable analysis.
2. Construct the interference graph.
3. Either simplify the interference graph by removing a node and push it on a stack, or spill a node to memory, until the interference graph is empty.
4. If there were any spill, create tiny live ranges to load and store the spilled variables, and goto 1.
5. If there were no spills, then assign colors to the nodes when popping them from the stack, and then change the program to use registers instead of variables.

More Details About Coalescing

- Two nodes can be coalesced into one if they do not interfere.
- By removing the source operand temporarily from the live set, the copy statement does not add an edge between the source and destination operands.
- However, in the following code there will be an edge between c and d.

```
c = a - b
d = c
e = d + 1
c = d + 2
g = d + 3
```

- With SSA Form, however, the assignments to c would be to two different variables so that problem is avoided.

Risks with Coalescing

- Assume two live ranges u and v are coalesced into uv.
- The new live range will have the union of the neighbors of u and v.
- If u and v have the same neighbors then its no problem.
- However, if $deg(u) < K \wedge deg(v) < K \wedge deg(uv) \geq K$ then the IG can become incolorable due the coalescing.
- Therefore, heuristics of when to coalesce have been developed.
- Chaitin's original algorithm coalesced everything it could.

Conservative Coalescing

- A node u has **significant degree** if $deg(u) \geq K$.
- Conservative coalescing, introduced by Briggs, does not merge nodes if the resulting node uv has K or more neighbors of significant degree.
- All neighbors without significant degree will be removed during simplification.
- All neighbors with significant degree might remain and if uv has K or more such neighbors, the IG cannot be colored.
- This approach is conservative due to that it might have been possible to coalesce u and v and still color the IG since some neighbors might have been allocated the same color, and leaving a color for uv.

Iterated Register Coalescing

- Both Chaitin and Briggs performed coalescing before simplification.
- In **Iterated Register Coalescing** by George and Appel, the coalescing is performed as a part of the main loop:
- In the main loop, the following are attempted in sequence:
  1. Simplify, but no ”move”-related nodes — they wait for coalescing.
  2. Coalescing
  3. Freeze — move-related nodes that could not be coalesced no longer are considered as move-related.
  4. Spilling

More Details

- The interference graph is represented in two ways. Both as a **bit matrix**, and as **adjacency lists**.
- Function call and return conventions introduces **precolored live ranges**. For example, the first integer parameter is passed in register R3 on Power machines.
- With coalescing this is simply solved by introducing copy statements and when possible merging a variable passed as a parameter with the precolored node. This way the variable gets the correct register when possible.
- In **Optimistic coloring** (Briggs) a variable can be removed from the *IG* and pushed even if it has significant degree. Whether it should be spilled or not is determined when it is re-inserted into *IG* after being popped. If there is no available color then it's spilled.

Caller vs Callee Save Registers

- The Application Binary Interface (ABI) specifies for UNIX which registers the caller and the callee are responsible for saving and restoring.
- An Example: General Purpose Registers (ie integer) on Power:
  Zero for some instructions: R0
  Stack pointer: R1
  Thread pointer: R2
  Caller-saved: R3..R12
  Callee-saved: R13..R31
- If a variable allocated to a caller-save register is live across a function call, it must be saved before the call and restored after it.
- A function may modify the callee-save registers but must save and restore them.

Neither is optimal

- If all registers are caller-save, then typically some unnecessary saving will take place unless the called function modifies all registers
- If all registers are callee-save, then it's likely the called function preserves a register which the caller does not care about
- When a color is to be selected for a variable, if it's live across function calls, it's preferable to use a callee-save register and hope that the called function will not use that register

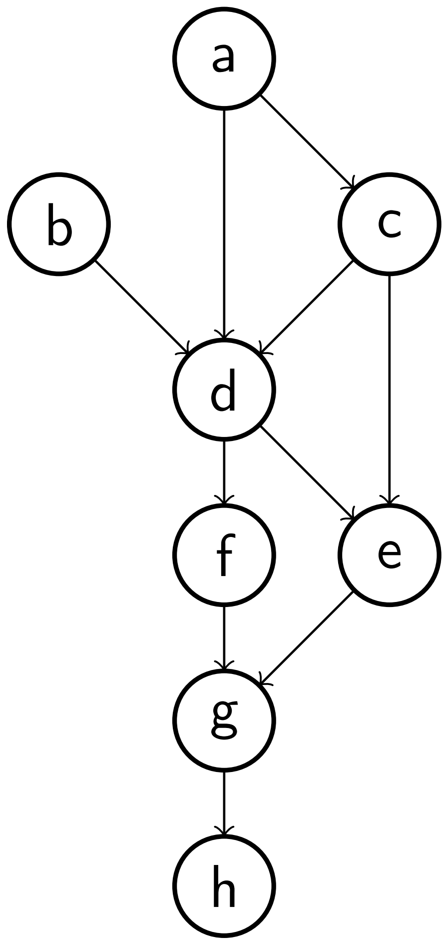
Interprocedural Register Allocation

- Intraprocedural register allocation can also assign global variables to registers but only after copying to a temporary and then saving them in memory before a function call or its own return (if the variable was modified).
- Interprocedural register allocation aims at three things:
  - Allocate global variables in registers in a region of several functions.
  - Make better choices with respect to caller/callee save registers.
  - Avoid doing callee-save and restore unless necessary.
- Interprocedural register allocation is most effective if the whole program can be analyzed.

Call Graph

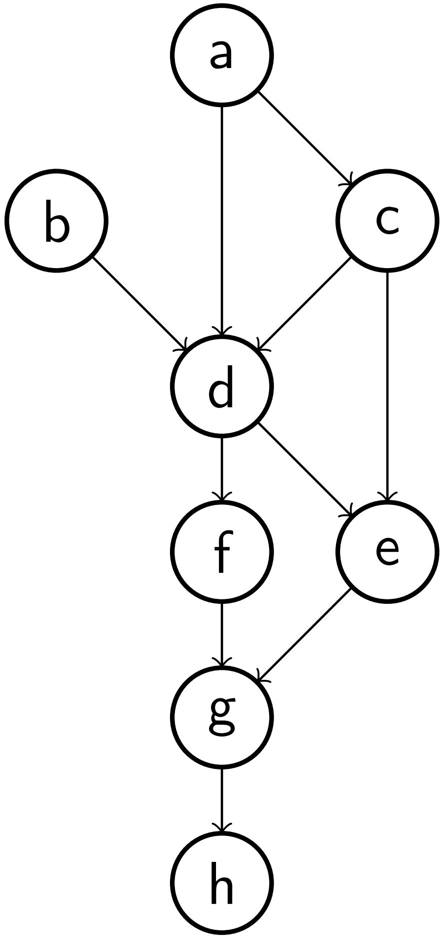
- The call graph has functions as nodes and function calls as edges.
- The linker (or a similar module) can construct the call graph after it has found all files needed for an application.

Global Variable Register Allocation



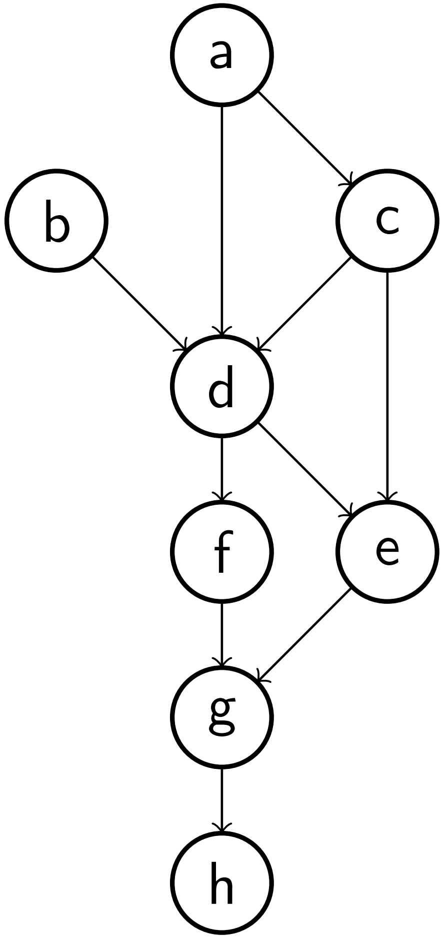
- In a first step each function f is analyzed to find which and how frequently global variables are accessed in f.
- In a second step the call graph is constructed and sets of functions, called **webs**, for each variable is constructed.
- A web is a subgraph of the call graph in which a global variable may be allocated a register.
- Let x be used in all functions except b, f, h.
- The web for x will be $\{a, b, c, d, e, f, g\}$.

Using the Webs



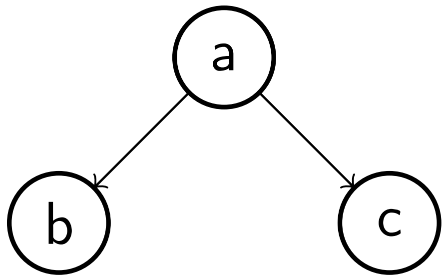
- A global variable can have many webs.
- When two webs for different variables have nodes in common, they interfere.
- The global variable register allocator **estimates** how useful it will be to allocate a certain web to a callee-save register.
- The webs compete and some are given a register.
- The program is then rewritten with some webs "precolored".
- Since a callee-save register is used, the function *h* will not destroy the global variable.

Modifying the Program



- Some nodes in a web are called entry nodes, and they are *a* and *b* in our example.
- The variable must be read from memory in the entry nodes.
- Note that in our example, the variable was not used in *b* but *b* must be part of the web and *b* must read the variable from memory.
- In addition to being responsible for reading the variable from memory to the allocated register, the entry nodes are also responsible for writing the value to memory if needed.

Moving Saves and Restores



- Assume *b* and *c* are called frequently.
- Instead of letting them do the callee-save and restore, it can be done in *a*.
- This can improve performance.

Live-range splitting

- Instead of spilling, it is sometimes useful to split a live range
- In the 1990's there were attempts to split to a large extent and then hoping for coalescing to nicely merge live ranges when suitable
- This did not work out very well
- Research by Cooper et al. found it is better to split a live range at the moment you find it should be spilled.
- Their approach is based on a separate graph, the **containment graph** constructed when constructing the interference graph

```
u =
while (...) {
        v = ...     // the live range of u contains
        ... v       // the live range of v and
}                   // u can be split around v
... u
```

Coloring-based coalescing

- A new approach to deciding what to coalesce was published by IBM Research Tokyo in 2010
- The Chaitin algorithm is used for this decision before the real coloring
- A new set of colors is used, called **extended colors**
- These extended colors are only used to decide whether two live ranges should be coalesced
- The normal colors are called **real colors**

Actions at a pop of live range u

- If there is a real color c, unused by neighbors of u, but used by a live range v which is move-related to u, then assign c to u
- Otherwise if there is a real color c, unused by neighbors of u, then assign c to u
- Otherwise, if there is an extended color c, unused by neighbors of u, but used by a live range v which is move-related to u, then assign c to u
- Otherwise assign a new extended color to u.

Actions after assignment

- When all live ranges have been assigned a real or extended color, move-related live ranges with the same color (real or extended) are coalesced
- This process can be repeated
- If extended colors where used during the final run of the algorithm, spilling or splitting is used.
- The effect was 1 % performance improvement on a machine with 16 integer and 16 floating point registers — good!