Register Allocation

Register Allocation can be split into two phases:

- Liveness analysis
- Graph coloring

Last week it was proved that liveness analysis can be done in polynomial time. But what about graph coloring?

Graph Coloring

The problem statement:
Given a Graph $G = (V, E)$, where V represents the set of vertices and E represents the set of edges, such that $E \subseteq V \times V$, the goal is to give a color to each vertex such that neighbors have different colors.

- From an algorithms class we know that this problem is **NP-complete**. In other words, unsolvable in polynomial time.

Approximate Graph Coloring

There exists an algorithm called *Linear Scan Register Allocation* that does a good job of graph coloring in linear time in the size of graph.

Note: It does a *good* job, not an *optimal* job. As in, this algorithm returns a suboptimal solution in exchange for fast execution.

Example

Let's start with a basic program with liveness analysis done:

```
                | a  b  c |
a = 1           | *       |   (1)
b = 2           | |  *    |   (2)
c = a + 3       | 0  |  * |   (3)
print(b + c)    |    0  0 |   (4)
```

Linear Scan Register Allocation is a greedy algorithm! There is no backtracking.

- (1) Assign r_1 to a.
- (2) Assign the next free register, r_2 to b.
- (3) Deallocate r_1 since a is no longer used, and assign the next free register, now r_1, to c.
- (4) After the print statement, deallocate r_1 and r_2 since they are no longer used.

The steps would be:

1. Order the intervals after start points.
2. Process the intervals from left to right.
3. The only situation where we change our mind about an assignment is when we 'spill' a variable to memory. (e.g., write it to the stack instead of register).
    - Pick the live range that extends the furthest into the future (see next subsection.)

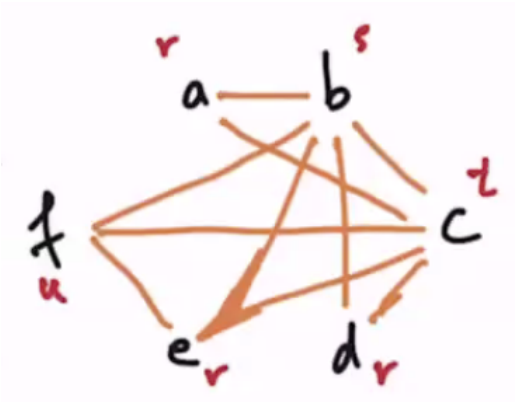
Live Ranges Example

Suppose we have 6 variables but only 4 registers.

```
                | a  b  c  d  e  f |
a = 1           | *                |  (1)
b = 10          | |  *             |  (2)
c = a + 9       | |  |  *          |  (3)
d = a + c       | 0  |  |  *       |  (4)
e = c + d       |    |  |  0  *    |  (5)
f = b + 8       |    |  |     |  * |  (6)
c = f + e       |    |  |     |  | |  (7)
f = e + c       |    |  |     0  | |  (8)
b = c + 5       |    |  0        | |  (9)
return b + f    |    0           0 |  (10)
```

- (1) Assign register r_1 to a.
- (2) Assign register r_2 to b.
- (3) Assign register r_3 to c.
- (4) Assign register r_1 to d. (deallocate a)
- (5) Assign register r_1 to e. (deallocate d)
- (6) Assign register r_4 to f

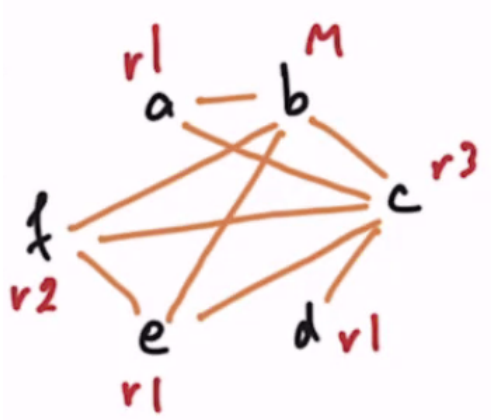
Below is the interference graph for this system, where r, s, t, and u correspond to r_1, r_2, r_3, and r_4.



What if in this case, we only have three registers?
Then, instead of allocating r_2 to b, we would store b in memory. The execution would look like:

- (1) Assign register r_1 to a.
- (2) Spill b to memory (M).
- (3) Assign r_3 to c.
- (4) Assign r_1 to d.
- (5) Assign r_1 to e.
- (6) Assign r_2 to f.

The interference graph would now look like:



Similarly, if we only had two registers, we would send f to memory too, since it is the variable that lasts the longest into the future (other than b, which is already in memory).
Note that when we are storing variables in memory, we **always** have to have two registers free to cycle values in and out of memory.

Aside - Sparrow

Consider the expression `x = a + b`. In Sparrow-V, we would need three lines:

```
u = [b]
u = s + u
[x] = u
```

Where `u` is a free register. `[b]` and `[x]` are the variables.
Note that you cannot assign `[x]` to the expression `s + u`, because of the way machine languages work; you cannot assign to expressions.

Colors and Cliques

A clique is defined as a group of variables that all share existence at the same time. The *maximal* clique is the clique with the most members. For the example above, the maximal clique would be variables b, c, e, and f, as those four variables are alive at the same time. A general rule is:

$$\text{The minimal number of colors} \geq \text{The size of the maximal clique}$$

This makes sense intuitively; we need at least n registers to store n coexisting variables.
When does this number go larger than the size of the maximal clique? Consider the following example:

Here, the maximal clique is size 2, since based on the graph, each variable is only bordering two others. However, we need three registers to color the graph.

Loops

What if instead of having a straight line program, we have a control flow program? (e.g., loops)

```
                          | a  b  c |       | c  a  b |
    (c from outer scope)  |       | |       | |       |
    a = 0                 | *     | |       | |  *    |
L1: b = a + 1             | |  *  | |       | |  |  * |
    c = c + b             | |  |  | |  -->  | |  |  | |
    a = b * 2             | |  0  | |       | |  |  0 |
    if (a < 10) goto L1   | *     | |       | |  *    |
    return c              |       0 |       | 0       |
```

On the right liveness analysis, we arrange them on order of appearance; in this case, if we only have two registers, then `C` can be sent to memory. `a` and `b` can be assigned to registers r_1 and r_2.
Note that in this case, `a` may stay alive after the goto statement, so its entry in the liveness analysis has a closed circle (`*`) on both ends. (e.g., it is inclusive on both sides).

Activation Records

Procedure Abstraction

Separate Compilation:

- Allows us to build large programs
- Keeps compile times reasonable
- Requires independent procedures

The linkage convention:

- a social contract: division of responsibility
- machine dependent
- ensures that every procedure inherits a valid run-time environment and that it restores one for its parent.

Code for linkage:

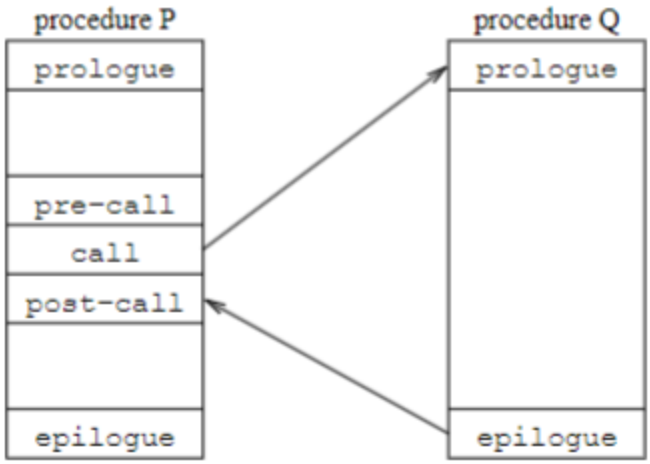
- Generated at *compile time*.
- Executes at *run time*.

Caller and Callee

The essentials:

- *on entry:* establish `p`'s environment
- *at a call:* preserve `p`'s environment
- *on exit:* tear down `p`'s environment
- *in between:* addressability and proper lifetimes

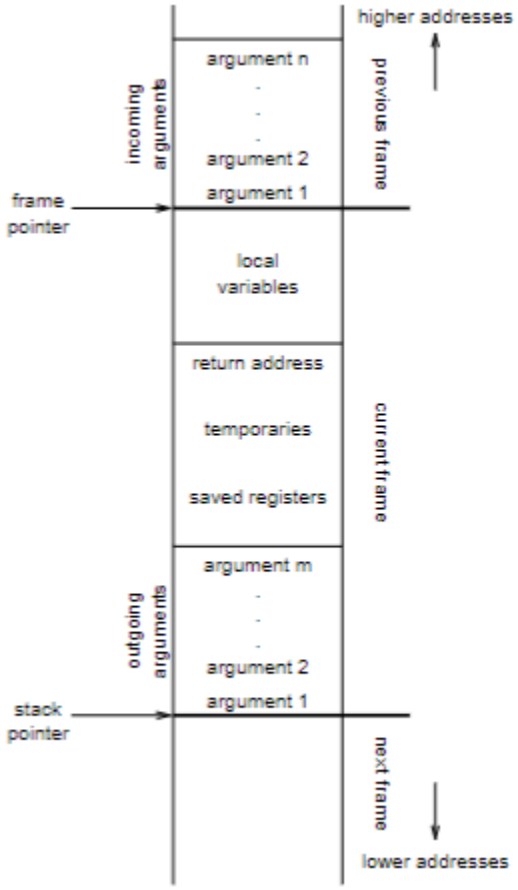
Each system has a standard linkage.



An activation record = a stack frame

Assume that each procedure activation has an associated *activation record* or *frame* (at run time). Assumptions:

- `RISC` architecture
- Can expand an allocated block
- Locals stored in frame



Procedure Linkages

The linkage divides responsibility between *caller* and *callee*.

	Caller	Callee
Call	*pre-call*	*prologue*
	1. allocate basic frame 2. evaluate & store params. 3. store return address 4. jump to child	1. save registers, state 2. store FP (dynamic link) 3. set new FP 4. store static link 5. extend basic frame (for local data) 6. initialize locals 7. fall through to code
Return	*post-call*	*epilogue*
	1. copy return value 2. deallocate basic frame 3. restore parameters (if copy out)	1. store return value 2. restore state 3. cut back to basic frame 4. restore parent's FP 5. jump to return address

Calls: Saving and restoring registers

	caller's registers	callee's registers	all registers
callee saves	1	3	5
caller saves	2	4	6

1. Hard: call includes bitmap of caller's registers.
2. Easy: the caller saves and restores its own registers.
3. Easy: the callee saves and restores its own registers.
4. Hard: the caller uses a bitmap in callee's stack frame to save and restore.
5. Easy: the callee saves and restores all registers.
6. Easy: the caller saves and restores all registers.

RISC-V Registers

Register	Name	Saver	Description
x0	zero	-	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	-	Global pointer
x4	tp	-	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register / frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments / return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-6	Caller	Temporaries

Philosophy: Use a general calling sequence only when necessary; omit portions of it where possible.