Register Allocation, i

Overview & spilling
p ::= (label f ...)
f ::= (label nat nat i ...)
i ::= (w <- s)
    | (w <- (mem x n8))
    | ((mem x n8) <- s)
    | (w aop= t)
    | (w sop= sx)
    | (w sop= num)
    | (w <- t cmp t)
    | label
    | (goto label)
    | (cjump t cmp t label label)
    | (call u nat)
    | (call print 1)
    | (call allocate 2)
    | (call array-error 2)
    | (tail-call u nat0-6)
    | (return)

aop ::= := + | - = * = & =
sop ::= <<= | >>=
cmp ::= < | <= | =
u ::= w | label
t ::= x | num
s ::= x | num | label
x ::= w | rsp
w ::= a | rax | rbx | rbp | r10 | r11 | r12 | r13 | r14 | r15
a ::= rdi | rsi | rdx | sx | r8 | r9
sx ::= rcx

label ::= sequence of chars matching #rx"^[a-zA-Z_][a-zA-Z_0-9]*$"
p ::= (label f ...)
f ::= (label nat nat i ...)
i ::= (w <- s)
    | (w <- (mem x n8))
    | ((mem x n8) <- s)
    | (w aop= t)
    | (w sop= sx)
    | (w sop= num)
    | (w <- t cmp t)
    | label
    | (goto label)
    | (cjump t cmp t label label)
    | (call u nat)
    | (call print 1)
    | (call allocate 2)
    | (call array-error 2)
    | (tail-call u nat0-6)
    | (return)
    | (w <- (stack-arg n8))
aop ::= + | - | * | &
sop ::= << | >>
cmp ::= < | <= | =
u ::= w | label
t ::= x | num
s ::= x | num | label
x ::= w | rsp
w ::= a | rax | rbx | rbp | r10 | r11 | r12 | r13 | r14 | r15
a ::= rdi | rsi | rdx | sx | r8 | r9
sx ::= rcx | var
label ::= sequence of chars matching #rx"^[a-zA-Z_][a-zA-Z_0-9]*"
L2 semantics: variables

L2 behaves just like L1, except that non-reg variables are function local, e.g.,

\[
\begin{align*}
\text{(define (f x)} & \Rightarrow (:m \\
& (:m 0 0 \\
& (rdi <- 21) \\
& (tail-call :f 1))) \\
\text{(define (g x))} & \\
& (+ x 2)) \\
\text{(f 10)} & \\
\end{align*}
\]

The assignment to \text{temp} in \text{g} does not break \text{f}, but if \text{temp} were a register, it would.
L2 semantics: stack-arg

L2 has a convenience for accessing stack-based arguments: \((\text{stack-arg \ n8})\). It is equivalent to \((\text{mem \ rsp \ ?})\) where the ? is the \n8, plus enough space for the spills. That is, \((\text{stack-arg \ 0})\) is always the last stack argument, \((\text{stack-arg \ 8})\) is always the second to last argument, etc.

This means that if the second natural number in the function header changes, then the \text{stack-arg} references don’t have to change – they will still be referring to the same arguments.
Register allocation, in three parts; for each function body we do:

• **Liveness analysis** ⇒ interference graph (nodes are variables; edges indicate “cannot be in the same register”)

• **Graph coloring** ⇒ register assignments

• **Spilling:** coping with too few registers

• Bonus part, **coalescing** eliminating redundant \((x \leftarrow y)\) instructions
Example Function

\[
\text{int } f(\text{int } x) = 2x^2 + 3x + 4
\]

(:f
  1 0
  (x2 <- rdi)
  (x2 *= x2)
  (dx2 <- x2)
  (dx2 *= 2)
  (tx <- rdi)
  (tx *= 3)
  (rax <- dx2)
  (rax += tx)
  (rax += 4)
  (return))
Example Function: live ranges

\[
\text{int } f(\text{int } x) = 2x^2 + 3x + 4
\]

dx2 tx x2

(:f
1 0
(x2 <- rdi)
(x2 *= x2)
(dx2 <- x2)
(dx2 *= 2)
(tx <- rdi)
(tx *= 3)
(rax <- dx2)
(rax += tx)
(rax += 4)
(return))
Example Function: live ranges

```plaintext
int f(int x) = 2x^2 + 3x + 4

dx2 tx x2 r10 r11 r12 r13 r14 r15 r8 r9 rax rbp rbx rcx rdi rdx rsi
(:f
1 0
(x2 <= rdi)
(x2 == x2)
(dx2 <= x2)
(dx2 == 2)
(tx <= rdi)
(tx == 3)
(rax <= dx2)
(rax += tx)
(rax += 4)
(return))
```
Example Function 2

\[ \text{int } f(\text{int } x) = x+x+x+x+x+x+x+x \] (in a stupid compiler)

\begin{align*}
&\text{a b c d e f g h r10 r11 r12 r13 r14 r15 r8 r9 rax rbp rbx rcx rdi rdx rsi}

&(:f
& 1 0
&(a <- rdi)
&(b <- rdi)
&(c <- rdi)
&(d <- rdi)
&(e <- rdi)
&(f <- rdi)
&(g <- rdi)
&(h <- rdi)
&(rax <- a)
&(rax += b)
&(rax += c)
&(rax += d)
&(rax += e)
&(rax += f)
&(rax += g)
&(rax += h)
&(return))
\end{align*}
No way to get all of a, b, c, d, e, f, g, and h into their own registers; so we need to *spill* one of them.
Spilling

Spilling is a program rewrite to make it easier to allocate registers

• Pick a variable

• Make a new location on the stack (increment the second nat in the function definition)

• Replace all writes to the variable with writes to the new stack location

• Replace all reads from the variable with reads from the new stack location

Sometimes that means introducing new temporaries
Spilling Example

Say we want to spill \( a \) to first location on the stack, 
\((\text{mem rsp 0})\); two easy cases:

\[(a \leftarrow 1) \Rightarrow ((\text{mem rsp 0}) \leftarrow 1)\]

\[(x \leftarrow a) \Rightarrow (x \leftarrow (\text{mem rsp 0}))\]
Example Function 2, need to spill

```c
int f(int x) = x+x+x+x+x+x+x+x
```

(in a stupid compiler)

```
: f
1 0
(a <- rdi)
(b <- rdi)
(c <- rdi)
(d <- rdi)
(e <- rdi)
(f <- rdi)
(g <- rdi)
(h <- rdi)
(rax <- a)
(rax += b)
(rax += c)
(rax += d)
(rax += e)
(rax += f)
(rax += g)
(rax += h)
(return)
```
Example Function 2, spilling a

\[ \text{int } f(\text{int } x) = x+x+x+x+x+x+x \text{ (in a stupid compiler)} \]

\[
\begin{align*}
&b \leftarrow \text{rdi} \\
&c \leftarrow \text{rdi} \\
&d \leftarrow \text{rdi} \\
&e \leftarrow \text{rdi} \\
&f \leftarrow \text{rdi} \\
&g \leftarrow \text{rdi} \\
&h \leftarrow \text{rdi} \\
&rax \leftarrow (\text{mem rsp 0}) \\
&rax += b \\
&rax += c \\
&rax += d \\
&rax += e \\
&rax += f \\
&rax += g \\
&rax += h \\
\text{return}
\end{align*}
\]
Spilling Example

A trickier case:

\[(a *= a) \Rightarrow (a_{\text{new}} <- (\text{mem rsp 0}))\]
\[(a_{\text{new}} *= a_{\text{new}})\]
\[((\text{mem rsp 0}) <- a_{\text{new}})\]

In general, make up a new temporary for each instruction that uses the variable to be spilled.

This makes for very short live ranges.
Example Function 2, spilling b

\textbf{int} \ f(\textbf{int} \ x) = x+x+x+x+x+x+x+x \ (\text{in a stupid compiler})

a c d e f g h s0 r10 r11 r12 r13 r14 r15 r8 r9 rax rbp rbx rcx rdi rdx rsi

\textbf{(:f}
\textbf{1 1}
(a <- rdi)
(((\text{mem rsp 0}) <- rdi)
(c <- rdi)
(d <- rdi)
(e <- rdi)
(f <- rdi)
(g <- rdi)
(h <- rdi)
(rax <- a)
(s0 <- (\text{mem rsp 0}))
(rax += s0)
(rax += c)
(rax += d)
(rax += e)
(rax += f)
(rax += g)
(rax += h)
(return))
Example Function 2, spilling b

Even though we still have eight temporaries, we can still allocate them to our seven unused registers because the live ranges of \( s0 \) and \( a \) don’t overlap and so they can go into the same register.
Your job

Implement:

```
spill : (label nat nat i ...) ;; original func
c
var          ;; to spill
c
var          ;; prefix for temporaries
c
-> (label nat nat i ...) ;; spilled func
c
```
Here’s how to two example spilled functions from the earlier slides would look like as calls to spill:

```
(spill «the original program»
 'a
 's)

(spill «the original program»
 'b
 's)
```

See the assignment handout for more details on the precise spec for test cases and your spill function’s interface