Register Allocation, i
Overview & spilling
p::= ((i ...) (label i ...) ...)
i::= (x <- s)
    | (x <- (mem x n4))
    | ((mem x n4) <- s)
    | (x aop= t)
    | (x sop= sx)
    | (x sop= num)
    | (cx <- t cmp t)
    | label
    | (goto label)
    | (cjump t cmp t label label)
    | (call u)
    | (tail-call u)
    | (return)
    | (eax <- (print t))
    | (eax <- (allocate t t))
    | (eax <- (array-error t t))

aop=::=:++ | -= | *= | &=
sop::=<= | >=
cmp::=< | <= | =
s::= x | num | label
t::= x | num
u::= x | label
x, y::= cx | esi | edi | ebp | esp
cx::= eax | ecx | edx | ebx
sx::= ecx
\texttt{p ::= ((i ... ) (label i ... ) ... )}
\texttt{i ::= (x <- s)}
  | (x <- (mem x n4))
  | ((mem x n4) <- s)
  | (x aop= t)
  | (x sop= sx)
  | (x sop= num)
  | (cx <- t cmp t)
\texttt{label}
  | (goto label)
  | (cjump t cmp t label label)
  | (call u)
  | (tail-call u)
  | (return)
  | (eax <- (print t))
  | (eax <- (allocate t t))
  | (eax <- (array-error t t))
\texttt{aop ::= ::= + | -= | *= | &=}
\texttt{sop ::= <,= | >>=}
\texttt{cmp ::= < | <= | =}
\texttt{s ::= x | num | label}
\texttt{t ::= x | num}
\texttt{u ::= x | label}
\texttt{x, y ::= cx | esi | edi | ebp | esp}
\texttt{cx ::= eax | ecx | edx | ebx | var}
\texttt{sx ::= ecx | var}
\texttt{var ::= variable matching regexp ^ [a-zA-Z\_] [a-zA-Z0-9\_]*$,}
  except registers and keywords (e.g., \texttt{print}, \texttt{call}, \texttt{cjump}, ...)
L2 semantics: variables

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

\[
\begin{align*}
\text{(define } (f \ x) \Rightarrow ( & (; :\text{main} \\
& (+ (g \ x) 1)) \\
& (call :f)) \\
\text{(define } (g \ x) ( & (:f (temp <- 1) \\
& (call :g) \\
& (eax += temp) \\
& (return)) \\
& (:g (temp <- 2) \\
& (eax += temp) \\
& (return))))
\end{align*}
\]

The assignment to temp in g does not break f, but if temp were a register, it would.
L2 semantics: esp & ebp

L2 programs must use neither esp nor ebp. They are in L2 to facilitate register allocation only, not for the L3 → L2 compiler’s use.
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

\[ f(x) = 2x^2 + 3x + 4 \]

```c
:int f(int x) = 2*x^2 + 3*x + 4
:
:int x2, eax, tx
:
(x2 <- eax)
(x2 *= x2)
(dx2 <- x2)
(dx2 *= 2)
(tx <- eax)
(tx *= 3)
(eax <- dx2)
(eax += tx)
(eax += 4)
(return)
```
Example Function: live ranges

```c
int f(int x) = 2x^2 + 3x + 4
dx2 tx x2
```

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

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

::f
(x2 <- eax)
(x2 *= x2)
(dx2 <- x2)
(dx2 *= 2)
(tx <- eax)
(tx *= 3)
(eax <- dx2)
(eax += tx)
(eax += 4)
(return)
Example Function 2

int $f(int x) = x + x + x + x$ (in a stupid compiler)

```
abcedx ebx ecx edi edx esi
```

```lang
: f
(a <- eax)
(b <- eax)
(c <- eax)
(d <- eax)
(eax <- a)
(eax += b)
(eax += c)
(eax += d)
(return)
```
No way to get all of $a$, $b$, $c$, and $d$ into their own registers; so we need to spill one of them.
Spilling is a program rewrite to make it easier to allocate registers

- Pick a variable and a location on the stack for it
- Replace all writes to the variable with writes to the stack
- Replace all reads from the variable with reads from the stack

Sometimes that means introducing new temporaries
Spilling Example

Say we want to spill $a$ to the location $(\text{mem ebp } -4)$. Two easy cases:

$$(a <- 1) \Rightarrow ((\text{mem ebp } -4) <- 1)$$

$$(x <- a) \Rightarrow (x <- (\text{mem ebp } -4))$$
Example Function 2, need to spill

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

```
:a  b  c  d  eax  ebx  ecx  edi  edx  esi
    \(\text{:f}\)
(a <- eax)
(b <- eax)
(c <- eax)
(d <- eax)
(eax <- a)
(eax += b)
(eax += c)
(eax += d)
(return)
```
Example Function 2, spilling a

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

\[
:\text{f} \\
((\text{mem ebp } -4) \leftarrow \text{eax}) \\
(b \leftarrow \text{eax}) \\
(c \leftarrow \text{eax}) \\
(d \leftarrow \text{eax}) \\
(eax \leftarrow (\text{mem ebp } -4)) \\
(\text{eax } + = b) \\
(\text{eax } + = c) \\
(\text{eax } + = d) \\
(\text{return})
\]
Spilling Example

A trickier case:

\[(a \,*=\, a) \Rightarrow (a_{\text{new}} \leftarrow (\text{mem} \, \text{ebp} \,-\, 4))\]
\[(a_{\text{new}} \,*=\, a_{\text{new}})\]
\[( (\text{mem} \, \text{ebp} \,-\, 4) \leftarrow a_{\text{new}})\]

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

```c
int f(int x) = x+x+x+x (in a stupid compiler)

:f
(a <- eax)
((mem ebp -4) <- eax)
(c <- eax)
(d <- eax)
(eax <- a)
(s0 <- (mem ebp -4))
(eax += s0)
(eax += c)
(eax += d)
(return)
```
Example Function 2, spilling b

Even though we still have four temporaries, we can still allocate them to our three 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 : (i ...) ;; original function
var    ;; to spill
offset ;; multiple of 4
var    ;; prefix for temporaries
-> (i ...) ;; spilled version
```