Programming in L1
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 read 0)
  | (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]*$"
Two topics to cover:

• Value encoding
• Calling convention
(:go) ; name of main function
(:go) ; define :go
 0 0 ; no args, no stack space
(rdi <- 5) ; rdi is first arg to ...
(call print 1) ; ... a runtime call
(return)

produces the output:
( :go ; name of main function
  (:go ; define :go
      0 0 ; no args, no stack space
      ( rdi <- 5) ; rdi is first arg to ...
      (call print 1) ; ... a runtime call
      (return)))

produces the output:

2
print (and allocate and array-error) need to
tell if they have an integer or an array; the lowest bit is
what determines that

• \( x \& 1 = 0 \Rightarrow X \) is a pointer to an array of values; the
  first word has the length of the array, the rest of the
  words are values

• \( x \& 1 = 1 \Rightarrow x \gg 1 \) is a 63 bit two’s complement
  integer

efficient trick for runtime representations that our
compiler will use
Register Overview:

<table>
<thead>
<tr>
<th>Args</th>
<th>Result</th>
<th>Caller Save</th>
<th>Callee Save</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdi</td>
<td>rax</td>
<td>r10</td>
<td>r12</td>
</tr>
<tr>
<td>rsi</td>
<td></td>
<td>r11</td>
<td>r13</td>
</tr>
<tr>
<td>rdx</td>
<td></td>
<td>r8</td>
<td>r14</td>
</tr>
<tr>
<td>rcx</td>
<td></td>
<td>r9</td>
<td>r15</td>
</tr>
<tr>
<td>r8</td>
<td></td>
<td>rax</td>
<td>rbp</td>
</tr>
<tr>
<td>r9</td>
<td></td>
<td>rcx</td>
<td>rbx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rdi</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rdx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rsi</td>
<td></td>
</tr>
</tbody>
</table>
prints an array of size two with two 3s in it:

\{s:2, 3, 3\}

because allocate’s first argument is the size and the second argument is what to initialize it with
(go
 (go
  0 0
  (rdi <- 5)
  (rsi <- 7)
  (call allocate 2)
  (rdi <- 7) ; make an array where
  (rsi <- rax) ; the elements all point
  (call allocate 2) ; at the same array
  (rdi <- rax)
  (call print 1)
  (return)))

prints

{s:3, {s:2, 3, 3}, {s:2, 3, 3}, {s:2, 3, 3}}
An array with \( n \) values is represented by a pointer to \( n+1 \) words of space. The first word is the size of the array. It is not encoded. This program:

```
(:main
 (:main
  0 0
 (rdi <- 7)
 (rsi <- 0)
 (call allocate 2)
 (rdi <- (mem rax 0))
 (call print 1)
 (return)))
```

produces 1
The **array-error** function accepts two encoded arguments and raises an error, aborting the program with an error message about array indexing out of bounds. The first argument must be the pointer to the array (i.e., the pointer to the size word) and the second argument must be the index that was attempted to index at. It prints an error message indicating that the program failed.

The intention is that the compilation of safe arrays compiles to a call to this function in the case that an array index goes out of bounds.
Making function calls: the calling convention

**Invariant:** \( \text{rsp} \) (the stack pointer) is never modified directly; instead the `call`, `tail-call`, and `return` instructions modify it to do their jobs.

(See the language grammar.)

Thus, the stack frame is fixed \( \text{wrt} \ \text{rsp} \) during the execution of any given function.
Stack Frame:
\( \leq 6 \) args, no locals

\[
\begin{align*}
(mem \ rsp \ 8) \rightarrow & \quad \vdots \\
(mem \ rsp \ 0) \rightarrow & \quad \text{ret addr} \\
(mem \ rsp \ -8) \rightarrow &
\end{align*}
\]
Stack Frame:
> 6 args, no locals

```
(mem rsp 8) → arg 7
(mem rsp 0) → arg n
(mem rsp -8) →
```

...
Stack Frame:
\( \leq 6 \) args, some locals

\[
\begin{align*}
\text{(mem rsp 8)} & \rightarrow \\
\text{(mem rsp 0)} & \rightarrow \\
\text{(mem rsp -8)} & \rightarrow \\
\text{spills} & \\
\text{ret addr} & \\
\end{align*}
\]
Stack Frame:
> 6 args, some locals

```
(mem rsp 8) → spils
(mem rsp 0) →
(mem rsp -8) →
```
Case 1: regular call, \( \leq 6 \) args, no stack space for callee

(:main
 (:main
  0 0
 ; set up a register argument
  \[\begin{array}{l}
  \text{==> } (\text{rdi }]<- 1) \\
  \text{; set up return address}
  \\
  \text{(}\text{mem }\text{rsp }-8)<- \text{:\text{f}_ret})
  \\
  \text{; 'call' bumps rsp & jumps}
  \\
  \text{(call :f 1)}
  \\
  \text{; thus we return here}
  \\
  \text{:f\_ret}
  \\
  \text{(return))}
  \\
  \text{x}\text{f}
  \\
  \text{1 0}
  \\
  \text{(\text{rax }]<- 1)}
  \\
  \text{; return decs rsp & jumps}
  \\
  \text{(return))}\end{array}\]

\[\vdots\]
\[\text{ret}\] \quad \text{rsp}
Case 1: regular call, \(\leq 6\) args, no stack space for callee

\[
\begin{align*}
(:\text{main} \\
(:\text{main} \\
0&\ 0 \\
;\ \text{set up a register argument} \\
(rdi &\leftarrow 1) \\
;\ \text{set up return address} \\
\Rightarrow &\ ((\text{mem} \ rst -8) &\leftarrow :f\_\text{ret}) \\
;\ \text{‘call’ bumps rst & jumps} \\
(call &\ :f\ 1) \\
;\ \text{thus we return here} \\
:f\_\text{ret} \\
(return)) \\
(:f \\
1&\ 0 \\
(rax &\leftarrow 1) \\
;\ \text{return decs rst & jumps} \\
(return))))
\end{align*}
\]
Case 1: regular call, ≤ 6 args, no stack space for callee

```scheme
(:main
 (:main
  0 0
  ; set up a register argument
  (rdi <- 1)
  ; set up return address
  ((mem rsp -8) <- :f_ret)
  ; ‘call’ bumps rsp & jumps
 ==>
  (call :f 1)
  ; thus we return here
  :f_ret
  (return))
 (:f
  1 0
  (rax <- 1)
  ; return decs rsp & jumps
  (return)))
```

![Diagram]

```scheme
ret
```

```scheme
rsp
```

```scheme
:f_ret
```
Case 1: regular call, ≤ 6 args, no stack space for callee

```plaintext
(:main
 (:main
  0 0
  ; set up a register argument
  (rdi <- 1)
  ; set up return address
  ((mem rsp -8) <- :f_ret)
  ; 'call' bumps rsp & jumps
  (call :f 1)
  ; thus we return here
  :f_ret
  (return))
 (:f
  1 0
===> (rax <- 1)
  ; return decs rsp & jumps
  (return)))
```
Case 1: regular call, $\leq 6$ args, no stack space for callee

(:main
 (:main
  0 0
  ; set up a register argument
  (rdi <- 1)
  ; set up return address
  ((mem rsp -8) <- :f_ret)
  ; 'call' bumps rsp & jumps
  (call :f 1)
  ; thus we return here
  :f_ret
  (return))
(:f
  1 0
  (rax <- 1)
  ; return decs rsp & jumps
  ==> (return)))
Case 1: regular call, \( \leq 6 \) args, no stack space for callee

\[
(:\text{main}
 (:\text{main}
  0 0
 ; \text{set up a register argument}
  (\text{rdi} <- 1)
 ; \text{set up return address}
  ((\text{mem} \text{ rsp} -8) <- :f_{\text{ret}})
 ; \text{‘call’ bumps \text{ rsp} & jumps}
  (\text{call} :f 1)
 ; \text{thus we return here}
 ==/> :f_{\text{ret}}
 (\text{return}))
 (:f
  1 0
 (\text{rax} <- 1)
 ; \text{return decs \text{ rsp} & jumps}
 (\text{return})))
\]
Case 1: regular call, \( \leq 6 \) args, no stack space for callee

```plaintext
(:main
 (:main
  0 0
 ; set up a register argument
 (rdi <- 1)
 ; set up return address
 ((mem rsp -8) <- :f_ret)
 ; 'call' bumps rsp & jumps
 (call :f 1)
 ; thus we return here
 :f_ret

 ===> (return))

 (:f
  1 0
 (rax <- 1)
 ; return decs rsp & jumps
 (return)))
```
Case 2: regular call, ≤ 6 args, need stack space for callee

```scheme
(:main
  (:main
    0 0
  ))
=> (rdi <- 1)
  ((mem rsp -8) <- :f_ret)
  (call :f 1)
  :f_ret
  (return))
(:f
  1 3
  ((mem rsp 0) <- rdi)
  (rax <- (mem rsp 0))
  ((mem rsp 8) <- 3)
  ((mem rsp 16) <- 5)
  (return)))
```
Case 2: regular call, $\leq 6$ args, need stack space for callee

```
(:main
  (:main
    0 0
    (rdi <- 1)
  )
  (call :f 1)
  (:f_ret
    (return))
)(:
  (:f
    1 3
    ((mem rsp 0) <- rdi)
    (rax <- (mem rsp 0))
    ((mem rsp 8) <- 3)
    ((mem rsp 16) <- 5)
    (return)))
```
Case 2: regular call, $\leq 6$ args, need stack space for callee

```scheme
(:main
 (:main
  0 0
  (rdi <- 1)
  ((mem rsp -8) <- :f_ret)
 (call :f 1)
 :f_ret
 (return))
 (:f
  1 3
  ((mem rsp 0) <- rdi)
  (rax <- (mem rsp 0))
  ((mem rsp 8) <- 3)
  ((mem rsp 16) <- 5)
  (return)))
```

\[
\begin{array}{c}
\text{ret} \\
: \quad \text{rsp} \\
: : \text{:f_ret}
\end{array}
\]

\[\cdots\]
Case 2: regular call, \(\leq 6\) args, need stack space for callee

\[
(:\text{main}
 (:\text{main}
  0 0
 (\text{rdi} \leftarrow 1)
 ((\text{mem} \ \text{rsp} \ -8) \leftarrow :\text{f\_ret})
 (\text{call} :f\ 1)
 :f\_ret
 (\text{return}))
 (:f
  1 3
 ==>)
 ((\text{mem} \ \text{rsp} \ 0) \leftarrow \text{rdi})
 (\text{rax} \leftarrow (\text{mem} \ \text{rsp} \ 0))
 ((\text{mem} \ \text{rsp} \ 8) \leftarrow 3)
 ((\text{mem} \ \text{rsp} \ 16) \leftarrow 5)
 (\text{return})))
\]
Case 2: regular call, $\leq 6$ args, need stack space for callee

```
(:main
 (:main
  0 0
  (rdi <- 1)
  ((mem rsp -8) <- :f_ret)
  (call :f 1)
  :f_ret
  (return))
 (:f
  1 3
  ((mem rsp 0) <- rdi)
  (rax <- (mem rsp 0))
  ((mem rsp 8) <- 3)
  ((mem rsp 16) <- 5)
  (return))
```

```
: ret
: f_ret
  ☠
  ☠
  1
  rsp
```
Case 2: regular call, ≤ 6 args, need stack space for callee

```scheme
(:main
 (:main
  0 0
  (rdi <- 1)
  ((mem rsp -8) <- :f_ret)
  (call :f 1)
  :f_ret
  (return))
(:f
  1 3
  ((mem rsp 0) <- rdi)
  (rax <- (mem rsp 0))
  ==>
  ((mem rsp 8) <- 3)
  ((mem rsp 16) <- 5)
  (return)))
```

```
|
| ret
| :f_ret
| ☠ ☠
| 1
| rsp
```
Case 2: regular call, \( \leq 6 \) args, need stack space for callee

```scheme
(:main
  (:main
    0 0
    (rdi <- 1)
    ((mem rsp -8) <- :f_ret)
    (call :f 1)
  :f_ret
    (return))
(:f
  1 3
  ((mem rsp 0) <- rdi)
  (rax <- (mem rsp 0))
  ((mem rsp 8) <- 3)
  ((mem rsp 16) <- 5)
  (return)))
```

```
ret
⋮
_f_ret
☠
3
1
```
Case 2: regular call, ≤ 6 args, need stack space for callee

```plaintext
(:main
  (:main
    0 0
    (rdi <- 1)
    ((mem rsp -8) <- :f_ret)
    (call :f 1)
    :f_ret
    (return))
  (:f
    1 3
    ((mem rsp 0) <- rdi)
    (rax <- (mem rsp 0))
    ((mem rsp 8) <- 3)
    ((mem rsp 16) <- 5)
  =>> (return)))
```

<table>
<thead>
<tr>
<th>ret</th>
<th>:f_ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>rsp</td>
</tr>
</tbody>
</table>
Case 2: regular call, ≤ 6 args, need stack space for callee

```
(:main
 (:main
  0 0
  (rdi <- 1)
  ((mem rsp -8) <- :f_ret)
  (call :f 1)
 )

 ==> :f_ret
 (return))

 (:f
  1 3
  ((mem rsp 0) <- rdi)
  (rax <- (mem rsp 0))
  ((mem rsp 8) <- 3)
  ((mem rsp 16) <- 5)
  (return)))
```
Case 2: regular call, \( \leq 6 \) args, need stack space for callee

\[
\begin{align*}
(:\text{main} \\
  (:\text{main} \\
    0 0 \\
    (\text{rdi} \leftarrow 1) \\
    ((\text{mem rsp} -8) \leftarrow :f_{\text{ret}}) \\
    (\text{call} :f\ 1) \\
    :f_{\text{ret}} \\
  ) \\
  \Rightarrow (\text{return}) \\
  (:f \\
    1\ 3 \\
    ((\text{mem rsp} 0) \leftarrow \text{rdi}) \\
    (\text{rax} \leftarrow (\text{mem rsp} 0)) \\
    ((\text{mem rsp} 8) \leftarrow 3) \\
    ((\text{mem rsp} 16) \leftarrow 5) \\
    (\text{return})
  )
\end{align*}
\]
Case 3: regular call, > 6 args, no stack space for callee

```plaintext
(:main
 (:main
  0 0
 0 0
 0 0
 0 0

  ((mem rsp -8) <- :f_ret)
  (rdi <- 3) (rsi <- 5) (rdx <- 7)
  (rcx <- 9) (r8 <- 11) (r9 <- 13)
  ((mem rsp -16) <- 15)
  ((mem rsp -24) <- 17)
  (call :f 8)
  :f_ret
  (rdi <- rax)
  (call print 1)
  (return))

 (:f
  8 0
 8 0
  (rax <- (mem rsp 8)) ; 7th arg
  (rax <- (mem rsp 0)) ; 8th arg
  (return)))
```
Case 3: regular call, > 6 args, no stack space for callee

```ml
(:main
 (:main
  0 0
  ((mem rsp -8) <- :f_ret)
 ==>
 (rdi <- 3) (rsi <- 5) (rdx <- 7)
 (rcx <- 9) (r8 <- 11) (r9 <- 13)
 ((mem rsp -16) <- 15)
 ((mem rsp -24) <- 17)
 (call :f 8)
 :f_ret
 (rdi <- rax)
 (call print 1)
 (return))
 (:f
  8 0
  (rax <- (mem rsp 8)) ; 7th arg
  (rax <- (mem rsp 0)) ; 8th arg
  (return)))
```
Case 3: regular call, > 6 args, no stack space for callee

```lisp
(:main
  (:main
    0 0
    ((mem rsp -8) <- :f_ret)
    (rdi <- 3) (rsi <- 5) (rdx <- 7)
    (rcx <- 9) (r8 <- 11) (r9 <- 13)
    ((mem rsp -16) <- 15)
    ((mem rsp -24) <- 17)
    (call :f 8)
    :f_ret
    (rdi <- rax)
    (call print 1)
    (return))
  (:f
    8 0
    (rax <- (mem rsp 8)) ; 7th arg
    (rax <- (mem rsp 0)) ; 8th arg
    (return)))
```
Case 3: regular call, > 6 args, no stack space for callee

```c
(:main
 (:main
  0 0
  ((mem rsp -8) <- :f_ret)
 (rdi <- 3) (rsi <- 5) (rdx <- 7)
 (rcx <- 9) (r8 <- 11) (r9 <- 13)

  ==>  ((mem rsp -16) <- 15)
  ((mem rsp -24) <- 17)
  (call :f 8)
  :f_ret
  (rdi <- rax)
  (call print 1)
  (return))

 (:f
  8 0
  (rax <- (mem rsp 8)) ; 7th arg
  (rax <- (mem rsp 0)) ; 8th arg
  (return)))
```

...
Case 3: regular call, > 6 args, no stack space for callee

```plaintext
(:main
 (:main
  0 0
  ((mem rsp -8) <- :f_ret)
  (rdi <- 3) (rsi <- 5) (rdx <- 7)
  (rcx <- 9) (r8 <- 11) (r9 <- 13)
  ((mem rsp -16) <- 15)
  ((mem rsp -24) <- 17)
  (call :f 8)
  :f_ret
  (rdi <- rax)
  (call print 1)
  (return))
  (:f
   8 0
   (rax <- (mem rsp 8)) ; 7th arg
   (rax <- (mem rsp 0)) ; 8th arg
   (return)))
```
Case 3: regular call, > 6 args, no stack space for callee

```plaintext
(:main
 (:main
  0 0
  ((mem rsp -8) <- :f_ret)
  (rdi <- 3) (rsi <- 5) (rdx <- 7)
  (rcx <- 9) (r8 <- 11) (r9 <- 13)
  ((mem rsp -16) <- 15)
  ((mem rsp -24) <- 17)

  --->
  (call :f 8)
  :f_ret
  (rdi <- rax)
  (call print 1)
  (return))

 (:f
  8 0
  (rax <- (mem rsp 8)) ; 7th arg
  (rax <- (mem rsp 0)) ; 8th arg
  (return)))
```

```
+---------------------+---+
|         ret         | 15|
|         rsp         | 17|
```

```
Case 3: regular call, > 6 args, no stack space for callee

```plaintext
(:main
 (:main
  0 0
  ((mem rsp -8) <- :f_ret)
  (rdi <- 3) (rsi <- 5) (rdx <- 7)
  (rcx <- 9) (r8 <- 11) (r9 <- 13)
  ((mem rsp -16) <- 15)
  ((mem rsp -24) <- 17)
  (call :f 8)
  :f_ret
  (rdi <- rax)
  (call print 1)
  (return))
 (:f
  8 0
  ==> (rax <- (mem rsp 8)) ; 7th arg
  (rax <- (mem rsp 0)) ; 8th arg
  (return)))
```
Case 3: regular call, > 6 args, no stack space for callee

```scheme
(:main
 (:main
  0 0
  ((mem rsp -8) <- :f_ret)
  (rdi <- 3) (rsi <- 5) (rdx <- 7)
  (rcx <- 9) (r8 <- 11) (r9 <- 13)
  ((mem rsp -16) <- 15)
  ((mem rsp -24) <- 17)
  (call :f 8)
  :f_ret
  (rdi <- rax)
  (call print 1)
  (return))
 (:f
  8 0
  (rax <- (mem rsp 8)) ; 7th arg
  ==> (rax <- (mem rsp 0)) ; 8th arg
  (return)))
```
Case 3: regular call, > 6 args, no stack space for callee

```plaintext
(:main
 (:main
  0 0
  ((mem rsp -8) <- :f_ret)
  (rdi <- 3) (rsi <- 5) (rdx <- 7)
  (rcx <- 9) (r8 <- 11) (r9 <- 13)
  ((mem rsp -16) <- 15)
  ((mem rsp -24) <- 17)
  (call :f 8)
  :f_ret
  (rdi <- rax)
  (call print 1)
  (return))
 (:f
  8 0
  (rax <- (mem rsp 8)) ; 7th arg
  (rax <- (mem rsp 0)) ; 8th arg
  =>> (return)))
```

```
 EVENTS

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>:f_ret</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>rsp</td>
</tr>
</tbody>
</table>
```
Case 3: regular call, > 6 args, no stack space for callee

```plaintext
(:main
 (:main
  0 0
  ((mem rsp -8) <- :f_ret)
  (rdi <- 3) (rsi <- 5) (rdx <- 7)
  (rcx <- 9) (r8 <- 11) (r9 <- 13)
  ((mem rsp -16) <- 15)
  ((mem rsp -24) <- 17)
  (call :f 8)

==> :f_ret
  (rdi <- rax)
  (call print 1)
  (return))

(:f
  8 0
  (rax <- (mem rsp 8)) ; 7th arg
  (rax <- (mem rsp 0)) ; 8th arg
  (return))
```

```
ret          rsp
+------------+
| :f_ret     |
| 15         |
| 17         |
```

...
Case 3: regular call, > 6 args, no stack space for callee

```lisp
(:main
  (:main
    0 0
    ((mem rsp -8) <- :f_ret)
    (rdi <- 3) (rsi <- 5) (rdx <- 7)
    (rcx <- 9) (r8 <- 11) (r9 <- 13)
    ((mem rsp -16) <- 15)
    ((mem rsp -24) <- 17)
    (call :f 8)
    :f_ret
  )
  :f_ret
  (rdi <- rax)
  (call print 1)
  (return))

(:f
  8 0
  (rax <- (mem rsp 8)) ; 7th arg
  (rax <- (mem rsp 0)) ; 8th arg
  (return)))
```

<table>
<thead>
<tr>
<th>ret</th>
<th>rsp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

...
Case 3: regular call, > 6 args, no stack space for callee

```plaintext
(:main
 (:main
  0 0
  ((mem rsp -8) <- :f_ret)
  (rdi <- 3) (rsi <- 5) (rdx <- 7)
  (rcx <- 9) (r8 <- 11) (r9 <- 13)
  ((mem rsp -16) <- 15)
  ((mem rsp -24) <- 17)
  (call :f 8)
  :f_ret
  (rdi <- rax)
  
  ==> (call print 1)
  (return))

(:f
  8 0
  (rax <- (mem rsp 8)) ; 7th arg
  (rax <- (mem rsp 0)) ; 8th arg
  (return)))
```
Case 3: regular call, > 6 args, no stack space for callee

```plaintext
(:main
 (:main
  0 0
  ((mem rsp -8) <- :f_ret)
  (rdi <- 3) (rsi <- 5) (rdx <- 7)
  (rcx <- 9) (r8 <- 11) (r9 <- 13)
  ((mem rsp -16) <- 15)
  ((mem rsp -24) <- 17)
  (call :f 8)
  :f_ret
  (rdi <- rax)
  (call print 1)
  (return))
==>
 (:f
  8 0
  (rax <- (mem rsp 8)) ; 7th arg
  (rax <- (mem rsp 0)) ; 8th arg
  (return)))
```
The \texttt{read} function accepts no arguments and produces an encoded integer, reading one line from the standard input stream of the program. If the line has anything other than ASCII digits (possibly with a leading \texttt{-}), if there are more than 6 digits, or if stdin is closed, \texttt{read} returns 1 (the encoded version of 0).

If \texttt{11} is typed on stdin, this program prints out \texttt{11}. In general, it echos a single integer on its output.

\begin{verbatim}
(:go
 (:go
  0 0
 (call read 0)
 (rdi <- rax)
 (call print 1)
 (return))
\end{verbatim}
This program accepts two numbers and prints their sum.

```
(:go
 (:go
  0 1
  (call read 0) ((mem rsp 0) <- rax)
  (call read 0) (rdi <- (mem rsp 0))
  (rdi += rax) (rdi -= 1) (call print 1)
  (return))
```

If stdin has this string: "11\n12\n" then this program produces < < 23. The < characters are the prompts; one for each call to `read`. Note that running this program in the interpreter will look like this:

```
< 11
< 12
23
```

because the \n s are part of the input, not the output.
Fill an array with numbers counting down

```plaintext
(:main
   (:main
      0 1
      (rdi <- 21)
      (rsi <- 1)
      (call allocate 2)
      ((mem rsp 0) <- rax)
      (rdi <- rax)
      ((mem rsp -8) <- :fill_done)
      (call :fill 1)
      :fill_done
      (rdi <- (mem rsp 0))
      (call print 1)
      (return))
   (:fill
      1 0
      (rax <- (mem rdi 0))
      (rdi += 8)
      :loop
      (cjump rax = 0 :done :more)
      :more
      (rsi <- rax)
      (rsi *= 2)
      (rsi += 1)
      ((mem rdi 0) <- rsi)
      (rax -= 1)
      (rdi += 8)
      (goto :loop)
      :done
      (return)))
```

Sum up the contents of an array

```plaintext
(:main
   (:main
      0 1
      (rdi <- 21)
      (rsi <- 5)
      (call allocate 2)
      ((mem rsp 0) <- rax)
      (rdi <- rax)
      (rsi <- 0)
      ((mem rsp -8) <- :sum_done)
      (call :sum 2)
      :sum_done
      (rdi <- rax)
      (call print 1)
      (return))
   (:sum
      ;; rdi: pointer to array
      ;; rsi: position in array
      2 1
      (rax <- (mem rdi 0))
      (cjump rax = rsi :done :more)
      :done
      (rax <- 1)
      (return)
   :more
      (rax <- rsi) ;; compute offset
      (rax += 1) ;; into the array
      (rax *= 8) ;; for the number
      (rax += rdi) ;; to add to the sum
      (rax <- (mem rax 0))
      ((mem rsp 0) <- rax) ;; stash that on stack
      (rsi += 1) ;; set up next args & call
      ((mem rsp -8) <- :sum_return)
      (call :sum 2)
      :sum_return
      (rdi <- (mem rsp 0))
      (rax += rdi) ;; compute the sum; make
      (rax -= 1) ;; sure it's encoded
      (return)))
```