

Programming in LI

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:

Args	Result	Caller Save	Callee Save
rdi	rax	r10	r12
rsi		r11	r13
rdx		r8	r14
rcx		r9	r15
r8		rax	rbp
r9		rcx	rbx
		rdi	
		rdx	
		rsi	

```
(:go
(:go
0 0
(rdi <- 5)           ; rdi is first arg,
(rsi <- 7)           ; rsi is the second arg,
(call allocate 2)
(rdi <- rax)         ; rax is the result
(call print 1)
(return)))
```

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: `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.)

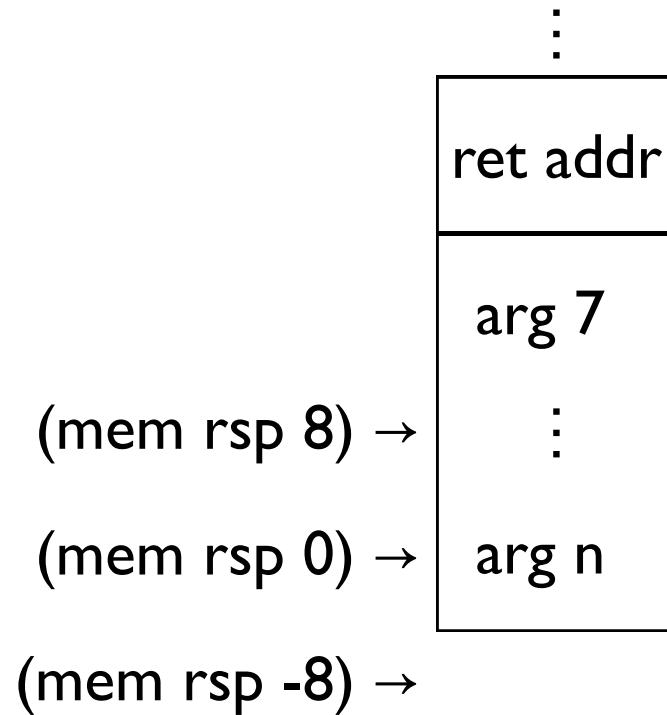
Thus, the stack frame is fixed wrt `rsp` during the execution of any given function.

Stack Frame:
 ≤ 6 args, no locals

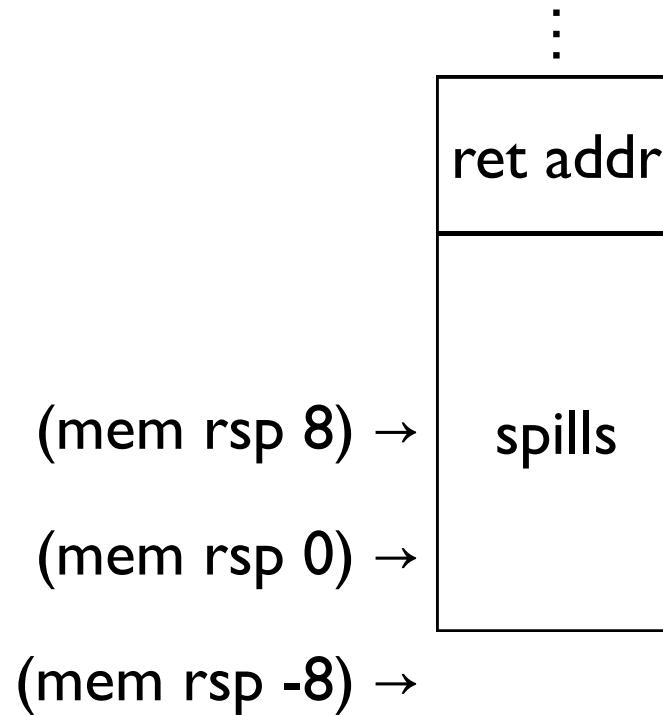
(mem rsp 8) → :
(mem rsp 0) → ret addr
(mem rsp -8) →

Stack Frame:

> 6 args, no locals

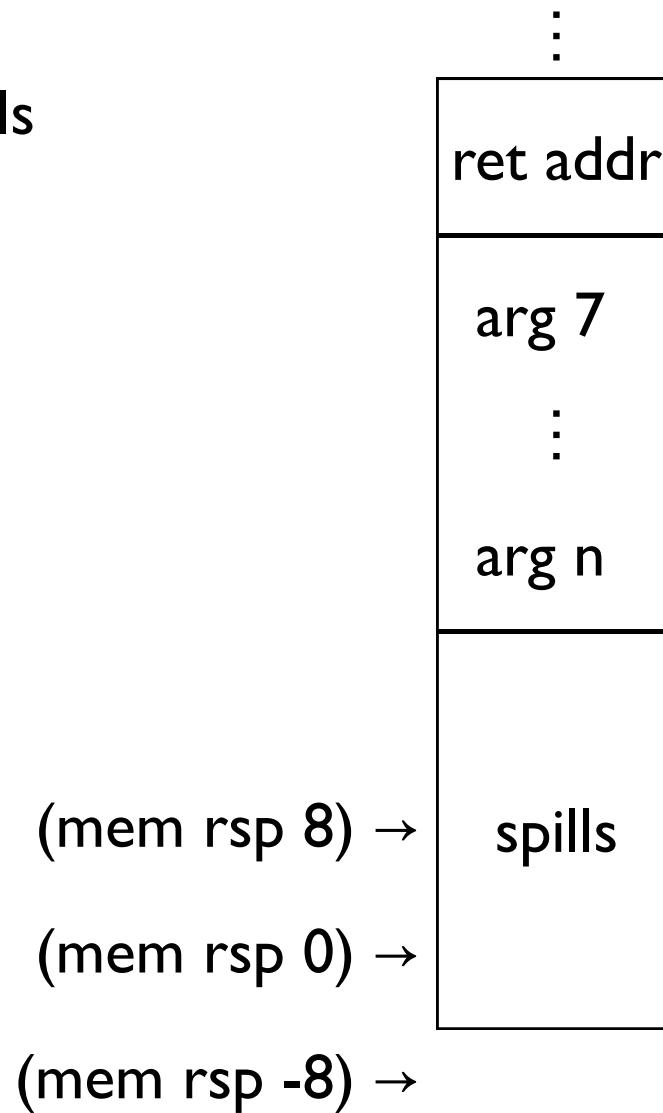


Stack Frame:
 ≤ 6 args, some locals



Stack Frame:

> 6 args, some locals



Case I: regular call, ≤ 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)))
```



The diagram shows the state of the stack after the call. A box labeled "rsp" contains the instruction "ret". Above the box is a colon ":".

Case I: regular call, ≤ 6 args, no stack space for callee

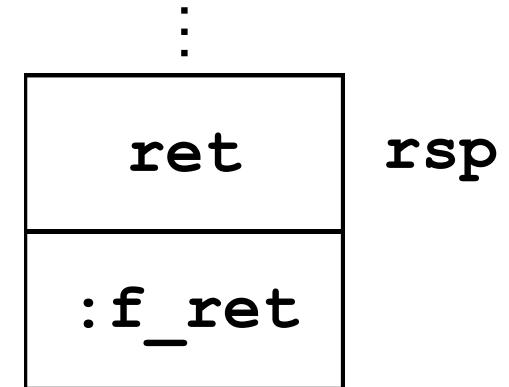
```
(:main
  (:main
    0 0
    ; set up a register argument
    (rdi <- 1)
    ; set up return address
==> ((mem rsp -8) <- :f_ret)
    ; 'call' bumps rsp & jumps
    (call :f 1)
    ; thus we return here
    :f_ret
    (return))
(:f
  1 0
  (rax <- 1)
  ; return decs rsp & jumps
  (return)))
```



: rsp

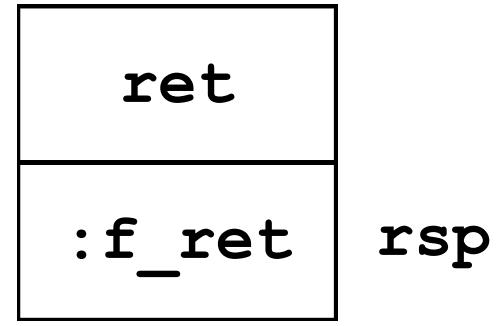
Case I: regular call, ≤ 6 args, no stack space for callee

```
(: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 I: regular call, ≤ 6 args, no stack space for callee

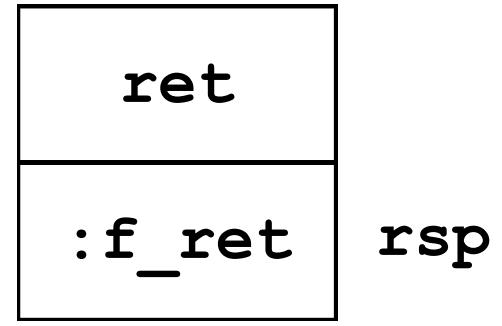
```
( :main
  (:main
    0 0
    ; set up a register argument
    (rdi <- 1)
    ; set up return address
    ((mem rsp -8) <- :f_ret)
    ; 'call' bumps rsp & jumps
    (call :f 1)
    ; thus we return here
    :f_ret
    (return))
  (:f
    1 0
==>  (rax <- 1)
    ; return decs rsp & jumps
    (return)))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



The diagram illustrates the state of the stack (rsp) after the call. It consists of three horizontal boxes. The top box contains the label ':f_ret'. Below it is a box containing the instruction 'ret'. The bottom box contains the label 'rsp' preceded by a colon.

Case I: regular call, ≤ 6 args, no stack space for callee

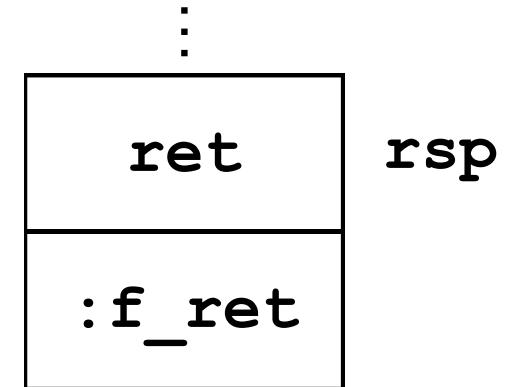
```
( :main
  (:main
    0 0
    ; set up a register argument
    (rdi <- 1)
    ; set up return address
    ((mem rsp -8) <- :f_ret)
    ; 'call' bumps rsp & jumps
    (call :f 1)
    ; thus we return here
    :f_ret
    (return))
  (:f
    1 0
    (rax <- 1)
    ; return decs rsp & jumps
==> (return)))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



The diagram illustrates the state of the stack (rsp) after the call. It consists of three horizontal boxes. The top box contains the label ':f_ret'. Below it is a box containing the instruction 'ret'. The bottom box contains the label 'rsp' preceded by a colon.

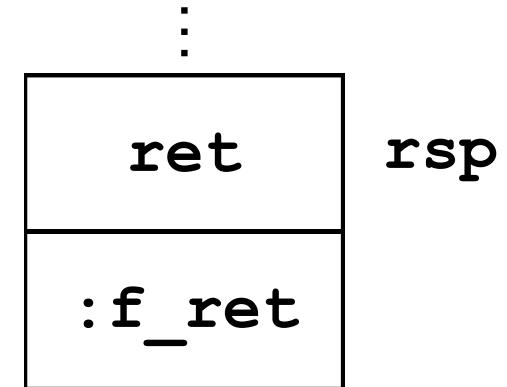
Case I: regular call, ≤ 6 args, no stack space for callee

```
(: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 I: regular call, ≤ 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 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)))
```



The diagram shows the stack layout after the call. It consists of a rectangular box containing the word "ret", followed by a colon, and then the register name "rsp". This indicates that the return address has been pushed onto the stack before the function call.

Case 2: regular call, ≤ 6 args, need stack space for callee

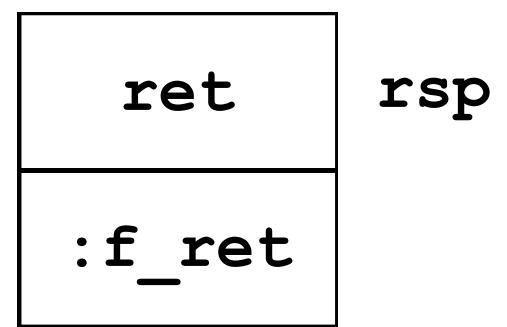
```
(:main
  (:main
    0 0
    (rdi <- 1)
==> ((mem rsp -8) <- :f_ret)
    (call :f 1)
    :f_ret
    (return))
  (:f
    1 3
    ((mem rsp 0) <- rdi)
    (rax <- (mem rsp 0)))
    ((mem rsp 8) <- 3)
    ((mem rsp 16) <- 5)
    (return)))
```



The diagram illustrates the stack layout after the call. A rectangular box contains the word "ret". Above this box is a colon ":". To the right of the box is the register name "rsp". This visualizes the state where the return address is stored at the top of the stack (offset -8 from the current stack pointer), and the current stack pointer points to the start of the local variables.

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)))
```



:

ret	rsp
:f_ret	

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)))
```

The diagram illustrates the stack layout for the function `f`. The stack grows from bottom to top. It contains three local variables (rax, rdi, and rsi), each with a size of 8 bytes, indicated by skull icons. The stack pointer (`rsp`) is at the bottom, pointing to the start of the stack frame. The return address (`:f_ret`) is located at offset -8 from the current stack pointer. The function ends with a `ret` instruction.

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)))
```

The stack diagram shows the state of the stack after the call. The stack grows downwards. It contains:

- A return address labeled **ret**.
- A slot for the function argument labeled **:f_ret**.
- Two slots for local variables, each containing a skull icon.
- The value **1** at the bottom, aligned with the register **rsp**.

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)))
```

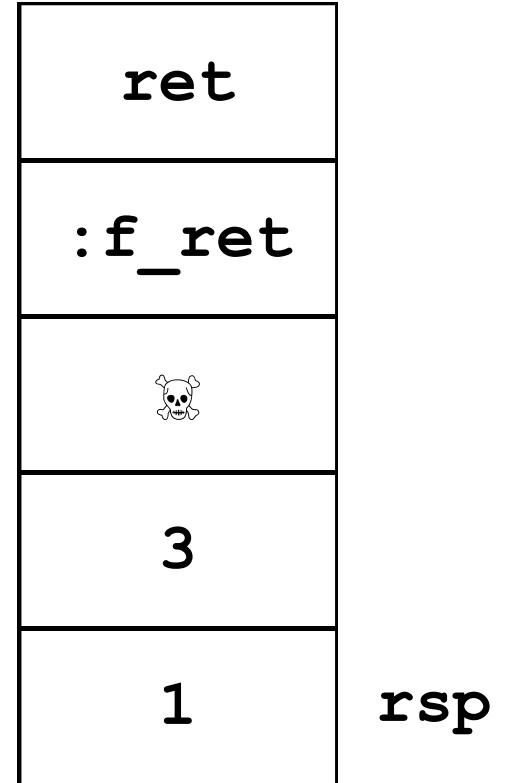
The stack diagram shows the state of the stack after the call. The stack grows downwards. It contains:

- A return address at the top labeled **ret**.
- A slot for the function's return value containing a skull icon.
- A slot for the function's return value containing a skull icon.
- The arguments 1 and 3.

The label **rsp** is positioned to the right of the stack diagram.

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)))
```



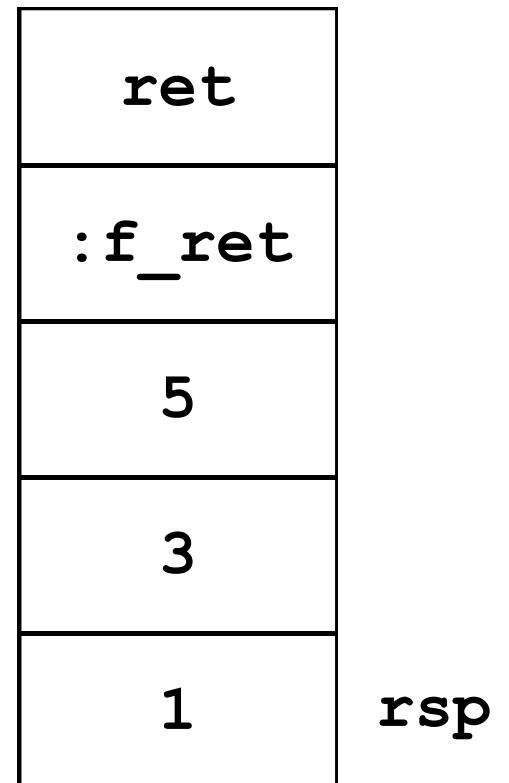
The stack diagram shows the state of the stack after the function call. The stack grows downwards. It contains:

- A return address (**ret**)
- A pointer to the function's return address (**:f_ret**)
- A temporary variable (represented by a skull icon)
- The argument **3**
- The value **1**

The base of the stack is labeled **rsp**.

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)))
```



The stack diagram shows the state of the stack after the call. The stack grows downwards. It contains the following elements from top to bottom:

- ret
- :f_ret
- 5
- 3
- 1

The base of the stack is labeled "rsp".

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)))
```

The stack diagram shows the state of the stack after the call. The stack grows downwards. It contains the return address (ret), the function label (:f_ret), the argument 5, the argument 3, and the argument 1.

ret	rsp
:f_ret	
5	
3	
1	

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)))
```

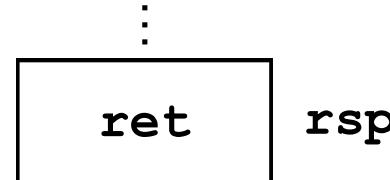
The stack diagram shows the state of the stack after the call. The stack grows downwards. It contains the following elements from top to bottom:

- ret
- :f_ret
- 5
- 3
- 1

On the right side of the stack, the label "rsp" is shown with a colon above it, indicating the current position of the stack pointer.

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

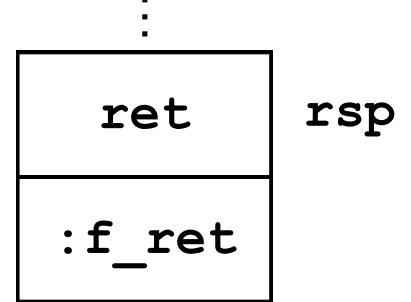
```
(:main
  (:main
    0 0
==>  ((mem rsp -8) <- :f_ret)
      (rdi <- 3) (rsi <- 5) (rdx <- 7)
      (rcx <- 9) (r8 <- 11) (r9 <- 13)
      ((mem rsp -16) <- 15)
      ((mem rsp -24) <- 17)
      (call :f 8)
      :f_ret
      (rdi <- rax)
      (call print 1)
      (return))
(:f
  8 0
  (rax <- (mem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



The diagram illustrates the stack layout after the call to :f. The stack grows downwards from the current value of `rsp`. The arguments are pushed onto the stack in reverse order: 8, 0, `rax`, `rax`, 0, 11, 13, 9, 7, 5, 3, and 15. The label `:f_ret` is positioned above the stack, indicating the return address.

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

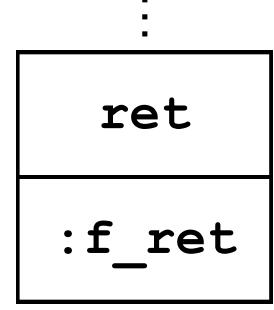
```
(:main
  (:main
    0 0
    ((mem rsp -8) <- :f_ret)
==>  (rdi <- 3) (rsi <- 5) (rdx <- 7)
        (rcx <- 9) (r8 <- 11) (r9 <- 13)
        ((mem rsp -16) <- 15)
        ((mem rsp -24) <- 17)
        (call :f 8)
        :f_ret
        (rdi <- rax)
        (call print 1)
        (return))
(:f
  8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



The stack diagram shows the state of the stack (rsp) after the call to :f. The stack grows downwards. The current top of the stack is at offset 0. The stack contains the return address (ret) at offset -8 and the label :f_ret at offset -16.

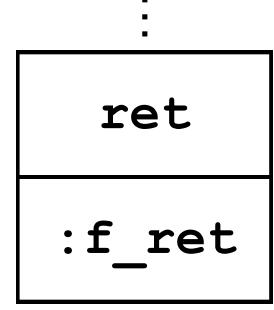
Case 3: regular call, > 6 args, no stack space for callee

```
( :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)))
```



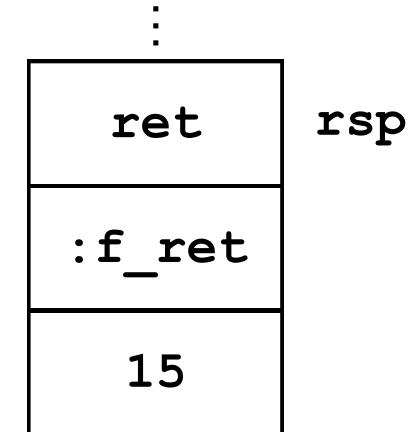
Case 3: regular call, > 6 args, no stack space for callee

```
( :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)))
```



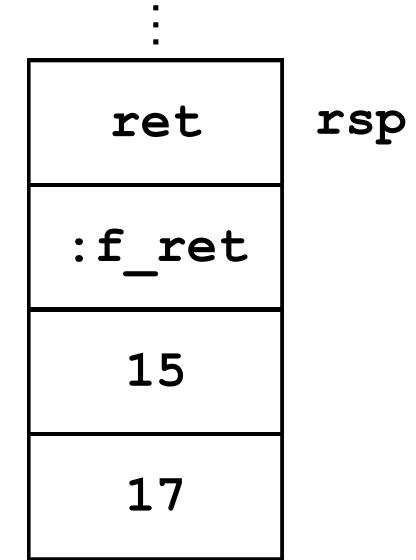
Case 3: regular call, > 6 args, no stack space for callee

```
( :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)))
```



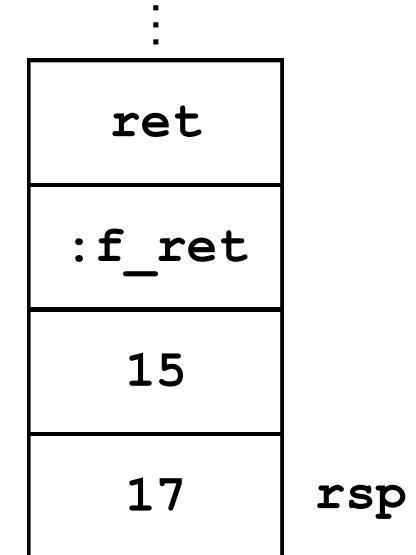
Case 3: regular call, > 6 args, no stack space for callee

```
(: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)))
```



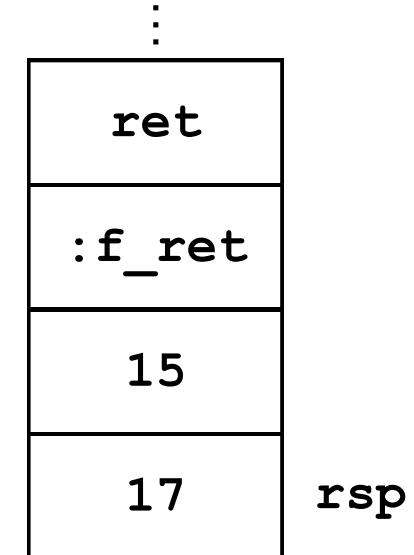
Case 3: regular call, > 6 args, no stack space for callee

```
(: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)))
```



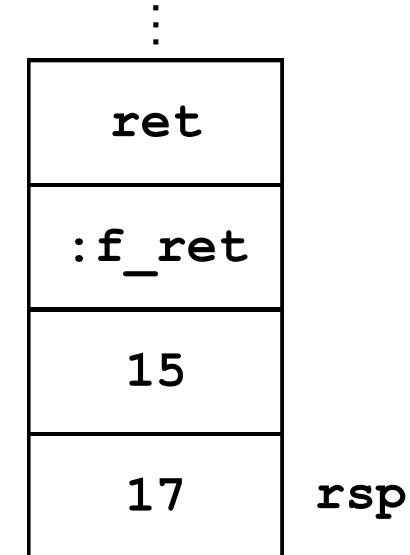
Case 3: regular call, > 6 args, no stack space for callee

```
(: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)))
```



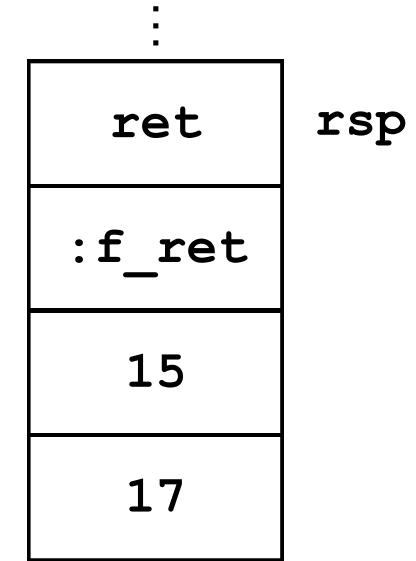
Case 3: regular call, > 6 args, no stack space for callee

```
( :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)))
```



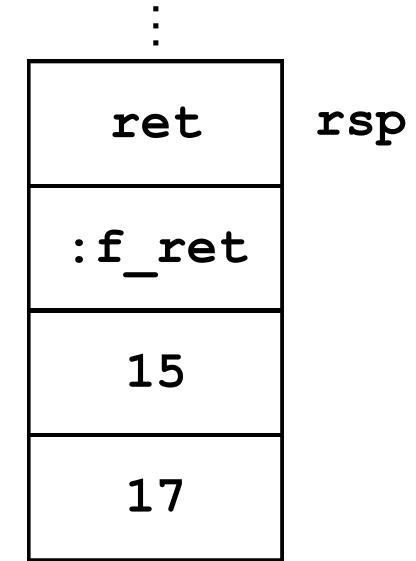
Case 3: regular call, > 6 args, no stack space for callee

```
(: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)))
```



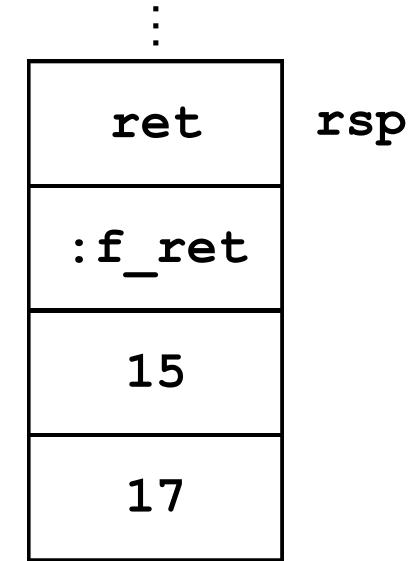
Case 3: regular call, > 6 args, no stack space for callee

```
(: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)))
```



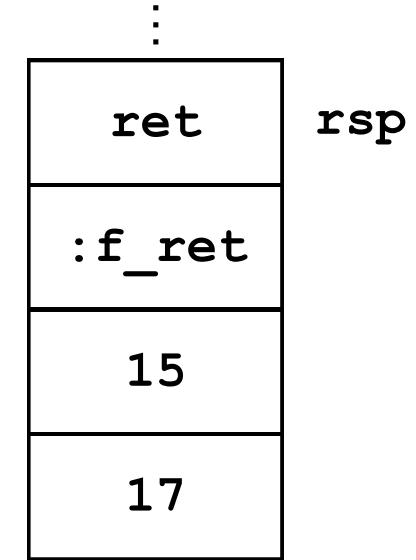
Case 3: regular call, > 6 args, no stack space for callee

```
(: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

```
( :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 `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 `-`), if there are more than 6 digits, or if `stdin` is closed, `read` returns `1` (the encoded version of `0`).

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

```
(:go
(:go
 0 0
(call read 0)
(rdi <- rax)
(call print 1)
(return)))
```

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

```
(: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

```
(: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)))
```