1 Introduction to Recursion

In this section, we'll talk about recursion. Note that, in our examples, we'll assume that the callee manages (i.e., moves) the stack pointer. In particular, this means everything will have a positive offset from `rsp`.

1.1 Recursive Sum Example

Let's consider the following code:

```
(fun (sumrec num)
    (if (= num 0)
        0
        (+ num (sumrec (+ num -1)))
    )
)
```

This program simply performs $1 + 2 + 3 + \ldots +$ `num`. The generated assembly would look something like what is shown below.

```
sumrec:
    sub rsp, 16
    mov rax, [rsp + 24]
    ... if (= num 0)
    cmp rax, 1
    je ifelse_1
         mov rax, 0
         jmp ifend_0
    ifelse_1:
         ... put temp num on stack for LHS
         mov [rsp + 0], rax
         mov rax, [rsp + 24]
         ... (+ num -1) stored in rax
         ... now do 1-arg calling conv
         sub rsp, 16
         mov [rsp], rax
         mov [rsp+8], rdi
         call sumrec
         mov rdi, [rsp+8]
         add rsp, 16
         ... do addition on the waiting num ...
         add rax, [rsp + 0]
    ifend_0:

    add rsp, 16
    ret
```

Note that only relevant assembly is shown. Some things to point out:

- In the second assembly line, `sub rsp, 16`, the `16` is the *depth* that we calculated.
- In the lines before the recursive call, i.e.,

```
sub rsp, 16
mov [rsp], rax
mov [rsp+8], rdi
call sumrec
```

we're moving the arguments into the correct position in memory so the recursive call can make use of them.

- When we run a `call` instruction, `rsp` is moved up one word and the return pointer to the next line of instruction (program counter) is put in that location in memory (where `rsp` is pointing to).

To see how the memory looks when each line of assembly is executed, see `Lec12Trace.pdf`.

1.2 Second Recursive Sum Example

Let's rewrite the recursive sum example a bit.

```
(fun (sumrec num sofar)
    (if (= num 0)
        sofar
        (sumrec (+ num -1) (+ sofar num))
    )
)
```

The generated assembly might look like

```
sumrec:
    sub rsp, 16

    mov rax, [rsp + 24]
    mov [rsp + 0], rax
    ... if (= num 0)
    cmp rax, 1
    je ifelse_1
        mov rax, [rsp + 32]
        jmp ifend_0
    ifelse_1:
        mov rax, [rsp + 24]
        ... add -1 to num, store on stack as tmp ...
        mov [rsp + 0], rax

        mov rax, [rsp + 32]
        ... add sofar to num, store in rax ...
        add rax, [rsp + 8]

        ... 2-arg calling convention from class ...
        sub rsp, 24
        mov rbx, [rsp+24]
        mov [rsp], rbx
        mov [rsp+8], rax
        mov [rsp+16], rdi
        call sumrec             ; (A)
        mov rdi, [rsp+16]       ; (B)
        add rsp, 24             ; (C)

    ifend_0:
    add rsp, 16                 ; (C)
    ret                         ; (D)
```

An interesting thing to note is that, after reaching the base case, there's no additional calculation that needs to be made. In particular, the steps after returning is

(a) Move `rsp` back. Remember that, after `call` is done (i.e., when `ret` is executed), `rsp` is moved back one word.

(b) Restore `rdi`.

(c) Move `rsp` back more.

(d) Return!

No local variables or arguments were accessed.