1 Calling Conventions

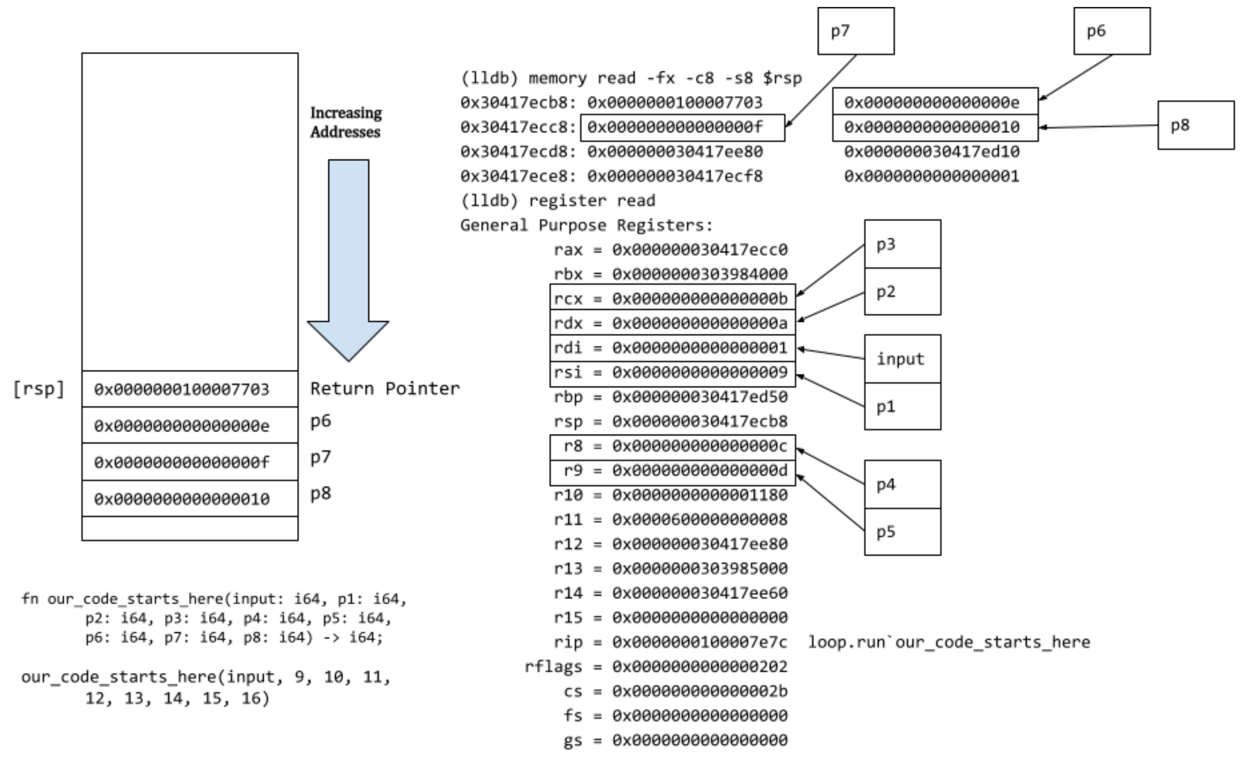
In this section, we'll talk more about calling conventions.

1.1 Argument Conventions in x86_64

Consider the following code (assuming that only this part of the code has changed.)

```
#[link(name = "our_code")]
extern "C" {
#[link_name = "\x01our_code_starts_here"]
    fn our_code_starts_here(input : i64, p1 : i64, p2 : i64, ..., p8 : i64) -> i64;
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let input = parse_arg(&args);
    let i : i64 = unsafe {
        our_code_starts_here(input, 9, 10, 11, 12, 13, 14, 15, 16)
    };
    snek_print(i);
}
```

Here, we added 8 parameters to the our_code_starts_here function so we can see how the arguments are represented in memory. Using `rust-lldb`, we can debug this code and, more importantly, see the memory layout of our program.



```
fn our_code_starts_here(input: i64, p1: i64,
      p2: i64, p3: i64, p4: i64, p5: i64,
      p6: i64, p7: i64, p8: i64) -> i64;

our_code_starts_here(input, 9, 10, 11,
      12, 13, 14, 15, 16)
```

```
(lldb) memory read -fx -c8 -s8 $rsp
0x30417ecb8: 0x0000000100007703    0x000000000000000e
0x30417ecc8: 0x000000000000000f    0x0000000000000010
0x30417ecd8: 0x000000030417ee80    0x000000030417ed10
0x30417ece8: 0x000000030417ecf8    0x0000000000000001
(lldb) register read
General Purpose Registers:
       rax = 0x000000030417ecc0
       rbx = 0x0000000303984000
       rcx = 0x000000000000000b
       rdx = 0x000000000000000a
       rdi = 0x0000000000000001
       rsi = 0x0000000000000009
       rbp = 0x000000030417ed50
       rsp = 0x000000030417ecb8
        r8 = 0x000000000000000c
        r9 = 0x000000000000000d
       r10 = 0x0000000000001180
       r11 = 0x0000600000000008
       r12 = 0x000000030417ee80
       r13 = 0x0000000303985000
       r14 = 0x000000030417ee60
       r15 = 0x0000000000000000
       rip = 0x0000000100007e7c  loop.run`our_code_starts_here
    rflags = 0x0000000000000202
        cs = 0x000000000000002b
        fs = 0x0000000000000000
        gs = 0x0000000000000000
```

The main takeaways to get are, in x86_64,

- The first 6 arguments are `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`.
- Any remaining arguments will go in order in increasing addresses **after** `[rsp]`, where `[rsp]` holds the return pointer.

1.2 Functions & Their Conventions for Us

How do we add functions to our programming language? We need to restructure our programming langauge a bit. First, our program will have the following definition:

```
<prog> := <defn>* <expr>

<defn> := (fun (<name> <name>) <expr>)
    | (fun (<name> <name> <name>) <expr>)

<expr> := ...
    | (<name> <expr>)
    | (<name> <expr> <expr>)
```

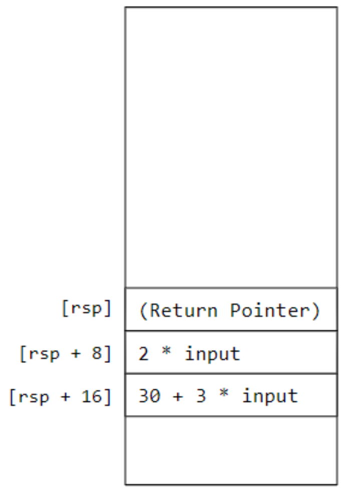
In particular,

- `<prgm>` is what our `snek` file will look like. `<prgm>` is saying that it takes zero or more function definitions, and then the expression. So, past `snek` files had zero function definitions. Now, `snek` files can have more than zero function definitions.
- In our version of `<defn>`,
  - `(fun (<name> <name>) <expr>)` means that the function declaration has a *name*, and then one argument after it, and then the following expression.
  - `(fun (<name> <name> <name>) <expr>)` means that the function declaration has a *name*, and then *two arguments* after the name, and then the following expression.
- In `<expr>`, not only do we have the usual language features (e.g., numbers, identifiers, binary operations, etc.), but we also have support for calling the above functions. Specifically,
  - `(<name> <expr>)` calls the function `<name>` with a single argument.
  - `(<name> <expr> <expr>)` calls the function `<name>` with two arguments.

Consider the following code.

```
(fun (sumsquare x y)
    (+ (* x x) (* y y)))

(sumsquare (* 2 input) (+ 30 (* 3 input)))
```

When `sumsquare` starts, we should expect the memory and stack to look something like:



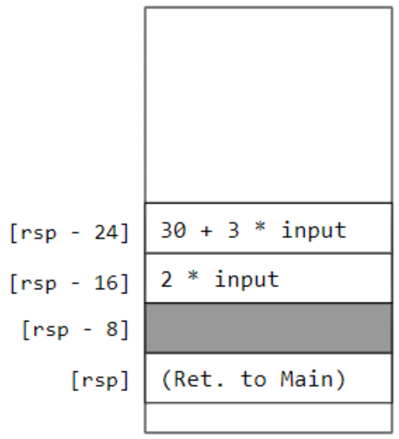
In other words, the value of the first argument, $2 \times$ `input`, when we called the function is in `[rsp + 8]`; likewise, the value of the second argument, $3 \times$ `input` $+ 30$, when we called the function is in `[rsp + 16]`. So, in general, how should we generate functions in our assembly code?

- For each function definition, we will generate a label.
- Arguments will be located in the following places:
  - The value of the first argument is located in `[rsp + 8]`.
  - The value of the second argument is located in `[rsp + 16]`.
  - In general, the value of the ith argument is located in `[rsp + 8(i + 1)]`.

  This is controlled by the environment used for the body.

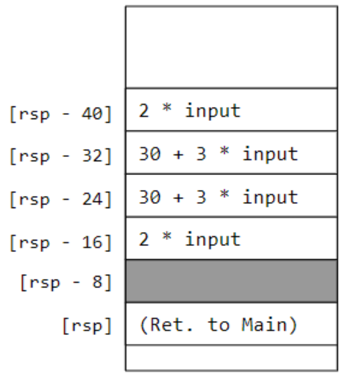
Let's try to sketch out the assembly of the above code, where `sumsquare` is defined and where it is being called, and the corresponding memory model after running it.

```
; [rsp + 8] is x
; [rsp + 16] is y
sumsquare:
    ; TODO
    ret

; basically, our main function
our_code_starts_here:
    ; code for (* 2 input) -> rax
    mov [rsp - 16], rax
    ; code for (+ 30 (* 3 input))
    mov [rsp - 24], rax
```



Notice that the order of the arguments isn't correct, so we can't use the arguments as they appear right here. So, we need to do some more moving.

```
; [rsp + 8] is x
; [rsp + 16] is y
sumsquare:
    ; TODO
    ret

; basically, our main function
our_code_starts_here:
    ; code for (* 2 input) -> rax
    mov [rsp - 16], rax
    ; code for (+ 30 (* 3 input))
    mov [rsp - 24], rax
    ; additional moves for args
    sub rsp, 48 ; 8 * 6
                ; put at "top" of stack
    call sumsquare
    add rsp, 48
```



We also may want to **save** any values that may be caller-saved (e.g., any registers like `rdi`). This should be done before doing any additional moves. Basically, we don't want to overwrite anything from a previous stack frame.

1.3 Improvements and Approaches

Some small ideas we could think about are:

1. The last argument can avoid copying. This is mainly so we can avoid doing some unnecessary minor work.
2. Evaluate arguments in reverse order (last to first). Then, they are in the right place on the stack.

Both of these run into issues with saving and restoring registers! So, a **big idea** is to **pre-allocate stack frames**[1]. In other words, we move `rsp` at the *start* of a function to accommodate *all* local variables. Then, all references are `[rsp + ___]`. The interesting thing to think about is how much we need to move `rsp` to accommodate all local variables. Once we figure this out, then we can adjust `rsp` as needed.

Introducing, the `depth` function. This calculates the number of local variables (named or unnamed) are in this expression. This is basically just tracking stack indices.

```
fn depth(e: &Expr) -> i32 {
    match e {
        Plus(e1, e2) => {
            max(depth(e1), depth(e2) + 1)
        }
        Let(id, val, body) => {
            max(depth(val), depth(body) + 1)
        }
        ...
    }
}
```

In a `Plus` expression, which has `e1` and `e2`, we're taking the maximum of

- `depth(e1)` (Because we don't increase the stack index when we go into `e1`), and
- `depth(e2) + 1` (Remember that the plus expression makes space for one word, and then reserves that space while it's working on `e2`. This corresponds to how we did `si + 1` in our compiler.)

Analogously, we can say the same for the `Let` expression.

So, going back to compiling `sumsquare`, the first thing we can do is

```
sumsquare:
    sub rsp, depth(body * 8)
    .
    .
    .
    add rsp, depth(body * 8)
    ret
```

Note that all local variables use `[rsp + ___]` in this function. This means less calculations and moving `rsp` at each call.

[1]This is what Rust and C does.