# 1 Our Calling Convention

We'll now talk more about the calling conventions we aim to use for our compiler, along with functions in general. Recall, from the previous section, that a `snek` program is defined by

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

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

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

where `<defn>` means that our function declarations can either take one argument, or two arguments, respectively.

## 1.1 Caller-Managed Stack Pointer

An approach for calling functions is to have the caller (i.e., the function that is *calling* a function) manage the stack pointer, `rsp`.

### 1.1.1 Compiling the Definition

Let's consider the following Rust code:

```
fn compile_definition(d: &Definition, labels: &mut i32) -> String {
    match d {
        Fun1(name, arg, body) => {
            ...
        }
        Fun2(name, arg1, arg2, body) => {
            let body_env = hashmap! {
                arg1.to_string() => -1,
                arg2.to_string() => -2
            };
            let body_is = compile_expr(body, 2, &body_env,
                &String::from(""), labels);
            format!(
                "{name}:
                {body_is}
                ret"
            )
        }
    }
}
```

This function is designed to compile a function declaration, specifically a function with one and two arguments. Some things to think about:

- Compiling a function is straightforward with this calling convention. All there is to the actual function is
  - The label (perhaps, the name of the function).
  - The body of the function.
  - And then, `ret` for returning.

- Note that the *environment* is set up so that negative stack indexes are used. This way, the compiler ends up accessing `[rsp + X]`, i.e., accessing memory downwards as opposed to upwards.
- In this approach, all the work of manipulating `rsp` is done by the **caller**.

### 1.1.2 Compiling the Function Calls

Now, let's look at the code that's responsible for compiling *function calls*.

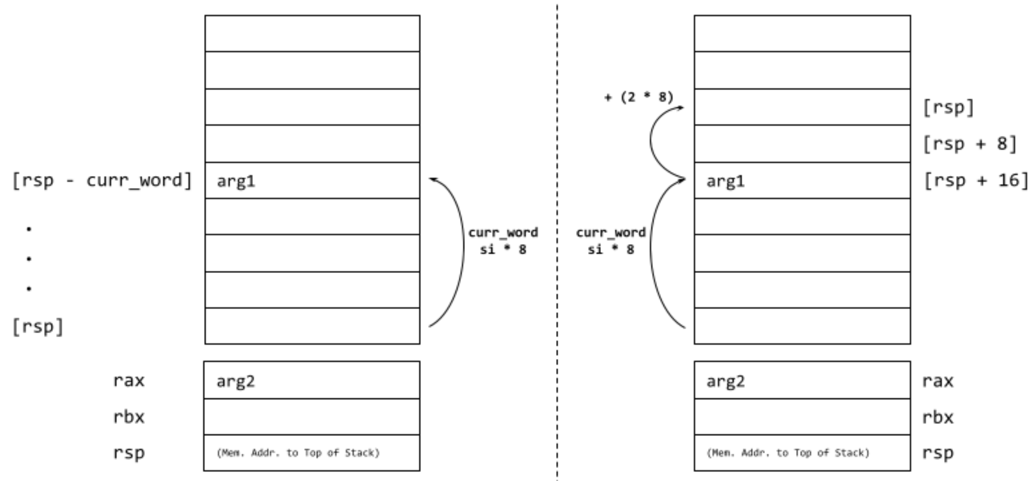
```
fn compile_expr(
    e: &Expr,
    si: i32,
    env: &HashMap<String, i32>,
    brake: &String,
    l: &mut i32,
) -> String {
    match e {
        ...
        Expr::Call2(name, arg1, arg2) => {
            let arg1_is = compile_expr(arg1, si, env, brake, l);
            let arg2_is = compile_expr(arg2, si + 1, env, brake, l);
            let curr_word = si * 8;
            let offset = (si * 8) + (2 * 8);
            // With this setup, the current word will be at [rsp+16],
            // which is where arg1 is stored. We then want to get rdi
            // at [rsp+16], arg2 at [rsp+8], and arg1 at [rsp], then call
            format!(
                "
                {arg1_is}
                mov [rsp-{curr_word}], rax
                {arg2_is}
                sub rsp, {offset}
                mov rbx, [rsp+16]
                mov [rsp], rbx
                mov [rsp+8], rax
                mov [rsp+16], rdi
                call {name}
                mov rdi, [rsp+16]
                add rsp, {offset}
            "
            )
        }
    }
}
```

There are some things we need to make sure:

- We need to make sure we set `rsp` high enough so that it doesn't interfere with any of the temporary variables.
- We also need to make sure we store the variables (arguments) in the right place before we actually call the function. `offset` is defined so that it will be used to move `rsp` above where we are, and then subtracting some more words, so we can make room for `rdi` and **two** arguments.
- Note that, even though we have three items (`rdi` and the two arguments), we only need to move the stack index up by an additional 16 spaces, hence why we're adding `2 * 8`.

### 1.1.3 Memory Layout

With the above code for function *calling*, let's look at a memory diagram of what's going on prior to calling a function. We know that `curr_word` is defined as `si * 8`.




(A) Before **sub rsp, {offset}** (B) After **sub rsp, {offset}**

So, what's going on?

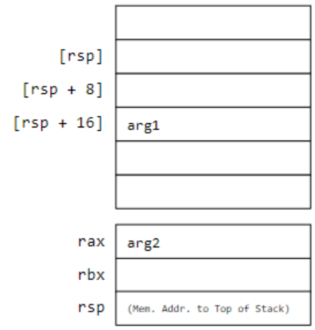
- Before we call `sub rsp, {offset}`, i.e., in diagram (A),
  - `[rsp]` is pointing to some initial return pointer (e.g., at a main expression or some function).
  - `[rsp - curr_word]` is where we stored `arg1` in the stack. Equivalently, `arg1` is stored `curr_word` space "above" `rsp`.
  - Remember that the result of `arg2` is stored in `rax`, since that was the last expression that was compiled.
- After we call `sub rsp, {offset}`, i.e., in diagram (B),
  - We moved `[rsp]` up by `(si * 8) + (2 * 8)` spaces, as seen by the two "jumps" in the diagram.

Our goal is to put `rdi` into `[rsp + 16]`, `arg2` into `[rsp + 8]`, and `arg1` into `[rsp]`. To make this happen, we need to move a few things around. That's where the following four assembly instructions,

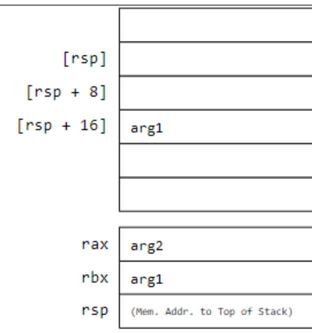
```
mov rbx, [rsp+16]
mov [rsp], rbx
mov [rsp+8], rax
mov [rsp+16], rdi
```

come from. To see how this works, let's visualize each line.

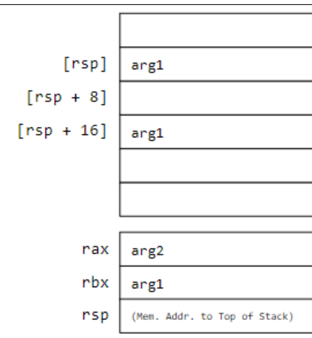
| After Running | Diagram |
|---|---|



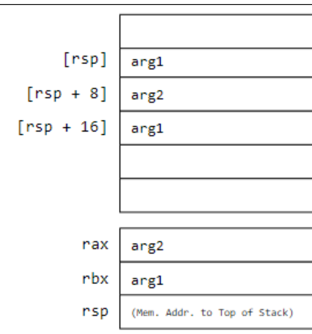

(Initial)



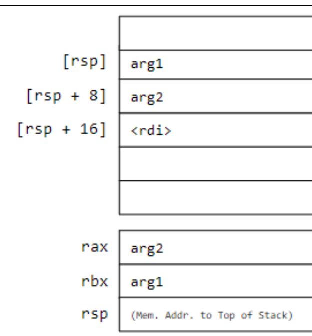

`mov rbx, [rsp + 16]`




`mov [rsp], rbx`




`mov [rsp+8], rax`




`mov, [rsp+16], rdi`

**Remarks:**

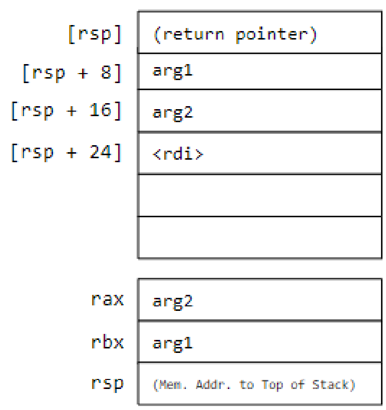
- Remember that the above only works for 2 arguments. For arbitrary arguments, the same idea still holds, but you need to generalize it.

  > Recall: *Before* we executed `sub rsp, ...`, we can imagine that `arg1` is at `[rsp - si * 8]`, `arg2` is at `[rsp - si * 8 - 8]`, `arg3` is at `[rsp - si * 8 - 16]`, and so on until `argN-1` is at `[rsp - si * 8 - 8(N - 1)]`. As usual, `rax` will hold `argN`.

  After we move `[rsp]`, we can expect `argN-1` to be at `[rsp + 16]`, `argN-2` to be at `[rsp + 24]`, and so on, with `arg1` being at `[rsp + 8(N + 1)]`. As usual, `rax` holds `argN`.

- `rdi` is a caller-saved register, hence why we're purposely saving it.
- In this calling convention, as one might have guessed, we're putting everything on the stack.
- While function arguments need positive offsets from `rsp`, **local variables** (temporaries) still use negative offsets from `rsp`.

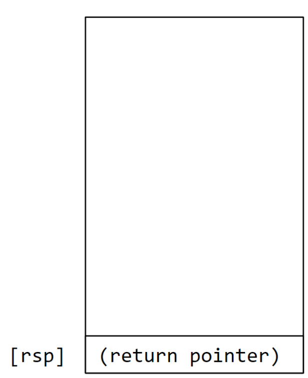
Now, after we do the `call` instruction, the memory diagram looks like



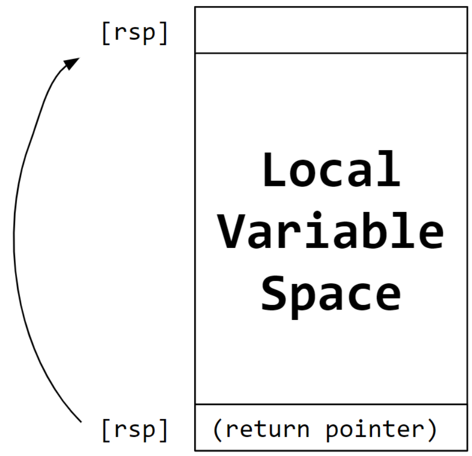

In other words, the `call` instruction will move `rsp` up and write the return pointer into that word. At that point, it's clear that `arg1` and `arg2` are in the correct offsets.

## 1.2 Callee-Managed Stack Pointer

Suppose, instead, we want to have the function itself manage the stack pointer by moving it sufficiently enough. Then, at the start of a function call, suppose the memory diagram looks like this:




Under this assumption where the callee manages the stack pointer, the first thing the function will do is `sub rsp` so that there's enough space for all local variables. That is, we should move `[rsp]` so that we end up with a memory diagram taht looks like




Then, all lookups, including for argument and local variables, will have its location in the stack be positive offsets from `rsp`. There are some advantages to doing this, especially in relation to garbage collection.

### 1.2.1 The Depth Function

First, we need to know how many local variables are needed in the body of a function so we know how much we need to move `rsp` by. We can use the `depth` function to calculate the maximum stack index needed in an expression to store all local variables and temporaries.

Using the power of ChatGPT, along with some corrections, we have the following implementation:

```
fn depth(e: &Expr) -> i32 {
    match e {
        Expr::Num(_) => 0,
        Expr::True => 0,
        Expr::False => 0,
        Expr::Add1(expr) => depth(expr),
        Expr::Plus(expr1, expr2) => depth(expr1).max(depth(expr2) + 1),
        Expr::Let(_, expr1, expr2) => depth(expr1).max(depth(expr2) + 1),
        Expr::Id(_) => 0,
        Expr::Eq(expr1, expr2) => depth(expr1).max(depth(expr2) + 1),
        Expr::If(expr1, expr2, expr3) => {
            depth(expr1).max(depth(expr2)).max(depth(expr3))
        },
        Expr::Loop(expr) => depth(expr),
        Expr::Block(exprs) => exprs.iter().map(|expr| depth(expr)).max().unwrap_or(0),
        Expr::Break(expr) => depth(expr),
        Expr::Print(expr) => depth(expr),
        Expr::Set(_, expr) => depth(expr),
        Expr::Call1(_, expr) => depth(expr),
        Expr::Call2(_, expr1, expr2) => depth(expr1).max(depth(expr2) + 1),
    }
}
```

### 1.2.2 Compiling the Definition

With this in mind, we have

```
    fn compile_definition(d: &Definition, labels: &mut i32) -> String {
        match d {
```

```
            Fun1(name, arg, body) => {
                ...
            }
            Fun2(name, arg1, arg2, body) => {
                let depth = depth(body);
                let offset = depth * 8;
                let body_env = hashmap! {
                    arg1.to_string() => depth + 1,
                    arg2.to_string() => depth + 2
                };
                let body_is = compile_expr(body, 0, &body_env,
                    &String::from(""), labels);
                format!(
                    "{name}:
                    sub rsp, {offset}
                    {body_is}
                    add rsp, {offset}
                    ret"
                )
            }
        }
    }
```

We also need to do the same thing with the main expression (the main program):

```
    sub rsp, {offset}
    {main}
    add rsp, {offset}
```

where `offset` is defined by:

```
    let depth = depth(&p.main); // p.main -> main program
    let offset = depth * 8;
```

### 1.2.3 Changing Offsets in Code

Recall how, before this section, any offsets we used were negative offsets (e.g., `mov [rsp - {offset}], rax`). With this change, we now can use **positive offsets** (e.g., `mov [rsp + {offset}], rax`). This scheme is similar to what most compilers like Rust, `g++`, and so on use.