1 Introduction to Calling Conventions: Print

In the previous section, we discussed throwing errors and how we can call an external (Rust) function from our generated assembly code. Note that the error case is interesting in the sense that it always terminated the program, so we didn't have to worry about coming back to our generated code.

Suppose we want to introduce a `print` expression, which takes an expression and then prints the result out. Like the error handling, the `print` expression will call an external Rust function. However, unlike the error handling, we expect the `print` expression to come back to the generated assembly code *after* calling the external fucntion.

Syntactically, `print` looks like

```
<expr> := ... | input | (print <expr>)
```

1.1 Modifying `start.rs`

Once again, as we need to define a function in Rust to handle printing so our generated assembly code can use it, we need to modify `start.rs` to include this information. Let's define the following function to handle printing:

```
#[no_mangle]
#[export_name = "\01snek_print"]
fn snek_print(val : i64) -> i64 {
    if val == 3 { println!("true"); }
    else if val == 1 { println!("false"); }
    else if val % 2 == 0 { println!("{}", val >> 1); }
    else {
        println!("Unknown value: {}", val);
    }
    return val;
}
```

Note that

- `3` is our representation of `true` (`3` is `0b11` in binary.)
- `1` is our representation of `false` (`1` is `0b01` in binary.)

1.2 Calling Convention Idea

At a high level, when calling a function, we need to do the following:

1. "Remember where" we left off (i.e., recovering the stack).
2. Pass in the appropriate arguments.
3. Call the `snek_print` function.

Regarding "remember where" we left off: remember that our compiler function has a parameter which represents the stack index. This stack index tells us how "deep" we are in the expressions that needed temporary storage/variables. So, whatever stack index we have, we need to set up the call so that it's above the stack index; after the call is done, we need to go back to where the stack index was originally.

1.3 Implementing `print`

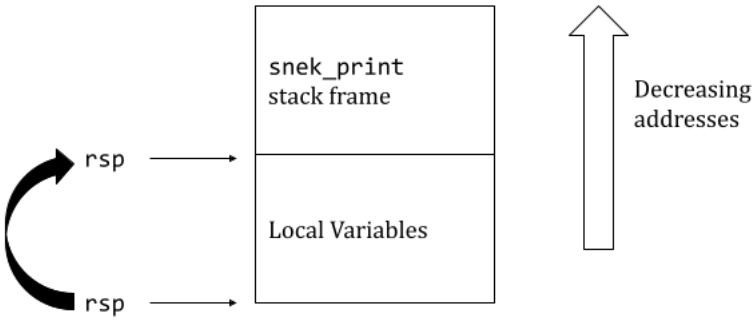
Let's now attempt to implement `print` into our compiler.

1.3.1 Attempt 1

Our initial implementation of `print` might look like

```
match e {
    ...
    | Expr::Id(s) if s == "input" => "mov rax, rdi".to_string(),
    | Expr::Print(val) => {
        let offset = si * 8;
        let v_is = compile_expr(val, si, env, l);
        format!("
            {v_is}
            sub rsp, {offset}
            mov rdi, rax
            call snek_print
            add rdi, {offset}
        ")
    }
}
```

In terms of what's going on, what `sub rsp, {offset}` is doing is moving `rsp`, the stack pointer, so that it's pointing *above* where all the local variables and temporaries are.



So, the stack frame for the `snek_print` is where all the work for this function will happen. So, being its own function, it will rely on `rsp` being "kind of" like at bottom of its stack frame, since it gets its own local variable space whatnot.

(Exercise.) Consider the following code:

```
(block
    (print 37)
    (print input)
)
```

Using our implementation above, what would be printed if `input` was `true`?

This would print `37` twice. To see why, let's consider the generated assembly.

```
mov rax, 74
sub rsp, 16
mov rdi, rax
call snek_print
add rsp, 16
mov rax, rdi
...
```

The issue is in the third line, `mov rdi, rax`. We're overwriting `rdi`, which contains the result of `input`, with a different value! More specifically, we used `rdi` to pass in the argument for `print`, but this means we lose the result of `rdi` when we passed in the argument.

So, we just need to remember to store data from registers somewhere else before we do the call. One thing we can do is store `rdi` somewhere on the stack before we make the function call. To do this, we can make use of `push` and `pop`. In fact, `rdi` is an example of a **caller-saved register**; that is, before we make a function call, we should save this register if we want to restore it afterwards.

1.3.2 Attempt 2

With what we just mentioned in mind, we have

```
match e {
    ...
    | Expr::Id(s) if s == "input" => "mov rax, rdi".to_string(),
    | Expr::Print(val) => {
        let offset = si * 8;
        let v_is = compile_expr(val, si, env, l);
        format!("
            {v_is}
            sub rsp, {offset}
            push rdi            ; added this line
            mov rdi, rax
            call snek_print
            pop rdi             ; added this line
            add rdi, {offset}
        ")
    }
}
```

Remark: Remember that we want `rsp` to point to the spot where the pop is going to be, which is why we call `pop rdi` immediately after the `call` call.

Note that

- `push` pushes a value (register, immediate, etc.) to the stack. `push` will subtract 1 from `rsp`.
- `pop` pops whatever is on top of the stack into a register. `pop` will add 1 to `rsp`.

(Example.) For example,

```
push 17
push 23
pop rax
pop rcx
```

will put `23` into `rax`, and `17` into `rcx`.

So, in terms of what the generated assembly would do, it would

- Push `rsp` into the stack.
- Then, move `rax` into `rsp`.
- Then, call the `snek_print` function.
- Then, put the most recently added value from stack (i.e, the old value of `rsp`) to the register `rsp`.

It just so happens that, in our example here, we only really care about `rdi`. But, if we have other registers that we care about (e.g., we use them a lot or need to save them between expressions), we need to save them.

1.4 Alignment Issues

To summarize some of the things we've said about the x86_64 calling convention[1]:

- Arguments go into `rdi`, and then 6 other registers, and then the stack.
- `[rsp]` (i.e., the value stored at location `rsp`) should be the return pointer.
- 16-byte alignment constraint: `rsp` (mod 16) = 0.

So, in our generated assembly, the third point may cause some problems with the stack pointer. One way we can resolve this is to check what `stack_offset` (mod 16) is, or equivalently `stack_index` (mod 2) is. For the latter, if we have an odd number, we can add 1 to the stack index so we have an even stack index.

[1] These are calling conventions specific to x86_64, and is the reason why we had to do this when calling `snek_print`. However, when we define our own functions in our own language, we can use whatever calling convention we want (including making one up) as long as it's consistent.