

# 1 An Overview of Loops, Set, Break, Input, Blocks, and Errors

In this section, we'll go over a high-level overview of some additional features.

## 1.1 Loops & Break

Loop and break expressions are relatively simple; they look something like

```
(loop <expr>)  
(break <expr>)
```

At its core, a `loop` will repeat an expression over and over again until it encounters a `break`, in which case the loop terminates.

### 1.1.1 Assembly Representation

The assembly representation of loops is straightforward:

```
loop:  
    ; body of loop  
    jmp loop  
done:
```

Here, we defined two labels:

- `loop`, indicating the beginning of the loop.
- `done`, indicating the end of the loop (where we should “break” out).

The idea is that, as long as we aren't breaking out, we will unconditionally jump back to `loop`. If we do want to incorporate a `break` statement, we can add a jump to that label.

### 1.1.2 Labeling

As is the case with `if`-statements, we can have many loops! So, we need to create a unique label for each loop. We can use the `new_label` function from the last section to do this for us.

### 1.1.3 Implementing Break

To implement `break`, the idea is for the compiler to include an additional argument. We can call this argument `loop_label`, which will be an `Option<String>` (recall that an `Option<T>` will either be a `Some(T)` or `None`, indicating some or no value, respectively).

Before we compile the expression associated with the loop, we need to create a unique label. Once we create this label, we can compile the expression, passing that label as our argument for `loop_label`. Because of the recursive nature of the compile function, if we end up inside another loop, compiling its associated expression will result in another label for *that* function call, but not for the current function call. In that sense, we don't need to worry about the possibility of overwriting the break labels.

## 1.2 Set

Set is relatively straightforward: it's analogous to reassignment in most other programming languages. Its syntax looks like

```
(set! <name> <expr>)
```

Here, we're assigning the result of evaluating `<expr>` to the identifier, `<name>`. If the identifier doesn't exist, an error should be thrown.

### 1.3 Blocks

Blocks are just a way of writing more than one statement for an expression. Syntactically, they look like

```
(block <expr>+)
```

In other words, it takes one or more expressions, and then runs each expression in the order that they appear.

Rust	Our Language
<pre>if x &gt; 10 {     x = x + y + z;     y = x - z;     z = z + 5; } else {     x += 10; }</pre>	<pre>(if (&gt; x 10)   (block     (set! x (+ x (+ y z)))     (set! y (- x z))     (set! z (+ z 5)))   (set! x (+ x 10)))</pre>

### 1.4 Handling Inputs

Our programming language is now expected to take a single input, which can either be a number or boolean value. Syntactically, the command is just

```
input
```

In order to handle any input, we need to think about the runtime and, more importantly, the role that `start.rs` plays (recall that `start.rs` is how we call into our assembly code.)

```
fn parse_input(input: &str) -> i64 {
    if input == "true" {
        0b11
    } else if input == "false" {
        0b01
    } else if let Ok(val) = input.parse::<i64>() {
        (val << 1) as u64
    } else {
        panic!("unsupported input: '{}'", input);
    }
}

fn main() {
    let args: Vec<String> = env::args().collect();
    // This is our single input, which by default will be
    // "false" if none are provided
    let input = if args.len() == 2 { &args[1] } else { "false" };
    // Call our assembly code with the given input.
    let i: i64 = unsafe { our_code_starts_here(parse_input(&input)) };
    // Finally, determine what was returned to us.
    if i & 1 == 0 {
        // Number
        println!("{}", i >> 1);
    } else {
        // Boolean
        println!("{}", i >> 1 == 1);
    }
}
```

Note that our input will be in the register `rdi`. So, we can modify the compiler to simply move `rdi` into `rax` before using it.

```
match e {
  ...
  Expr::Id(s) if s == "input" => "mov rax, rdi"
}
```

That's it!

## 1.5 Errors

Recall that, in the previous sections, we didn't want things like `(+ true 1)` to work. This should cause a *runtime error*. So, how do we invoke a runtime error from our assembly code?

- First, we want to create a function in Rust that our assembly code will call.

```
#[export_name = "\x01snek_error"]
pub extern "C" fn snek_error(errcode: i64) {
    eprintln!("error code {errcode} received.");
    std::process::exit(1);
}
```

- Next, in our assembly header, we want to define this function as an external function.

```
section .text
global our_code_starts_here
extern snek_error                ; right here!
```

- From there, we can define a `throw_error` label which will handle calling the Rust code. It will look like

```
throw_error:
    mov rdi, 7                ; rdi is the register representing the first arg
                                ; 7 is some error code we made up
    push rsp
    call snek_error           ; calls the snek_error rust function
    pop rsp
```

We'll worry about the `push` and `pop` instruction later.