1 Structured Data: Pairs (Continued)

In this section, we'll discuss more about structured data, in particular **pairs**.

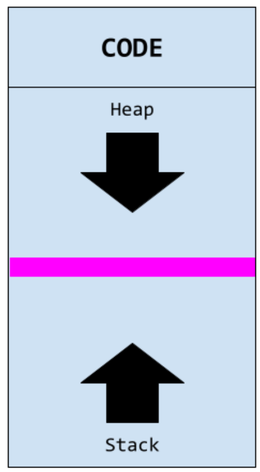
1.1 Heap Allocation

We actually have two things we need to do here:

- We need to create some sort of a *heap* that our compiler can use to store pair values. Once we create
- We need to put the heap pointer into `r15`.

Let's think about some ideas for how we can set the heap up.

- One idea is to set `r15` to `rsp - {a lot}`. In other words, if we move `rsp` very high up, then we can use `r15` as the "heap" pointer. **However**, let's think about the process layout:



  The typical convention is that the heap grows downwards and the stack grows upwards. In around the middle of memory (denoted by the pink rectangle), there's some special addresses where touching them will result in an error (heap overflows and stack overflows). So, if we set `rsp` high enough, then we might either hit that special address, or if we have a depth-heavy recursion[1], or if we somehow point `r15` to the heap space[2], then we'll be in trouble.

- We can also call Rust's equivalent of `malloc`, and use its value[3]. In particular,
  - Call `malloc` for each pair[4], or
  - One *big* `malloc`.

  The suggestion we'll use is similar to having a **big** `malloc`. This has the added benefit of being easy to `free` at the end.

1.2 Modifying Our Rust Code

Now, we need to modify our Rust code to account for these changes.

[1] Since this can hit our proposed "heap" space.
[2] Since this is where Rust might allocate memory, so we might end up overwriting memory that Rust needs.
[3] `malloc` gives us the address to some allocated memory in the heap.
[4] Note that heap size is *unknowable* iin general **statically**.

1.2.1 Modifying the Runtime

In `runtime.rs`, we'll essentially do the following:

```
fn main() {
    let args = ...;
    let input = parse_arg(&args);
    let mut memory = Vec::with_capacity(100_000);       // New!
    let buffer: *mut u64 = memory.as_mut_ptr();         // New!
    let i: i64 = unsafe {
        our_code_starts_here(input, buffer);
    };

    snek_print(i);
}
```

Essentially, we'll create a vector with an initial capaity of `100_000` elements. This will represent our heap. Then, we can get a pointer to that vector, and then pass that pointer into our generated assembly. This means our signature for `our_code_starts_here` will look like:

```
extern "C" {
    #[link_name = "\x01our_code_starts_here"]
    fn our_code_starts_here(input: i64, buffer: *mut u64) -> i64;
}
```

1.2.2 The Generated Assembly

In our generated assembly, we'll have

```
our_code_starts_here:
    mov r15, rsi
    ...
```

Here, `rsi` represents the *second* argument[5].

1.2.3 Printing Values

Finally, we need to adjust the `snek_print` function. In particular, our function now needs to account for the fact that it can either receive

- a number (with tag `0`).
- a boolean (with tag `11`).
- a pair (with tag `01`).
- `nil` (with tag `1`).

With this in mind, we have

```
fn snek_print(val: i64) -> i64 {
    if val == 7 {
        println!("true");
    } else if val == 3 {
        println!("false");
    } else if val % 2 == 0 {
        println!("{}", val >> 1);
    } else if val & 3 == 3 {
```

[5] Recall the x86_64 calling convention.

```
            if val == 1 {
                println!("nil");
            } else {
                let addr: *const u64 = (val - 1) as *const u64;
                snek_print(unsafe { *addr });
                snek_print(unsafe {*addr.offset(1) });
            }
        } else {
            println!("Unknown value: {val}");
        }

        return val;
    }
```

Note that `offset` is used so we don't need to do direct pointer arithmetic[6]. Note that we'll need to do some work printing parentheses and whatnot.

[6] We're looking at the memory location directly next to the current memory location since pairs are contiguous.