1 Structured Data: Pairs

In this section, we'll introduce **structured data** to our programming language. In particular, we'll introduce **pairs**, which is essentially a two-element tuple where both elements can be anything – numbers or pairs.

1.1 Pair Expressions

Our language now has the following syntax:

```
expr := ... | (pair <expr> <expr>) | (fst <expr>) | (snd <expr>) | nil
```

Here, `pair` defines a pair of expressions. `fst` and `snd` returns the first and second element of a pair[1], respectively.

(Exercise.) What will the following program evaluate to?

```
    (fun (inc lst)
        (if (= lst nil)
            nil
            (pair (+ (fst lst) 1) (inc (snd lst)))
        )
    )

(inc (pair 70 (pair 800 nil)))
```

The answer is `(71, (801, nil))`. In this function, we first check if the given pair is `nil`; if it is, return `nil`. Otherwise, we create a new pair where the first expression is just the first element of the original pair incremented by 1, and the second element is the result of recursively calling `inc` on the second element of the original pair.

We can think of the `(snd lst)` as the *rest of the list.*

(Exercise.) What will the following program evaluate to?

```
(fun (sum lst)
    (let (total 0)
        (loop
            (if (= lst nil)
                (break total)
                (block
                    (set! total (+ total (fst lst)))
                    (set! lst (snd lst))
                )
            )
        )
    )
)

(sum (pair 70 (pair 800 nil)))
```

[1] Although `fst` and `snd` takes any expressions, it expects a pair expression.

The answer is 870. This program iterates through each element of the pair, getting its value and adding it to `total`. In particular, if we ran the program, we see that

Expression	(fst lst)	Total
(pair 70 (pair 800 nil))	70	70
(pair 800 nil)	800	870
nil	-	-

1.2 Representing Pairs

Recall that we used a tagging system, where we dedicated one bit, to differentiate numbers and booleans. However, with a new type, we need to rethink the tagging system.

Our tagging system will now consist of the following:

- **Numbers** will still use `0` as its tag value.
- **Booleans** will use `11` as its tag value.
  - `true` will be represented in binary as `111` (7).
  - `false` will be represented in binary as `011` (3).
- **Pairs** will use `01` as its tag value.
- **Nil** will use `1` as its tag value.

With a tagging system in hand, how do we represent pairs themselves? One approach is as follows:

- An idea is to store each of the pair's value as 31-bits. For example, to represent 2 numbers, we would represent the first number as 31 bits, and the next number as another 31 bits, with the tag value being 2 bits.
- **However**, this won't really work if we have nested pairs. For example, if we have a pair with pairs as its element, then how do we represent this?

Another thing we can think about is heap allocation.

1.3 Heap Allocation and Compiler Design

As implied, we'll have to allocate pair element on the **heap** (we'll need to work with the Rust runtime for this). So, our representation is that the pair's value will be a 62-bit address on the heap.

How do we know *where* to allocate pair elements on the heap? An idea is to dedicate a register that just keeps track of the current heap location. In our class, we'll use `r15` for our purposes. With this in mind, here are a few assumptions we will be making:

- `r15` is expected to keep growing for now; it's not like `rsp` where it can increase or decrease depending on how stack space is used.
- `r15` only refers to available memory, never used memory.
- `r15` will be 16-byte aligned (it will end with `0000`)

With this in mind, how do we modify our compiler to support pairs? A sketch of an implementation we'll use is as follows:

```
Pair(e1, e2) => {
    let fst = compile_expr(e1, ...);
    let snd = compile_expr(e2 ...);
    // e1 will be somewhere in [rsp], e2 in rax
    format!("
        {fst}
        {snd}
        mov [r15 + 8], rax
        mov rbx, [rsp + offset]
        mov [r15], r15
        mov rax, r15
        add rax, 1
        add r15, 16
    ")
}
```

Remarks:

- We should probably first check and see if we have space left before allocating. We didn't do this part yet.
- Note that we move `rax` into `[r15 + 8]` (and not `[r15]`) because `rax` has the value of the *second* element of the pair, not the first.
- `mov rax, r15` and `add rax, 1` is designed to put the location in heap of the pair's values into `rax` and then add 1 to `rax` for tagging purposes. Note that we can add `1` to `rax` like this because we know that `r15` will end with `0000` (this is one of the assumptions we made).
- `add r15, 16` moves `r15` by 2 words, thus ensuring that it's always pointing to free memory in the heap.

As one might have suspected, once we execute a pair, it should return the memory location to that pair.