1 Structured Data: Pairs (Continued)

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

1.1 Memory Representation

Suppose we have the following program:

```
(pair 5 (pair 6 (pair 7 nil)))
```

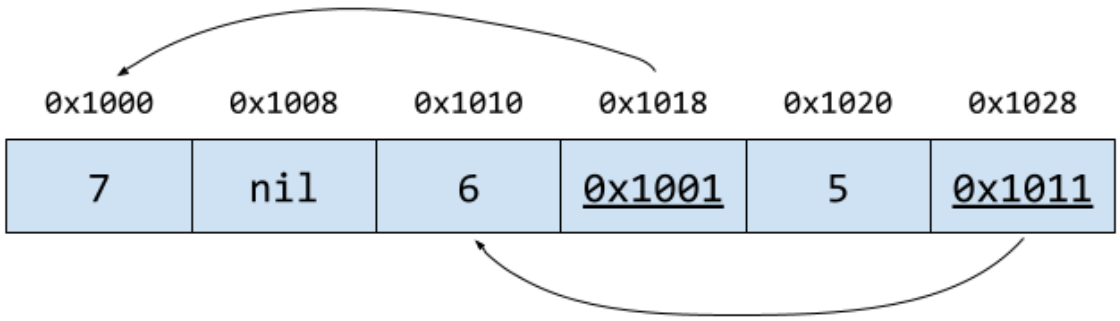
The way we compile this is to first compile the left-most part of the pair (`5`), and then the right-most part of the pair (the rest of the pair, in this case). In particular,

- The instructions for any nested expressions get evaluated before we move those values onto the heap for the current pair.

  > In this program's case, we evaluate `5` first, and then `6`, and then `7`. We put all of them on the stack first. *Then*, we put `7` onto the heap first (while `6` and `5` wait).

- The first pair that gets allocated is the innermost pair.

If we assume left-to-right evaluation order, then `7` goes on the heap first. In this sense, allocation happens inside-out. In any case, in the heap, we expect the memory layout to look like



Remarks:

- We have `0x1001` and `0x1011` (instead of `0x1000` and `0x1010`) because we use `0x01` as the tag value for the pair.
- Note that, in our implementation of the compiler, we're actually going to store the integer multiplied by 2 (e.g., 14, 12, and 10, respectively), since in memory we're storing the *tagged* value. In this example, we're just showing the integers as is (with no tagging).
- This program would evaluate to `0x1021`, since this is the memory address to the first element in the pair.

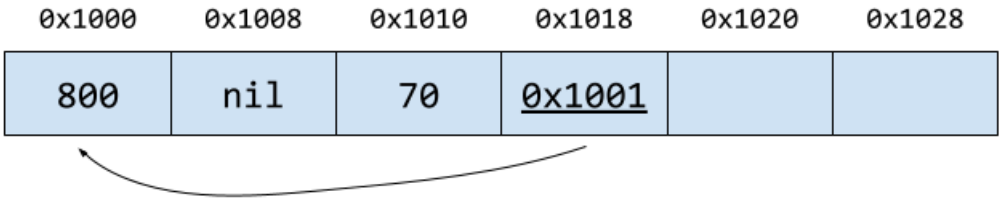
> Another way to think about this is as follows: if we wanted to translate this program into something like Python, syntatically this would look like
>
> ```
> p1 = (7, nil);
> p2 = (6, p1);
> p3 = (5. p2);
> ```
>
> We had to *allocate* memory for `p1` (the innermost pair), and then allocate memory for `p2`, and then finally for `p3`.

Let's now suppose we have the following program:

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

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

The heap diagram might look something like



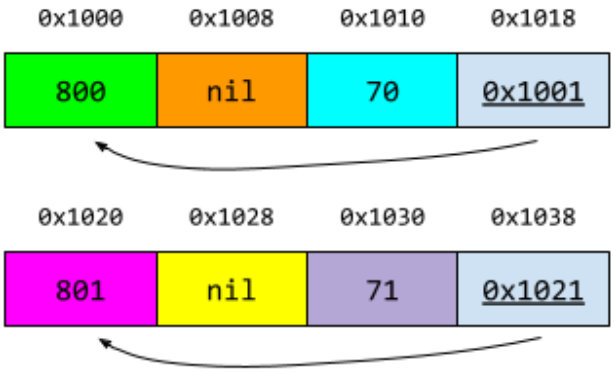
Here, the result of `(pair 70 (pair 800 nil))` is `0x1011`, so `0x1011` is passed into the `inc` function. If we look at the function itself, we can see that the function itself will return the same pair, pair, nil structure.

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

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

Result

```
(pair 71 (pair 801 nil))
```



And, in this exmaple. this function would return `0x1031`, the address to the newly created pair.

1.2 Equality

Let's consider the following program:

```
(let (point1 (pair 6 5))
    (let (point2 (pair 6 5))
        (block
            (print (= point1 point1))                    // A
            (print (= point1 point2))                    // B
            (let (pointpair1 (pair point1 point2))
                (let (pointpair2 (pair point1 point2))
                    (block
                        (print (= pointpair2 pointpair2))   // C
                        (print (= pointpair1 pointpair2))   // D
```

```
                    )
                )
            )
        )
    )
)
```

We now need to decide what this program should print. More specifically, however, we need to figure out how equality of pairs will work. This introduces two types of equalities:

- **Reference equality:** are the two operands referring to the same memory address? For example, in Python, this is `==`.
- **Structural equality:** are the two operands equal when considering their structures? For example, in Python, this is `is`.

With this in mind, we have

Statement	Reference Equality	Structural Equality
A	`true`	`true`
B	`false`	`true`
C	`true`	`true`
D	`false`	`true`

Note statements (C) and (D); if we want structural equality, we probably want to do *recursive structural equality!*