1 Structured Data: Pairs (Continued)

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

1.1 Revisiting Print

Let's suppose we have the following program:

```
(let (p (pair 1 2))
    (block
        (setfst! p p)
        (print p)
    )
)
```

One thing we should note is that we have a **cycle** in the sense that the pair is referring to itself. Therefore, if we tried to `print` the pair, we would end up with infinite recursion since we would constantly recurse through the first element of the pair.

To fix this, we should consider checking if we've *seen* the pair before. If we've seen it, we can print something indicating that a cycle is detected. Otherwise, we can print out the pair as normal.

```
fn snek_str(val: i64, seen: &mut Vec<i64>) -> String {
    if val == 7 { "true".to_owned()}
    else if val == 3 { "false".to_owned() }
    else if val % 2 == 0 { format!("{}", val >> 1) }
    else if val == 1 { "nil".to_owned() }
    else if val & 1 == 1 {
        if seen.contains(&val) { return "...".to_owned() }
        seen.push(val);
        let addr = (val - 1) as *const i64;
        let fst = unsafe { *addr };
        let snd = unsafe { *addr.offset(1) };
        let v = format!("(pair {} {})", snek_str(fst), snek_str(snd));
        seen.pop();
        v
    } else { format!("unknown value: {val}") }
}

#[export_name = "\x01snek_print"]
fn snek_print(val: i64) -> i64 {
    println!("{}", snek_str(val, &mut vec![]));
    val
}
```

1.2 A Brief Sketch of Equality

Given two pairs, how can we check if they are equal? We can use the following Rust implementation,

```
fn snek_eq_helper(val1: i64, val2: i64, seen: &mut Vec<(i64, i64)>) -> bool {
    if seen.contains(&(val1, val2)) { return true }

    seen.push((val1, val2));
    // continue on.
}
```

The idea is that if we come across the same two cycles, we can assume that they're equal and return. Otherwise, we can evaluate the pairs as usual.