Egg Eater

Egg Eater is an extension of previous Snek languages (currently extended from Diamondback) that supports the usage of arbitrarily-sized tuples. The tuple's values can not only be accessed, but also updated, whether available in a let binding or from any arbitrary expression that evaluates to an existing and valid tuple. Tuples in Egg Eater are 0-indexed.

Concrete Grammar

```
<prog> := <defn>* <expr>
<defn> := (fun (<name> <name>*) <expr>)
<expr> :=
  | <number>
  | true
  | false
  | nil
  | input
  | <identifier>
  | (let (<binding>+) <expr>)
  | (<op1> <expr>)
  | (<op2> <expr> <expr>)
  | (set! <name> <expr>)
  | (update! <expr> <expr> <expr>) (new!)
  | (tuple <expr>+)                (new!)
  | (if <expr> <expr> <expr>)
  | (block <expr>+)
  | (loop <expr>)
  | (break <expr>)
  | (<name> <expr>*)

<op1> := add1 | sub1 | isnum | isbool | print
<op2> := + | - | * | < | > | >= | <= | = | index (new!) | == (new!)

<binding> := (<identifier> <expr>)
```

Heap Layout

Each tuple is stored in a heap allocated in the Rust runtime of the language. Tuples have the following internal structure:

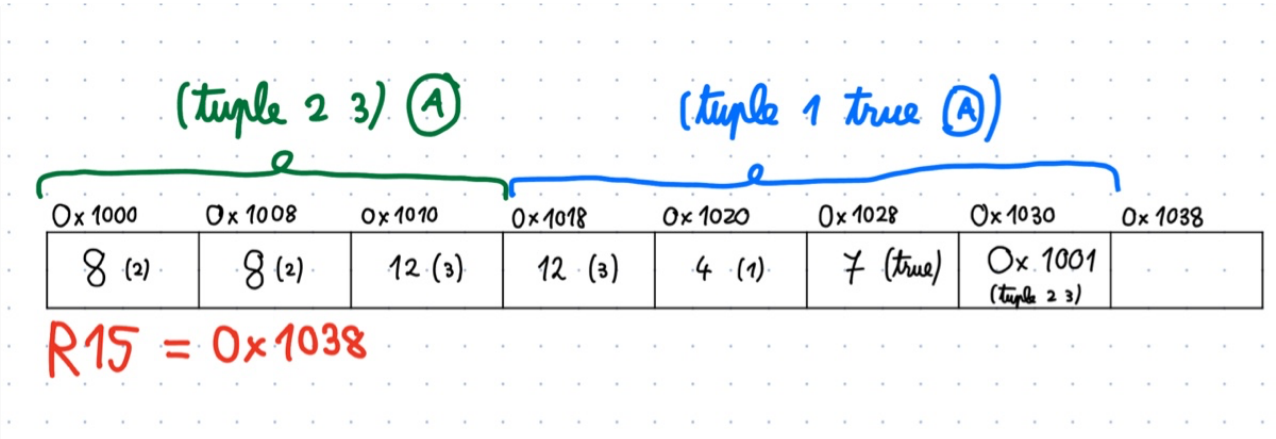
- one quad-word dedicated to the size of the tuple, called `n`
- `n` quad-words for each value in tuple afterwards, written as an Egg Eater number.

Values are laid in the heap depending on their allocation time, so typically inner-most tuples are placed in the heap first. Additionally, the value of the tuple stored as a variable is the address of the tuple in the heap with the type tag 1 added to it. Note that this causes changes in our languages value representation:

- Numbers are represented with type tag `0b00` and are reduced to the 62-bit integer range of numbers.

- Booleans are represented with type tag 0b11, where `0b111 (7)` is `true` and `0b011 (3)` is `false`.

Below is the memory layout of tuples on the heap, assuming the heap address starts initially at `0x1000` and that the executed program is `(tuple 1 true (tuple 2 3))`. All values in memory are listed as present in memory, with their Snek interpretations in parentheses.



Tests

The following tests were run on Egg Eater.

Simple Example of Tuple Usage (`simple_examples.snek`

```
(let ((tuple1 (tuple 1)) (tuple2 (tuple (+ 1 3) (= 3 3) (add1 6))))
(block
  (print (index tuple1 0))
  (print (index tuple2 1))
  (index tuple2 2)
))
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/simple_examples.run
1
true
7
```

Note that our tuple definition evaluates the innermost expressions first before storing them in the tuple. Additionally, tuple can be any arbitrary size, including 1-element tuples.

Tag Checking for Tuple operations (`error-tag.snek`)

```
(index 0 1)
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/error-tag.run
invalid argument - not a tuple: 0x4
```

The program throws a runtime error due to attempting to run an operation intended for tuples on a number, which is an incorrect operation.

Bounds Checking for Tuple operations (`error-bounds.snek`)

```
(index (tuple 1 2 3) 3)
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/error-bounds.run
out of bounds tuple indexing error - tried to index at 3
```

The program throws a runtime error due to attempting to index outside of the tuple's size. We can easily tell that the Tuple size is 3 elements, and since tuples are 0-indexed, this is considered out of bounds (2 is the maximum index for this tuple).

No Arguments in Tuple Construction (`error3.snek`)

```
(+ (tuple) 5)
```

The output of this program in our implementation of Egg Eater is the following:

```
$ make tests/error3.run
cargo run -- tests/error3.snek tests/error3.s
   Compiling boa v0.1.0 (/Users/ghost/Repos/github.com/StormFireFox1/CSE231)
    Finished dev [unoptimized + debuginfo] target(s) in 2.18s
     Running `target/debug/boa tests/error3.snek tests/error3.s`
thread 'main' panicked at 'Invalid expression provided', src/compiler.rs:251:21
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
make: *** [tests/error3.s] Error 101
```

The compiler catches this as a parsing error, since tuples must have at least one element in them, according to our concrete grammar.

Points Implementation (`points.snek`)

```
(fun (new_point x y) (tuple x y))

(fun (add_points p1 p2)
  (new_point (+ (index p1 0) (index p2 0)) (+ (index p1 1) (index p2 1)))
)

(block
    (print (new_point 1 2))
    (let ((x (new_point 2 3)) (y (new_point 5 6))) (add_points x y))
)
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/points.run
(tuple 1 2)
(tuple 7 9)
```

We can see that Tuples can be used in functions, and can also be used in order to produce more complex operations. In this case, we can see the potential of creating structured data, such as Points, and combining them together for various operations. Here we also observe the printing functionality of our runtime being able to handle tuples.

Binary Search Trees (`bst.snek`)

```
(fun (new_bst init) (tuple init nil nil))
(fun (insert_bst tree x)
 (if (= tree nil) (tuple x nil nil)
  (if (< (index tree 0) x) (update! tree 2 (insert_bst (index tree 2) x))
   (if (> (index tree 0) x) (update! tree 1 (insert_bst (index tree 1) x)) tree))))

(fun (in_bst tree x)
 (if (= tree nil) false
  (if (= (index tree 0) x) true
   (if (< (index tree 0) x) (in_bst (index tree 2) x)
     (in_bst (index tree 1) x)))))

(let ((x (new_bst 5))) (block
  (print (insert_bst x 2))
  (print (insert_bst x 6))
  (print (in_bst x 2))
  (print (in_bst x 1))
  (print (insert_bst x 1))
  (in_bst x 1)
))
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/bst.run
(tuple 5 (tuple 2 nil nil) nil)
(tuple 5 (tuple 2 nil nil) (tuple 6 nil nil))
true
false
(tuple 5 (tuple 2 (tuple 1 nil nil) nil) (tuple 6 nil nil))
true
```

Here we can see that the implementation of BST's works perfectly. We can add elements to the same BST in our variables and thus can save complex long-lived structured data. We can also traverse the BST using recursive functions and update it accordingly, thanks to our `update!` expression.

Comparisons to Other Languages' Heap Usage

Languages that use heap-allocated data include Rust and Java.

Rust, in particular, effectively abstracts most of the overhead for heap-allocated values away by simply storing the address of a heap value in the stack's location, rather than the entire used memory of the object. Since the language is statically typed, the compiler already knows what the contents' size in the heap is, and thus does not need additional overhead to check for the length of the heap-allocated value, which is different from what Egg Eater does. Additionally, not all values are heap-allocated in Rust, including structs; in fact, values must be explicitly declared as heap-allocated in Rust.

Java allocates almost everything to the heap and allows a garbage collector to handle heap-allocated values. This requires all heap-allocated values to maintain metadata along with their information, and Java maintains two important words for metadata; the klass word and the mark word. The mark word is used to manage garbage collection metadata, whereas the klass word contains information about the object stored at the address. This makes Egg Eater more akin to Java than Rust, as we also store metadata for our tuples, similar to the klass word used in Java objects stored in the heap.

Extension - Structural Equality and Updating

Updating

Updating tuples is allowed in Egg Eater, with the intention of updating already-existing tuples' values to another. Note that the length of a tuple cannot be changed after creation, regardless of updating. As such, `update!` will error in case an out-of-bounds index is provided.

Updating is done by checking whether our index is out-of-bounds and afterwards taking the Snek number representation, henceforth listed as `length`, converting it to an index we can use to properly access the correct memory location (`((length >> 2) + 1) * 😎` and then eventually moving the calculated value to the memory address calculated using the provided tuple address (with its type tag removed by subtracting 1):

```
result.push(Instr::IMov(Val::Reg(Reg::RCX), Val::Reg(Reg::RAX)));
result.append(&mut vec![
    // Check to see if the index is out of bounds. Has to be bigger than 0,
    // but less than the size of the tuple.
    // First compare to 0 and if it is lower, throw an out of bounds error.
    Instr::IMov(Val::Reg(Reg::RAX), Val::Imm(0)),
    Instr::ICmp(index.clone(), Val::Reg(Reg::RAX)),
    Instr::IJl(Val::Label("out_of_bounds_err".to_string())),
    // Afterwards, compare the index to the size of the tuple from memory.
    // If it is greater than or equal to the size, throw an out of bounds error.
    Instr::IMov(Val::Reg(Reg::RBX), tuple.clone()),
    // Also check if the tuple is nil, because that's a null pointer exception.
    Instr::ICmp(Val::Reg(Reg::RBX), Val::Imm(1)),
    Instr::IJe(Val::Label("null_ptr_err".to_string())),
    Instr::ISub(Val::Reg(Reg::RBX), Val::Imm(1)),
    Instr::IMov(Val::Reg(Reg::RAX), Val::RegOffset(Reg::RBX, 0)),
    Instr::ICmp(index.clone(), Val::Reg(Reg::RAX)),
    Instr::IJge(Val::Label("out_of_bounds_err".to_string())),
    Instr::IMov(Val::Reg(Reg::RAX), index),
    Instr::ISar(Val::Reg(Reg::RAX), Val::Imm(2)),
    Instr::IAdd(Val::Reg(Reg::RAX), Val::Imm(1)),
    Instr::IMul(Val::Reg(Reg::RAX), Val::Imm(8)),
    Instr::IJo(Val::Label("overflow_err".to_string())),
    Instr::IAdd(Val::Reg(Reg::RBX), Val::Reg(Reg::RAX)),
    Instr::IJo(Val::Label("overflow_err".to_string())),
    Instr::IMov(Val::RegOffset(Reg::RBX, 0), Val::Reg(Reg::RCX)),
    Instr::IMov(Val::Reg(Reg::RAX), tuple),
]);
// --->8---
// rest of compile_expr snipped
```

Update expressions are parsed as a ternary operation:

```
[Sexp::Atom(S(op)), e1, e2, e3] if op == "update!" => {
    Expr::Update(Box::new(parse_expr(e1)), Box::new(parse_expr(e2)),
Box::new(parse_expr(e3)))
}
// --->8---
// rest of parse_expr snipped
```

Structural Equality

Structural equality among tuples, at its essence, revolves around the idea that tuples reside in different places in memory due to our allocation procedure, regardless of whether the data contained within the tuple is identical with another. Therefore, typical operators like `=` will not yield `true`, despite two arbitrary tuples potentially containing the same data (this, of course, excludes the case where one compares one tuple with itself, in which case `(= x x)` is vacuously true).

Therefore, one must implement a special case of structural equality where each individual element in a tuple is checked for equality. The implementation here resides under the Rust runtime, and as such, we pass the two tuples we are structurally comparing as arguments to a Rust function.

We only allow tuples to be compared for structural equality, and as such any other value type passed to `==` is expressly rejected as an invalid argument and an error will be thrown. In the snippet below, the implementation in assembly of checking for structural equality from `compile_expr` is presented. We simply pass the arguments to the Rust function after checking for their type:

```
Op2::TupleEqual => {
    result.append(&mut vec![
        // Before that, check if both arguments are tuples. If not, throw the "Invalid
argument error".
        Instr::IMov(Val::Reg(Reg::RBX), Val::Reg(Reg::RAX)),
        Instr::IMov(Val::Reg(Reg::RCX), Val::RegOffset(Reg::RBP, si * 8)),
        Instr::IAnd(Val::Reg(Reg::RBX), Val::Imm(3)),
        Instr::ICmp(Val::Reg(Reg::RBX), Val::Imm(1)),
        Instr::IJne(Val::Label("invalid_arg_err".to_string())),
        Instr::IAnd(Val::Reg(Reg::RCX), Val::Imm(3)),
        Instr::ICmp(Val::Reg(Reg::RCX), Val::Imm(1)),
        Instr::IJne(Val::Label("invalid_arg_err".to_string())),
        // Pass the two tuples over to the Rust runtime to evaluate equality.
        Instr::IMov(Val::Reg(Reg::RDI), Val::Reg(Reg::RAX)),
        Instr::IMov(Val::Reg(Reg::RSI), Val::RegOffset(Reg::RBP, si * 8)),
        Instr::ICall(Val::Label("snek_tuple_equal".to_string())),
    ])
},
// --->8---
// rest of compile_expr snipped
```

We parse structural equality expressions as a binary operation expression with the `==` operator in the `parse_expr` function:

```
[Sexp::Atom(S(op)), e1, e2] if op == "==" => Expr::BinOp(
    Op2::TupleEqual,
    Box::new(parse_expr(e1)),
    Box::new(parse_expr(e2)),
),
// --->8---
// rest of parse_expr snipped
```

With regards to the Rust runtime function, we use a recursive implementation. For the two tuples we first check whether their lengths are equal. No tuples can be structurally identical without equal lengths. Next, we check whether all of their elements at each index are equal, with the exception of any tuple values. Note the early-return paradigm:

```
let FALSE = 3;
let TRUE = 7;
// We are guaranteed the provided values are tuples here, so we need to just check for
equality
// on tuples, not anything else.
// First, check if length is equal. If that's not equal, then nothing is.
let addr_1 = (tuple_1 - 1) as *const i64;
let addr_2 = (tuple_2 - 1) as *const i64;
let len_1 = unsafe { *addr_1 } >> 2;
let len_2 = unsafe { *addr_2 } >> 2;
if len_1 != len_2 {
    return FALSE
}

let mut tuple_vec_1: Vec<i64> = Vec::new();
let mut tuple_vec_2: Vec<i64> = Vec::new();
let mut pairs_of_tuples_to_check: Vec<(i64, i64)> = Vec::new();
// Now iterate through each value in both tuples.
for index in 1..=len_1 {
    let val_1 = unsafe { *addr_1.offset(index.try_into().expect("runtime tuple
structural equality error")) };
    let val_2 = unsafe { *addr_2.offset(index.try_into().expect("runtime tuple
structural equality error")) };
    tuple_vec_1.push(val_1);
    tuple_vec_2.push(val_2);
    // Let's get the simple cases out.
    // If the type tags differ, they're different.
    if val_1 & 0b11 != val_2 & 0b11 {
        return FALSE
    }
    // At this point, the type tags are identical.
    // If the two elements are NOT tuples, check for simple equality.
    if val_1 & 0b11 != 1 {
        if val_1 != val_2 {
            return FALSE
        }
// --->8---
```

In the case of tuple values, we store for later processing.

```
    if val_1 & 0b11 == 1 {
        pairs_of_tuples_to_check.push((val_1, val_2));
    }
```

In order to address potential cyclical tuples, we save the currently processed tuple into a set that keeps track of previously calculated tuples. Then, we check if any of the found sub-tuples of the current one also follow the structural equality property. If we do not fail up to this point, we simply return `true` in Snek representation.

```
// Otherwise, add our current tuple to the set of traversed values and recurse with the
pairs of tuples
// we have to check. If any fail, our entire comparison fails.
traversed_values.insert((tuple_vec_1, tuple_vec_2));
for pair in pairs_of_tuples_to_check {
    if snek_compare_tuples(pair.0, pair.1, traversed_values) == FALSE {
        return FALSE
    }
}

return TRUE
// --->8---
```

In the case that our tuples happen to be cyclical, eventually they will reach a point where we will be evaluating a previously checked pair of tuples in our computation. If that happens, that must mean that our original tuple is not only cyclical, but follows the same path in terms of structurally equal values as the other, and thus is still equal regardless of its cyclical nature:

```
// At this point, the tuples seem equal. We'll want to check if we've done this
comparison before.
// If we have, then that must mean we're reached a cyclic position in our structure and
thus can easily
// return TRUE without worries.
if traversed_values.contains(&(tuple_vec_1.clone(), tuple_vec_2.clone())) {
    return TRUE
}
```

Note that we use the definition of “observational equality” mentioned in an Edstem post when implementing our definition. Functionally, what we are doing here is checking whether any of the sub-tuples have structurally the same values and eventually “go back” to a previously computed structurally equal value from our cycle. Therefore, indexing from our program would effectively follow the way the above runtime function interprets the tuple, and as such would be structurally equal in the eyes of any user of Egg Eater.

This property holds despite tuples being potentially different in their internal representations, as we’ll see in some demonstrated tests.

Tests & Demonstrations

Structural And Reference Equality on Non-Cyclic Tuple Values (`equal.snek`)

```
(let
((x (tuple 1 true 2))
(y (tuple 1 true 2))
(z (tuple 3 false (tuple true 6 7)))
(t (tuple 3 false (tuple true 6 7)))
(v x))
(block
    (print (= x x))
    (print (== x x))
    (print (= x y))
    (print (== x y))
    (print (= z t))
    (print (== z t))
    (print (= x v))
    (== (tuple 1 2 3 4) (tuple 1 2 3))
))
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/equal.run
true
true
false
true
false
true
true
false
```

Reference equality is demonstrated, as the only reference equality checks that yield `true` are those when tuples are compared with themselves (whether `x` with `x` or `v` with `x` where `v` is actually bound to `x`). However, note that for all other operations, structural equalities only yield `true` when all elements internally are equal. This works for all element types, including tuples, as each tuple element from the original tuples has their own sub-elements evaluated as per the above attached code. Note that tuples of different lengths also cannot be equal.

Simple Cyclical Tuple Structural Equality (`cycle-equal1.snek`)

```
(let ((x (tuple 1 2 3)) (y (tuple 1 2 3))) (block
    (update! x 1 x)
    (update! y 1 y)
    (print (= x y))
    (== x y)
))
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/cycle-equal1.run
false
true
```

A reference equality check obviously yields `false` but as you can see, since each tuple has itself as the value at index `1` and all of the other elements are equal with one another at their respective indices.

Circular List Structural Equality (`cycle-equal2.snek`)

```
(let
((x (tuple 1 (tuple 2 (tuple 3 nil))))
 (y (tuple 1 (tuple 2 (tuple 3 nil)))))
(block
    (update! (index (index x 1) 1) 1 x)
    (update! (index (index y 1) 1) 1 y)
    (print (= x y))
    (== x y)
))
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/cycle-equal2.run
false
true
```

Both tuples are circular `cons` lists, as they eventually have their last element point to the head tuple in the list. As both these circular `cons` lists start at 1 and effectively represented as `(1 (2 (3 (1 (2 (3 (...` they are both structurally equal and our operator accordingly returns `true` .

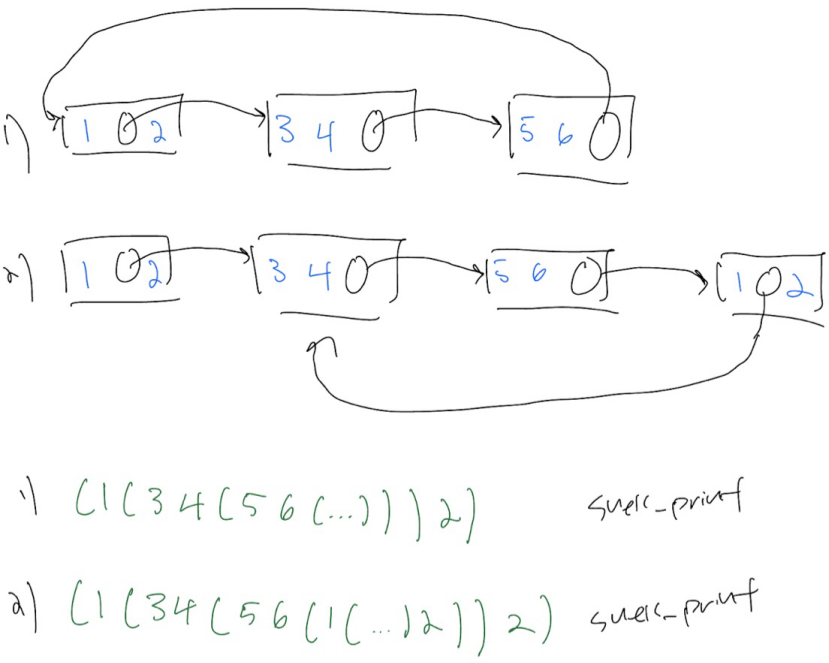
Structural Equality with Differing Internal Tuples (`cycle-equal3.snek`)

```
(let
((x (tuple 1 (tuple 3 4 (tuple 5 6 nil)) 2))
(y (tuple 1 (tuple 3 4 (tuple 5 6 (tuple 1 nil 2))) 2)))
(block
    (update! (index (index x 1) 2) 2 x)
    (update! (index (index (index y 1) 2) 2) 1 (index y 1))
    (print (= x y))
    (== x y)
))
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/cycle-equal3.run
false
true
```

This program effectively tests the tuples discussed on this Ed post where one may find the structural equality of these two tuples ambiguous. Their representations are attached below, as per the Ed post:



The idea is that, despite tuples 1 and 2 having different representations in memory, by virtue of observational equality, they are effectively identical in structure: indexing in them will yield the same value no matter what indices one takes in each. Our code accounts for this and correctly evaluates this as structurally equal.

Printing Simple Cyclical Tuples (`cycle-print1.snek`)

```
(let ((x (tuple 1 2 3))) (block
    (update! x 1 x)
    x
))
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/cycle-print1.run
(tuple 1 (tuple ...) 3)
```

Since the second element in the tuple links to `x` , we have a cycle that is properly displayed.

Printing Simple Cyclical Tuples (`cycle-print2.snek`)

```
(let ((x (tuple 1 (tuple 3 (tuple true nil nil) 4) 2))) (block
    (update! (index (index x 1) 1) 1 (index x 1))
    (update! (index (index x 1) 1) 2 x)
    x
))
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/cycle-print2.run
(tuple 1 (tuple 3 (tuple true (tuple ...) (tuple ...)) 4) 2)
```

Here, the inner-most tuple has two `nil` elements that are replaced; the first one is replaced with a pointer to the second inner-most tuple, whereas the second one is replaced with a pointer to the outermost tuple. In either case, regardless of recursion, both are displayed as cyclical references.

Cyclical Tuples Across Two Let Bindings (`cycle-print3.snek`)

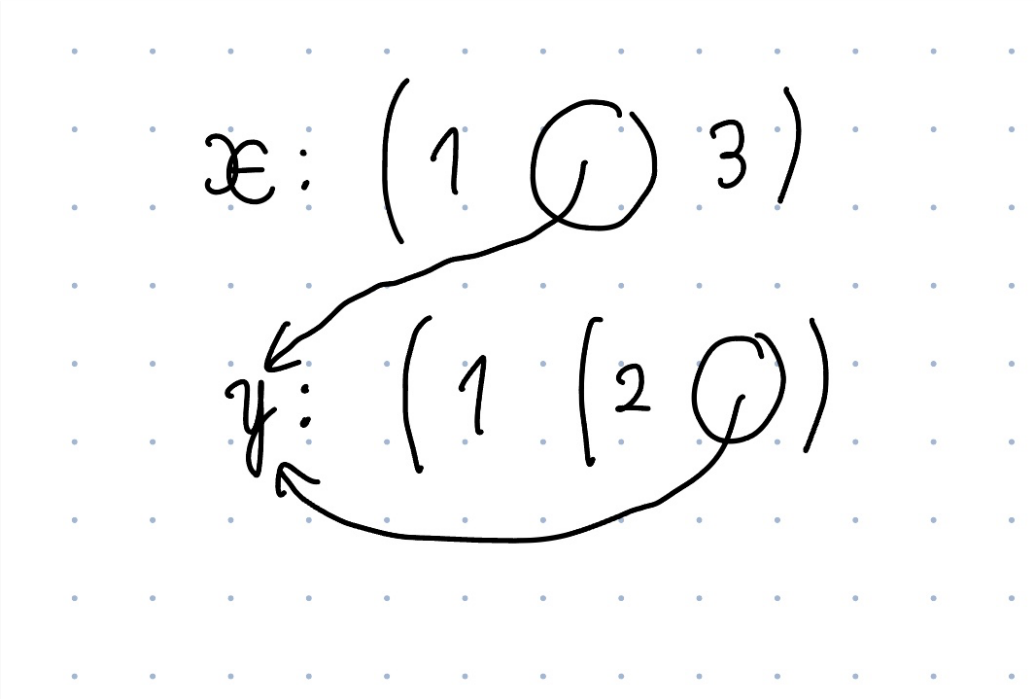
```
(let ((x (tuple 1 nil 3)) (y (tuple 1 (tuple 2 nil)))) (block
    (update! x 1 y)
    (update! (index y 1) 1 y)
    x
))
```

The output of this program in our implementation of Egg Eater is the following:

```
$ ./tests/cycle-print3.run
(tuple 1 (tuple 1 (tuple 2 (tuple ...))) 3)
```

Here, the cycle occurs in a different tuple as opposed to the one we are printing, albeit in one of `x` 's elements. As displayed, the cycle occurs regardless, since `y` had already been encountered within `x` . Therefore, cyclical printing works no matter where the cycle is located, whether in the same original tuple allocation or another.

A simpler diagram to see the structure of the above tuple would be the following:



Other Features

No other features were implemented or Egg Eater outside of `update!` and structural equality. Note, however, that previously `update!` would not be able to update arbitrary tuple expressions, only let bound let bindings.

Credits

I would like to credit Professor Politz for the discussion on printing cyclical tuples, and providing starter code for the implementation of printing tuples from the Rust runtime, present on the GitHub repository for class starter code.

Additionally, I would like to thank Yuchen Jing for our discussion regarding the semantic interpretation of structural equality that made the most sense, along with the aforementioned Edstem post for clarifying the exact required implementation of structural equality boiling down to "observational equality".