Egg Eater

Concrete Syntax

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

<op1> := add1 | sub1 | isnum | isbool | print
<op2> := + | - | * | < | > | >= | <= | =

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

The new additions are the `nil` value, which is now tagged as `0x1` in our internal representation, which makes the representations for true and false now `0x3` and `0x7` respectively. Tuples and in general address references to heap values are tagged with `0b1` in the least significant bit, with the other 63 bits denoting the address in the heap.

The tuple data structure is more accurately an n-tuple, similar to the tuple implementation in python. As such, we can initialize it with any number of elements, being able to act partially as an array.

```
❯ cat tests/tuple1.snek
(tuple 1 nil true)

❯ ./tests/tuple1.run
[1, nil, true]

❯ cat tests/tuple2.snek
(tuple 1 2 3 4 5 6 7 8 9 10)

❯ ./tests/tuple2.run
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

❯ cat tests/tuple_nested.snek
(tuple 5 (tuple 2 1 3) (tuple 8 6 9))

❯ ./tests/tuple_nested.run
[5, [2, 1, 3], [8, 6, 9]]
```

The index method takes an expression that evaluates to a tuple as the first argument, and the index of the element to return as the second argument, using 0 based indexing.

```
❯ cat tests/tuple_index1.snek
(index (tuple 1 2 3 4 5 6 7 8 9 10) 2)

❯ ./tests/tuple_index1.run
3
```

Heap Allocation

```
let mut instrs = args
        .iter()
        // args: Expr -> (offset: i32, instrs: Vec<Instr>)
        .map(|expr| {
            let curr_instrs = compile_to_instrs(
                expr,
                &Context {
                    si: curr_si,
                    env: &ctx.env.clone(),
                    break_target: ctx.break_target.clone(),
                    ..*ctx
                },
            );
            let curr_offset = curr_si * WORD_SIZE;
            curr_si += 1;
            (curr_offset, curr_instrs)
        })
        // (offset: i32, instrs: Vec<Instr>) -> Vec<Instr>
        // Combine instrs for expr with instruction to store result
in designated
        // heap location
        .map(|(offset, instrs)| {
            [
                &instrs[..],
                &[Instr::IMov(Val::RegOffset(RSP, offset),
Val::Reg(RAX))],
            ]
            .concat()
        })
        // turn iterator of Vec<Instr> into one long vector,
essentially concatenating
        // all of the separate Vec<Instr> we made from the previous
map
        .fold(Vec::new(), |accum, instrs: Vec<Instr>| {
            [&accum[..], &instrs].concat()
        });
```

The code snippet above is the first stage of initializing the tuple, where we iterate over all the arguments in the tuple constructor and compile instructions and save the expressions onto the stack.

```
let mut heap_offset = size;
// move stack values onto heap
for i in (ctx.si..curr_si).rev() {
    // Move value to RAX
    instrs.push(Instr::IMov(
        Val::Reg(RAX),
        Val::RegOffset(RSP, i * WORD_SIZE),
    ));
    // Store in corresponding heap location
    instrs.push(Instr::IMov(
        Val::RegOffset(R15, heap_offset * WORD_SIZE),
        Val::Reg(RAX),
    ));
    heap_offset -= 1;
}
```

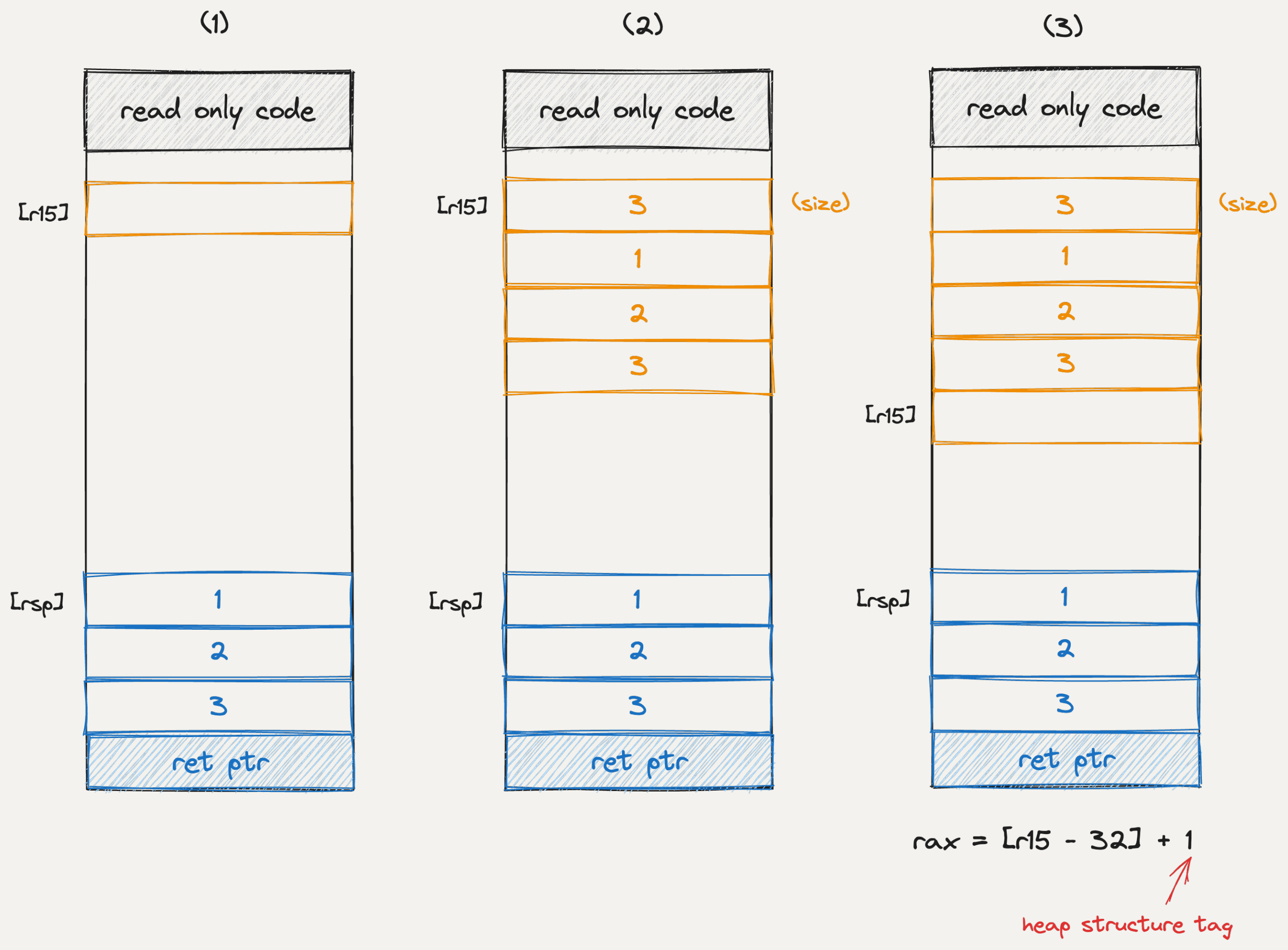
We then calculate the total heap structure size, which in this case would be 1 word for each tuple entry along with an extra word as metadata to denote the size of the tuple. This makes it easier to find out of bounds errors dynamically. Once the size of the heap is known, we iterate over the stored stack variables in reverse order and copy them over to the address pointed to by `r15` with the corresponding offset so that the entries are all laid out contiguous in memory. Finally, we save the size of the tuple is the first word of the structure at `[r15]`.

```
// Store size of tuple in first heap location
instrs.push(Instr::IMov(Val::Reg(RAX), Val::Imm(to_num63(size as
i64))));
instrs.push(Instr::IMov(Val::RegOffset(R15, 0), Val::Reg(RAX)));

// Return R15 to RAX
instrs.push(Instr::IMov(Val::Reg(RAX), Val::Reg(R15)));
// TAG RAX
instrs.push(Instr::IAdd(Val::Reg(RAX), Val::Imm(1)));
// Increment R15
instrs.push(Instr::IAdd(
    Val::Reg(R15),
    Val::Imm(((size + 1) * WORD_SIZE) as i64),
));
```

Finally, as the value returned to `RAX` would be the current address pointed to by `r15` (the start of the struct), and tagging the least significant bit by `1` after which we move the heap address by the size of the structure to point to the next available memory location for the next heap structure.

This also allows us to nest tuples inside each other as each argument in the constructor would be compiled recursively and stored in stack variables initially. If one of the arguments is a heap structure, it would be stored on the stack as a tagged address, and thus once it is copied over to its corresponding entry in the heap, the nested tuple's address is copied over, allowing us to keep the heap structure contiguous in memory and using references to obtain the nested tuple's values.

The following diagram illustrates how `(tuple 1 2 3)` would be mapped out in memory.



We compile the 3 values to initialize the tuple with and save them on the stack. Afterwards, we copy them to the heap in reverse order from `[r15+3]` to `[r15+1]` in reverse order to maintain the elements contiguously in memory. We finally store the size of the tuple to `[r15+0]` as the first word of the structure before saving that address to RAX and tagging the address and finally moving `r15` to `r15 = [r15+32]` to make room for the next heap structure.

Testing

input/simple_examples.snek

```
(block
    (print (tuple 1 2 3 4 5))
    (print (tuple (tuple 1 2 3) 4 5 6 (tuple 7 8) 9))
    (let ((x (tuple 10 20 30 40 50)))
        (index x 1)
    )
)
```

```
❯ ./input/simple_examples.run
[1, 2, 3, 4, 5]
[[1, 2, 3], 4, 5, 6, [7, 8], 9]
20
```

This program aims to demonstrate some simple cases of initializing and accessing values in the tuple. The first line, we print a tuple after initializing it with 5 values. The second line is to test initializing tuples with a nested structure, which prints out the nested tuples with another set of `[]` around them. The final example is demonstrating the ability to save the tuple to a variable and access values from the tuple using the index method on the variable itself.

input/error-tag.snek

```
(index 100 1)
```

```
❯ ./input/error-tag.run
invalid argument
```

This program aims to showcase dynamic errors being thrown when expressions that expect a heap structure is not given one. This is achieved thanks to the tag checking that occurs in the index operation that checks if the two least significant bits of the first expression is `0b01`.

input/error-bounds.snek

```
(index (tuple 1 2 3) -6)
```

```
❯ ./input/error-bounds.run
out of bounds
```

This program aims to show that when an out of bounds index is provided to the index operation, we stop the program with a dynamic error. This is done when either `index < 0` or `index >= size`, where size is the first word stored in the memory address of the tuple.

input/error3.snek

```
(tuple ())
```

```
❯ make input/error3.run
thread 'main' panicked at 'Invalid', src/parser.rs:209:18
```

This program aims to demonstrate a static error that would be generated if we tried to initialize a tuple with no arguments in its constructor. The tuple constructor needs at minimum one argument.

input/points.snek

```
(fun (create_pt x y)
    (tuple x y)
)

(fun (add_pt x y)
    (tuple (+ (index x 0) (index y 0)) (+ (index x 1) (index y 1)))
)

(block
    (print (create_pt 10 12))
    (print (create_pt -5 -10))
    (let (
        (p1 (create_pt 1 5))
        (p2 (create_pt 7 9))
        )

        (block
            (print p1)
            (print p2)
            (add_pt p1 p2)
        )
    )
)
```

```
❯ ./input/points.run
[10, 12]
[-5, -10]
[1, 5]
[7, 9]
[8, 14]
```

In this program, we have defined two functions. The first, `create_pt` takes in an `x` and `y` coordinate as its arguments and creates a new point structure on the heap using a tuple. The second function `add_pt` takes in two point structures as arguments and returns the sum of the two points, by creating a new point who's x coordinate is `p1[0] + p2[0]` and y coordinate is `p1[1] + p2[1]`. We achieve this by using the index operation to get the x and y coordinates for each of the points.

The expressions in the main body are used to test the correct functionality of both the functions and the heap structure implementation. First, we create two points and print them out. One using positive integers and one using negative integers to ensure no bugs are introduced due to the presence of signed integers. Next we test whether we can store the result of calling `create_pt` to a variable by creating two variables `p1` and `p2` and storing the outputs of `create_pt` with different arguments in them. We then print these points to ensure that they have been created successfully and finally test the `add_pt` function by returning the output of adding the previously stored points `p1` and `p2`. Running the program output shows that all of the output is as expected.

It is important to note that in the current state of the functions, it is possible to create points with booleans as well, which would dynamically return an error when trying to add 2 points as shown in the example below. This could be mitigated by adding an isnum operation to each of the arguments in `create_pt` to ensure that only integers are valid arguments to the function.

```
(fun (create_pt x y)
    (tuple x y)
)

(fun (add_pt x y)
    (tuple (+ (index x 0) (index y 0)) (+ (index x 1) (index y 1)))
)

(let (
    (p1 (create_pt true 5))
    (p2 (create_pt 7 9))
    )
    (block
        (print p1)
        (print p2)
        (add_pt p1 p2)
    )
)
```

```
❯ ./input/points-err.run
[true, 5]
[7, 9]
invalid argument
```

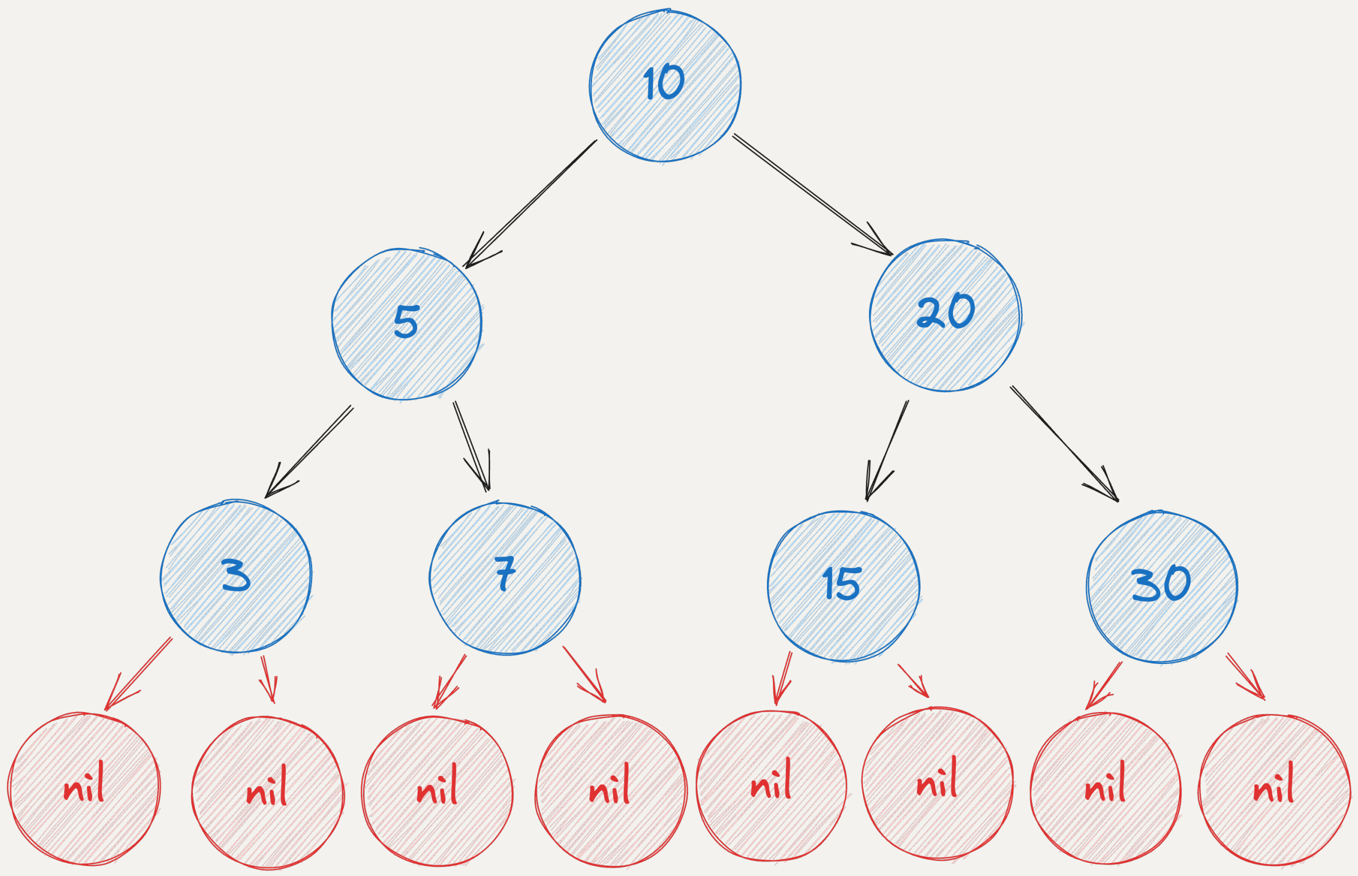
input/bst.snek

```
(fun (bst x)
    (tuple x (tuple nil) (tuple nil))
)

(fun (bst_add tree n)
    (if (= (index tree 0) nil)
        (bst n)

        (block
            (if (< n (index tree 0))
                (tuple (index tree 0) (bst_add (index tree 1) n)
(index tree 2))
                (tuple (index tree 0) (index tree 1) (bst_add (index
tree 2) n))
            )
        )
    )
)

(fun (contains tree n)
    (if (= (index tree 0) nil)
        false

        (if (= (index tree 0) n)
            true

            (block
                (if (< n (index tree 0))
                    (contains (index tree 1) n)
                    (contains (index tree 2) n)
                )
            )
        )
    )
)

(block
    (print (bst 10))
    (let (
        (t (bst 10))
    )

        (block
            (print (set! t (bst_add t 5)))
            (print (set! t (bst_add t 20)))
            (print (set! t (bst_add t 15)))
            (print (set! t (bst_add t 7)))
            (print (set! t (bst_add t 3)))
            (print (set! t (bst_add t 30)))

            (print (contains t 5))
            (print (contains t 12))
            (print (contains t 13))
            (print (contains t 15))
            (print (contains t 20))
            (print (contains t 3))
            (print (contains t 0))
        )
    )
)
```

```
❯ ./input/bst.run
[10, [nil], [nil]]
[10, [5, [nil], [nil]], [nil]]
[10, [5, [nil], [nil]], [20, [nil], [nil]]]
[10, [5, [nil], [nil]], [20, [15, [nil], [nil]], [nil]]]
[10, [5, [nil], [7, [nil], [nil]]], [20, [15, [nil], [nil]], [nil]]]
[10, [5, [3, [nil], [nil]], [7, [nil], [nil]]], [20, [15, [nil],
[nil]], [nil]]]
[10, [5, [3, [nil], [nil]], [7, [nil], [nil]]], [20, [15, [nil],
[nil]], [30, [nil], [nil]]]]
true
false
false
true
true
true
false
false
```

In this program, we implement a binary search tree to test that our heap structure implementation holds for a recursive structure. We have a function `bst` which creates a BST from a single element. The format of our BST is `[node [left_subtree] [right_subtree]]`, Thus we can see from the output of the first line of the main body that calling `bst 10` results in a single node BST `[10, [nil], [nil]]`

The `bst_add` function recursively adds an element to the tree. We have a base case where the tree's node is `nil`, which denotes that we have reached a leaf and thus we return a new BST with the element in it. Otherwise, if the element is less than the current node as obtained using the index operation, we recursively add the element to the left subtree otherwise we add the elemnt to the right subtree. Note that in this implementation we treat a value being equal to an element the same as being greater than it, allowing for duplicates in the BST. We can see the result of this by defining a bst `t` as `[10, [nil], [nil]]`, and adding elements to it as seen in the main function. The final BST tree, according to the printed output can be better visualized as the tree below



As we can see, the output of the program is a valid BST.

We also have a `contains` method that recursively checks whether a given value is present in the BST. We first check for a base case if the current node is `nil`, which indicates that we have reached a leaf and that the value does not exist, hence we return `false`. Otherwise, if the current node is equal to the value in question, we return `true`, else we recursively check the corresponding subtree according to whether the value is less than or greater than the current node. As seen from the test case, we check if the calues 5, 12, 13, 15, 20, 3 and 0 are contained within the BST, of which the program only output `true` for 5, 15, 20 and 3, which is the expected output.

Design Comparison

I will be comparing my design of the n-tuple with Python's tuple, and the closest data structure there is in Javascript Lists. I believe that my design is almost exactly the same as Python's implementation of its tuple since they are both immutable with a fixed length, and also allocate exactly the required amount of memory **in a single block**. Javascript lists on the other hand over allocate memory and is split between **multiple blocks** of memory in the heap. It is also mutable and of variable length that supports dynamic resizing, which is the complete opposite of mine and python's implementation. The one similarity there is across all three implementations is that they support multiple data types in the same list, i.e. the lists/tuples are not required to only have one type of element in it (`(10, true, "hello")` is valid in Python, `["hello", 123, false]` is valid in Javascript, and `[10, false]` is valid in Egg Eater)

Collaboration

A part from reading posts on EdStem, I have only reached out to the professor, just to find out about x86 Assembly syntax for specifying a memory offset value in a register (like `[rsp + rax]`)