Eggeater design

Concrete Syntax

The concrete syntax of Eggeater builds on top of Diamondback by adding operators to create lists, index into and mutate them.

```
<prog> := <defn>* <expr>
<defn> := (fun (<name> <name>*) <expr>)
<expr> :=
  | <number>
  | true
  | false
  | 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>*)
  | nil                                    // Empty list
  | (list (<expr>*))                       // Initialize with given
values
  | (index <expr> <expr>)                  // Index into the list
  | (set! <expr> <expr> <expr>)            // Mutate the list's index
to the value

// The below are of no real use until resizable lists are part of the
language, thus unimplemented.
// Maybe useful when functions return variable sized lists.
//  | (len <expr>)                           // Length of list
//  | (list <expr>)                          // Initialize with size
//  | (loop (<binding>{1-2} <expr>) <expr>)  // Range over the list

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

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

- This syntax introduces heap allocated fixed size `list`s with `1` based indexing.
- Lists can take arbitrary number of elements. Zero size lists become `nil`.
- The `=` operator can now also check reference equality between lists.
- `nil` represents an empty list. It cannot be indexed into. Can only be compared with other lists. Comparing with other types is a dynamic error.

- `index` returns the element in the given index of the list. `set!` mutates the list's index to have the given value.
- `index` and the three argument version of `set!` both raise dynamic errors for index out of range, including indexing into `nil`.

Heap structure

This compiler implements a simple heap structure with memory drawn from the Rust's `Vec<u64>`. Each list has an extra word at the very beginning to indicate the length of the list. Heap grows sequentially forward (to higher memory addresses). Garbage collection is not implemented. Heap has a fixed size of `16384` words or `128KiB`.

E.g. `(list 1 2 3 nil)` would translate to the following layout, assuming a `0x100` start address.

Heap				
0x100	0x108	0x110	0x118	0x120
4(len)	2(1)	4(2)	6(3)	1(nil)

Tests

This section shows the tests for the newly introduced syntax. Each test shows the `.snek` file contents and the output/error of each run. The `.snek` files contain the test values in comments (lines starting with `;`) towards the end.

Simple Examples

```
❯ cat ./tests/ee_simple_example1.snek
(list 1 2 3 4 5)
; (output "(list 1 2 3 4 5)")
❯ ./tests/ee_simple_example1.run
(list 1 2 3 4 5)
```

The list output is in the same representation as the code.

```
❯ cat ./tests/ee_simple_example2.snek
(print (list 1 input 3 (print 4) 5))
; (output 4 "(list 1 false 3 4 5)" "(list 1 false 3 4 5)")
; ((input 10) (output 4 "(list 1 10 3 4 5)" "(list 1 10 3 4 5)"))
❯ ./tests/ee_simple_example2.run
4
(list 1 false 3 4 5)
(list 1 false 3 4 5)
❯ ./tests/ee_simple_example2.run 10
4
(list 1 10 3 4 5)
(list 1 10 3 4 5)
```

```
❯ cat ./tests/ee_simple_example3.snek
(list)
; (output "nil")
❯ ./tests/ee_simple_example3.run
nil
```

An empty list automatically becomes nil.

Error Tag

```
❯ cat ./tests/ee_error_tag1.snek
(> 2 (list 2))
; (dynamic "invalid argument")
❯ ./tests/ee_error_tag1.run
an error ocurred 23: invalid argument
```

This comes from the runtime type checking for comparison operators which require both arguments to be numbers.

```
❯ cat ./tests/ee_error_tag2.snek
(index 4 2)
; (dynamic "invalid argument")
❯ ./tests/ee_error_tag2.run
an error ocurred 21: invalid argument
```

This comes from runtime type checking in the `index` implementation that requires the first argument to be a list and the second one to be a number.

Error Bounds

```
❯ cat ./tests/ee_error_bounds1.snek
(index (list 1 2 3) input)
; (dynamic "invalid argument")
; ((input 0) (dynamic "out of range"))
; ((input 4) (dynamic "out of range"))
; ((input 2) (output 2))
❯ ./tests/ee_error_bounds1.run
an error ocurred 25: invalid argument
❯ ./tests/ee_error_bounds1.run 0
an error ocurred 40: index out of range
❯ ./tests/ee_error_bounds1.run 4
an error ocurred 40: index out of range
❯ ./tests/ee_error_bounds1.run 2
2
```

The first runtime error comes from `index` requiring the second argument to be a number. The next two runtime errors are from the out of bounds check for the index w.r.t to the list's length.

Error3

```
❯ cat ./tests/ee_error3.snek
(+ 1 (index (list 4611686018427387903) 1))
; (dynamic "overflow")
❯ ./tests/ee_error3.run
an error ocurred 32: overflow
```

Regular errors still work when used with lists, like overflow.

Points

```
❯ cat ./tests/ee_points1.snek
(fun (point x y) (list x y))
(fun (addpoints p1 p2) (list (+ (index p1 1) (index p2 1)) (+ (index p1 2)
(index p2 2))))
(addpoints (point input 0) (point (- 0 input) (* 2 input)))
; (dynamic "invalid argument")
; ((input 2) (output "(list 0 4)"))
; ((input -2) (output "(list 0 -4)"))
❯ ./tests/ee_points1.run
an error ocurred 25: invalid argument
❯ ./tests/ee_points1.run 2
(list 0 4)
❯ ./tests/ee_points1.run -2
(list 0 -4)
```

Calling `point` and `addpoints` with different input values.

```
❯ cat ./tests/ee_points2.snek
(fun (point x y) (list x y))
(fun (addpoints p1 p2) (list (+ (index p1 1) (index p2 1)) (+ (index p1 2)
(index p2 2))))
(let ((x (point input 0))) (addpoints x (point input (* 2 input))))
; (dynamic "invalid argument")
; ((input 2) (output "(list 4 4)"))
; ((input -2) (output "(list -4 -4)"))
❯ ./tests/ee_points2.run
an error ocurred 25: invalid argument
❯ ./tests/ee_points2.run 2
(list 4 4)
❯ ./tests/ee_points2.run -2
(list -4 -4)
```

Calling `point` and `addpoints` with different input values with a response bound to a variable.

Binary Search Tree

```
❯ cat ./tests/ee_bst.snek
; Add value to a bst, creating one if nil
(fun (add_val root x) (
    if (= root nil) (
        ; Create one if nil
        list nil x nil
    ) (block
            ; Add to the right spot
            (let ((val (index root 2))) (
                ; If equal, already present, return
                if (= x val) nil (
                    if (< x val)
                        ; Left child update
                        (set! root 1 (add_val (index root 1) x))
                        ; Right child update
                        (set! root 3 (add_val (index root 3) x))
                )
            ))
            ; Return root
            root
    )
))

; Check for value in BST
(fun (val_in bst x) (
    if (= bst nil) false (
        let ((val (index bst 2))) (
            if (= x val) true (
                if (< x val)
                    (val_in (index bst 1) x)
                    (val_in (index bst 3) x)
            )
        )
    )
))

; Main body
(let
    ((tree (add_val (add_val (add_val (add_val (add_val nil 3) 2) 4) 1)
5)))
    (block
        (set! tree (add_val tree -1))
        (val_in tree input)
    )
)

; (dynamic "invalid argument")
```

```
; ((input -1) (output true))
; ((input 0) (output false))
; ((input 1) (output true))
; ((input 2) (output true))
; ((input 3) (output true))
; ((input 4) (output true))
; ((input 5) (output true))
; ((input 6) (output false))
❯ ./tests/ee_bst.run
an error ocurred 22: invalid argument
❯ ./tests/ee_bst.run -1
true
❯ ./tests/ee_bst.run 0
false
❯ ./tests/ee_bst.run 1
true
❯ ./tests/ee_bst.run 2
true
❯ ./tests/ee_bst.run 3
true
❯ ./tests/ee_bst.run 4
true
❯ ./tests/ee_bst.run 5
true
❯ ./tests/ee_bst.run 6
false
```

Binary Search Tree implementation using mutable lists (`set!` on `list`). Shows the usage of `add_val` and `val_in` along with the overloaded `set!` usage on lists and variables. `0` and `6` are not in the BST. This implementation mutates the BST rather than constructing a new one along the path. Uses the first index for the left child, second index for the value and third index for the right child.

Comparisons

Python

Primitive types can go on the heap and allocated dynamically, which is how this compiler does it as well. Metadata is stored by wrapping the primitive types in data objects, which means even constants have overhead. This is similar to how we have tags for numbers and an extra word for representing length in lists. Memory for heap allocated data is managed by the Python memory manager, similar to how we take care of memory allocation and layout in Snek.

C

Primitive types in C do not go on the heap. Heap needs to be manually allocated and managed unlike Snek. Memory allocation is explicit via the `malloc`/`free` library calls. Pointers are exposed for heap values, making it visible to the programmer. Snek does not have any of these and uses the heap in a way that is transparent to the programmer.

References

- GDB to LLVM command map
- BST Implementation
- Python built in STD types
- Python memory management C interface
- Python data model
- Python number class
- PEP-3141 type hierarchy for numbers