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, provided the tuples are present in a let binding.

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! <name> <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!)

<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 a diagram of how tuples are laid out in the heap, assuming the heap address starts initially at `0x1000`:

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.

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.