CSE 231 Egg Eater

Concrete Syntax

The concrete syntax for Egg Eater is the same as that of Diamondback, but there are 3 new additions to support heap-allocated values:

1. The `tuple <expr>+` expression creates a heap-allocated tuple of 1 or more values, where each value is obtained by evaluating the corresponding expression. Its return value is the memory address of the start of the tuple.
    a. The expressions are evaluated in left-to-right order.
2. The `index <expr> <expr>` expression returns the value of the element that is stored at an offset equal to the evaluated result of the second expression, starting from the memory address equal to the evaluated result of the first expression.
    a. The indexes are 0-based; that is, an offset of 0 will return the first element of the tuple whose memory address is specified by the first expression.
    b. If the first expression does not evaluate to a memory address, then the running program reports a dynamic error containing the string `“invalid tuple address”`.
    c. If the second expression does not evaluate to a number, then the running program reports a dynamic error containing the string `“invalid tuple offset”`.
    d. If the offset is negative or greater than or equal to the number of elements in the tuple, then the running program reports a dynamic error containing the string `“index out of bounds”`.
3. The `nil` expression indicates a memory address that points to nowhere; it serves a similar purpose as the `null` value in Java and `None` value in Python.

The full specification of the concrete syntax is given below:

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

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

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

How Heap Values Are Arranged

The Egg Eater heap consists of a single contiguous region of memory that is allocated by a Rust `Vec`. Heap values are allocated at increasing memory addresses.

Heap values are represented by `tuple` structures. The first element of every tuple is a number `len` specifying the size of the tuple. Consequently, the return value of every tuple is the memory address to this `len`. However, the offsets specified by `index` expressions are translated by the compiler into offsets from the "true" first element of the `tuple`.

For the expression:

```
(let ((pair (tuple 1 10)) (triple (tuple 4 pair nil)))
     (block (
          (print pair)
          (print triple)
     )
)
```

The following is a diagram of how these values are arranged on the heap. The starting address is chosen to be 0x1000 for illustration but is not necessarily this value. The value as it is stored on the heap is shown, and its true value is shown in parentheses. The diagrams use square brackets to indicate the elements of a tuple at a particular index.

Address	0x1000	0x1008	0x1020	0x1028
Value (actual)	4 (2)	2 (1)	20 (10)	6 (3)
Explanation	Size of `pair`	`pair[0]`	`pair[1]`	Size of `triple`

Address	0x1030	0x1038	0x1040	0x1048
Value (actual)	8 (4)	0x1001 (0x1000)	1 (nil)	Arbitrary
Explanation	`triple[0]`	`triple[1]`	`triple[2]`	The heap pointer points to this address

Test Cases

`input/simple_examples.boa`

Code

```
(fun (make_pair a b)
    (tuple a b)
)

(fun (make_triple a b c)
    (tuple a b c)
)

(let ( (pair (make_pair 1 2)) (triple (make_triple 3 4 5)) (struct (make_pair
pair nil) ) )
     (block
        (print (index pair 0))
        (print (index pair 1))
        (print pair)
        (print (index triple 0))
        (print (index triple 1))
        (print (index triple 2))
        (print triple)
        (print struct)
     )
)
```

Explanation

This test demonstrates the construction, accessing, and printing of heap-allocated values in the following ways:

- Construction: Heap-allocated values of varying lengths (i.e. 2-tuples and 3-tuples).
- Accessing: Each element of each heap-element value is accessed.
- Printing: A heap-allocated value that contains a pointer to another heap-allocated value is printed, and the contents are printed recursively.

Result of running program

```
1
2
(tuple 1 2)
3
4
5
(tuple 3 4 5)
(tuple (tuple 1 2) nil)
(tuple (tuple 1 2) nil)
```

This is the expected output. One interesting feature of the output is that nested tuples are enclosed in parentheses to make it easier to visually inspect the structure of the tuples.

`input/error-tag.boa`

Code

```
(let ((not_an_address 10)) (index not_an_address 0) )
```

Explanation

This test demonstrates how the runtime checks the tag of memory addresses to ensure that only valid addresses are accessed during runtime. This check happens at runtime, and the resulting error is a dynamic error. In the test program, a non-memory address value is used in the `index` expression.

Result of running program

```
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ make tests/error-tag.run
cargo run -- tests/error-tag.snek tests/error-tag.s
    Finished dev [unoptimized + debuginfo] target(s) in 0.23s
     Running `target/debug/egg_eater tests/error-tag.snek tests/error-tag.s`
nasm -f elf64 tests/error-tag.s -o tests/error-tag.o
ar rcs tests/liberror-tag.a tests/error-tag.o
rustc -L tests/ -lour_code:error-tag runtime/start.rs -o tests/error-tag.run
rm tests/error-tag.s
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ ./tests/error-tag.run
an error occurred: invalid tuple address
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$
```

`input/error-bounds.boa`

Code

```
(let ((obj (tuple 1 2 3))) (index obj input))
```

Explanation

This test demonstrates how the runtime checks the offset value used in `index` expressions and throws an error if the offset is out of bounds of the associated tuple. This check happens during runtime, and the resulting error is a dynamic error.

Result of running program

Three different runs of the program are shown. The first two use out-of-bound indexes, which cause an error. The last run uses a valid index, so the expected result is returned.

```
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ make tests/error-bounds.run
cargo run -- tests/error-bounds.snek tests/error-bounds.s
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
     Running `target/debug/egg_eater tests/error-bounds.snek tests/error-bounds.s`
nasm -f elf64 tests/error-bounds.s -o tests/error-bounds.o
ar rcs tests/liberror-bounds.a tests/error-bounds.o
rustc -L tests/ -lour_code:error-bounds runtime/start.rs -o tests/error-bounds.run
rm tests/error-bounds.s
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ ./tests/error-bounds.run 3
an error occurred: index out of bounds
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ ./tests/error-bounds.run -1
an error occurred: index out of bounds
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ ./tests/error-bounds.run 1
2
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$
```

input/error3.boa

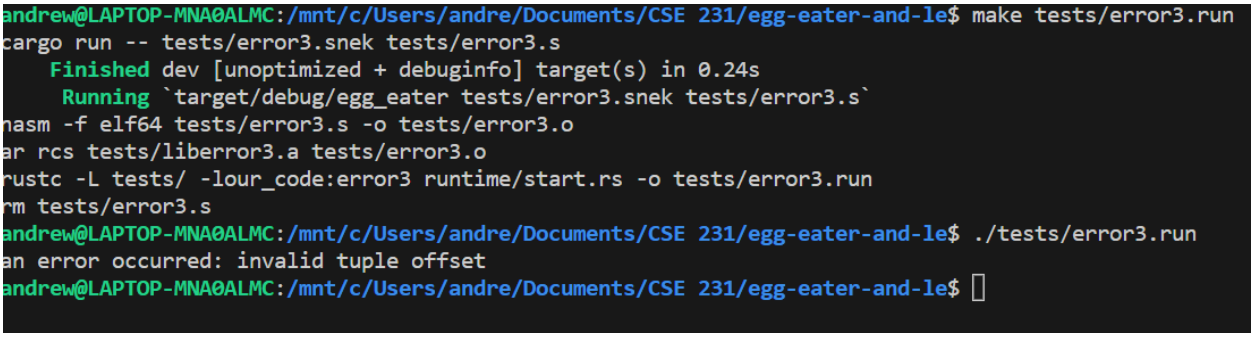
Code

```
(let ((pair (tuple 1 2))) (index pair false) )
```

Explanation

This test demonstrates how the runtime checks that the offset value used in `index` expressions is a number and throws an error if the offset is not a number. This check happens during runtime, and the resulting error is a dynamic error. In the test program, a Boolean value is used to access an element of the tuple.

Result of running program



input/points.boa

Code

```
(fun (make_point x y)
    (tuple x y)
)

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

(let ( (p1 (make_point 5 10)) (p2 (make_point 30 60)) )
     (block
        (print p1)
        (print p2)
        (print (add_points p1 p2))
     )
)
```

Explanation

This program defines a function that creates a 2-dimensional point from two numbers as a 2-tuple. It also defines a function that creates a 2-dimensional point by summing elementwise the coordinates of 2 other input points. In the test program, 2 points are created, and a third point is constructed from them.

Result of running program

```
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ make clean
rm -f tests/*.a tests/*.s tests/*.run tests/*.o
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ make tests/points.run
cargo run -- tests/points.snek tests/points.s
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
     Running `target/debug/egg_eater tests/points.snek tests/points.s`
nasm -f elf64 tests/points.s -o tests/points.o
ar rcs tests/libpoints.a tests/points.o
rustc -L tests/ -lour_code:points runtime/start.rs -o tests/points.run
rm tests/points.s
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ ./tests/points.run
(tuple 5 10)
(tuple 30 60)
(tuple 35 70)
(tuple 35 70)
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$
```

This is the expected output. The two points are constructed as they were specified in the program, and the points resulting from adding the points together has the correct coordinate values.

input/bst.boa

Code

```
(fun (make_bst root_val left right)
    (tuple root_val left right)
)


(fun (bst_insert root val)
    (if (= root nil)
        (tuple val nil nil)
        (let ( (root_val (index root 0)) (left (index root 1)) (right (index root 2)) )
             (if (= val root_val)
                 (make_bst root_val left right)
                 (if (< val root_val)
                     (make_bst root_val (bst_insert left val) right)
                     (make_bst root_val left (bst_insert right val) )
                 )
             )
        )
    )
)
```

```
(fun (bst_contains root query)
    (if (= root nil)
        false
        (let ( (root_val (index root 0)) (left (index root 1)) (right (index
root 2)) )
             (if (= query root_val)
                true
                (if (< query root_val)
                    (bst_contains left query)
                    (bst_contains right query)
                )
             )
        )
    )
)
```

```
(let ( (tree (make_bst 50 nil nil)) )
     (block
        (print tree)
        (set! tree (bst_insert tree 25))
        (print tree)
        (set! tree (bst_insert tree 75))
        (print tree)
        (set! tree (bst_insert tree 100))
        (print tree)
        (set! tree (bst_insert tree 0))
        (print tree)
        (print (bst_contains tree 50))
        (print (bst_contains tree 100))
        (print (bst_contains tree 0))
        (print (bst_contains tree 37))
        (print (bst_contains tree 86))
     )
)
```

Explanation

This program represents a Binary Search Tree (BST) node as a 3-tuple in which the first element is the value at the node, the second element is a pointer to the left subtree, and the third element is a pointer to the right subtree. The value `nil` is used to indicate that there is no subtree.

This program defines a function that creates a BST from a given number value, a pointer to a left subtree, and a pointer to a right subtree. It also defines a function that creates a new BST given the root of an existing BST and a number; this function does not support duplicates; that is, if this function encounters a value that already exists in the tree, it returns a new copy of the existing tree, *without* a duplicate copy of the value. The program also defines a function that returns true or false depending on whether the BST contains a query value. The main expression initializes a tree, inserts several values into it, and queries it for several values.

Result of running program

```
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ make tests/bst.run
cargo run -- tests/bst.snek tests/bst.s
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
     Running `target/debug/egg_eater tests/bst.snek tests/bst.s`
nasm -f elf64 tests/bst.s -o tests/bst.o
ar rcs tests/libbst.a tests/bst.o
rustc -L tests/ -lour_code:bst runtime/start.rs -o tests/bst.run
rm tests/bst.s
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ ./tests/bst.run
(tuple 50 nil nil)
(tuple 50 (tuple 25 nil nil) nil)
(tuple 50 (tuple 25 nil nil) (tuple 75 nil nil))
(tuple 50 (tuple 25 nil nil) (tuple 75 nil (tuple 100 nil nil)))
(tuple 50 (tuple 25 (tuple 0 nil nil) nil) (tuple 75 nil (tuple 100 nil nil)))
true
true
true
false
false
false
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$
```

This is the expected output.

Comparisons with Other Programming Languages

I've chosen to compare my implementation of heap allocation for Egg Eater with heap allocation in Java and Rust. My design is more like Java than Rust's. In Java, the programmer must store all objects on the heap; in Egg Eater, all tuples (the equivalent of objects) must be stored on the heap. In contrast, Rust stores objects on the stack by default, but the programmer can specify a heap-allocated object with a `Box<T>` pointer. In Egg Eater, there is no way to store a tuple on the stack.

Attributions

Course Resources

I used the lecture code for Egg Eater and the class handouts to help me implement Egg Eater.

Online Resources

Understanding Memory Management. https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html

Box, stack, and heap. https://doc.rust-lang.org/rust-by-example/std/box.html#:~:text=All%20values%20in%20Rust%20are,on%20the%20heap%20is%20freed.