Egg Eater + Green Snake

Table of Contents

Concrete Syntax

The concrete syntax for Egg Eater is the same as that of Diamondback with the following additions:

`nil`

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. It has an internal value of 1.

`vec <expr>+`

The `vec <expr>+` expression creates a heap-allocated vector of 1 or more contiguous values, where each value is obtained by evaluating the corresponding expression. Its return value is the memory address of the start of the vector. For an input of N expressions, N words of space in the heap are pre-allocated, and the expressions are evaluated in left-to-right order and stored sequentially at these heap locations.

`vec-get <vec_expr> <index_expr>`

The `vec-get <vec_expr> <index_expr>` expression attempts to return the value at index `<index_expr>` within the vector `<vec_expr>`. The indexing is 0-based.

If `<vec_expr>` does not evaluate to a non-nil vector, then the program reports a dynamic error containing the string `"invalid vector address"`.

If `<index_expr>` does not evaluate to a number, then the program reports a dynamic error containing the string `"invalid vector offset"`.

If `<index_expr>` evaluates to a number that is negative or greater than or equal to the size of the vector, then the program reports a dynamic error containing the string `"index out of bounds"`.

The full specification of the concrete syntax is:

```
<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>*)
  | (vec <expr>+) (new!)
  | (vec-get <expr> <expr>) (new!)


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

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

Heap Value Layout

The Egg Eater heap consists of a single contiguous region of memory that is allocated by a Rust `Vec`. To place values in the heap, we use `vec` data types. The first element of every vector is a positive integer specifying the size of the vector.

Consider the expression:

```
(let ((pair (vec 1 10)) (triple (vec 4 pair nil)))
       (block (
              (print pair)
              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, if different from its stored 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)	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_1.snek`

Code

```
(let ((pair (vec 10 20)) (triple (vec 30 40 nil)))
     (block
        (print (vec-get pair 0))
        (print (vec-get pair 1))
        (print pair)
        (print (vec-get triple 0))
        (print (vec-get triple 1))
        (print (vec-get triple 2))
        triple
     )
)
```

Explanation

This test demonstrates:

- Construction of heap values via `vec`
- Indexing of heap values via `vec-get`
- Printing of heap values

Program Output

```
10
20
[10, 20]
30
40
nil
[30, 40, nil]
```

This is the expected output.

input/simple_examples_2.snek

Code

```
(let ((a (vec (vec 10 20 nil) (vec 30 40 nil))))
     a
)
```

Explanation

This test demonstrates:

- Construction of heap values with nested heap values

Program Output

```
[[10, 20, nil], [30, 40, nil]]
```

This is the expected output.

input/error-tag.snek

Code

```
(let ((not_an_address 10)) (vec-get not_an_address 0))
```

Explanation

This test demonstrates how the runtime checks the tag of vectors to ensure that only valid addresses are accessed during runtime. This check happens at runtime, so the resulting error is a dynamic error.

Program Output

```
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ ./tests/error-tag.run
an error occurred: invalid vector address
```

input/error-bounds.snek

Code

```
(let ((obj (vec 1 2 3))) (vec-get obj input))
```

Explanation

This test demonstrates how the runtime checks the offset value used to access vector values and throws an error if the offset is out of bounds of the associated vector. This check happens during runtime, so the resulting error is a dynamic error.

Program Output

```
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
```

input/error3.snek

Code

```
(let ((pair (vec 1 2))) (vec-get pair false))
```

Explanation

This test demonstrates how the runtime checks that the offset value used in expressions is a number and throws an error if the offset is not a number. This check happens during runtime, so the resulting error is a dynamic error.

Program Output

```
andrew@LAPTOP-MNA0ALMC:/mnt/c/Users/andre/Documents/CSE 231/egg-eater-and-le$ ./tests/error3.run
an error occurred: invalid vector offset
```

input/points.snek

Code

```
(fun (make_point x y)
    (vec x y)
)
(fun (add_points p1 p2)
    (vec (+ (vec-get p1 0) (vec-get p2 0) ) (+ (vec-get p1 1) (vec-get p2 1)) )
)
(let ((p1 (make_point 5 10)) (p2 (make_point 30 60)))
     (block
        (print p1)
        (print p2)
        (add_points p1 p2)
     ))
```

Explanation

This program defines a function that creates a 2-dimensional point from two numbers as a 2-element vector. It also defines a function that creates a 2-dimensional point by summing element-wise the coordinates of 2 other input points.

Program Output

```
[5, 10]
[30, 60]
[35, 70]
```

This is the expected output.

input/bst.snek

Code

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

(fun (bst_insert root val)
    (if (= root nil)
        (vec val nil nil)
        (let ( (root_val (vec-get root 0)) (left (vec-get root 1)) (right (vec-get 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 (vec-get root 0)) (left (vec-get root 1)) (right (vec-get 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
        (set! tree (bst_insert tree 25))
        (set! tree (bst_insert tree 75))
        (set! tree (bst_insert tree 100))
        (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))
        (bst_contains tree 86)
     )
)
```

Explanation

This program represents a Binary Search Tree (BST) node as a 3-element vector in which the first element is the value stored by 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.

Program Output

```
[50, [25, [0, nil, nil], nil], [75, nil, [100, nil, nil]]]
true
true
true
false
false
```

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.

New Concrete Syntax

The concrete syntax for the extended Egg Eater is the same as that of the original Egg Eater with the following additions:

`vec-set! <vec_expr> <index_expr> <value_expr>`

The `vec-set! <vec_expr> <index_expr> <value_expr>` expression attempts to set the value at index `<index_expr>` within the vector `<vec_expr>` to be the value `<value_expr>`.

If `<vec_expr>` does not evaluate to a non-nil vector, then the program reports a dynamic error containing the string `"invalid vector address"`.

If `<index_expr>` does not evaluate to a number, then the program reports a dynamic error containing the string `"invalid vector offset"`.

If `<index_expr>` evaluates to a number that is negative or greater than or equal to the size of the vector, then the program reports a dynamic error containing the string `"index out of bounds"`.

`==`

The `==` operator compares two values for structural equality.

The full specification of the concrete syntax is:

```
<prog> := <defn>* <expr>
<defn> := (fun (<name> <name>*) <expr>)
<expr> :=
  | <number>
  | true
  | false
  | input
  | nil
  | <identifier>
  | (let (<binding>+) <expr>)
  | (<op1> <expr>)
  | (<op2> <expr> <expr>)
  | (set! <name> <expr>)
  | (if <expr> <expr> <expr>)
  | (block <expr>+)
  | (loop <expr>)
  | (break <expr>)
  | (<name> <expr>*)
  | (vec <expr>+)
  | (vec-get <expr> <expr>)
  | (vec-set! <expr> <expr> <expr>) (new!)
<op1> := add1 | sub1 | isnum | isbool | print
<op2> := + | - | * | < | > | >= | <= | = | == (new!)

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

Handling Structural Equality

I use a Rust function linked at runtime with the assembly code of generated programs to perform the check for structural equality.

I use the definition of structural equality that we discussed in class. Two vectors are defined to be structurally equal if they contain the same number of elements and the non-vector elements are equal. For elements that are themselves vectors, structural equality is met if the vectors themselves are the same memory reference or if these vectors contain the same contents; that is, within these vectors, the values at corresponding positions are equal (i.e., they have the same structure).

If values of different types are compared using structural equality, no error is thrown, which differs from the behavior of reference equality.

To implement structural equality, I maintain a collection of seen pairs of vector addresses and query this collection to avoid entering infinite loops in cyclic vector structures.

The relevant code snippets:

```rust
// Checks structural equality of two values.
#[export_name = "\x01snek_equals"]
pub unsafe extern "C" fn snek_equals(val1: i64, val2: i64) -> i64 {
    snek_equals_helper(val1, val2, &mut HashSet::<(i64, i64)>::new())
}
```

```rust
// Helper function for structural equality
unsafe fn snek_equals_helper(val1: i64, val2: i64, seen: &mut HashSet<(i64, i64)>) -> i64 {
    if val1 & 3 == 1 && val2 & 3 == 1 {
        if val1 == val2 {
            // println!("Pointers are equal");
            seen.remove(&(val1, val2));
            return TRUE;
        }
        if val1 == NIL || val2 == NIL {
            // println!("Pointer is NIL");
            seen.remove(&(val1, val2));
            return FALSE;
        }
        if !seen.insert((val1, val2)) {
            // println!("Pointers seen before");
            seen.remove(&(val1, val2));
            return TRUE;
        }

        let addr1 = (val1 - 1) as *const u64;
        let addr2 = (val2 - 1) as *const u64;
        let size1 = addr1.read();
        let size2 = addr2.read();
        if size1 != size2 {
            return FALSE;
        }
        // Compare each of the values of val1 and val2
        for i in 0..size1 {
            let elem1 = addr1.add(1 + i as usize).read() as i64;
            let elem2 = addr2.add(1 + i as usize).read() as i64;
            if snek_equals_helper(elem1, elem2, seen) == FALSE {
                return FALSE;
            }
        }
        seen.remove(&(val1, val2));
        // println!("Structurally equal");
        TRUE
    } else {
        // If val1 and val2 aren't pointers, use reference equality
        seen.remove(&(val1, val2));
        if val1 == val2 {
            // println!("Referentially equal");
            return TRUE;
        } else {
            // println!("Referentially unequal");
            return FALSE;
        }
    }
}
```

Handling Cycles

I modified the implementation of the `snek_print` Rust function to account for cycles.

To implement cycle detection, I maintain a collection of seen vector addresses and query this collection to avoid entering infinite loops when printing vector structures containing cycles. The string `[…]` is inserted into the output to indicate the presence of a cycle.

The relevant code snippets:

```rust
// Prints the formatted representation of the value and returns the original input value.
#[export_name = "\x01snek_print"]
pub unsafe extern "C" fn snek_print(val: i64) -> i64 {
    let print_val = snek_str(val, &mut HashSet::<i64>::new());
    println!("{print_val}");
    val
}

// Converts the internal representation of the value to its true value, formatted as a string.
unsafe fn snek_str(val: i64, seen: &mut HashSet<i64>) -> String {
    if val == 7 {
        String::from("true")
    } else if val == 3 {
        String::from("false")
    } else if val % 2 == 0 {
        (val >> 1).to_string()
    } else if val == 1 {
        String::from("nil")
    } else if val & 1 == 1 {
        if !seen.insert(val) {
            return String::from("[...]");
        }
        let addr = (val - 1) as *const u64;
        let size = addr.read() as usize;
        let mut result_str = String::from("[");
        for i in 1..size + 1 {
            let elem = addr.add(i).read() as i64;
            result_str = result_str + &snek_str(elem, seen);
            if i < size {
                result_str = result_str + ", ";
            }
        }
        seen.remove(&val);
        result_str + "]"
    } else {
        format!("Unknown value: {}", val)
    }
}
```

New Test Cases

`input/equal.snek`

Code

```
(let ((p (vec 10 20)) (q (vec 10 20)) (r (vec 30 40)))
     (block
        (print (== 5 5))
        (print (== 5 10))
        (print (== true true))
        (print (== true false))
        (print (== nil nil))
        (print (== p nil))
        (print (== p p))
        (print (== p q))
        (== q r)
     )
)
```

Explanation

This test demonstrates structural equality on the different types of values that can be compared.

- Structural equality applies to numbers and Booleans is the same as referential equality.
- `nil` is equal to itself but not to other vectors.
- Two vectors are structurally equal if they are the same reference (e.g., `(== p p)`)
- Two vectors are structurally equal if their contents are the same, even if their references are different (e.g., `(== p q)`).

Program Output

```
true
false
true
false
true
false
true
true
false
```

This is the expected output.

input/cyclic-print1.snek

Code

```
(let ((p (vec 10 20 30)))
    (block
        (print p)
        (vec-set! p 1 p)
        p
    )
)
```

Explanation

This test demonstrates cyclic printing for a vector that contains an element which loops back to the start of the vector.

Program Output

```
[10, 20, 30]
[10, [...], 30]
```

This is the expected output.

input/cyclic-print2.snek

Code

```
(let ((p (vec 10 20 30)) (q (vec 40 50 60)) )
    (block
        (print p)
        (print q)
        (vec-set! p 1 q)
        (vec-set! q 1 q)
        (print p)
        print q
    )
)
```

Explanation

This test demonstrates cyclic printing for a vector containing a nested vector that has a cycle.

Program Output

```
[10, 20, 30]
[40, 50, 60]
[10, [40, [...], 60], 30]
[40, [...], 60]
```

This is the expected output.

input/cyclic-print3.snek

Code

```
(let ( (a (vec 10 nil)) (b (vec 20 nil)) (c (vec 30 nil)) (d (vec 40 nil)) (e (vec 50 nil))  )
    (block
        (vec-set! a 1 b)
        (vec-set! b 1 c)
        (vec-set! c 1 d)
        (vec-set! d 1 e)
        (print a)
        (vec-set! d 1 b)
        a
    )
)
```

Explanation

This test demonstrates cyclic printing for a linked list in which a non-tail element cycles back to a previous element that is not the head of the linked list.

Program Output

```
[10, [20, [30, [40, [50, nil]]]]]
[10, [20, [30, [40, [...]]]]]
```

This is the expected output.

input/cyclic-equal1.snek

Code

```
(let ((p (vec 10 20 30)) (q (vec 10 20 30)) (r (vec 40 50 60)) (s (vec 40 50 60)))
    (block
        (vec-set! r 1 r)
        (vec-set! s 1 s)
        (vec-set! p 1 r)
        (vec-set! q 1 s)
        (== p q)
    )
)
```

Explanation

This test demonstrates cyclic equality between the vectors p and q; both contain references to two different vectors (s and r, respectively) whose contents are the same.

Program Output

```
true
```

This is the expected output.

input/cyclic-equal2.snek

Code

```
(let ((p (vec 10 20 30)) (q (vec 10 20 30)) (r (vec 40 50 7)) (s (vec 40 50 900)))
    (block
        (vec-set! r 1 r)
        (vec-set! s 1 s)
        (vec-set! p 1 r)
        (vec-set! q 1 s)
        (== p q)
    )
)
```

Explanation

This test demonstrates cyclic inequality between the vectors p and q; both contain references to two different vectors, r and s respectively, whose contents are different; therefore, p and q are not equal.

Program Output

```
false
```

This is the expected output.

input/cyclic-equal3.snek

Code

```
(let ((p (vec 10 20 30)) (q (vec 10 20 30)))
    (block
        (vec-set! p 1 p)
        (vec-set! q 1 q)
        (== p q)
    )
)
```

Explanation

This test demonstrates cyclic equality between the vectors p and q; both have the same structure because their non-vector elements are equal, and they both have a cycle that loops back to the same part of the vector.

Program Output

```
true
```

This is the expected output.

Other New Features

Calling Convention Revision

I revised my compiler's calling convention to emulate the calling convention implemented by the Forest Flame compiler.

`isvec <expr>`

The `isvec <expr>`expression returns true if `<expr>` evaluates to a vector (possibly nil) type and false otherwise.

The concrete syntax for this expression is:

```
  | isvec <expr>
```

`vec-len <vec_expr>`

The `vec-len <vec_expr>`expression returns the length of the vector given by `<vec_expr>`. If `<vec_expr>` does not evaluate to a non-nil vector, this throws an error containing the string `"invalid vector address"`.

The concrete syntax for this expression is:

```
  | vec-len <expr>
```

`make-vec <size_expr> <elem_expr>`

The `make-vec <size_expr> <elem_expr>` expression creates a vector of size `<size_expr>` all initialized to the value `<elem_expr>`. If `<size_expr>` evaluates to an integer less than 1, this throws an error containing the string `"invalid vector size"`.

The concrete syntax for this expression is:

```
  | make-vec <expr> <expr>
```

Attributions

Course Resources

I used the lecture code on GitHub, class handouts, and Forest Flame starter code to help me revise and extend 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.