The Egg Eater Language

1. Concrete Syntax

The concrete syntax of Egg Eater has added `nil` value, `tuple` and `index` expressions from the past Diamondback.

```
<prog> := <defn>* <expr>
<defn> := (fun (<name> <name>*) <expr>)
<expr> :=
  | <number>
  | nil
  | 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>*)
  | (tuple <expr>+)
  | (index <expr> <expr>)


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

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

2. Semantics

The semantics of added features is as follows:

2.1. Tuples

Tuples are created using the `(tuple <expr>+)` syntax.

They can hold any number of expressions and dynamically allocate memory to store the evaluated values. Tuples allow heap-allocation of an arbitrary number of values. Expressions within the tuple are evaluated, and memory is allocated accordingly.

If a heap-allocated value is the result of a program or printed by print, all of its contents will be printed to show its structure.

2.2. Indexed Lookup

Indexed lookup is performed using the `(index <expr> <expr>)` syntax.

The first expression represents a heap-allocated value, and the second expression is the index.

The lookup retrieves the value at the specified index.

If the index is out of bounds, a dynamic error is reported.

2.3. Nil

`nil` is a value that represents the absence of a meaningful or valid object or data. It has the same type-tag as tuples.

2.4. Structural Update

Structural Update is performed using the `(setindex <expr> <expr> <expr>)` syntax.

The first expression represents a heap-allocated value, the second expression is the index, and the third expression is value to set for the specified index of the heap-allocated value.

The operation updates the value at the specified index of the heap-allocated value.

If the index is out of bounds, a dynamic error is reported.

2.5 Printing Cyclic Values

If a heap-allocated value is a cyclic one, the program will print `...` when it finds out that some heap-allocated value was seen before.

2.6. Structural Equality

Operator `==` is used for determining structural equality. It can only be used on two heap-allocated values. Otherwise, a dynamic error is reported.

In structural equality determination, the program will compare the structure of two heap-allocated values.

First, if they have different lengths, they are not structural equal.

Second, the program will look into each same indices of the two values, if the values at the same index are different, they are not structural equal.

Third, when it comes to heap-allocated values in some indices, the program will recursively determine whether the two heap-allocated values in the same index of two oprands are structural equal.

Finally, in the recursive process, if the program finds that both heap-allocated values in the same index are seen before previous comparison, it will report equality for these two values.

2.7. Errors

Beyond Diamondback, several dynamic error types are added in Egg Eater.

- If the operators other than `=` or `==` are used on any heap-allocated values, an error containing "invalid argument" will be raised from the running program.
- If the operator `==` is used on any non-heap-allocated values, an error containing "invalid argument" will be raised from the running program.
- If an out-of-bounds index is given in the `index` or `setindex` expression, an error containing "index out of bound" and the given index causing error will be raised.
- If the program tries to index or setindex into a `nil` object, an error containing "try to index of nil" will be raised.

3. Heap-allocated Values Arrangement

The heap-allocated values are arranged as follows:

There first comes an 8-byte value representing the size of the tuple, followed by some (the number is the same as the size of the tuple) 8-byte values representing the element of the tuple.

For example, suppose the heap allocated address starts at `0x1000`, and we have `(tuple 1 2 3)`, the diagram of the values will look like:

0x1000	0x1008	0x1010	0x1018
3 (size)	2 (1)	4 (2)	6 (3)

For nesting tuples, the inner tuples will be allocated memory first.

For example,suppose the heap allocated address starts at `0x1000`, and we have `(tuple 1 2 (tuple 3 4))`, the diagram of the values will look like:

0x1000	0x1008	0x1010	0x1018	0x1020	0x1028	0x1030
2 (size)	6 (3)	8 (4)	3 (size)	2(1)	4 (2)	0x1001

4. Approaches of Adding New Features to the Existed Egg Eater Language

4.1 Approaches of Handling Structural Equality

In the new version of the Egg Eater language, a new oprator `==` is added for structural equality determination. It can only be used for two heap-allocated values.

When the compiler finds such an operator, it will first check whether the two oprands are both heap-allocated values.

```
Op2::Eqq => {
    v.push(Instr::IMov(Val::Reg(Reg::RBX), Val::Reg(Reg::RAX)));
    v.push(Instr::And(Val::Reg(Reg::RBX), Val::Imm(3)));
    v.push(Instr::Cmp(Val::Reg(Reg::RBX), Val::Imm(1)));
    v.push(Instr::Jne(Label::TYPEERROR));
}
```

After that, it will call a runtime function `snek_equal` to help determine whether the two heap-allocated values are structural equal.

```
v.push(Instr::Call(Label::LName("snek_equal".to_string())));
```

```
#[export_name = "\x01snek_equal"]
fn equal_value(i1:i64, i2:i64) -> i64 {
    if sn_equal(i1, i2, Vec::new(), Vec::new()) {
        return 7;
    } else {
        return 3;
    }
}
```

This function will iterate through the two tuples to determine structural equality.

4.2. Approaches of handling cycles

The `sn_value()` function will determine whether two values are structurally equal. For any heap-allocated values, before recursively checking into them, the value will be recorded in the `env Vec` structure.

```
fn sn_equal(i1:i64, i2:i64, env1:Vec::<i64>, env2:Vec::<i64>) -> bool {
    let mut new_env1 = env1.clone();
    let mut new_env2 = env2.clone();
    new_env1.push(i1);
    new_env2.push(i2);
    if i1 % 2 == 0 || i1 == 7 || i1 == 3 || i1 == 1 {
        return i1 == i2
    } else if i1 & 3 == 1 {
        if i2 & 3 != 1 {
            return false;
        }
        if env1.contains(&i1) && env2.contains(&i2) {
            return true;
        }
        let mut new_env1 = env1.clone();
        let mut new_env2 = env2.clone();
        new_env1.push(i1);
        new_env2.push(i2);
        let addr1: *const u64 = (i1 - 1) as *const u64;
        let addr2: *const u64 = (i2 - 1) as *const u64;
        let len_tp1 = unsafe{ *addr1 };
        let len_tp2 = unsafe{ *addr2 };
        if len_tp1 != len_tp2 {
            return false;
        }
        for j in 1..=len_tp1 {
            if !sn_equal(unsafe{ *addr1.offset(j as isize)} as i64,
```

```
unsafe{ *addr2.offset(j as isize)} as i64, new_env1.clone(),
new_env2.clone()) {
                return false;
            }
        }
        return true;
    } else {
        panic!("Unknown:{}", i1);
    }
}
```

If finding two values both seen in previous checking, the function will determine equality for them.

5. Required Tests

5.1. `simple_example.snek`

This is a simple example that prints two tulples (it will print the second twice since it is the result of the program).

```
(let ((a (tuple 1 2 3)) (b (tuple 4 5 6))) (block (print a) (print b)))
```

```
> ./tests/simple_examples.run
(tuple 1 2 3)
(tuple 4 5 6)
(tuple 4 5 6)
```

5.2. `error_tag.snek`

This is an example that tries to compare between two tuples and gets an error.

```
(< (tuple 2 3) (tuple 2))
```

```
> ./tests/error_tag.run
invalid argument
```

The runtime catches the error when evaluating `<` operation. It checks the type of the oprands. After finding the program tries to compare between two tuples, it jumps into the error handling code.

5.3. `error_bound.snek`

This is an example that tries to index out of bound and gets an error.

```
(index (tuple 1 2 3) 4)
```

```
> ./tests/error_bounds.run
index out of bound, 4
```

The runtime catches the error when evaluating `index`. It compares the size of the tuple and the intended index. When discovering index out of bound, it jumps into the error handling code.

5.4. `error3.snek`

This is an example that tries to index into a `nil` object and gets an error.

```
(index nil 2)
```

```
> ./tests/error3.run
try to index of nil
```

The runtime catches the error when evaluating `index`. It checks whether the heap-allocated object is `nil`, if so, it jumps into the error handling code.

5.5. `points.snek`

This is a program with a function (`point`) that takes an `x` and a `y` coordinate and produces a structure with those values, and a function (`add2`) that takes two points and returns a new point with their `x` and `y` coordinates added together, along with several tests that print example output from calling these functions.

```
(fun (point x y) (tuple x y))
(fun (takex pnt) (index pnt 1))
(fun (takey pnt) (index pnt 2))
(fun (add2 pnt1 pnt2) (point (+ (takex pnt1) (takex pnt2)) (+ (takey pnt1)
(takey pnt2))))
(let (
  (a (point 2 3)) (b (point 4 5)) (c (point 6 7))
) (
    block (
      print (add2 a b)
    ) (
      print (add2 b c)
    ) (
      add2 a c
    )
))
```

```
> ./tests/points.run
(tuple 6 8)
(tuple 10 12)
(tuple 8 10)
```

5.6. bst.snek

This is a program that builds a binary search trees, and implements functions to add an element and check if an element is in the tree. The program includes several tests that print example output from calling these functions.

```
(fun (value bst) (index bst 1))
(fun (left bst) (index bst 2))
(fun (right bst) (index bst 3))
(fun (node le ri el) (tuple le ri el))
(fun (find bst elt) (
    if (= bst nil) false (
        if (> elt (value bst)) (
            find (right bst) elt
        ) (
            if (< elt (value bst)) (
                find (left bst) elt
            ) true
        )
    )
))
(fun (insert bst elt) (
    if (= bst nil) (
        node elt nil nil
    ) (
        if (> elt (value bst)) (
            node (value bst) (left bst) (insert (right bst) elt)
        ) (
            if (< elt (value bst)) (
                node (value bst) (insert (left bst) elt) (right bst)
            ) bst
        )
    )
))
(let ((bst (tuple 4 (tuple 2 (tuple 1 nil nil) (tuple 3 nil nil)) (tuple 6
(tuple 5 nil nil) (tuple 7 nil nil))))) (
    block (
        print (insert bst 0)
    ) (
        print (insert bst 8)
    ) (
        print (find bst 5)
    ) (
        find bst 20
```

```
    )
))
```

```
> ./tests/bst.run
(tuple 4 (tuple 2 (tuple 1 (tuple 0 nil nil) nil) (tuple 3 nil nil))
(tuple 6 (tuple 5 nil nil) (tuple 7 nil nil)))
(tuple 4 (tuple 2 (tuple 1 nil nil) (tuple 3 nil nil)) (tuple 6 (tuple 5
nil nil) (tuple 7 nil (tuple 8 nil nil))))
true
false
```

5.7. `equal.snek`

This is a program that checks structural equality for heap-allocated values. `x`, `y`, and `z` are all non-cyclic tuples. Among them, `x` and `y` are structurally equal. `(tuple x y)` and `(tuple y x)` are also structurally equal.

```
(let ((x (tuple 1 2 3)) (y (tuple 1 2 3)) (z (tuple 2 false 3))) (
    block
    (print (== x y))
    (print (== x z))
    (== (tuple x y) (tuple y x))
))
```

```
> ./tests/equal.run
true
false
true
```

5.8. `cycle-print[1-3].snek`

These are programs printing cyclic values.

```
(let ((x (tuple 1 2 3))) (
    block (
        setindex x 2 x
    ) x
))
```

```
> ./tests/cycle-print1.run
(tuple 1 ... 3)
```

```
(let ((x (tuple 1)) (y (tuple 2)) (z (tuple 3))) (
    block (
        setindex z 1 x
    ) (
        setindex x 1 y
    ) (
        setindex y 1 z
    ) x
))
```

```
> ./tests/cycle-print2.run
(tuple (tuple (tuple ...)))
```

```
(let ((x (tuple 1)) (y (tuple 2)) (z (tuple 3))) (
    block (
        setindex z 1 y
    ) (
        setindex x 1 y
    ) (
        setindex y 1 z
    ) (
        print x
    )
    y
))
```

```
> ./tests/cycle-print3.run
(tuple (tuple (tuple ...)))
(tuple (tuple ...))
```

5.9. `cycle-equal[1-3].snek`

These are programs determining structural equality for cyclic values.

```
(let ((x (tuple 1 2 3)) (y (tuple 1 2 3))) (
    block (
        setindex x 2 x
    ) (
        setindex y 2 y
    ) (
        print x
    ) (
        print y
```

```
    ) (
        == x y
    )
))
```

```
> ./tests/cycle-equal1.run
(tuple 1 ... 3)
(tuple 1 ... 3)
true
```

```
(let ((x (tuple 1 2 3)) (y (tuple 1 x 3))) (
    block (
        setindex x 2 x
    ) (
        print x
    ) (
        print y
    ) (
        == x y
    )
))
```

```
> ./tests/cycle-equal2.run
(tuple 1 ... 3)
(tuple 1 (tuple 1 ... 3) 3)
true
```

```
(let (
    (a (tuple 1))
    (b (tuple 2))
    (c (tuple a))
    (x (tuple 1))
    (y (tuple 2))
    (z (tuple y))
) (block
    (setindex a 1 b)
    (setindex b 1 c)
    (setindex x 1 y)
    (setindex y 1 z)
    (print a)
    (print x)
    (== a x)
))
```

```
> ./tests/cycle-equal3.run
(tuple (tuple (tuple ...)))
(tuple (tuple (tuple ...)))
true
```

6. Comparison with TWO Other Programming Languages

6.1. Tuples in Python

In Python, tuples are immutable sequences, meaning their elements cannot be modified once created. When a tuple object is created, the Python interpreter calculates the required memory size to accommodate the tuple and allocates a contiguous block of memory on the heap. The individual elements of the tuple are allocated within the same block of memory, preserving their order. The values of the elements are assigned within the allocated memory block. After creation, a reference to the allocated memory block is returned.

6.2. Vectors in C++

In C++, vectors are dynamic arrays that can grow or shrink in size at runtime. When a vector is declared, the C++ compiler generates code that automatically allocates memory for the vector on the heap. Initially, the total number of elements the vector can hold without resizing (capacity) is determined based on the specified or default initial size. The vector object contains a pointer to the dynamically allocated memory block, which is initially empty. C++ allows adding new elements to a vector or removing existed elements from one, with it resized on demand efficiently.

6.3. Comparison with Tuples in the Egg Eater Language

Since the tuples cannot be changed after creation, they are more like tuples in Python than vectors in C++. Also, the heap memory allocated to tuples will not change after their allocation, which also makes them more like tuples in Python.

7. References

How to Insert into a BST without Modifying Existed Data Structure

Memory Allocation of Vectors in C++

Memory Allocation of Tuples in Python