

CSE 231 - Egg Eater

Introduction and syntax

This is the design document for the **Egg Eater** language. It builds on syntax from the [CSE 231 Diamondback Language](#). Our language is written with the following syntax, with the new syntax supporting heap-allocated data denoted below:

Type	Syntax
<prog>	<defn>* <expr>
<defn>	(fun (<name> <name>) <expr>)
	(fun (<name> <name> <name>) <expr>)
	(fun (<name> <name> <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>)
	(<names> <exprs>*)

Type	Syntax
	<code>(tuple <expr+>) *(new)</code>
	<code>(index <expr> <expr>) *(new)</code>
	<code>(set-tup! <expr> <expr> <expr>) *(new)</code>
<op1>	<code>add1</code>
	<code>sub1</code>
	<code>isnum</code>
	<code>print</code>
<op2>	<code>-</code>
	<code>+</code>
	<code><</code>
	<code>></code>
	<code>=</code>
	<code>== *(new)</code>
<binding>	<code>(<identifier> <expr>)</code>

As you can see, we've added three primitive expressions, one primitive value, and one binary operator:

- `tuple <expr+>` Allocates memory on the heap for an arbitrary number of `expr` 's which are then stored contiguously. The expression evaluates to the address at which the data can be located using `index` . Empty `tuples` are not allowed, and causes a parsing error. Pointer arithmetic is not allowed and will cause a runtime exception. Equality using the `=` is based on *reference equality*, i.e. `(= tup1 tup2)` evaluates to `true` if both point to the same address.
- `index <expr> <expr>` The first argument to `index` evaluates to a heap-allocated value, i.e. the address of the tuple we would like to index. The second evaluates to a number and the value at that index is returned. Note: our tuples are `1` -indexed; indexing a `tuple` at `0` returns the length of that `tuple` . Negative indexing or indexing out of the bounds of the `tuple` 's allocated causes a runtime exception.
- `set-tup! <expr> <expr> <expr>` Structurally updates an index of a tuple (tuples are mutable in this language `¬(ヾ)/¬`). The two arguments evaluate similarly to `index` , i.e. the address and position we

want to set (starting at 1). The third argument is the value which we want to set the respective index to.

- `nil` Evaluates to a *null pointer*, pointing to the address `0x0` . Causes a runtime exception when passed as an argument to `index` or `set-tup!` .
- `==` Evaluates the structural equality of the two operands. We will explain our approach to structural equality in the later section **Structural Equality**. Note that our implementation allows for the comparison of different types.

How values and metadata are arranged on the heap

Consider the following code:

```
(block
  (let (x (tuple 1 2 3)) (x)) # Expr1
  (let (y (tuple 4 5)) (y)) # Expr2
)
```

Below is a diagram depicting how our language arranges heap-allocated values and metadata when the above code is run. Supposing that `r15` starts at `0x1000` , we can see that for each `tuple` the memory is stored contiguously, with the length metadata taking the first slot of memory. The memory is placed as aligned to an `8-byte` address, so each value gets `64` bits. Finally, we can see that the heap address is allocated in increasing order, with tuples that were allocated earlier being given lower addresses on the heap.

Memory Address	Allocated Value	
0x1000	0x6 (3)	<— x=0x1001
-----	-----	-----
0x1010	0x2 (1)	
-----	-----	-----
0x1018	0x4 (2)	<— r15 after Expr1
-----	-----	-----
0x1020	0x4 (2)	<— y=0x1021
-----	-----	-----

Memory Address	Allocated Value	
0x1028	0x8 (4)	
-----	-----	-----
0x1030	0xA (10)	\leftarrow r15 after Expr2
-----	-----	-----
...
-----	-----	-----
0x2000		End of allocated memory
-----	-----	-----

Structural Equality

Our language evaluates the structural equality operator `==` through the runtime environment. Consider the following snippet of assembly code for

```
(== tup1 tup2)
```

Outsourcing to the runtime environment

```
; Calling snek_eq with Id("tup1"), Id("tup2")
mov rax, [rsp + 0]
mov [rsp + 24], rax
mov rax, [rsp + 8]
sub rsp, 24
mov [rsp+16], rdi
mov rsi, rax

mov rax, [rsp+48]
mov rdi, rax
call snek_eq ; Here we call the runtime function snek_eq
```

```
mov rdi, [rsp+16]
add rsp, 24
```

snek_eq and snek_eq_mut

Our call to `snek_eq` immediately utilizes a helper function `snek_eq_mut` to do the computation.

```
pub extern "C" fn snek_eq(val1: i64, val2: i64) -> i64 {
    snek_eq_mut(val1, val2, &mut Vec::new())
}
```

In `snek_eq_mut`, our approach to checking the structural equality of two tuples is as follows:

1. Type checking: Make sure the values we are working with both non-nil tuples (equality is still defined for non-tuples, but it isn't really worth discussing here as it reduces to standard equality checking)
2. Cycle checking: We keep a list of the addresses we've seen in *both* tuples. This is so that if the tuples have the same *cyclic* structure (i.e. with respect to the depth of unravelling the infinitely nested tuples), we will return true. We then add this pair of addresses (below this is `val1`, `val2`) to the `seen` list.
3. Check equality of each element: We check the equality of each element in the respective tuples and the equality of each respective element is evaluated recursively, to make sure we reflect the full depth of the tuple. Note that what we are looking for is a witness *against* equality, i.e. a pair of elements that evaluate to not equal in the tuple.
4. Pop and return: After we've checked each element of the tuple, we can verifiably assume that there is no cyclic equality, so we pop our pair of addresses from our `seen` list. More importantly, we've failed to find a witness against equality and from the previous step we've exhausted both tuples (even if they are infinitely sized), so we return `true`.

```
fn snek_eq_mut(val1 : i64, val2 : i64, seen : &mut Vec<(i64, i64)>) -> i64 {
```

```
    // Step 1
    if (val1 & 7) != (val2 & 7) {
        return 3; // Different types
```

```
}

// If not tuples, then we don't have to do anything special
if (val1 & 1 != 1) || (val2 & 1 != 1) {
    return if val1 == val2 {7} else {3};
}

// If nil, then we don't have to do anything special
if (val1 == 1) || (val2 == 1) {
    return if val1 == val2 {7} else {3};
}

// Now we know that they are:
//   - Same Types
//   - Tuples
//   - Non-Nil

// Step 2
if seen.contains(&(val1, val2)) { return 7 }
seen.push((val1, val2));

// Step 3
let addr1 = (val1 - 1) as *const i64;
let addr2 = (val2 - 1) as *const i64;

// Note: pair will NOT print correctly anymore
let size1: i64 = unsafe { *addr1 } / 2;
let size2: i64 = unsafe { *addr2 } / 2;

if size1 != size2 {
    return 3;
}

for i in 1..(size1 + 1) {
    let el1 = unsafe { *addr1.offset(i as isize) };
```

```

    let e12 = unsafe { *addr2.offset(i as isize) };
    if snek_eq_mut(e11, e12, seen) == 3 {
        return 3;
    }
}

// Step 4
seen.pop();
return 7;
}

```

Structural Equality: Remark on Cycles

Our language evaluates structural equality by searching for a witness *against* equality; if we find one, return **false**, if we don't return **true**. Thus, we evaluate the equality of cycles by implicitly checking if, were we to unravell them infinitely, all the elements are the same. As stated above this is why we keep the **seen** list, because if we have found a pair of addresses that we've already seen before and we have not found a witness against equality, we will *never* find one, by the cyclic nature of the tuples we're checking.

Example: Constructing and accessing heap-allocated data

Code

```

(let (tup (tuple 11 102 53 42 15)) (block
  (print tup)
  (print (index tup 2))
  (print (index tup 0))
))

```

Output

```
(base) ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ test/simple_examples.run  
(tuple 11 102 53 42 15)  
102  
5  
5
```

This program constructs a **tuple** called **tup** and then indexes it at **2**. Notice that when we call **(index tup 0)**, we get the length of the array.

Example: Runtime tag-checking error

Code

```
(index 10 10)
```

Output

```
(base) ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ test/error-tag.run  
An error occurred: invalid argument!
```

Here, we try to index something that is not a **tuple**. We get an illegal argument error.

Example: Indexing out-of-bounds

Code

```
(let (tup (tuple 11 102 53 42 15)) (block  
  (print tup)  
  (print (index tup 2))  
  (print (index tup 0))  
  (print (index tup 6))  
))
```

Output


```
(base) ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ test/error-bounds.run
(tuple 11 102 53 42 15)
102
5
An error occurred: index out of bounds!
```

Here we correctly index the `tuple`, `tup`, but on the last line we try to get the 6th element, which is outside the bounds of the `tuple`, giving a runtime error.

Example: Indexing a `nil` pointer

Code

```
(index nil 0)
```

Output

```
Ⓜ (base) ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ test/error3.run
An error occurred: null pointer exception!
```

Here we try to index a `nil` pointer, which gives a runtime error.

Example: Points

Code

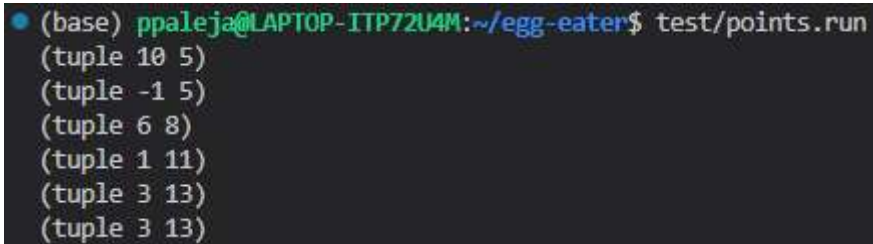
```
; Define the point structure
(fun (make_point x y)
  (tuple x y))

; Define a function to add two points
(fun (add_points point1 point2)
  (make_point (+ (index point1 1) (index point2 1))
              (+ (index point1 2) (index point2 2))))

; Test the functions
```

```
(block
  (print (make_point 10 5))
  (print (make_point -1 5))
  (print (add_points (make_point 2 3) (make_point 4 5)))
  (print (add_points (make_point 2 3) (make_point -1 8)))
  (print (add_points (make_point 4 5) (make_point -1 8)))
)
```

Output



```
(base) ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ test/points.run
(tuple 10 5)
(tuple -1 5)
(tuple 6 8)
(tuple 1 11)
(tuple 3 13)
(tuple 3 13)
```

Here we have a program with a function that takes an **x** and a **y** coordinate and produces a tuple with those values, and a function that takes two points and returns a new point with their **x** and **y** coordinates added together. We show how we can make points and add them together.

Example: Binary Search Tree

Code

```
; Function to create a binary search tree node with 3 values: value, left subtree, right
subtree
(fun (make_node value left right)
  (tuple value left right))

(fun (insert value tree)
  (if (= tree nil)
    (make_node value nil nil)
    (if (< value (index tree 1))
      (make_node (index tree 1) (insert value (index tree 2)) (index tree 3))
```

```

    (if (> value (index tree 1))
      (make_node (index tree 1) (index tree 2) (insert value (index tree 3)))
      tree))))

; Function to check if an element exists in a binary search tree
(fun (contains? value tree)
  (if (= tree nil)
    false
    (let (node_value (index tree 1)) (
      let (left_tree (index tree 2)) (
        let (right_tree (index tree 3)) (
          if (= value node_value)
            true
            (if (< value node_value)
              (contains? value left_tree)
              (contains? value right_tree))))))))

; Test: Create a binary search tree, insert an element, and check if it exists in the tree
(
  (let (tree (print (make_node 5 nil nil))) (block
    (print (set! tree (insert 7 tree)))
    (print (set! tree (insert 3 tree)))
    (print (set! tree (insert 1 tree)))
    (print (set! tree (insert 9 tree)))
    (print (contains? 5 tree))
    (print (contains? 3 tree))
    (print (contains? 9 tree))
    (print (contains? 4 tree))
    (print (contains? 0 tree))
  ))
)

```

Output

```

• (base) ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ test/bst.run
(tuple 5 nil nil)
(tuple 5 nil (tuple 7 nil nil))
(tuple 5 (tuple 3 nil nil) (tuple 7 nil nil))
(tuple 5 (tuple 3 (tuple 1 nil nil) nil) (tuple 7 nil nil))
(tuple 5 (tuple 3 (tuple 1 nil nil) nil) (tuple 7 nil (tuple 9 nil nil)))
true
true
true
false
false
false

```

An implementation of the binary search tree data structure using our `tuple` expression. Note that when calling `insert`, we create a new node using the information of the last tree, so each time we are allocating a new tree, not updating the existing one. We include some tests of the various functions, which both add to both sides of the BST and check for contains in the root, leafs, and body of the tree for elements that are/are not there.

Example: Structural and reference equality on *non-cyclic* values

Code

```

(let (tup1 (tuple 1 2 3))
  (let (tup2 (tuple 1 2 3))
    (let (tup3 (tuple 4 5 6))
      (block

        ; Surface level equality
        (print (== tup1 tup2))
        (print (== tup1 tup3))

        ; Reference equality
        (print (= tup1 tup1))
        (print (= tup1 tup2))

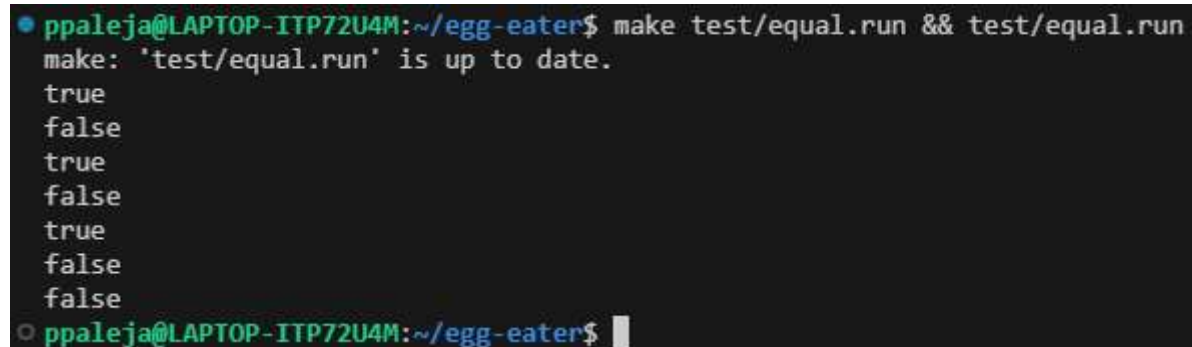
        ; Next level equality
        (set-tup! tup1 1 (tuple 1))

```

```
(set-tup! tup2 1 (tuple 1))
(print (== tup1 tup2))

; Next level inequality
(set-tup! tup2 1 (tuple 2))
(print (== tup1 tup2))))))
```

Output



```
ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ make test/equal.run && test/equal.run
make: 'test/equal.run' is up to date.
true
false
true
false
true
false
false
ppaleja@LAPTOP-ITP72U4M:~/egg-eater$
```

Here we show tests for reference and structural equality at various levels of depth.

Cyclic Example: Self-Cycle

Code: Printing

```
(let (p (tuple 1 2))
  (let (p2 (tuple 1 2))
    (let (p3 (tuple 2 3)) (block
      (set-tup! p 1 p)
      (set-tup! p2 1 p2)
      (set-tup! p3 1 p3)
      (print p)
      (print p2)
      (print p3))))))
```

Output

```
ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ test/cycle-print1.run
(tuple (tuple <cyclic>) 2)
(tuple (tuple <cyclic>) 2)
(tuple (tuple <cyclic>) 3)
(tuple (tuple <cyclic>) 3)
ppaleja@LAPTOP-ITP72U4M:~/egg-eater$
```

Code: Equality

```
(let (p (tuple 1 2))
  (let (p2 (tuple 1 2))
    (let (p3 (tuple 2 3)) (block
      (set-tup! p 1 p)
      (set-tup! p2 1 p2)
      (set-tup! p3 1 p3)
      (print (== p p2))
      (print (== p p3))))))
```

Output

```
ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ make test/cycle-equal1.run && test/cycle-equal1.run
make: 'test/cycle-equal1.run' is up to date.
true
false
false
```

Here we have three tuples which both refer back to themselves in the second element. They all clearly have the same cyclic structure, so we should expect the first one to return true, because it also has the same elements. However the `p` and `p3` have different elements in the second position, even though they have the same self-referential structure, so we return `false`.

Cyclic Example: Cross-Referential Self-Cycles

Code: Printing

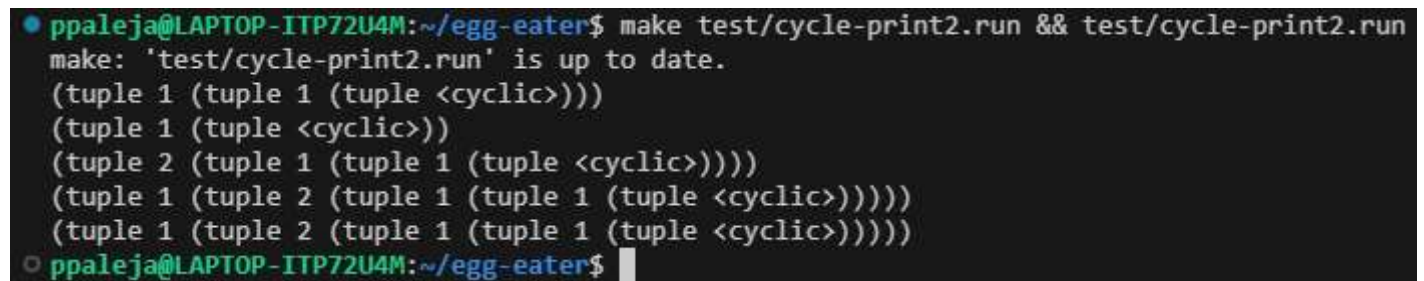
```
(let (p (tuple 1 2))
  (let (p2 (tuple 1 2))
```

```

(let (p3 (tuple 2 p))
  (let (p4 (tuple 1 p3)) (block
    (set-tup! p 2 p2)
    (set-tup! p2 2 p2)
    (print p)
    (print p2)
    (print p3)
    (print p4))))))

```

Output



```

● ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ make test/cycle-print2.run && test/cycle-print2.run
make: 'test/cycle-print2.run' is up to date.
(tuple 1 (tuple 1 (tuple <cyclic>)))
(tuple 1 (tuple <cyclic>))
(tuple 2 (tuple 1 (tuple 1 (tuple <cyclic>))))
(tuple 1 (tuple 2 (tuple 1 (tuple 1 (tuple <cyclic>))))))
(tuple 1 (tuple 2 (tuple 1 (tuple 1 (tuple <cyclic>))))))
○ ppaleja@LAPTOP-ITP72U4M:~/egg-eater$

```

Code: Equality

```

(let (p (tuple 1 2))
  (let (p2 (tuple 1 2))
    (let (p3 (tuple 2 p))
      (let (p4 (tuple 1 p3)) (block
        (print (== p p2))
        (print (== p p3))
        (set-tup! p 2 p2)
        (set-tup! p2 2 p2)
        (print (== p p2))
        (print (== p p4))))))

```

Output

```

• ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ make test/cycle-equal2.run && test/cycle-equal2.run
make: 'test/cycle-equal2.run' is up to date.
true
false
true
false
false

```

Here we have an example of equality and inequality that are shown to then persist through depth-1. We have that before our `set-tup!` operations, `p` and `p2` were equal to each other, so setting the second element of both to `p2` should maintain structural equality, as is reflected in the output. The same is true of the inequality of `p` and `p3`, whose inequality also persists.

Cyclic Example: Cross-Cycles

Code: Printing

```

(let (x1 (tuple 10 20 30))
  (let (x2 (tuple 10 20 30))
    (let (x3 (tuple 10 20 30))
      (let (y1 (tuple 10 20))

        (block
          (set-tup! x1 2 x2)
          (set-tup! x2 2 x1)
          (set-tup! x3 2 x2)
          (set-tup! y1 2 x2)
          (print x1)
          (print x2)
          (print x3)
          (print y1))))))

```

Output


```

● ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ make test/cycle-print3.run && test/cycle-print3.run
make: 'test/cycle-print3.run' is up to date.
(tuple 10 (tuple 10 (tuple <cyclic>) 30) 30)
(tuple 10 (tuple 10 (tuple <cyclic>) 30) 30)
(tuple 10 (tuple 10 (tuple 10 (tuple <cyclic>) 30) 30) 30)
(tuple 10 (tuple 10 (tuple 10 (tuple <cyclic>) 30) 30))
(tuple 10 (tuple 10 (tuple 10 (tuple <cyclic>) 30) 30))

```

Code: Equality

```

(let (x1 (tuple 10 20 30))
  (let (x2 (tuple 10 20 30))
    (let (x3 (tuple 10 20 30))
      (let (y1 (tuple 10 20))

        (block
          (set-tup! x1 2 x2)
          (set-tup! x2 2 x1)
          (set-tup! x3 2 x2)
          (set-tup! y1 2 x2)
          (print (== x1 x2))
          (print (== x1 x3))
          (== x1 y1))))))

```

Output

```

● ppaleja@LAPTOP-ITP72U4M:~/egg-eater$ make test/cycle-equal3.run && test/cycle-equal3.run
make: 'test/cycle-equal3.run' is up to date.
true
true
false

```

One can think about **x1**, **x2**, **x3** as being in various stages of unraveling the cycle. However, since the cycles are infinite, one can continue the unravelling and see that they all have the same structure, i.e. a cyclic tuple nested in between an equal amount of 10s and 30s. However, **y1** has conflicting depth because of its second value so we return false, even though unravelling it we would count the “same” number of 10’s and 30’s. This conveys that our equality accurately reflects the *structure* of the tuple, and that the depth of the infinite cycle is respected.

Similarities/differences to other programming languages

- MATLAB:
 - Arrays allocated contiguously in memory,
 - Arrays must contain elements of the same type,
 - Indexed starting at 1,
 - Len is always known by the array,
- C:
 - Arrays allocated contiguously in memory,
 - Arrays must contain elements of the same type,
 - Indexed starting at 0,
 - Len is NOT always known by the array (unfortunately for Wells Fargo),

Based on this comparison, our language is more similar to MATLAB in how it supports heap-allocated data, as it has all the above characteristics EXCEPT that our tuples can have different types of elements in it simultaneously.

Resources

- Some tests were constructed with ChatGPT using the prompts from the egg eater [test specs](#).
- As mentioned the start-off point for this codebase is from the [CSE 231 Diamondback Language](#).
- The following EdStem posts were used by the author for help, information, and test ideas:
 - <https://edstem.org/us/courses/38748/discussion/3152183>
 - <https://edstem.org/us/courses/38748/discussion/3150092>
 - <https://edstem.org/us/courses/38748/discussion/3150044>
 - <https://edstem.org/us/courses/38748/discussion/3142607>
 - <https://edstem.org/us/courses/38748/discussion/3191257>
 - <https://edstem.org/us/courses/38748/discussion/3138761>
 - <https://edstem.org/us/courses/38748/discussion/3199653>