

# 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>)

Type	Syntax
	( <code>&lt;name&gt;</code> <code>&lt;expr&gt;*</code> )
	( <code>tuple &lt;expr+&gt;</code> ) <i>*(new)</i>
	( <code>index &lt;expr&gt;</code> <code>&lt;expr&gt;</code> ) <i>*(new)</i>
<code>&lt;op1&gt;</code>	<code>add1</code>
	<code>sub1</code>
	<code>isnum</code>
	<code>print</code>
<code>&lt;op2&gt;</code>	<code>-</code>
	<code>+</code>
	<code>&lt;</code>
	<code>&gt;</code>
	<code>=</code>
<code>&lt;binding&gt;</code>	( <code>&lt;identifier&gt;</code> <code>&lt;expr&gt;</code> )

As you can see, we've added two primitive expressions and one primitive value:

- `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.
- `nil` Evaluates to a *null pointer*, pointing to the address `0x0` . Causes a runtime exception when passed as an argument to `index` .

# 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
-----	-----	-----
0x1028	0x8 (4)	
-----	-----	-----
0x1030	0xA (10)	<— r15 after Expr2
-----	-----	-----
...	...	...

Memory Address	Allocated Value	
-----	-----	-----
0x2000		End of allocated memory
-----	-----	-----

## 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:

```
❶ (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:

```
❶ (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

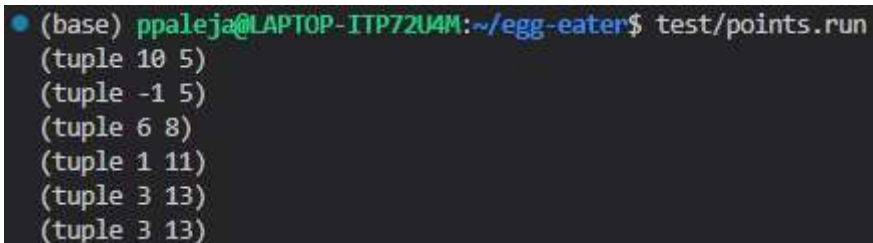
## Code:

```
; 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))

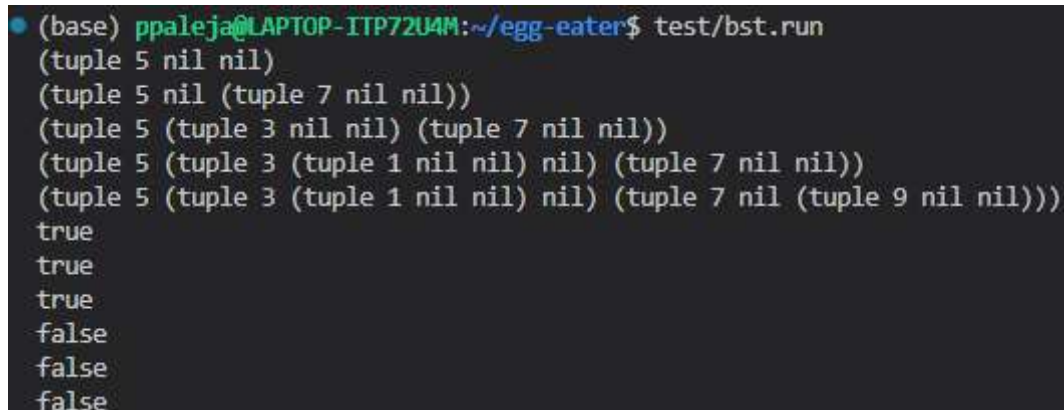
(fun (insert value tree)
  (if (= tree nil)
    (make_node value nil nil)
    (if (< value (index tree 1))
      (make_node (index tree 1) (insert value (index tree 2)) (index tree 3))
      (if (> value (index tree 1))
        (make_node (index tree 1) (index tree 2) (insert value (index tree 3)))
        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
e
(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.

## 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

- The 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 and information:
  - <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>