

# CSE 231 SP23 - Egg Eater

## 1 EggEater's Concrete Syntax

The concrete syntax followed by Egg Eater is as follows:

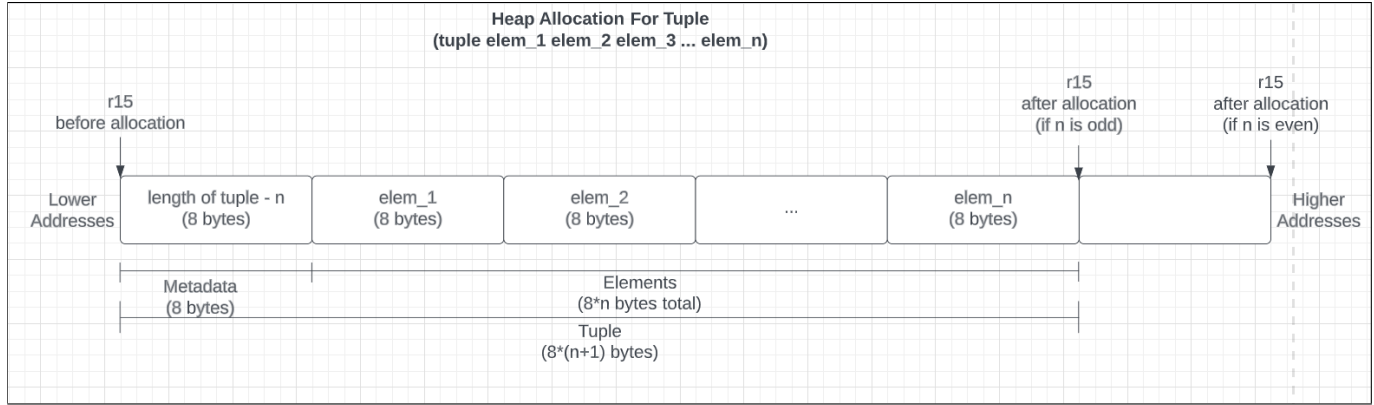
```
1 <prog> := <defn>* <expr>
2 <defn> := (fun (<name> <name>*) <expr>)
3 <expr> :=
4   | <number>
5   | true
6   | false
7   | input
8   | <identifier>
9   | nil (new!)
10  | (tuple <expr>+) (new!)
11  | (tuple-set! <expr> <expr> <expr>) (new!)
12  | (let (<binding>+) <expr>)
13  | (<op1> <expr>)
14  | (<op2> <expr> <expr>)
15  | (set! <name> <expr>)
16  | (if <expr> <expr> <expr>)
17  | (block <expr>+)
18  | (loop <expr>)
19  | (break <expr>)
20  | (<name> <expr>*)
21
22 <op1> := add1 | sub1 | isnum | isbool | print
23 <op2> := + | - | * | < | > | >= | <= | = | tuple-get (new!) | deep-equal (new!)
24
25 <binding> := (<identifier> <expr>)
```

The language expands on a previous implementation of a compiler for a functional language (diamondback) by allowing developers to create heap-allocated tuples of arbitrary sizes. The language also introduces a tuple-get operation so as to expand the capabilities of the tuple data structure. This binary operation accepts a heap allocated address as its first argument and a number as its second argument, and returns the element stored in the tuple corresponding to the heap address (first argument) at the index given by the number (second argument), using 0-based indexing. The language further expands on the capabilities of the tuple data structure by allowing them to be mutable through the tuple-set! expression. This operation accepts a tuple as its first parameter, an index as a number as its second parameter and a value of any type (number, boolean, or tuple) as its third parameter, and sets the element at that given index in the tuple to hold the new value. While the = operator checks for referential equality between two values with type safety, the newly introduced *deep-equal* operator checks for structural equality between two values, with the drawback of increased runtime resources and no type-safeness (discussed further below). Finally, the language also introduces a “nil” identifier to allow developers to create uninitialized heap allocated data structures.

## 2 InOrder, Contiguous and 16 byte Aligned Heap Allocation

At any point during the compilation of the program, the *r15* register holds the value of the next available address that can be used for heap allocation. When an individual tuple is being created, it is allocated a contiguous section of the memory starting at the address stored in *r15*. The first 8 bytes from *r15* store the

length of the tuple as necessary metadata for future operations involving the tuple. The  $n$  elements of the tuple are then stored in the next  $n$  8 byte locations immediately after the location that the length was stored.  $r15$  is then updated to store the next 16-byte aligned address immediately after the tuple for future allocations. The allocation operation of a tuple also returns its heap address to allow for future references to the newly allocated heap data structure. The following diagram depicts how individual tuples are allocated in the heap:



As mentioned above, the first 8 bytes of memory corresponding to the tuple's heap address store the length of the tuple, and the elements of the tuple are stored immediately after the length in their own individual 8 byte segments. Finally,  $r15$  is updated to point to the smallest 16 byte aligned address immediately after the address of the last element of the newly allocated tuple. If the tuple has an odd length, then  $r15$  would be updated to store the address of the next byte in memory immediately after the 8 bytes corresponding to the last element of the tuple. On the other hand, if the tuple has an even length, then  $r15$  would be updated to store the address of the 8th byte in memory after the 8 bytes corresponding to the last element of the tuple. Thus tuples with smaller addresses were allocated in memory before tuples with larger addresses. Further, if a tuple contains one or more tuples as its elements, to maintain the above arrangement of memory, the inner tuples are allocated in memory first before the outer tuple, and the outer tuple stores the references to the inner tuple elements in the elements' corresponding 8 byte segments within the outer tuple's allocated segment.

## 3 Testing

The following tests were implemented to test the correctness of the EggEater compiler.

### 3.1 Simple Examples

```

1 (fun (tuplemaker single) (tuple single))
2 (
3   block
4   (print (tuple 3))
5   (print (tuple -1 5))
6   (print (tuple -1 true 4 false))
7   (print (let ((x (tuple 100 70 true (+ 5 5)))) x))
8   (print (let ((x (tuple (tuple 3)))) x))
9   (print (let ((x (tuple 3 (tuple 1 (tuple 100) (tuple true false 3 4))))) x))
10  (
11    let (
12      (x (tuple (tuple 1) (tuple (tuple 3 4) 5 (tuple 6 7))))
13      (y (tuple true input 3 false))
14      (z (tuple y x))

```

```

15         )
16
17     (
18         block
19         (print x)
20         (print (tuple-get x 1))
21         (print (tuple-get (tuple-get x 1) 2))
22         (print (tuple-get (tuple-get (tuple-get x 1) 0) 1))
23         (print (= (tuple-get (tuple-get (tuple-get x 1) 0) 0) (tuple-get y 2)))
24         (print (= x (tuple-get z 1)))
25         (print (= y (tuple true input 3 false)))
26         (print (= y nil))
27         (print (tuplemaker 10))
28         (isbool (tuple 3))
29     )
30 )
31 )

```

(Source: [https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/simple\\_examples.snek](https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/simple_examples.snek))

The goal of this test file was to test the different cases regarding creating, allocating and accessing heap allocated data structures in the EggEater language. Each of the print lines in the above test file (as well as the last *isbool* expression) represent an individual test regarding the creation or accessing of heap allocated data structures for unique scenarios. The output from executing this Egg Eater program was as follows:

```

1 (tuple 3)
2 (tuple -1 5)
3 (tuple -1 true 4 false)
4 (tuple 100 70 true 10)
5 (tuple (tuple 3))
6 (tuple 3 (tuple 1 (tuple 100) (tuple true false 3 4)))
7 (tuple (tuple 1) (tuple (tuple 3 4) 5 (tuple 6 7)))
8 (tuple (tuple 3 4) 5 (tuple 6 7))
9 (tuple 6 7)
10 4
11 true
12 true
13 false
14 false
15 (tuple 10)
16 false

```

A quick analysis of the tests suggests that the compiler works as expected:

1. `(print (tuple 3))`
  - (a) This test simply creates a tuple with just a single element and prints it.
  - (b) The expected output of `(tuple 3)` was received.
2. `(print (tuple -1 5))`
  - (a) This test creates a tuple with two elements and prints it.
  - (b) The expected output of `(tuple -1 5)` was received.
3. `(print (tuple -1 true 4 false))`
  - (a) This test creates a tuple with multiple elements that are either numbers or booleans and prints it.

- (b) The expected output of `(tuple -1 true 4 false)` was received, since we don't limit tuples to only store a single data type.
4. `(print (let ((x (tuple 100 70 true (+ 5 5)))) x))`
- (a) This test creates a tuple and binds it to an identifier, and then calls `print` on the corresponding identifier.
- (b) The expected output of `(tuple 100 70 true 10)` was received, showing that the compiler correctly detects if identifiers are storing heap addresses instead of numeric or boolean constants.
5. `(print (let ((x (tuple (tuple 3)))) x))`
- (a) This test creates a tuple within a tuple and then calls `print` on the simple nested tuple.
- (b) The expected output of `(tuple (tuple 3))` was received, showing that the compiler is able to support nested tuple definitions.
6. `((print (let ((x (tuple 3 (tuple 1 (tuple 100) (tuple true false 3 4))))) x))`
- (a) This test creates a more complex nested tuple structure and calls `print` on the outermost tuple.
- (b) The expected output of `(tuple 3 (tuple 1 (tuple 100) (tuple true false 3 4)))` was received, showing that the compiler is able to support even complex nested tuple definitions.
7. `(print x)`
- (a) `x` is a complex nested tuple structure defined within a `let` binding. This test verifies that the creation, allocation, binding, de-referencing and printing of complex nested tuples works as expected.
- (b) The expected output of `(tuple (tuple 1) (tuple (tuple 3 4) 5 (tuple 6 7)))` was received, showing that the compiler is able to support complex nested tuple definitions.
8. `(print (tuple-get x 1))`
- (a) This test verifies the correctness of the `tuple-get` operation by attempting to access and print the second element of a tuple, which is also another tuple.
- (b) The expected output of `(tuple (tuple 3 4) 5 (tuple 6 7))` was received showing that the `tuple-get` operation correctly works with complex nested tuple definitions.
9. `(print (tuple-get (tuple-get x 1) 2))`
- (a) This test verifies the correctness of the `tuple-get` operation by attempting to index into the output of another `tuple-get` operation on a nested tuple.
- (b) The expected output of `(tuple 6 7)` was received showing that the `tuple-get` operation correctly works with complex nested tuple definitions.
10. `(print (tuple-get (tuple-get (tuple-get x 1) 0) 1))`
- (a) This test further verifies the correctness of the `tuple-get` operation by nesting `tuple-get` operations on a complex nested tuple definition.
- (b) The expected output of 4 since it is the second element within the first element of the second element of `x`.

11. `(print (= (tuple-get (tuple-get (tuple-get x 1) 0) 0) (tuple-get y 2)))`
- (a) Due to the introduction of a new type, the manner in which types were being stored in memory as 8 byte segments also had to change. This test ensures the correctness of both the new operation compilation, as well as the `tuple-get` operation compilation.
  - (b) The expected output of `true` was received since that specific element within the definition of `x` was 3 and that was also the value of the specific element being lookedup from `y`.
12. `(print (= x (tuple-get z 1)))`
- (a) This test verifies the correctness of the `=` operation on tuple arguments in that it checks for referential equality.
  - (b) The expected output of `true` was received since the second element of `z` was exactly `x`.
13. `(print (= y (tuple true input 3 false)))`
- (a) This test verifies the correctness of the `=` operation on tuple arguments in that it checks only for referential equality.
  - (b) The expected output of `false` was received since even though the elements of both the LHS and RHS operations are the same, they are not equal referentially (they are only equal structurally).
14. `(print (= y (tuple true input 3 false)))`
- (a) This test verifies the correctness of the `=` operation on tuple arguments in that it checks only for referential equality.
  - (b) The expected output of `false` was received since even though the elements of both the LHS and RHS operations are the same, they are not equal referentially (they are only equal structurally).
15. `(print (= y nil))`
- (a) This test verifies the correctness of the `=` operation on the `nil` identifier.
  - (b) The expected output of `false` was received since `y` is not `nil`.
16. `(print (tuplemaker 10))`
- (a) This test verifies the correctness of the creation of tuples by functions, as well as its succesful returning between functions.
  - (b) The expected output of `(tuple 10)` was received showing that the compiler correctly returns tuples from functions.
17. `(isbool (tuple 3))`
- (a) In the compiler heap allocated addresses were made type secure from numbers by making their last bits 1, while on the other hand numbers always had their last bits set to 0. This test ensures that tuples are not confused with booleans, since even booleans are being type secured by setting their last bit to 1 (`true` was stored as `0b111` and `false` was stored as `0b1`)
  - (b) The expected output of `false` was received since tuples aren't booleans, showing that the language can successfully differentiate between tuples and booleans.

## 3.2 Error Tag

```
1 (+ (tuple 3) 3)
```

(Source: <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/error-tag.snek>)

The above test validates that a runtime error is thrown during the execution of the program, since the language should not allow the addition of tuples and numbers. This has to be a runtime error, since the language implements runtime type checking. The expected output of “invalid argument occurred with error code: 2” was received since the + operation received an argument that wasn’t a number (the first argument). The compiler introduces the relevant runtime type-checking logic into the code as follows. The following code implements the compilation for + expressions.

```
1 match op {
2     Op2::Plus => {
3         assert_both_num(instrs, Val::StaticRegOffset(RSP, 8 * context.si), Val::Reg(
4             RAX));
5         instrs.push(Instr::IAdd(
6             Val::Reg(RAX),
7             Val::StaticRegOffset(RSP, 8 * context.si),
8         ));
9         check_for_overflow(instrs);
10    }
11    ...<compilation of other operations>...
12
13 fn assert_both_num(instrs: &mut Vec<Instr>, r1: Val, r2: Val) {
14     instrs.push(Instr::IMov(Val::Reg(RBX), r1));
15     instrs.push(Instr::IOp(Val::Reg(RBX), r2));
16     instrs.push(Instr::IAnd(Val::Reg(RBX), Val::Imm(1)));
17     instrs.push(Instr::ITest(Val::Reg(RBX), Val::Imm(1)));
18     instrs.push(Instr::IMov(Val::Reg(RBX), Val::Imm(INVALID_ARGUMENT_CODE)));
19     instrs.push(Instr::IJnz(ERROR_LABEL.to_string()));
20 }
```

(Source: <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/src/compiler.rs>)

As we can see before the Add instruction is introduced to the compiled instructions, relevant logic is introduced to the instructions first that handles type checking the operands to ensure that both operands are numbers. The execution of these instructions causes the program to jump to the ERROR\_LABEL label, that handles throwing runtime errors.

## 3.3 Error Bounds

```
1 (tuple-get (tuple 3 7) 2)
```

(Source: <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/error-bounds.snek>)

The above test validates that a runtime error is thrown during the execution of the program, since the developer was attempting to lookup/index an element of the tuple outside its actual size. This has to be a runtime error, since the value of the index argument will only be resolved during the execution of the program. The expected output of “out of bounds error occurred with error code: 3” was received since the tuple-get operation received an index (second argument) that was larger than the tuple’s (first argument’s) length. The compiler introduces the relevant runtime logic for out of bounds checking into the code as follows. The following code implements the compilation for tuple-get expressions.

```
1 Op2::TupleGet => {
2     /* tuple is in RAX, index is in RSP + 8*context.si */
```

```

3
4 // Check if RAX hold a heap address
5 assert_heap_address(instrs);
6 // Copy index from RSP + 8*context.si into RBX and check if it is a number
7 instrs.push(Instr::IMov(Val::Reg(RBX), Val::StaticRegOffset(RSP, 8*context.si)
8 ));
9 assert_num(instrs, Val::Reg(RBX));
10 // Check if it is nil and throw out of bounds
11 instrs.push(Instr::ICmp(Val::Reg(RAX), Val::Imm(1)));
12 instrs.push(Instr::IMov(Val::Reg(RBX), Val::Imm(OUT_OF_BOUNDS_ERROR_CODE)));
13 instrs.push(Instr::IJe(ERROR_LABEL.to_string()));
14 // Throw runtime error if index is less than 0
15 instrs.push(Instr::IMov(Val::Reg(RBX), Val::StaticRegOffset(RSP, 8*context.si)
16 ));
17 instrs.push(Instr::ICmp(Val::Reg(RBX), Val::Imm(0)));
18 instrs.push(Instr::IMov(Val::Reg(RBX), Val::Imm(OUT_OF_BOUNDS_ERROR_CODE)));
19 instrs.push(Instr::IJl(ERROR_LABEL.to_string()));
20 // Throw runtime error if index is greater than or equal to length of tuple
21 instrs.push(Instr::IMov(Val::Reg(RBX), Val::StaticRegOffset(RSP, 8*context.si)
22 ));
23 instrs.push(Instr::ICmp(Val::Reg(RBX), Val::StaticRegOffset(RAX, -1)));
24 instrs.push(Instr::IMov(Val::Reg(RBX), Val::Imm(OUT_OF_BOUNDS_ERROR_CODE)));
25 instrs.push(Instr::IJge(ERROR_LABEL.to_string()));
26 // Move element from heap at index location into rax
27 instrs.push(Instr::IMov(Val::Reg(RBX), Val::StaticRegOffset(RSP, 8*context.si)
28 ));
29 instrs.push(Instr::ISar(Val::Reg(RBX), Val::Imm(1))); // Shift the index to
30 the right to get the "correct index value"
31 instrs.push(Instr::IAdd(Val::Reg(RBX), Val::Imm(1))); // Add in for metadata
32 instrs.push(Instr::IMul(Val::Reg(RBX), Val::Imm(8))); // Calculate actual heap
33 offset
34 instrs.push(Instr::ISub(Val::Reg(RAX), Val::Imm(1))); // mangle due to type
35 instrs.push(Instr::IMov(Val::Reg(RAX), Val::DynamicRegOffset(RAX, RBX)));
36 }
37 ...<compilation of other operations>...

```

(Source: <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/src/compiler.rs>)

As we can see before actually looking up the element at the index value, we introduce the relevant out of bounds checking logic to the compiled instructions (lines 11 to 20 in the above code snippet). These instructions verify that the index being queried isn't less than 0 or greater than or equal to the number of elements in the tuple. Since our queried index (2) is greater than or equal to the number of elements in our tuple (2), the jump in line 20 of the above code snippet takes place during the execution of the program, and the relevant error message is thrown.

### 3.4 Error 3

```
1 (tuple-get (tuple 4 5) true)
```

(Source: <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/error3.snek>)

The above test validates that a runtime error is thrown during the execution of the program, since the developer was attempting to lookup/index an element of the tuple using an index argument that wasn't a number. This has to be a runtime error, since the type of the index argument will only be resolved during the execution of the program. The expected output of "invalid argument occurred with error code: 2" was received since the tuple-get operation received an index (second argument) that wasn't a number. The compiler introduces



the relevant runtime logic for type checking the index argument into the code as follows. The following code implements the compilation for `tuple-get` expressions.

```

1 Op2::TupleGet => {
2     ...<same code snippet as previous section>...
3 }
4 ...<compilation of other operations>...
5
6 fn assert_num(instrs: &mut Vec<Instr>, v: Val) {
7     instrs.push(Instr::ITest(v, Val::Imm(1)));
8     instrs.push(Instr::IMov(Val::Reg(RBX), Val::Imm(INVALID_ARGUMENT_CODE)));
9     instrs.push(Instr::IJnz(ERROR_LABEL.to_string()));
10 }

```

(Source <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/src/compiler.rs>)

As we can see before actually looking up the element, we introduce the relevant runtime type checking logic for the index argument to the compiled instructions (lines 6 to 8 in the previous code snippet and lines 6 to 9 in the above code snippet). These instructions verify that the index being queried is a number. Since our queried index (true) isn't a number, the jump in line 9 of the above code snippet takes place during the execution of the program, and the relevant error message is thrown.

### 3.5 Points

```

1 (fun (createPoint x y)
2     (tuple x y)
3 )
4
5 (fun (addPoints point1 point2)
6     (tuple (+ (tuple-get point1 0) (tuple-get point2 0)) (+ (tuple-get point1 1) (tuple-
7     get point2 1))))
8
9 (
10     block
11     (print (createPoint 1 2))
12     (print (createPoint -1 0))
13     (print (createPoint -1 -200))
14     (print (addPoints (createPoint 1 2) (createPoint 3 -1)))
15     (print (addPoints (addPoints (createPoint 1 2) (createPoint 3 -1)) (createPoint 100
16     150)))
17     (print (addPoints (addPoints (createPoint 1 2) (createPoint 3 -1)) (addPoints (
18     createPoint 2 50) (createPoint 7 3))))
19     (addPoints (addPoints (addPoints (createPoint 1 2) (createPoint 3 -1)) (addPoints (
20     createPoint 2 50) (createPoint 7 3))) (createPoint 1 2))
21 )

```

(Source <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/points.snek>)

The above program utilizes the `tuple` and `tuple-get` operations to implement two methods in the EggEater language. The first method takes two arguments `x` and `y` and constructs a point with them by representing the point as a `tuple`. The second method takes two points `point1` and `point2` (which are stored as `tuples`), adds them and returns a third point in the form of a `tuple`. Each of the `print` lines in the above test file (as well as the last `addPoints` expression in line 17) represent an individual test for a unique scenario regarding points. The output from executing this Egg Eater program was as follows:

```

1 (tuple 1 2)

```



```

2 (tuple -1 0)
3 (tuple -1 -200)
4 (tuple 4 1)
5 (tuple 104 151)
6 (tuple 13 54)
7 (tuple 14 56)

```

A quick analysis of the tests suggests that the compiler works as expected:

1. `(print (createPoint 1 2))`
  - (a) This test simply creates a point with an x-coordinate of 1 and a y-coordinate of 2, and prints it.
  - (b) The expected output of `(tuple 1 2)` was received, since points are being stored as tuples in the above program.
2. `(print (createPoint -1 0))`
  - (a) This test simply creates a point with an x-coordinate of -1 and a y-coordinate of 0, and prints it.
  - (b) The expected output of `(tuple -1 0)` was received, since points are being stored as tuples in the above program.
3. `(print (createPoint -1 -200))`
  - (a) This test simply creates a point with an x-coordinate of -1 and a y-coordinate of -200, and prints it.
  - (b) The expected output of `(tuple -1 -200)` was received, since points are being stored as tuples in the above program.
4. `(print (addPoints (createPoint 1 2) (createPoint 3 -1)))`
  - (a) This test adds the two points (1,2) and (3,-1)
  - (b) The expected output of `(tuple 4 1)` was received.
5. `((print (addPoints (addPoints (createPoint 1 2) (createPoint 3 -1)) (createPoint 100 150)))`
  - (a) This test adds the points (1,2) and (3,-1), and then adds the point (100, 150) to the resulting point from the previous addition. The final point created from the combination of these additions is printed.
  - (b) The expected output of `(tuple 104 151)` was received.
6. Test 6 - Line 16
  - (a) This test adds the points (1,2), (3,-1), (2,50) and (7,3) in a specific order, and prints the final result.
  - (b) The expected output of `(tuple 13 54)` was received.
7. Test 7 - Line 17
  - (a) This test adds the points (1,2), (3,-1), (2,50), (7,3) and (1,2) in a unique branched order, and prints the final result.
  - (b) The expected output of `(tuple 14 56)` was received.

## 3.6 BST

```
1 (fun (createNode curVal leftNode rightNode) (tuple curVal leftNode rightNode))
2
3 (fun (getValFromNode node) (tuple-get node 0))
4
5 (fun (getRightTree node) (tuple-get node 2))
6
7 (fun (getLeftTree node) (tuple-get node 1))
8
9 (fun (insertIntoBST bst elem)
10   (
11     if (= bst nil)
12       (createNode elem nil nil)
13     (
14       if (> elem (getValFromNode bst))
15         (createNode (getValFromNode bst) (getLeftTree bst) (insertIntoBST (
getRightTree bst) elem))
16       (
17         if (< elem (getValFromNode bst))
18           (createNode (getValFromNode bst) (insertIntoBST (getLeftTree bst) elem) (
getRightTree bst))
19         bst
20       )
21     )
22   )
23 )
24
25 (fun (findInBST bst elem)
26   (
27     if (= bst nil)
28       false
29     (
30       if (> elem (getValFromNode bst))
31         (findInBST (getRightTree bst) elem)
32       (
33         if (< elem (getValFromNode bst))
34           (findInBST (getLeftTree bst) elem)
35         true
36       )
37     )
38   )
39 )
40
41 (fun (createBST rootVal) (createNode rootVal nil nil))
42
43 (
44   block
45   (
46     let ((linked_list_tree (createBST 1)))
47     (
48       block
49       (set! linked_list_tree (insertIntoBST linked_list_tree 2))
50       (set! linked_list_tree (insertIntoBST linked_list_tree 3))
51       (set! linked_list_tree (insertIntoBST linked_list_tree 4))
52       (set! linked_list_tree (insertIntoBST linked_list_tree 5))
```

```

53     (set! linked_list_tree (insertIntoBST linked_list_tree 6))
54     (set! linked_list_tree (insertIntoBST linked_list_tree 7))
55     (print linked_list_tree)
56 )
57 )
58 (
59   let ((right_kink_tree (createBST 5)))
60   (
61     block
62       (set! right_kink_tree (insertIntoBST right_kink_tree 2))
63       (set! right_kink_tree (insertIntoBST right_kink_tree 3))
64       (print right_kink_tree)
65     )
66   )
67   (
68     let ((left_kink_tree (createBST -1)))
69     (
70       block
71         (set! left_kink_tree (insertIntoBST left_kink_tree 10))
72         (set! left_kink_tree (insertIntoBST left_kink_tree 5))
73         (print left_kink_tree)
74       )
75     )
76     (
77       let ((duplicate_node_tree (createBST 10)))
78       (
79         block
80           (set! duplicate_node_tree (insertIntoBST duplicate_node_tree 5))
81           (set! duplicate_node_tree (insertIntoBST duplicate_node_tree 11))
82           (set! duplicate_node_tree (insertIntoBST duplicate_node_tree 5))
83           (print duplicate_node_tree)
84         )
85       )
86       (
87         let ((root_node_tree (createBST 10)))
88         (
89           block
90             (print root_node_tree)
91           )
92         )
93         (
94           let ((regular_bst (createBST 10)))
95           (
96             block
97               (set! regular_bst (insertIntoBST regular_bst 5))
98               (set! regular_bst (insertIntoBST regular_bst 12))
99               (set! regular_bst (insertIntoBST regular_bst 4))
100              (set! regular_bst (insertIntoBST regular_bst 6))
101              (set! regular_bst (insertIntoBST regular_bst 11))
102              (set! regular_bst (insertIntoBST regular_bst 15))
103              (print regular_bst)
104              (print (findInBST regular_bst 6))
105              (print (findInBST regular_bst 12))
106              (print (findInBST regular_bst 9))
107              (set! regular_bst (insertIntoBST regular_bst 9))
108              (print regular_bst)

```

```

109         (findInBST regular_bst 9)
110     )
111 )
112 )

```

(Source <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/bst.snek>)

The above program creates a general Binary Search Tree implementation using the `tuple` data structure, and also implements a method to check if an element is in the binary search tree (the `findInBST` method). The program creates different types of binary search trees and prints the structure of the trees. The program also tests the `findInBST` method implementation.

The following output was received on execution of the above program:

```

1 (tuple 1 nil (tuple 2 nil (tuple 3 nil (tuple 4 nil (tuple 5 nil (tuple 6 nil (tuple 7 nil
   nil))))))
2 (tuple 5 (tuple 2 nil (tuple 3 nil nil)) nil)
3 (tuple -1 nil (tuple 10 (tuple 5 nil nil) nil))
4 (tuple 10 (tuple 5 nil nil) (tuple 11 nil nil))
5 (tuple 10 nil nil)
6 (tuple 10 (tuple 5 (tuple 4 nil nil) (tuple 6 nil nil)) (tuple 12 (tuple 11 nil nil) (
   tuple 15 nil nil)))
7 true
8 true
9 false
10 (tuple 10 (tuple 5 (tuple 4 nil nil) (tuple 6 nil (tuple 9 nil nil))) (tuple 12 (tuple 11
   nil nil) (tuple 15 nil nil)))
11 true

```

All of the expected results were received. Short descriptions of the tests have been provided below:

#### 1. Test 1 - LinkedList Binary Search Tree

- If the elements of a binary search tree are inserted in increasing order, then the final structure of the binary search tree mimics a linked list with no branching.
- The output in line 1 shows this exact correct behavior as the elements of the bst are only the right side children of their corresponding parents.

#### 2. Test 2 - Right Kink Binary Search Tree

- This is a common binary search tree that is tested in introductory data structures classes. The unique ordering in which the elements are introduced provides a right kink look to the tree.
- The output in line 2 shows this exact correct behavior as 2 is the left child of 5, but 3 is the right child of 2.

#### 3. Test 3 - Left Kink Binary Search Tree

- This too is a common binary search tree that is tested in introductory data structures classes. The unique ordering in which the elements are introduced provides a left kink look to the tree.
- The output in line 3 shows this exact correct behavior as 10 is the right child of -1, but 5 is the right child of 10.

#### 4. Test 4 - Duplicate Node Binary Search Tree

- The elements of a binary search tree must be unique. Thus, when a duplicate element is inserted, there should be no changes to the original bst.

- (b) The output in line 4 shows this exact correct behavior as the second insertion of the element 5 does not change the bst.
5. Test 5 - Root Node Binary Search Tree
- (a) This is a unique BST with only the root node, since no elements are inserted into it after its creation.
  - (b) The output in line 5 shows this exact correct behavior as there is only a single tuple representing the complete tree with both the left and right sub-trees being `nil`.
6. Test 6 - Regular Binary Search Tree
- (a) This test creates a regular binary search tree and queries elements from it. It first queries for 6 and 12 which are in the bst. It then queries for 9 which is not in the bst. Finally, it adds in 9 to the bst and queries for 9 again now that the element is in the bst.
  - (b) The expected output is received, since the program prints true when the `findBST` method is queried with 6 and 12, and it prints false when the `findBST` method is queried with 9. After 9 is inserted into the bst, the program prints true when the `findBST` method is queried with 9.

## 4 Comparison of Heap Allocation with other Languages

Both Java and Python are commonly used languages that support heap allocated data structures. The tuple heap allocated data structure that has been introduced to Egg Eater most resembles the tuple data structure in Python. Java doesn't have an in-built equivalent to the tuple data structure, and the two closest data structures it has are its `ArrayList` and an external `Javatuples` library's `Tuple`. The tuple data structure introduced to EggEater has the following similarities with Python's tuples:

1. **Arbitrary Length** - Tuples can be created in both Python and EggEater with an arbitrary length, only limited by the constraints of memory and integer overflow regarding the storage of the data structure's metadata. This provides for increased customizability and modularity for EggEater developers to mimic other data structures using tuples, as shown above with the BST example. Java's `ArrayList` data structure can also be created with an arbitrary length only limited by integer overflow regarding storing the datastructure's metadata. On the other hand the tuple data structure in the `Javatuple` library can only support at most 10 elements.
2. **Type Agnostic** - Just like tuples in python, the tuple data structure introduced to Egg Eater is type agnostic in that it can store elements of different types within the same tuple. This benefit arises from the commonality that both EggEater and Python utilize runtime type checking and are thus not limited based on the types of the objects. On the other hand Java's `ArrayList` can only store objects of the same type. Java's external library tuples can support different types, but this comes with the trade-off of only allowing at most 10 elements in the tuple, since the tuples for each number of elements from 1 to 10 have different implementations.

## 5 Structural Equality with Deep-Equal

We define structural equality between two values in a recursive manner as follows. If both values aren't heap allocated objects, then the two values are structurally equal by the regular definition of equality. If only one value isn't a heap allocated object, then the two values are not structurally equal. If both values are heap allocated

objects, then the two values are structurally equal through a recursive definition of structural equality requiring all elements of one tuple to be structurally equal to their corresponding elements at the same index in the other tuple. A corollary of this is that tuples with different sizes cannot be structurally equal.

The Deep-Equals operation was introduced to allow Egg-Eater developers to check for structural equality between two values. Unlike the regular `=` operation, this operation does not provide any runtime type safety, as in Python. Further, this operation requires extra runtime resources due to memory allocation and loop executions, unlike the regular `=` operation. But, this operation expands the capabilities of the language past referential equality between heap allocated objects, by actually recursing through these objects to check for structural equality. To implement such an operation we thus require 3 critical steps:

1. A successful call within the x86 instructions to an externally defined function that checks for structural equality.
2. An externally defined function that steps through and recurses through dynamically allocated tuples to check for structural equality.
3. The extra capability for such a function to avoid cyclic references in such heap allocated objects, to provide deterministic results to the end-user.

## 5.1 X86 Call to the Structural Equality function

When `deep-equal` operation is being compiled, the following X86 instructions are inserted into the final compiled instructions set to switch the control flow to an externally defined function to check for structural equality:

```

1 Op2::StructEqual => {
2     /* arg1 is in RAX, arg2 is in RSP + 8*context.si */
3     instrs.push(Instr::IMov(Val::Reg(RSI), Val::StaticRegOffset(RSP, 8*context.si)
4         ));
5     let offset = 8 * (context.si + 1 + (context.si % 2));
6     instrs.push(Instr::ISub(Val::Reg(RSP), Val::Imm(offset.into())));
7     instrs.push(Instr::IMov(Val::StaticRegOffset(RSP, 0), Val::Reg(RDI)));
8     instrs.push(Instr::IMov(Val::Reg(RDI), Val::Reg(RAX)));
9     instrs.push(Instr::ICall(STRUCT_EQ_LABEL.to_string()));
10    instrs.push(Instr::IMov(Val::Reg(RDI), Val::StaticRegOffset(RSP, 0)));
11    instrs.push(Instr::IAdd(Val::Reg(RSP), Val::Imm(offset.into())));
12 }
```

We follow the default X86 calling convention of storing the first argument to the function in *RDI* and the second argument to the function in *RSI* before the call. As mentioned in the code snippet, *RAX* holds the second argument of the `deep-equals <expr> <expr>` operation and the first argument is stored in the stack with an offset of  $8 * si$  away from *RSP*. After the call to the function, the original value of *RDI* is restored into the register and the result of the operation is stored in the *RAX* register. Since, the value stored in *RSI* is never used for any other purpose in the compiled instruction set of any Egg-Eater program, the original value doesn't get restored into *RSI* after the call to the function.

## 5.2 Checking for Structural Equality

The structural equality function discussed above (that gets called with the above X86 instructions) is handled by the following code snippet:

```

1 #[no_mangle]
2 #[export_name = "\x01snek_struct_eq"]
```

```

3 fn snek_struct_eq(val1: i64, val2: i64) -> i64 {
4     let mut seen = &mut HashSet::();
5     if is_struct_eq(val1, val2, &mut seen) {
6         return 7;
7     } else {
8         return 3;
9     }
10 }
11
12 fn is_struct_eq(val1: i64, val2: i64, seen: &mut HashSet<(i64, i64)>) -> bool {
13     if !seen.insert((val1, val2)) {
14         return true;
15     }
16
17     let is_val1_heap_addr = is_heap_addr(val1);
18     let is_val2_heap_addr = is_heap_addr(val2);
19
20     if is_val1_heap_addr && is_val2_heap_addr {
21         let addr1 = (val1 - 1) as *const i64;
22         let length1 = (unsafe { *addr1 }) >> 1;
23
24         let addr2 = (val2 - 1) as *const i64;
25         let length2 = (unsafe { *addr2 }) >> 1;
26
27         if length1 != length2 {
28             return false;
29         }
30
31         let mut i = 1;
32         while i <= length1 {
33             let elem1 = unsafe { *addr1.offset(i as isize) } as i64;
34             let elem2 = unsafe { *addr2.offset(i as isize) } as i64;
35             if !is_struct_eq(elem1, elem2, seen) {
36                 return false;
37             }
38             i = i + 1;
39         }
40         seen.remove(&(val1, val2));
41         return true;
42     } else if is_val1_heap_addr ^ is_val2_heap_addr {
43         return false;
44     } else {
45         return val1 == val2;
46     }
47 }

```

We discuss the usage and responsibilities of the *seen* vector in the section below. The remaining sections of the code snippet implement the exact logic mentioned above for the recursive definition of structural equality. The base case of our recursion involves at least one value that isn't a heap allocated object being compared against another value. If the latter value is a heap allocated object, then as seen in lines 42 and 43 of the above code snippet, the function immediately returns *false*. Else, the values are compared against each other for the regular definition of equality, without any runtime type checking based errors being thrown. For the inductive case of the above function, we iterate through the two dynamically allocated tuples to check for structural equality between each of their pairs of elements at the same index. Lines 20 to 41 implement this recursive check. We also short circuit the check and exit early without comparing the inner elements for structural equality, if the



current pair of values represents tuples of different sizes, because these values can never be structurally equal. These lines in the code snippet first grab the lengths of the two tuples stored in their corresponding metadata sections (discussed in Section 2 of this report). After conducting the short circuit check for incompatibly sized tuples, the function iterates through the two tuples and recursive checks for structural equality between the elements in the two tuples at the same index. A pair of tuples is thus only structurally equal, if all of the pairs of elements being compared are structurally equal themselves.

### 5.3 Avoiding Cyclic References

Since tuples are now mutable in the Egg-Eater language, and since we support type-agnostic tuples in this language, programs can easily update tuple elements to store cyclic references. Further, these cyclic references could also be of different length cycles, or could be represented as complex cycles when drawn as a graph. Without necessary defensive checks in place, this would lead to operations like *print* and *deep – equal* running into infinite loops when they enter into the cycle. To avoid such issues, we cache visited addresses as we recurse through complex tuple objects, and we short-circuit and return to the recursive caller when a visited address is being queried by the function again. This logic is handled in line 13 and 14 of the above code snippet, and lines 11 and 12 of the code snippet below (that handles the print function):

```

1 fn snek_str(val: i64, seen: &mut HashSet<i64>) -> String {
2     if val == 7 {
3         "true".to_string()
4     } else if val == 3 {
5         "false".to_string()
6     } else if val % 2 == 0 {
7         format!("{}", val >> 1)
8     } else if val == 1 {
9         "nil".to_string()
10    } else if val & 1 == 1 {
11        if !seen.insert(val) {
12            return "...".to_string();
13        }
14        let addr = (val - 1) as *const i64;
15        let length = (unsafe { *addr }) >> 1;
16        let mut i = 1;
17        let mut result = "(" .to_string();
18        while i <= length {
19            let elem = unsafe { *addr.offset(i as isize) };
20            result = result + &snek_str(elem, seen);
21            if i < length {
22                result = result + ", ";
23            }
24            i = i + 1;
25        }
26        seen.remove(&val);
27        result + ")"
28    } else {
29        format!("Unknown value: {}", val)
30    }
31 }
32
33 #[no_mangle]
34 #[export_name = "\x01snek_print"]
35 fn snek_print(val: i64) -> i64 {
36     let mut seen = &mut HashSet::new();

```

```

37     println!("{}", snek_str(val, &mut seen));
38     return val;
39 }

```

During a single call to the `snek_print` function or the `snek_struct_eq` function by the caller, every address that gets visited during the recursive definitions of these functions gets cached before the recursive call is made. This way if before we return back to the original call that began the recursive steps, we find the same address again, that means that there exists a cyclic reference in the tuple being printed (in the case of the structural equality function, if the same pair of addresses being queried shows up again, then a cyclic reference exists in either one of the tuples). Thus in this specific case, both the print function and the structural equality function short-circuit their execution to return back to their recursive caller step early. In the case of the print function, a (...) is printed to represent a cyclic reference, and in the case of the structural equality function we simply return *True*, since the exact same structure within the two tuples is being maintain to this point, because both tuples have a cyclic reference at the same position.

## 6 Testing for Structural Equality and Mutability

Since structural equality and tuple mutability were only recently introduced to the Egg-Eater language, the following tests were conducted on the compiler to verify the correctness of its implementation.

### 6.1 Equal

```

1 (
2   let ((x (tuple 1 2)) (y (tuple true true)))
3   (
4     block
5       (print (= x x))
6       (print (deep-equal x x))
7       (print (= (tuple 1 true false) (tuple 1 true false)))
8       (print (deep-equal (tuple 1 true false) (tuple 1 true false)))
9       (print (= (tuple 1 true false 5) (tuple 1 true false)))
10      (print (deep-equal (tuple 1 true false 5) (tuple 1 true false)))
11      (print (= x y))
12      (print (deep-equal x y))
13      (tuple-set! y 0 1)
14      (tuple-set! y 1 2)
15      (print (= x y))
16      (print (deep-equal x y))
17      (print (= (tuple 1 (tuple 5 4) false) (tuple 1 (tuple 5 4) false)))
18      (print (deep-equal (tuple 1 (tuple 5 4) false) (tuple 1 (tuple 5 4) false)))
19    )
20 )

```

(Source <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/equal.snek>)

The above program creates tuples with different types of tuples (simple, nested, different element types, etc.) and performs both referential and structural equality on the tuples.

The following output was received on execution of the above program:

```

1 true
2 true
3 false
4 true

```

```
5 false
6 false
7 false
8 false
9 false
10 true
11 false
12 true
13 true
```

A quick analysis of the tests suggests that the compiler works as expected:

1. `(print (= x x))`
  - (a) This test simply creates a tuple and binds it to the identifier *x*.
  - (b) The expected output of `true` was received, since both operands store the same address of a heap allocated object.
2. `(print (deep-equal x x))`
  - (a) This test simply creates a tuple and binds it to the identifier *x*.
  - (b) The expected output of `true` was received, since both operands are tuples storing the same values (since they are both the same tuple).
3. `(print (= (tuple 1 true false) (tuple 1 true false)))`
  - (a) This test simply creates two tuples with the same values and tests for referential equality.
  - (b) The expected output of `false` was received, since while both operands are tuples with the same values, they are stored at different addresses, and are thus not referentially equal.
4. `(print (deep-equal (tuple 1 true false) (tuple 1 true false)))`
  - (a) This test simply creates two tuples with the same values and tests for structural equality.
  - (b) The expected output of `true` was received, since both operands are tuples with the same values.
5. `(print (= (tuple 1 true false 5) (tuple 1 true false)))`
  - (a) This test simply creates two tuples with of different sizes.
  - (b) The expected output of `false` was received, since both operands are tuples stored at different addresses.
6. `(print (deep-equal (tuple 1 true false 5) (tuple 1 true false)))`
  - (a) This test simply creates two tuples with of different sizes.
  - (b) The expected output of `false` was received, since both operands are tuples with different elements.
7. `(print (= x y))`
  - (a) This test checks for referential equality between two different tuples
  - (b) The expected output of `false` was received, since both operands are tuples stored at different addresses.

8. `(print (deep-equal x y))`

- (a) This test checks for structural equality between two different tuples
- (b) The expected output of `false` was received, since both operands are tuples with different elements.

9. `(print (= x y))` (After calling `tuple-set!` on `y`)

- (a) After setting `y`'s values to be the same as `x`, this test tests for referential equality between `x` and `y`.
- (b) The expected output of `false` was received, since both operands are tuples stored at different addresses.

10. `(print (deep-equal x y))` (After calling `tuple-set!` on `y`)

- (a) After setting `y`'s values to be the same as `x`, this test tests for structural equality between `x` and `y`.
- (b) The expected output of `true` was received, since both operands are tuples with the same elements.

11. `(print (= (tuple 1 (tuple 5 4) false) (tuple 1 (tuple 5 4) false)))`

- (a) This test creates two nested tuples with the same elements and checks for referential equality.
- (b) The expected output of `false` was received, since both operands are tuples stored at different addresses.

12. `(print (deep-equal (tuple 1 (tuple 5 4) false) (tuple 1 (tuple 5 4) false)))`

- (a) This test creates two nested tuples with the same elements and checks for structural equality.
- (b) The expected output of `true` was received, since both operands are nested tuples with elements that structurally equal.

## 6.2 Cycle Print

### 6.2.1 Cycle Print 1 - Simple Cycle

```
1 (
2   let ((simple_cycle (tuple 0 1 2 3 4 5)))
3   (
4     block
5       (tuple-set! simple_cycle 5 simple_cycle)
6       simple_cycle
7   )
8 )
```

(Source <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/cycle-print1.snek>)

The above program creates a tuple with 6 elements and sets the last element of the tuple to be a reference to the original tuple. The program then prints the tuple that represents a simple cycle.

The following output was received on execution of the above program:

```
1 (0, 1, 2, 3, 4, (...))
```

This is the expected output, since the last element of the tuple is the cyclic reference, and all the other elements remain unchanged.

## 6.2.2 Cycle Print 2 - Multi Cycle

```
1 (
2   let ((x (tuple 1 2)) (multi-cycle (tuple x x x x x x x x x x)))
3   (
4     block
5     (tuple-set! multi-cycle 1 multi-cycle)
6     (tuple-set! multi-cycle 5 multi-cycle)
7     (tuple-set! multi-cycle 8 multi-cycle)
8     multi-cycle
9   )
10 )
```

(Source <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/cycle-print2.snek>)

The above program creates a tuple with 10 elements that are themselves tuples and sets multiple elements of the tuple to be references to the original tuple. The program then prints the tuple that represents a multi-cycle.

The following output was received on execution of the above program:

```
1 ((1, 2), (...), (1, 2), (1, 2), (1, 2), (...), (1, 2), (1, 2), (...), (1, 2))
```

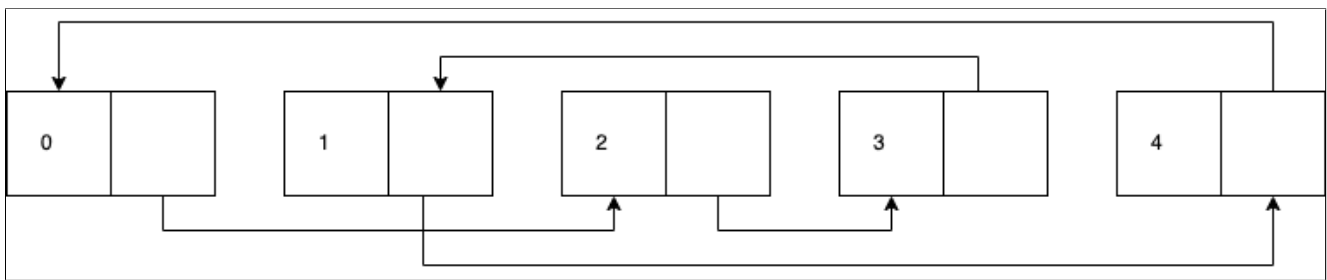
This is the expected output, since exactly the 2nd, 6th and 9th elements of the tuple are printed as cyclic references (due to 0-based indexing) and all other elements remaining unchanged. Note the correctness of the output that even though the same address is stored by the first element, the third element, the fourth element, etc., they are not considered as cyclic references.

## 6.2.3 Cycle Print 3 - Complex Cycle

```
1 (
2   let ((complex-cycle (tuple (tuple 0 0) (tuple 1 0) (tuple 2 0) (tuple 3 0) (tuple 4 0)
3   )))
4   (
5     block
6     (tuple-set! (tuple-get complex-cycle 0) 1 (tuple-get complex-cycle 2))
7     (tuple-set! (tuple-get complex-cycle 1) 1 (tuple-get complex-cycle 4))
8     (tuple-set! (tuple-get complex-cycle 2) 1 (tuple-get complex-cycle 3))
9     (tuple-set! (tuple-get complex-cycle 3) 1 (tuple-get complex-cycle 1))
10    (tuple-set! (tuple-get complex-cycle 4) 1 (tuple-get complex-cycle 0))
11    complex-cycle
12  )
13 )
```

(Source <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/cycle-print3.snek>)

The above program creates a tuple with 5 elements that are themselves tuples. Using the *tuple-set!* operation, the following cyclic references are constructed:



Finally, the complex-cycle tuple is printed. The following output was received on execution of the above program:

```

1 ((0, (2, (3, (1, (4, (...))))) , (1, (4, (0, (2, (3, (...))))) , (2, (3, (1, (4, (0, (...))))) , (3, (1, (4, (0, (2, (...))))) , (4, (0, (2, (3, (1, (...))))) )

```

This is the expected output, since each element forms a cycle exactly after 5 references to the “next” element. For instance, from the diagram above we can see that the cycle for the first element is  $0 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 0$ . Similarly, the other 4 cycles in the graph also visit all of the other elements of the graph (or tuple) once before returning back to their original starting elements.

## 6.3 Cycle Equal

### 6.3.1 Cycle Equal 1 - Self-Referential Tuples

```

1 (
2   let ((x (tuple 1 2)) (y (tuple 1 false)))
3   (
4     block
5       (tuple-set! x 1 x)
6       (tuple-set! y 1 y)
7       (deep-equal x y)
8   )
9 )

```

(Source <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/cycle-equal1.snek>)

The above program creates two tuples and sets their last elements to be cyclic references to their own respective addresses, then checks for structural equality between the two tuples.

The following output was received on execution of the above program:

```

1 true

```

This is the expected output, since both tuples are of the form  $(1, \dots)$ , where  $\dots$  is a cyclic reference back to the original tuple.

### 6.3.2 Cycle Print 2 - Multi Cycle

```

1 (
2   let ((x (tuple 1 2)) (y (tuple 1 false)))
3   (
4     block
5       (tuple-set! x 1 y)
6       (tuple-set! y 1 x)
7       (deep-equal x y)
8   )
9 )

```

(Source <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/cycle-print2.snek>)

The above program creates two tuples and sets their last elements to be a reference to the other tuple, then checks for structural equality between the two tuples.

The following output was received on execution of the above program:

```

1 true

```

This is the expected output, since both tuples are of the form  $(1, (1, \dots))$ , where  $\dots$  is a cyclic reference back to the original tuple.

### 6.3.3 Cycle Print 3 - Complex Cycle

```
1 (
2   let ((x (tuple 0 0 0)) (y (tuple 0 0 0)))
3   (
4     block
5       (tuple-set! x 2 y)
6       (tuple-set! y 2 y)
7       (tuple-set! x 1 y)
8       (tuple-set! y 1 y)
9       (deep-equal x y)
10  )
11 )
```

(Source <https://github.com/Manav-Ram19/eggeater-cse231/blob/main/input/cycle-print3.snek>)

The above program creates two tuples, sets cyclic references as follows, before checking for structural equality between the two tuples:

1. Tuple  $x$  becomes  $(0, (0, \dots, \dots), (0, \dots, \dots))$
2. Tuple  $y$  becomes  $(0, \dots, \dots)$

where  $\dots$  is a cyclic reference to the tuple which the  $\dots$  element exists in.

The following output was received on execution of the above program:

```
1 true
```

This is the expected output for the following reason:

1. First element of  $x$  is 0 and First element of  $y$  is 0.
2. Second element of  $x$  is  $(0, \dots, \dots)$  and Second element of  $y$  is  $\dots$ , which is  $(0, \dots, \dots)$ .
3. Third element of  $x$  is  $(0, \dots, \dots)$  and Third element of  $y$  is  $\dots$ , which is  $(0, \dots, \dots)$ .

## 7 Updates from EggEater

The main update to the compiler outside the addition of structural equality and tuple mutability through the *tuple-set!* operation, was the replacement of the *index* operation with the *tuple-get* operation to match the newly added *tuple-set!* operation.

## 8 References

1. Java Heap Management - <https://www.baeldung.com/java-stack-heap>
2. Python Heap Memory Management - <https://docs.python.org/3/c-api/memory.html>
3. Javatuple - <https://www.javatpoint.com/java-tuple>