CSE 231 SP23 - Egg Eater

1 EggEater's Concrete Syntax

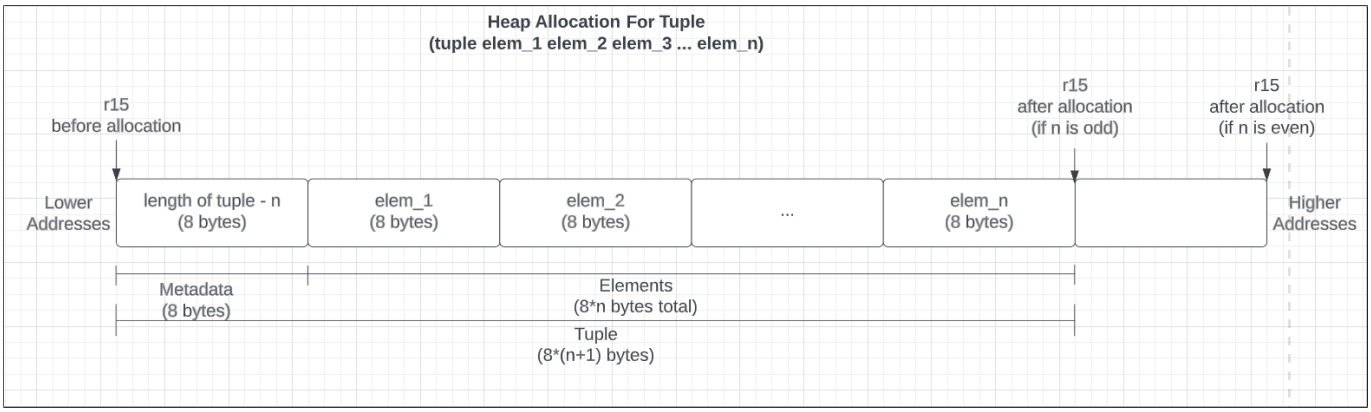
The concrete syntax followed by Egg Eater is as follows:

```
<prog> := <defn>* <expr>
<defn> := (fun (<name> <name>*) <expr>)
<expr> :=
  | <number>
  | true
  | false
  | input
  | <identifier>
  | nil                          (new!)
  | (tuple <expr>+)              (new!)
  | (let (<binding>+) <expr>)
  | (<op1> <expr>)
  | (<op2> <expr> <expr>)
  | (set! <name> <expr>)
  | (if <expr> <expr> <expr>)
  | (block <expr>+)
  | (loop <expr>)
  | (break <expr>)
  | (<name> <expr>*)

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

<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 an index operation 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. 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

```
(fun (tuplemaker single) (tuple single))
(
    block
    (print (tuple 3))
    (print (tuple -1 5))
    (print (tuple -1 true 4 false))
    (print (let ((x (tuple 100 70 true (+ 5 5)))) x))
    (print (let ((x (tuple (tuple 3)))) x))
    (print (let ((x (tuple 3 (tuple 1 (tuple 100) (tuple true false 3 4))))) x))
    (
        let (
            (x (tuple (tuple 1) (tuple (tuple 3 4) 5 (tuple 6 7))))
            (y (tuple true input 3 false))
            (z (tuple y x))
            )

        (
            block
            (print x)
            (print (index x 1))
```

```
            (print (index (index x 1) 2))
            (print (index (index (index x 1) 0) 1))
            (print (= (index (index (index x 1) 0) 0) (index y 2)))
            (print (= x (index z 1)))
            (print (= y (tuple true input 3 false)))
            (print (= y nil))
            (print (tuplemaker 10))
            (isbool (tuple 3))
        )
    )
)
```

(Source)

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:

```
(tuple 3)
(tuple -1 5)
(tuple -1 true 4 false)
(tuple 100 70 true 10)
(tuple (tuple 3))
(tuple 3 (tuple 1 (tuple 100) (tuple true false 3 4)))
(tuple (tuple 1) (tuple (tuple 3 4) 5 (tuple 6 7)))
(tuple (tuple 3 4) 5 (tuple 6 7))
(tuple 6 7)
4
true
true
false
false
(tuple 10)
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 (index x 1))`

(a) This test verifies the correctness of the `index` 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 `index` operation correctly works with complex nested tuple definitions.

9. `(print (index (index x 1) 2))`

(a) This test verifies the correctness of the `index` operation by attempting to index into the output of another index operation on a nested tuple.

(b) The expected output of `(tuple 6 7)` was received showing that the `index` operation correctly works with complex nested tuple definitions.

10. `(print (index (index (index x 1) 0) 1))`

(a) This test further verifies the correctness of the `index` operation by nesting index 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 (= (index (index (index x 1) 0) 0) (index 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 `index` 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 (index 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

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

(Source)

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 occured 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.

```
match op {
        Op2::Plus => {
            assert_both_num(instrs, Val::StaticRegOffset(RSP, 8 * context.si), Val::Reg(
   RAX));
            instrs.push(Instr::IAdd(
                Val::Reg(RAX),
                Val::StaticRegOffset(RSP, 8 * context.si),
            ));
            check_for_overflow(instrs);
        }
        ...<compilation of other operations>...

fn assert_both_num(instrs: &mut Vec<Instr>, r1: Val, r2: Val) {
    instrs.push(Instr::IMov(Val::Reg(RBX), r1));
    instrs.push(Instr::IOr(Val::Reg(RBX), r2));
    instrs.push(Instr::IAnd(Val::Reg(RBX), Val::Imm(1)));
    instrs.push(Instr::ITest(Val::Reg(RBX), Val::Imm(1)));
    instrs.push(Instr::IMov(Val::Reg(RBX), Val::Imm(INVALID_ARGUMENT_CODE)));
    instrs.push(Instr::IJnz(ERROR_LABEL.to_string()));
}
```

(Source)

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_LABE label, that handles throwing runtime errors.

3.3 Error Bounds

```
(index (tuple 3 7) 2)
```

(Source)

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 actualy 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 occured with error code: 3`" was received since the `index` 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 `index` expressions.

```
        Op2::Index => {
            /* 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             assert_num(instrs, Val::Reg(RBX));
9             // Check if it is nil and throw out of bounds
10            instrs.push(Instr::ICmp(Val::Reg(RAX), Val::Imm(1)));
11            instrs.push(Instr::IMov(Val::Reg(RBX), Val::Imm(OUT_OF_BOUNDS_ERROR_CODE)));
12            instrs.push(Instr::IJe(ERROR_LABEL.to_string()));
13            // Throw runtime error if index is less than 0
14            instrs.push(Instr::IMov(Val::Reg(RBX), Val::StaticRegOffset(RSP, 8*context.si)
    ));
15            instrs.push(Instr::ICmp(Val::Reg(RBX), Val::Imm(0)));
16            instrs.push(Instr::IMov(Val::Reg(RBX), Val::Imm(OUT_OF_BOUNDS_ERROR_CODE)));
17            instrs.push(Instr::IJl(ERROR_LABEL.to_string()));
18            // Throw runtime error if index is greater than or equal to length of tuple
19            instrs.push(Instr::IMov(Val::Reg(RBX), Val::StaticRegOffset(RSP, 8*context.si)
    ));
20            instrs.push(Instr::ICmp(Val::Reg(RBX), Val::StaticRegOffset(RAX, -1)));
21            instrs.push(Instr::IMov(Val::Reg(RBX), Val::Imm(OUT_OF_BOUNDS_ERROR_CODE)));
22            instrs.push(Instr::IJge(ERROR_LABEL.to_string()));
23            // Move element from heap at index location into rax
24            instrs.push(Instr::IMov(Val::Reg(RBX), Val::StaticRegOffset(RSP, 8*context.si)
    ));
25            instrs.push(Instr::ISar(Val::Reg(RBX), Val::Imm(1))); // Shift the index to
    the right to get the "correct index value"
26            instrs.push(Instr::IAdd(Val::Reg(RBX), Val::Imm(1))); // Add in for metadata
27            instrs.push(Instr::IMul(Val::Reg(RBX), Val::Imm(8))); // Calculate actual heap
     offset
28            instrs.push(Instr::ISub(Val::Reg(RAX), Val::Imm(1))); // mangle due to type
29            instrs.push(Instr::IMov(Val::Reg(RAX), Val::DynamicRegOffset(RAX, RBX)));
30        }
31        ...<compilation of other operations>...
```

(Source)

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 (index (tuple 4 5) true)
```

(Source)

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 occured with error code: 2`" was received since the `index` 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 `index` expressions.

```
Op2::Index => {
    ...<same code snippet as previous section>...
}
...<compilation of other operations>...

fn assert_num(instrs: &mut Vec<Instr>, v: Val) {
    instrs.push(Instr::ITest(v, Val::Imm(1)));
    instrs.push(Instr::IMov(Val::Reg(RBX), Val::Imm(INVALID_ARGUMENT_CODE)));
    instrs.push(Instr::IJnz(ERROR_LABEL.to_string()));
}
```

(Source)

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

```
(fun (createPoint x y)
    (tuple x y)
)

(fun (addPoints point1 point2)
    (tuple (+ (index point1 0) (index point2 0)) (+ (index point1 1) (index point2 1)))
)

(
    block
    (print (createPoint 1 2))
    (print (createPoint -1 0))
    (print (createPoint -1 -200))
    (print (addPoints (createPoint 1 2) (createPoint 3 -1)))
    (print (addPoints (addPoints (createPoint 1 2) (createPoint 3 -1)) (createPoint 100
    150)))
    (print (addPoints (addPoints (createPoint 1 2) (createPoint 3 -1)) (addPoints (
    createPoint 2 50) (createPoint 7 3))))
    (addPoints (addPoints (addPoints (createPoint 1 2) (createPoint 3 -1)) (addPoints (
    createPoint 2 50) (createPoint 7 3))) (createPoint 1 2))
)
```

(Source)

The above program utilizes the `tuple` and `index` oeprations 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:

```
(tuple 1 2)
(tuple -1 0)
```

```
(tuple -1 -200)
(tuple 4 1)
(tuple 104 151)
(tuple 13 54)
(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

```
(fun (createNode curVal leftNode rightNode) (tuple curVal leftNode rightNode))

(fun (getValFromNode node) (index node 0))

(fun (getRightTree node) (index node 2))

(fun (getLeftTree node) (index node 1))

(fun (insertIntoBST bst elem)
    (
        if (= bst nil)
        (createNode elem nil nil)
        (
            if (> elem (getValFromNode bst))
            (createNode (getValFromNode bst) (getLeftTree bst) (insertIntoBST (getRightTree bst) elem))
            (
                if (< elem (getValFromNode bst))
                (createNode (getValFromNode bst) (insertIntoBST (getLeftTree bst) elem) (getRightTree bst))
                bst
            )
        )
    )
)

(fun (findInBST bst elem)
    (
        if (= bst nil)
        false
        (
            if (> elem (getValFromNode bst))
            (findInBST (getRightTree bst) elem)
            (
                if (< elem (getValFromNode bst))
                (findInBST (getLeftTree bst) elem)
                true
            )
        )
    )
)

(fun (createBST rootVal) (createNode rootVal nil nil))

(
    block
    (
        let ((linked_list_tree (createBST 1)))
        (
            block
            (set! linked_list_tree (insertIntoBST linked_list_tree 2))
            (set! linked_list_tree (insertIntoBST linked_list_tree 3))
            (set! linked_list_tree (insertIntoBST linked_list_tree 4))
            (set! linked_list_tree (insertIntoBST linked_list_tree 5))
```

```
            (set! linked_list_tree (insertIntoBST linked_list_tree 6))
            (set! linked_list_tree (insertIntoBST linked_list_tree 7))
            (print linked_list_tree)
        )
    )
    (
        let ((right_kink_tree (createBST 5)))
        (
            block
            (set! right_kink_tree (insertIntoBST right_kink_tree 2))
            (set! right_kink_tree (insertIntoBST right_kink_tree 3))
            (print right_kink_tree)
        )
    )
    (
        let ((left_kink_tree (createBST -1)))
        (
            block
            (set! left_kink_tree (insertIntoBST left_kink_tree 10))
            (set! left_kink_tree (insertIntoBST left_kink_tree 5))
            (print left_kink_tree)
        )
    )
    (
        let ((duplicate_node_tree (createBST 10)))
        (
            block
            (set! duplicate_node_tree (insertIntoBST duplicate_node_tree 5))
            (set! duplicate_node_tree (insertIntoBST duplicate_node_tree 11))
            (set! duplicate_node_tree (insertIntoBST duplicate_node_tree 5))
            (print duplicate_node_tree)
        )
    )
    (
        let ((root_node_tree (createBST 10)))
        (
            block
            (print root_node_tree)
        )
    )
    (
        let ((regular_bst (createBST 10)))
        (
            block
            (set! regular_bst (insertIntoBST regular_bst 5))
            (set! regular_bst (insertIntoBST regular_bst 12))
            (set! regular_bst (insertIntoBST regular_bst 4))
            (set! regular_bst (insertIntoBST regular_bst 6))
            (set! regular_bst (insertIntoBST regular_bst 11))
            (set! regular_bst (insertIntoBST regular_bst 15))
            (print regular_bst)
            (print (findInBST regular_bst 6))
            (print (findInBST regular_bst 12))
            (print (findInBST regular_bst 9))
            (set! regular_bst (insertIntoBST regular_bst 9))
            (print regular_bst)
```

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

(Source)

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
   (a) 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.
   (b) 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
   (a) 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.
   (b) 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
   (a) 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.
   (b) 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
   (a) 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 regukar 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. **Immutability** - Immutability of tuples in both Python and EggEater keep programs secure from circular references by preventing the updating of the elements within the tuple after its creation. The ArrayList in Java is mutable, but the Tuple in the Javatuples library follows the same principle of being immmutable.

2. **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.

3. **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 References

1. Java Heap Allocation
2. Python Heap Memory Management
3. Javatuple