# The Egg-Eater Language

## Concrete syntax

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

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

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

In enum Reg,  R15, RCX are added

## Abstract Syntax

enum Op1 { Add1, Sub1, IsNum,  IsBool, Print, IsNil, }

enum Op2 { Plus, Minus, Times, Equal,  Greater,  GreaterEqual,  Less,  LessEqual, CheckTypeMatch,}

```
enum Expr {
    Number(i64),
    True,
    False,
    Input,
    Nil,
    Id(String),
    Let(Vec<(String, Expr)>, Box<Expr>),
    UnOp(Op1, Box<Expr>),
    BinOp(Op2, Box<Expr>, Box<Expr>),
    Set(String, Box<Expr>),
    If(Box<Expr>, Box<Expr>, Box<Expr>),
    Block(Vec<Expr>),
    Loop(Box<Expr>),
    Break(Box<Expr>),

    Function(String, Vec<Expr>),
    Array(Vec<Expr>),
    GetArrayIndex(Box<Expr>, Box<Expr>),
    SetArrayIndex(Box<Expr>, Box<Expr>, Box<Expr>),
    GetArraySize(Box<Expr>),
}
```

## Semantics

An Egg-eater program always evaluates to a single integer, a single boolean, an array of elements (can be nested and elements can be of different types like boolean and numbers together) or ends with an error. When ending with an error, it prints a relevant message to the standard error.

An Egg-eater starts by evaluating the <expr> at the end of the <prog>. In addition to the diamondback semantics, the expressions have the following semantics:

- (array (<expr>*)) is an initialization of an array which is a heap-allocated data structure. It is 1-indexed. It first calculates the length of the (<expr>*), i.e. the number of (<expr>) as part of the array. This is the metadata that is stored along with the values of the <expr> in the heap. Next, it calculates the current address of R15 to find the heap location in which the elements of the array will be added to. It stores this address in a temporary register to incrementally add the elements of the array after they are processed. It then increments R15 to point to the next available memory after space for the metadata and current array elements is reserved. Then it starts evaluating the expressions to their values and incrementally writes the values to the heap by offsetting to the index offset of the element. Here, index is the location in which the element occurs within the array.

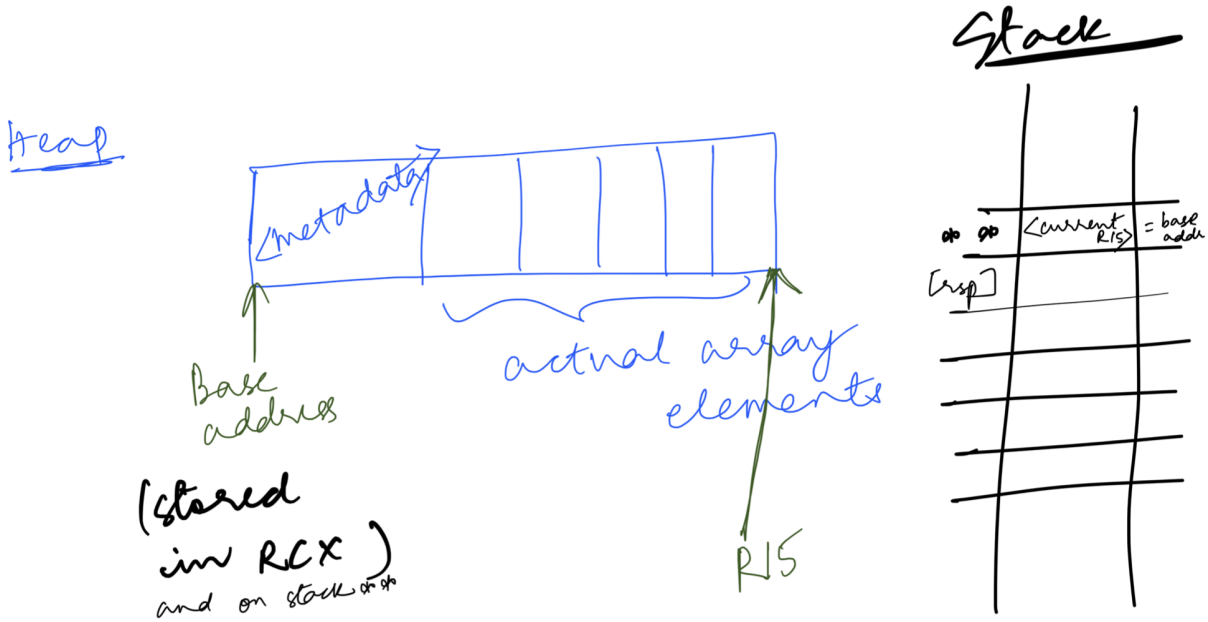
- ‘nil’ is used to represent ‘nil’ in arrays as discussed in class. It evaluates to itself. It is defined separately so that functions requiring isnil check can be handled. It is added to the enum Expr.
- (getIndex <expr> <expr>) is an expression for lookup that allows computed indexed access. The first expr evaluates to an address of heap-allocated memory and the second expr evaluates to a number which is the *index* of the heap-allocated memory(array) that is to be accessed and returned. The output of this will be the value stored in the heap-allocated memory (array) at the *index* position. It represents (index <expr> <expr>) from the assignment writeup. The index is 1-indexed position in the array.
- (setIndex <expr> <expr> <expr>) is an expression for modifying the value of a heap-allocated memory. The first expr evaluates to an address of heap-allocated memory and the second expr evaluates to a number which is the index of the heap-allocated memory(array) which should be accessed and modified. The third expr evaluates to the new value which needs to be put in the index position. The new value can be anything from the known grammar like a number, boolean, array, nested array, nil. The output of this will be the first expr, which is the starting address of the heap-allocated memory, which when passed to print will print the entire array with the starting address as the first expr.
- (getSize <expr>) -> An expression to get the size of the heap-allocated memory assigned to a single variable denoted by the address <expr>. It gives the size for the <expr> which is the starting memory address for the array. In heap-memory, the first value for any variable is the size of the array, i.e. the number of elements that are grouped under a single array. So ‘getSize’ will represent the value stored at the address specified by <expr>
- (isnil <expr>) is an expression to check if the value that the <expr> evaluates to is nil or no. It evaluates <expr> and compares its value to the value of nil and returns true if it matches and false otherwise.
- (checkTypeMatch <expr1> <expr2>) is an expression used to check if the types of <expr1> and <expr2> match. It evaluates <expr1> to its value and <expr2> to its value and then compares the last two tag bits. It returns true if the tag bits match and false if they don’t.

<u>The compiler should stop and report an error if:</u>

- Any of the <expr> in array evaluates to a value other than a number, boolean, array (maybe nested array), or nil. Incorrect addresses (like no tag bits) should also report error.
- Any of the <expr> in array evaluates to any of the reserved keywords including ‘input’.
- getIndex or setIndex tries to access an index greater than the size of the array. It should report “index_out_of_bounds_error”
- (getIndex <addr_expr> <index_expr>) - If addr_expr does not evaluate to a valid address with the array tag bits (0x01) (number, boolean, should report error). If it evaluates to nil, it should report error. (Ex - (getIndex nil 2) should throw invalid error) or index_expr does not evaluate to a number (boolean, array should report error)

- (getSize <addr_expr>) - If addr_expr does not evaluate to a valid address with the array tag bits (0x01) (number, boolean, should report error)
- Any of the specified grammar syntaxes are not followed. For example - getSize receiving a number instead of an array (getSize 5), checkTypeMatch receives a single expr instead of 2 exprs(checkTypeMatch true).
- If there are multiple errors, the compiler can report any non-empty subset of them.

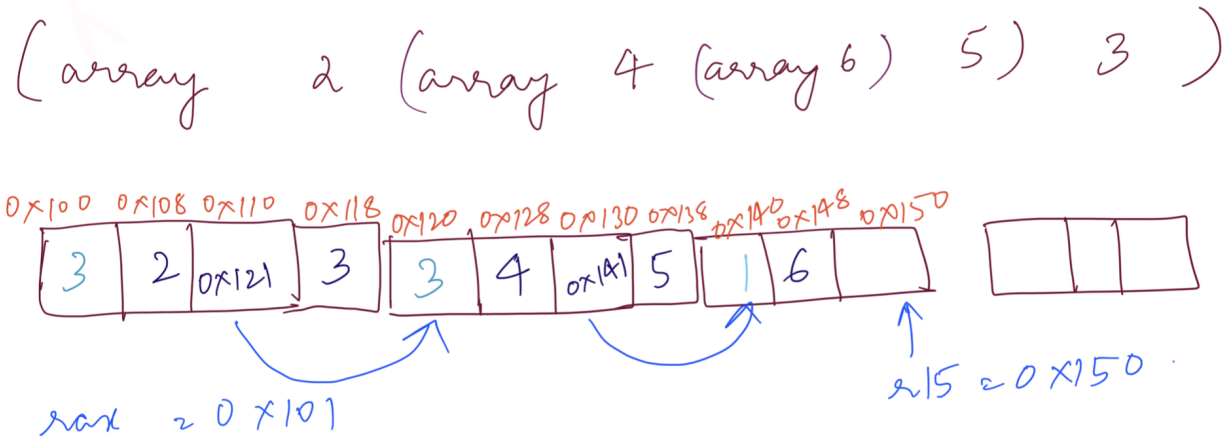
# A diagram of how heap-allocated values are arranged on the heap, including any extra words like the size of an allocated value or other metadata.




Description

Array is stored in the heap. For each array, the heap also stores metadata, the size of the array, as the first element at the base address. 1 word of memory is used for storing metadata along with each array element being stored as 1 word each. Memorywise, metadata consumes 8 bytes of memory along with each array element occupying 1 word = 8 bytes each. The base address along with the array tag is the representative of the array. Followed by the metadata, the elements of the array are added to the heap in the order in which they occur. For any nested arrays, the address of the nested array along with the array tag is inserted in the index in which it occurs in the outer array. Internally, as the array is received by the compiler, the size can be calculated and slots are reserved for the array elements. R15's current address is stored on the stack and it is incremented by the number of elements + 1 (for the size of the array). Thus, R15 points to the next available address, and any of the nested elements in the array will get the base address starting at R15 (next available address).

For example,

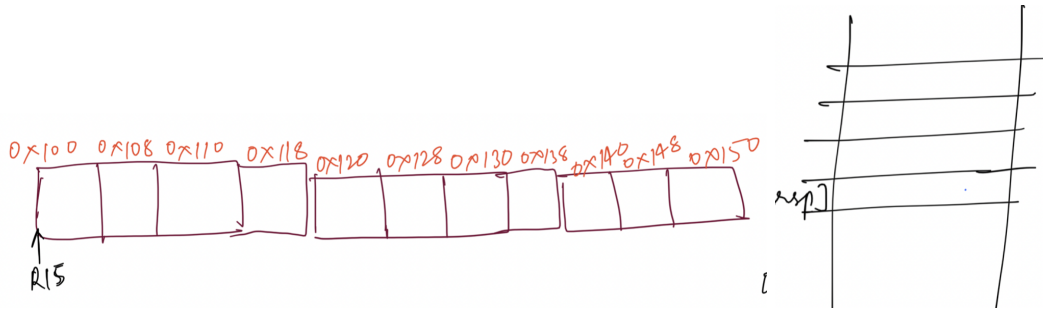



Here, the outer array has 3 elements, 2, a nested array with 3 elements (4, array(6), 5) and 3. So initially R15 points to 0x100. This becomes the base address for the outer array and the address along with the array tag would be returned in RAX. The first value at the base address would be the metadata which is the size of the array. Thus, 0x100 stores the size which is 3 in this case. Then, the following 3 offsets from the base address would store the respective array elements namely, 2, the address for the nested array and 3. R15 is incrememeted by 3 and then each of the array elements are compiled. 2 is compiled and returned in RAX. It is then stored in the heap memory with offset = index which is 1 here. Then, the inner array is compiled and it is allocated a base address which is the current value of R15. As slots for the outer array were already reserved, and R15 now points to the next available memory, thus the inner array is allocated at 0x120 as shown. Again the metadata, size of elements = 3, is saved at 0x120. R15 is offset by 3 + 1 = 4. Then each of the compiled values of the nested array is compiled and the inner sub-nested array is compiled and its address is allocated at index 2. Finally, 5 is also written to heap at index 0x138. After that, the base address of the nested array 0x120 is added with the array tag and it is returned to the outer array, which stores 0x121 at the 3rd position (2 is the actual index of the array + 1 because of the size metadata). Then, 3 is stored. And finally the base address of the outer array 0x100 + 1 = 0x101 is returned.
The array elements are evaluated and written directly to the heap memory, they are not written to the stack as any temporary storage. This is possible because R15 is incremented by reserving slots (1+num of elements) for the current array being processed.

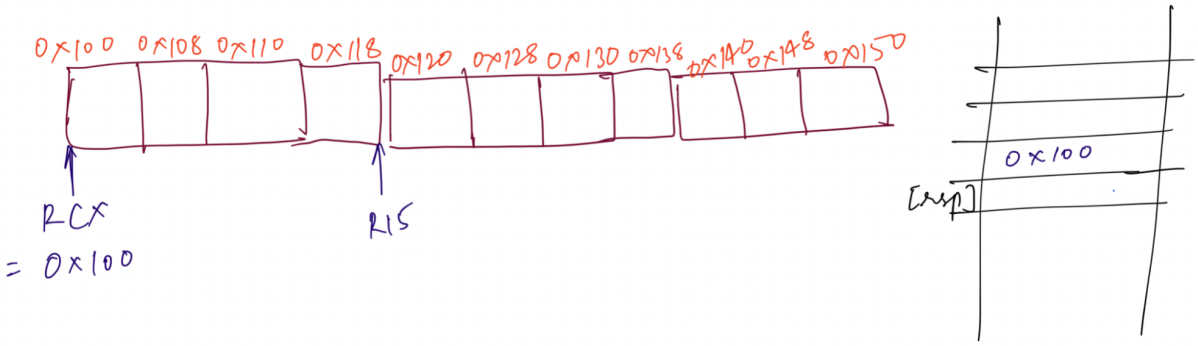
<u>Detailed explanation</u>
If array:

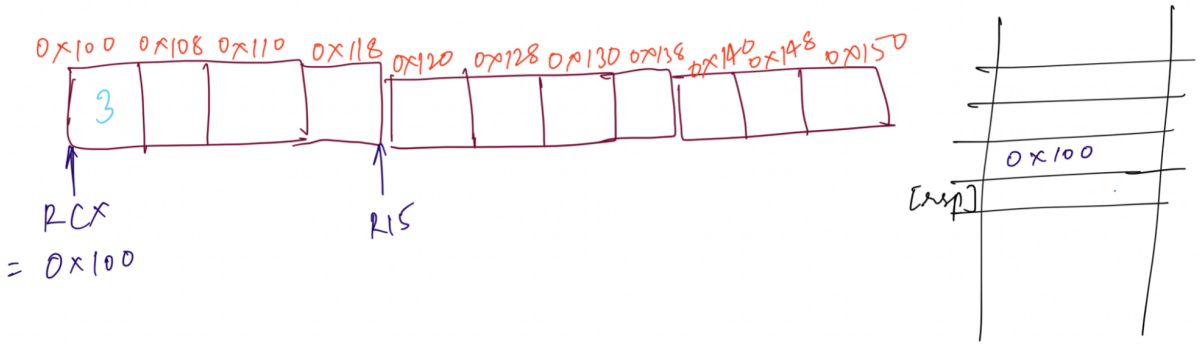
1. Calculate array size
2. Store R15 on stack as base location address
3. Increment R15 with (1 + num of elements offset)
4. Store size in base location address stored on stack
5. For each index element of array:
    a. Call compile_expr with si = si + 2 + index of element - 1 (index is 1-indexed)
        i. +2 is because of alignment requirements as 1 stack word will be used for storing the base location (address)

b. Access base location from stack into RCX
c. Offset by index position and write to heap



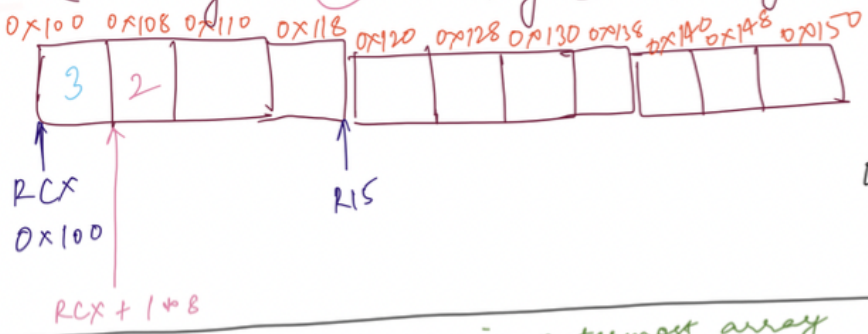

(array 2 array(4 (array 6) 5) 3)

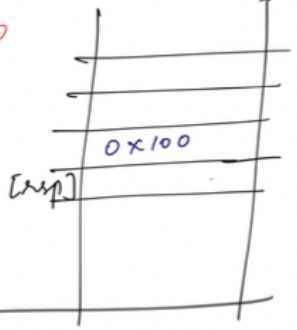


① evaluating 1st element of outermost array = 2

(array 2 array L4 (array 6) 5) 3)

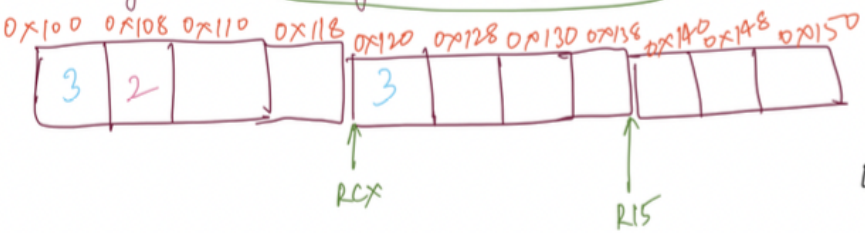


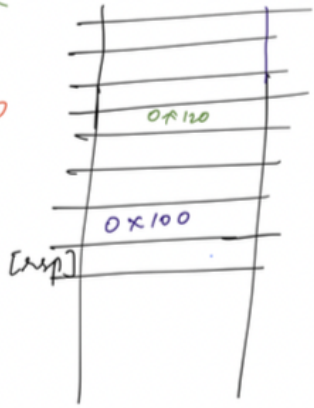




② evaluating array within outermost array
2nd index position =

= array (4 array (6) 5)

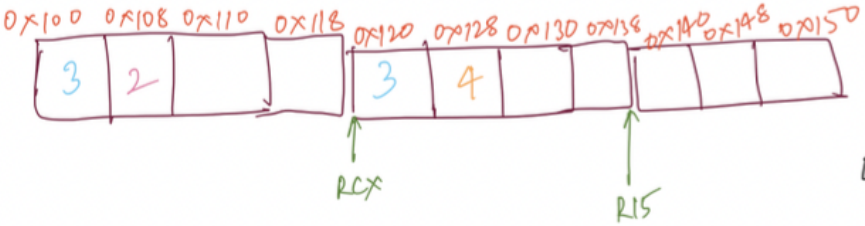
(array 2 array L4 array(6) 5) 3)

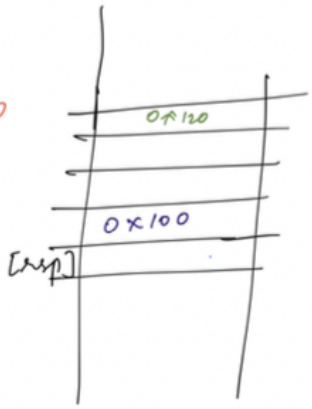



③ Evaluate 4

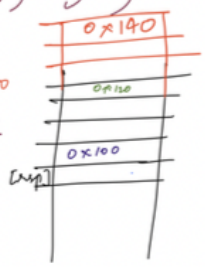
(array 2 array L4 (array 6) 5) 3)




④ Evaluate (array 6)

(array 2 array L4 (array 6) 5) 3)

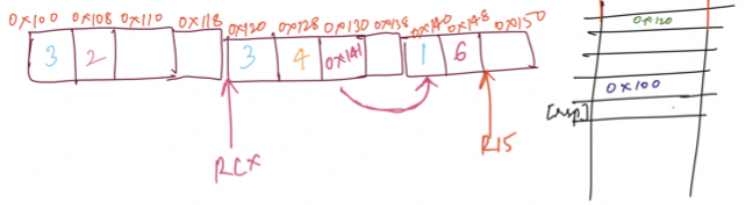



⑤ Evaluate 6

(array 2 array L4 (array 6) 5) 3)

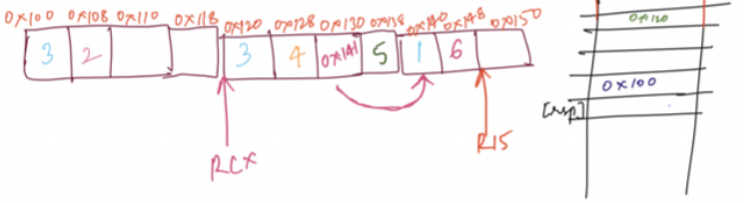

⑥ Return from processing (array 6)

(array 2 array L4 (array 6) 5) 3)



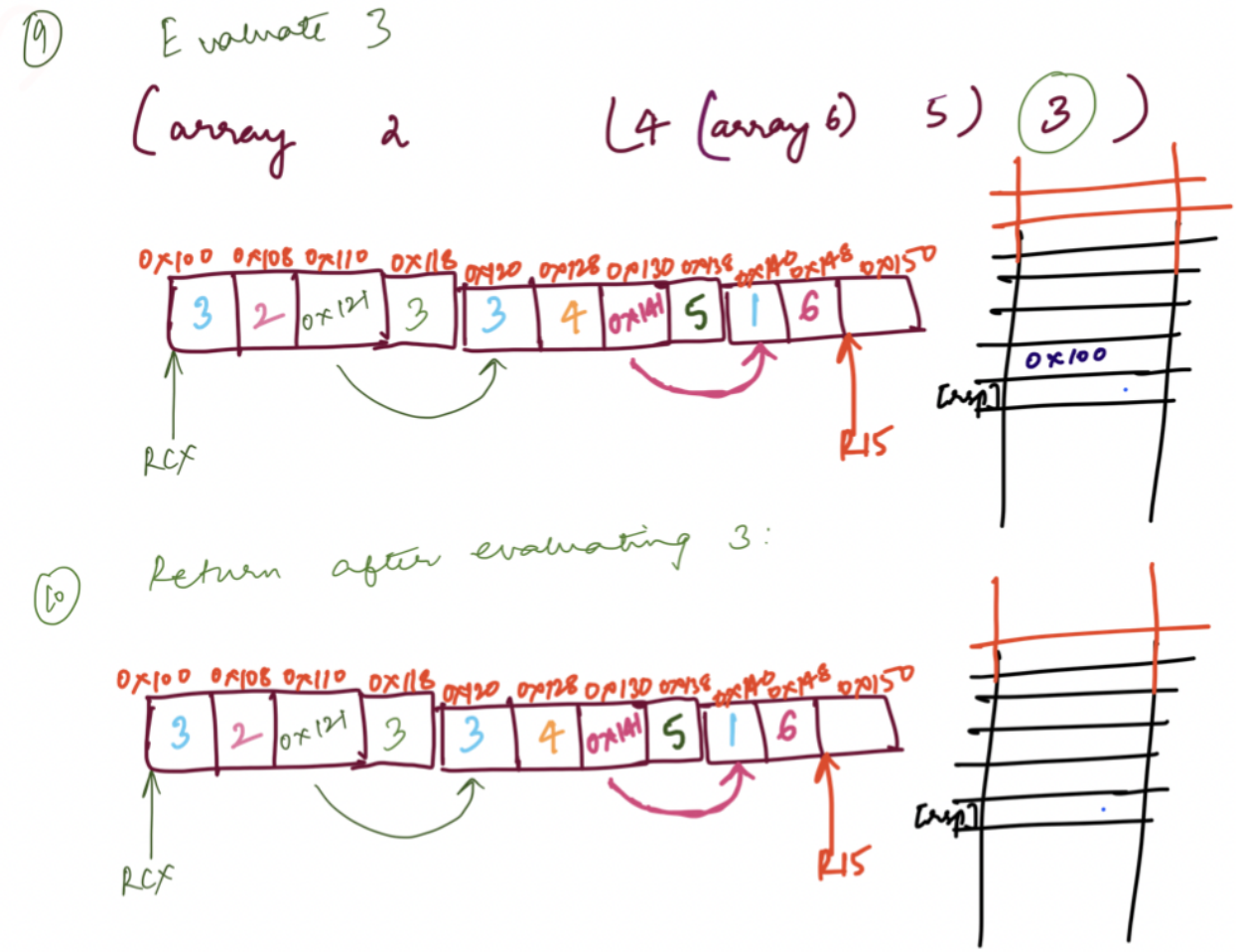

⑦ Evaluate 5

(array 2 array L4 (array 6) (5)) 3)




⑧ Return after processing 5

(array 2 array L4 (array 6) 5) 3)




Memory representations

True is represented as 7.

False is represented as 3.

Nil is represented as 1.

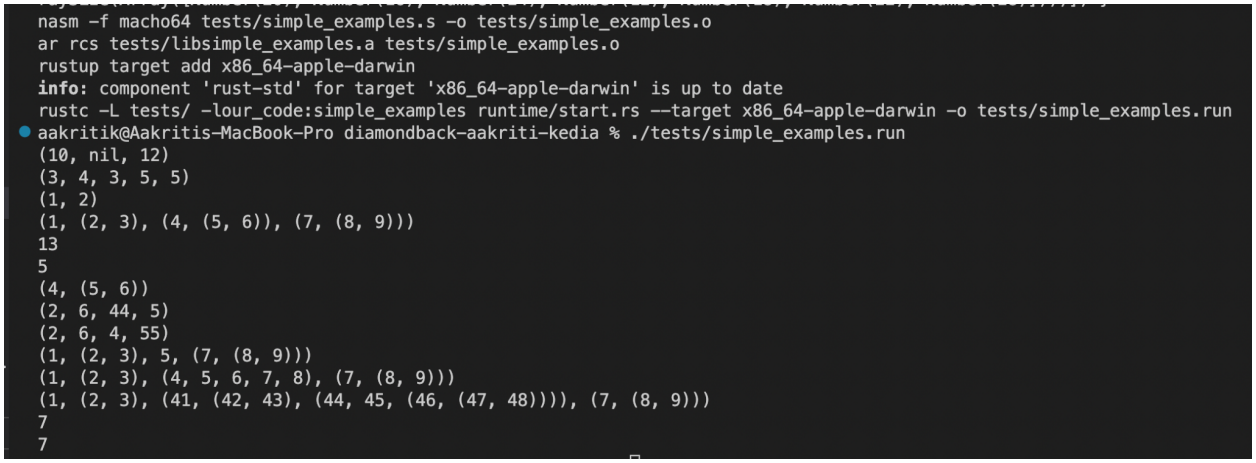
Arrays are represented with tag bits of 01 added to the starting address of heap-memory in which they are stored. Example - 0x1001 represents memory address 0x1000

# Example tests

In all the following examples, wherever print statements are used, the output of the last command is replicated twice because the first output is due to the print statement whereas the other is due to the final value being returned in RAX which is further printed as the end of the program.

## Simple_examples.snek

```
(block
   (print (array 10 nil 12))
   (print (array 3 4 3 5 5))
   (print (array 1 2))
   (print (array 1 (array 2 3) (array 4 (array 5 6)) (array 7 (array 8 9))))

   (print (getIndex (array 2 6 13 4 5) 3))
   (print (getIndex (array 2 6 4 5) 4))
   (print (getIndex (array 1 (array 2 3) (array 4 (array 5 6)) (array 7 (array 8 9))) 3))

   (print (setIndex (array 2 6 4 5) 3 44))
   (print (setIndex (array 2 6 4 5) 4 55))
   (print (setIndex (array 1 (array 2 3) (array 4 (array 5 6)) (array 7 (array 8 9))) 3 5))

   (print (setIndex (array 1 (array 2 3) (array 4 (array 5 6)) (array 7 (array 8 9))) 3
            (array 4 5 6 7 8)))

   (print (setIndex (array 1 (array 2 3) (array 4 (array 5 6)) (array 7 (array 8 9))) 3
            (array 41 (array 42 43) (array 44 45 (array 46 (array 47 48))))))

   (print (getSize (array 10 8 12 6 9 11 14)))
)
```



Interesting comments

Even nested arrays are initialized correctly, and setIndex is setting the new value at the specified index position correctly without affecting the other elements of the array. It is because the nested arrays are stored on the heap in the next available address (R15) without interfering with the already allocated addresses.

Description

The print statements with **(array <expr>*)** indicate the creation of array or the allocation of heap-allocated memory. It will return the starting address of the allocated memory along with the tag bit of array added to it and then the snek_print function will identify it as an array, remove the tag bit, get the actual address of memory, offset 1 (8 bytes) (because the 1st index stores the number of elements contributing to this array) and start printing the number of elements which were stored in the 1st index. Internally, when a new array is initialized, the base address becomes the current value of R15. The number of elements + 1 slots are reserved for the array and R15 is incremented by the (number of elements + 1) * 8 bytes. +1 is for storing metadata, which is the size of the array.

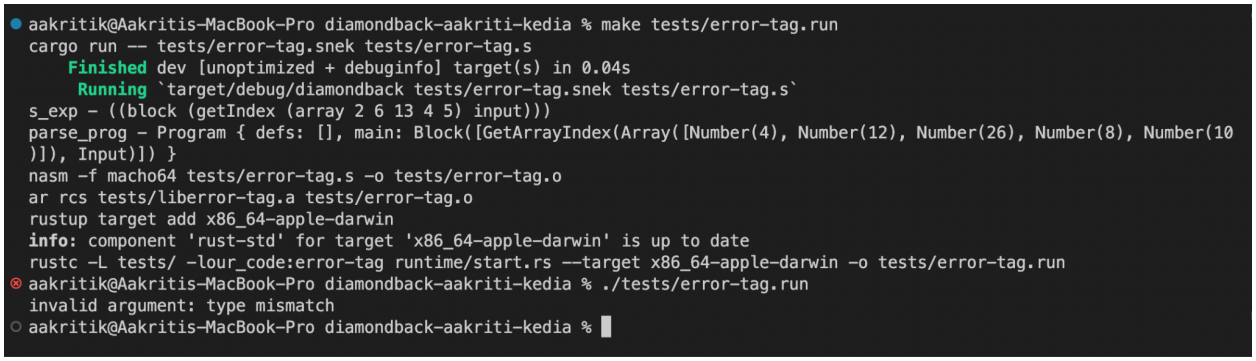
The print statements with **(getIndex <expr1> <expr2>)** are showing the lookup functionality. <expr1> evaluates to the starting address returned by (array <expr>*). Internally the compiler removes the tag bit, and accesses the actual memory location. <expr2> is the index of the array which the user wants to access. The compiler compares the value at the address with the index and if the index is greater than that value (number of elements in the array), it throws an out of bounds error, else it offsets (1 (the first index stores the size of the array) + index - 1 (as we need to return the start address location for this element) and returns the value stored at that heap location.

The print statements with **(setIndex <expr1> <expr2> <expr3>)** are showing the modification functionality. <expr1> evaluates to the starting address returned by (array <expr>*). Internally the compiler removes the tag bit, and access the actual memory location. <expr2> is the index of the array which the user wants to modify. The compiler compares the value at the address with the index and if the index is greater than that value (number of elements in the array), it throws an out of bounds error, else it offsets (1 (the first index stores the size of the array) + index - 1 (as we need to access the start address location for this element)). <expr3> is the value that the user wants to put in the index location. The compiler updates the value in the heap-allocated memory at the required address. Internally, when an array is initialized, the number of elements slots are reserved in heap-allocated memory. This is done by storing the current value of R15 in stack and incrementing R15 by the (1 + number of elements) * 8. Everytime the element of the array (expressed as <expr>) is evaluated, it is written to its index position [((index *(1-indexed)* - 1) *position on heap* + 1 *for metadata*) = (index)] location from the starting address of the array.

**(getSize <expr>)** -> is used for returning the size of the array. It is a helper functionality which will be used by complex functionality like bst to check if the array is empty or not.

# Error-tag.snek

```
(block
   (getIndex (array 2 6 13 4 5) input)
)
```



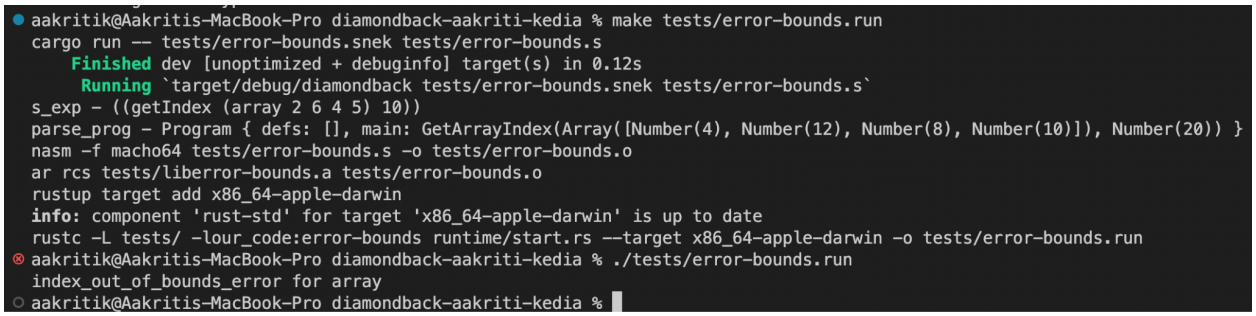

Description

The following error occurs because the program expects the index-expr to be a number. The program has specified it as input. The user was supposed to enter a number but as the user did not enter anything, the default value of input (which is false) was considered and hence the type mismatch occurred, leading to a runtime error. The is_num tag check for index_expr is included in the compile phase of the compiler. As the check fails, a runtime error is thrown reporting type mismatch.

(getIndex nil 2) will also have the same error.

# Error-bounds.snek

```
(getIndex (array 2 6 4 5) 10)
```




Description

getIndex internally compares the size of the array stored as the first element in the starting address of the array in heap-allocated memory with the index expression. If the value of the index expression is greater than the size of the array, it throws the error out of bounds. The check is included in the compile phase of the compiler. In this example, as the size of the array is 4 and the user is accessing index 10, it throws a runtime error out of bounds.

# Error3.snek

(array input 5)

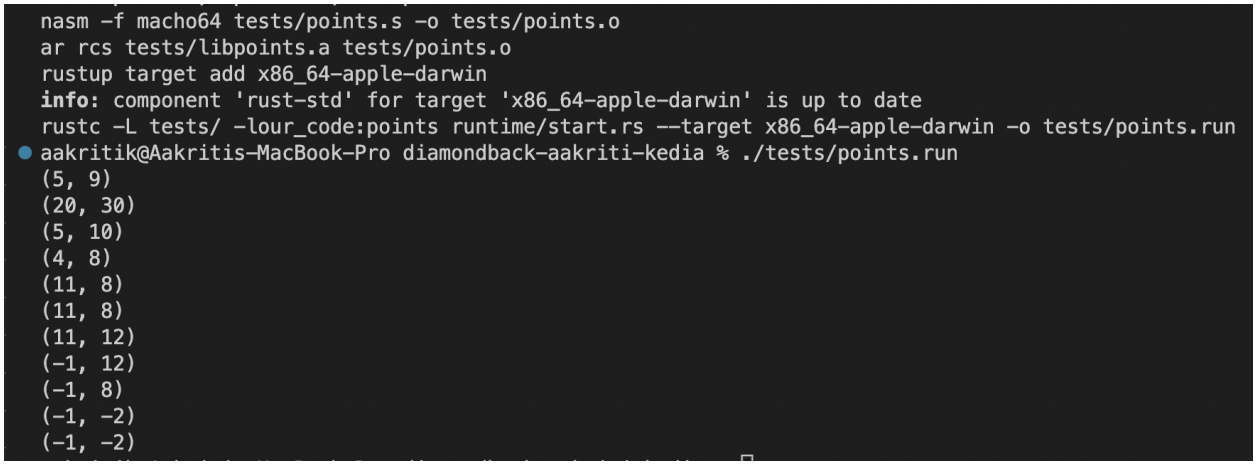
```
aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % make tests/error3.run
cargo run -- tests/error3.snek tests/error3.s
    Finished dev [unoptimized + debuginfo] target(s) in 0.08s
     Running `target/debug/diamondback tests/error3.snek tests/error3.s`
s_exp - ((array input 5))
thread 'main' panicked at 'array binding contains a keyword', src/main.rs:1167:25
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
make: *** [tests/error3.s] Error 101
aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia %
```

Description

The array is not allowed to have reserved keywords as its elements. So when the user uses ‘input’ as an element of the array, the compiler detects while parsing that it is a reserved keyword, and throws a static error saying 'array binding contains a keyword'.

# Points.snek

```
(fun (points x y) (array x y))

(fun (pointsAddition p1 p2)
      (array (+ (getIndex p1 1) (getIndex p2 1)) (+ (getIndex p1 2) (getIndex p2 2))))

(block
   (print (pointsAddition (points 2 4) (points 3 5)))
   (print (pointsAddition (points 5 10) (points 15 20)))
   (print (pointsAddition (points 0 0) (points 5 10)))
   (print (pointsAddition (points -1 -2) (points 5 10)))
   (print (pointsAddition (points (* 2 3) -2) (points 5 10)))
   (print (pointsAddition (points (* 2 3) (- 2 4)) (points 5 10)))
   (print (pointsAddition (points (* 2 3) (- 4 2)) (points 5 10)))
   (print (pointsAddition (points (* -2 3) (- 4 2)) (points 5 10)))
   (print (pointsAddition (points (* -2 3) (- 2 4)) (points 5 10)))
   (print (pointsAddition (points (* -2 3) (- 2 4)) (points 5 0)))

)
```




Interesting comments

The code is working properly even for negative inputs.

Description

'points' function takes in 2 arguments x and y and creates an array with x as the first element and y as the second element. The 'pointsAddition' adds the same index elements together, resulting in addition of 'x' coordinates and 'y' coordinates respectively. Finally, an array is created with the sum of 'x' coordinates as the first element and the sum of 'y' coordinates as the second element and it is returned. Several tests with different variations of binops are used, and different variations as mentioned in the example program with +ve, -ve values of x and y coordinates are tested.

# Bst.snek

**Insert functions**

```
(fun (insert_bst_recur bst_base_addr elm)
   (
      block
      (
         if (isnil bst_base_addr)
            (array elm nil nil)
            (
               if (= (getIndex bst_base_addr 1) elm)
                  bst_base_addr
                  (
                     if (> (getIndex bst_base_addr 1) elm)
                        (array (getIndex bst_base_addr 1) (insert_bst_recur (getIndex
bst_base_addr 2) elm) (getIndex bst_base_addr 3))
                        (array (getIndex bst_base_addr 1) (getIndex bst_base_addr 2)
(insert_bst_recur (getIndex bst_base_addr 3) elm))
                  )
            )
      )
   )
)

(fun (insert_bst_elm bst_base_addr elm)
   (
      print (insert_bst_recur bst_base_addr elm)
   )
)
```

Description

The insert function takes as input the starting address of the array (heap-allocated memory) in which the element is to be inserted. The address is a value along with the array-tag. This is the output of the array syntax as defined above. Then, the function checks if the address is nil, if yes, it returns a new array with the element to be inserted, and its left and right child as nil each. If not, the function checks if the value at the root equals the value to be inserted. If yes, it does not do anything and just returns the starting address of this sub_array which contains the required element. Else, it inserts that element in the correct half of the tree.
If the element to be inserted is less than the root element, it inserts it in the left half of the tree and returns a new array that has the root element, the new left array formed by inserting this element, and the right array which was already a part of the original array before the element was inserted.

If the element to be inserted is greater than the root element, it inserts it in the right half of the tree and returns a new array that has the root element, the left array which was already a part of the original array before the element was inserted and the new right array formed by inserting this element.

**Search functions**

```
(fun (search_bst_recur bst_base_addr elm)
   (
      block
      (print bst_base_addr)
      (
         if (isnil bst_base_addr)
         false
         (
            block
            (if (= (getIndex bst_base_addr 1) elm)
            true
            (
               if (> (getIndex bst_base_addr 1) elm)
               (search_bst_recur (getIndex bst_base_addr 2)  elm)
               (search_bst_recur (getIndex bst_base_addr 3)  elm)
            ))
         )
      )
   )
)

(fun (search_bst bst_base_addr elm)
   (
      block
         (print bst_base_addr)
         (print elm)
         (search_bst_recur bst_base_addr elm)
   )
)
```

Description

The search function takes as input the starting address of the array (heap-allocated memory) in which the element is to be searched. The address is a value along with the array-tag. This is the output of the array syntax as defined above. Then, the function checks if the address is nil. If it is, it means the element is not found in the array, and the function returns false. If the address is not nil, the function compares the value at the root with the element being searched. If they are equal, it means the element is found, and the function returns true. If the element is less than

the root value, the function continues the search in the left half of the tree by recursively calling the search function with the left array as the input. If the element is greater than the root value, the function continues the search in the right half of the tree by recursively calling the search function with the right array as the input. The search process continues until the element is found or the appropriate nil value is encountered, indicating the element is not present in the tree.

Interesting feature

As insert is creating new array everytime, it will be allocated new addresses, which will the next available memory addresses. As the bst is represented like a tree structure, it is always guaranteed to have array of size 3 for each node. The root element is present in index position 1 and the left child of the node is at index position 2 and the right child is in index position 3.

**Example test 1 - search found**

```
(search_bst
   (insert_bst_elm
      (insert_bst_elm
         (insert_bst_elm
            (insert_bst_elm
               (insert_bst_elm
                  (insert_bst_elm
                     (insert_bst_elm nil 10)
                  8)
               12)
            6)
         9)
      11)
   15)
9)
```

```
(10, nil, nil)
(10, (8, nil, nil), nil)
(10, (8, nil, nil), (12, nil, nil))
(10, (8, (6, nil, nil), nil), (12, nil, nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, nil, nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
9
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
(8, (6, nil, nil), (9, nil, nil))
(9, nil, nil)
true
```

The output is incremental, i.e. it prints the entire array when an insert is called. Thus, the output shows the status of the new array created after every insert operation. The search operation prints the array it is taking as input and the element. Then it prints the relevant array for the element to be searched, i.e it shows the new array being considered for searching (either left or right of the original array) and finally prints if the element was found or no. If found, outputs true else false.

**Example test 2 - search not found**

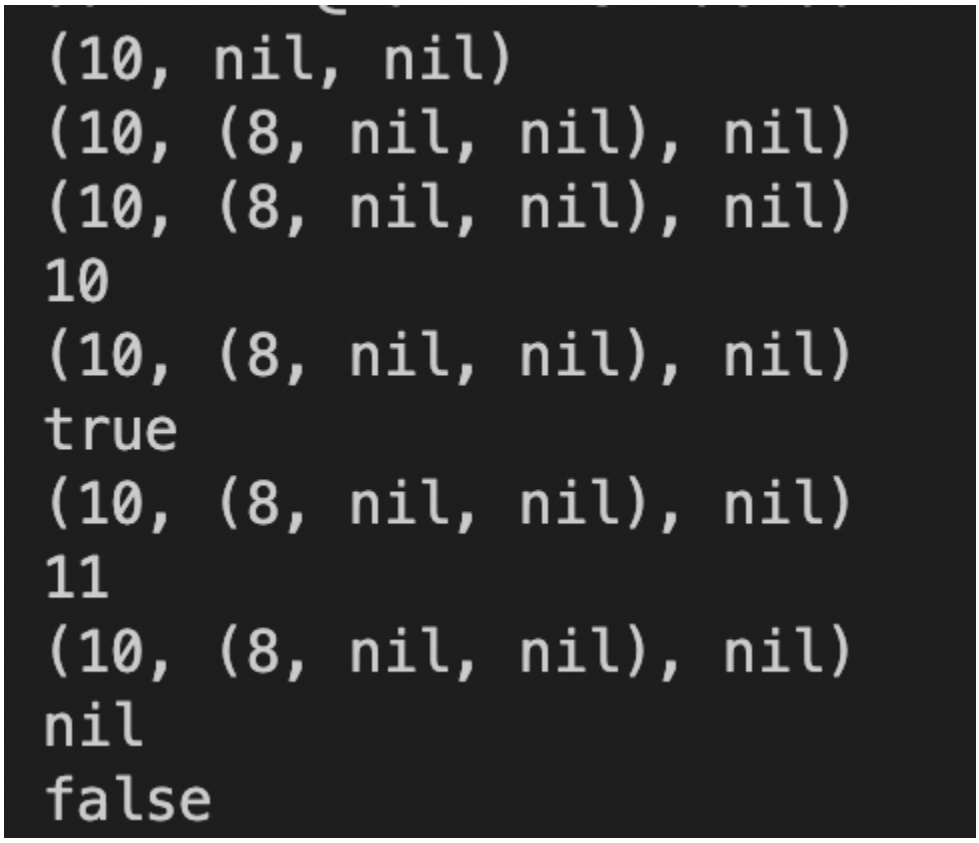
```
(search_bst
        (insert_bst_elm
           (insert_bst_elm
              (insert_bst_elm
                 (insert_bst_elm
                    (insert_bst_elm
                       (insert_bst_elm
                          (insert_bst_elm nil 10)
                       8)
                    12)
                 6)
              9)
           11)
        15)
     14)
```

```
(10, nil, nil)
(10, (8, nil, nil), nil)
(10, (8, nil, nil), (12, nil, nil))
(10, (8, (6, nil, nil), nil), (12, nil, nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, nil, nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
14
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
(12, (11, nil, nil), (15, nil, nil))
(15, nil, nil)
nil
false
```

The output is incremental, i.e. it prints the entire array when an insert is called. Thus, the output shows the status of the new array created after every insert operation. The search operation prints the array it is taking as input and the element. Then it prints the relevant array for the element to be searched, i.e it shows the new array being considered for searching (either left or right of the original array) and finally prints if the element was found or no. If found, outputs true else false.

**Example test 3 - bst with let binding**

```
(let ((y
        (let ((x (insert_bst_elm nil 10)))
                (insert_bst_elm x 8))
        ))
        (block
                (print (search_bst y 10))
                (print (search_bst y 11))
      )
)
```




The output is incremental, i.e. it prints the entire array when an insert is called. Thus, the output shows the status of the new array created after every insert operation. The search operation prints the array it is taking as input and the element. Then it prints the relevant array for the element to be searched, i.e it shows the new array being considered for searching (either left or right of the original array) and finally prints if the element was found or no. If found, outputs true else false.

```
rustup target add x86_64-apple-darwin
info: component 'rust-std' for target 'x86_64-apple-darwin' is up to date
rustc -L tests/ -lour_code:bst runtime/start.rs --target x86_64-apple-darwin -o tests/bst.run
aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % ./tests/bst.run
(10, nil, nil)
(10, (8, nil, nil), nil)
(10, (8, nil, nil), nil)
10
(10, (8, nil, nil), nil)
true
(10, (8, nil, nil), nil)
11
(10, (8, nil, nil), nil)
nil
false
(10, nil, nil)
(10, (8, nil, nil), nil)
(10, (8, nil, nil), (12, nil, nil))
(10, (8, (6, nil, nil), nil), (12, nil, nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, nil, nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
9
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
(8, (6, nil, nil), (9, nil, nil))
(9, nil, nil)
true
(10, nil, nil)
(10, (8, nil, nil), nil)
(10, (8, nil, nil), (12, nil, nil))
(10, (8, (6, nil, nil), nil), (12, nil, nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, nil, nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), nil))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
14
(10, (8, (6, nil, nil), (9, nil, nil)), (12, (11, nil, nil), (15, nil, nil)))
(12, (11, nil, nil), (15, nil, nil))
(15, nil, nil)
nil
false
false
```

The accumulative output of the entire bst program.

# Pick two other programming languages you know that support heap-allocated data, and describe why your language's design is more like one than the other.

C language

It requires all the elements to be of the same type, which means that if it is an array of ints, then it cannot allow nested arrays as its elements.
(array 4 (array 5 6)) -> This structure cannot be represented in C because the elements are an integer and a nested array. C allows arrays with same type of elements only.

For Example -

```
// compiles successfully
int arr[5] = {1, 2, 3, 4, 5};

// compilation error. We cannot directly initialize an array with another array.
int arr[2] = {4,  {5, 6}};
```

Python language

It can allow nested arrays within an array with other elements as ints. Allowing nested arrays requires dynamic memory allocation.
(array 4 (array 5 6)) -> This structure can easily be represented in python with lists.

For Example -

```
# compiles successfully
arr = [4, 5, 6]

# compiles successfully
arr = [4, [5, 6]]
```

In this code, arr2 is a list containing the integers 5 and 6. Then, arr is a list containing the integer 4 and the list arr2. So, arr has a nested structure with arr2 as one of its elements.

We can access individual elements in arr using indexing. For example, arr[0] would give the value 4, and arr[1] would give the list [5, 6]. Similarly, we can access elements in arr2 using indexing like arr2[0] (giving 5) and arr2[1] (giving 6).

Egg-eater language

Egg-eater language is more similar to python as it also supports dynamic memory allocation and can represent (array 4 (array 5 6)), which is a nested array within an array with the other element being an integer.
We can access individual elements in array using indexing. For example, (getIndex (array 4 (array 5 6)) 1) would give the value 4, and (getIndex (array 4 (array 5 6)) 2)  would give the list (array 5 6). Similarly, we can access elements in (array 5 6) using indexing  (getIndex (array 5 6) 1)  like (giving 5) and (getIndex (array 5 6) 2) (giving 6).

# list of the resources used to complete the assignment

- Professor Joe Politz's office hours
- Edstem posts
    - https://edstem.org/us/courses/38748/discussion/3130184
    - https://edstem.org/us/courses/38748/discussion/3136182
    - https://edstem.org/us/courses/38748/discussion/3125816
    - https://edstem.org/us/courses/38748/discussion/3121483