CSE 231: Egg Eater

Praveen Kumar Ramesh (A59020245)

May 24, 2023

1 Introduction

The goal of the assignment is to make the `Snek` language support heap-allocated structures. By introducing heap-allocated structures, `Snek` offers enhanced data manipulation capabilities and allows programmers to implement more sophisticated data structures like lists and trees. This report outlines the design choices, challenges faced, and the solutions employed during the development of the heap allocation feature in `Snek` compiler, highlighting the use of heap memory to store such structures and how different operations on these heap-allocated data affect the heap memory.

2 Grammar

The below production rules represent the grammar of the updated language.

```
<prog> := <defn>* <expr>
<defn> := (fun (<name> <name>*) <expr>)
<expr> :=
  | <number>
  | true
  | false
  | null                                                    (new!)
  | input
  | <identifier>
  | (let (<binding>+) <expr>)
  | (<op1> <expr>)
  | (<op2> <expr> <expr>)
  | (set! <name> <expr>)
```

```
  | (if <expr> <expr> <expr>)
  | (block <expr>+)
  | (loop <expr>)
  | (break <expr>)
  | (array <expr>*)                                              (new!)
  | (getIndex <expr> <expr>)                                     (new!)
  | (setIndex <expr> <expr>)                                     (new!)
  | (append <expr> <expr> <expr>)                                (new!)
  | (<name> <expr>*)

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

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

3 Design Choices

Memory representations

The size of a word in the memory is 64 bits. Unless specified, all the values occupy one word.

Number: Signed 63-bit numbers are stored with the least significant bit as 0. For example, $(0000\ 0000\ 0000\ 0008)_{16}$ represents the value 4. As shorthand notation, we represent this value as 0x8. We follow this shorthand notation throughout this report.

Boolean: To distinguish from numbers, the first two least significant bits of boolean are set to 1. Therefore 0x3 represents $false$ and 0x7 represents $true$.

Null: 0x1 represents null.

Array: To distinguish between boolean and numbers, the least significant bit is set to 1 and the second least significant bit is set to 0. The reference to the starting address of the array in the heap memory can be easily obtained by subtracting 1 from the value. Since the word size is 8 bytes, the last two bits of the addressable memory address are always zero (assuming alignment conditions are satisfied). This distinguishes array from null in the memory. For example 0x4f68bb31 represents the array with starting address, 0x4f68bb30

Other design choices regarding heap allocation are discussed later in this report.

4 Documentation

Array Creation: `(array <expr>*)` → `<expr>`

Input: zero or more expressions.
Output: evaluates the expressions and returns a reference to the starting address of the array in the heap memory.
Details: `array` takes zero or more expressions and allocates heap memory for the result of the expression. For n expressions, it allocates $1 + \sum_{i=1}^{n} size(< expr >_i)$ words in the heap, where,

$$size(< number > | < identifier > | input | true | false | null) = 1$$

That is each expression can evaluate to a number, boolean, command line input, identifier, null, or another array. The first word on the array contains the length of the array followed by n words to store the result of the n expressions. It returns the

Example 1:

```
(let ((a (array 2 (+ -3 4) true null)))
    (print a))
```

Output:

```
[Array: 2, 1, true, null]
[Array: 2, 1, true, null]
```

Example 2:

```
(print (
    array
        (array 1 2)
        (array 1)
        (array
            1
            2
            (array 1 2)
        )
    )
)
```

Output:

```
[Array: [Array: 1, 2], [Array: 1], [Array: 1, 2, [Array: 1, 2]]]
[Array: [Array: 1, 2], [Array: 1], [Array: 1, 2, [Array: 1, 2]]]
```

Length: `(len <expr>) → <number>`

Input: an expression that would evaluate to an array.
Output: length of the array.
Details: `len` evaluates the expressions and returns the length of the resultant array. The length of the array is the number of elements stored in the heap that can be directly referenced by the array at $depth = 1$.
Exceptions:

- If the expression in the argument does not evaluates to an array, a runtime exception is thrown and the execution halts immediately.

Example 1:

```
(print (len (array )))
```

Output:

```
0
0
```

Example 2:

```
(print
    (len
        (array 3 (array 100) (array -4 true (array 1 null))
              (array ))
    )
)
```

Output:

```
4
4
```

Example 3:

```
(print (len (+ 2 3)))
```

Output:

```
runtime error: invalid - expected an array
```

Retrieve: (getIndex e_1:<expr> e_2:<expr>) $\rightarrow$ <number>

Input:

- e_1:<expr> := expression that evaluates to an array
- e_2:<expr> := expression that evaluates to a number; the index of the value to be looked-up in the array.

Output: value stored in the index specified.
Details: `getIndex` accepts two expressions, where the first expression must evaluate to an array and the second expression must evaluate to a number. The array follows 0-based indexing. Therefore `(getIndex a 0)` returns the first element in the array `a`, `(getIndex a 1)` returns the second element in the array `a` and so on.
Exceptions: A runtime exception is thrown and the execution halts immediately in the following cases.

- If the expression in the first argument does not evaluate to an array
- If the expression in the second argument does not evaluate to a number
- If the index value is greater than the size of the list.

Example 1:

```
(let
    (
        (a  (array 3 4 (array 77 88)))
    )
    (block
        (print (getIndex a 0))
        (print (getIndex a 2))
        (print (getIndex (getIndex a 2) 0))
    )
)
```

Output:

```
3
[Array: 77 88]
77
77
```

Example 2:

```
(let ((a (array 1 2 3 5 7 11)))
    (block
        (print (getIndex a 3))
        (print (getIndex a false))
    )
)
```

Output:

```
5
runtime error: invalid - expected number for index
```

Example 3:

```
(let ((a (array flase true false)))
    (block
        (print (getIndex a 2))
        (print (getIndex a 3))
    )
)
```

Output:

```
false
runtime error: invalid - index out of bounds
```

Example 4:

```
(print (getIndex (&& true false) 0)
```

Output:

```
runtime error: invalid - expected an array
```

Update: `(setIndex` e_1`:<expr>` e_2`:<expr>` e_3`:<expr>)` $\rightarrow$ `<expr>`

Input:

- e_1`:<expr>` := expression that evaluates to an array
- e_2`:<expr>` := expression that evaluates to a number; the index of the value to be updated in the array.
- e_2`:<expr>` := expression that evaluates to a value with which update should happen.

Output: the value at the index is updated with the newly evaluated value and the newly evaluated value is returned.
Details: `setIndex` updates the element in the array in the specified index with the new value. The update happens in place. That is no new memory is created in the heap, rather, the value is overwritten in the heap memory.
Exceptions: A runtime exception is thrown and the execution halts immediately in the following cases.

- If the expression in the first argument does not evaluate to an array
- If the expression in the second argument does not evaluate to a number
- If the index value is greater than the size of the list.

Example 1:

```
(let ((a (array 2 4 5)))
    (block
        (print a)
        (setIndex a 0 1)
        (print a)
    )
)
```

Output:

```
[Array: 2, 4, 5]
[Array: 1, 4, 5]
[Array: 1, 4, 5]
```

Example 2:

```
(let ((a (array 22 87)))
    (let ((b (array 26 34 88)))
        (let ((c (array 56 78 90 16246)))
            (let ((d (array 10 11 12 13)))
                (block
                    (setIndex d 2 a)
                    (setIndex c 3 d)
                    (setIndex b 2 c)
                    (setIndex a 1 b)
                    (print a)
                )
            )
        )
    )
)
```

Output:

```
[Array: 22, [Array: 26, 34, [Array: 56, 78, 90,
                                    [Array: 10, 11, [...], 13]]]]
[Array: 22, [Array: 26, 34, [Array: 56, 78, 90,
                                    [Array: 10, 11, [...], 13]]]]
```

Example 3:

```
(let ((a (array 2 4 5)))
    (block
        (setIndex a 3 1)
        (print a)
    )
)
```

Output:

```
runtime error: invalid - index out of bounds
```

Append: (append e_1:<expr> e_2:<expr>) $\rightarrow$ <expr>

Input:

- e_1:<expr> := expression that evaluates to an array
- e_2:<expr> := expression that evaluates to a value to be appended

Output: The reference to the newly created array with the element appended to the end of the array is returned.
Details: Appends the new value to the end of the array. This operation creates a new array with $n + 1$ from an array with n values by copying the first n elements and appending the new value at the end. The reference to the newly created array is returned.
Exceptions: A runtime exception is thrown and the execution halts immediately in the following cases.

- If the expression in the first argument does not evaluate to an array

Example 1:

```
(let ((a (array 6762 3279 25)) (b (array 1 2 3)))
    (block
        (print a)
        (set! a (append a 69))
        (print a)
        (set! a (append a b))
        (print a)
    )
)
```

Output:

```
[Array: 6762, 3279, 25]
[Array: 6762, 3279, 25, 69]
[Array: 6762, 3279, 25, 69, [Array: 1, 2, 3]]
[Array: 6762, 3279, 25, 69, [Array: 1, 2, 3]]
```

Example 2:

```
(print (append null 3))
```

Output:

```
runtime error: invalid - expected an array
```

5 Heap Allocation

Before the program execution beings, we allocate a huge chuck of memory using rust and pass the starting address of the memory as an argument to the assembly code. The heap is destroyed by the rust's garbage collector once the assembly instructions finish execution or throw an exception during the course of execution.

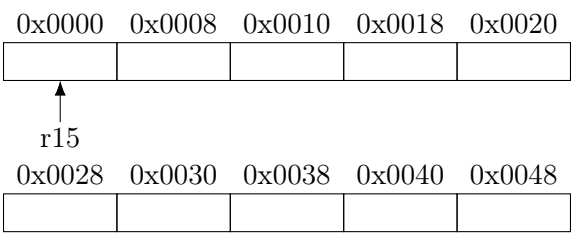
5.1 Heap Structure

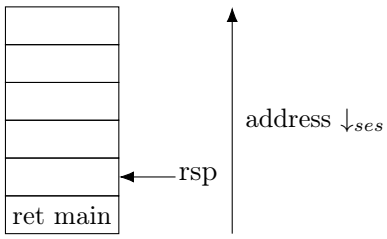
When an array is constructed, we first compute the individual elements of the array and allocate the heap for them if necessary and then allocate the heap for the array. The first element in the memory allocated for the array represents the length of the array(n). The next n words in the heap contain the values of the elements.

The rest of the section discusses in detail how the heap memory is used during code execution.

5.2 Example-1:

Let's assume that the heap memory starts from the address 0x0000. Although the memory address is a 64-bit value, we assume that the most significant 48 bits are always 0. Therefore at the start of the execution, the heap memory and tack memory would look like this,

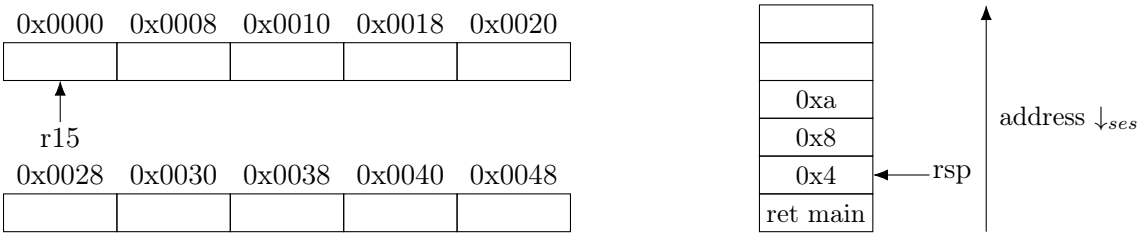




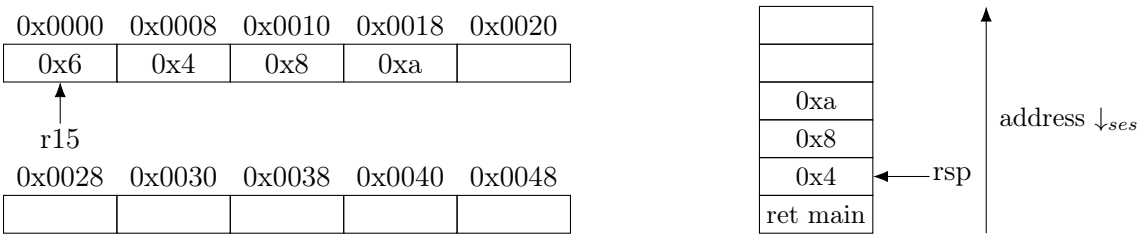
Consider the following program,

```
(let ((a (array 2 4 5)))
    (print a)
)
```

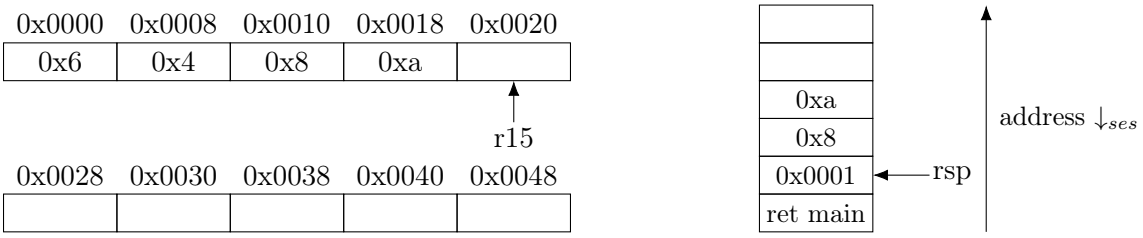
First, the expressions in the array are individually evaluated and stored in the stack. The first element is evaluated and temporarily stored in [rsp], second element in [rsp − 8] and third element in [rsp − 16]. Therefore the memory would now look like this,



Then the size of the array is stored in the location pointed by the r15, and the next 3 consecutive words are used to store the 3 values in the array. Therefore the memory would now look like this,



Finally, 1 is added to the address pointed by r15 and that value is copied to [rsp] bound to the variable `a`. The r15 is now made to point to the next available memory in the heap. Hence at the end of the binding operation, the memory would look like this,

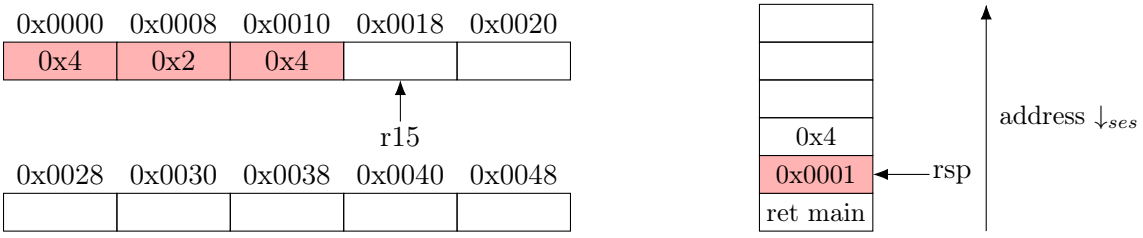


5.3 Example-2:

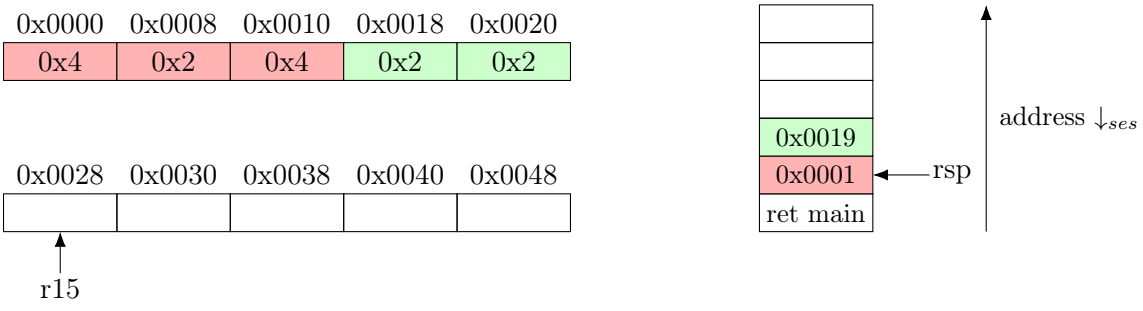
Now consider the following program,

```
(let ((a (array
            (array 1 2)
            (array 1)
            true
            2
         )
      ))
      (print a)
)
```

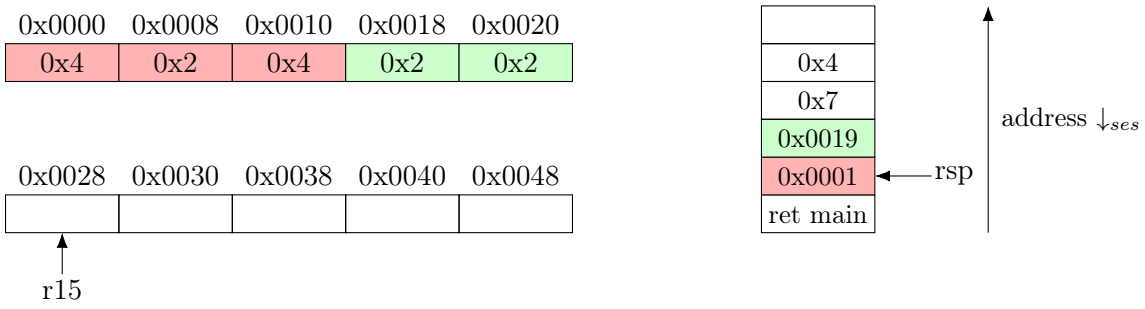
To allocate the heap for the variable a, we must first compute and allocate the heap(if necessary) for each of its elements. Therefore the first expression in the array is evaluated first. Since the first element is an array, we need the heap to store it. We do this as illustrated in the previous example. The result of this evaluation (the address + 1, of the starting heap address) is stored in the stack. After evaluating the first expression in the array, the memory would look like this,



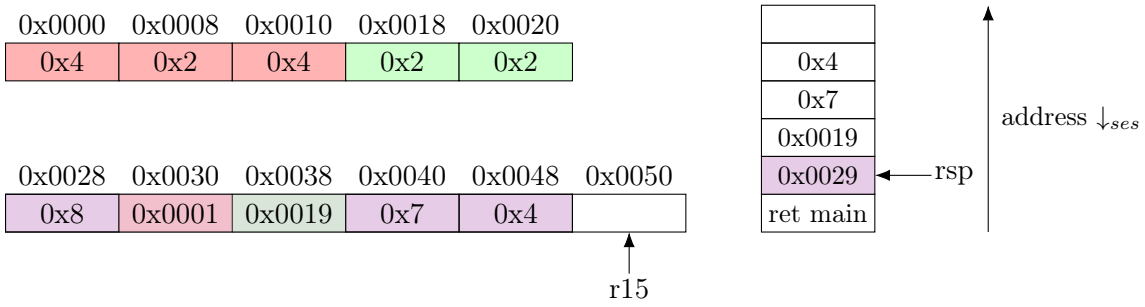
Similarly, the second element is also evaluated, the heap is allocated and the address is stored in the stack. After evaluating the second element the heap would look like this,



Similarly, the last two elements are evaluated and pushed to the stack to be later copied to the heap. After evaluating all the expressions in the array and before copying them to the heap, the memories would look like this,



Finally, all the values are moved to the heap. The state of the memory after the copy from stack to heap is,

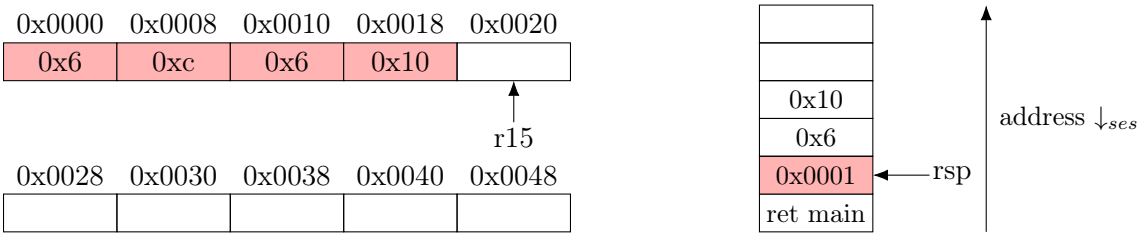


5.4 Example-3:

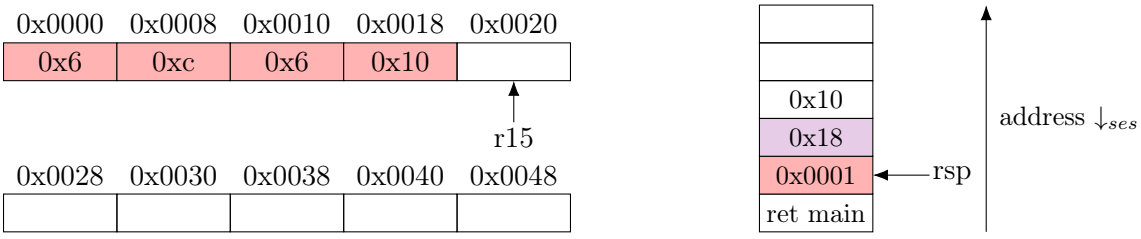
Now consider the below program for appending an element to the array,

```
(let ((a (array 6 3 8)))
    (set! a (append a 12))
)
```

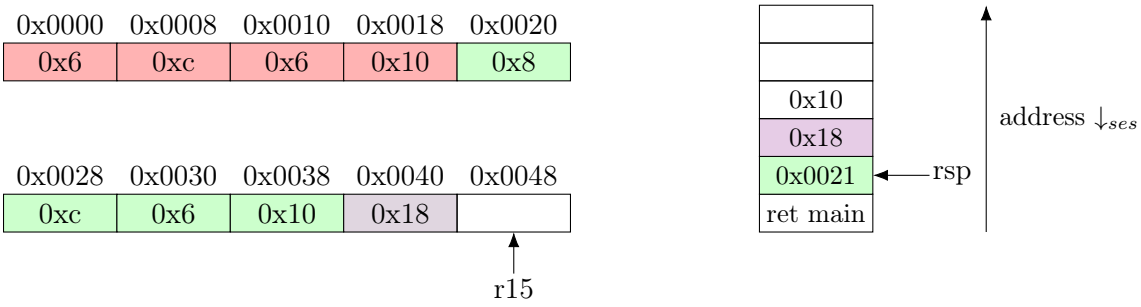
After the array construction, the heap and stack would be in the below state,



The new value is now computed and stored in the stack. Therefore the memory would look like this,



Finally, we copy the entire array to the next available memory in the heap and append the element to the last of the array. After copy operation, the `set!` updates the reference stored in `a`. Therefore after the execution, the memory would look like this,



6 Tests:

In this section, we look into some of the test cases for heap allocation.

simple_examples.snek

Code:

```
(let
    ((a (array 44 true (array -1 null))))
    (block
        (print (getIndex a 0))
        (print (getIndex a 2))
        (print (getIndex (getIndex a 2) 1))
    )
)
```

Output

```
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler » ./tests/simple_examples.run
44
[Array: -1, null]
null
null
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler »
```

Interesting feature: The nested indexing to access the `null` mimics how $2D$ arrays are indexed in other programming languages. And this idea can be implemented in any n dimensional array. For convenience, we can write a helper function as below to fetch an element from a $2D$ array.

```
(fun (get_element mat i j)
    (getIndex (getIndex mat i) j)
)
```

error_tag.snek

Code:

```
(let ((a (if input
             (array 1 2 3)
             false
         )
     ))
    (print (getIndex a 1))
)
```

Output:

```
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler » ./tests/error_tag.run true
2
2
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler » ./tests/error_tag.run false
runtime error: invalid - expected an array
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler 1 »
```

Interesting feature: The type checking happens during run time. Depending on the command line input, the type of the value bound to `a` is either an array or a boolean.
Explanation: The error is thrown when the instructions for `getIndex` are executed, during runtime. After evaluating the expression for the first argument in `getIndex`, we check for the type of the evaluated value. If it is not an array, then the execution halts immediately and throws an exception.

error_bound.snek

Code:

```
(let ((a (array true true false null))
      (index (if input 4 2)))
     (block
        (setIndex a index 4)
        (print a)
     )
)
```

Output:

```
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler » tests/error_bound.run true
runtime error: invalid - index out of bounds
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler 1 » tests/error_bound.run false
[Array: true, true, 4, null]
[Array: true, true, 4, null]
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler »
```

Explanation: During runtime, the expression for the index is evaluated and compared with the size of the array, if the index value is not less than the size of the array, then the execution halts immediately and an exception is thrown.

error3.snek

Code:

```
(let ((a (array 1 2 3 5 7 11)))
    (block
        (print (getIndex a 3))
        (print (getIndex a false))
    )
)
```

Output:

```
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler » ./tests/error3.run
5
runtime error: invalid - expected number for index
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler 1 »
```

Explanation: The second argument to the `getIndex` must evaluate to a whole number (less than the size of the array). In this case, the type checks for the index happen before retrieving the value from the heap. Since the second `getIndex` operation had a boolean for the index, an exception is thrown during the runtime and the executions halt immediately.

points.snek

Code:

```
(fun (add_points point1 point2)
    (let (
            (p1 (len point1))
            (p2 (len point2))
         )
         (if (= p1 p2)
             (let ((result (array )) (i 0))
                  (loop
                        (if (= p1 0)
                            (break result)
                            (block
                               (set! result (append
                                             result
                                             (+ (getIndex point1 i)
                                             (getIndex point2 i))
                                            )
                               )
                               (set! i (+ i 1))
                               (set! p1 (- p1 1))
                            )
                        )
                  )
             )
             point1
         )
    )
)
(let ((point1 (array 1 2)) (point2 (array 4 5)))
    (print (add_points point1 point2))
)
```

Output:

```
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler » tests/points.run
[Array: 5, 7]
[Array: 5, 7]
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler »
```

Interesting feature: This code is generalised to any dimensional points. The function iterates through each dimension and appends the value to the array.

bst.snek

Code:

```
(fun (is_leaf_node node)
    (if (&& (isnull (getIndex node 1))
            (isnull (getIndex node 2)))
        true
        false
    )
)

(fun (check_if_null_and_compare_ge node element)
    (if (isnull node)
        false
        (>= (getIndex node 0) element)
    )
)

(fun (create_node element)
    (array element null null )
)

(fun (search_tree root element)
    (if (isnull root)
        false
        (if (= (getIndex root 0) element)
            true
            (if (is_leaf_node root)
                false
                (if (check_if_null_and_compare_ge
                                (getIndex root 1)
                                element
                    )
                    (search_tree (getIndex root 1) element)
                    (search_tree (getIndex root 2) element)
                )
            )
        )
    )
)
```

```
(fun (insert_tree root element)
    (if (isnull root)
        (create_node element)
        (block
            (if (>= (getIndex root 0) element)
                    (setIndex root 1
                            (insert_tree (getIndex root 1) element))
                    (setIndex root 2
                            (insert_tree (getIndex root 2) element))
            )
            root
        )

    )
)

(let ((a (array 7
                (array 5
                        (array 2 null  null)
                        null
                )
                (array 15
                        (array 9 null null)
                        (array 20 null null)
                )
        )
     ))
    (block
        (print (search_tree a 20))
        (print (search_tree a 1))
        (set! a (insert_tree a 1))
        (print a)
        (set! a (insert_tree a 10))
        (print a)
    )
)
```

Output:

```
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler » ./tests/b
st.run
true
false
[Array: 7, [Array: 5, [Array: 2, [Array: 1, null, null], null], null],
[Array: 15, [Array: 9, null, null], [Array: 20, null, null]]]
[Array: 7, [Array: 5, [Array: 2, [Array: 1, null, null], null], null],
[Array: 15, [Array: 9, null, [Array: 10, null, null]], [Array: 20, null
, null]]]
[Array: 7, [Array: 5, [Array: 2, [Array: 1, null, null], null], null],
[Array: 15, [Array: 9, null, [Array: 10, null, null]], [Array: 20, null
, null]]]
(base) Praveens-MacBook-Pro :: Desktop/cse231/Snek-Compiler »
```

Interesting feature: The above logic can be modified to be generic to an n-ary tree. The first element in the array representing a node is the value of the node and the rest of the elements in the array is the child of the node. Since the language supports `append` operation, we can also add functions to balance the binary tree.

7 Comparison with Python and C

Like Python and C, Snek uses 0-based indexing. However, unlike C, Snek supports dynamic memory allocation in the heap. Arrays in C have constant size and cannot be changed after initialising them. Further, we need a separate variable to keep track of the size of the array in C. In Snek, we track the size of the array in the memory allocated for the array.

Arrays in Snek, closely resemble lists in Python. Like lists in Python, the size of the array can change during the course of the execution. Further, the `append` operation in Snek mimics the low-level implementation of `append` in Python. Every time an append is called for an array, a new copy of the array is created with that additional element at the end of the array.

8 Acknowledgements and credits

The runtime error presented in `error3.snek` was based on the discussion in EdStem thread, (thanks to Edward Wang for his input). I, however, have extended the idea to check for negative indices as well. I would like to thank Professor Joe Politz for his discussions in classes and starter code to print recursive structures. I adapted the code in my compiler to support arrays. I would like to thank all the TAs for their constant support and timely responses to the doubts raised in EdStem.