# CSE 231: Green Snake

Praveen Kumar Ramesh (A59020245)

June 12, 2023

## 1 Introduction

The goal of the assignment is to extend the `Snek` language to structural update and comparison for heap-allocated data. We allow structural update by introducing syntax to update an element in an array and to append an element to the array. Further, we also explore algorithms to perform reference and structural similarity checks over these heap-allocated data. Later, we extend this feature to support cyclic structures. This report outlines the design choices, challenges faced, and the solutions employed during the development of the aforementioned 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
  | 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>*)
  | (getIndex <expr> <expr>)
  | (setIndex <expr> <expr>)                                    (new!)
  | (append <expr> <expr> <expr>)                               (new!)
  | (<name> <expr>*)

<op1> := add1 | sub1 | isnum | isbool | print | len | isnull
<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

**Update:** (setIndex $e_1$:<expr> $e_2$:<expr> $e_3$:<expr>) → <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.

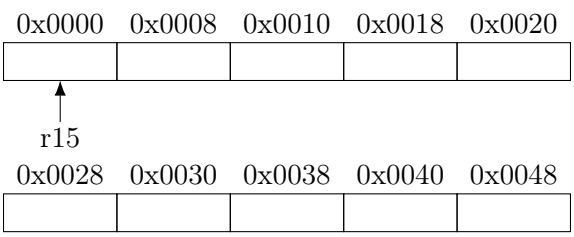
## 5.1 Heap Structure

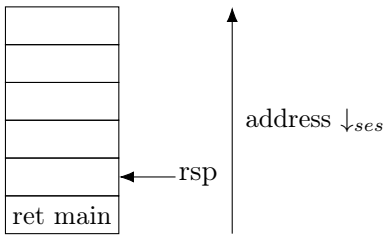
When an array is constructed, we first compute the individual elements of the array and allocate the heap for them if necessary and then allocate the heap for the array. The first element in the memory allocated for the array represents the length of the array($n$). The next $n$ words in the heap contain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

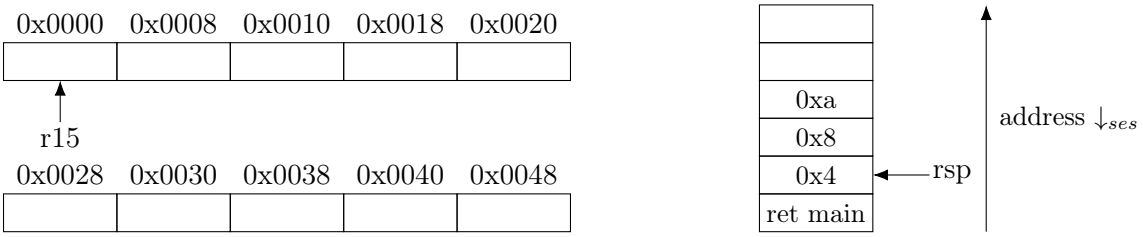


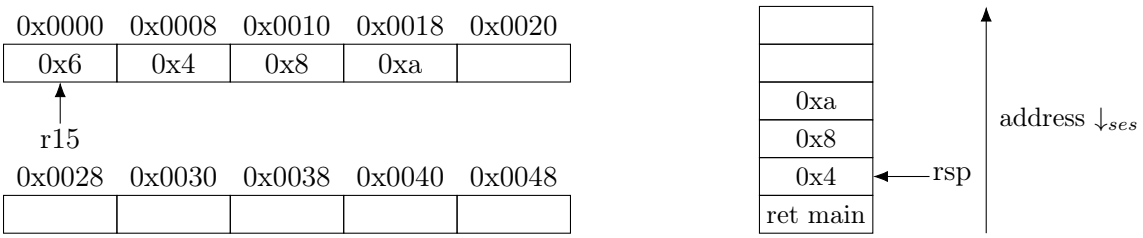

Consider the following program,

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

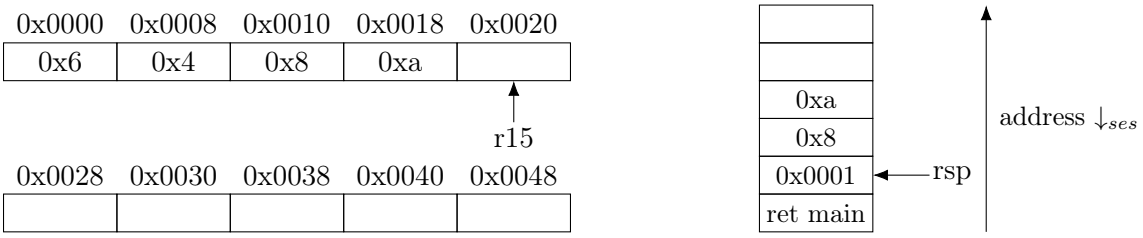
First, the expressions in the array are individually evaluated and stored in the stack. The first element is evaluated and temporarily stored in [rsp], second element in [rsp − 8] and third element in [rsp − 16]. Therefore the memory would now look like this,




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,

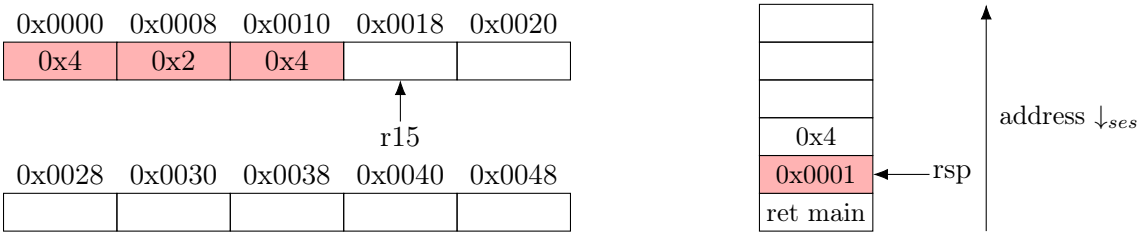


## 5.3 Example-2:

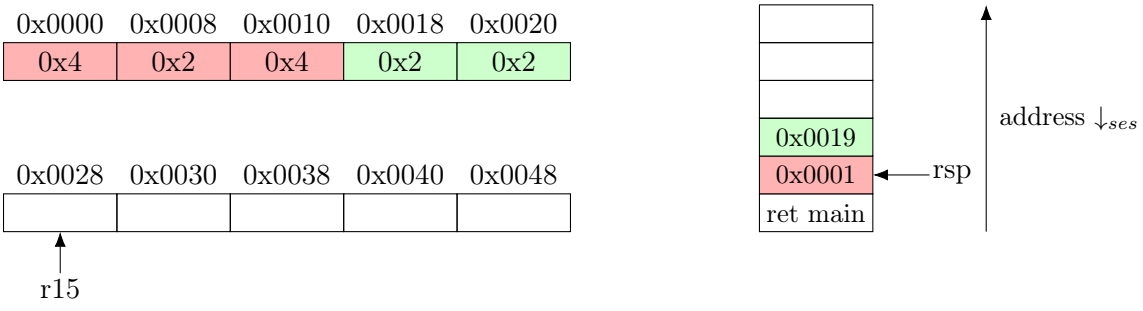
Now consider the following program,

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

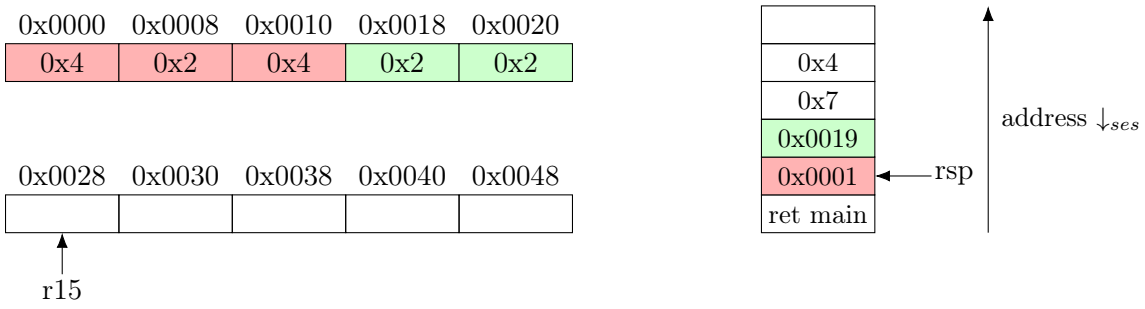
To allocate the heap for the variable $a$, we must first compute and allocate the heap(if necessary) for each of its elements. Therefore the first expression in the array is evaluated first. Since the first element is an array, we need the heap to store it. We do this as illustrated in the previous example. 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,



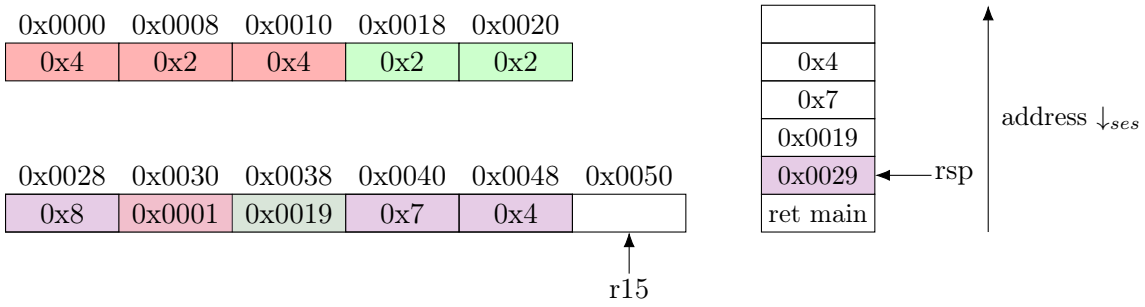

Similarly, the second element is also evaluated, the heap is allocated and the address is stored in the stack. After evaluating the second element the heap would look like this,



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,

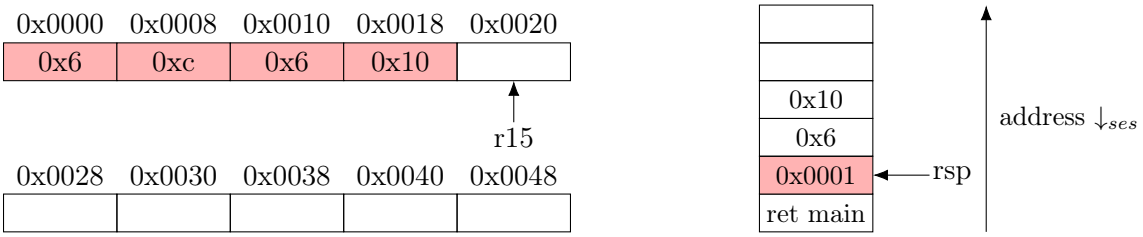


## 5.4 Example-3:

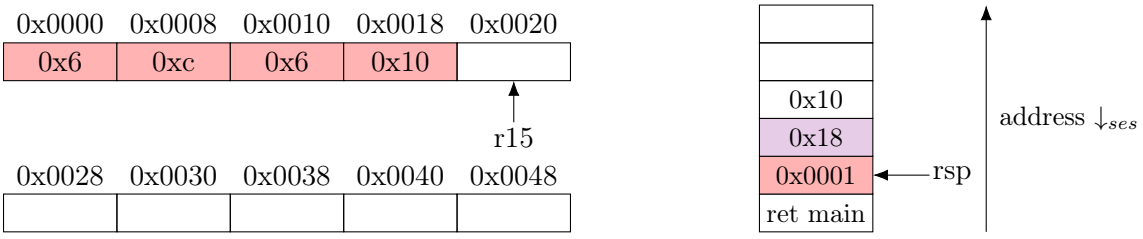
Now consider the below program for appending an element to the array,

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

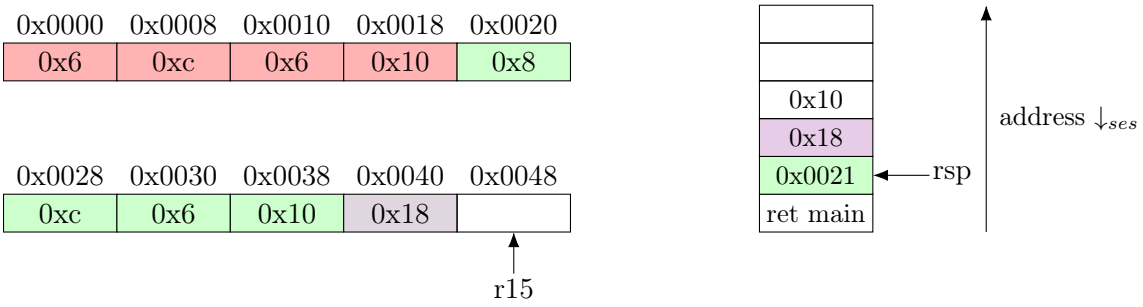
After the array construction, the heap and stack would be in the below state,




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

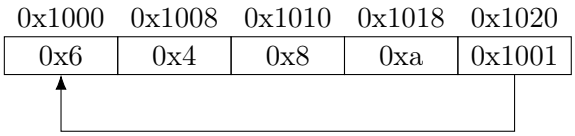



Finally, we copy the entire array to the next available memory in the heap and append the element to the last of the array. After copy operation, the `set!` updates the reference stored in `a`. Therefore after the execution, the memory would look like this,



# 6 Printing Cyclic Structure

We encounter cyclic structure when an element in the array has a direct or transitive reference to the same array. If such cases are not handled then the routine that prints this array would endlessly recur over the same set of elements in the cycle. For example, consider the following abstract depiction of heap memory.




The last element in the array is a direct reference to the array itself. Therefore, printing this array as it is would lead to infinite recursion. To handle this case we maintain a set “seen", which contains the reference to the arrays that have been visited during the current print cycle. So when the printing for the above array begins, the `seen` list contains a single element {0x1001}. Hence when we arrive at the last element in the array, we see that the reference in the current list of arrays is being printed. So we print the text `[...]` in place of the element to indicate the cycle. Formally, the below is the algorithm for the logic,

**Algorithm 1** An algorithm for printing cyclic arrays

**Require:** $a$ : reference to array,
  $seen$ : set of reference to actively printed array
**Ensure:** $a \neq null$
  **if** $a \in seen$ **then**
    print(“[...]")
  **end if**
  $n \leftarrow *a$ ▷ get the size of the array
  $seen \leftarrow seen \cup a$
  $i \leftarrow 1$
  **while** $i \leq n$ **do**
    **if** $a[i]$ is reference to array **then**
      print_array($a[i]$, $seen$) ▷ recursive call to the same function
    **else**
      print(a[i])
    **end if**
    $i \leftarrow i + 1$
  **end while**
  $seen \leftarrow seen \setminus a$

## Relevant code

```
fn snek_str(val : i64, seen : &mut Vec<i64>) -> String {
    if val == 7 { "true".to_string() }
    else if val == 3 { "false".to_string() }
    else if val % 2 == 0 { format!("{}", val >> 1) }
    else if val == 1 { "null".to_string() }
    else if val & 1 == 1 {
        if seen.contains(&val)  { return "[...]".to_string() }
        seen.push(val);
        let addr = (val - 1) as *const i64;
        let size_array = unsafe{ *addr } >> 1;
        let mut index = 1;
        let mut builder = vec![];
        while index <= size_array {
            let element = unsafe{ *addr.offset(index.try_into().unwrap()) };
            let stringified_element = snek_str(element, seen);
            builder.push(stringified_element);
            index += 1;
        }
        seen.pop();
        let stringified_array = "[Array: ".to_owned() + &builder.join(", ") + "]";
        return stringified_array;
    } else {
        format!("Unknown value: {}", val)
    }
}
```

## Tests

**cycle_print1.snek**

```
(let ((a (array 43 22 87)))
    (let ((b (array 75 26 34 88)))
        (let ((c (array 12 34 56 78 90 16246)))
            (let ((d (array 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)))
                (block
                    (setIndex d 11 a)
                    (setIndex c 5 d)
                    (setIndex b 3 c)
                    (setIndex a 2 b)
                    (print a)
                )
            )
        )
    )
)
```

**Output:**

```
[Array: 43, 22, [Array: 75, 26, 34, [Array: 12, 34, 56, 78, 90,
    [Array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, [...], 13, 14, 15]]]]
[Array: 43, 22, [Array: 75, 26, 34, [Array: 12, 34, 56, 78, 90,
    [Array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, [...], 13, 14, 15]]]]
```

**cycle_print2.snek**

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

**Output:**

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

**cycle_print3.snek**

```
(let
    ((a (array 1 32)) (b (array 3 a)) (c (array a b)))
    (block
        (setIndex a 0 b)
        (setIndex a 1 c)
        (setIndex b 0 c)
        (print a)
        (print b)
        (print c)
        (print (= a b))
        (print (= a c))
        (print (= b c))
        (print (== a b))
        (print (== a c))
        (print (== b c))
    )
)
```

**Output:**

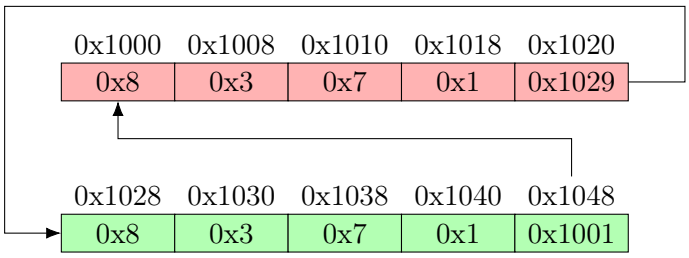
```
[Array: [Array: [Array: [...], [...]], [...]], [Array: [...],
            [Array: [...], [...]]]]
[Array: [Array: [Array: [...], [...]], [...]], [Array: [...],
            [Array: [...], [...]]]]
[Array: [Array: [Array: [...], [...]], [...]], [Array: [...],
            [Array: [...], [...]]]]
false
false
false
true
true
true
true
```

# 7 Structural equality

When comparing two values in the heap, we have two options,

- **Reference equality:** we compare the reference of the array and conclude that two arrays are same if they point to the same memory in the heap. This is a simple direct comparison of address at the first level (i.e., not iterating through the elements of the array and comparing them). In most cases, this comparison does not provide any useful insights.
- **Structural equality:** In this we check if corresponding individual elements of the array have the same value. We can extend this idea to cyclic arrays as well. In the case of cycles, we keep track of the values being currently compared and if the same comparison is encountered again we conclude that we had hit the base case in the recursion.

Consider the following example,




In this case, the two array are structurally similar, when queried for any index $i$ the values will be same in both arrays. The following depicts the algorithms for structural comparison

**Algorithm 2** An algorithm for structural comparison of arrays

**Require:**
$a1$ : reference to first array,
$a2$ : reference to second array,
$seen$ : set of references to actively compared arrays
**Ensure:** $a1 \neq null$ and $a2 \neq null$
**if** $(a1, a2) \in seen$ **then**
**return** $true$
**end if**
$n1 \leftarrow *a1$ ▷ get the size of the array
$n2 \leftarrow *a2$
**if** $n1 \neq n2$ **then**
**return** $false$
**end if**
$seen \leftarrow seen \cup (a1, a2)$
$i \leftarrow 1$
**while** $i \leq n$ **do**
**if** $a2[i]$ and $a2[i]$ are reference to arrays **then**
compare_array($a1[i]$, $a2[i]$, $seen$) ▷ recursive call
**else if** $a1[i] \neq a2[i]$ **then**
**return** $false$
**end if**
$i \leftarrow i + 1$
**end while**
$seen \leftarrow seen \setminus (a1, a2)$
**return** $true$

## Relevant Code Snippet

```
fn snek_compare(val1: i64, val2: i64, seen: &mut Vec<Vec<i64>>) -> i64 {
    if val1 == val2 {
        return 7;
    } else {
        if val1 == 1 || val2 == 1 {
            return 3;
        } else if val1 & 1 == 1 && val2 & 1 == 1 {
            let entry = vec![val1, val2];
            if seen.contains(&entry) {
                return 7;
            } else {
```

```
                seen.push(entry);
                let addr1 = (val1 - 1) as *const i64;
                let addr2 = (val2 - 1) as *const i64;
                let size_array1 = unsafe { *addr1 } >> 1;
                let size_array2 = unsafe { *addr2 } >> 1;
                if size_array1 != size_array2 {
                    return 3;
                } else {
                    let mut index = 1;
                    while index <= size_array1 {
                        let element1 = unsafe{
                                        *addr1.offset(index.try_into().unwrap())
                                       };
                        let element2 = unsafe{
                                        *addr2.offset(index.try_into().unwrap())
                                       };
                        let result = snek_compare(element1, element2, seen);
                        if result == 3 {
                            return 3;
                        }
                        index += 1;
                    }
                    seen.pop();
                    return 7;
                }
            }
        } else {
            return 3;
        }
    }
}
```

## Tests

**equal.snek**

```
(let ((a (array 1 2 3 4)))
    (let ((b a) (c (array 1 2 3 4)) (d (array 1 2 3 4 5)))
        (block
            (print (= a b))
            (print (== a b))
            (print (= a c))
            (print (== a c))
            (print (= a d))
            (print (== a d))
        )
    )
)
```

**Output:**

```
true
true
false
true
false
false
false
```

**cycle_test1.snek**

```
(let (
        (a1 (array 1 null 2))
        (a2 (array 3 4 null))
        (a3 (array 5 6 null))
        (b1 (array 1 null 2))
        (b2 (array 3 4 null))
        (b3 (array 5 6 null))
        (b4 (array 1 null 2))
     )
    (block
        (setIndex a1 1 a2)
        (setIndex a2 2 a3)
        (setIndex a3 2 a1)
        (setIndex b1 1 b2)
        (setIndex b2 2 b3)
        (setIndex b3 2 b4)
        (setIndex b4 1 b2)
        (print a1)
        (print b1)
        (print (= a1 b1))
        (print (== a1 b1))
    )
)
```

**Output:**

```
[Array: 1, [Array: 3, 4, [Array: 5, 6, [...]]], 2]
[Array: 1, [Array: 3, 4, [Array: 5, 6, [Array: 1, [...], 2]]], 2]
false
true
true
```

**cycle_test2.snek**

```
(let ((a (array 66 72 12 3)) (b (array 66 72 12 a)))
    (block
        (setIndex a 3 b)
        (print (= a b))
        (print (== a b))
    )
)
```

**Output:**

```
false
true
true
```

**cycle_test3.snek**

```
(let
    ((a (array 1 32)) (b (array 3 a)) (c (array a b)))
    (block
        (setIndex a 0 b)
        (setIndex a 1 c)
        (setIndex b 0 c)
        (print a)
        (print b)
        (print c)
        (print (= a b))
        (print (= a c))
        (print (= b c))
        (print (== a b))
        (print (== a c))
        (print (== b c))
    )
)
```

**Output:**

```
[Array: [Array: [Array: [...], [...]], [...]],
        [Array: [...], [Array: [...], [...]]]]
[Array: [Array: [Array: [...], [...]], [...]],
        [Array: [...], [Array: [...], [...]]]]
[Array: [Array: [Array: [...], [...]], [...]],
```

```
        [Array: [...], [Array: [...], [...]]]
false
false
false
true
true
true
true
```

# 8 Acknowledgements and credits

The test case `cycle_test1.snek` presented in this report was based on the discussion on the EdStem thread and also on this thread, (thanks to Adyanth Hosavalike asking this question and Professor for his input on this special case). 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.