

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 to 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.

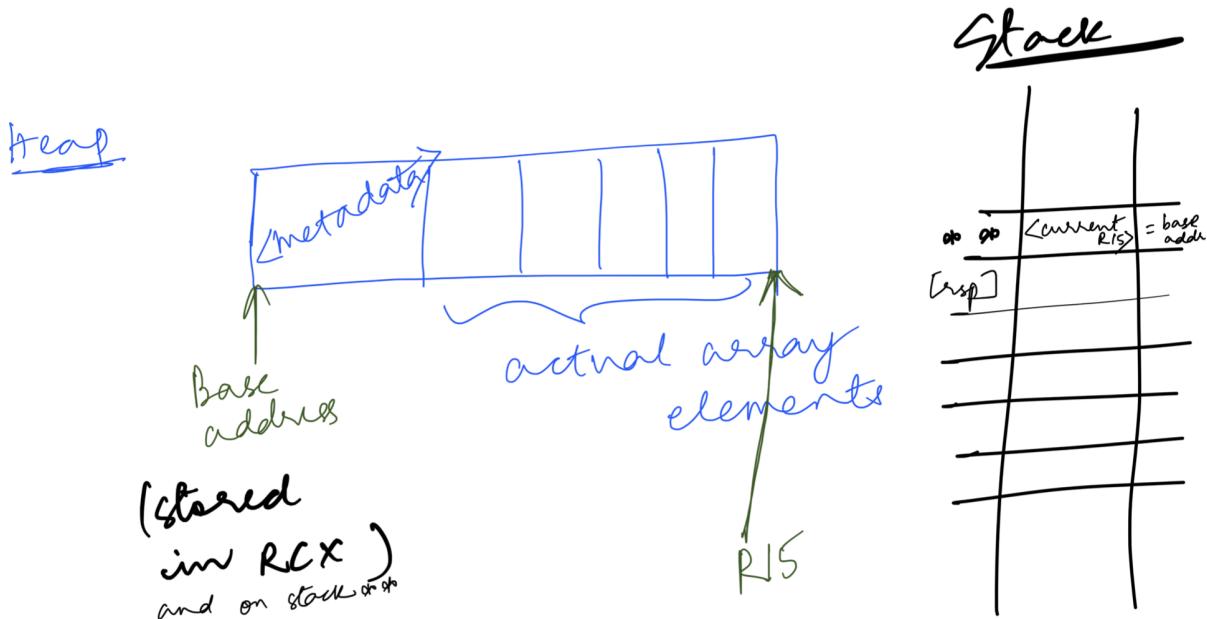
- ‘nil’ is used to represent ‘nil’ in arrays as discussed in class. It evaluates to itself. It is defined separately so that functions requiring isnil check can be handled. It is added to the enum Expr.
- (getIndex <expr> <expr>) is an expression for lookup that allows computed indexed access. The first 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.

The compiler should stop and report an error if:

- 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.

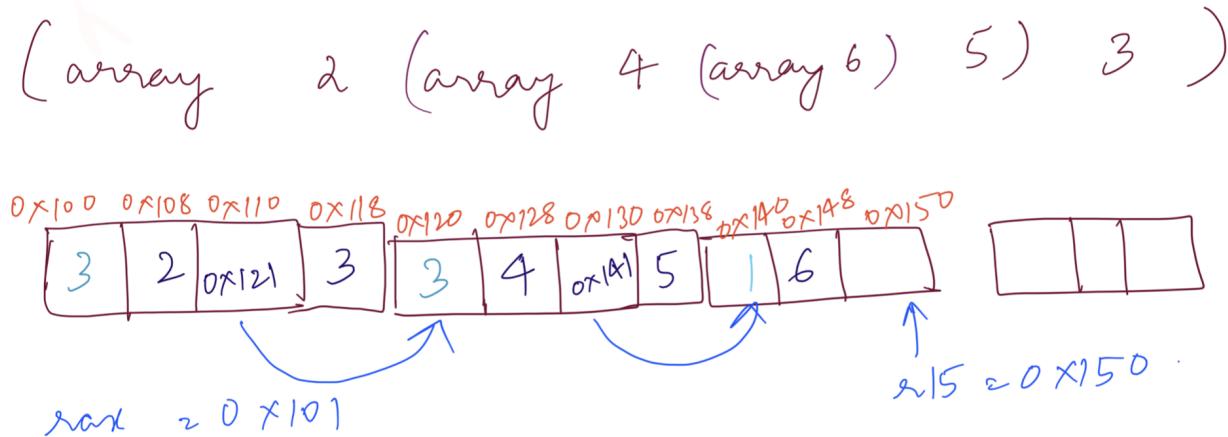
A diagram of how heap-allocated values are arranged on the heap, including any extra words like the size of an allocated value or other metadata.



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 incremented by 3 and then each of the array elements is 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.

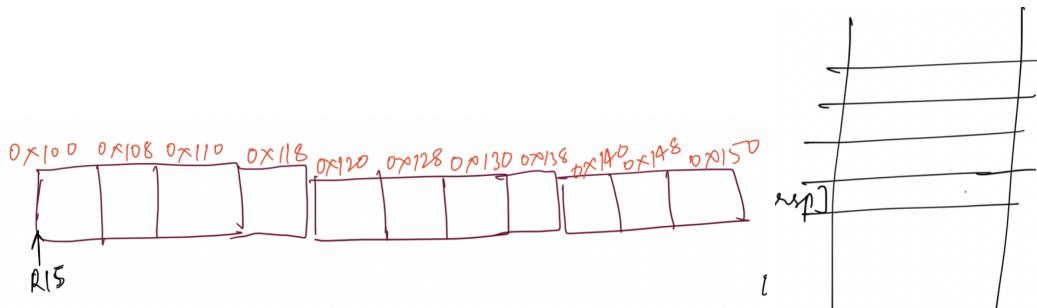
The array elements are evaluated and written directly to the heap memory, they are not written to the stack as any temporary storage. This is possible because R15 is incremented by reserving slots ($1 + \text{num of elements}$) for the current array being processed.

Detailed explanation

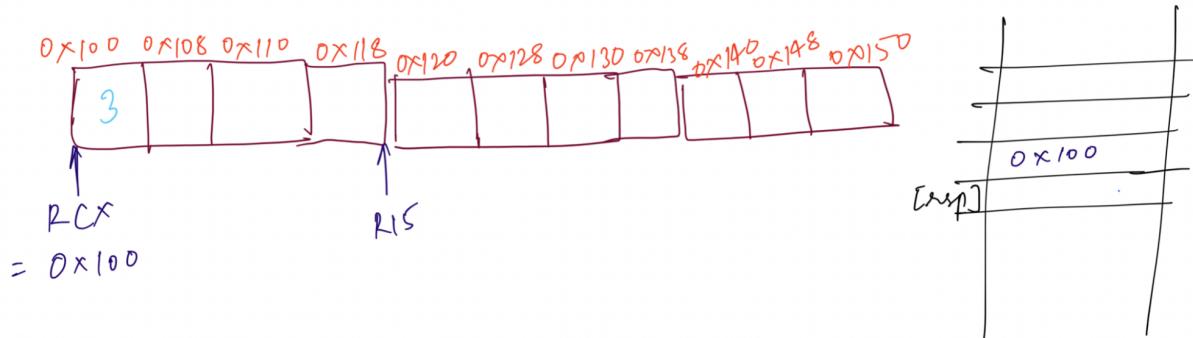
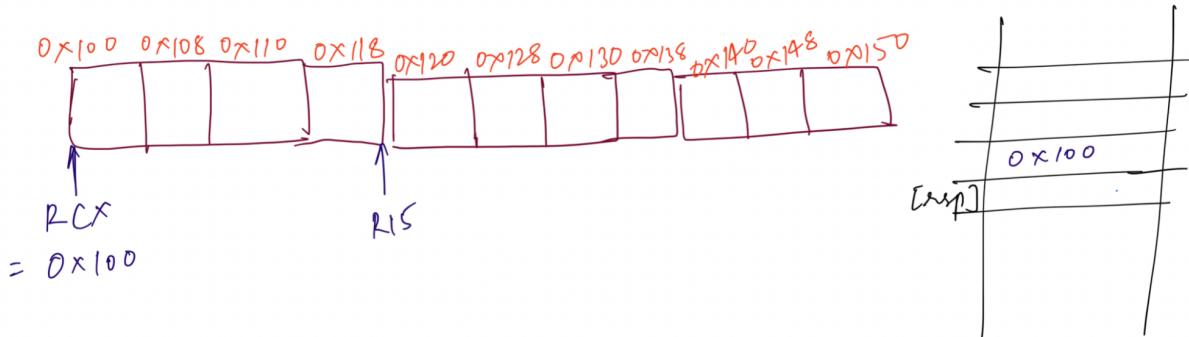
If array:

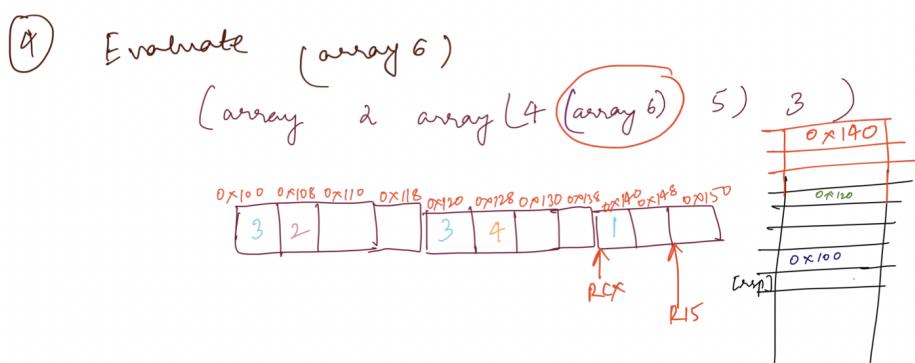
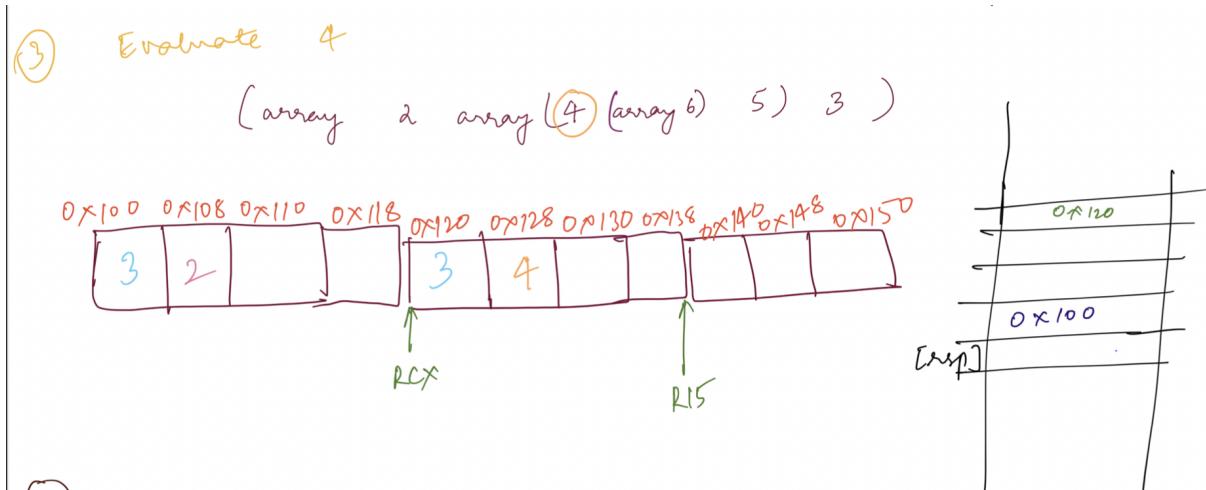
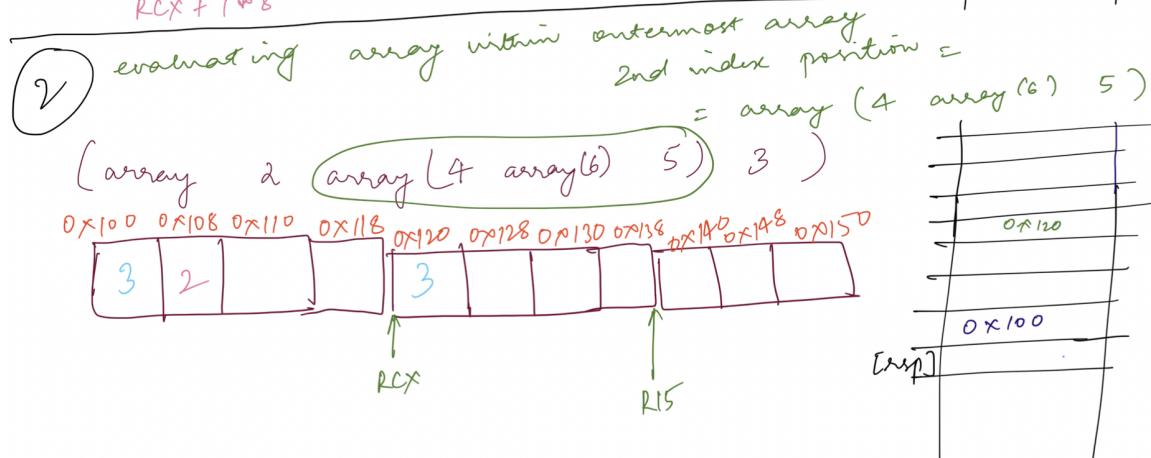
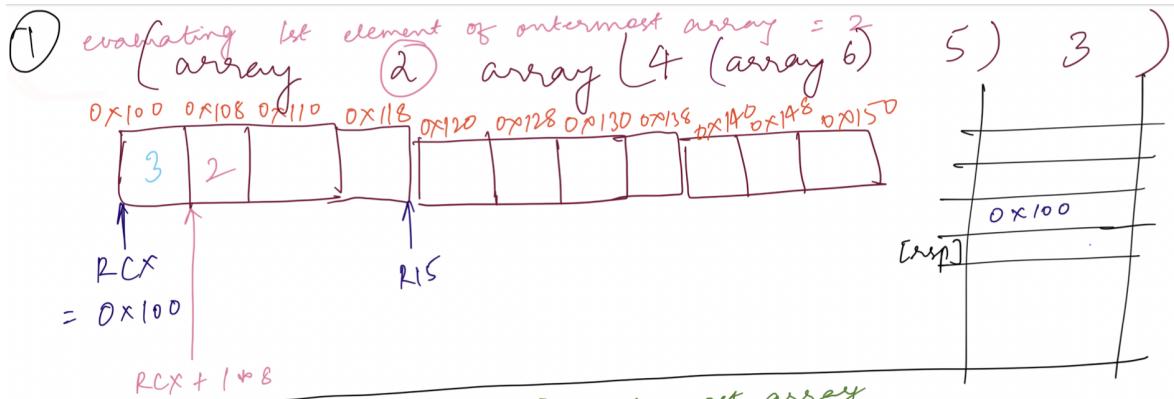
1. Calculate array size
2. Store R15 on stack as base location address
3. Increment R15 with $(1 + \text{num of elements offset})$
4. Store size in base location address stored on stack
5. For each index element of array:
 - a. Call `compile_expr` with $si = si + 2 + \text{index of element} - 1$ (index 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



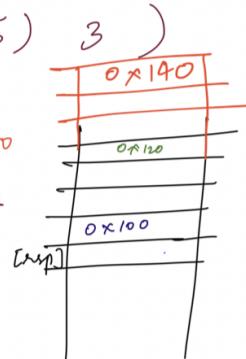
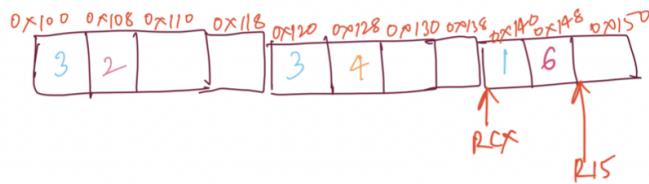
(array 2 array 4 (array 6) 5) 3)





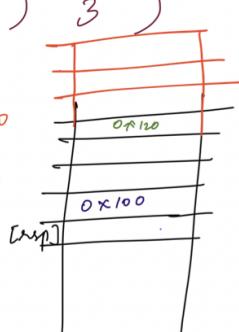
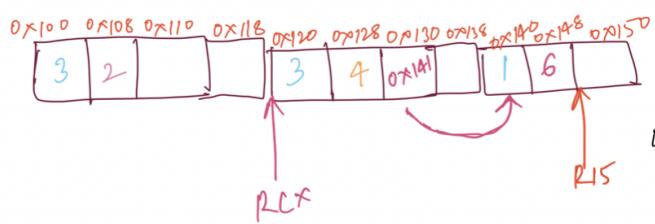
⑤ Evaluate 6

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



⑥ Return from processing (array 6)

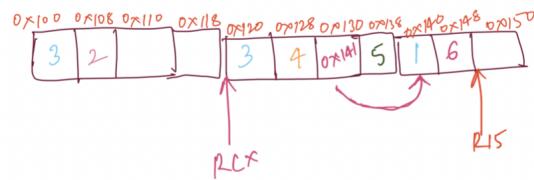
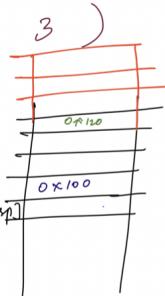
(array 2 array [4 (array 6) 5) 3)



⑦

Evaluate 5

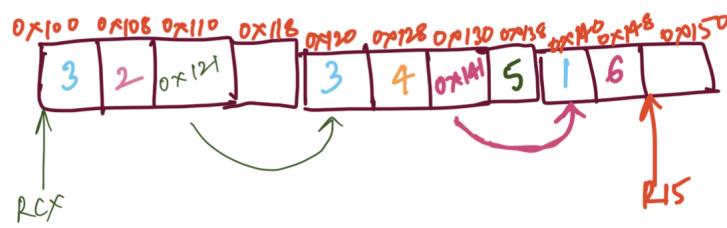
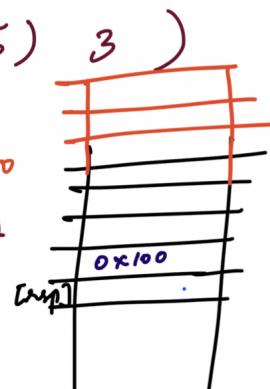
(array 2 array 4 (array 6) 5)



⑧

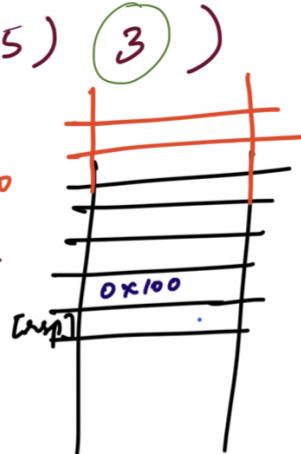
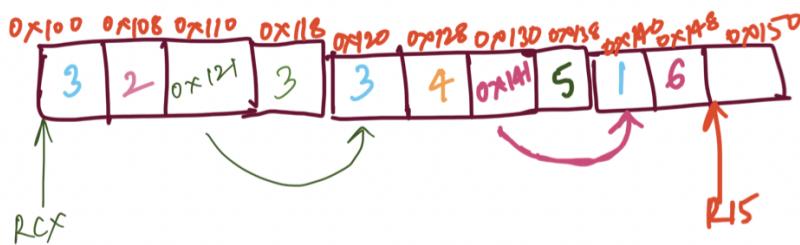
Return after processing 5

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

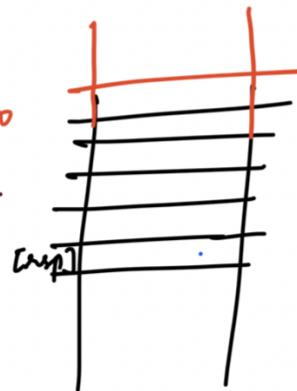
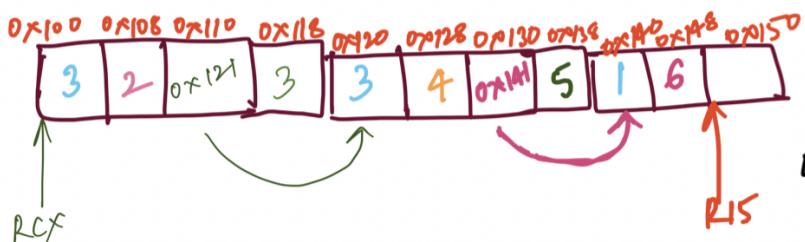


⑨ Evaluate 3

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



⑩ Return after evaluating 3:



RAX = 0x101

R15 = 0x150

The values from the nested arrays are written onto the stack, but not necessarily erased but they are not represented in the above diagrams to indicate the value is not needed anymore and can be overwritten. The array representation is storing only the address value as the base location. All the array elements are directly written onto the heap.

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)))
)
```

```
nasmd -f macho64 tests/simple_examples.s -o tests/simple_examples.o
ar rcs tests/libsimple_examples.a tests/simple_examples.o
rustup target add x86_64-apple-darwin
info: component 'rustc' for target 'x86_64-apple-darwin' is up to date
rustc -L tests/ -lour_code:simple_examples runtime/start.rs --target x86_64-apple-darwin -o tests/simple_examples.run
● aakritik@Akrritis-MacBook-Pro diamondback-aakriti-kedia % ./tests/simple_examples.run
(10, nil, 12)
(3, 4, 3, 5, 5)
(1, 2)
(1, (2, 3), (4, (5, 6)), (7, (8, 9)))
13
5
(4, (5, 6))
(2, 6, 44, 5)
(2, 6, 4, 55)
(1, (2, 3), 5, (7, (8, 9)))
(1, (2, 3), (4, 5, 6, 7, 8), (7, (8, 9)))
(1, (2, 3), (41, (42, 43), (44, 45, (46, (47, 48)))), (7, (8, 9)))
7
7
```

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 the 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 function that 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)  
)
```

```
● aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % make tests/error-tag.run  
cargo run -- tests/error-tag.snek tests/error-tag.s  
  Finished dev [unoptimized + debuginfo] target(s) in 0.04s  
    Running `target/debug/diamondback tests/error-tag.snek tests/error-tag.s`  
s_exp - ((block (getIndex (array 2 6 13 4 5) input)))  
parse_prog - Program { defs: [], main: Block([GetArrayIndex(Array([Number(4), Number(12), Number(26), Number(8), Number(10)]), Input)]) }  
nasm -f macho64 tests/error-tag.s -o tests/error-tag.o  
ar rcs tests/liberror-tag.a tests/error-tag.o  
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:error-tag runtime/start.rs --target x86_64-apple-darwin -o tests/error-tag.run  
④ aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % ./tests/error-tag.run  
invalid argument: type mismatch  
○ aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % █
```

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)
```

```
● aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % make tests/error-bounds.run  
cargo run -- tests/error-bounds.snek tests/error-bounds.s  
  Finished dev [unoptimized + debuginfo] target(s) in 0.12s  
    Running `target/debug/diamondback tests/error-bounds.snek tests/error-bounds.s`  
s_exp - ((getIndex (array 2 6 4 5) 10))  
parse_prog - Program { defs: [], main: GetArrayIndex(Array([Number(4), Number(12), Number(8), Number(10)]), Number(20)) }  
nasm -f macho64 tests/error-bounds.s -o tests/error-bounds.o  
ar rcs tests/liberror-bounds.a tests/error-bounds.o  
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:error-bounds runtime/start.rs --target x86_64-apple-darwin -o tests/error-bounds.run  
④ aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % ./tests/error-bounds.run  
index_out_of_bounds_error for array  
○ aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % █
```

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)

```
④ 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)))

)
```

```
nasm -f macho64 tests/points.s -o tests/points.o
ar rcs tests/libpoints.a tests/points.o
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:points runtime/start.rs --target x86_64-apple-darwin -o tests/points.run
● aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % ./tests/points.run
(5, 9)
(20, 30)
(5, 10)
(4, 8)
(11, 8)
(11, 8)
(11, 12)
(-1, 12)
(-1, 8)
(-1, -2)
(-1, -2)
```

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 be 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
                (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
                (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))
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))
  )
)
```

```
(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
```

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 the 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
- Lecture code shared on <https://github.com/ucsd-compilers-s23/lecture1/tree/egg-eater>
- 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>

Green-Snake features

Structural equality

Description

```
<op2> := + | - | * | < | > | >= | <= | = | checkTypeMatch | == (new!)
enum Op2 { Plus, Minus, Times, Equal, Greater, GreaterEqual, Less, LessEqual,
CheckTypeMatch, StructuralEquality}
```

`==` refers to structural equality. It is parsed into `Op2::StructuralEquality`, which means that it takes two expressions and each of it will be compiled in the exact way as expressions of other binary operators like `'+'`, `'-`, etc are compiled. This operator calls the `snek_structural_equality` runtime function if the parsing phase or the compiling phase does not throw any errors. The type match between the two variables is checked during the runtime function execution. Values are sent to the runtime function via RDI, and RSI registers and result (true or false in egg-eater language; true = 7, false=3) is returned back and can be read in assembly code via RAX registers. In the actual `check_structural_eq` function (internally called by `snek_structural_equality` function), values are checked for different available types -

1. If there is a type mismatch, it returns false
2. If numbers or booleans, then normal equality is performed and the result is returned
3. If arrays, if the
 - a. address representation along with the array tag is same for both the values, it returns true
 - b. If the sizes of the arrays differ, return false
 - c. Recursively check structural equality of each index element of both the arrays and return false if any comparison returns false
 - d. Everything matches, return true
 - e. To handle cyclic cases, a hashmap of the already compared egg-eater array representations is maintained (seen hashmap). If the pair is present in hashmap, return true
 - f. As soon as the comparison for the pairs being compared for structural equality is completed, they are removed from the seen hashmap and the result (either true or false) is returned

Code snippets

Parsing phase

```
[Sexp::Atom(S(op)), e1, e2] if op == "==" => Expr::BinOp(Op2::StructuralEquality,
Box::new(parse_expr(e1, &defs)), Box::new(parse_expr(e2, &defs))),
```

Compiling phase

```
Op2::StructuralEquality => {

    let index = if si % 2 == 0 { si + 2 } else { si + 3 };
    let offset = index * 8;

    let rdi_offset = 0;
    let rsi_offset = 8;

    instr_vector.push(Instr::IMov(Val::Reg(Reg::RBX),
Val::RegNegOffset(Reg::RSP, stack_offset)));

    instr_vector.push(Instr::ISub(Val::Reg(Reg::RSP),
Val::ImmInt(offset.into())));

    instr_vector.push(Instr::IMov(Val::RegPlusOffset(Reg::RSP,
rdi_offset), Val::Reg(Reg::RDI)));
    instr_vector.push(Instr::IMov(Val::RegPlusOffset(Reg::RSP,
rsi_offset), Val::Reg(Reg::RSI)));

    instr_vector.push(Instr::IMov(Val::Reg(Reg::RDI),
Val::Reg(Reg::RAX)));
    instr_vector.push(Instr::IMov(Val::Reg(Reg::RSI),
Val::Reg(Reg::RBX)));

instr_vector.push(Instr::ICall("snek_structural_equality".to_owned()));

    instr_vector.push(Instr::IMov(Val::Reg(Reg::RDI),
Val::RegPlusOffset(Reg::RSP, rdi_offset)));
    instr_vector.push(Instr::IMov(Val::Reg(Reg::RSI),
Val::RegPlusOffset(Reg::RSP, rsi_offset));

    instr_vector.push(Instr::IAdd(Val::Reg(Reg::RSP),
Val::ImmInt(offset.into())));

}
```

Runtime phase

```
#[export_name = "\x01snek_structural_equality"]
pub extern "C" fn x01snek_structural_equality(val1: i64, val2: i64) -> i64 {
```

```

let mut seen = Vec::<(i64, i64)>::new();
let struc_equality = check_structural_eq(val1, val2, &mut seen);
if struc_equality == true {
    return TRUE_INT
} else {
    return FALSE_INT
}
}

fn check_structural_eq(val1: i64, val2: i64, seen: &mut Vec<(i64, i64)>) -> bool {
    if ((val1 & 1) == 0 && (val2 & 1) == 0) || ((val1 & 3) == 3 && (val2 & 3) == 3) {
        val1 == val2
    } else if (val1 & 3) == 1 && (val2 & 3) == 1 { // arrays
        if val1 == val2 {
            return true;
        } else if seen.contains(&(val1, val2)) {
            return true;
        }
        else {
            seen.push((val1, val2));
            let addr_v1 = (val1 - 1) as *const i64;
            let addr_v2 = (val2 - 1) as *const i64;
            let size_v1 = unsafe { *addr_v1.offset(0) };
            let size_v2 = unsafe { *addr_v2.offset(0) };
            if size_v1 != size_v2 {
                seen.pop();
                return false;
            }
            for i in 0..size_v1 {
                let value_v1 = unsafe { *addr_v1.offset(i as isize + 1) };
                let value_v2 = unsafe { *addr_v2.offset(i as isize + 1) };
                if !check_structural_eq(value_v1, value_v2, seen) {
                    seen.pop();
                    return false;
                }
            }
            seen.pop();
            return true;
        } else { // type mismatch
            return false;
        }
    }
}

```

```
}
```

Simple program example

```
(let (
      (x (array 10 20 30))
      (y (array 40 45 x))
    )
  (block
    (print (== x y)))
))
```

Relevant assembly code

```
mov rax, [rsp + 8]
mov rbx, [rsp - 16]
sub rsp, 32
mov [rsp + 0], rdi
mov [rsp + 8], rsi
mov rdi, rax
mov rsi, rbx
call snek_structural_equality
mov rdi, [rsp + 0]
mov rsi, [rsp + 8]
add rsp, 32
sub rsp, 40
```

Handling cycles

For handling cycles in regards to structural equality and printing cyclic arrays, a hashmap is used. It keeps a track of which address values are encountered previously and as soon as an address processing is completed, it is popped out to ensure that it does not interfere with another independent array. It is handled in runtime phase of the compiler.

Printing cyclic arrays

For printing cyclic arrays, a seen hashmap of value type (i64) is maintained to account for the egg-eater representation of the array address being referenced in the current iteration.

The function checks if the current address is already present in the hashmap:

1. If found, it presents "(array <cyclic>)" and returns
2. Else, it appends the address (as egg-eater representation of array) to seen hashmap, starts "(" and iteratively goes through each of the array element and recursively calls the "snek_to_str" method on each element. If the element is number, boolean or nil, it is printed and not recursed anymore (base case). If the element is array, then again all the elements are recursed on.
3. Finally, after all the elements of the array are processed, it closes the output string with ")" indicating the end of the array output and pops the current element from the seen hashmap.
4. As the address will be looked for in the seen hashmap, so the function will ensure that an infinite loop is not triggered and cyclic values are handled gracefully.

Relevant runtime phase rust code

```
#[export_name = "\x01snek_print"]
pub extern "C" fn snek_print(val: i64) -> i64 {
    let mut seen = Vec::<i64>::new();
    println!("{}", snek_to_str(val, &mut seen));
    val
}
```

```
fn snek_to_str(val: i64, seen : &mut Vec<i64>) -> String {
    if val == TRUE_INT { return format!("true"); }
    else if val == FALSE_INT { return format!("false"); }
    else if val % 2 == 0 {
        return format!("{}", (val >> 1));
    } else if val == NIL_INT {
        return format!("nil");
    } else if val & 1 == 1 { // indicates array
        if seen.contains(&val) { return "(array <cyclic>)".to_string() }
    }
}
```

```

seen.push(val);
let addr = (val - 1) as *const i64;
let mut array_str = String::new();
array_str.push_str("(");
let size = unsafe { *addr };
for i in 0..size {
    let value = unsafe { *addr.offset(i as isize + 1) };
    array_str.push_str(&snek_to_str(value, seen).to_string());
    if i != size - 1 {
        array_str.push_str(", ".to_string());
    }
}
array_str.push_str(")");
seen.pop();
return format!("{}{}", array_str);
}
else {
    return format!("Unknown value: {}", val);
}
}

```

Simple example program

```
(let ((x (array 10 20)))
(block
  (setIndex x 1 x)
  (setIndex x 2 x)
  (print x)
))
```

Relevant assembly code

```

, Compiling
mov rax, [rsp + 0]
sub rsp, 24
mov [rsp + 0], rdi
mov rdi, rax
call snek_print
mov rdi, [rsp + 0]
add rsp, 24
add rsp, 32

```

Structural equality cycles

As explained in the Structural equality section above.

Required tests code and output

input/equal.snek

Code

```
(block
  (let ((x (array 10 20 30)))
    (block
      (print (= x x))
      (print (== x x))
      (print (= x (array 10 20 30)))
      (print (== x (array 10 20 30)))
    ))
  (let (
    (x (array 10 20 (array 10 20)))
    (y (array 10 20 (array 10 20)))
    (z1 (array 20 10 (array 10 20)))
    (z2 (array 10 20 10 20)))
  )
  (block
    (print (= x x))
    (print (= x y))
    (print (= x z1))
    (print (= x z2))
    (print (= x (array x)))

    (print (== x x))
    (print (== x y))
    (print (== x z1))
    (print (== x z2))
    (print (== x (array x)))
  )))
)
```

Output

```

nasm -f macho64 tests/structural_reference_equality.s -o tests/structural_reference_equality.o
ar rcs tests/libstructural_reference_equality.a tests/structural_reference_equality.o
# needed for macos only
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:structural_reference_equality runtime/start.rs --target x86_64-apple-darwin -o tests/structural_reference_equality.run
# needed for linux / windows
# rustc -L tests/ -lour_code:structural_reference_equality runtime/start.rs -o tests/structural_reference_equality.run
● aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % ./tests/structural_reference_equality.run
true
true
false
true
true
false
false
false
false
true
true
false
false
false
false
false
○ aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia %

```

Description

1. (print (= x x)) - Reference equality for a simple non-cyclic array with itself
2. (print (== x x)) - Structural equality for a simple non-cyclic array with itself
3. (print (= x (array 10 20 30))) - Reference equality of x with same-valued non-cyclic array
4. (print (== x (array 10 20 30))) - Structural equality of x with same valued non-cyclic array
5. (print (= x x)) - Reference equality of nested non-cyclic array with itself
6. (print (= x y)) - Reference equality of x with same-valued nested non-cyclic array
7. (print (= x z1)) - Reference equality of x with different-valued nested non-cyclic array
8. (print (= x z2)) - Reference equality of x with different-valued non-nested non-cyclic array
9. (print (= x (array x))) - Reference equality of x with nested array of itself
10. (print (== x x)) - Structural equality for a nested non-cyclic array with itself
11. (print (== x y)) - Structural equality of x with same-valued nested non-cyclic array
12. (print (== x z1)) - Structural equality of x with different-valued nested non-cyclic array
13. (print (== x z2)) - Structural equality of x with different-valued non-nested non-cyclic array
14. (print (== x (array x))) - Structural equality of x with nested array of itself

input/cycle-print1.snek

Code

```

(let ((x (array 10 20)))
  (block
    (setIndex x 1 x)
    (setIndex x 2 x)
    (print x)
  ))

```

Output

```

● aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % make tests/cycle_print1.run
make: `tests/cycle_print1.run' is up to date.
● aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % ./tests/cycle_print1.run
((array <cyclic>), (array <cyclic>))
((array <cyclic>), (array <cyclic>))
○ aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia %

```

Description

This code creates an array, sets the values at indices 1 and 2 to the array itself to create cycles, and then prints the array.

input/cycle-print2.snek

Code

```
(block
  (let (
    (x (array 10 20))
    (y (array 30)))
  )
  (block
    (setIndex x 1 y)
    (setIndex y 1 x)
    (print x)
    (print y)
  ))
  (let (
    (x (array 10 20 30))
    (y (array 40 45 x)))
  )
  (block
    (setIndex x 1 y)
    (print x)
    (print y)
  ))
)
```

Output

```
" Needed for Linux / Windows
# rustc -L tests/ -lour_code:cycle_print2 runtime/start.rs -o tests/cycle_print2.run
● aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % ./tests/cycle_print2.run
((array <cyclic>), 20)
((array <cyclic>), 20)
((40, 45, (array <cyclic>)), 20, 30)
(40, 45, ((array <cyclic>), 20, 30))
(40, 45, ((array <cyclic>), 20, 30))
○ aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % 
```

Description

This code creates 2 arrays x, and y with different values and interlinks them to create cycles. Then they are printed to demonstrate that printing handles cycles. The first block creates a cycle with an even number of elements and the second block statement creates a cycle with an odd number of elements.

input/cycle-print3.snek

Code

```
(let (
      (x (array 10 20))
      (y (array 20 30))
      (z (array 30 40))
    )
  (block
    (setIndex x 1 z)
    (setIndex x 2 y)
    (setIndex y 2 x)
    (setIndex z 1 x)
    (print x)
    (print y)
    (print z)
  ))
)
```

Output

```
# rustc -L tests/ -lour_code:cycle_print3 runtime/start.rs -o tests/cycle_print3.run
● aakritik@aakritis-MacBook-Pro:~/diamondback-aakriti-kedia% ./tests/cycle_print3.run
(((array <cyclic>), 40), (20, (array <cyclic>)))
(20, (((array <cyclic>), 40), (array <cyclic>)))
(((array <cyclic>), (20, (array <cyclic>))), 40)
(((array <cyclic>), (20, (array <cyclic>))), 40)
```

Description

This code exploits a cycle with 3 variables. It intermingles these 3 to form a cycle and checks print to see if this variety of cycles can be handled.

input/cycle-equal1.snek

Code

```
(let (
      (x (array 10 20))
      (y (array 10 20))
    )
  (block
    (setIndex x 1 x)
    (setIndex x 2 x)

    (setIndex y 1 y)
    (setIndex y 2 y)
  ))
```

```

(print x)
(print y)
(print (== x x))
(print (== y y))
(print (== x y))

(setIndex y 1 x)
(print (== x y))
(setIndex y 2 x)
(print (== x y))
))

```

Output

```

● aakritik@aakritis-MacBook-Pro diamondback-aakriti-kedia % ./tests/cycle_equal1.run
((array <cyclic>), (array <cyclic>))
((array <cyclic>), (array <cyclic>))
true
true
true
true
true
true

```

Description

It initializes two independent arrays x, and y. Then each element of the array is referenced to the array itself to form cycles. This is them compared for structural equality. Then, x and y are also interconnected to form a cycle and then comparerd for structural equality. Finally, both elements of y are made to point to x reference and x and y are for structural equality.

input/cycle-equal2.snek

Code

```

(block
  (let (
    (x (array 10 20))
    (y (array 30 40))

    (x1 (array y 20))
    (y1 (array 30 x)))

    (x2 (array y1 20))
    (y2 (array 30 x1)))
  )
  (block
    (setIndex x 1 y)
    (setIndex y 2 x)
    (print x)
  )
)
```

```

(print y)

(print x1)
(print y1)

(print x2)
(print y2)

(print (= x x1))
(print (= x x2))
(print (= x1 x2))

(print (= y y1))
(print (= y y2))
(print (= y1 y2))

(print (== x x1))
(print (== x x2))
(print (== x1 x2))

(print (== y y1))
(print (== y y2))
(print (== y1 y2))

)))
)

```

Output

```

# needed for linux / windows
# rustc -L tests/ -lour_code:cycle_equal2 runtime/start.rs -o tests/cycle_equal2.run
● aakritik@aakritis-MacBook-Pro diamondback-aakriti-kedia % ./tests/cycle_equal2.run
((30, (array <cyclic>)), 20)
(30, ((array <cyclic>), 20))
((30, ((array <cyclic>), 20)), 20)
(30, ((30, (array <cyclic>)), 20))
((30, ((30, (array <cyclic>)), 20)), 20)
(30, ((30, ((array <cyclic>), 20)), 20))
false
false
false
false
false
false
true
true
true
true
true
true
true
true
aakritik@aakritis-MacBook-Pro diamondback-aakriti-kedia %

```

Description

This code initializes x and y as arrays of 2 elements each. Then x1, y1 are initialized referencing y and x respectively. Then, x2 and y2 are initialized referencing y1 and x1 respectively. With setIndex, x and y are assigned the same values as x1 and y1, i.e. x references y in the first element, and y references x in the second element. This creates a chain-like relationship among the variables, forming a cyclic structure. Then various combinations of structural and reference equality are tested.

input/cycle-equal3.snek

Code

```
(let (
  (x (array 10 20))
  (y (array 20 30))
  (z (array 30 40))
  (y1 (array 20 30)))
)
(block
  (setIndex x 1 z)
  (setIndex x 2 y)
  (setIndex y 2 x)
  (setIndex y1 2 x)
  (setIndex z 1 x)

  (print x)
  (print y)
  (print z)
  (print y1)

  (print (== (array (getIndex y 1) x) y))

  (print (== (array (getIndex y 1) (getIndex y 2)) y))

  (print (== y y1))

  (print (== x z))

  (print (== (array (getIndex y 1) x) y))

  (print (== (array (getIndex y 1) y) y))

))
```

Output

```
# rustc -L tests/ -lour_code:cycle_equal3 runtime/start.rs -o tests/cycle_equal3.run
● aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia % ./tests/cycle_equal3.run
(((array <cyclic>), 40), (20, (array <cyclic>)))
(20, (((array <cyclic>), 40), (array <cyclic>)))
(((array <cyclic>), (20, (array <cyclic>))), 40)
(20, (((array <cyclic>), 40), (20, (array <cyclic>))))
true
true
true
false
true
false
false
○ aakritik@Aakritis-MacBook-Pro diamondback-aakriti-kedia %
```

Description

In this code, $x = (z, y)$, y and y_1 have the same values (20, x) and $z = (x, 40)$. Cycles exist between x and y , x and y_1 , x and z . Then same and different valued arrays are made on the fly and tested for structural equality.

Other features

Reference equality

```
<op2> := + | - | * | < | > | >= | <= | = | checkTypeMatch | ==
enum Op2 { Plus, Minus, Times, Equal, Greater, GreaterEqual, Less, LessEqual,
CheckTypeMatch, StructuralEquality}
```

As reference equality for arrays is supposed to compare the actual address value of the array only and not the contents stored in the heap, binary operator = is capable of handling this feature and hence, no additional work was required to get the reference equality feature working.

It checks for type match between expr1 and expr2 and throws errors if there is a type mismatch. If the types are same, it compares the values. For arrays specifically, it compares the value returned from the compilation of expr1 and expr2, which is the egg-eater representation of the array.

Code

Parsing phase

```
[Sexp::Atom(S(op)), e1, e2] if op == "=" => Expr::BinOp(Op2::Equal,
Box::new(parse_expr(e1, &defs)), Box::new(parse_expr(e2, &defs))),
```

Compiling phase

```
Op2::Equal => {
    let v1 = Val::Reg(Reg::RAX);
    let v2 = Val::RegNegOffset(Reg::RSP, stack_offset);
    let match_end_label = new_label(l, "equal_type_match_end");
    instr_vector.extend(error_if_type_mismatch(v1.clone(), v2.clone(),
&match_end_label));
    instr_vector.push(Instr::ILabel(match_end_label, Vec::new()));
    instr_vector.push(Instr::ICmp(v1.clone(), v2.clone()));
    instr_vector.push(Instr::IMov(Val::Reg(Reg::RBX),
Val::ImmInt(TRUE_INT)));
    instr_vector.push(Instr::IMov(Val::Reg(Reg::RAX),
Val::ImmInt(FALSE_INT)));
    instr_vector.push(Instr::ICmove(Val::Reg(Reg::RAX),
Val::Reg(Reg::RBX)));
}
```

```
fn error_if_type_mismatch(v1: Val, v2: Val, match_end_label: &String) -> Vec<Instr> {
    let mut instr_vect = Vec::new();
    instr_vect.push(Instr::IComment("check_type_match_result".to_string()));
```

```

instr_vect.push(Instr::IMov(Val::Reg(Reg::RBX), v1.clone()));
instr_vect.push(Instr::IXor(Val::Reg(Reg::RBX), v2.clone()));
instr_vect.push(Instr::ITest(Val::Reg(Reg::RBX), Val::ImmInt(3)));
instr_vect.push(Instr::IJe(match_end_label.to_string())); // jz -> all matched

instr_vect.push(Instr::IMov(Val::Reg(Reg::RBX), v1));
instr_vect.push(Instr::IOr(Val::Reg(Reg::RBX), v2));
instr_vect.push(Instr::ITest(Val::Reg(Reg::RBX), Val::ImmInt(1)));
instr_vect.push(Instr::IJe(match_end_label.to_string())); // last digit is 0

instr_vect.push(Instr::IJmp("throw_error".to_string()));
instr_vect
}

```

setIndex

Redefined here. Was already implemented as part of PA6 and examples and tests of PA6 hold for PA8 as well. The examples of cycles and structural equality also use setIndex to demonstrate more robust tests.

<expr> := (setIndex <expr> <expr> <expr>)

Enum Expr {.... SetArrayIndex(Box<Expr>, Box<Expr>, Box<Expr>),}

(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.

Improvements to PA6

None. A high pass was achieved for PA6 with no feedback comments and hence no changes were required.

List of resources

Edstem posts

1. <https://edstem.org/us/courses/38748/discussion/3191257>
2. <https://edstem.org/us/courses/38748/discussion/3138761>