EGGEATER

Chirag Dasannacharya
A59016593

Heap

The heap consists of an array generated in rust, whose address is passed to the program. The first 8 bytes of the heap are reserved for the initial RSP, the next 8 bytes for remaining heap space. For now, the heap is set to grow downward, so an additional marker for heap end is not needed.

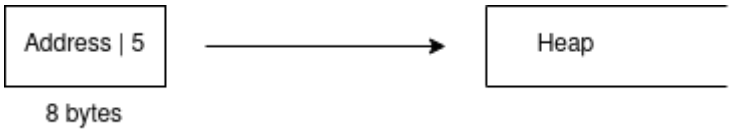
Types

The following new ‘datatypes’ are introduced -

- Pointer -

  An address to some data (at present, only on the heap). A pointer counts as an L-value. The type uses 8 bytes, fits in a register, and has the 0x4 ‘pointer’ bit set (a convention I use). In addition, the last bit, 0x1 is also set to differentiate it from a number. As a result, a pointer ends in 0b101, i.e. 5.

  This is acceptable, as all addresses are aligned to 8 bytes, leaving the last 3 bits free. These bits, however, must be (and are) stripped before any operation and added back before storing.



- Array -

  A fixed size collection of elements. Elements in an array can themselves be of different types. Each element is exactly 8 bytes long, for types that are larger the array always stores a pointer to the data.

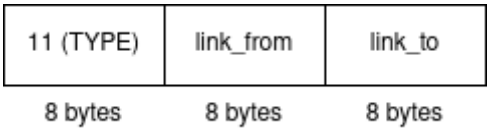
  In memory, the array is stored starting with a type byte, set to 0b10011, or 19. The set 0x2 bit signifies that the value is a type, 0x1 that the value is not numeric, and the unset 0x4 that the value is not a pointer. The 0b00011 type is reserved for ‘True’, and is not used.

  The type byte is followed by a size marker, containing the size in bytes of all the elements (not of the array itself, this is an additional 16 bytes due to the type and size values). This is followed by the elements, in order.

- Link -

  This is identical in practice to a pair, but is called a link due to its intended conceptual use - to hold a value and a pointer (this is not enforced). This can be used in a variety of ways, such as link lists, graph nodes (by filling the pointer with the address of an array of neighbor nodes), bst nodes, etc.

  Similar to an array, a pair is a fixed size object (24 bytes), when holding a larger object within it, a pointer is used. The first 8 bytes in the pair hold the type, 0b01011, or 11. Bit conventions are described above, in the explanation for arrays. The following 16 bytes hold the two values.



These structures can be used to simulate most, if not all, heap structures as required. A performance loss may be seen from excessive use of pointers, more specialized types can be introduced natively in addition to these as the need arises.

New Functions

Similar to print, new functions are introduced to use the above types -

- Fill (ptr, val) -> val - Fill an L-value (ptr, a pointer) with val. If val is larger than 8 bytes (e.g. an array), a pointer to the object is filled instead. Returns val. This function is similar to set!, except that set! takes an identifier. ptr, here, may be the result of an expression.

- Deref (ptr) -> val - Read 8 bytes at a location specified by pointer.

- Array (size) -> ptr - Allocate an array on heap with specified size, and return a pointer to it. There is no array initialization, values can be filled using the index and fill functions.

- Index (ptr, i) -> ptr - Get a pointer to array[i], where ptr points to array. Array indices vary from 0 to size - 1, negative indices are not used. This function cannot be used on a link.

- Link () -> ptr - Allocate a link on heap and return a pointer to it. There is no link initialization, values are filled using the fill function and link_from or link_to.

- Link_from (ptr) -> val - Get the value stored in the first field of link, where link is stored at ptr. This function does not work on arrays.

- Link_to (ptr) -> val - Get the value stored in the second field of link, where link is stored at ptr. This function does not work on arrays.

Libraries

The new functions are placed in 'libraries' - these are written in assembly, built separately, and linked at compile time (makefile steps shown in later section). A list of these follows -

- lib_error - Contains error handling labels for different errors and error codes (7 at present - failed setup, no more memory, type mismatch (generic), bad memory access, type mismatch (bool), type mismatch (int), overflow).

- lib_io - Contains a wrapper for snek_print.

- lib_heap - Contains fill, deref, as well as internal functions setup_heap (used to initialize the heap) and alloc (used to allocate memory on the heap).

- lib_array - Contains array, index.

- lib_link - Contains link, link_from, link_to.

- lib_memory_manager - contains an alternative, additional heap implementation using the mmap system call. Linux system calls use a slightly different call interface - r10 is used instead of rcx for the 4th argument.

These libraries are present under the root/lib folder, alongside the root/src and root/runtime folders.

Grammar

Libraries of functions are used to cover the heap related tasks required in eggeater. As a result, the grammar of the language is exactly the same as in Diamondback.

NEW - There are 7 predefined functions, listed and described in the new functions section. Introducing these as functions rather than syntactic structures introduces some small overhead, this can be eliminated if a function inlining optimization is added later.

Diamondback's grammar, identical to the one used, is as follows -

<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>*)
<op1> := add1 | sub1 | isnum | isbool | print
<op2> := + | - | * | < | > | >= | <= | =
<binding> := (<identifier> <expr>)

Heaps in Other Languages

C does not have an explicit heap, but rather allocates memory at each call from either an existing available region, or with an mmap system call. Multiple such mmap'd regions exist, these are grouped into 'arenas', which can be locked and used independently in a multi-threaded program. Pages can be unmapped when objects in it are freed, releasing physical memory back to the system. Java, however, does not allow explicit pointer references and dereferences, this allows for objects to be relocated at runtime. Separate regions are maintained for different regions.

The heap implementation in this project used to run the tests is along the lines of the latter - the heap is a continuous space of memory, rather than a collection of disjoint arenas. However, lib_memory_manager contains the option to use a mmap-based heap, with a number of disjoint allocations, similar to C. This has not been pursued further at present due to likely future issues regarding testing compatibility in the upcoming GC assignment. This method does not take heap size as an argument from runtime/start.rs, but runs out of memory only when the OS refuses to allocate further memory.

Building

The introduction of separate libraries means that some additional steps are required while building. The following make targets are set -

```
tests/%.run: tests/%.s runtime/start.rs lib/*.o
   nasm -f $(ARCH) tests/$*.s -o tests/intermediate_$*.o
   ld -r -o tests/$*.o tests/intermediate_$*.o lib/*.o
   ar rcs tests/lib$*.a tests/$*.o
   rustc -L lib/ -L tests/ -lour_code:$* runtime/start.rs -o
tests/$*.run
```

```
lib/%.a: lib/%.s
   nasm -f $(ARCH) lib/$*.s -o lib/$*.o
   ar rcs lib/$*.a lib/$*.o

glib: lib/lib_memory_manager.a lib/lib_error.a lib/lib_heap.a
lib/lib_link.a lib/lib_array.a lib/lib_io.a
```

When a library is updated or the code is run for the first time, it is necessary to run make glib (global libraries). This updates (or creates) the associated object files in the lib folder.

A regular snek build has some additional steps - the .s file is first converted to a .o file, then partially linked with the global libraries (with ld -r). A full link is not possible at this stage, as some functions, namely those in runtime/start.rs, have not yet been generated. The resulting intermediate is archived and passed to rustc to resolve the remaining symbols (such as snek_print).

Printing

While printing, snek_print dereferences pointers and prints the internal data in brackets. For arrays, an annotation is added specifying size and that it is an array. Similarly, links are annotated to mention that a link structure is being printed.

References

https://nullprogram.com/blog/2015/05/15/ - for how to trigger a linux system call
https://sourceware.org/glibc/wiki/MallocInternals - for how C implements malloc
https://wiki.openjdk.org/display/shenandoah/Main#Main-ImplementationOverview - for the implementation of Java's Shenandoah GC
https://stackoverflow.com/questions/29391965/what-is-partial-linking-in-gnu-linker - for working with partial linking
https://app.diagrams.net/ - to generate the drawings in this document.

Tests

A description is provided for a subset of the tests in the tests/input subfolder -

Input_simple_examples_1.snek - Initialize an array

(array 5)

Output -
(array - size 5 - 0 0 0 0 0)

Explanation - Memory set to 0 on obtaining from rust.

Input_simple_examples_2.snek - Print an array

```
(let ((x (array 5)) (i 0)) (print x))
```

Output -
(array - size 5 - 0 0 0 0 0)
(array - size 5 - 0 0 0 0 0)

Explanation - the same result is shown twice, once from the print and once from the return. A print returns its input, even if it is an array, link, etc.

Input_simple_examples_3.snek - Initialize a link

```
(link)
```

Output -
(link - 0 0)

Explanation - A link, with from and to both set to 0. There is no enforcement of the guideline that the latter value should be a pointer.

Input_simple_examples_4.snek - Access an array element

```
(let ((x (array 5))) (deref (index x 0)))
```

Output -
0

Explanation - (index x 0) returns the address of x[0], adding deref makes this *(&x[0]) (in C terms). The value is presently unset and 0 by default.

Input_simple_examples_5.snek - Fill an array

```
(let ((x (array 5)) (i 0)) (
        loop (if (= i 5) (break x) ( block
                (fill (index x i) i)
                (set! i (+ i 1))
        ) )
) )
```

Output -
(array - size 5 - 0 1 2 3 4)

Explanation - (index x 0) returns the address of x[0], the fill function is equivalent to *(ptr) = i, or *(&(x[i])) = i.

Input_simple_examples_6.snek - Access and set an array

```
(let ((x (array 5)) (i 1)) ( block
        (fill (index x 0) 1)
```

```
        ( loop (if (= i 5) (break x) ( block
                (fill (index x i) (+ 1 (deref (index x (- i 1)))))
                (set! i (+ i 1))
        ) ) )
) )
```

Output -
(array - size 5 - 1 2 3 4 5)

Explanation - The first element is set to 1. Each succeeding element is set to one more than the preceding one.

Input_simple_examples_7.snek - Create and fill a 2D array

```
(let ((x (array 5)) (i 0) (count 1)) (
        loop (if (= i 5) (break x) ( block
                (let ((y (array 5)))
                        (fill (index x i) y)
                )
                (let ((j 0)) (
                        loop (if (= j 5) (break j) ( block
                                (fill (index (deref (index x i)) j) count)
                                (set! count (+ count 1))
                                (set! j (+ j 1))
                        ) )
                ) )
                (set! i (+ i 1))
        ) )
) )
```

Output -
(array - size 5 - (array - size 5 - 1 2 3 4 5) (array - size 5 - 6 7 8 9 10) (array - size 5 - 11 12 13 14 15) (array - size 5 - 16 17 18 19 20) (array - size 5 - 21 22 23 24 25))

Explanation - Initialize an array, fill each element with an array (forming a 2d array of size 5*5). Loop through the 2d array, fill each location with an incrementing count.

Input_simple_examples_8.snek - Create and populate a link

```
(let ((x (link))) (block
        (fill (link_from x) 1)
        (fill (link_to x) 2)
        x
))
```

Output -
(link - 1 2)

Explanation - Initialize a link, fill its fields with 1 and 2. Return the link.

Input_simple_examples_9.snek - Create, populate and access a link

```
(let ((x (link))) (block
        (fill (link_from x) 1)
        (fill (link_to x) 2)
        (deref (link_from x))
))
```

Output -
1

Explanation - Initialize a link, fill its fields with 1 and 2. Return the first (from) field of the link.

Input_simple_examples_10.snek - Create, populate and access a link

```
(let ((x (link))) (block
        (fill (link_from x) 1)
        (fill (link_to x) 2)
        (deref (link_to x))
))
```

Output -
2

Explanation - Initialize a link, fill its fields with 1 and 2. Return the second (to) field of the link.

Input_error_tag_1.snek - Attempt to index a link

```
(let ((x (link))) (block
        (fill (link_from x) 1)
        (fill (link_to x) 2)
        (deref (index x 0))
))
```

Output -
Error - 3 - Type mismatch
(Runtime error)

Explanation - Try to apply the index function on a link pointer. This fails as index can only be applied to array pointers.

Input_error_tag_2.snek - Attempt to use link_from in an array

```
(let ((x (array 5))) (block
        (link_from x)
))
```

Output -
Error - 3 - Type mismatch
(Runtime error)

Explanation - Link_from and link_to can only be applied to link pointers, and not to arrays or other types.

Input_error_bounds_1.snek - Attempt a negative index

(let ((x (array 5))) (index x -1))

Output -
Error - 4 - Bad memory access
(Runtime error)

Explanation - Unlike Python, this compiler does not support negative indexing

Input_error_bounds_2.snek - Attempt an excessively large index

(let ((x (array 5))) (index x 5))

Output -
Error - 4 - Bad memory access
(Runtime error)

Explanation - A standard index out of bounds exception. This array's indices go from 0 to 4 inclusive.

Input_error_bounds_3.snek - Attempt to dereference a non-pointer object

(deref 1)

Output -
Error - 4 - Bad memory access
(Runtime error)

Explanation - 1 is an integer, and not a pointer. Unlike C, pointers and numbers cannot be treated as identical.

Input_error3_1.snek - Attempt to allocate excessive memory

(array 50000000)

Output -
Error - 2 - Ran out of memory
(Runtime error)

Explanation - Here, the heap size was set to 8MB. The array of specified size could not be allocated.

Input_points_1.snek - Define a vector add function, use it on two points.

```
(fun (add_points p1 p2) (
        let ((p_new (link))) ( block
                (fill (link_from p_new) (+ (deref (link_from p1)) (deref (link_from p2))))
                (fill (link_to p_new) (+ (deref (link_to p1)) (deref (link_to p2))))
                p_new
        )
) )
(let ((p1 (link)) (p2 (link))) (block
        (fill (link_from p1) 1)
        (fill (link_to p1) 1)
        (fill (link_from p2) 1)
        (fill (link_to p2) 1)
        (add_points p1 p2)
))
```

Output -
(link - 2 2)

Explanation - A function, add_points, is defined to add two pairs. This is applied on (1, 1) + (1, 1)

Input_points_3.snek - Define a vector add function, use it on two points. Use set! on a link.

```
(fun (add_points p1 p2) (
        let ((p_new (link))) ( block
                (fill (link_from p_new) (+ (deref (link_from p1)) (deref (link_from p2))))
                (fill (link_to p_new) (+ (deref (link_to p1)) (deref (link_to p2))))
                p_new
        )
) )
(let ((p1 (link)) (p2 (link)) (p3 (link))) (block
        (fill (link_from p1) 1)
        (fill (link_to p1) 2)
        (fill (link_from p2) 3)
        (fill (link_to p2) 4)
        (set! p3 (add_points p1 p2))
        (print p3)
        (deref (link_from p3))
))
```

Output -
(link - 4 6)
4

Explanation - A function, add_points, is defined to add two pairs. This is applied on (1, 2) + (3, 4). The resulting point is saved in a new point, p3. P3 is printed, and its first field returned.

Input_bst_1.snek - Define functions to create bst nodes, create bst, check membership in bst

```
(fun (make_node val) (block
	(let ((node (link)) (children (link))) ( block
		(fill (link_from children) 0)
		(fill (link_to children) 0)
		(fill (link_from node) val)
		(fill (link_to node) children)
		node
	) )
) )
(fun (get_left node) ( let ((children (deref (link_to node))))
	(deref (link_from children))
) )
(fun (get_right node) ( let ((children (deref (link_to node))))
	(deref (link_to children))
) )
(fun (get_val node) (deref (link_from node)))
(fun (set_val node val) (fill (link_from node) val))
(fun (set_left node val) ( let ((children (deref (link_to node))))
	(fill (link_from children) val)
) )
(fun (set_right node val) ( let ((children (deref (link_to node))))
	(fill (link_to children) val)
) )
(fun (insert tree node) (block
	(if (= (get_val node) (get_val tree)) 0 (
	if (< (get_val node) (get_val tree))
	(if (isnum (get_left tree)) (set_left tree node) (insert (get_left tree) node))
	(if (isnum (get_right tree)) (set_right tree node) (insert (get_right tree) node))
	))
))
(fun (check_present tree val) (block
	(if (isnum tree) false (
	if (= val (get_val tree)) true (
	if (< val (get_val tree))
		(check_present (get_left tree) val)
		(check_present (get_right tree) val)
	)
	) )
))
(fun (make_bst node_list size) (
```

```
        let ((tree (deref (index node_list 0)))) (block
                (let ((i 1)) (loop (if (= i size) (break tree) (block
                        (insert tree (deref (index node_list i)))
                        (set! i (+ i 1))
                ))))
                tree
        )
))
(let ((x (array 5))) (block
        (let ((i 0)) (loop (if (= i 5) (break i) (block (fill (index x i) (make_node i)) (set! i (+ i
1))))))
        (let ((tree (make_bst x 5))) (block
        (check_present tree -1)
        ))
) )
```

Output -
false

Explanation - Nodes are made as a link from a value to a pair of children (itself a link, with from as left and to as right). The numeric 0 is treated as a null pointer. A number of helpers are defined -

make_node(val) creates a node with value val and children set to null
get_left(node) returns the left subchild of a node
get_right(node) returns the right subchild of a node
get_val(node) returns the value at a node
set_val(node, val), set_left(node, val), set_right(node, val) - set the value, left and right subchild respectively of node as val.
insert(tree, node) inserts a node into the tree
check_present(tree, val) sees if there is a node in the tree with value val, returns bool
make_bst(node_list, size) makes and returns a tree from an array of nodes of size specified

An array of nodes is created in main, with values 0, 1, 2, 3 and 4. A tree is created with this array. We check if -1 is present in the tree, this returns false.

Input_bst_2.snek - Same as above but with a more interesting tree, and tree printing.

```
(fun (make_node val) (block
        (let ((node (link)) (children (link))) ( block
                (fill (link_from children) 0)
                (fill (link_to children) 0)
                (fill (link_from node) val)
                (fill (link_to node) children)
                node
        ) )
) )
(fun (get_left node) ( let ((children (deref (link_to node))))
```

```
        (deref (link_from children))
) )
(fun (get_right node) ( let ((children (deref (link_to node))))
        (deref (link_to children))
) )
(fun (get_val node) (deref (link_from node)))
(fun (set_val node val) (fill (link_from node) val))
(fun (set_left node val) ( let ((children (deref (link_to node))))
        (fill (link_from children) val)
) )
(fun (set_right node val) ( let ((children (deref (link_to node))))
        (fill (link_to children) val)
) )
(fun (insert tree node) (block
        (if (= (get_val node) (get_val tree)) 0 (
        if (< (get_val node) (get_val tree))
        (if (isnum (get_left tree)) (set_left tree node) (insert (get_left tree) node))
        (if (isnum (get_right tree)) (set_right tree node) (insert (get_right tree) node))
        ))
))
(fun (check_present tree val) (block
        (if (isnum tree) false (
        if (= val (get_val tree)) true (
        if (< val (get_val tree))
                (check_present (get_left tree) val)
                (check_present (get_right tree) val)
        )
        ) )
))
(fun (make_bst node_list size) (
        let ((tree (deref (index node_list 0)))) (block
                (let ((i 1)) (loop (if (= i size) (break tree) (block
                        (insert tree (deref (index node_list i)))
                        (set! i (+ i 1))
                ))))
                tree
        )
))
(let ((x (array 5))) (block
        (fill (index x 0) (make_node 4))
        (fill (index x 1) (make_node 5))
        (fill (index x 2) (make_node 2))
        (fill (index x 3) (make_node 1))
        (fill (index x 4) (make_node 3))
        (let ((tree (make_bst x 5))) (block
                (print tree)
                (print (check_present tree -1))
                (print (check_present tree 2))
```

))
))

Output -
(link - 4 (link - (link - 2 (link - (link - 1 (link - 0 0)) (link - 3 (link - 0 0)))) (link - 5 (link - 0 0))))
false
true
true

Explanation - An array of nodes is created with values [4,5,2,1,3]. A tree is created inserting these in order. The tree is printed. -1 is checked in the tree, found not present and false printed. Similarly, true is printed for 2, the same is returned.