Exercise 07

Task 1: Reference Counting

One of the shortcomings of Reference counting that was presented in the lecture is the inability to handle cyclic references.

1. Can a cyclic reference scenario occur in the SRCFLAE language? If so, give an example. If not, explain why it is not possible.

Consider:

```
Let('x, NewBox(10),
Let('y, NewBox('x),
Seqn(
  SetBox('x, 'y),
  OpenBox(OpenBox(OpenBox(OpenBox('x)))))))
```

Or

```
Let('x, NewBox(10),
Seqn(
  SetBox('x, 'x),
  OpenBox(OpenBox(OpenBox(OpenBox('x)))))))
```

2. Reference counting or variations are still used in modern programming languages. Can you think of ways to avoid or handle cyclic references in combination with reference counting? Can you imagine that reference counting and garbage collection could combined to have the advantages of both concepts?

Ideas:

- Disallow cycles by dynamic checks. Requires traversing all dependencies.
- One could use reference counting, but still do GC occasionally to detect cyclic structures. But while this has the advantages, that most memory is immidiately freed, and cyclic structures are detected, it also combines the overhead of both approaches.

3. Take a look at the following interpreter cases implemented for Reference counting. Explain in your own words why each call of `deref` and `enref` is required. Think about the possible consequences if some of the statements would be removed. Try to find examples that would produce memory leaks or lookup failures.

```
case SetBox(boxExpr, valueExpr) =>
  val (boxV, s1) = interp(boxExpr, stack, store)
  val (value, s2) = interp(valueExpr, stack, s1)
  boxV match {
    case Box(loc) =>
```

```
          // deref the old value in the box,
          // otherwise would leak memory
          val s4 = deref(loc, s2)
          // enref the new value we put in the box,
          // otherwise would fail lookup
          enref(value)
          (value, s4.update(loc, value))
        case _ => sys.error("can only set to boxes, but got: " + boxV)
      }


    case Let(boundId, namedExpr, boundBody) =>
      val (namedVal, s1) = interp(namedExpr, stack, store)
      val (newLoc, s2) = s1.malloc(stack, namedVal)
      val (v, s3) = interp(boundBody,
          stack.head + (boundId -> newLoc) :: stack.tail, s2)
      // deref the location when the Id
      // goes out of scope, would leak memory
      val s4 = deref(newLoc, s3)
      (v, s4)
```

Task 2: Non-Moving GC vs Moving GC vs Copying GC

Consider the following program:

```
val res = {
  val x1 = Box(10)
  val x2 = Box(x1)
  val x3 = Box(11)
  val x4 = Box(x3)
  val x5 = Box(x1)
  val x6 = Box(x4)
  val x7 = Box(x6)
  Box(x6)
}

system.gc()
```

It computes a value for `res` using some local variables `x1`-`x7`, and then calls the garbage collector of our system. Using SRCFLAE, the definition of `res` can be written as

```
Let('res, Let('x1, NewBox(10),
        Let('x2, NewBox('x1),
          Let('x3, NewBox(11),
            Let('x4, NewBox('x3),
```

```
            Let('x5, NewBox('x1),
              Let('x6, NewBox('x4),
                Let('x7, NewBox('x6),
                  NewBox('x6)))))))),
    ...)
```

Your task: Simulate the garbage collector run by modeling the environment and store before and after garbage collection. Do this for three different GC strategies:

Solution:

Before we call the garbage collector, we have the following environment and store:

Environment: res $\rightarrow$ 15

(As a reference, the environment for the local variables is x1 $\rightarrow$ 1, x2 $\rightarrow$ 3, x3 $\rightarrow$ 5, x4 $\rightarrow$ 7, x5 $\rightarrow$ 9, x6 $\rightarrow$ 11, x7 $\rightarrow$ 13)

Store:

Loc	Data
0	Num(10)
1	Box(0)
2	1
3	Box(2)
4	Num(11)
5	Box(4)
6	5
7	Box(6)
8	1
9	Box(8)
10	7
11	Box(10)
12	11
13	Box(12)
14	11
15	Box(14)

Note: The numbers in the Data column are store locations and are referencing the same Scala object the given store location, e.g. store location 2 references the same Scala object as store location 1 (Box(0)).

Now, we call the garbage collector with the environment.

1. non-moving GC, as we implemented in the lecture

Environment: res $\rightarrow$ 15

Store:

Loc	Data
0	-
1	-
2	-
3	-
4	Num(11)
5	Box(4)
6	5
7	Box(6)
8	-
9	-
10	7
11	Box(10)
12	11
13	-
14	11
15	Box(14)

2. moving GC, which defragments the heap by collecting all non-released objects in one part of the heap

Environment: res $\rightarrow$ 8

Store:

Loc	Data
0	Num(11)
1	Box(0)
2	1
3	Box(2)
4	3
5	Box(4)
6	5
7	5
8	Box(7)
9	-
10	-
11	-
12	-
13	-
14	-
15	-

3. copying GC, which splits the heap in two spaces: From-Space and To-Space. Modifications to the store are done on the From-Space. When garbage collection is performed, then all live references are copied from the From-Space to the To-Space. Then the spaces are flipped, i.e. the former To-Space becomes the From-Space and vice versa.

Before GC:

Environment: res $\rightarrow$ 15

Store: At the beginning the From-Space is from location 0 - 15 and the To-Space is from location 16 - 31.

Loc	Data
0	Num(10)
1	Box(0)
2	1
3	Box(2)
4	Num(11)
5	Box(4)
6	5
7	Box(6)
8	1
9	Box(8)
10	7
11	Box(10)
12	11
13	Box(12)
14	11
15	Box(14)
16	-
17	-
18	-
19	-
20	-
21	-
22	-
23	-
24	-
25	-
26	-
27	-
28	-
29	-
30	-

Loc	Data
31	-

After GC:

Environment: res $\rightarrow$ 24

Store: The From-Space is from location 16 - 31 and the To-Space is from location 0 - 15.

Loc	Data
0	-
1	-
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-
13	-
14	-
15	-
16	Num(11)
17	Box(16)
18	17
19	Box(18)
20	19
21	Box(20)
22	21
23	21
24	Box(23)
25	-
26	-
27	-
28	-
29	-
30	-
31	-