Homework: RefLang

Learning Objectives:

1. RefLang programming
2. Implement a complete interpreter for RefLang

Instructions:

- Total points: 66 pt
- Early deadline: Nov 6 (Wed) 2019 at 11:59 PM; Regular deadline: Nov 8 (Fri) 2019 at 11:59 PM (you can continue working on the homework till TA starts to grade the homework)
- Download hw7code.zip from Canvas
- Set up the programming project following the instructions in the tutorial from hw2 (similar steps)
- How to submit:
  - Please submit your solutions in one zip file with all the source code files (just zip the complete project's folder).
  - Write your solutions to question 5 in a file named "hw7.scm" and store it under your code directory.
```
hw7code
└── reflang
    └── src
        └── main
            └── java
                └── Example
```
  - Submit the zip file to Canvas under Assignments, Homework 7.
- In this homework, you will implement the interpreter for Reflang. **Here are all the changes that are required:**
  - Extend the set of values in Value.java to add RefVal which stores the location.(Question 1)
  - Implement memory in form of an array. You need to create a file Heap.java (Question 2)
  - Implement ASTs for required expressions in AST.java.(Question 3 a)
  - Extend Formatter for these required expressions.(Question 3 b)
  - Implement semantics of these required expressions in Evaluator.java(Question 4)
  - Extend the grammar for these newly added expressions.(Question 4)
  - Test your newly built Reflang interpreter for expressions developed in Reflang.(Question 5)

Reflang is a language with references. Once you complete building this interpreter, it will contain expressions for allocating a memory location, dereferencing a location reference, assigning a new value to an existing memory location and freeing a previously allocated memory location.

For example, you will be able to allocate a new piece of memory using the reference expression as follows.
$ (ref 1)
loc:0

Then, it will allow you to dereference a previously allocated memory location using the dereference expression.
$ (deref (ref 1))
1

You can also mutate the value stored at a memory location using the assignment expression.
$ (let ((loc (ref 1))) (set! loc 2))
2

Finally, it allows explicitly freeing a previously allocated memory location using the free expression as follows.
$ (free (ref 1))

Questions:

1. (4 pt) First, you will add a new kind of value to the set of values. This new kind of value will represent reference values in our language. You can do that by:
   - Adding a new Java class, RefVal, to the interface Value.
   - Internally this class will maintain an integer index, `_loc` (Additionally, it should have some initial value to indicate the value is not set yet), and
   - RefVal class must provide methods (a constructor method and a getter *loc*() method) to access this `_loc`.
   - The string representation of a RefVal (in method toString) will be created by prepending the string "loc" to the value of `_loc`.

2. (16 pt) Design and implement Heap, a new abstraction representing area in the memory reserved for dynamic memory allocation.
   (a) (4 pt) Implement Heap as a Java interface named Heap with four methods ref, deref, setref, and free.
      - The return type of all four methods is Value.
      - The method ref takes a single parameter of type Value.
      - The method deref takes a single parameter of type Value.RefVal.
      - The method setref takes a two parameters of type Value. First parameter is of type RefVal while second parameter is Value.
      - The method free takes a single parameter of type Value.RefVal.

   (b) (12 pt) Implement a 16 bit heap as a Java class Heap16Bit inside the interface Heap.

- The class Heap16Bit must implement the interface Heap, and thus provide implementation of each method ref, deref, setref, and free inside the interface.
- The class would model memory as an array named _rep of type Value[] with a size 65,536
- (3 pt) The method ref
  - takes a single parameter *value*, of type Value and if the heap is full, a DynamicError is returned if you run out of memory.
  - allocates memory and stores *value* in allocated memory at location l
  - returns a RefVal containing location l.
- (3 pt) The method deref
  - takes a single parameter *loc*, of type RefVal
  - returns value stored at location l, where l is stored in *loc*.
  - this method raises a DynamicError if an attempt to access a memory location outside the legal heap has been made.
- (3 pt) The method setref
  - takes two parameters
  - first parameter *loc*, is of type RefVal which encapsulates location l
  - second parameter *value*, is of type Value
  - this method replaces the value stored at l with *value*
  - returns *value*
  - this method raises a DynamicError if an attempt to access a memory location outside the legal heap has been made.
- (3pt) The method free
  - takes a single parameter *loc*, of type RefVal which encapsulates location l
  - deallocates the memory location l from _rep.
  - returns *loc*
  - this method raises a DynamicError if an attempt to access a memory location outside the legal heap has been made.

3. (10 pt) Question 1 and Question 2 helped you creating the RefVal and Heap representation. Now, in this question, you will create the AST node for the Reflang expressions.

   (a) (8 pt) Extend the AST.java and add the representation of the following nodes.
   - refexp
   - derefexp
   - assignexp
   - freeexp

   (b) (2 pt) Extend the Formatter for these new AST nodes in a manner consistent with existing AST nodes.

4. (17 pt) Implement the semantics of the Reflang expressions. You may refer to the introduction provided before the questions for your understanding.

   (a) (1 pt) Add a global heap of type Heap16Bit to the evaluator. This object will be the heap used by all expressions in the evaluator.

(b) (3 pt) Implement visit method for refexp in Evaluator.java according to the semantics of ref expression.

(c) (3 pt) Implement visit method for derefexp in Evaluator.java according to the semantics of deref expression.

(d) (3 pt) Implement visit method for assignexp in Evaluator.java according to the semantics of assignexp expression.

(e) (3 pt) Implement visit method for freeexp in Evaluator.java according to the semantics of freeexp expression.

(f) (4 pt) In order to get your Reflang working, you would be required to extend the grammar file. Extend the grammar file for supporting four new expressions of Reflang.

5. (19 pt) Test your implementation. Write your solutions to this question in a hw7.scm file and store it under your code directory mentioned in the instructions.

(a) (2 pt) Perform the following operations on your implementation of Reflang and provide of running those operations in hw7.scm file.
(deref (ref 342))
(free (ref 342))
(let ((loc (ref 342))) (set! loc 541))
(let ((loc (ref 540))) (set! loc (deref loc)))

(b) (2 pt) Write 2 Reflang programs which use aliases and also provide the transcript of running those programs.

(c) (15 pt) In this question you will implement a *binary tree* using Reflang. In a binary tree, one node is the root node. Every node other than the root must have a parent node, and has optional left and right child node. If there's no node child associated with this node, it is a leaf node. Each node of the tree can hold a value. See a similar linked list data structure below:

```
(define pairNode (lambda (fst snd) (lambda (op) (if op fst snd))))
(define node (lambda (x) (pairNode x (ref (list)))))
(define getFst (lambda (p) (p #t)))
(define getSnd (lambda (p) (p #f)))
```

The binary tree implementation uses a similar approach except that a node in this linked list only has one successor, but a node in a binary tree can have both left and right children.

i. (10 pt) Constructing the data structure:

A. (4 pt) write a lambda method `node` that accept one numeric argument as the value, and construct the node.

B. (2 pt) write a lambda method `value` that accept the node and return the numeric value.

C. (4 pt) write lambda methods `left` and `right` that takes a node and return its left or right child node.

ii. (5 pt) write a lambda method `add` that takes three parameters:

A. first parameter `p` is the parent node that the new node is going to append to.

B. second parameter `which` is a boolean variable where `#t` means add to left, and `#f` means add to right.
C. the last parameter `c` is the child node that is going to be added.
D. The `add` function adds `c` as a left or right child node to the `p` node.

Following transcripts will help you understand the functions more:

```
(define root (node 1))
(add root #t (node 2))
(add root #f (node 3))
(add (deref (left root)) #f (node 4))
(add (deref (right root)) #t (node 5))
(add (deref (right root)) #f (node 6))
```