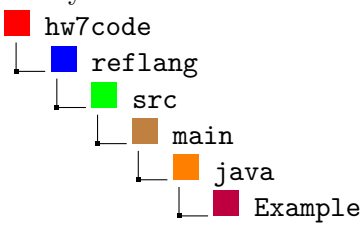
Homework Solutions: RefLang

Learning Objectives:

1. RefLang programming
2. Understand and expand RefLang interpreter

Instructions:

- Total points: 48 pt
- Early deadline: April 7 (Wed) at 11:59 PM; Regular deadline: April 9 (Fri) 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 1 in a file named "hw7.scm" and store it under your code directory.



  - Submit the zip file to Canvas under Assignments, Homework 7.

Questions:

1. (19 pt) [RefLang Programming] 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) Write a Reflang program that uses aliases and provide the transcript/output of running the programs.

   (b) (17 pt) 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 an optional left and right child node. If there's no child node 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. (8 pt) Construct the data structure:
   A. (3 pt) write a lambda function `treeNode` to define the tree node data structure.
   B. (3 pt) write a lambda function `node` that accepts one numeric argument as the value, and construct the node using treeNode.
   C. (3 pt) write a lambda function `value` that accepts the node and returns the numeric value.
   D. (3 pt) write lambda functions `left` and `right` that takesa node and return its left or right child node.

ii. (5 pt) Write a lambda function `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.

Here are some transcripts will help you understand the implementation:

```
(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))
```

Sol:

(a)

```
// if interpreter is restarted ...
$ (define a (ref 0))
loc:0
$ (let ((x a)) x)
loc:0

$ (let ((loc3 (ref 4))) (let ((loc4 loc3)) (deref loc4)))
4
```

(b) i.

```
;; node
(define treenode
        (lambda (one two three)
```

```
                    (lambda (num)
                        (if (= num 1) one
                        (if (= num 2) two
                        (if (= num 3) three #f))))))

        (define node
            (lambda (val)
                (treenode val (ref (list)) (ref (list)))))

        ;; value, left, right
        (define value (lambda (n) (n 1)))

        (define left (lambda (n) (n 2)))

        (define right (lambda (n) (n 3)))
```

ii.

```
        (define add
            (lambda (p which c)
                (if which
                    (set! (left p) c)
                    (set! (right p) c))))
```

2. (5 pt) [Alias detection] In Reflang an expression like the following creates two aliases (`class` and `course`) to the memory cell storing 342.

```
    (let ((class (ref 342))) (let ((course class)) (deref course)))
```

Modify the Reflang interpreter so that it prints a message when an alias is created. An example appears below.

$ (let ((class (ref 342))) (let ((course class)) (deref course)))
Alias created : name class ref value loc:0
Alias created : name course ref value loc:0
342
$ (define x (ref 23)) Alias created :name x ref value loc:1

Sol:

Evaluator.java

```
    @Override
    public Value visit(LetExp e, Env env) { // New for varlang.
        List<String> names = e.names();
        List<Exp> value_exps = e.value_exps();
        List<Value> values = new ArrayList<Value>(value_exps.size());

        for(Exp exp : value_exps)
        values.add((Value) exp.accept(this, env));

        Env new_env = env;
        for (int index = 0; index < names.size(); index++) {
```

```
                    new_env = new ExtendEnv(new_env, names.get(index), values.get(index));

                    if ( values.get(index) instanceof RefVal ){
                        System.out.print("Alias created :" + names.get(index) +" class ref value "
                            + values.get(index).tostring() + "\n");
                    }

                }

                return (Value) e.body().accept(this, new_env);
        }


@Override
public Value visit(DefineDecl e, Env env) { // New for definelang.
        String name = e.name();
        Exp value_exp = e.value_exp();
        Value value = (Value) value_exp.accept(this, env);
        ((GlobalEnv) initEnv).extend(name, value);
        if ( value instanceof RefVal ){
        System.out.print("Alias created : name " + e.name() +" ref value " + value.tostring() + "\n");
        }
        return new Value.UnitVal();
}
```

3. (5 pt) [Understanding the memory deallocation] In Reflang, the free expression deallocates memory. Current semantics of free expression is permissive in that it allows a memory location to be deallocated even if it has been deallocated previously. Change the semantics of the free expression such that the attempts to deallocate a value that is already deallocated results in a dynamic error with a message "Illegal deallocation of ref value loc:0". For example,
$ (let ((c (ref 342))) (let ((d (free c))) (free c)))
Illegal deallocation of ref value loc:0
$ (define x (ref 120))
$(free x)
$(free x)
Illegal deallocation of ref value loc:1

Sol:

```
        public Value visit(FreeExp e, Env env) { // New for reflang.
                Exp value_exp = e.value_exp();
                Value.RefVal loc = (Value.RefVal) value_exp.accept(this, env);
                if (heap.deref(loc) != null) {
                        heap.free(loc);
                        return new Value.UnitVal();
                } else {
                        return new DynamicError("Illegal deallocation of ref value " + loc.tostring());
                }
        }
```

4. (5 pt) [Typed heap] This question is about a semantic variation of the heap abstraction. Typed heap enforces the property that in a memory location, only values of compatible type can be stored. Two types are compatible if one is the subtype of the other. Extend the Reflang interpreter to support typed heap. Modify the semantics of assign expression assignexp to check that upon setting the value of a location the type of the new value is compatible with the type of the old value already stored in that location. Otherwise, raise a dynamic error.
[Hint: in Java you can use `isAssignableFrom` to check for compatibility of types.]
The following example scripts illustrates the semantics of typed heap.
$ (let ((x (ref 0))) (set! x 12))
12
$ (let ((x (ref 0))) (set! x #t))
Assigning a value of an incompatible type to the location in (set ! x #t)
$ (let ((x (ref (ref 0)))) (set! x (ref (ref (ref 5)))))
loc:4

Sol:

```
@Override
public Value visit(AssignExp e, Env env) { // New for reflang
        Exp rhs = e.rhs_exp();
        Exp lhs = e.lhs_exp();
        //Note the order of evaluation below.
        Value rhs_val = (Value) rhs.accept(this, env);
        Value.RefVal loc = (Value.RefVal) lhs.accept(this, env);
        //added for this question
        if(heap.deref(loc) != null && rhs_val.getClass().isAssignableFrom(heap.deref(loc).getClass
            ())) {
                Value assign_val = heap.setref(loc, rhs_val);
                return assign_val;
        } else return new Value.DynamicError("Assigning a value of an incompatible type to the
            location in " + ts.visit(e, null) ) ;
}
```

5. (14 pt) [Adding new features to RefLang] Add a new predicate, ref? to check if the value of an expression is a RefVal. The following example scripts illustrates the semantics of this predicate:
$ (define loc (ref 23))
$ (ref? loc)
#t
$ (ref? (let ((c (ref 342))) (set! c (ref 541))))
#t
$ (ref? (+ 3 (- 4 2)))
#f

Sol:

(a) (5pt) Grammar file(reflang.g)

```
exp returns [Exp ast]:
  | refcheck=refcheckexp { $ast = $refcheck.ast; }
```

```
        ;


refcheckexp returns [RefCheckExp ast] :
        '(' 'ref?'
        e=exp
        ')' { $ast = new RefCheckExp($e.ast); }
        ;
```

(b) (3pt) AST

```
        public static class RefCheckExp extends Exp {    // New for reflang
                private Exp _value_exp;

                public RefCheckExp(Exp value_exp) {
                        _value_exp = value_exp;
                }

                public Object accept(Visitor visitor, Env env) {
                        return visitor.visit(this, env);
                }

                public Exp value_exp() { return _value_exp; }

        }
```

(c) (5pt) Evaluator

```
        @Override
        public Value visit(RefCheckExp e, Env env) { // New for reflang
                Exp value_exp = e.value_exp();
                Value value = (Value) value_exp.accept(this, env);
                if (value instanceof RefVal){
                        return new BoolVal(true);
                }
                else
                return new BoolVal(false);
        }
```

(d) (1 pt) Printer

```
                public String visit(AST.RefCheckExp e, Env env) {       // New for reflang
                String result = "(ref?␣";
                result += e.value_exp().accept(this, env);
                return result + ")";
        }
```