Homework: FuncLang (Part II)

Questions:

1. (20 pt) Goal of the question is to understand the semantics and implementation of conditional expressions by extending the interpreter to support other features like equality on all data types.

   Current implementation of Funclang allows equality check only for the numeric values. Comparing two string values, Boolean values or list values results in DynamicError. For example, following transcript shows the result of comparing two Strings

   ```
   $ (= "name" "name")
   funclang.Value$StringVal cannot be cast to funclang.Value$NumVal
   ```

   Extend the Funclang language and its implementation such that the equality operator works for each kind of value supported by the language. For example,

   ```
   $ (= "abc" "abc")
   #t
   $ (= "abc" "abcdef")
   #f

   $ (= #t #t)
   #t
   $ (= #t #f)
   #f
   ```

   Two list values are considered equal if they have the same size and each element of the list is equal to corresponding element in the other list.

   ```
   $ (= (list) (list))
   #t
   $ (= (list 1 2 3 4) (list 1 2 3 4))
   #t
   $ (= (list 1 2 3 4) (list 1 2 3 4 5))
   #f
   $ (= (list 1 2 3 4 5) (list 1 2 3 4))
   #f
   $ (= (list 1 2 3 4 (list)) (list 1 2 3 4 (list)))
   #t
   $ (= (car (list 1 2 3)) 1)
   #t
   $ (= (car (list 1 2 3)) 2)
   #f
   $ (= (cdr (list 1 2 3)) 2)
   #f
   $ (= (cdr (list 1 2 3)) (list 2 3))
   #t
   ```

```
$ (= (cdr (list 1 2 3)) (cdr (list 4 2 3)))
#t
$ (= (cons 0 (list 1 2)) (list 0 (list 1 2)))
#f
$ (= (cons 0 (list 1 2)) (list 0 1 2))
#t
```

Two function values are only considered equal, if they point to the same FunVal object.

```
$ (define test (lambda (x) x))
$ (define test1 (lambda (y) y))
$ (= test test1)
#f
$ (= test test)
#t
```

Sol (20pt)

```java
public class Evaluator implements Visitor<Value> {
  public static boolean compareValue(Value v1, Value v2) {
    if (v1 instanceof NumVal && v2 instanceof NumVal) {
      NumVal first = (NumVal) v1;
      NumVal second = (NumVal) v2;
      return first.v() == second.v();
    } else if (v1 instanceof StringVal && v2 instanceof StringVal) {
      String s1 = ((StringVal)v1).v();
      String s2 = ((StringVal)v2).v();
      return s1.equals(s2);
    } else if (v1 instanceof PairVal && v2 instanceof PairVal) {
      boolean b1 = compareValue(((PairVal)v1).fst(), ((PairVal)v2).fst());
      boolean b2 = compareValue(((PairVal)v1).snd(), ((PairVal)v2).snd());
      return b1 && b2;
    } else if (v1 instanceof FunVal && v2 instanceof FunVal) {
      return v1 == v2;
    } else if (v1 instanceof BoolVal && v2 instanceof BoolVal) {
      return ((BoolVal)v1).v() == ((BoolVal)v2).v();
    } else if (v1 instanceof Null && v2 instanceof Null){
      // list
      return true;
    }else {
      return false;
    }
  }

  @Override
  public Value visit(EqualExp e, Env env) { // New for funclang.
    Value v1 = (Value) e.first_exp().accept(this, env);
    Value v2 = (Value) e.second_exp().accept(this, env);
    return new BoolVal(compareValue(v1, v2));
  }
}
```

2. (15 points) This question's goal is to strengthen the notation of higher-order functions. It does not require you to extend the Funclang interpreter.

   If f is a numerical function and n is a positive integer, then we can form the nth repeated application of f, which is defined to be the function whose value at x is f(f(...(f(x))...)). For example, if f is the function f(x) = x + 1, then the nth repeated application of f is the function f(x) = x + n.

(a) Define a function repeated that takes as inputs a procedure that computes f and a positive integer n and returns a function that computes the nth repeated application of f. Your function repeated should permit following usage.

```
$ (define inc (lambda (x) (+ x 1)))
$ ((repeated inc 2) 3)
5
$ ((repeated inc 10) 3)
13
$ ((repeated inc 100) 3)
103
$ ((repeated inc 3) 0)
3
$ ((repeated inc 3) -1)
2
$ ((repeated inc 3) -23)
-20
```

(b) Define two more examples of f to test the function repeated. Show both the functions and results

Sol

(a) (10pt)

```
(define repeated
  (lambda (op num)
    (if (= num 1)
        op
      (lambda (x)
        (op ((repeated op (- num 1)) x))))))
```

(b) examples (5pt):

```
      (define f1 (lambda (x) (+ x x)))
      (define f2 (lambda (x) (/ x 2)))
```

3. (20 points) This question's goal is to practice functional programming that combines all the language features we have learned, including recursion, high-order function, pairs and lists.

Implement a sorting algorithm using FuncLang as per the following specification.

Specification of pivot function.

- pivot function has two formal parameters: the first parameter is a number and the second parameter is a list of numbers.
- pivot function returns a list containing a pair of lists: the first element of the pair is the list of numbers in the input list that are less than the input number; and the second element of the pair is the list of numbers in the input list that are greater than or equal to the input number.
- Signature. pivot: $Z \times List \rightarrow List \times List$

Definition of pivot function,

```
(define pivot
(lambda (n)
(lambda (lst)
(if (null? lst)  lst
(if (> n (car lst))
(add-to-first (car lst) ((pivot n) (cdr lst)))
(add-to-second (car lst) ((pivot n) (cdr lst))))))))

(define (add-to-first x lst)
(list (cons x (car lst)) (cadr lst)))

(define (add-to-second x lst)
(list (car lst) (append (list x) (cadr lst))))
```

Specification of sort function.

- sort function has one formal parameter: a list of numbers.
- sort functions returns a list where the numbers from the input list are present in an ascending order.
- Signature. sort: $List \rightarrow List$

Write the definition of sort function using pivot. You can use the built-in functions such as null?, append, car, cdr and cons. You must not use any comparison operation over numbers. Explain your definition of the sort function.

Sol

(15 pt code, 5 pt explanation)

```
(define sort
  (lambda (mylist)
    (if (null? mylist) (list)
      (let ((middle (car mylist)))
        (let ((pair (pivot middle (cdr mylist))))
          (append (sort (car pair)) (cons middle (sort (cdr pair)))))))))
```

4. (25 points) This question's goal is to familiarize you with inbuilt function on different data structures. Extend the functionality of the Funclang interpreter to add support for following eight predicates: number?, boolean?, string?, procedure?, pair?, list?, null?, unit? A predicate is a function from Funclang value to boolean, i.e. #t or #f.

The predicate number? evaluates to #t if its input is a number. Similarly, boolean?, string? procedure? pair?, list?, null?, and unit? evaluate to #t if their inputs are boolean, string, procedure, pair, list, null, and unit respectively. Following transcript illustrates some of these predicates.

> (number? 342)
#t
> (boolean? (> 300 42))
#t
> (string? "342")
#t
> (procedure? (lambda (x) x))
#t

```
> (list? (list 3 4 2))
#t
> (list? (cons 1 2))
#f
> (pair? (cons 1 2))
#t
> (list? (cons 1 (list)))
#t
> (list? (list ))
#t
> (null? (list ))
#t
```

Funclang doesn't currently support expressions that produce unit values, so you wouldn't be able to test unit?, but you should implement it and we will verify your code directly to see if it behaves as intended.

Sol

Grammar file (5pt):

```
    exp returns [Exp ast]:
          ...
          | isnum=isNumExp { $ast = $isnum.ast; }
          | isbool=isBoolExp { $ast = $isbool.ast; }
          | isstring=isStringExp { $ast = $isstring.ast; }
          | isproc=isProcedureExp { $ast = $isproc.ast; }
          | islist=isListExp { $ast = $islist.ast; }
          | ispair=isPairExp { $ast = $ispair.ast; }
          | isunit=isUnitExp { $ast = $isunit.ast; }
          ;

isNumExp returns [IsNumExp ast] :
                '(' 'number?'
                    e=exp
                ')' { $ast = new IsNumExp($e.ast); }
                ;

isBoolExp returns [IsBoolExp ast] :
        '(' 'boolean?'
            e=exp
        ')' { $ast = new IsBoolExp($e.ast); }
        ;

isStringExp returns [IsStringExp ast] :
        '(' 'string?'
            e=exp
        ')' { $ast = new IsStringExp($e.ast); }
        ;

isProcedureExp returns [IsProcedureExp ast] :
        '(' 'procedure?'
            e=exp
        ')' { $ast = new IsProcedureExp($e.ast); }
        ;

 isListExp returns [IsListExp ast] :
        '(' 'list?'
            e=exp
        ')' { $ast = new IsListExp($e.ast); }
        ;

isPairExp returns [IsPairExp ast] :
        '(' 'pair?'
            e=exp
        ')' { $ast = new IsPairExp($e.ast); }
        ;

isUnitExp returns [IsUnitExp ast] :
        '(' 'unit?'
            e=exp
        ')' { $ast = new IsUnitExp($e.ast); }
        ;
```

AST.java (5pt)

```
public interface AST {
  public static class IsNumExp extends Exp {
    private Exp _arg;
    public IsNumExp(Exp arg){
      _arg = arg;
    }
    public Exp arg() { return _arg; }
    public Object accept(Visitor visitor, Env env) {
      return visitor.visit(this, env);
    }
  }

  public static class IsBoolExp extends Exp {
    private Exp _arg;
    public IsBoolExp(Exp arg){
      _arg = arg;
    }
    public Exp arg() { return _arg; }
    public Object accept(Visitor visitor, Env env) {
      return visitor.visit(this, env);
    }
  }

  public static class IsStringExp extends Exp {
    private Exp _arg;
    public IsStringExp(Exp arg){
      _arg = arg;
    }
    public Exp arg() { return _arg; }
    public Object accept(Visitor visitor, Env env) {
      return visitor.visit(this, env);
    }
  }

  public static class IsProcedureExp extends Exp {
    private Exp _arg;
    public IsProcedureExp(Exp arg){
      _arg = arg;
    }
    public Exp arg() { return _arg; }
    public Object accept(Visitor visitor, Env env) {
      return visitor.visit(this, env);
    }
  }

  public static class IsListExp extends Exp {
    private Exp _arg;
    public IsListExp(Exp arg){
      _arg = arg;
    }
    public Exp arg() { return _arg; }
    public Object accept(Visitor visitor, Env env) {
      return visitor.visit(this, env);
    }
  }

  public static class IsPairExp extends Exp {
    private Exp _arg;
    public IsPairExp(Exp arg){
      _arg = arg;
    }
    public Exp arg() { return _arg; }
    public Object accept(Visitor visitor, Env env) {
      return visitor.visit(this, env);
    }
  }

  public static class IsUnitExp extends Exp {
    private Exp _arg;
    public IsUnitExp(Exp arg){
      _arg = arg;
    }
    public Exp arg() { return _arg; }
    public Object accept(Visitor visitor, Env env) {
      return visitor.visit(this, env);
    }
  }
  public interface Visitor <T> {
    public T visit(AST.IsNumExp e, Env env);
    public T visit(AST.IsBoolExp e, Env env);
    public T visit(AST.IsStringExp e, Env env);
    public T visit(AST.IsProcedureExp e, Env env);
    public T visit(AST.IsListExp e, Env env);
    public T visit(AST.IsPairExp e, Env env);
    public T visit(AST.IsUnitExp e, Env env);
  }
}
```

Evalutor.java (10pt)

```
  public class Printer {
    public static class Formatter implements AST.Visitor<String> {

      public class Evaluator implements Visitor<Value> {
        @Override
        public Value visit(IsNumExp e, Env env) {
          Value val = (Value) e.arg().accept(this, env);
          return new BoolVal(val instanceof Value.NumVal);
        }

        @Override
        public Value visit(IsBoolExp e, Env env) {
          Value val = (Value) e.arg().accept(this, env);
          return new BoolVal(val instanceof Value.BoolVal);
        }

        @Override
        public Value visit(IsStringExp e, Env env) {
          Value val = (Value) e.arg().accept(this, env);
          return new BoolVal(val instanceof Value.StringVal);
        }

        //TODO
        @Override
        public Value visit(IsProcedureExp e, Env env) {
          Value val = (Value) e.arg().accept(this, env);
          return new BoolVal(val instanceof Value.FunVal);
        }

        @Override
        public Value visit(IsListExp e, Env env) {
          Value val = (Value) e.arg().accept(this, env);
          if(val instanceof Value.PairVal){
            return new BoolVal(((PairVal) val).isList());
          }
          else if(val instanceof Value.Null){
            return new BoolVal(true);
          }

          return new BoolVal(false);
        }

        @Override
        public Value visit(IsPairExp e, Env env) {
          Value val = (Value) e.arg().accept(this, env);
          return new BoolVal(val instanceof Value.PairVal);
        }

        @Override
        public Value visit(IsUnitExp e, Env env) {
          Value val = (Value) e.arg().accept(this, env);
          return new BoolVal(val instanceof Value.UnitVal);
        }
      }
```

Printer.java (5pt)

```
    public String visit(AST.IsNumExp e, Env env) {
      String result = "(number?␣";
      result += e.arg().accept(this, env);
      return result + ")";
    }

    public String visit(AST.IsBoolExp e, Env env) {
      String result = "(boolean?␣";
      result += e.arg().accept(this, env);
      return result + ")";
    }

    public String visit(AST.IsStringExp e, Env env) {
      String result = "(string?␣";
      result += e.arg().accept(this, env);
      return result + ")";
    }

    public String visit(AST.IsProcedureExp e, Env env) {
      String result = "(procedure?␣";
      result += e.arg().accept(this, env);
      return result + ")";
    }

    public String visit(AST.IsPairExp e, Env env) {
      String result = "(pair?␣";
      result += e.arg().accept(this, env);
      return result + ")";
    }

    public String visit(AST.IsListExp e, Env env) {
```

```
    String result = "(list? ";
    result += e.arg().accept(this, env);
    return result + ")";
  }

  public String visit(AST.IsUnitExp e, Env env) {
    String result = "(unit? ";
    result += e.arg().accept(this, env);
    return result + ")";
  }
    }
  }
```