Homework Solutions: FuncLang (Part II)

Learning Objectives:

1. Functional programming
2. Understand and expand FuncLang interpreter

Instructions:

- Total points: 54 pt
- Early deadline: Mar 24 (Wed) at 11:59 PM; Regular deadline: Mar 26 (Fri) at 11:59 PM (you can continue working on the homework till TA starts to grade the homework).
- We will grade functional programming based on our tests
- Download hw5code.zip from Canvas
- Set up the programming project following the instructions in the tutorial from hw2 (similar steps)
- How to submit:
  - For questions 1–2, please submit one pdf that contains all the source code
  - For questions 3–4, please submit your solutions in one zip file with all the source code files (just zip the complete project's folder).
  - Submit the zip file and one pdf file to Canvas under Assignments, Homework 5 submission.

Questions:

1. (8 pt) Write FuncLang programs to process a list of strings. Here, a string is a list of characters and each character is represented using a number.

   (a) (4 pt) Given a *character* and *a list of strings*, find strings that *do not* contain the given character. See the example below.

   ```
   $ (find 88 (list (list 77 73) (list 89)))
   ((77 73) (89))
   $ (find 88 (list (list 77 73) (list 88) (list 88 90 76)))
   ((77 73))
   ```

   Sol.

```
(define find
        (lambda (n lst)
                (if (null? lst)
                        (list)
                        (if (isPresent n (car lst))
                                (find n (cdr lst))
                                (appendChar (car lst) (find n (cdr lst)))
                        )
                )
        )
)

(define isPresent
        (lambda (n lst)
                (if (null? lst)
                        #f
                        (if (= (car lst) n)
                                #t
                                (isPresent n (cdr lst))
                        )
                )
        )
)

(define appendChar
        (lambda (lst1 lst2)
                (if (null? lst1) lst2
                        (if (null? lst2) lst1
                                        (list lst1 lst2)
                        )
                )
        )
)
```

(b) (4 pt) Given *a list of strings*, return a string that *concatenates* all the strings in the list and characters are sorted in *ascending* form. See the example below.

$ (sort (list (list 1 2) (list 3 4) (list 5)))
(1 2 3 4 5)
$ (sort (list (list 1 2) (list 5 4) (list 5)))
(1 2 4 5 5)
$ (sort (list (list 1 2) (list 4 3) (list 5)))
(1 2 3 4 5)

Sol.

```
(define append
```

```
        (lambda (lst1 lst2)
                (if (null? lst1) lst2
                        (if (null? lst2) lst1
                                (cons (car lst1) (append (cdr lst1) lst2))))))


(define Concatenate
        (lambda (lst)
                (if (null? lst)
                        (list)
                        (if (null? (cdr (cdr lst)))
                                (append (car lst) (car (cdr lst)))
                                (Concatenate
                                  (cons (append (car lst) (car (cdr lst)))
                                  (cdr (cdr lst))))))))


(define smaller (lambda (x y)
                  (if (< x y) x y)))

(define minimum (lambda (lst)
  (if (null? lst) (list)
      (if (null? (cdr lst))
          (car lst)
          (smaller (car lst) (minimum (cdr lst)))))))

(define removefromlist (lambda (ele lst)
  (if (null? lst) (list)
      (if (= ele (car lst))
          (cdr lst)
          (cons (car lst) (removefromlist ele (cdr lst)))))))

(define sortlist (lambda (lst)
               (if (null? lst)
               (list)
               (let ((x (minimum lst)))(cons x (sortlist (removefromlist x lst)))))))

(define sort (lambda (lst)
                  (sortlist (Concatenate lst))))
```

2. (6 pt) Write a function, `triangle`, which takes a *number* and produces *a list*; each element of the *list* is a list of symbols. When triangle is called with a non-negative integer, n, it returns a list containing n number of lists. The first inner list has n elements, the second inner list has n-1 element, and so on until you reach the top with only one element list, which forms the shape of a triangle. Each of the inner lists contain only the numbers 0 and 1 and they alternate across lists. The result always has the 0 as the first element of the first inner list, if any. In the following examples, we have formatted the

output to show the result more clearly, but your output will not look the same; it is sufficient to just get the outputs that are equal to those shown. Spaces in the lists are just for displaying purposes and you are not required to print them.

```
$ (triangle 0)
()
$ (triangle 1)
((0))
$ (triangle 2)
((0 1)
  (1))
$ (triangle 3)
((0 1 0)
  (1 0)
    (0))
$ (triangle 4)
((0 1 0 1)
  (1 0 1)
    (0 1)
      (1))
$ (triangle 5)
((0 1 0 1 0)
  (1 0 1 0)
    (0 1 0)
      (1 0))
        (0))
```

Sol.

```
(define alt (lambda (n)
              (if (= n 0) 1 0)))

(define alternate (lambda (x y)
                    (if (= x 0) (list)
                        (cons y (alternate (- x 1) (alt y))))))

(define helper (lambda (lst)
                    (if (null? lst) (list)
                              (cons lst (helper (cdr lst))))))

(define triangle (lambda (n)
                   (helper (alternate n 0))))
```

3. (20 pt) Extend the FuncLang interpreter by supporting ">" and "<" and "=" on strings and lists, supporting "=" on boolean values. For "=", we return true if the two strings have the exact length and

content. 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. For "<" and ">", the string and list comparison is done using the length of the strings and lists. That is, "> first second" returns true if the first string/list is longer than the second string/list; and "< first second" returns true if the first string/list is shorter than the second string/list.

For example,

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

$ (> "abc" "abcd")
#f
$ (< "abc" "abcdef")
#t
$ (= #t #t)
#t
$ (= #t #f)
#f


$ (= (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 (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
$ (> (list 1 2) (list))
#t
$ (> (list) (list 1))
```

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

Sol.

The logic in this answer can vary, but, be sure to check that the given tests hold. We made the method *isList* in *PairVal public* to use it in the comparison and also created *size* to return the *size* of the list (2 for pairs). Then, we defined the methods *equalValue* that checks for equality and *compareValue* which returns 0 for equal, -1 for less and 1 for greater. This logic can change and perhaps done using just one method.

Value (5 pt):

```
static class PairVal implements Value {
        // make isList public
        public boolean isList() {
                if(_snd instanceof Value.Null) return true;
                if(_snd instanceof Value.PairVal &&
                  ((Value.PairVal) _snd).isList()) return true;
                        return false;
        }

        // define size
        public int size() {
                if (!isList()) return 2;
                int result = 0;
                if (!(_fst instanceof Value.Null)) {
                        result += 1;
                }

                Value next = _snd;
                while(!(next instanceof Value.Null)) {
                        result += 1;
                        next = ((PairVal) next)._snd;
                }

                return result;
        }
}
```

Evaluator (15 pt):

```
public class Evaluator implements Visitor<Value> {

        public static boolean equalValue(Value v1, Value v2) {
```

```
        if (v1 instanceof NumVal && v2 instanceof NumVal) {
                NumVal first = (NumVal) v1;
                NumVal second = (NumVal) v2;
                return Double.compare(first.v(), second.v()) == 0;
        } 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 = equalValue(((PairVal)v1).fst(), ((PairVal)v2).
                boolean b2 = equalValue(((PairVal)v1).snd(), ((PairVal)v2).s
                if (b1 && b2) return true;
                return false;
        } 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;
        }
}

public static int compareValue(Value v1, Value v2) {
        if (equalValue(v1, v2)) {
                return 0;
        }

        if (v1 instanceof NumVal && v2 instanceof NumVal) {
                NumVal first = (NumVal) v1;
                NumVal second = (NumVal) v2;
                return Double.compare(first.v(), second.v());
        } else if (v1 instanceof StringVal && v2 instanceof StringVal) {
                String s1 = ((StringVal)v1).v();
                String s2 = ((StringVal)v2).v();
                return s1.compareTo(s2);
        } else if (v1 instanceof PairVal && v2 instanceof PairVal) {
                PairVal p1 = (PairVal) v1;
                PairVal p2 = (PairVal) v2;
                if (p1.isList() && p2.isList()) {
                        // we define a size method in PairVal
                        return Integer.compare(p1.size(), p2.size());
                } else if (!p1.isList() && !p2.isList()) {
                        // this case can be omitted in hw
                        // if they are both pairs,
                        // the result cannot be applied
```

```
                        return 0;
                    }
                }

                // default case can be 1 or -1 to define different
                return -1;
            }

            @Override
            public Value visit(LessExp e, Env env) { // New for funclang.
                Value first = (Value) e.first_exp().accept(this, env);
                Value second = (Value) e.second_exp().accept(this, env);
                return new Value.BoolVal(compareValue(first, second) < 0);
            }

            @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(equalValue(v1, v2));
            }

            @Override
            public Value visit(GreaterExp e, Env env) { // New for funclang.
                Value first = (Value) e.first_exp().accept(this, env);
                Value second = (Value) e.second_exp().accept(this, env);
                return new Value.BoolVal(compareValue(first, second) > 0);
            }
    }
```

4. (20 pt) Extend the syntax and semantics of the Funclang language to add support for a switch expression. The signature of switch-case is following:

   (switch (e0)
     (case e1 body)
     (case e2 body)*
     (default body)
)

   The switch expression will check whether the value of e0 is equal to the following cases from one by one. If equal, value of the corresponding body expression is returned as the result. If no matching is found, the value of the body of default is returned. There must be at least one `case` clause and exactly one `default`. Some examples:

   $ (define x 0)

```
$ (switch (x) (case 0 3) (case 1 4) (case 2 2))
error
$ (switch (x) (case 0 3) (case 1 4) (default 2))
3
$ (define foo (lambda (var) (switch (var) (case 1 (+ var 2)) (case 2 (- var 2)) (case 3 (* var 2)) (case
4 (/ var 2)) (default var))))
$(foo 1)
3
$ (foo 2)
0
$ (foo 3)
6
$ (foo 4)
2
$ (foo 5)
5
```

Sol.

Found in hw5code-sol.zip.
Files to modify: (The functions or classes that include modification are following):

Grammar file (6pt) Evaluator (10pt) AST (2pt) Printer (2pt)

Grammar file (Funclang.g)

```
exp returns [Exp ast]:
..
 | sw=switchexp { $ast = $sw.ast; } //added for switch
   ;


switchexp returns [SwitchExp ast]
 locals [ArrayList<Exp> cases = new ArrayList<Exp>();, ArrayList<Exp> body = new Ar
  '(' Switch
        '(' e0=exp ')'
         ('(' 'case' e1=exp  e2=exp  { $cases.add($e1.ast); $body.add($e2.ast);
} ')' )+
        '(' ('default' e3=exp) ')'
  ')' { $ast = new SwitchExp($e0.ast, $cases, $body, $e3.ast); }
  ;

  Switch : 'switch' ;
```

AST.java

```
    public static class SwitchExp extends Exp {
                Exp _e0;
```

```
                List<Exp> _cases;
                List<Exp> _body;
                Exp _e3;

                public SwitchExp(Exp e0, List<Exp> cases, List<Exp> body, Exp e3) {
                        _e0 = e0;
                        _cases = cases;
                        _body = body;
                        _e3 = e3;
                }

                public Exp e0() { return _e0; }

                public List<Exp> cases() { return _cases; }

                public List<Exp> body() { return _body; }

                public Exp e3() { return _e3; }

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

  public interface Visitor <T> {
// This interface should contain a signature for each concrete AST node.
    ...
 public T visit(AST.SwitchExp e, Env env); //Added for switch
```

Evaluator.java

```
        public Value visit(SwitchExp e, Env env) { // New for switch.
                NumVal e0 = (NumVal) e.e0().accept(this, env);
                List<Exp> cases = e.cases();
                List<Exp> body = e.body();
                NumVal e3 = (NumVal) e.e3().accept(this, env);

                List<NumVal> case_values = new ArrayList<NumVal>(cases.size());
                List<NumVal> body_values = new ArrayList<NumVal>(body.size());

                for(Exp exp : cases)
                        case_values.add( (NumVal) exp.accept(this, env));

                for(Exp exp : body)
                        body_values.add( (NumVal) exp.accept(this, env));

                for(int i=0; i< case_values.size(); i++){
```

```
                        if( case_values.get(i).v() == e0.v()){
                                return body_values.get(i);
                        }
                }
                return e3;
        }
```

Printer.java

```
    public String visit(AST.SwitchExp e, Env env) {
                String result = "(switch (";
                result+= e.e0().accept(this, env) + ")";
                List<Exp> cases = e.cases();
                List<Exp> body = e.body();
                int num_decls = cases.size();
                for (int i = 0; i < num_decls ; i++) {
                        result += " ( case ";
                        result += cases.get(i) + " ";
                        result += body.get(i).accept(this, env) + ")";
                }
                result +=" (default ";
                result += e.e3().accept(this, env) + ")";
                return result + ")";
        }
```