$Id: cse112-2020q4-final.mm,v 1.58 2020-12-10 11:12:28-08 - - $

Answer each question in the specific language given in the question. Do not use pseudo-code. Your answers should compile and run if copied verbatim into the appropriate language processor.

Use proper indentation. Points will be deducted for messy or unreadable answers.

Question 1. **[[3✔]]** <PRE>

Ocaml. Code the function find.

* The first argument is a list of tuples representing (key,value) pairs.
* The second argument is a comparison function. The key in the third argument should be the first argument to this function, and the key from the tuple should be the second argument.
* The third argument is a key to search for.

Return the first value in the list whose associated key is equal to the key argument.

Return the value as an 'a option: Some if the key is found. None if not. Its signature is:

```
('a * 'b) list -> ('c -> 'a -> bool) -> 'c -> 'b option
```

Example:

```
# find [1,2; 3,4; 5,6] (=) 7;;
- : int option = None
# find [1,2; 3,4; 5,6] (=) 3;;
- : int option = Some 4
```

Question 2. **[[3✔]]** <PRE>

Scheme. Code find.

* The first argument is a list of tuples representing (key,value) pairs. For Scheme, a tuple is a list of length 2 with the car as the key and the cadr as the value.
* The second argument is a equality comparison function, which you may assume is symmetric.
* The third argument is a key to search for.

Return the first value in the list whose associated key is equal to the key argument. Return #f if not found. Use cond, not if, to test the three cases.

Example:

```
> (find '((1 2) (3 4) (5 6)) = 3)
4
> (find '((1 2) (3 4) (5 6)) = 7)
#f
> (find '(("a" "b") ("c" "d") ("e" "f")) eq? "a")
"b"
```

Question 3. **[[5✔]]** <PRE>

Smalltalk. Code find. Extend class Array by adding a new method find:with:, which has two parameters.

* The object receiving the method is any element of class Array.
* The argument passed to the find: part is the key to be found.
* The argument passed to with: is the equality comparison function.

Each element of the array being searched is an array representing a tuple. Of that tuple, at:1 is the key, and at:2 is the value. Return the value associated with the first key that is found. If not found, return nil.

Hint on methods:

```
3 perform: #+ with: 7 returns 10.
```

Array extend [... allows you to add a method to existing class Array.

Example:

```
st> #(#(1 2) #(3 4) #(5 6)) find: 3 with: #=.
4
st> #(#(1 2) #(3 4) #(5 6)) find: 9 with: #=.
nil
```

Question 4. **[[7✔]]** <PRE>

Perl. Write some lines of code to read in a a sequence of graph links. Then print out the corresponding graph.

(a) Write some code to read in a a sequence of graph links. Each line is of the form

```
name1 -> name2
```

Every line has a name1, an arrow, and a name2, which indiates that name1 points at name2. There may be arbitary white spce between each element. Skip over any input data line with invalid syntax.

A name is any sequence of non-space characters. Load the data into %graph. Each key is a name1 from the data. Each value is a pointer to an array containing all of the name2 values that are pointed at by a name1. Use <> in a loop to read each line of input.

Example input:

```
0xFFFF -> 0xFFFF
  Foo -> Bar-Baz
0xFFFF -> Foo
Foo -> Helium
0xFFFF -> Hydrogen
Argentum -> Quux
Foo -> Quux
```

(b) Write a loop that prints out the contents of %graph, one item per line with the keys sorted into lexicographic order. For each key, print out the associated values in lexicographic order.

Example output:

```
0xFFFF -> 0xFFFF Foo Hydrogen
Argentum -> Quux
Foo -> Bar-Baz Helium Quux
```

Clearly indicate which of your code is for (a) and which of your code is for (b).

Question 5. **[[3✔]]** <PRE>

Fold left. Define the function sum using fold left.
Do not code the recursive function explicitly.

(a) Ocaml. Sum adds a list of floats.

```
# List.fold_left;;
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
# sum [1.;2.;3.];;
- : float = 6.
```

(b) Scheme. The foldl arguments are the same as in Ocaml. Assume sum works with any operand for which + works.

```
> foldl
#<procedure:foldl>
> (sum '(1 2 3 4))
10
```

Question 6. **[[1✔]]** <PRE>

```
Inheritance, part 1.
This question is connected to parts 2 and 3 and are to
be answered together.

Example program:
   nl := Character nl.
   a := NumExpr new: 6.
   stdout << a << nl.
   b := NumExpr new: 8.
   stdout << b << nl.
   c := AddExpr new: a with: b.
   stdout << c eval << nl.
   stdout << c << nl.

Output from this program:
   6
   8
   14
   (6+8)

Define class Expr as a subclass of Object.  It has no
method and no fields.  It is only used as a superclass
for the questions in parts 2 and 3.
```

Question 7. **[[4✔]]** <PRE>

```
Inheritance, part 2.

Define class NumExpr which is a subclass of Expr.  It
has a single value field which is a number.

Methods:
   * Class method new: has an argument which is a num-
     ber.
   * Instance method init:.
   * Instance method eval returns the value in the
     object.
   * Instance method printOn: prints the value to the
     stream argument.
```

Question 8. **[[6✔]]** <PRE>

Inheritance, part 3.

Define class AddExpr which is a subclass of Expr. It has data fields left and right which point at other instances of Expr.

Methods:

* Class method new:with: takes arguments which are instances of other Expr objects.
* Instance method init:with: stores those arguments in the object.
* Instance method eval returns the sum of the values of the two child nodes.
* Instance method printOn: prints output to its argument stream as follows: a left parenthesis, the left subchild, a plus sign (+), the right subschild, a right paremthesis.

Question 9. **[[4✔]]** <PRE>

Ocaml. Define merge that takes a comparison function and two sorted lists and returns a list merged into sorted order.

```
merge : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
```

```
# merge (<) [1;3;5;7;9] [2;4;6;8];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9]
# merge (>) [99;88;77] [95;85;76];;
- : int list = [99; 95; 88; 85; 77; 76]
# merge (<) [1;2;3;4] [1;2;3;4];;
- : int list = [1; 1; 2; 2; 3; 3; 4; 4]
```

Question 10. **[[4✔]]** <PRE>

Scheme. Define merge that takes a comparison function and two sorted lists and returns a list merged into sorted order. The order of the arguments and execution of the function should work in the same way as the Ocaml version of this question. Use cond. Do not use if.

Question 11. **[[3✔]]** <PRE>

Perl. Write a program that reads in numbers from the files specified on the command line, or from the standard input if no files are specified. Hint: <> does this.

Scan each line for numbers, and at end of file, print the sum of all the numbers on the line. A number is any sequence of decimal digits, with no signs, decimal point, or exponent. Any non-digit charcters are ignored.

```
-bash-92$ cat /tmp/num
33 44 55
838
100 3
-bash-93$ ./count.perl /tmp/num /tmp/num
2146
-bash-94$ echo 44 55 | ./count.perl
99
```

Question 12. **[[4✔]]** <PRE>

Ocaml.

(a) Define grep. It has two arguments: a predicate and a list. It returns a list containing all elements of the list for which the predicate returns true. The result list has elements in the same order as the input list.

```
grep : ('a -> bool) -> 'a list -> 'a list
```

(b) Define oddpos. It uses grep to return all odd positive ints in the argument list.

```
oddpos : int list -> int list
# oddpos [1;2;3;-4;-5;-6;0;7];;
- : int list = [1; 3; 7]
```

Do not mention the list in the definition of oddpos. Your answer will be a copy of the following statement, but with the dots replaced by valid code.

```
let oddpos = ..........
```

Question 13. **[[3✔]]** <PRE>

Scheme. Following is Euclid's algorithm to compute the greatest common divisor of two positive integers. Translate it from C into Scheme. Assume the arguments are both positive integers.

```
int gcd (int x, int y) {
   while (x != y) {
      if (x > y) x -= y;
            else y -= x;
   }
   return x;
}
```

Question 14. **[[1✔]]** <PRE>

Write a shell script (bash script) that reads in the contents of any number of files, then translates all upper case letters into lower case, then translates anything that is not a letter into a newline charaac-ter, then sorts all of the words in the pipeline, then removes any duplicates from the sorted list, then for-mats all of the words into a single paragraph with a maximum of 65 characters in any given line of output.

SCORE-TOTAL=51