CS403: Programming Languages

Assignment 2

Preliminary information

This is your second Scam assignment. At the top of your assignment, place a definition similar to the following:

```
(define (author)
    (println "AUTHOR: Rita Recursion rrita@crimson.ua.edu")
    )
```

with the name and email replaced by your own name and email.

For each numbered task (unless otherwise directed), you are to provide a function with a name of the form *runN*, with the *N* corresponding to the task number (as in *run1*, *run2*, etc.). This function is in addition to all other requested functions. These run functions will test your implementation and are to take no arguments. For example, if task 5 is:

5. Implement factorial so that it implements a recursive process. Name your function *fact*.

you should provide a *run* function similar to:

```
(define (run5)
    (exprTest (fact 0) 1)
    (exprTest (fact 3) 6)
    ...
    )
```

Woe betide students who provide insufficient testing should their implementations prove to be incorrect! If you do not complete an exercise, do not provide a run function for that exercise. If you omit the run function corresponding to an exercise, it will be assumed that you are skipping that part and you will receive no credit for that exercise.

When you have completed testing of your run functions, comment out any calls to them (but do not comment out the definitions).

I will provide a test script which performs minimal testing of your implementation. If your program does not pass the provided test script without failing, I will not grade your exercise and you will receive zero credit for the assignment.

There is a test script which performs minimal testing of your implementation. If your program does not pass the provided test script without failing, I will not grade your exercise and you will receive zero credit for the assignment. If your program passes the provided test script, it will be graded with a more thorough set of tests. You can get the test script with the command:

```
wget beastie.cs.ua.edu/proglan/test2.scm
```

You may not use assignment (e.g. *assign* or *set!*) in any of the code you write, unless otherwise instructed. Nor may you use any looping function such as *while* or *for*.

You will also be graded on style, including, but not limited to, the proper nesting of helper functions that are not expected to be called directly.

Tasks

1. You are to define an iterator loop that binds each item in a list, in turn, to a variable. With a binding accomplished, the loop executes its body. Here is an example:

   ```
   (iterate i (list 1 2 3 4)
       (inspect i)
       (inspect (* i i))
       )
   ```

   which should produce the output:

```
i is 1
(* i i) is 1
i is 2
(* i i) is 4
...
```

You will need to grab the calling environment and delay the evaluation of the arguments. Construct an appropriate lambda-like list and then evaluate the lambda-like list in the calling environment. Use the resulting closure to process each item in the second argument.

See the TSRM chapter on *parameter passing*.

2. Partial function evaluation is the process of breaking up the arguments to a function into two groups. When the first group of arguments and the function itself is passed to a partial-evaluator, a function that accepts the remaining arguments is returned. Define a variadic function, named *peval*, that partially evaluates a given function and some of the supplied arguments. The missing arguments are marked by the symbol `MISSING`. As an example, the last six expressions in the following list should evaluate to the same result:

```
(define . 'MISSING)
(define (f x y z) (+ x y z))
(f a b c)
((peval f . . .) a b c)
((peval f . b .) a c)
((peval f . . c) a b)
((peval f a . c) b)
((peval f a b c))
```

3. Define the following classes and methods:

   - Stack: constructor *Stack*; methods *push*, *pop*, *speek*, *ssize*
   - Queue: constructor *Queue*: methods *enqueue*, *dequeue*, *qpeek*, *qsize*

   Note: any method that would normally modify the state of the data structure has to return a new data structure, instead. All methods must work in amortized constant time. You may assume that every call to a peek method is followed by a pop or a dequeue, as the case may be.

   Here is a routine which uses such classes:

```
(define (loop stack queue)
    (define x (readInt))
    (if (eof?)
        (list stack queue)
        (loop (push stack x) (enqueue queue x))
        )
    )
(define (popper s)
    (cond
        ((!= (ssize s) 0)
            (inspect (speek s))
            (popper (pop s))
            )
        )
    )
(define (dequeuer q)
    (cond
        ((!= (qsize q) 0)
            (inspect (qpeek q))
            (dequeuer (dequeue q))
            )
        )
    )
```

   Here is a call:

```
(define oldstream (setPort (open "data.ints" 'read)))
(define data (loop (Stack) (Queue)))
(popper (car data))
(dequeuer (cadr data))
(setPort oldStream)
```

4. Define a function named *no-locals* which takes a source code list representing a legal Scam function definition and returns a source code list representing an equivalent definition with local defines replaced by a lambda. For example,

```
(no-locals
    (quote
        (define (nsq a)
            (define x (+ a 1))
            (* x x)
            )
        )
    )
```

should evaluate to the list:

```
(define (nsq a)
    ((lambda (x) (* x x))
        (+ a 1)
        )
    )
```

You may assume that any and all local defines are at the beginning of the function body. Note: Scam's local defines may refer to previous local definitions and are processed sequentially. You may also assume that all local defines are of the form:

```
(define varname expr)
```

You do not need to recursively remove local defines, just those at the top level of the function body.

5. Implement the function *pred*, suitable for taking the predecessor of the style of Church numerals found in the textbook. The function should take a Church numeral as its sole argument and return the appropriate Church numeral as its result.

You will likely need to peruse the web for this task.

6. Define a function, named *treedepth*, which computes that average depth of the leaves of a binary tree. The depth of any particular leaf in a tree is the number of nodes on the path from the root of the tree to the leaf, exclusive of the root.

Assume a binary tree is represented as a triple:

- the root value
- the left subtree
- the right subtree

with the following constructor:

```
(define (treeNode value left right)
    (list value left right)
    )
```

Given a non-empty binary tree as its sole argument, *treedepth* should return a real number representing the average depth. The only helper function you can use is the *treeflatten* function, which you must define. That function produces a list of leaf depth - leaf value pairs. You are to determine the final average using *treeflatten*, *map*, and *accumulate*, exclusively, with no other user-defined functions.

7. Exercise 2.42 in the text. Unlike the textbook version, the return value of the *queens* function should be a list of row-column lists with the row numbers in decreasing order. For example, here is an example return value for a board size of eight:

```
(((7 4) (6 7) (5 3) (4 4) (3 6) (2 1) (1 5) (0 0)) ...)
```

Note that the row numbers and column numbers start with 0 (unlike the textbook version). Note also that the first solution in the above return value is incorrect.

If there is no solution, *queens* should return the empty list.

8. Define a function, named *cxr*, which when given a symbol composed of `'a`s and `'d`s, generates a list function that extracts the appropriate element(s). An `'a` implies taking the *car* while a `'d` implies taking the *cdr*. Actions should be performed from right to left. For example:

```
(cxr 'd)
```

should return a function equivalent to *cdr*. Also,

```
(cxr 'da)
```

should return a function equivalent to `(lambda (x) (cdr (car x)))`.

```
(cxr 'ad)
```

should return a function equivalent to `(lambda (x) (car (cdr x)))`. Here's one last example:

```
((cxr 'add) '(1 2 3 4 5 6))
```

should return 3.

To convert a symbol to a string, one uses the *string* function. Like lists, one can take the head and tail of a string:

```
(car (string 'add))
(cdr (string 'add))
```

The two expressions above evaluate to `"a"` and `"dd"`.

To compare two strings, one should use the *equal?* function, which tests for structural equality, not the *eq?* function, which tests for pointer equality. Finally, the empty string `""` is equivalent to nil.

9. In this exercise, you will overload the `+` operator to call a supersized addition function. You will accomplish this task (and others) using a data-directed approach. In particular, you will implement the following logic for adding, subtracting, multiplying and dividing strings and integers.

```
(+ "x" "y")   ->  "xy"
(+ "123" 4)   ->  "1234"
(+ 123 "4")   ->  127
(- 123 "4")   ->  119
(- "abc" 1)   ->  "bc"
(* "abc" 3)   ->  "abcabcabc"
(* 3 "33")    ->  99
(/ 8 "2")     ->  4
```

Note, you can convert anything to a string with the *string* function and anything to an integer with the *int* function.

Next, you will need to include the *table.scm* file, which you can get with this command:

```
wget beastie.cs.ua.edu/proglan/table.scm
```

This file contains definitions for `getTable` and `putTable` (analogous to the SICP's *get* and *put*). You can get the type of an entity with the *type* function:

```
(type 3)  ->  INTEGER
```

Then, you will save old versions of the operators and define a function named *install-generic* for redefining operators to use the *apply-generic* function (from the text). and for installing the appropriate mathematical functions into the table:

```
(define old+ +)
(define old- -)
(define old* *)
(define old/ /)
(define (install-generic)
    (clearTable)
    (set! + (lambda (a b) (apply-generic '+ a b)))
    (set! - (lambda (a b) (apply-generic '- a b)))
    (set! * (lambda (a b) (apply-generic '* a b)))
    (set! / (lambda (a b) (apply-generic '/ a b)))
    (putTable '+ '(STRING STRING) addStrings)
    (putTable '* '(STRING INTEGER) mulStringInteger)
    ;other functions installed here
    ...
    'generic-system-installed
    )
```

You will also define a method for uninstalling your changes:

```
(define (uninstall-generic)
    (set! + old+)
    (set! - old-)
    (set! * old*)
    (set! / old/)
    'generic-system-uninstalled
    )
```

Your version of *apply-generic* should take exactly three arguments (operator and two operands) and should call the old version of the operator if *get* fails to find a function. Do this by installing the old version of the operator into the table with some special type symbols and have *apply-generic* retrieve it. NOTE: your version of *apply-generic* should not throw any errors.

Also, do not use assignment anywhere except as shown in *install-generic* and *uninstall-generic*.

10. Implement the function *coerce* using a data-directed approach. It should be able to perform the following coercions:

    - integer to real
    - integer to string
    - real to integer
    - real to string
    - string to integer
    - string to real
    - list to string

    Lists will be restricted to lists of integers, reals, strings (recursively).

    Use *table.scm* (see Task 9) to install your coercion functions. Your coerce function should grab the type of its first argument and retrieve the correct coercion function from the table. It should then return the result of applying the recursion function. Name your function for installing coercion functions in the table *install-coercion*.

    Examples:

```
scam> (coerce "123.4" 'INTEGER)
123
scam> (coerce '(1 (2.2) ((3 4) "5")) 'STRING)
(1 (2.2) ((3 4) "5"))
scam> (type (coerce '(1 (2.2) ((3 4) "5")) 'STRING))
STRING
```

Assignment submission

Submitting the assignment The entire assignment should be contained in a single file named *assign2.scm*. Any explanatory text should be in the form of Scam comments, which begin with a semicolon. The file should load into the Scam interpreter cleanly. The last line of your file should be:

```
(println "assignment 2 loaded!")
```

If you do not see the message ”assignment 2 loaded” when executing your file with the Scam interpreter, then there is an error somewhere that needs to be fixed. If your file does not load properly (i.e. I do not see the message), you will receive no credit for your work. To submit your assignment, delete extraneous files from your working directory. Then, while in your working directory, type the command:

```
submit proglan lusth assign2
```

The *submit* program will bundle up all the files in your current directory and ship them to me. Thus it is very important that only the files related to the assignment are in your directory (you may submit test cases and test scripts). This includes subdirectories as well since all the files in any subdirectories will also be shipped to me, so be careful. You may submit as many times as you want before the deadline; new submissions replace old submissions.