Winter 2024
Victoria Sakhnini

Assignment: 08

Due: Tuesday, April 2, 2024 9:00 pm
Coverage: End of Module 15
Language level: Intermediate Student with lambda
Allowed recursion: Simple, Accumulative, and Mutual
Files to submit: `partition.rkt`, `funabst.rkt`, `tree-pred.rkt`, `nested.rkt`, `matrix.rkt`, `bonus-a08.rkt`

- Make sure you read the OFFICIAL A08 post on Piazza for the answers to frequently asked questions.
- Policies from Assignment A07 carry forward.
- Any helper functions you write that are used by only one function **must be encapsulated within a `local`**. Helper functions used by more than one function can be defined globally, unless stated otherwise in the question. Functions written for testing purposes can be defined globally.
- **Pro Tip:** functions encapsulated with `local` cannot use `check-expect` directly. If you are having trouble getting your helper functions to work, you might want to develop them outside the main function, test them thoroughly, and then encapsulate them with `local` after you trust they work.

Here are the assignment questions you need to solve and submit.

1. **(15%)**: In this question, you will practice **using `local`** helper functions.

   (a) **(5%)**: In this question you will perform step-by-step evaluations of Racket programs, as you did in assignment one. Please review the instructions on stepping in A01 and complete the five required questions under the "Module 13: Locals" category on the CS135 Stepping Practice website.

   (b) **(10%)**: `partition` consumes a predicate and a list. It produces a two element list, `(list X Y)`, where `X` is a list of those items in the consumed list that satisfy the predicate and `Y` is a list of those items that don't satisfy the predicate. The order of items in each list must be the same as the original list.

   Do not use `filter`.

   Place your solution to the file `partition.rkt`.

2. **(15%)**: For this question, you **may not use any global or local helper functions**.

   (a) Write a function or-pred that consumes a predicate (that consumes one argument) and a list, and produces true if the application of the consumed predicate on any element of the consumed list produces true, otherwise the function produces false. If the consumed list is empty the function should produce false. For example:

   ```
   (or-pred even? empty) =>  false
   (or-pred odd? (list 6 10 4)) =>  false
   (or-pred string? (list 5 "wow")) => true
   ```

   (b) In class, we have seen that we are now able to put functions into lists. What can we do with lists of functions? One thing is to apply each function in the list to a common set of arguments. Write a function map2argfn which consumes a list of functions (each of which takes two numbers as arguments) and a list containing two numbers. It should produce the list of the results of applying each function in turn to the given two numbers. For example,

   ```
   (map2argfn (list + - * / list) (list 3 2)) =>
                                      (list 5 1 6 1.5 (list 3 2))

   (map2argfn empty (list 3 2)) => empty
   ```

   Note that in the above first example, the first list being passed to map2argfn has five elements, each of which is a function that can take two numbers as input. The resulting list is also of length five.

   Hint: Pay close attention to the contract for your function.

   (c) Write a predicate function arranged? that consumes a (list predicate-function binary-relational-operator) pair and a list of values (operands). The predicate function in the first list is to determine the data type of the values in the second list and the binary relational operator consumes the same data type (see contract below). Note that the first list is of length 2 and the second list may be empty.
   Here is a contract for arranged? (Hooray! Free mark!):

   ```
   ;; arranged?: (list (Any -> Bool) (X X -> Bool)) (listof Any) -> Bool
   ;; requires: if binary-relational-operator is applied on any
   ;;           elements, then predicate-function produces true on
   ;;           elements of type X
   ```

   The predicate arranged?:

   - produces true if the list of operands is empty or has one value and applying the predicate on it produces true.

```
(check-expect (arranged? (list integer? <) (list)) true)
(check-expect (arranged? (list integer? >) (list 1)) true)
(check-expect (arranged? (list integer? >) (list 'red)) false)
```

- produces `false` if applying the predicate on any of the operands produces `false`.

  ```
  (check-expect (arranged? (list string? >) (list "wow" 'red))
     false)
  ```

- produces `true` if applying the predicate on every operand produces `true` and applying the binary relational operator on all consecutive elements of the list of operands produces `true`, otherwise the function produces `false`.

  ```
  (check-expect (arranged? (list string? string>?) (list "wow"
     "cs135" "amazing")) true)
  ```

Place your solutions in `funabst.rkt`.

3. **(10%)**: Recall the structure and data definition of a binary tree from Module 10:

```
(define-struct node (key left right))
;; A Node is a (make-node Nat BT BT)

;; A Binary Tree (BT) is one of:
;; * empty
;; * Node
```

Write a function `tree-pred` which consumes a one-argument predicate (that consumes a `Nat`) and produces a function. That function will consume a binary tree and produce `true` if the predicate produces `true` for every value in the tree and `false` otherwise. If the tree is `empty`, the produced function should produce `true`.

For example:

```
(define t (make-node 5
                     (make-node 10 empty empty)
                     (make-node 15
                                (make-node 20 empty empty)
                                (make-node 33 empty empty))))
(check-expect ((tree-pred even?) t) false)
(check-expect ((tree-pred positive?) t) true)
```

Place your solution in `tree-pred.rkt`.

4. **(40%):** For this question, we have a new data definition:

```
;; A (nested-listof X) is one of:
;; * empty
;; * (cons (nested-listof X) (nested-listof X))
;; * (cons X (nested-listof X))
;; Requires: X itself is not a list type
```

Place your solution for the following parts in a file named `nested.rkt`.

(a) **(5%)**: Write a template function named `nested-listof-X-template` that processes a `(nested-listof X)`.

(b) **(15%)**: Write a function `nested-filter` that consumes a predicate function and a nested list (in that order) and removes every element that appears anywhere in the nested list where the predicate function is false for that element.

(c) **(5%)**: Write a function `ruthless` which consumes a nested list of symbols and produces an identical list except that all instances of `'ruth` have been removed.

You must use `nested-filter` in your solution.

```
(check-expect (ruthless '(rabbit (apple pluto (ruth blue) ruth)
   hello))
              '(rabbit (apple pluto (blue)) hello))
```

(d) **(5%)**: Write a function `keep-between` that consumes two numbers, *a* and *b*, and a nested list of numbers. It produces a nested list, keeping only the values between *a* and *b* inclusive.

You must use `nested-filter` in your solution.

```
(check-expect (keep-between 5 10 '(1 3 5 (7 9 10) (8 (3 4)) 8 15))
              '(5 (7 9 10) (8 ()) 8))
```

(e) **(10%)**: After applying `nested-filter` function from the previous part, the result may have empty nested lists. Write a function `nested-cleanup` that removes all `empty` lists anywhere in the consumed `(nested-listof Any)`. For example:

```
(nested-cleanup '(1 () 2 () () 3)) => '(1 2 3)
```

`nested-cleanup` will also remove nested empty lists:

```
(nested-cleanup '(1 (()()) 2 ((3 () (()))) )) => '(1 2 ((3)))
```

And if there are no non-list elements anywhere in the list, it produces `false`:

```
(nested-cleanup '(()(()())(())())) => false
```

To implement nested-cleanup, **you may not define any helper functions. This restriction includes local helper functions,** but you may define **local constants** if you wish.

5. **(20%)**: Matrices are very useful tools in mathematics and computer science. For the sake of simplicity, consider a matrix to be a 2D grid of elements/numbers where each number is indexed by row and column. An $m \times n$ matrix is a matrix with m rows and n columns.

   For example, the following 3x3 matrix A has 3 rows and 3 columns with element a_{ij} at row i and column j:

   $$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

   Using this notation, row 0 is

   $$\begin{pmatrix} a_{00} & a_{01} & a_{02} \end{pmatrix}$$

   and column 1 is

   $$\begin{pmatrix} a_{01} \\ a_{11} \\ a_{21} \end{pmatrix}$$

   You can add and subtract matrices of the same size by applying the operators element-wise.

   In Racket, we can model a matrix as a list of rows, i.e., a list of lists, and each row is a non-empty list of the same length (i.e. each row contains the same number of elements). Note that the number of rows in a matrix might not be the same as the number of columns in a matrix. A matrix that has no elements is represented by empty.

   For example, the matrix:

   $$M = \begin{pmatrix} -1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8.5 & 9 \end{pmatrix}$$

   can be represented in Racket as:

```
(define M (list (list -1 2 3)
                (list 4 5 6)
                (list 7 8.5 9)))
```

```
;; A Matrix is one of:
;; * empty
;; * (cons (listof Num) Matrix)
;; requires: each (listof Num) is non-empty and has the same length
```

(a) Write a function `matrix-apply` which consumes a list of functions (each with contract `Num → Num`, `Num → Int`, or `Num → Nat`) and a matrix, and produces a list of matrices. The first matrix should be the result of applying the first function to each matrix element, the second matrix should be the result of applying the second function to each matrix element, and so on. For example:

```
(matrix-apply (list abs floor (lambda (x) (+ x 3)))
              '((7 4.5 -3.2)(-3 3 13)))
```

produces

```
(list (list (list 7 4.5 3.2)
            (list 3 3 13))
      (list (list 7 4 -4)
            (list -3 3 13))
      (list (list 10 7.5 -0.2)
            (list 0 6 16)))
```

(b) Write a function `scale-smallest` which consumes a non-empty matrix and a real number (the `offset`). This function produces a second function that consumes a number, multiplies that number by the smallest element of the matrix, and adds the offset.

Then `((scale-smallest '((7 4.5 3.2) (-3 3 13)) 2.4) 7)` produces `-18.6` because $-3 \cdot 7 + 2.4 = -18.6$. Similarly, `((scale-smallest '((7 4.5 3.2) (-3 3 13)) 2.4) -2.7)` produces `10.5`.

Be careful to avoid exponential blowups in your function implementation.

Place your solution in `matrix.rkt`.

This concludes the list of questions for you to submit solutions (but see the following pages as well). Don't forget to always check the basic test results after making a submission.

Assignments will sometimes have additional questions that you may submit for bonus marks.

6. **(4% Bonus (each part worth 1%))**: In this question, you will write some convenient functions that operate on functions, and demonstrate their convenience. In addition to the other restrictions in this assignment, you may not use the built-in `compose` function. Place your solution in the file `bonus-a08.rkt`.

(a) Write the function `my-compose` that consumes two functions f and g in that order, and produces a function that when applied to an argument x gives the same result as if g is applied to x and then f is applied to the result (i.e., it produces `(f (g x))`).

(b) Write the function curry that consumes one two-argument function f, and produces a one-argument function that when applied to an argument x produces another function that, if applied to an argument y, gives the same result as if f had been applied to the two arguments x and y.

(c) Write the function uncurry that is the opposite of curry, in the sense that for any two-argument function f, (uncurry (curry f)) is functionally equivalent to f.

(d) Using the new functions you have written, together with filter and other allowed built-in Racket functions, give a nonrecursive definition of eat-apples from Module 14. You may not use any helper functions or **lambda**.

The name curry has nothing to do with delicious food in this case, but it is instead attributed to Haskell Curry, a logician recognized for his contribution in functional programming. The technique is called "currying" in the literature, and the functional programming language Haskell, which provides very simple syntax for currying, was also named after him. The idea of currying is actually most correctly attributed to Moses Schönfinkel. "Schönfinkeling" however does not have quite the same ring.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

The material below first explores the implications of the fact that Racket programs can be viewed as Racket data, before reaching back seventy years to work which is at the root of both the Racket language and of computer science itself.

The text introduces structures as a gentle way to talk about aggregated data, but anything that can be done with structures can also be done with lists. Section 14.4 of HtDP introduces a representation of Racket expressions using structures, so that the expression (+ (* 3 3) (* 4 4)) is represented as

```
(make-add
  (make-mul 3 3)
  (make-mul 4 4))
```

But, as discussed in lecture, we can just represent it as the hierarchical list '(+ (* 3 3) (* 4 4)). Racket even provides a built-in function eval which will interpret such a list as a Racket expression and evaluate it. Thus a Racket program can construct another Racket program on the fly, and run it. This is a very powerful (and consequently somewhat dangerous) technique.

Sections 14.4 and 17.7 of HtDP give a bit of a hint as to how eval might work, but the development is more awkward because nested structures are not as flexible as hierarchical lists. Here we will use the list representation of Racket expressions instead. In lecture, we saw how to implement eval for expression trees, which only contain operators such as +,-,*,/, and do not use constants.

Continuing along this line of development, we consider the process of substituting a value for a constant in an expression. For instance, we might substitute the value 3 for x in the expression (+ (* x x) (* y y)) and get the expression (+ (* 3 3) (* y y)). Write the function subst which consumes a symbol (representing a constant), a number (representing its value), and the list representation of a Racket expression. It should produce the resulting expression.

Our next step is to handle function definitions. A function definition can also be represented as a hierarchical list, since it is just a Racket expression. Write the function interpret-with-one-def which consumes the list representation of an argument (a Racket expression) and the list representation of a function definition. It evaluates the argument, substitutes the value for the function parameter in the function's body, and then evaluates the resulting expression using recursion. This last step is necessary because the function being interpreted may itself be recursive.

The next step would be to extend what you have done to the case of multiple function definitions and functions with multiple parameters. You can take this as far as you want; if you follow this path beyond what we've suggested, you'll end up writing a complete interpreter for Racket (what you've learned of it so far, that is) in Racket. This is treated at length in Section 4 of the classic textbook "Structure and Interpretation of Computer Programs", which you can read on the Web in its entirety at http://mitpress.mit.edu/sicp/ . So we'll stop making suggestions in this direction and turn to something completely different, namely one of the greatest ideas of computer science.

Consider the following function definition, which doesn't correspond to any of our design recipes, but is nonetheless syntactically valid:

```
(define (eternity x)
  (eternity x))
```

Think about what happens when we try to evaluate (eternity 1) according to the semantics we learned for Racket. The evaluation never terminates. If an evaluation does eventually stop (as is the case for every other evaluation you will see in this course), we say that it *halts*.

The non-halting evaluation above can easily be detected, as there is no base case in the body of the function eternity. Sometimes non-halting evaluations are more subtle. We'd like to be able to write a function halting?, which consumes the list representation of the definition of a function with one parameter, and something meant to be an argument for that function. It produces true if and only if the evaluation of that function with that argument halts. Of course, we want an application of halting? itself to always halt, for any arguments it is provided.

This doesn't look easy, but in fact it is provably impossible. Suppose someone provided us with code for halting?. Consider the following function of one argument:

```
(define (diagonal x)
  (cond
    [(halting? x x) (eternity 1)]
    [else                true]))
```

What happens when we evaluate an application of `diagonal` to a list representation of its own definition? Show that if this evaluation halts, then we can show that `halting?` does not work correctly for all arguments. Show that if this evaluation does not halt, we can draw the same conclusion. As a result, there is no way to write correct code for `halting?`.

This is the celebrated *halting problem*, which is often cited as the first function proved (by Alan Turing in 1936) to be mathematically definable but uncomputable. However, while this is the simplest and most influential proof of this type, and a major result in computer science, Turing learned after discovering it that a few months earlier someone else had shown another function to be uncomputable. That someone was Alonzo Church, about whom we'll hear more shortly.