Assignment: 8
Due: Tuesday, March 14, 2017 9:00pm
Language level: Intermediate Student
Allowed recursion: Pure structural recursion only, except for Q3
Files to submit: creativity.rkt, funabst.rkt, nest.rkt
Warmup exercises: HtDP 17.3.1, 17.6.4, 19.1.5, 20.1.1, 20.1.2, 24.0.7, 24.0.8
Practise exercises: HtDP 17.6.5, 17.6.6, 19.1.6, 20.1.3, 21.2.3, 24.0.9

- Coverage: until M10-36
- All helper functions must be local definitions, unless you are using a helper function defined in a previous part of the same question.
- You are not required to provide examples or tests for local function definitions. You are still encouraged to do informal testing of your helper functions outside of your main function to ensure they are working properly. Functions defined at the “top level” must include the complete design recipe.
- You may define global constants if they are used for more than one part of a question. This includes defining constants for examples and tests. Constants that are only used by one top level function should be included in the local definitions.
- In this assignment your Code Complexity/Quality grade will be determined both by how clear your approach to solving the problem is, and how effectively you use local constant and function definitions. You should include local definitions to avoid repetition of common subexpressions, to improve readability of expressions and to improve efficiency of code.
- You may only use the list functions that have been discussed in the notes up to the end of Module 9, unless explicitly allowed in the question.
- You may reuse the provided examples, but you should ensure you have an appropriate number of examples and tests.
- Your solutions must be entirely your own work.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide.
Here are the assignment questions you need to submit.

1. Perform the Assignment 8 questions using the online evaluation “Stepping Problems” tool linked to the course web page and available at

   https://www.student.cs.uwaterloo.ca/~cs135/stepping

   The instructions are the same as A03 and A04; check there for more information, if necessary. Reminder: You should not use DrRacket’s Stepper to help you with this question, for a few reasons. First, as mentioned in class, DrRacket’s evaluation rules are slightly different from the ones presented in class; you are to use the ones presented in class. Second, in an exam situation, of course, you will not have the Stepper to help you. Third, you can re-enter steps as many times as necessary to get them correct, so you might as well maximize the educational benefit.

2. Place your solution in the file funabst.rkt.

   (a) Write a function or-pred that consumes a predicate (boolean function which takes 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. For empty consumed list the function should produce false. For example:

   \[(\text{or-pred odd? empty}) \Rightarrow \text{false}\]
   \[(\text{or-pred even? (list 5 9 3)}) \Rightarrow \text{false}\]
   \[(\text{or-pred string? (list 5 "wow"))} \Rightarrow \text{true}\]

   (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 inputs. 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,

   \[(\text{map2argfn (list + − * / list) '(3 2))} \Rightarrow '(5 1 6 1.5 (3 2))\]

   Note that the above 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.
(c) Consider the data definition below.

;; An ASExp is one of:
;; * a Num
;; * a Sym (compliant with Racket rules for identifier)
;; * (cons Sym ASExplist) (where Sym is either ’+ or ’*)

;; An ASExplist is one of:
;; * empty
;; * (cons ASExp ASExplist)

;; Association List (AL) is one of
;; * empty
;; * (cons (list Sym Num) AL)
;; All Symbols are unique.

By allowing ASExp to be symbol, we can construct arithmetic expression lists that contain constants. For example: ’(+ 3 x (* y y)). Write the function evaluate to compute the value of an ASExp. The function consumes an ASExp and an AL, and produces the value of the expression. The AL (from symbols to numbers, where the key is Sym and the value is Num) will act as a dictionary, indicating the numeric value of each constant (matching constants names to values). Example: (evaluate ’(+ x 4) ’((x 5) (y 7))) should produce the value 9.

You may assume that each constant in the ASExp is present in the association list.

Note that apply is a defined function in Intermediate Student that does not do what you want.

You should develop your own version with your own name. Restrict your use of built-in functions to primitives such as mathematical functions, basic list functions (such as cons, list, first, etc. but not append, reverse, apply, etc.).
3. For this question, you may NOT use any helper function for any part of this question.
   For this question, consider the following data definition:

   ;; A nested list of numbers (Nest-List-Num) is one of the following:
   ;;  * empty
   ;;  *(cons Num Nest-List-Num)
   ;;  *(cons Nest-List-Num Nest-List-Num)

   (a) Write a function \texttt{nln-equal?} which consumes two nested lists of numbers and produces \texttt{true} if both are exactly the same and \texttt{false} otherwise. You may NOT use \texttt{equal?} for this part.

   (b) Write a function \texttt{list-elem-level} which consumes a nested list of numbers (\texttt{nln}) and a positive natural number (\texttt{level}) and produces a list of all numbers on nesting depth \texttt{level} of \texttt{nln}. For example:

   \[
   \begin{align*}
   (\text{list-elem-level} \ '(3 \ (4)) \ 1) & \Rightarrow \ '(3) \\
   (\text{list-elem-level} \ '(1 \ 2 \ 4 \ (5 \ (6 \ 7 \ (8 \ ((10) \ (12))(11)))))) \ 3 \ 6) & \Rightarrow \ '(10 \ 12)
   \end{align*}
   \]

   Submit your code for this question in a file named \texttt{nest.rkt}
4. For this question only, you may use any type of recursion. Place your solution in the file creativity.rkt.

(a) Write the function `infix` which consumes a prefix expression (as in Racket) and produces an equivalent infix expression (as in math). Assume operations are: `+` `*` `/ which are binary and `-` can be either unary or binary. For example:

\[
\text{infix } (\text{infix } (+ a (* b c ))) \Rightarrow (a + (b * c)) \\
\text{infix } (\text{infix } (- a (* b (- c))) ) \Rightarrow (a - (b * (- c))) \\
\text{infix } (- c) \Rightarrow (- c)
\]

**Note:** use the contract: `infix: PreExp -> InfExp`.

(b) A car approaching an intersection can go one of three ways: left, straight, or right. An intersection is represented as a list (left straight right) of possibilities. Each possibility is an intersection or a destination (symbol). Write the function `(drive intersec directions)` which will follow the list of `directions` (each of which is 'L', 'S' or 'R) and produces the destination denoted by the `directions`. If the `directions` continue past the `intersec` list, produce `false`. Also the function produces `intersec` for empty `directions`. For example:

\[
\text{drive } ((a b c) 'R) \Rightarrow 'c \\
\text{drive } ((a (b c (d e f)) g) '(S R L)) \Rightarrow 'd
\]

**Note:** For this part, consider the data definition below:

```
;; An Intersection is 
;; (list (anyof Intersection Sym) (anyof Intersection Sym) (anyof Intersection Sym))
```

(c) A family tree is a list (mother name father) where mother and father are family trees (or `false` if unknown)and name is a symbol. Write the function `(relative famtree aname)` which produces a list of `(anyof 'mother 'father)` giving the relation of `aname` to the person whose family tree (`famtree`) is consumed, or `false` if there is no relationship. For example:

\[
\text{relative } ((\text{false john false}) 'john) \Rightarrow () \\
\text{relative } ((\text{false john false}) 'sara) \Rightarrow false \\
\text{relative } ((\text{false mary (false bill false)) john (false fred false)}) 'bill) \Rightarrow '(mother father) \\
\]

**Note:** for this part, consider the data definition below:

```
;; A FamilyTree is 
;; (list (anyof FamilyTree false) Sym (anyof FamilyTree false))
```

This concludes the list of questions for which you need to submit solutions. Don’t forget to always check your email for the public test results after making a submission.
**Enhancements:** Reminder—enhancements are for your interest and are not to be handed in.

Professor Temple does not trust the built-in functions in Racket. In fact, Professor Temple does not trust constants, either. Here is the grammar for the programs Professor Temple trusts.

\[
\langle \text{exp} \rangle = \langle \text{var} \rangle \mid (\lambda \langle \text{var} \rangle \langle \text{exp} \rangle) \mid (\langle \text{exp} \rangle \langle \text{exp} \rangle)
\]

Although Professor Temple does not trust `define`, we can use it ourselves as a shorthand for describing particular expressions constructed using this grammar.

It doesn’t look as if Professor Temple believes in functions with more than one argument, but in fact Professor Temple is fine with this concept; it’s just expressed in a different way. We can create a function with two arguments in the above grammar by creating a function which consumes the first argument and returns a function which, when applied to the second argument, returns the answer we want (this should be familiar from the `addgen` example from class, slide 09-39). This generalizes to multiple arguments.

But what can Professor Temple do without constants? Quite a lot, actually. To start with, here is Professor Temple’s definition of zero. It is the function which ignores its argument and returns the identity function.

\[
(\text{define my-zero} (\lambda f (\lambda x) x))
\]

Another way of describing this representation of zero is that it is the function which takes a function \(f\) as its argument and returns a function which applies \(f\) to its argument zero times. Then “one” would be the function which takes a function \(f\) as its argument and returns a function which applies \(f\) to its argument once.

\[
(\text{define my-one} (\lambda f (\lambda x) (f x)))
\]

Work out the definition of “two”. How might Professor Temple define the function `add1`? Show that your definition of `add1` applied to the above representation of zero yields one, and applied to one yields two. Can you give a definition of the function which performs addition on its two arguments in this representation? What about multiplication?

Now we see that Professor Temple’s representation can handle natural numbers. Can Professor Temple handle Boolean values? Sure. Here are Professor Temple’s definitions of true and false.

\[
(\text{define my-true} (\lambda x (\lambda y) x))
\]

\[
(\text{define my-false} (\lambda x (\lambda y) y))
\]

Show that the expression \(((c a) b)\), where \(c\) is one of the values `my-true` or `my-false` defined above, evaluates to \(a\) and \(b\), respectively. Use this idea to define the functions `my-and`, `my-or`, and `my-not`. 

CS 135 — Winter 2017 Assignment 8 6
What about my-cons, my-first, and my-rest? We can define the value of my-cons to be the function which, when applied to my-true, returns the first argument my-cons was called with, and when applied to the argument my-false, returns the second. Give precise definitions of my-cons, my-first, and my-rest, and verify that they satisfy the algebraic equations that the regular Scheme versions do.

What should my-empty be?

The function my-sub1 is quite tricky. What we need to do is create the pair (0,0) by using my-cons. Then we consider the operation on such a pair of taking the “rest” and making it the “first”, and making the “rest” be the old “rest” plus one (which we know how to do). So the tuple (0,0) becomes (0,1), then (1,2), and so on. If we repeat this operation $n$ times, we get $(n-1,n)$. We can then pick out the “first” of this tuple to be $n-1$. Since our representation of $n$ has something to do with repeating things $n$ times, this gives us a way of defining my-sub1. Make this more precise, and then figure out my-zero?.

If we don’t have define, how can we do recursion, which we use in just about every function involving lists and many involving natural numbers? It is still possible, but this is beyond even the scope of this challenge; it involves a very ingenious (and difficult to understand) construction called the Y combinator. You can read more about it at the following URL (PostScript document):

http://www.ccs.neu.edu/home/matthias/BTLS/tls-sample.ps

Be warned that this is truly mindbending.

Professor Temple has been possessed by the spirit of Alonzo Church (1903–1995), who used this idea to define a model of computation based on the definition of functions and nothing else. This is called the lambda calculus, and he used it in 1936 to show a function which was definable but not computable (whether two lambda calculus expressions define the same function). Alan Turing later gave a simpler proof which we discussed in the enhancement to Assignment 7. The lambda calculus was the inspiration for LISP, a predecessor of Racket, and is the reason that the teaching languages retain the keyword lambda for use in defining anonymous functions.