Assignment: 8
Due: Tuesday, March 20, 2018, 9:00pm
Language level: Intermediate Student with Lambda
Allowed recursion: See individual questions
Files to submit: card-run.rkt, tree-abstractions.rkt, alf-dejavu.rkt
Practice exercises: HtDP 18.1.5, 19.1.6, 20.1.3, 20.2.4, 21.2.1, 21.2.2, 21.2.3, 24.0.9, 24.3.1, 24.3.2

• Make sure you read the OFFICIAL A08 post on Piazza for the answers to frequently asked questions.
• The Piazza post will also contain a list of the allowed Abstract List Functions (and other built-in functions).

Note that map has a form that consumes multiple lists. You may use this form.

For example: (map (lambda (x y) (list x y)) '(0 1 2 3) '(a b c d)) produces the list '((0 a) (1 b) (2 c) (3 d)).

foldr has a similar form, which is also permitted.

• 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 Abstract List Functions and local constants/functions. You should include local definitions to avoid repetition of common subexpressions, to improve readability of expressions and to improve efficiency of code.
• Encapsulate helper functions with local, and when appropriate use local to reduce repeated computation.
• Use lambda for single functions, as demonstrated in lecture.
• 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


   The instructions are the same as those in assignment 3; check there for more information if necessary.

2. Recall from Assignment 03 the Card structure:

   `(define-struct card (rank suit))
   ;; A Card is a (make-card Nat Sym)
   ;; requires:
   ;; 1 <= rank <= 13, where
   ;; 1 represents an Ace, 11 represents a Jack,
   ;; 12 represents a Queen and 13 represents a King
   ;; suit is one of 'clubs 'diamonds 'hearts or 'spades

   Also recall the assignment of ranks to values in the game Thirty-One:

   • An Ace is worth 11 points
   • A King, Queen, or Jack is worth 10 points
   • All other cards are worth their face value. For example the 5 of hearts is worth 5 points.

   Consider a list of cards. We will define a run of cards to be a sequence of adjacent cards in the list that have the same suit (but the ranks of these cards need not be in any particular order). The value of a run of cards is the sum of the values of those cards, according to the above rules. In a sequence of cards, the run that has the highest value is the highest-value run.

   Write a function biggest-run-value which consumes a nonempty list of cards. It produces a list of length two. The first element of the list is a symbol representing the suit of the highest-value run, and the second element is a natural number representing the value of that highest-value run.

   If multiple suits have a highest-value run, then the symbol produced should be the first suit in alphabetic order (so if 'hearts and 'clubs both have the highest-value run, the symbol produced is 'clubs). You may use symbol->string to determine this order, but there are solutions that do not require this.

   For example,
(check-expect
 (biggest-run-value
  (list (make-card 4 'spades) (make-card 3 'spades)
        (make-card 2 'spades) (make-card 5 'spades)
        (make-card 12 'diamonds) (make-card 2 'hearts)
        (make-card 6 'diamonds) (make-card 9 'diamonds)))
   (list 'diamonds 15))

You may use explicit recursion in this question. All helper functions must be encapsulated
using local. Also use local to avoid recomputing the same value multiple times.

Place your solution in card-run.rkt.

3. In this question you will write an Abstract Tree Function called bt-fold, and then use it
to solve some problems involving binary trees. The implementation of bt-fold will be
explicitly recursive; functions you implement using bt-fold will not be.

Recall the definition of Binary Trees from the lecture slides. Note that these binary trees are
not necessarily binary search trees.

(define-struct node (key val left right))
;; A Node is a (make-node Num Str BT BT)

;; A binary tree (BT) is one of:
;; * empty
;; * Node

Also remember that in this binary tree definition, empty is not a leaf. Rather, a leaf is a Node
with two empty subtrees.

The implementations of the functions below may not use external helper functions (but they
may use lambda and local). However, you may define additional helper functions for testing
purposes (such as combine functions to test bt-fold or a function to implement the testing
hint below).

Place your functions in tree-abstractions.rkt.

Testing Hint: We expect you to test these functions on non-trivial trees. You have some
options to construct non-trivial trees for testing:

- You could hand-code the trees. This gets awkward quickly, but is permitted.
- You could take the approach used in Assignment 07 and use constants that encode small
trees, then put those constants together to construct bigger trees.
- You could write a function that generates binary trees from a simpler representation
  such as a string or nested list. This function may be recursive.

We do not care which approach you use. We do care that the trees you use for testing are
non-trivial.
(a) Write a function `bt-fold` which behaves similarly to `foldr`, but on trees. The function has three parameters:
   - A combine function which consumes a key, a value, and the transformed results of the left and right subtrees.
   - A base value which is produced when the BT is empty
   - A BT

As such, the `bt-fold` function has the following contract:

```
btfold: (Num Str Y Y -> Y) Y BT -> Y
```

This function should abstract a template for binary trees in the same way `foldr` abstracts the template for lists.

(b) Use `bt-fold` to write a function `height` which consumes a binary tree and computes its height. The height of the empty tree is 0. The height of a tree of one node is 1. The height of any other tree is one more than the height of its tallest subtree.

(c) Use `bt-fold` to write a function `transform-values` which consumes two arguments. The first argument is a function with contract `Str -> Str`. The second argument is a binary tree. The function produces a binary tree that is a copy of the consumed tree except that every value is transformed according to the given function.

For example, given a tree `tree01`,

```
(bt-fold (lambda (x) (string-append x "?")) tree01)
```

would produce a tree identical to `tree01` except that all values have a question mark appended to the end.

(d) Use `bt-fold` to write a function `search` which consumes a number and a binary tree in that order. The function produces the value of the node whose key matches the number if such a key exists, and `false` otherwise. You may assume that every key is unique in the consumed tree. Remember that the consumed tree is an arbitrary binary tree, not necessarily a binary search tree.

(e) Use `bt-fold` to write a function `full?` which consumes a binary tree and produces `true` if every node of the tree has either zero or two nonempty children. The empty tree produces `true` (but be careful! This may not be the result you want to produce for the base value of `bt-fold`. `false` might not be a good base value either. What else might you produce?)

4. In this question, you will use Abstract List Functions to implement some familiar functions. Place your solution in the file `alf-dejavu.rkt`.

   You must use abstract list functions to solve these problems. No explicit recursion is allowed.

   Note that you may reuse many of your examples and tests from previous assignments when reimplementing these functions. (But if you lost marks for missing test cases in previous assignments, you should fix those problems first.)
(a) Implement prod-sqr from A04.
(b) Implement pyramid-vol from A04.
(c) Implement count-less-than from A04.
(d) Implement prime? from A05.
(e) Implement largest-even? from A04.
(f) Implement alternating? from A04. (Hint: use the trick from biggest-run-value to produce multiple values.)
(g) Implement column from A06.

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.

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

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 make-adder example from class, Section 10). 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 \textit{add1}? Show that your definition of \textit{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.

\begin{verbatim}
(define my-true (lambda (x) (lambda (y) x)))
(define my-false (lambda (x) (lambda (y) y)))
\end{verbatim}

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

What about \textit{my-cons}, \textit{my-first}, and \textit{my-rest}? We can define the value of \textit{my-cons} to be the function which, when applied to \textit{my-true}, returns the first argument \textit{my-cons} was called with, and when applied to the argument \textit{my-false}, returns the second. Give precise definitions of \textit{my-cons}, \textit{my-first}, and \textit{my-rest}, and verify that they satisfy the algebraic equations that the regular Scheme versions do. What should \textit{my-empty} be?

The function \textit{my-sub1} is quite tricky. What we need to do is create the pair \((0, 0)\) by using \textit{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 \textit{my-sub1}. Make this more precise, and then figure out \textit{my-zero}.

If we don’t have \textit{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):

\begin{verbatim}
http://www.ccs.neu.edu/home/matthias/BLTS/tls-sample.ps
\end{verbatim}

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 \textit{lambda} for use in defining anonymous functions.