Assignment: 9

Due: Tuesday, March 29, 2022 9:00pm

Language level: Intermediate Student with lambda

Allowed recursion: Simple, accumulative, and generative recursion

Files to submit: holf.rkt, ca.rkt

Warmup exercises: Without using explicit recursion complete HtDP 9.5.2, 9.5.4, and write your own versions of member? and append.

Practice exercises: HtDP 20.2.4, 24.0.7 24.0.8, and 24.0.9

• Make sure you read the OFFICIAL A09 post on Piazza for the answers to frequently asked questions.

• Unless stated otherwise, all policies from Assignment 08 carry forward.

• This assignment covers material up to the end of Module 16.

• You may use only the following higher order (list) functions: build-list, filter, map, foldr, and foldl.

• You may use any primitive functions, such as mathematical functions, cons, first, second, third, list, empty?, and equal?.

• You may use cond.

• You may use lambda.

• You may not use any non-primitive list functions, including length, member?, reverse, append, and list-ref.

• All helper functions must be local definitions unless otherwise specified. You may use earlier question parts as helpers without redefining them locally unless otherwise specified.

• Please note that some of the restrictions on local and helper functions may not be caught in the basic tests, and it is your responsibility to make sure your solutions does not violate any of the restrictions.

• Solutions will be marked for both correctness [85%] and style [15%]. Follow the guidelines in the Style Guide.
Here are the assignment questions you need to submit.

1. [5% Correctness] 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. To begin, visit this web page:

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

   **Note:** the use of https is important; that is, the system will not work if you omit the s. This link is also under the Assignments menu on the course web page.

   When you are ready, complete the required questions under "Module 16: Lambdas" (2 questions) using the semantics given in class.

2. [10 × 4= 40% Correctness] Implement the following functions. You may not use local. You may not define any helper functions, but you may use lambda to define anonymous functions to be used by higher order functions. Place your solutions in holf.rkt.

   **For this question, you may not use explicit recursion for any question.** That is, functions that involve an application of themselves, either directly or via mutual recursion. Use the higher order (list) functions instead. Also you must not must not use local in this question.

   **Note:** For this question, you can use list->string and string->list.

   (a) **occurrences** consumes a list of numbers and a number, in that order, and produces the number of times that the given number occurs in the list of numbers.

   (b) **absolutely-odd** consumes a list of integers and produces the sum of the absolute values of the odd integers in the list.

   (c) **zip** consumes two lists of equal length, and produces a list of pairs (two element lists) where the i\(^{th}\) pair contains the i\(^{th}\) element of the first list followed by the i\(^{th}\) element of the second list.

   (d) **unzip** consumes a list of pairs, and produces a list of two lists. The first list contains the first element from each pair, and the second list contains the second element from each pair, in the original order. Unzipping an empty list produces '(() ()).

   (e) Write a function **extract-keys** which consumes an Association List (AL) and produces a list of all of the keys in the association list (in the same order they appear in the AL).

   (f) **dedup** ("de-duplicate") consumes a list of numbers and produces a new list with only the first occurrence of each element of the original list.

   (g) (subsequence lst from to) consumes a list and two natural numbers. It produces the subsequence from lst that begins at index from and ends just before index to. Indexing starts at 0. If to < from, the function should produce empty.

   (h) Write **countup-to** from Section 07 Slide 23 (hint: use build-list).
(i) Write a function `shout` that consumes a list of strings and produces the same list of strings, but in all UPPERCASE. You **must** use the built-in function `char-upcase` to do the conversion.

(j) Write a function `make-validator` that consumes a (listof Any) and produces a predicate function (similar to the way that `make-adder` produces a function in Module 15). The produced function consumes a single item of type `Any` and produces a Boolean that determines if the item appears in the list that was consumed by `make-validator`.

```scheme
(check-expect (occurrences '(1 2 1 2 2 3 1) 2) 3)
(check-expect (absolutely-odd '(1 -5 4 6 5)) 11)
(check-expect (zip '(1 2 3) '(a b c)) '(((1 a)(2 b)(3 c))))
(check-expect (unzip '(((1 a)(2 b)(3 c))) '(((1 2 3) (a b c))))
(check-expect (unzip '())) '(()())
(check-expect (extract-keys '(((5 "five") (6 "six"))) '((5 6))
(check-expect (dedup '(1 2 1 3 3 2 4)) '1 2 3 4)
(check-expect (subsequence '(a b c d e f g) 1 4) '(b c d))
(check-expect (subsequence '(a b c d e f g) 1 1) '())
(check-expect (subsequence '(a b c d) 0 400) '(a b c d))

(check-expect (shout '("get" "off" "my" "lawn") '("GET" "OFF" "MY" "LAWN")
(define primary-colour? (make-validator '(red blue green)))
(check-expect (primary-colour? 'red) true)
```

3. **[10+10+10+10 = 40% Correctness]** A one-dimensional elementary cellular automaton (CA) is a computational machine that evolves a row of “cells”, each coloured black or white, according to a fixed “rule”. In each successive generation, the rule determines the colour of a cell based on the colours of that cell and its immediate left and right neighbours in the previous generation.

In this question we will use the binary representation of numbers. You may find the following sections from Wikipedia helpful: Representation and Counting in Binary. A cell and its two neighbours can be in eight possible configurations.

```
  111  110  101  100  011  010  001  000
```

Therefore, a rule is described by saying whether each of these configurations should produce a black or white cell in the next generation. Here is one possible rule.

```
  0  1  1  0  0  0  1  1
```
Then, given a row of black and white cells, the rule is applied to each trio of cells to generate the cells in a new row. A few examples are highlighted below, though of course the rule is applied everywhere. (Don’t worry for now about what happens at the start or end of the row; we’ll work around that later.)

We can produce interesting pictures by iterating this process. We draw a sequence of rows of black and white squares, where each row is generated by applying the rule to the row above it.

Because the rule depends on making a binary (black vs. white) choice for each of the eight possible trios of input squares, there are precisely \(2^8 = 256\) possible rules. We encode a rule as a number between 0 and 255 (inclusive) by assigning the number 0 to white squares and 1 to black squares and converting the rule’s outputs into a binary number, as in the diagram below, where we learn that the rule we’re working with is number 86.
Constructing images showing the generations of a CA’s operation is a prime example of generative recursion: each successive row is generated from the row before it through a general computation that doesn’t follow purely from the structure of the data. In this question you will build up to a Racket function to execute a CA, producing a list of rows where each row follows from the previous one through the application of a given rule.

(a) Write a function `apply-rule` that consumes four natural numbers \(a, b, c,\) and \(r\). The numbers \(a, b,\) and \(c\) are either 0 or 1, representing consecutive white and black squares in a row, from left to right. The number \(r\) is between 0 and 255, and encodes a CA rule as described above. The function should apply the given CA rule and produce either 0 or 1 depending on whether the resulting square should be white or black. Based on the diagram above, we would expect \((\text{apply-rule} 1 1 1 86) \Rightarrow 0, (\text{apply-rule} 1 1 0 86) \Rightarrow 1,\) and so on.

The best approach is first to convert \(a, b,\) and \(c\) into a single natural number between 0 and 7, as shown below. (Hint: treat the three numbers as bits in a three-bit binary number).

Note: any helper function that is not used multiple times must be lambda expression.

If the three cell colours produce some combined number \(v\), we then check whether the \(v\)th bit is set in \(r\). Note that in Racket, one way to perform this test would be to check \((\text{odd?} \ (\text{floor} \ (/ \ r \ (\text{expt} \ 2 \ v))))\).

(b) Using the function `apply-rule`, write a `next-row` that consumes two values: a list of 0s and 1s of length at least 1, representing a row in a CA, and a rule number between 0 and 255. It produces a new list of 0s and 1s, of the same length as the input list, containing the result of applying the given rule at every position in the original list.

The consumed list does not contain enough information to determine what should happen to the first and last cells. You should assume that all cells that are not explicitly encoded in the list are white.

You may find the functions `append`, `second`, and `third` useful in this question. A reasonable approach is for `next-row` to be a small wrapper around a simply recursive helper function.

(c) Write a higher-order function `iterate`. It consumes three values: a function \(f\), a base value \(b\), and a natural number \(n\), and produces a list of length \(n\) containing \((b, f(b), f(f(b)), f(f(f(b))), \ldots, f^{(n-1)}(b))\), where each \(f^{(i)}(b)\) is the result of applying \(f\) \(i\) times to \(b\). For example, \((\text{iterate} \ \text{sqr} \ 2 \ 4)\) would produce the list \(\'(2 \ 4 \ 16 \ 256)\).

Your function should use explicit recursion, and not use any helper functions defined globally or in a `local`. Each element of the resulting list should be generated from the one before it (meaning that your function will use generative recursion). That is, don’t
generate \( f^{i}(b) \) by starting from \( b \) every time and applying \( f \) to it \( i \) times; a given call to iterate should require only \( n - 1 \) applications of \( f \) in total. (Hint: in your recursive call, both \( n \) and \( b \) should change.)

(d) Write a function run-automaton. It consumes three values: an initial non-empty row of 0s and 1s, a rule number, and a number of generations \( n \). It produces a list of \( n \) lists, where the first sub-list is the consumed row, and each subsequent row is the result of applying the CA rule to the row before it.

Your function should be a one-liner: aside from the header of the function, it should consist of a single short line of code with no explicit recursion. Do not write any additional helper functions and do not use local. You might wish to use lambda. You may, of course, use the functions you have written for previous parts of this question.

You can write tests for this function by manually computing a few (short) rows of CA generations for a few different rules. You could also try using an online CA simulator like the one at http://www.mattlag.com/html5/automata.html to create tests; try choosing small values for the numbers of columns and iterations, and a larger number (around 25) for the Cell Size.

Place your solution in a file named ca.rkt.

Just for fun: It’s natural to want to draw the result of run-automaton into an actual image. If you want to try that, download the auxiliary file drawinglib-a09.rkt into the same directory as ca.rkt and add the line (require "drawinglib-a09.rkt") at the top of your solution. The file provides a function draw-ca that consumes two values: a list of lists, as produced by run-automaton, and a natural number giving the side length of a square in the grid. The function produces an image of the grid like the diagrams here (without the grey grid lines). Use draw-ca for interactive testing, but do not include any applications of that function in your submission. Also, do not include "drawinglib-a09.rkt" in your submission. Using the other functions in the auxiliary file, you can draw other images and shapes if you want to, as long as they are constructed using only rectangles and circles.

Note: If you used the drawing library, make sure to comment out the line (require "drawinglib-a09.rkt").

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 \textit{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.

\begin{verbatim}
(define my-zero (lambda (f) (lambda (x) x)))
\end{verbatim}

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.

\begin{verbatim}
(define my-one (lambda (f) (lambda (x) (f x))))
\end{verbatim}

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 `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/BLTL/tls-sample.ps

Be warned that this is truly mind bending.

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.