Winter 2024
Victoria Sakhnini

Assignment: 9

Due: Monday, April 8, 2024 9:00 pm
Coverage: End of Module 17
Language level: Intermediate Student with Lambda
Allowed recursion: No explicit recursion (see note below)
Files to submit: `alf.rkt`, `bw-images.rkt`, `subsets.rkt`

- Make sure you read the A09 Official Post and FAQ post on Piazza for the answers to frequently asked questions.
- For this assignment, there is no early (Friday) examples submission.
- Policies from Assignment A08 carry forward.
- **Unless otherwise stated, you may not use explicit recursion for any question.** This implies that functions which involve an application of themselves, either directly or via mutual recursion, are not allowed.
- If you need a helper function, it must be defined using **`lambda`**. Note that if the question does not allow explicit recursion.
- You should only use the higher-order functions covered in lectures: `filter`, `map`, `foldr`, `foldl` and `build-list`. DrRacket does have other high-order functions not covered in lectures, that you can review under under Help→ Help Desk → How to Design Programs Languages → 4.22 Higher-Order Functions (with Lambda).

Here are the assignment questions that you need to submit:

1. **(10%)**: In this question you will perform step-by-step evaluations of Racket programs, as you did in A01. Please review the instructions on stepping in A01 and complete the following:

   (a) The two required problems in "Module 15: Lambdas" and

   (b) The two required problems in "Module 16: Functional Abstraction"

   on the CS135 Stepping Practice website.

2. **(60%)**: Implement the following functions. You may not use **`local`**. You may not use helper functions other than with **`lambda`**. Place your solutions in `alf.rkt`. No design recipe is required for this question.

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

```
(check-expect (absolutely-odd '(1 -5 4 6 5)) 11)
(check-expect (absolutely-odd '()) 0)
```

(b) `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 `'(() ())`.

```
(check-expect (unzip '((1 a)(2 b)(3 c))) '((1 2 3) (a b c)))
(check-expect (unzip '()) '(()()))
```

(c) `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.

Note: you are **not** allowed to use `member?`.

```
(check-expect (dedup '(1 2 1 3 3 2 4)) '(1 2 3 4))
```

(d) `zero-fill` consumes a string no longer than 20 characters. It produces the same string but with zeros added to the beginning, as necessary, so that the string is exactly 20 characters long.

This problem is taken from computer networks where the string represents a "datagram". For the network to work efficiently, all the datagrams must be the same length – in our case, 20 characters.

For this question, you are not allowed to use any built-in string functions, except `list->string` and `string->list`. Here are a few examples for `zero-fill`:

```
(check-expect (zero-fill "abcdefghijklmn") "000000abcdefghijklmn")
(check-expect (zero-fill "he00llo") "0000000000000he00llo")
```

(e) `(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.

(f) `occ` that 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. Here are a few examples:

```
(check-expect (occ (list 1 1 1 2 1) 1) 4)
(check-expect (occ (list 1 2 3) 4) 0)
```

(g) `pocket-change` that consumes a list of symbols, and produces a number that is the total value of the change. The symbols that have values are

- `'penny` (worth \$0.01 each)
- `'nickel` (worth \$0.05 each)
- `'dime` (worth \$0.1 each)
- `'quarter` (worth \$0.25 each)
- `'loonie` (worth \$1.00 each)
- `'toonie` (worth \$2.00 each)

Any other symbol has zero value. If the consumed list is empty, the function returns 0. Note that the produced values do not need to have two decimal places (i.e., `1.2` or `6` are both acceptable, rather than `1.20` or `6.00`). Here are a few examples:

```
(check-expect (pocket-change (list 'dime 'dime 'penny 'fluff)) 0.21)
(check-expect (pocket-change (list 'loonie)) 1)
```

(h) The function `sum-at-zero`, which consumes a list of functions $(f_1,\ldots,f_n)$ (where each consumes and produces a number), and produces the value $f_1(0)+\cdots+f_n(0)$. For example,

```
(sum-at-zero (list add1 sqr add1)) => 2
```

If the consumed list of functions is empty, produce 0.

3. **(30%)**: This question focusses on simple black and white 2-dimensional images. We will represent a 2D black and white image as a list of lists of 0's (white) and 1's (black). We will define a black and white 2D image as:

```
;; BW-Pixel is (anyof 0 1)
;; 2D-Image is (listof (ne-listof BW-Pixel))
;;   Requires: inner lists of 2D-Image are of same length
```

Let us define a 2D black and white image, `image-L`, as follows:

```
(define image-L '((1 0 0 0)
                  (1 0 0 0)
                  (1 0 0 0)
                  (1 1 1 1)))
```

Then, the reflection of `image-L` across the x-axis, that is, the horizontal number line in the Cartesian coordinate system is:

```
(define image-L-reflect-x '((1 1 1 1)
                            (1 0 0 0)
                            (1 0 0 0)
                            (1 0 0 0)))
```

And the reflection of `image-L` across the y-axis, that is, the vertical number line in the Cartesian coordinate system is:

```
(define image-L-reflect-y '((0 0 0 1)
                            (0 0 0 1)
                            (0 0 0 1)
                            (1 1 1 1)))
```

(a) Write the function `(invert image)` that consumes a `2D-Image image`, and produces a `2D-Image` that is the inverted, such that, the black pixels are white and the white pixels are black. For example:

```
(define image-L-inverted '((0 1 1 1)
                           (0 1 1 1)
                           (0 1 1 1)
                           (0 0 0 0)))
 (check-expect (invert image-L) image-L-inverted)
```

(b) Write the function `reflect-x-axis` that consumes a `2D-Image` and produces a `2D-Image` that represents the reflection of the `2D-Image` across the x-axis. For example,

```
(check-expect (reflect-x-axis image-L) image-L-reflect-x)
```

(c) Write the function `reflect-y-axis` that consumes a `2D-Image` and produces a `2D-Image` that represents the reflection of the `2D-Image` across the y-axis. For example,

```
(check-expect (reflect-y-axis image-L) image-L-reflect-y)
```

(d) Write the function `transpose` which consumes a `2D-Image` and produces a `2D-Image` that represents the transposed image. To transpose an image, we exchange the rows and columns: the first row becomes the first column, the second row becomes the second column, and so on. For example:

```
(check-expect (transpose image-L)
              '((1 1 1 1)
                (0 0 0 1)
                (0 0 0 1)
                (0 0 0 1)))
(check-expect (transpose '((1 0 0 1 1)))
             '((1) (0) (0) (1) (1)))
```

You may use the function `length` for part (d).

Place all your functions for this question in the file `bw-images.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.

---

4. **Bonus (5%)**:

   Warning: Part (C) is a serious challenge. You have been warned!

   Place your solution in the file `subsets.rkt`.

   You do not need to include the design recipe for any of these bonus questions.

   (a) Write the Racket function `subsets1`, which consumes a list of numbers and produces a list of all of its subsets. For example, `(subsets1 '(1 2))` should produce something like `(list '(1 2) '(1) '(2) '())`. The order of subsets in the list may vary - any complete ordering will be accepted. You can assume the consumed list does not contain any duplicates. Write the function any way you want. (Value: 1%)

   (b) Now write the Racket function `subsets2`, which behaves exactly like `subsets1` but which does not use any explicit recursion or helper functions. You must rely on abstract list functions and **`lambda`** (and potentially standard list functions like `cons`, `first`, `rest`, `append`, etc.). Your solution must only be two lines of code, one of which is the function header. Note that if you solve this question, you can also use it as a solution to the previous one—just copy the function and rename the copy `subsets1`, or have `subsets1` call `subsets2`. (Value: 1%)

   (c) For the ultimate challenge, write the Racket function `subsets3`. As always, the function produces the list of subsets of a consumed list of numbers. Do not write any helper functions, and do not use any explicit recursion (i.e., your function cannot call itself by name). Do not use any abstract list functions. In fact, use only the following list of Racket functions, constants and special forms: `cons`, `first`, `rest`, `empty?`, `empty`, **`lambda`**, and **`cond`**. You are permitted to use **`define`** exactly once, to define the function itself. (Value: 3%)

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.

⟨exp⟩ = ⟨var⟩|(**lambda** (⟨var⟩) ⟨exp⟩) | (⟨exp⟩⟨exp⟩)

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 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.

```
(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.

```
(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.

```
(define my-true (lambda (x) (lambda (y) x)))
(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`.

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 Racket 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. Here are a few reading resources on the Y combinator

- University of Toronto lecturer David Liu has an explanation of the Y combinator,
- Chapter 9 has an example of the Y combinator, and
- medium.com also has a discussion on the Y combinator.

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 8. The lambda calculus was the inspiration for LISP, the predecessor of Racket, and is the reason that the teaching languages retain the keyword **`lambda`** for use in defining anonymous functions.