Homework 2a

Jason Hemann

December 6, 2017

1 Preamble

- Do consult class notes, online lecture notes, and test cases when completing this assignment.
- The only use of `letrec` that should be in this assignment is the use already present in problem 7.
- You should not need to use `pmatch` on this part (Part a) of this assignment.
- You may, however, find Scheme‘s `assv` and `remv` useful.

2 Part 1: Natural Recursion Refresher

1. Define and test a procedure `memv` that takes an element and a list and returns the first `cdr` whose `car` is `eqv?` to the element, or `#f` if the element is absent from the list. Your answer should not rely on Scheme‘s `memv`.

```
> (memv 'a '(a b c))
(a b c)
> (memv 'b '(a ? c))
#f
> (memv 'b '(a b c b))
(b c b)
```

2. Define and test a procedure `append` that takes two lists, `ls1` and `ls2`, and appends `ls1` to `ls2`. Your answer should not rely on Scheme‘s `append`.

```
> (append '(a b c) '(1 2 3))
(a b c 1 2 3)
```

3. Define and test a procedure `reverse` that takes a list and returns the reverse of that list. Your answer should not rely on Scheme‘s `reverse`. You may find it useful to use your definition of `append` in your answer.

```
> (reverse '(a 3 x))
(x 3 a)
```

3 Part 2: Useful Utility Functions and Function Extension

1. Define and test a procedure union that takes two flat sets (lists with no duplicate elements), and returns a list containing the union of the two input lists. You may find it helpful to use your `memv` in this definition. The order of the elements in your answer does not matter.

```
> (union '() '())
()
> (union '(x) '())
(x)
> (union '(x) '(x))
(x)
> (union '(x y) '(x z))
(x y z)
```

2. Define and test a procedure `extend` that takes two arguments, say `x` and `pred`. The second argument pred is a predicate. (Predicates are functions that return `#t` or `#f`.) What `extend` returns should be another predicate. The returned predicate should return `#t` if its input is `eqv?` to `x` or if its input satisfies `pred`.

```
> ((extend 1 even?) 0)
#t
> ((extend 1 even?) 1)
#t
> ((extend 1 even?) 2)
#t
> ((extend 1 even?) 3)
#f
> (filter (extend 1 even?) '(0 1 2 3 4 5))
(0 1 2 4)
> (filter (extend 3 (extend 1 even?)) '(0 1 2 3 4 5))
(0 1 2 3 4)
> (filter (extend 7 (extend 3 (extend 1 even?))) '(0 1 2 3 4 5))
(0 1 2 3 4)
```

3. Define and test a procedure `walk-symbol` that takes a symbol `x` and an association list `s`. An association list is a list of pairs of associated values. For example, the following is an association list:

```
((a . 5) (b . (1 2)) (c . a))
```

Your procedure should search through `s` for the value associated with `x`. If the associated value is a symbol, it too must be walked in `s`. If `x` has no association, then `walk-symbol` should return `x`.

```
> (walk-symbol 'a '((a . 5)))
5
```

```
> (walk-symbol 'a '((b . c) (a . b)))
c
> (walk-symbol 'a '((a . 5) (b . 6) (c . a)))
5
> (walk-symbol 'c '((a . 5) (b . (a . c)) (c . a)))
5
> (walk-symbol 'b '((a . 5) (b . ((c . a))) (c . a)))
((c . a))
> (walk-symbol 'd '((a . 5) (b . (1 2)) (c . a) (e . c) (d . e)))
5
> (walk-symbol 'd '((a . 5) (b . 6) (c . f) (e . c) (d . e)))
f
```

4 Part 3: `letrec`

7. Consider the following partial definition of the `list-ref` function. We intend it to operate similarly to Scheme's `list-ref`.

```
(define list-ref
  (lambda (ls n)
    (letrec
      ((nth-cdr
         (lambda (n)
           ;; complete the definition
           )))
      (car (nth-cdr n)))))
```

The body of the function that is the right-hand side of `nth-cdr` is missing. Complete the definition of `list-ref` with a naturally-recursive implementation of `nth-cdr`, so that the following work correctly. You should not need to modify the provided code beyond completing the function body containing a comment.

```
> (list-ref '(a b c) 2)
c
> (list-ref '(a b c) 0)
a
```

Remember, you need not consider bad data in your definition.