CSE 341 - Programming Languages
Midterm - Answer Key - Spring 2018

75 points total. You can bring a maximum of 2 (paper) pages of notes. No laptops, tablets, or smart phones. Please continue any long answers on the back of the page.

1. (8 points) Suppose that in Racket we have the following declarations. (This exam is on May 4th after all.)

```
(define x '(luke skywalker))
(define y '(han solo))
(define z '(jabba the hutt))
(define (f a) (cons a (cdr z)))
```

Given those declarations, what is the result of evaluating each of these Racket expressions?

(a)
```
(let ([x y]
      [y x])
  (append x y z))

'(han solo luke skywalker jabba the hutt)
```

(b)
```
(let* ([x y]
       [y x])
  (append x y z))

'(han solo han solo jabba the hutt)
```

(c)
```
(let ([x '(darth vader)]
      [z '(princess leia)])
  (f (car x)))

'(darth the hutt)
```

(d) Finally, suppose that Racket were dynamically scoped. What would be the result of evaluating this expression? This is the same expression as in Part (c).

```
(let ([x '(darth vader)]
      [z '(princess leia)])
  (f (car x)))

'(darth leia)
```

2. (10 points) We could define a `double_const` function in Haskell as follows:

```
double_const k x = [k,k]
```

This is like `const` from the Prelude, except that it returns a list with its first argument repeated.

(a) What is the Haskell type of `double_const`?

```
a -> b -> [a]
```

(b) Why can't we have a similar function `double_const` in Racket?

The second argument to `double_const` is never used, and so Haskell will not evaluate it. In Racket, all the arguments to functions are evaluated before the function is called, so the second argument *would* be evaluated. It might have a side effect or an error, so this could make a difference.

(c) Write a Racket macro `double_const`. Given your macro, the Racket expression `(double_const (+ 2 3) (cdr '()))` should evaluate to `'(5 5)`. You should only evaluate `(+ 2 3)` one time. As a hint to remind you of the syntax for macros, here is a definition of `my-or` from the lecture notes.

```
(define-syntax my-or
  (syntax-rules ()
    [(my-or) #f]
    [(my-or e1 e2 ...)
     (let ([temp e1])
       (if temp
           temp
           (my-or e2 ...)))]))
```

```
(define-syntax double_const
  (syntax-rules ()
    [(double_const e1 e2)
     (let ([temp e1])
       (list temp temp))]))

; an example showing that the first argument is only evaluated once,
; and the second not at all (this example isn't required in your answer)
(double_const (begin (printf "evaluating arg 1\n") (+ 3 4))
              (printf "evaluating arg 2\n"))
```

3. (5 points) Write a Racket function `evens` that takes an integer `n` and returns a list of all even numbers between 0 and `n`. Return the empty list if `n` is negative. You can use a helper function. You don't need to check for an argument that is not an integer. Examples:

```
(evens -2)  =>  '()
(evens 0)  =>  '(0)
(evens 4)  =>  '(0 2 4)
(evens 5)  =>  '(0 2 4)
```

```
(define (evens n)
  (evens-helper 0 n))

(define (evens-helper start stop)
  (if (> start stop)
      '()
      (cons start (evens-helper (+ 2 start) stop))))
```

4. (5 points) Let `x` be a Haskell infinite list, and `take 6 x` be the following:

```
[(2,"JEDI"),(3,"SITH"),(4,"JEDI"),(5,"SITH"),(6,"JEDI"),(7,"SITH")]
```

Write a definition for `x`. You can use additional variables or helper functions, although this isn't required. Hint: what is an expression that evaluates to the infinite list `2,3,4,` etc? What is an expression that evaluates to the infinite list `"JEDI", "SITH", "JEDI", "SITH"`, etc?

```
x = zip [2 ..] (cycle ["JEDI", "SITH"])
```

5. (6 points) State what `take 5` of each of these infinite Haskell lists would be.

```
b = 1 : -1 : 10 : map (+1) b
[1,-1,10,2,0]

c = zip [ x | x <- [1..], x `mod` 5 == 0] [1..]
[(5,1),(10,2),(15,3),(20,4),(25,5)]

d = foldr (++) [] $ repeat [1,2,3]
[1,2,3,1,2]
```

6. (12 points) The parser provided for the OCTOPUS interpreter represents Racket lists using the constructor `OctoList` followed by a Haskell list of `OctoValues`:

```
data OctoValue
      = OctoInt Int
      | OctoBool Bool
      | OctoSymbol String
      | OctoList [OctoValue]
        .....
```

Using this representation, `parse "()"` evaluates to `OctoList []`, and `parse "(1 2 3)"` evaluates to `OctoList [OctoInt 1, OctoInt 2, OctoInt 3]`.

Suppose instead that we represent OCTOPUS lists using explicit cons cells:

```
data OctoValue
      = OctoInt Int
      | OctoBool Bool
      | OctoSymbol String
      | OctoConsCell OctoValue OctoValue
      | OctoNil
        .....
```

Using the new representation, `parse "()"` evaluates to `OctoNil`.

(a) What does `parse "(x)"` evaluate to? This should still be a proper list.

```
OctoConsCell (OctoSymbol "x") OctoNil
```

(b) What does `parse "(5 (b c))"` evaluate to? Similarly, this should still be a proper list.

```
OctoConsCell
  (OctoInt 5)
  (OctoConsCell
    (OctoConsCell (OctoSymbol "b") (OctoConsCell (OctoSymbol "c") OctoNil))
    OctoNil)
```

(c) Write a primitive for `octocons` (i.e., a Haskell function to implement the primitive for `cons`), using the new representation for lists. Hint: here is the code for `octocons` as was demonstrated in lecture.

```
octocons [x, OctoList xs] = OctoList (x:xs)
```

The new version should still take a Haskell list of the arguments to the primitive.

```
octocons [x,y] = OctoConsCell x y
```

(d) With the new representation for OCTOPUS lists, you can also construct improper lists. Give a simple example of a Haskell expression that constructs an improper OCTOPUS list. (Hint: your expression will use the `OctoConsCell` constructor.) Draw a box-and-arrow diagram for the improper list.

```
OctoConsCell (OctoInt 3) (OctoInt 4)

                 _______________
                |       |       |
                |   o   |   o   |
                |___|___|___|___|
                    |       |
                    |       |
                    3       4
```

Or you could also draw it like this:

```
                 _______________
                |       |       |
                |   3   |   4   |
                |_______|_______|
```

7. (5 points) Haskell includes an `if-then-else` expression. However, this isn't actually needed: there could just be an `if'` function, which would be just an ordinary Haskell function.[1] For example,

```
if x==10
  then y
  else z
```

could be replaced by

```
if' x==10 y z
```

(a) Give a definition for the `if'` function. Use pattern matching.

```
if' True  x y = x
if' False x y = y
```

(b) What is its type?

```
Bool -> a -> a -> a
```

[1]There is some controversy in the Haskell community about whether to include such a function in the Prelude and to eliminate if-then-else.

8. (12 points) Consider various extensions to the OCTOPUS. For each extension, circle all of the changes or additions that would be needed. There will always be at least one, and there may be more than one. The choices are the same for each of the four parts of this question.

   For example, for a question: "Add Racket-style comments," if this weren't already one of the questions on HW 4, you would just circle "changing the lexer and/or parser," since you only needed to modify the lexer in `OctoParser.y`.

   (a) Add support for `let*`

   - changing the lexer and/or parser
   - changing the definition of `OctoValue`
   - $\boxed{\text{changing the eval function}}$
   - adding a new user-defined function
   - adding or changing one or more primitive functions (only if adding a new user-defined function wouldn't be enough)

   (b) Add a function `zero?` that returns `#t` if its argument is 0

   - changing the lexer and/or parser
   - changing the definition of `OctoValue`
   - changing the `eval` function
   - $\boxed{\text{adding a new user-defined function}}$
   - adding or changing one or more primitive functions (only if adding a new user-defined function wouldn't be enough)

   (c) Add integer division using the `quotient` function (for example `(quotient 11 2)` returns `5`)

   - changing the lexer and/or parser
   - changing the definition of `OctoValue`
   - changing the `eval` function
   - adding a new user-defined function
   - $\boxed{\text{adding or changing one or more primitive functions (only if adding ...)}}$

   (d) Add support for improper lists, using the dot notation from Racket, for example `(2 . 3)`. Assume that you use the new representation for OCTOPUS lists described in Question 6. Hint: you will need a new token for **.** (that is, a period).

   The item "changing the definition of `OctoValue`" is already circled for you, since that is one thing that needs to be changed from the version in `OctopusInterpreter-starter.hs`. There may be others.

   - $\boxed{\text{changing the lexer and/or parser}}$
   - $\boxed{\text{changing the definition of OctoValue}}$
   - $\boxed{\text{changing the eval function}}$
   - adding a new user-defined function
   - $\boxed{\text{adding or changing one or more primitive functions (only if adding ...)}}$

9. (6 points) Write a pointfree Haskell function `total_length` that takes a list of lists, and returns the sum of the lengths of the lists. You can use any of the functions in the Prelude, including `length` and `foldr`. Hint: use function composition.

   There are lots of possibilities. Here is one:

```
total_length = sum . map length
```

10. (6 points)

    (a) Write a Haskell function `find` that finds the first element of a list for which the given predicate is true, if there is one. To handle this properly, `find` should return `Just x` if such an element is found, and otherwise `Nothing` if none is found. For example, the `isUpper` function from `Data.Char` returns true if the argument is an upper-case letter and otherwise false. If we've imported `Data.Char`, then

```
find isUpper "testing 1 2 3 Squid Clam"
```

    evaluates to `Just 'S'`, and

```
find isUpper "testing 1 2 3 urchin"
```

    evaluates to `Nothing`.

    Hint: `Just a` and `Nothing` are the two constructors for the `Maybe` type. (If we didn't use `Maybe` or something similar, we'd need to just halt with an error if no element were found for which the predicate returns true.) The `Maybe` type was used in the OCTOPUS interpreter starter code, for example, when looking up a symbol in an environment.

```
find f [] = Nothing
find f (x:xs)
  | f x = Just x
  | otherwise = find f xs
```

    Or another version using `filter` from the Prelude:

```
find f xs =
  case ys of
    [] -> Nothing
    _ -> Just (head ys)
  where ys = filter f xs
```

    (b) What is the type of `find`?

```
(a -> Bool) -> [a] -> Maybe a
```