Problem Set 2: Scheme

Write all your solution code in a single file named `ps2.scm`. Make sure to use exactly the same function names and arguments (otherwise the marking software will give you 0!).

You can (and probably should!) create helper functions.

You don' t need to do much error checking: you can assume that valid data is passed to the functions you write.

Questions

1. (1 mark) Write a function called `(double-the-cheese pizza)` that takes a single list input (named `pizza` here) and replaces each top-level symbol on the list that equals `cheese` with two occurrences of `cheese`.

   Your function can assume `pizza` is a list; if passed a non-list, it is okay if your function crashes.

   For example:

```
1 ]=> (double-the-cheese '(salami onion))

;Value 12: (salami onion)

1 ]=> (double-the-cheese '(cheese salami onion))

;Value 13: (cheese cheese salami onion)

1 ]=> (double-the-cheese '(cheese salami onion cheese))

;Value 14: (cheese cheese salami onion cheese cheese)

1 ]=> (double-the-cheese '(cheese salami onion cheese cheese))

;Value 15: (cheese cheese salami onion cheese cheese cheese cheese)

1 ]=> (double-the-cheese '(cheese (salami cheese onion) cheese cheese))

;Value 16: (cheese cheese (salami cheese onion) cheese cheese cheese cheese)

1 ]=> (double-the-cheese '((cheese)))

;Value 17: ((cheese))
```

   Notice that only top-level `cheese` symbols get doubled; `cheese` symbols inside a list aren' t changed.

2. (2 marks) Write a function called `(my-last lst)` that returns the *last* element of `lst`. For example:

```
1 ]=> (my-last '(cat))

;Value: cat

1 ]=> (my-last '(cat dog))

;Value: dog
```

```
1 ]=> (my-last '(cat dog (1 2 3)))

;Value 11: (1 2 3)

1 ]=> (my-last '())

;empty list
```

Notice that calling `my-last` on an empty prints the error message “empty list” . To do this, evaluate `(error "my list")`.

MIT Scheme has a built-in function called `last` that does that same thing as `my-last`. Of course, don’ t use `last` in your implementation of `my-last`! Implement it using recursion and basic Scheme functions.

3. (1 mark) Write the function `(deep-sum lst)` returns the sum of all the numbers in `lst`, including numbers within lists. For example:

```
1 ]=> (deep-sum '(a 2 (b (1 c)) 3))

;Value: 6
```

You can assume `lst` is always a list. Return 0 if `lst` has no numbers.

Use `number?` to test for numbers, and `list?` to test for lists.

4. (1 mark) Write a function called `(is-bit? x)` that returns `#t` when `x` is the number 0 or 1, and `#f` otherwise.

For example:

```
1 ]=> (is-bit? 0)

;Value: #t

1 ]=> (is-bit? 1)

;Value: #t

1 ]=> (is-bit? 2)

;Value: #f

1 ]=> (is-bit? 'cow)

;Value: #f

1 ]=> (is-bit? '(0 1))

;Value: #f
```

Notice that `#f` is return for *every* input that is not either 0 or 1, even if the input is not a number.

5. (1 mark) Write a function called `(is-bit-seq? lst)` that returns `true` if `lst` is the empty list, or if it contains only bits (as defined by `is-bit?`). You can assume that `lst` is a list.

Note: MIT Scheme has a special built-in syntax, and some special functions, for bit strings. Don’ t use any of those for these questions!

6. (3 marks) Write a function called `(all-bit-seqs n)` that returns a list of all the bit sequences of length `n`. The order of the sequences doesn’ t matter. If `n` is less than 1, then return an empty list. You can assume that `n` is an integer.

   For example:

```
1 ]=> (all-bit-seqs 0)

;Value: ()

1 ]=> (all-bit-seqs 1)

;Value 14: ((0) (1))

1 ]=> (all-bit-seqs 2)

;Value 15: ((0 0) (0 1) (1 0) (1 1))

1 ]=> (all-bit-seqs 3)

;Value 16: ((0 0 0) (0 0 1) (0 1 0) (0 1 1)
            (1 0 0) (1 0 1) (1 1 0) (1 1 1))
```

7. (1 mark) Write a function called `(range n)` that returns a list with the values `(0 1 2 ... n-1)`. For example:

```
1 ]=> (range 4)

;Value 22: (0 1 2 3)

1 ]=> (range 9)

;Value 23: (0 1 2 3 4 5 6 7 8)

1 ]=> (range 0)

;Value: ()

1 ]=> (range -3)

;Value: ()
```

   MIT Scheme has a built-in function called `iota` that does that same thing as `range`. Of course, don’ t use `iota` in your implementation of `range`! Implement it using recursion and basic Scheme functions.

8. (2 marks) Write a function called `(count-primes n)` that returns the number of primes less than, or equal to, n. For example:

```
1 ]=> (count-primes -10)

;Value: 0

1 ]=> (count-primes 0)

;Value: 0

1 ]=> (count-primes 10)
```

```
;Value: 4

1 ]=> (count-primes 100)

;Value: 25

1 ]=> (count-primes 1000)

;Value: 168

1 ]=> (count-primes 10000)

;Value: 1229
```

While you should try to make `count-primes` reasonably efficient, the point of this question is to learn basic Scheme programming. So, while calling `(count-primes 1000)` should return its answer nearly instantaneously, it’ s okay if `(count-primes 10000)` takes, say, a few seconds to calculate its answer.