Problem Set 3
COMP301 Fall 2021
Week 4: 18.10.2021 - 22.10.2021

Please use the code boilerplate, which includes several tests for you to see if your code is correct. Submit your code to BlackBoard as `yourIDno_username.rkt` (scm extension is also fine). Example: `1234567_merciyes18.rkt`. You are expected to submit by the end of PS, however, you have until midnight to submit. The solutions will be available on the course Blackboard after Friday. **Read the questions carefully. Good luck!**

Problem 1 - Data Abstraction

In the previous lectures, you have seen that there are 2 implementations to represent natural numbers (other than Scheme Number Representation[1]):

- Unary Representation[2]
- BigNum Representation[3]

Part A. Please carefully read these references and explain how natural numbers are represented in Unary and BigNum Representations.

Part B. Natural numbers can be regarded as an *abstract data type*. An abstract data type consists of an interface and an implementation. In this part, you will create the implementations of Unary and Bignum representations. Keep in mind that these two representations actually refer to the same data type and must satisfy the same interface. Please implement them according to the interface given below:

- `create`: gets an integer number (and a non-zero integer number as base number only for BigNum representation) as input and creates the representation for that particular number.
- `is-zero?`: returns `#t` if the representation belongs to 0, otherwise returns `#f`.
- `predecessor`: gets a representation of a number (and a non-zero integer number as base number only for BigNum Representation) as input and returns the representation of the predecessor number.

Hint: Since you are representing natural numbers, return an error if predecessor is called on $\lceil 0 \rceil$.

Problem 2 - Recursively Specified Data and Programs

In this problem, you will define a recursive procedure to act on a recursively defined data type. Remember the very important guideline on writing procedures that operate on recursively defined data: *FOLLOW THE GRAMMAR!*[4]

Part A. Implement `count-free-occurrences`, which returns how many times a variable occurs free in a lambda-calculus expression. Figure 1 shows the grammar definition for lambda-calculus expressions.

Hint 1: In the tests of the starter code, check how LcExp data type is represented.
Hint 2: This example is a combination of `occurs-free?` and `count-occurrences`. You may find it easier to think how you can implement them separately and then combine.

[1]EOPL 3rd ed. p. 33
[2]EOPL 3rd ed. p. 33
[3]EOPL 3rd ed. p. 34
[4]EOPL 3rd ed. p. 22

```
LcExp ::= Identifier
      ::= (lambda (Identifier) LcExp)
      ::= (LcExp LcExp)
```

Figure 1. Grammar definition for LcExp

Part B (Optional). "Follow the Grammar" is a very powerful technique for writing recursive code. However, sometimes, it is not enough and auxiliary procedures are needed. Please implement procedure `product`: the expression `(product sos1 sos2)` where `sos1` and `sos2` are each a list of symbols without repetitions, returns a list of 2-lists that represent the Cartesian product of `sos1` and `sos2`. The 2-lists may appear in any order.

```
$ (product (a b c) (x y))
((a x) (a y) (b x) (b y) (c x) (c y))
```