CMPSC 461: Programming Language Concepts

Assignment 2. Due: Sep. 23, 11:59PM

For this assignment, you need to submit your solution as one single file named as "code.rkt" to Gradescope. You may NOT use any of Scheme's imperative features (assignment/loops) or anything else not covered in class. You should use Racket (https://racket-lang.org) for your implementation. For all problems, you can assume all inputs obey the types as specified in a problem.

Testing and Auto Grading We will use the Gradescope Autograder feature to grade the assignment. So it is important to start from the code template "code.rkt" on Canvas, which only has dummy implementationsbut is nevertheless useful since it obeys a few important requirements for auto grading, such as file name, function names etc. For you submission, just replace the dummy definitions with your implementations.

Before the submission deadline, you can see your partial score based on a public set of tests on Gradescope. You can debug your code and resubmit any time before the deadline. After the deadline, a final grade will be assigned based on the entire testsuite, including the public and a few extra private tests.

Problem 1 [20pt] In lecture, we define Church number $\underline{n}$ as $\lambda f\ n.\ (f^n\ z)$, where $(f^n\ z)$ represents the n-fold composition of function f applied to z (i.e., $f(f(\ldots(f\ z)))$ where f is repeated for n times).

a) (5pt) Implement a function `funPower`, which takes a function f, an integer n and returns the function f^n. For example, ((`funPower sqrt 2`) `16`) should return 2.

b) (5pt) Implement a function `encode`, which takes a natural number n and returns the church encoding of n. For example, (`encode 2`) should return 2 (i.e., the function $\lambda f.\lambda z.\ f\ (f\ z)$).

c) (5pt) Implement a function `decode`, which takes $\underline{n}$ (the church encoding of some natural number n) and returns n. For example

d) (5pt) Implement a functio [illegible] returns the church number of $n_1 \times n_2$ (i.e., $\underline{n_1 \quad _2}$ e 3)))) should be 6.

Problem 2 [5pt] Define a **recursive function** `dot`, whic [illegible] ns the dot product of them. The dot product of two lists of numbers $(x_1\ x_2\ \ldots\ x_m)$ nd $(y_1\ y_2\ \ldots\ y_n)$ is $\sum_{i=1}^{\min(m,n)} x_i \times y_i$. If any one of the lists is empty, your implementation should return 0. For example,

```
(dot '(2) '(3)) ; returns 6
(dot '(2) '(3 4)) ; returns 6
(dot '(1 -1 1 -1) '(1 1 1 1)) ; returns 0
```

Problem 3 [5pt] Define a **recursive function** `in`, which takes an element and a list, and returns a Boolean value indicating if that element appears in the list. Your implementation should return `#f` if the list is empty. Note that your implementation should be able to handle nested lists, such as the third example below:

```
(in 4 '(1 2 3)) ; returns #f
(in "abc" '(1 #f "abc")) ; returns #t
(in 2 '(1 (1 2) 3)) ; returns #t
```

Problem 4 [5pt] Define a **recursive function** lstOddSum, which when given a list of numbers, it returns the sum of numbers at *odd* positions in the list. Your implementation should return 0 if the list is empty. You can assume that the input is a list of numbers without nested lists. For example,

```
(lstOddSum '(1)) ; returns 1
(lstOddSum '(1 2)) ; returns 1
(lstOddSum '(1 2 3)) ; returns 4
```

Problem 5 [15pt] Implement the following functions in Scheme using `map` and `foldl`. DO NOT use recursive definition for this problem.

a) (5pt) Define a function `trunc`, which takes a lower bound a, an upper bound b and a list of numbers, and replaces all numbers that are $< a$ with a and all numbers that are $> b$ with b. For example,

```
(trunc 0 1 '(-2 -1 0 1 2)) ; returns '(0 0 0 1 1)
(trunc 0.5 1.5 '(-2 -1 0 1 2)) ; returns '(0.5 0.5 0.5 1 1.5)
(trunc 1 1 '(-2 -1 0 1 2)) ; returns '(1 1 1 1 1)
```

b) (5pt) Define a function `leq`, which takes two lists of numbers $(x_1\ x_2\ \ldots\ x_n)$, $(y_1\ y_2\ \ldots\ y_n)$ and returns `#t` if and if and only if $x_i \leq y_i$ for all $1 \leq i \leq n$. You can assume that both lists have the same length. For example,

```
(leq  '(1 2 3) '(2 3 3)) ; returns #t
(leq  '(1 0) '(0 1)) ; returns #f
(leq  '() '()) ; returns #t
```

c) (5pt) Define a function `dup`, which takes a list and duplicates every element in that list. For example,

```
(dup '()) ; returns '()
(dup '("abc" #t 7 7)) ; returns '("abc" "abc" #t #t 7 7 7 7)
(dup '(1 (2 3) (2 3))) ; returns '(1 1 (2 3) (2 3))
```