CMPSC 461: Programming Language Concepts

Assignment 2. Due: Sep. 16, 11:59PM

For this assignment, you need to submit your solution as one single file to Canvas. You may NOT use any of Scheme's imperative features (assignment/loops) or anything else not covered in class. Define auxiliary functions where appropriate. While you may use whatever implementation of Scheme you like, we highly recommend using Petite Scheme (www.scheme.com), which provides a standard implementation and is the one we will be testing your code on. For all problems, you can assume all inputs obey the types as specified in a problem. We have provided a test file "hw2-test.scm" on Canvas for your testing.

We will be running your programs against a script. So it is important to check the following requirements before you submit: 1) the function names in your submission must match exactly as specified in this assignment; 2) make all of your function definitions global (i.e., use "define"); 3) name your submission as psuid.scm (e.g., xyz123.scm); 4) make sure the file you submit can be directly loaded into Scheme (to test, use command load "file_name" in Scheme). Failing to follow these requirements may result in NO CREDIT for your submission.

Problem 1 [6pt] In assignment 1, we have encoded a pair and operations on pair as follows:

$$\texttt{PAIR} \triangleq \lambda a\ b\ p.\ (p\ a\ b)$$

$$\texttt{LEFT} \triangleq \lambda p.\ (p\ (\lambda t\ f\ .\ t))$$

$$\texttt{RIGHT} \triangleq \lambda p.\ (p\ (\lambda t\ f\ .\ f))$$

Implement functions `PAIR`, `LEFT` and `RIGHT` in Scheme. For example, (`LEFT` (`PAIR` 1 2)) should return 1. Hint: you need to curry the definition of `PAIR` in an appropriate way.

Problem 2 [8pt] In the Church encoding of natural numbers, we have used the n-fold composition of f, written f^n. Intuitively, f^n means applying function f for n times.

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.

Problem 3 [8pt] We define the *depth* of a value as follows: the depth of a non-list value is 0; the depth of a list value is 1 plus the maximum depth of its elements.

Implement a recursive function `depthOfList` that takes a list l and returns the depth of l. For example, (`depthOfList` '()) should return 1 and (`depthOfList` '(0 (0 ()) ())) should return 3.

Problem 4 [10pt] Consider the problem of computing the exponential of a given number. With a base b and a positive integer exponent n, the naive way of computing b^n is to repeat multiplication for $n-1$ times. For example, we can compute b^8 as

$$b \times (b \times (b \times (b \times (b \times (b \times (b \times b))))))$$

However, there is a more efficient way of computing b^8 using just three multiplications:

$$b^2 = b \times b$$
$$b^4 = b^2 \times b^2$$
$$b^8 = b^4 \times b^4$$

Follow this idea and implement a recursive function `exptFast`, which takes base b, exponent n and returns b^n, so that computing b^n involves at most $\log(2n)$ recursive calls to itself. You can use `even?` to test whether a number is even.

Problem 5 [18pt] Implement the following functions in Scheme using `fold-left`, `map`. DO NOT use recursive definition for this problem.

a) (6pt) Define a function `allTrue`, which takes a list of Booleans and returns `#t` if all of them are true; `#f` if one of them is false. For example, `(allTrue '(#t #t))` should return `#t`, and `(allTrue '(#f))` should return `#f`. For your convenience, `(allTrue '())` is defined as `#t`.

b) (6pt) Define a function `sum`, which takes a list of numbers and returns the sum of them; takes a list of strings and returns the concatenation of them; or takes a list of lists and returns the concatenation of them. For example, `(sum '(1 2 3 4))` should return 10, `(sum '("1" "2" "34"))` should return "1234", and `(sum '((1 2) (3 4)))` should return (1 2 3 4) (Use `string-append` for string concatenation).

c) (6pt) Define a function `zip`, which takes a list of several lists with the same length, and returns another list of lists where the n-th list is composed by the n-th elements of each list. For example, `(zip '((1 2) (3 4)))` should return ((1 3) (2 4)).