**CS 2500, Spring 2014**
**Problem Set 12**

---

**Due date: 7pm Friday April 11, 2014**

**Programming Language:** Intermediate Student Language with Lambda

You **must** follow the design recipe. The graders will look for data definitions, signatures, purpose statements, examples/tests, and properly organized function definitions. For the latter, you **must** design templates. You do not need to include the templates however. If you do, make sure to comment them out.

---

**Problem 1**.
A number, $n>1$, is prime if it is not divisible by any numbers besides 1 and itself, such as 2, 3, 5 and 7.

**a)** Design the function `prime?` which consumes a Natural Number and returns true if it is prime and false otherwise. Use the fact that if *n* is not prime, one if its factors must be less than or equal to the square root of *n*.

**b)** Design the function `list-primes` which consumes a Natural Number, *n*, and produces the list of prime numbers up to *n*.

**Problem 2**.
A palindrome is a word, number or phrase that reads the same forward and backward.

**a)** Design the function `make-palindrome`, which consumes a non-empty String and constructs a palindrome by mirroring the String around the last letter. For example, `(make-palindrome "fundies")` should produce "fundieseidnuf".

**b)** Design the function `is-palindrome?,` which consumes a non-empty String and determines whether the String is a palindrome or not.

**Problem 3**.
The Fibonacci numbers, which are the numbers in the following sequence:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

are defined by the recurrence relation: $F_n = F_{n-1} + F_{n-2}$, with seed values $F_0 = 0$ and $F_1 = 1$.

**a)** Design the function `fibonacci`, (without accumulators) which consumes a Natural Number and produces its Fibonacci number. For example, `(fibonacci 11)` should produce 89.

**b)** Use accumulators to design a more efficient version of `fibonacci`.

**c)** Design a function, `list-fibonacci`, that consumes a Natural Number and produces the list of Fibonacci numbers from $F_0$ to $F_n$.

**Problem 4.**

a) Many card games are structured around the taking of tricks, a set of four cards where one card is contributed from each player. Examples of trick-taking games include hearts, bridge and whist.

Define a function that takes a trick and returns the winning player according to the following rules:

1. High card wins
2. In the case of a tie i.e. two or more cards of the same value are played the trick goes to the player with the better suit. (Clubs < Diamonds < Hearts < Spades)

Note that the problem is purposely left ambiguous in terms of data definitions and requirements. It is up to you to decide how to implement this problem. There is not one right answer, but some answers are better and simpler than others.

Make sure to follow the design recipe.

b) A game is a series of tricks. Design a function that takes a game and returns a winner. The winner is the person who won the most tricks overall.

**Problem 5.**

a) Develop data definitions for binary trees of `Symbols` and binary trees of `Numbers`. The numbers and symbols should occur at the leaf positions only.

Create two instances of each, and *abstract* over the data definitions.
Design the function `height`, which consumes any binary tree of either type and computes its height. That is, the maximum number of `nodes` from the root of the given tree to any leaf (including the leaf, but not including the root node in the count). Here are some tests to further explain:

```
(check-expect (height 5) 0)
(check-expect (height (make-node 'yes (make-node 'no
                                         'maybe))) 2)
```

b) A leafy binary tree is a binary tree with the symbol `'leaf` at its leaves.

Design a function that consumes a natural number `n` and creates (a list of) all leafy binary trees of height `n`. Hint: Design a function that consumes a natural number `n` and creates (a list of) all leafy binary trees of height *equal or less than* `n`.
For instance, `(all-bt 2)` should produce

```
(list
    (make-node (make-node 'leaf 'leaf)
               (make-node 'leaf 'leaf))
    (make-node (make-node 'leaf 'leaf) 'leaf)
    (make-node 'leaf (make-node 'leaf 'leaf)))
```