Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.826 Principles of Computer Systems

PROBLEM SET 1

Issued: Tuesday, February 5, 2002 **Due: Tuesday, February 12, 2002**

The following problems are intended to help you understand the notions of specification and implementation. Show that you can use the high-level features of `Spec` effectively, and don't worry about getting every detail of syntax exactly right. You will need to familiarize yourself with the `Spec` language before writing the solutions. The best source is Handout 3 (Introduction to `Spec`), plus Section 9 of Handout 4 (`Spec` Reference Manual); you should not need to read the rest of Handout 4. You will also need to understand what it means to implement a spec. Handout 3 explains this, and Handout 5 gives some more examples of specs and implementations.

In each problem you are expected to write a) specification of the problem and b) implementation of the specification. You should write both specification and implementation in the `Spec` language. Try to avoid using loops or recursion to write the specs.

Each *specification* should be a `Spec` function (`FUNC`) that accepts input and output values and returns a `Bool` value. The specification should formalize the condition between the input and output values of the problem. Make your specification as clear as possible taking advantage of the mathematical expressions and the nondeterminism in the `Spec` language. Do not worry about the implementability or the efficiency in this part of the problem, instead focus on clarity. If you need an algorithm to do one of the implementations, look it up in an algorithms book.

Each *implementation* should be a `Spec` atomic procedure (`APROC`) that accepts input values and returns the output values. Each implementation in this problem set should depend only on the parameters explicitly passed as arguments to `APROC` and not on other parts of the program state. Make your implementation deterministic: when called with same arguments, the procedure should return the same result. Your goal in this part is to capture the essential idea of a deterministic implementation that could easily be translated to a language such as sequential Java.

There are four problems in this problem set; please turn in each problem on a separate sheet of paper. Also give the amount of time you spend on each problem.

Problem 1: Matrix Multiplication

A product of two matrices $[a_{ij}]$ and $[b_{ij}]$ is a matrix $[c_{ij}]$ with $c_{ij} = \sum_k a_{ik} b_{kj}$. We can represent square integer matrices in `Spec` as functions from indices to integers:

```
TYPE  Range    = IN 0 .. n-1
      Matrix   = (Range,Range) -> Int
```

a) Write a `Spec` function `isMatMul` that accepts matrices `ma`, `mb`, `mc` and tests whether the matrix `mc` is a product of the matrices `ma` and `mb`.

```
FUNC isMatMul(ma: Matrix, mb: Matrix, mc: Matrix) -> Bool = ...
```

Do not use loops or recursion in this specification.

b) Write a `Spec` atomic procedure `MatMul` that uses loops to compute the product of two matrices.

```
APROC MatMul(ma: Matrix, mb: Matrix) -> Matrix = ...
```

Problem 2: Distribution of Prime Numbers

Let x and y be positive integers.

We say that y is a divisor of x if the division of x by y yields no remainder.

We say that x is a *prime number* if $x > 1$ and the set of the divisors of x is $\{1, x\}$.

Theorem [Chebyshev] For every integer $n > 1$ there exists a prime number p such that $n < p < 2n$.

a) Write a `Spec` function `isPrimeBetween` that checks if p is a prime number between n and $2n$.

```
FUNC isPrimeBetween(p: Int, n: Int) -> Bool = ...
```

b) Write a `Spec` atomic procedure `primeBetween` that, given n, finds a prime p such that $n < p < 2n$.

```
APROC primeBetween(n: Int) -> Int = ...
```

c) Show an example of positive integers n and p such that `isPrimeBetween(p,n)` is true, but your implementation in the *b*) part never returns p when called with `primeBetween(n)`.

Problem 3. Shortest Path

A *directed graph* is a binary relation on some set. See Section 9 of Handout 4 for the definitions of some graph operations. Let graph nodes be defined as

```
TYPE Node = IN 1 .. n
```

a) Write a `Spec` function `isShortestPath` that tests whether `path` is the shortest path from `n1` to `n2` in the graph `g`.

```
FUNC isShortestPath(g: Graph[Node].G,
                    n1: Node, n2: Node,
                    path: SEQ Node) -> Bool = ...
```

b) Write a `Spec` procedure `shortestPath` that given a graph `g` and two nodes `n1` and `n2` finds a shortest path from node `n1` to node `n2`. Make sure that your solution implements the specification in part *a*), is deterministic, and has polynomial time complexity.

```
APROC shortestPath(g: Graph[Node].G,
                   n1: Node, n2: Node) -> SEQ Node = ...
```

c) Show an example of a graph `g`, nodes `n1` and `n2` and path `path` such that the value of the expression `isShortestPath(g,n1,n2,path)` is true, but your implementation in part *b*) never returns `path` as a result if invoked with the procedure call `shortestPath(g,n1,n2)`.