Problem 1: Matrix Multiplication

a) Each element $mc(i,j)$ of the matrix is equal to a sum of products. We calculate the sum by generating the sequence of the elements in the sum and then folding the elements of the sequence using the $+:$ construct.

```plaintext
FUNC isMatMul(ma: Matrix, mb: Matrix, mc: Matrix) -> Bool =
<< RET (ALL i: Range | ALL j: Range | mc(i,j) = +: { k :IN 0 .. n-1 | ma(i,k)*mb(k,j) }) >>
```

b) The following implementation corresponds to an implementation of matrix multiplication in a conventional imperative language.

```plaintext
APROC MatMul(ma: Matrix, mb: Matrix) -> Matrix =
<< VAR mc: Matrix |
  VAR i: Int := 0 |
  DO i < n =>
    VAR j: Int := 0 |
    DO j < n =>
      VAR sum: Int := 0 |
      VAR k: Int := 0 |
      DO k < n =>
        sum := sum + ma(i,k)*mb(k,j);
        k := k+1
      OD;
      mc(i,j) := sum;
      j := j + 1
    OD;
    i := i+1
  OD;
  RET mc
>>
```

Problem 2: Distribution of Prime Numbers

a) The following `Spec` function closely follows the mathematical definitions of prime numbers. (Operator `//` denotes the remainder in division of integers.)

```plaintext
FUNC isPrime(p: Int) -> Bool =
  (p > 1) \/
  { n:Int | n > 0 \ p // n = 0 } = {1, p}

FUNC isPrimeBetween(p: Int, n: Int) -> Bool =
  isPrime(p) \ n < p \ p < 2*n
```
This is a simple-minded implementation of the specification in the previous part. The atomic procedure `primeBetween` does a linear search for prime numbers from \(n + 1\) to \(2n - 1\) and returns the least number that is prime. The primality test is implemented in the `isPrimeImpl` atomic procedure by a linear search that attempts to find the smallest factor \(k\) of \(p\) where \(2 < k \leq \sqrt{p}\).

```plaintext
APROC isPrimeImpl(p: Int) -> Bool =
<<
  VAR k: Int := 2 |
  DO (k*k <= p) =>
    IF (p // k = 0) => RET false [*] SKIP FI;
    k := k+1
  OD;
  RET true
>>

APROC primeBetween(n: Int) -> Int =
<<
  VAR x: Int := n+1 |
  DO x < 2*n =>
    IF isPrimeImpl(x) => RET x [*] x := x+1 FI
  OD
>>
```

c) For example, let \(n = 7\) and \(p = 13\). Procedure `primeBetween` returns always 11, never 13.

### Problem 3. Shortest Path

*a) The shortest path predicate considers the set of all paths from \(n_1\) to \(n_2\) and then ensures that `path` has the minimum length.*

```plaintext
FUNC isPathFromTo(g: Graph[Node].G, n1: Node, n2: Node, path: SEQ Node) -> Bool =
  g.paths(path) /\ 
  path.head=n1 /\ path.last=n2

FUNC isShortestPath(g: Graph[Node].G, n1: Node, n2: Node, path: SEQ Node) -> Bool =
  isPathFromTo(g,n1,n2,path) /\ 
  path.size = { path2: SEQ Node | isPathFromTo(g,n1,n2,path2) | path2.size }.min
```

*b) The implementation performs a breadth-first search in the graph finding the shortest distance to every reachable node from the node \(n_1\). The breadth-first search is implemented using a queue represented as a list of nodes `queue`. After reaching the target node \(n_2\), the path is reconstructed using the atomic procedure `recoverPath`. The reconstruction traverses the path backwards using the fact that `dist(path(i+1)) = dist(path(i)) + 1` on the shortest path.*

```plaintext
APROC recoverPath(g : Graph[Node].G, 
  dist : Node -> IN 0 .. n+1, 
  n2 : Node) -> SEQ Node =
<<
  IF dist(n2)=0 => RET {n2}
  [*] VAR nd: Node := { nd2: Node | 
    g(nd2,n2) \ dist(n2)=dist(nd2)+1 }.min | 
  OD
>>
```
RET recoverPath(g,dist,nd) + {nd}
FI

APROC shortestPath(g: Graph[Node].G,
   n1: Node, n2: Node) -> SEQ Node =
<<
VAR queue: SEQ Node := { n1 } |
VAR dist: Node -> IN 0 .. n+1 := (\ nd:Node | n+1 ) |
dist(n1) := 0;
DO queue.size > 0 =>
   VAR first: Node := queue.head |
   IF first=n2 => RET recoverPath(g,dist,n2)
   [*] queue := queue.tail;
   VAR succ: SEQ Node :=
      \ nd :IN 1 .. n | g(first,nd) /\ dist(nd) = n+1 } |
   queue := queue + succ;
   DO succ.size > 0 =>
      dist(succ.head) := dist(first) + 1;
      succ := succ.tail;
   OD
FI
OD;
% There is no path from n1 to n2. Procedure fails.
false => SKIP
>>

c) One of the examples is the following. Let \( n = 4, n1 = 1, n2 = 4 \), and let the graph \( g \) be
\[
g = \{(1,2), (1,3), (2,4), (3,4)\}
\]
The path \( \text{path} = [1,3,4] \) satisfies \( \text{isShortestPath}(n1,n2,\text{path}) \) but the result of \( \text{shortestPath}(g,n1,n2) \)
is always the path \( [1,2,4] \).