```
// a)
method ComputeFusc(N: int) returns (b: int)
  requires N >= 0                                    // P
  ensures b == fusc(N)                               // R
{
  b := 0;
  var n, a := N, 1;

  assert
    0 <= n <= N;                                // J

  assert
    fusc(N) == a * fusc(n) + b * fusc(n + 1);  // J

  while (n != 0)                                        // B
    invariant 0 <= n <= N                               // J
    invariant fusc(N) == a * fusc(n) + b * fusc(n + 1)  // J
    decreases n // D
  {
    ghost var d := n; // D

    assert
      fusc(N) == a * fusc(n) + b * fusc(n + 1); // J

    assert
      n != 0; // B
    assert
      (n % 2 != 0 && n % 2 == 0) || fusc(N) == a * fusc(n) + b * fusc(n + 1);
    assert
      (n % 2 != 0 || n % 2 == 0) ==> fusc(N) == a * fusc(n) + b * fusc(n + 1);

    assert
      n % 2 != 0 || fusc(N) == a * fusc(n) + b * fusc(n + 1);
    assert
      n % 2 == 0 || fusc(N) == a * fusc(n) + b * fusc(n + 1);

    assert
      n % 2 == 0 ==> fusc(N) == a * fusc(n) + b * fusc(n + 1);
    assert
      n % 2 != 0 ==> fusc(N) == a * fusc(n) + b * fusc(n + 1);

    if (n % 2 == 0)
    {
      assert
        fusc(N) == a * fusc(n) + b * fusc(n + 1);   // J

      rule3(n/2);
      assert
        fusc(n/2) == fusc(n);

      assert
        fusc(N) == a * fusc(n/2) + b * fusc(n + 1);

      assert
        fusc(N) == a * fusc(n/2) + b * fusc(n/2) + b * fusc(n + 1) - b * fusc(n/2);

      assert
        fusc(N) == a * fusc(n/2) + b * fusc(n/2) + b * (fusc(n + 1) - fusc(n/2));

      rule4(n/2);
      assert
        fusc((n/2) + 1) == fusc(n + 1) - fusc(n/2);
```

```
    assert
      fusc(N) == a * fusc(n/2) + b * fusc(n/2) + b * fusc((n/2) + 1);

    assert
      fusc(N) == (a + b) * fusc(n/2) + b * fusc((n/2) + 1);

    a := a + b;

    assert
      fusc(N) == a * fusc(n/2) + b * fusc((n/2) + 1);

    n := n / 2;

    assert
      fusc(N) == a * fusc(n) + b * fusc(n + 1);
  } else {
    assert
      fusc(N) == a * fusc(n) + b * fusc(n + 1);    // J

    rule3((n + 1)/2);
    assert
      fusc((n + 1)/2) == fusc(n + 1);

    assert
      fusc(N) == b * fusc(((n - 1)/2) + 1) + a * fusc(n);

    assert
      fusc(N) ==
        b * fusc(n) - b  * fusc(n) + b * fusc(((n - 1)/2) + 1) + a * fusc(n);

    assert
      fusc(N) ==
        b * fusc(n) - b  * (fusc(n) - fusc(((n - 1)/2) + 1)) + a * fusc(n);

    rule4((n - 1)/2);
    assert
      fusc((n - 1)/2) == fusc(n) - fusc(((n - 1)/2) + 1);

    assert
      fusc(N) == b * fusc(n) - b  * fusc((n - 1)/2) + a * fusc(n);

    rule3((n - 1)/2);
    assert
      fusc(n-1) == fusc((n - 1)/2);

    assert
      fusc(N) == b * fusc(n) - b  * fusc(n - 1) + a * fusc(n);

    assert
      fusc(N) == b * fusc(n) - b  * fusc(n - 1) + a * fusc(n);

    assert
      fusc(N) ==                                          { simplify }
        a * fusc(n - 1) + b  * fusc(n) - b  * fusc(n - 1)
        + a * fusc(n) - a * fusc(n - 1);
                                                   { expand (b + a) * }
    assert
      fusc(N) == a * fusc(n - 1) + (b + a) * (fusc(n) - fusc(n - 1));

    rule3((n - 1)/2);
    assert
      fusc(n - 1) == fusc((n - 1) / 2);
```

```
   assert
    fusc(N) == a * fusc(n - 1) + (b + a) * (fusc(n) - fusc((n - 1)/2));

   rule3((n - 1)/2);
   assert
    fusc(n - 1) == fusc((n - 1) / 2);

   assert
    fusc(N) == a * fusc((n - 1) / 2) + (b + a) * (fusc(n) - fusc((n - 1)/2));

   rule4((n - 1)/2);
   assert
    fusc(((n - 1)/2) + 1) == fusc(n) - fusc((n - 1)/2);

   assert
    fusc(N) == a * fusc((n - 1) / 2) + (b + a) * fusc(((n - 1)/2) + 1);

   b := b + a;

   assert
    fusc(N) == a * fusc((n - 1) / 2) + b * fusc(((n - 1)/2) + 1);

   n := (n - 1) / 2;

   assert
    fusc(N) == a * fusc(n) + b * fusc(n + 1);
  }

  assert
   n < d; // D < d

  assert
   fusc(N) == a * fusc(n) + b * fusc(n + 1);  // J
 }
 assert
   fusc(N) == a * fusc(n) + b * fusc(n + 1);  // J

 assert
  n == 0; // !B

 assert
   fusc(N) == a * fusc(0) + b * fusc(0 + 1);  // J

 assert
   fusc(N) == a * fusc(0) + b * fusc(1);      // J

 rule1();
 assert
  fusc(0) == 0;

 assert
   fusc(N) == a * 0 + b * fusc(1);         // J

 rule2();
 assert
  fusc(1) == 1;

 assert
   fusc(N) == a * 0 + b * 1;                // J

 assert
  fusc(N) == b;                             // R
}
```

```
// b)
method ComputePos(num: int, den: int) returns (n: int)
  requires num > 0 && den > 0                           // P
  ensures n > 0 && num == fusc(n) && den == fusc(n + 1) // R
{
  var nu, de := 1, 1;
  n := 1;

  assert
    n == 1;

  rule2();
  assert
    fusc(n) == nu;

  rule3(n);
  assert
    fusc(n + 1) == de;

  assert
    nu == fusc(n) && de == fusc(n + 1); // J

  while !(nu == num && de == den) // B
    invariant n > 0               // J
    invariant nu == fusc(n)       // J
    invariant de == fusc(n + 1)   // J
  {
    assert nu == fusc(n);         // J
    assert de == fusc(n + 1);     // J

    var t := ComputeFusc(n+2);

    // Method Call Rule

/*
    method call
    method ComputeFusc(N: int) returns (b: int)
    requires N >= 0       // P
    ensures  b == fusc(N) // R

    WP[t := ComputeFusc(E), Q]   =
    P[N \ E] && forall b' :: R[N, b \ E, b'] ==> Q[t \ b']

    WP[t := ComputeFusc(E), Q]   =
      P[N \ (n+2)] && forall b' :: R[N, b \ (n+2), b'] ==> Q[t \ b']
      N >= 0[N \ (n+2)] && forall b' :: b == fusc(N)[N, b \ (n+2), b'] ==> Q[t \ b']
      (n+2) >= 0 && forall b' :: b' == fusc((n+2)) ==> Q[t \ b']

    // One-point rule

      (n+2) >= 0 && b' == fusc((n+2)) ==> Q[t \ b']

      (n+2) >= 0 && b' == fusc((n+2)) ==> t == fusc(n+2)[t \ b']
      (n+2) >= 0 && b' == fusc((n+2)) ==> b' == fusc(n+2)
*/
    assert
      t == fusc(n+2);

    nu, de := de, t;

    assert
      nu == fusc(n + 1);

```

```
    assert
      de == fusc(n + 2);

    assert
      nu == fusc(n + 1) && de == fusc(n + 2);

    n := n + 1;

    assert
      nu == fusc(n) && de == fusc(n + 1);
  }

  assert
    (nu == num && de == den);           // !B

  assert
    nu == fusc(n) && de == fusc(n + 1); // J

  assert
    nu == num && de == den;             // R
}
```