Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 18 September 29, 1993

Solutions to Problem Set #1

The maximum score for this problem set was 100 points. The main idea was to test your understanding of Spec syntax and semantics.

Problem 1 (10 points)

What is the final value of `s` *in the following program?*

```
TYPE S = SEQ Int
TYPE I = Int
VAR i := 3, s, s', s'' |
s  := S{2,5,11,17,3,5,7,13};
s' := S{2,11,5};
s'' := S{5,2,17,3};
s  := ((s ++ i) + s') - s''
```

The intended answer to the problem is $\{2, 5, 11, 3, 5, 7, 13, 11\}$ according to the common concept of the *difference of two multi-sets*. When the difference operation is applied to a sequence, it treats a sequence as a multiset (Handout 4, p. 17).

Unfortunately, the definition of the difference operation on a sequence is incorrect in this year's Spec Reference Manual; therefore, the answer $\{11, 7, 13, 11\}$ is also acceptable for this problem set. However, for the rest of this course, the following definition of *difference of two sequences* should be taken. **Please take note!**

From Spec Reference Manual of 92 Fall:

```
FUNC Difference(s1, s2) -> S =
        s2.size = 0 => RET s1                    % s1 - s2
    [] ~(s2.head IN s1) => RET  s1 - s2.tail
    [] s2.head IN s1 => VAR s, s' |
         s1 = s ++ s2.head + s' /\ ~(s2.head IN  s') =>
              RET  (s + s') - s2.tail
```

Comments: You are more than welcome to come up with an even better and more concise specification of the `Difference` operation on `SEQ T`.

Problem 2 (10 points)

Describe the input-output relation denoted by the following procedure.

```
TYPE S = SEQ Int
APROC F (i: Int, s) -> Int =
   <<   VAR s' | (s' ++ i) <<= s => RET 0
     [] BEGIN i # 0 => i := 0 [*] SKIP END; RET 1>>
```

The I/O relation is:

```
    i IN s   ⟶  0
              ↘
 ~(i IN s)   ⟶  1
```

Comments: Spec has call-by-value semantics, so the value of `i` remains unchanged after exiting the procedure. Since the call-by-value semantics of the procedural call is not stated until in some handouts after PS#1, we overlooked your guesses on the value of `i` in this problem set.

Problem 3 (10 points)

Explain how the `F` *routine above would behave differently if* `BEGIN` *and* `[*] SKIP END` *were left out?*

```
BEGIN i # 0 => i := 0 [*] SKIP END; RET 1
```

behaves very differently from

```
i # 0 => i := 0; RET 1
```

The modified line is the same as

```
i # 0 => BEGIN i := 0; RET 1 END
```

Therefore, `RET 1` is now also guarded by `i # 0`. This changes the behaviors of the whole procedure under two conditions:

1. `i IN s /\ i = 0`: Here, `0` is the only result, and it no longer behaves non-deterministically as before.
2. `~(i IN s) /\ i = 0`: Here, neither clause succeeds, and the procedure fails.

Comments: Please also refer to Handout 3 pp. 11–12 and Handout 4 p. 13 for the order of precedence of `[] [*] | => ; EXCEPT` in Spec so that you know how to parse correctly.

Problem 4 (10 points)

Describe the input-output relation denoted by the following procedure.

```
APROC G (i: Int, j: Int) -> Int =
   <<   i // 3 = 0 \/ j // 3 = 2 => RET 0
     [] i // 3 = 1 \/ j // 3 = 2 => RET 1 >>
```

The I/O relation is:

	j//3 = 0	j//3 = 1	j//3 = 2
i//3 = 0	0	0	0 or 1
i//3 = 1	1	1	0 or 1
i//3 = 2	fail	fail	0 or 1

Comments: Be sure to state explicitly and exactly when the procedure may fail and when it may return non-deterministic results.

Problem 5 (5 points)

Can the `G` routine above be a `FUNC` instead of an `APROC`? Why or why not?

No, because `FUNC` must be deterministic, and the routine above gives `0` or `1` when `j//3 = 2`.

Comments: A `FUNC` can have zero or one values for any argument, if zero, the `FUNC` is undefined at that point.

Problem 6 (5 points)

For what initial values of `x` is the following loop guaranteed to terminate? (`x` has type `Int`.) Explain briefly how the other initial values of `x` may make the loop non-terminating.

```
DO
  x < 100  => x := x + 1
 []
  x//2 = 0 => x := (x-2)**2
OD
```

An initial value of x which is greater than 100 and is an odd integer guarantees the termination of the loop.

If x is an even integer, then the second clause can be executed forever.

If `x < 100` and x is odd, then the first clause can makes x even for the next loop and the second clause can be executed forever thereafter.

Problem 7 (5 points)

Write an expression whose value is a function of type `(Int, Int -> Int)` that returns the remainder of its first argument being divided by its second argument.

The most common correct answer was

```
(LAMBDA (i:Int, j:Int) -> Int = RET i//j)
```

An even more concise answer is “`Int.//`”.

Comments: Note that a declaration with `FUNC` is not an expression in Spec.

Problem 8 (5 points)

Let `modulus` *denote the value of the expression from Problem 1. [This should have been "7".] Write an expression using a function constructor whose value is the same as* `modulus` *except when the second argument is* 0*, it maps to* -1.

This was an (unintentional) trick question, since the Spec function constructors do not, in fact, allow you to write such an expression. Full credit was given to those who realized this; partial credit was given for either coming up with a correct answer that didn't use a function constructor or coming up with a correct answer that used a plausible but invalid function constructor.

Problem 9 (40 points)

(A) (13 points)

Write a specification in Spec of a routine that finds a shortest path from node 0 to node $n - 1$ *of a graph whose nodes are labeled by integers in the range 0 to* $n - 1$*. Assume that the input graph is represented by an adjacency list* `al` *of type* `SEQ SEQ Int`*, where* `al[i]` *is a sequence whose elements are the successors of the node labeled by* `i`*. Also, assume that all edges are of unit length. The output of the routine should be of type* `SEQ Int`.

```
TYPE AL = SEQ SEQ Int
     P = SEQ Int

APROC ShortestPath (al) -> P RAISES {NoPath}
<< ~ValidGraph(al) => HAVOC
   [*] VAR p | IsPath(p,al)
                /\ (ALL p' | IsPath(p',al) ==> p.size <= p'.size)
               => RET p
              [*] RAISE NoPath >>

FUNC ValidGraph(al) -> Bool
  RET (ALL l: Int, m: Int | al(l)(m) >= 0 /\ al(l)(m) < al.size)

FUNC IsPath(p,al) -> Bool
  RET p.head = 0
   /\ p.last = al.size - 1
   /\ (ALL i: Int | 0 <= i /\ i < p.size - 1 ==> p(i+1) IN al(p(i)))
```

Common errors here included:

- Neglecting the possibility of `SEQ SEQ Int`'s which don't denote graphs, or of graphs without paths through them (especially the latter).
- Giving an overly detailed specification. If you found that there was little difference between your specification and your implementation in this problem, then either your specification was overly detailed or your implementation was under-specified.

(B) (5 points)

What type should your routine be? (`FUNC`*,* `APROC`*, or* `PROC`*?)*

Although `ShortestPath` doesn't change the global state, it's nondeterministic, returning any one of a possible many shortest paths, so it can't be a `FUNC`. Given that there's no clear reason not to make the routine atomic, and that atomic procedures are easier to reason with, it makes the most sense to make `ShortestPath` an `APROC`.

(C) (5 points)

Describe in words the input-output relation denoted by the specification you wrote in part (A).

The relation can be described as the set of *all* pairs `(al,p)` (of types `AL` and `P+{NoPath}` above) such that either

1. `al` denotes a valid graph and `p` is a shortest path through it, or
2. `al` denotes a valid graph with no path through it p is the exception NoPath, or
3. `al` does not denote a valid graph

Comments: Regardless of the sort of style of description used, it was an extremely common error to not realize (or, at least, not make it clear) that the relation maps a given graph to *all* of its shortest paths, not merely a single arbitrary one.

(D) (12 points)

Write an implementation in Spec of the module you specified in part (A).

Since the graphs here are unweighted, a simple breadth-first search (as given in Cormen, Leiserson and Rivest, for example) is sufficient. Here is a sample implementation:

```
TYPE Queue = SEQ Int

APROC ShortestPath(al) -> P RAISES {NoPath}
<< VAR n : Int                                  % next node to examine.
       queue := Queue{0}                        % nodes to examine.
       parents := P.fill(-1, al.size) |         % parents in a shortest
     DO ~queue.isEmpty =>                       %    path.
        n := queue.head;
        queue := queue.tail;
        VAR i := 0 |
           DO i < al(n).size =>
              << parents(al(n)(i)) = -1 =>
                  => parents(al(n)(i)) := n;
                     queue := queue ++ al(n)(i)
                  [*] SKIP >>;
              << al(n)(i) = al.size-1 =>
                  => RET UnwindPath(parents)
```

```
                    [*] SKIP >>
              OD
        OD;
        RAISE NoPath >>

  APROC UnwindPath(p) -> P
  << VAR i := p.size-1
           i' := p.size-1
           p' := SEQ Int.fill(0,p.size) |
        DO i # 0 =>
            p'(i') := i;
            i := p(i);
            i' := i' - 1
        OD;
        RET p'.sub(i,p'.size-1) >>
```

(E) (5 points)

Describe in words the input-output relation denoted by the implementation you wrote in part (D).

The relation can be described as the set of *all* pairs `(al,p)` (of types `AL` and `P+{NoPath}` above) such that either

1. `al` denotes a valid graph and `p` is the unique shortest path through it which comes first in a breadth-first search of `al` starting at node 0, or
2. `al` denotes a valid graph with no path through it p is the exception `NoPath`, or
3. `al` does not denote a valid graph

Note the difference from the answer to part (C): Since the implementation is deterministic, at most one path can be returned for any graph.