SWEN421 – Lecture 4
Contracts and Correctness

Recap - Contracts for subprograms

- Subprograms are specified in terms of:
  - Precondition defining allowable inputs
  - Postcondition defining allowable outputs for any allowable input
- procedure Sqrt(x: in Integer; z: out Integer)

```
  with Pre  =>  x >= 0,
       Post  =>  z**2 <= x and (z+1)**2 > x;
```

- function Max(x, y: in Integer) return Integer

```
  with Post  =>  (Abs’Result = x or Abs’Result = y) and
                 Abs’Result >= x and Abs’Result >= y
```

A note on nondeterminism

- Quite often the postcondition does not specify a unique output for all possible inputs. In the case we say the operation is nondeterministic.
- E.g. Sqrt may allow a positive or negative square root.
- Ex: Does my spec for Sqrt allow a negative square root?
  If not, modify it so that it does.
- Searching for the position of a value in an array may have different possible results if the value can occur more than once.
- There can be many possible results for a topological sort of an acyclic directed graph (i.e. list the node values so that every node is followed by all of its successors/preceded by all of its predecessors).

Recap - Contracts for subprograms

- ```
  procedure Insert1(x: in Integer; a: in array(<>) Integer; b: out array(<>) Integer)
    with Post  =>  b'Length = a'Length+1;
  ```

- ```
  procedure Insert2(x: in Integer; a: in array(<>) Integer ; b: out array(<>) Integer)
    with Pre  =>  (for all i in a'First .. a'Last-1 => a(i) <= a(i+1)),
            Post  =>  (for all i in b'First .. b'Last-1 => b(i) <= b(i+1));
  ```

- ```
  procedure Insert3(x: in Integer; a: in array(<>) Integer ; b: out array(<>) Integer)
    with Pre  =>  (for all i in a'First .. a'Last-1 => a(i) <= a(i+1)),
            Post  =>  (for all i in b'First .. b'Last-1 => b(i) <= b(i+1)) and
                      (for some k in b'Range =>
                        (for all i in b'First .. k-1 => b(i) = a(i)) and
                        b(k) = x and
                        (for all i in k+1 .. b'Last => b(i) = a(i-1)))
  ```
  ```

Making contracts readable

- Use functions and other notation to make contracts more concise!!
- Bertrand Meyer: Don’t write quantifiers in contracts.
- `function asc(a: in array(<>) Integer) return Boolean is …;`
- procedure Insert2:

```
procedure Insert2(x: in Integer; a: in array(<>) Integer ; b: out array(<>) Integer)
  with Pre  =>  asc(a),
           Post  =>  asc(b);
```

- procedure Insert3:

```
procedure Insert3(x: in Integer; a: in array(<>) Integer ; b: out array(<>) Integer)
  with Pre  =>  asc(a),
           Post  =>  asc(b) and
                        (for some k in b’Range =>
                           b(b’First .. k-1) = a(a’First .. k-1) and
                           b(k) = x and
                           b(k+1 .. b’Last) = a(k .. a’Last))
```

Check notation!!

Exercise

Write specifications for functions/procedures to:

- Determine whether a given value is in an array
- Delete a give value from an array, assuming that it is there
- Concatenate two ordered arrays, giving a new ordered array (what precondition does this need?)
- Merge two ordered arrays to give a new ordered array (what precondition does this need?)

Proving correctness

- We can prove correctness properties of programs using the specifications for the operations they invoke.
- Suppose procedure p has precondition P and postcondition Q, and has no parameters.
- To show that a call on p establishes postcondition Q’ provided precondition P’ holds beforehand, written {P’} p {Q’}, we must show:
  - P’ -> P   precondition of call implies precondition of p
  - Q -> Q’   postcondition of p implies postcondition of call
- To add parameters, just need to substitute for the in P and Q.

Proving correctness - Example

- If a is initially empty, and we insert 3, 1 and 2,
  ie: Insert(3,a,b); Insert(1,b,c); Insert(2,c,d);
  we should be able to show that the result is (1,2,3).
- In principle, we need to find an assertion that holds after each operation.
- Insert(3,a,b); pragma Assert(b = (3));
  Insert(1,b,c); pragma Assert(c = (1,3));
  Insert(2,c,d); pragma Assert(d = (1,2,3));

Proving correctness

- To prove correctness of a function/procedure body, we need rules for proving correctness of different kinds of statements.
- Ada can work out most of them, but it helps to know the ideas.
- To prove {P} x := e {Q},
  prove P -> Q[e/x]   (i.e. P implies Q with x replaced by e)
- To prove {P} S1; S2 {R},
  find Q such that you can prove {P} S1 {Q} and {Q} S2 {R}.
- To prove {P} if B then S1 else S2 end if {Q}
  prove {P and B} S1 {Q} and {P and not B} S2 {Q}.

Loop invariants

- Loops are tricky because you don’t know how many times the body will be executed!
- A loop invariant is a special kind of assertion is one that holds each time around a loop - Ada can’t (usually?) work them out!