# Solutions

1. True/False [10 pts]

   (parts a–e; **4** points off for each wrong answer, 2 points off for each blank answer, minimum problem score 0.)

   (a) SML variables have lexical scope.

   *True*

   (b) A value is polymorphic if it has a single type.

   *False*

   (c) Weak specifications should be avoided.

   *False. A lot of people got this one wrong. In general a specification should be just strong enough to allow users to do what they need. A weak specification gives the implementer flexibility.*

   (d) Both 'requires' and 'checks' clauses state preconditions.

   *True. It is an error for a caller to violate the condition stated in either a requires or a checks clause.*

   (e) Higher-order functions are functions that return types rather than ordinary values.

   *False*

2. Abstraction [30 pts] (parts a–e)

   The following is the interface and implementation of a data abstraction:

```
signature SORTED_SEQUENCE = sig
  (* A "sequence" is a sorted sequence: a possibly empty
   * sequence of integer elements [m1, m2, ... mn] in
   * ascending (nondecreasing) order.
   * Examples: [~1,2,3], [0,2,3,3], []
   *)
  type sequence
  (* empty() is an empty sequence. *)
  val empty : unit -> sequence
  (* add(s,n) is the sequence containing n and all the
   * elements in s. *)
  val add: sequence*int -> sequence
  (* prepend([m1,...,mn], m') is the sequence [m', m1,...mn]. *)
  val prepend: sequence*int -> sequence
  (* nth(s,n) is the nth element of s (indices start at zero). *)
  val nth: sequence*int -> int
  (* size(s) is the number of elements in s. *)
  val size: sequence -> int
end
structure SortedSeq : SORTED_SEQUENCE = struct
  type sequence = int * int list
  fun empty() = (0, [])
  fun add((sz, lst), m) = raise Fail "Implement me!"
  fun prepend((sz, lst), m) = (sz+1, m::lst)
  fun nth((sz, lst), n) = List.nth(lst, n)
  fun size(sz,lst) = sz
end
```

(a) [8 pts] Two of these methods need requires clauses. Which are they? For each method, explain why a requires clause is needed, and give a requires clause that addresses the problem you have identified.

**Answer:**

`prepend`*: The output according to specification is not a sorted sequence unless* `m` *is no larger than any element in the sequence. Needs a requires clause, "Requires: no element in sequence is smaller than* `m`*".*
`nth`*: The argument* `n` *could be out of bounds. Need a requires clause, "Requires:* $0 \leq$ `n` $\leq$ `size(s)` $- 1$*".*
*Many people identified the right preconditions, but gave a subtly wrong reason. They wanted to change the specification because the implementation couldn't handle those inputs. The real reason that the spec needs to be changed is that in both cases the spec does not make sense when considered for the bad inputs. For example, they noted that allowing prepend to accept a number greater than the first element would break the rep invariant. The real problem is that* $[m', m_1, ..., m_n]$ *would not be a sorted sequence in that case.*

(b) [4 pts] The implementer has failed to specify an abstraction function. Give a specification of the abstraction function that this implementation uses.

**Answer:**

*The representation* `(sz,[m1,...,mn])` *represents the ordered sequence* $[m_1, \ldots, m_n]$.

(c) [4 pts] The implementer has failed to write a representation invariant. Give a representation invariant that is strong enough to prove that all of the implemented functions work correctly. (Do not give the proofs.)

**Answer:**

*Representation invariant: given a representation* `(sz, lst)`, `sz` *is the length of the list* `lst` *and the elements of* `lst` *are in ascending order.*

(d) [8 pts] The `add` function is unimplemented. Write an implementation that satisfies the given specification and preserves the representation invariant.

```
let fun ins(lst, m) =
  case lst of
    nil => [m]
  | h::t => if h > m then m::lst
                     else h::(ins(t, m))
in
  (sz+1, ins(lst, m))
end
```

(e) [6 pts] One test case for the function `add` is `add(`$[1, 2, 4, 5]$`, 3)`. Give up to eight more good black-box test cases for the function `add`, which collectively provide good coverage. For each test case, write three or four words explaining how it improves coverage. You will lose points for providing redundant test cases.

**Note:** in the example test case, $[1, 2, 4, 5]$ represents a sequence, not an SML list, which is hinted at by its font. For brevity, use this list-like notation in your test cases.

**Answer:**

`add(`$[1, 3, 5]$`, 0)` *Add at beginning*
`add(`$[1, 3, 5]$`, 6)` *Add at end*
`add(`$[1, 3]$`, 1)` *Add duplicate*
`add(`$[]$`, 1)` *Add to empty list*
`add(`$[1]$`, 1)` *Add to one-element list*
`add(`$[1, 1, 1]$`, 1)` *Add duplicate of repeated element*

*Other useful tests include adding the maximum and minimum integers, and checking whether negative numbers are handled correctly.*

3. Zardoz [20 pts] (parts a–d)

For each of the following expressions, give a *value* that causes the expression to evaluate

to 42 if the box $\square$ is replaced by that value. A list of values $[v_1, \ldots, v_n]$ is considered a value.

(a) [5 pts]

```
let val f: string list = □
in
 case List.map(fn(x) =>
                case Int.fromString(x) of
                  SOME n => n
                | _ => 0) f of
   x::y::_ => x*10 + y
 | _ => 41
end
```

**Answer:** `["4", "2 "]`

(b) [5 pts]

```
let fun zardoz(x:int):int list =
      if x = 0 then [] else x::zardoz(x-1)
in
  case zardoz( □ ) of
    w::x::y::z => x*y
end
```

**Answer:** `8`

(c) [5 pts]

```
let fun zardoz(b: bool list):int =
      case b of
        nil => 0
      | true::b' => 1 + 2*zardoz(b')
      | false::b' => 2*zardoz(b')
in
  zardoz □
end
```

**Answer:** `[false, true, false, true, false, true]`

(d) [5 pts]

```
let val f = □ in
  17 + (#2 (f(12, "hello"))) + (#1 (f("goodbye", 13)))
end
```

**Answer:** `fn(x,y) => (y,x)`

4. Correctness and Evaluation [30 pts] (parts a–e)

Consider the following implementation of a factorial function:

```
fun f(m,n) = if n = 1 then m else f(m*n, n-1)

(* fact(n) is n! *)
fun fact(n) = f(1,n)
```

(a) [3 pts] Critique the specification of the function `fact`. How would you change this specification to make it better?

**Answer:**

*This code will only work for* `n` *at least 1, but the specification doesn't say so. It should have a requires clause:* `Requires: n>=1`

(b) [6 pts] Show each of the evaluation steps involved in the evaluation of `fact(3)` according to the substitution model.

**Answer:**

```
    fact(3)
--> f(1,3)
--> if 3=1 then 1 else f(1*3,3-1)
--> if false then 1 else f(1*3,3-1)
--> f(1*3,3-1)
--> f(3,3-1)
--> f(3,2)
--> if 2=1 then 3 else f(3*2,2-1)
--> if false then 3 else f(3*2,2-1)
--> f(3*2,2-1)
--> f(6,2-1)
--> f(6,1)
--> if 1=1 then 6 else f(6*1,1-1)
--> if true then 6 else f(6*1,1-1)
--> 6
```

(c) [6 pts] The function `f` has no specification above. Complete the following specification in such a way that `f` satisfies it *and* it is strong enough to argue that `fact` computes the factorial of its argument.

```
(* f(m,n) is m*n!.
 * Requires: n>0
 *)
```

(d) [3 pts] Using the specification you just wrote in 4(c), argue that `fact` satisfies its own specification. In your argument, rely only on the specification of `f` and do not use any other information about the implementation of `f`.

**Answer:**

`fact(n)` *evaluates to* `f(1,n)`*, which according to the specification for* `f` *is* $1 \cdot$ `n!` $=$ `n!`*. Therefore* `fact` *is correctly implemented assuming* `f` *implements its specification.*

(e) [12 pts] Prove by induction that `f` satisfies the specification of 4(c). Make sure to perform all of the steps involved in a proof by induction.

**Answer:**

i. *Proposition:* `f(`$m$`,`$n$`)` *computes* $m \cdot n!$ *for* $n \geq 1$. *Proved by induction on* $n$.

ii. *Base case: Consider arbitrary* $m$. *Then,* `f(`$m$`,1)` $\longrightarrow$`if 1=1 then` $m$ `else f(`$m$`*1,1-1)` $\longrightarrow$`if true then` $m$ `else f(`$m$`*1,1-1)` $\longrightarrow m$. *This is correct because* $m = m \cdot 1!$

iii. *Induction hypothesis:* `f(`$m$`,`$n$`)` *computes* $m \cdot n!$

iv. *Induction step: Want to show* `f(`$m$`,`$\underline{n+1}$`)` *computes* $m \cdot (n+1)!$. *Proceed by evaluation. We will underline terms such as* $\underline{n+1}$ *to indicate that they represent simple SML integers which depend on* $m$ *or* $n$, *and are not arithmetic expressions to be evaluated in the program..*
`f(`$m$`,`$\underline{n+1}$`)` $\longrightarrow$`if` $\underline{n+1}$`=1 then` $m$ `else f(`$m$`*`$\underline{n+1}$`,`$\underline{n+1}$`-1)` $\longrightarrow$`if false then` $m$ `else f(`$m$`*`$\underline{n+1}$`,`$\underline{n+1}$`-1)` $\longrightarrow$`f(`$m$`*`$\underline{n+1}$`,`$\underline{n+1}$`-1)` $\longrightarrow$`f(`$\underline{m \cdot (n+1)}$`,`$\underline{n+1}$`-1)` $\longrightarrow$`f(`$\underline{m \cdot (n+1)}$`,`$n$`)`
*By the induction hypothesis, this evaluates to* $m \cdot (n+1) \cdot n!$, *which is equal to* $m \cdot (n+1)!$ *as desired.*

v. *Conclusion: By induction,* `f(`$m$`,`$n$`)` *computes* $m \cdot n!$ *for* $n \geq 1$.

5. Complexity [10 pts] (parts a–e)

For each of the following pairs of functions $f(n)$ and $g(n)$, circle the one that is of the asymptotically largest order of growth, or write SAME if they are of the same order of growth.

(a) [2 pts]

$$f(n) = n, \qquad g(n) = n \lg n$$

**Answer:**
$n \lg n$

(b) [2 pts]

$$f(n) = 1000n^2 + 10000, \qquad g(n) = n^3 - 5n^2$$

**Answer:**
$n^3 - 5n^2$

(c) [2 pts]

$$f(n) = n^{2.001}, \qquad g(n) = n^2 \lg n$$

**Answer:**
$n^{2.001}$. *Recall* $n^\epsilon$ *where* $\epsilon > 0$ *is asymptotically larger than* $\lg n$.

(d) [2 pts]

$$f(n) = n^2 + n \lg n, \qquad g(n) = 2n^2 + n$$

**Answer:**

*SAME. Both are* $\Theta(n^2)$

(e) [2 pts]

$$f(n) = \lg^2 n, \qquad g(n) = \lg n$$

**Answer:**

$\lg^2 n$ *is a factor of* $\lg n$ *larger.*