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) In a data abstraction, a given abstract value may be represented by more than one concrete value.

   True

   (b) In SML, unbound variables are always detected at compile time.

   True

   (c) Polymorphism is the language feature that provides recursive datatypes.

   False

   (d) The `foldl` function is tail-recursive.

   True

   (e) Black-box test cases are written without reading the implementation that is being tested.

   True

2. Specifications [20 pts] (parts a–e)

   Consider the following specifications and code for functions named `f` of type `int*int->int`:

   (a)
   ```
   (* f(x,y) = x + 1.
    * Requires: x >= 0 *)
   ```

   (b)
   ```
   (* f(x,y) >= 0.
    * Requires: x >= 0 & y >= 0 *)
   ```

   (c)
   ```
   (* f(x,y) = x + y
    * Requires: x >= 0 & y >= 0 *)
   ```

   (d)
   ```
   (* f(x,y) >= 0
    * Checks: x > 0 *)
   ```

   (e)
   ```
   (* f(x,y) = x
    * Requires: x > y *)
   ```

   (1)
   ```
   fun f(x:int, y:int):int =
     Int.abs(x) + 1
   ```

   (2)
   ```
   fun f(x:int, y:int):int =
     x+y
   ```

   (3)
   ```
   fun f(x:int, y:int):int =
     if x < y then y else x
   ```

   (4)
   ```
   fun f(x:int, y:int):int =
     case (x,y) of
       (0,_) => y
     | _ => x
   ```

   (5)
   ```
   fun f(x:int, y:int): int =
     let fun g(z: int) =
       if z = 0
         then x
         else g(z-1)
     in
       g(y)
     end
   ```

   (6)
   ```
   fun f(x:int, y:int):int =
     let val z = (if x < 0
                  then 0
                  else x)
     in
       y div z
     end
   ```

For each of the specifications (a)–(e), identify the names of the function declaration or declarations in (1)–(6) that implement it. Some specifications might not be implemented by any function; some might be implemented by more than one function; some functions might implement no specifications or might implement multiple specifications.

(a) [4 pts] 1

(b) [4 pts] 1,2,3,4,5

(c) [4 pts] 2

(d) [4 pts] none (6 fails when $y < 0$)

(e) [4 pts] 3 (5 fails when $y < 0$)

3. Evaluation [17 pts] (parts a–c)

Consider the following function definition:

```
fun f(p, xs: int list) =
  case xs of
    [] => ([], [])
  | x::t =>  let val (ys,zs) = (f(p, t)) in
      if x > p then (ys, x::zs) else (x::ys, zs)
    end
```

(a) [4 pts] The implementer has not been reading the CS 312 style guide carefully and have left off some type declarations. What are the types of `p`, `ys`, and `zs`, and `f`?

Answer:

`p:int, ys: int list, zs: int list, f: (int * int list) -> (int list * int list)`

(b) [5 pts] Given this function definition, what is the value that results from evaluating the expression `f(3, [2,4,1,5])`?

Answer:

`([2,1], [4,5])`

(c) [8 pts] Write a specification for the function `f` that is deterministic, definitional, and has no precondition. Be concise but accurate.

Answer:

`f(p, xs) is a pair of lists (ys, zs) where ys contains all the elements in xs that are at most p, in the order they occurred in xs, and zs contains all the elements in xs greater than p, also in the order they occurred.`

4. Data Abstraction [33 pts] (parts a–f)

The following is the interface and implementation of a data abstraction written by someone who missed the lecture on documenting implementations.

```
signature INTERVAL = sig
  (* An interval is a (possibly empty) contiguous set of integers; that is,
   * a (possibly empty) set of n integers [a,b] = {a, a+1, * a+2, ..., b}.
   * (Note that [a,b] is ordinary mathematical notation, NOT an SML list!)
   * Examples: ∅, [0,2], [~2,~1], [~1,102] *)
  type interval
  (* create(x,y) is the set of integers n such that x<=n<=y. *)
  val create: int*int -> interval
  (* top(a) is the largest integer in a. *)
  val top: interval -> int
  (* bottom(b) is the smallest integer in a. *)
  val bottom: interval -> int
  (* The number of elements in the interval. *)
  val size: interval -> int
  (* join(a,b) is the smallest interval including both a and b *)
  val join: interval*interval -> interval
  (* intersect(a,b) is the largest interval included in a and b *)
  val intersect: interval*interval -> interval
end
```

```
structure Interval :> INTERVAL = struct
  type interval = int*int
  fun create(a0: int, a1:int): interval = ...
  fun size((a0,a1)) => a1 - a0 + 1
  fun top(a0,a1) = a1
  fun bottom(a0,a1) = a0
  fun intersect((a0,a1), (b0, b1)) =
    create (Int.max(a0,b0), Int.min(a1, b1))
  fun join(a,b) = ...
end
```

(a) [8 pts] Some 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:
`top` *and* `bottom` *need a precondition that the interval in question cannot be empty. Technically, the result of the* `size` *method also is undefined if the number of elements in the interval is at least* 2^{31}*, so we might rule that out with a precondition, too. Alternatively we could say that an interval can only represent intervals containing at most* $2^{31} - 1$ *elements; this would show up in the ADT overview and as a requires clause for* `create`*.*

A number of people imposed a precondition on the `create` *function in an attempt to rule out calls like* `create(5,3)`*. This wasn't necessary because the specification for* `create` *is perfectly clear about what should happen in this case: you get the empty set. Further, the people who did this often either made it impossible to create the empty set or broke the* `intersect` *implementation. Some went further and imposed new requirements on* `intersect` *and* `join`*; however, this unnecessarily made the whole abstraction much less useful.*

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

Answer:
The representation `(a0,a1)` *represents the (possibly empty) set of integers that are at least* `a0` *and at most* `a1`*.*

(c) [4 pts] Give a representation invariant that is strong enough to prove that all of the implemented functions work correctly. (Do not give the proofs.) Make sure your rep invariant agrees with the abstraction function given just previously. **Hint:** look at the function `size`.

Answer:
There is more than one right answer to this question, depending on how `create` *is implemented. Here is the weakest rep invariant that permits a working implementation:* `A representation (a0,a1) satisfies a1 > a0 - 1`.

(d) [6 pts] The `create` function is unimplemented. Write an implementation that meets the specification and ensure the rep invariant.

Answer:
There is more than one way to implement this function because there is more than one representation for the empty set with the rep invariant given above. Here is one implementation:

```
fun create(a0: int, a1:int): interval = if (a1 < a0) then (1,0) else (a0,a1)
```

(e) [6 pts] The `join` function is unimplemented. Write an implementation that satisfies the given specification and preserves the representation invariant. **Hint:** Look at `intersect`, but don't copy it.

Answer:

```
case (size(a), size(b))
    (_, 0) => a
  | (0, _) => b
  | _ => create (Int.min(a0,b0), Int.max(a1, b1))
```

(f) [5 pts] One possible test case for the function `join` is `join(`$[1,3]$`, `$[2,4]$`)` (where Roman font is used here to indicate abstract interval values rather than SML lists).

Suggest five more good black-box test cases for `join` that provide as much coverage as you can achieve. For each test case, write three or four words (a full sentence is not necessary) explaining how it improves coverage. You will lose points for providing redundant test cases.

Answer:

`join(`$[1,2]$`, `$[3,3]$`)`	*disjoint abutting, single-element set*
`join(`$[10,100]$`, `$[0,2]$`)`	*disjoint separate, higher is first arg*
`join(`$[-1,0]$`, `$[-1,0]$`)`	*same value, negative range, zero top*
`join(`$\emptyset$`, `$[0,$`MAXINT`$]$`)`	*empty set, maximum int*
`join(`$\emptyset$`, `$\emptyset$`)`	*single-element set, empty set as second arg*

Other useful tests include intervals with the minimum integer, an interval that covers all ints, intervals that contain one another, intervals that overlap in exactly one element. As long as each of your tests was sufficiently distinct from the others and you identified why, you received full credit.

5. Complexity and Induction [20 pts] (parts a–c)

Consider the following function:

```
fun f(n: int): int =
  if n <= 1
    then n
    else f(n div 2) + f((n+1) div 2)
```

(a) [6 pts] Given this function definition, show every step in the evaluation of the expression `f(3)` according to the evaluation model given in class. For each step, put an underscore under the subexpression that is being reduced. For brevity, you may use ellipses (. . .) to abbreviate unchanged parts of the expression carried over from the previous evaluation step or from the definition of `f`.

Answer:

```
f(3) ⟶if 3 <= 1 then 3 else f(3 div 2) + f((3+1) div 2)
⟶if false then 3 else f(3 div 2) + f((3+1) div 2)
⟶f(3 div 2) + f((3+1) div 2)
⟶f(1) + f((3+1) div 2)
⟶(if 1 <= 1 then 1 else f(1 div 2) + f((1+1) div 2)) + f((3+1) div 2)
⟶(if true then 1 else ...)  + f((3+1) div 2)
⟶1 + f((3+1) div 2 ⟶1 + f(4 div 2) ⟶1 + f(2)
⟶1 + (if 2 <= 1 then 2 else f(2 div 2) + f((2+1) div 2))
⟶1 + (if false then 2 else f(2 div 2) + f((2+1) div 2))
⟶1 + (f(2 div 2) + f((2+1) div 2)) ⟶1 + (f(1) + f((2+1) div 2))
⟶1 + ((if 1 <= 1 then 1 else ...)  + f((2+1) div 2))
⟶1 + ((if true then 1 else ...)  + f((2+1) div 2))
⟶1 + (1 + f((2+1) div 2))) ⟶1 + (1 + f(3 div 2)) ⟶1 + (1 + f(1))
⟶1 + (1 + (if 1 <= 1 then 1 else ...))
⟶1 + (1 + (if true then 1 else ...))
⟶1 + (1 + 1) ⟶1 + 2 ⟶3
```

(b) [8 pts] Prove using induction that `f(n) = n` for all nonnegative `n`. Be sure to state clearly what you are proving and what your induction hypothesis is.

Claim: We claim `f(`n`)` is n for all nonnegative n. We will show this using strong induction on n.

Looking at the definition of `f`, we see that neither `f(0)` nor `f(1)`, will follow the same pattern as `f` when applied to a larger integer. While strictly speaking only `f(0)` needs to be considered as a base case, `f(1)` needs to be handled specially.

First Base Case: `n = 0`

Apply the substitution model to `f(0)`

```
⟶ f(0)
⟶ if 0 <= 1 then 0 else f(0 div 2) + f((0+1) div 2)
⟶ if true then 0 else f(0 div 2) + f((0+1) div 2)
⟶ 0
```

Therefore `f(0) = 0`.

Second Case: `n = 1`

Apply the substitution model to `f(1)`

```
⟶ f(1)
⟶ if 1 <= 1 then 1 else f(1 div 2) + f((1+1) div 2)
⟶ if true then 1 else f(1 div 2) + f((1+1) div 2)
⟶ 1
```

Therefore `f(1) = 1`.

Inductive Step: Assuming the inductive hypothsis, `f(n) = n` for all $n \leq k$, where k is a fixed integer and $k \geq 1$, we want to prove that `f(`$\underline{k+1}$`) =` $\underline{k+1}$ (using the underline to distinguish between the integer value $\underline{k+1}$ and the language expression `k+1`).

Apply the substitution model to `f(`$\underline{k+1}$`)`:

```
⟶ f(k+1)
⟶ if  k+1  <= 1 then 1 else f((k+1) div 2) + f(((k+1)+1) div 2)
```

By assumption $k \geq 1$, so $k + 1 \geq 2$ and

```
⟶ if false then 1 else f((k+1) div 2) + f(((k+1)+1) div 2)
⟶ f((k+1) div 2) + f(((k+1)+1) div 2)
```

Noting that `x div 2 < x` and applying the inductive hypothesis we find

$\longrightarrow$ ((k+1) div 2) + (((k+1)+1) div 2)

In the event that $k+1$ is even we reduce to:

$\longrightarrow$ ((k+1) div 2) + (((k+1)+1) div 2)
$\longrightarrow$ (k/2) + (((k+1)+1) div 2)
$\longrightarrow$ (k/2) + ((k+2) div 2)
$\longrightarrow$ (k/2) + (k/2) + 1
$\longrightarrow$ k+1

In the event that $k+1$ is odd we reduce to:

$\longrightarrow$ (k+1 div 2) + ((k+1+1) div 2)
$\longrightarrow$ (k+1)/2 + ((k+1+1) div 2)
$\longrightarrow$ (k+1)/2 + ((k+2) div 2)
$\longrightarrow$ (k+1)/2 + (k+1)/2
$\longrightarrow$ k+1

In both $k+1$ positive and negative cases we see that f(k+1) = Vk+1. This proves the inductive case. Therefore, we find that f(n)=n for all nonnegative n.

(c) [6 pts] Give a recurrence for the running time $T(n)$ of the function. You may simplify constant factors.

Answer:

$$T(n) = c_0 + T\left(\frac{n}{2}\right) + T\left(\frac{n+1}{2}\right)$$

or simply

$$T(n) = 1 + 2 * T(n/2)$$

is good enough. The solution is $O(n)$. You can't drop the $O(1)$ term entirely because then the recurrence has a different solution: $T(n) = 0$!.