Solutions

1. True/False [18 pts]
   (parts a–i; 4 points off for each wrong answer, 2 points off for each blank answer, minimum problem score 0.)

(a) A good interface supports many operations so its users can find exactly the operation they want for a given task.
   False

(b) The representation type of an abstract data type is not visible outside the module that defines the ADT.
   True

(c) Representation invariants must be satisfied on exit from an module operation, but need not be satisfied on entry.
   False

(d) The function \( f(n) = n^2 + n + 1 \) is \( O(3n^3) \)
   True

(e) The SML expression \( \text{fn}(f) \Rightarrow f(f) \) is well-typed.
   False

(f) The SML expression \( \text{fn} x \Rightarrow \text{fn} y \Rightarrow 1 \) has a polymorphic type.
   True

(g) The Boyer-Moore string matching has worst-case run time linear in the size of its input (the text string and the pattern string).
   False

(h) Shared-memory communication between threads involves one thread imperatively modifying a ref and another thread reading from it.
   True

(i) In regression testing, we run a program backward to trace where in the program incorrect output must have come from.
   False

2. Programming Languages [17 pts] (parts a–c)
   The following defines (most of) an implementation of the Fibonacci function, without using the keyword fun:

```plaintext
val fibo =
  let
  val z = ...\(^1\)
  val f = fn x => if x <= 1 then 1.0 else ...
  in
  \( x \Rightarrow \ldots \) \((x-1) + \ldots (x-2)\)
  in
  \ldots\(^3\); f
end
```

1
(a) [2 pts] What is the type of fibo?

Answer:

```
int -> real
```

(b) [7 pts] There are three \ldots 's to be filled in to complete the code. Provide code for each of them.

You may not use the keywords `fun` or `and`.

The easiest (but not the only) way to solve this is to use a ref:

```
... 1 = ref fn(z) => 0.0
... 2 = (z)
... 3 = z := f
```

Consider the following function:

```latex
fun f(i:int):int =
  let
    val r = ref i
    val z = let
      val x = (2,r,"hello")
      val y = #2x
    in
      (* X *)
      !y
    end
    (* Garbage collection occurs here. *)
    (* X *)
  in
    z
  end
```

(c) [8 pts] Suppose we call this function as `f(3)`. Draw pictures of how memory is laid out at the two points marked “X”, assuming that the implementation uses a copying garbage collector and the collector is run at the indicated point in the program. Make sure you draw both the heap and the stack and label them clearly. If you prefer you may draw the environment instead of the stack that implements it.

3. Streams [15 pts] (parts a–b)

In class we saw a stream abstraction with the following signature:

```latex
signature STREAM = sig
  (* An 'a stream is an arbitrarily long (possibly infinite) sequence
   * of values of type 'a, e.g. [a0, a1, a2, ...] *)
  type 'a stream
  (* make(b,f) is a stream [a0, a1, a2, ...] where
   * (ai, bi) = f(bi-1) for i > 0 and (a0, b0) = f(b). *)
  val make : ('b * ('b -> 'a*'b)) -> 'a stream
  val next : 'a stream -> ('a * 'a stream)
end
structure Stream :> STREAM = struct ... end
```
(a) [5 pts] The specification for `next` is missing. Provide one:

**Answer:**

(* Given that \(a = [a0, a1, a2, ...]\), \(next(a) = (a0, [a1, a2, ...])\)*)

The `ListPair` structure is a useful part of the SML/NJ Basis Library. Let’s design something similar for streams.

(* Given that \(a = [a0, a1, a2, ...]\) and \(b = [b0, b1, b2, ...]\),
* `stream˙pair(a,b)` is a stream \([(a0,b0), (a1, b1), (a2, b2), ...]\) *)

```plaintext
val stream_pair: 'a stream * 'b stream -> ('a*'b) stream
```

(b) [10 pts] Finish the following implementation of `stream_pair` by writing code to replace the ...

```plaintext
fun stream_pair(a: 'a stream, b: 'b stream) = Stream.make((a,b),
  fn(a,b) => let
    val (x, a2) = Stream.next(a)
    val (y, b2) = Stream.next(b)
  in
    ((x,y), (a2,b2))
  end)
```

4. Miscellaneous [13 pts] (parts a–c)

(a) [6 pts] You are approaching a traffic light in your car and jot down the sequence of colors of the oncoming light that you see from the moment the light becomes visible until you pass it. This sequence forms a string containing the letters “R”, “Y”, and “G” for red, green, and yellow; for example, “G”, “RG”, “GY”, “GYRG” are all possible observations. Suppose that when you first see the light it is green and and when you last see it is green. Write a regular expression that describes the possible strings that you might have jotted down.

**Answer:**

\(G(YRG)^*\)

(b) [2 pts] Depth-first and breadth-first graph search rely on an implementation of sets. Explain briefly why.

**Answer:**

*A set is needed to keep track of the nodes already visited.*

(c) [5 pts] Suppose we choose a set implementation with worst-case run time \(O(\sqrt{n})\) for the `member` and `add` operations (on a set of size \(n\)). What effect would this choice have on the worst-case run time of breadth-first search?

**Answer:**

*Suppose that there are \(V\) nodes and \(E\) edges. BFS does \(V\) queue operations taking \(O(1)\) time each, which takes \(O(N)\) time. It also does \(E\) set insertions taking \(O(\sqrt{V})\) time each. Therefore the total run time is \(O(E\sqrt{V})\) (or \(O(V^{5/2})\)) rather than the \(O(E)\) time we’d get with an \(O(1)\) set implementation such as hash tables or just a ref attached to every graph node.*

5. The Return of Cycle Detection [37 pts] (parts a–f)

We saw in the second prelim that we could efficiently detect cycles in graphs by extending depth-first search to keep track of some extra information per node. It turns out that if every node in the graph has at most one outgoing edge, we can do much better. There are two possible structures for such a graph: it may be either a linked list or a list that contains a loop (a cycle). For convenience we will call these graphs “chains”. The two kinds of chains can be visualized as follows:
It turns out that there is an elegant, asymptotically efficient algorithm to detect cycles in such a graph. Here is the SML code:

```sml
datatype chain = nil | node of chain ref

(* looped(c) is whether there is a loop reachable from c. *)
fun looped(c: chain): bool = let
  (* looped'(c1,c2) is whether there is a loop reachable from c1.
  * Requires: If c1 is not nil, then c2 is reachable from c1
  * (c2 may be equal to c1). *)
  fun looped'(c1: chain, c2: chain): bool =
    case (c1,c2) of
      (node(r1: chain ref), node(ref(node(r2: chain ref)))) =>
        if r1 = r2 then true
        else looped'(!r1, !r2)
    | _ => false
  in looped'(c,c)
end
```

It is not immediately obvious that this algorithm works; in this problem we will show that it does by proving that `looped'` correctly implements its specification.

(a) [5 pts] The execution of this SML algorithm only requires a constant amount of memory above that taken up by the chain being traversed. Explain why this is.

**Answer:**

`looped` only calls `looped'` once. `looped'` is tail-recursive, so there only needs to be one stack frame in existence for `looped'` at any given time, storing at most the variables `c1`, `c2`, `r1`, `r2`.

(b) [5 pts] To help you understand how this algorithm works, the following figure shows the relationship between the variables `c1`, `c2`, `r1` and the result of evaluating `!r1` in the first arm of the `case` expression in `looped'`.

Finish the figure to show `r2` and `!r2`, drawing new boxes and arrows as necessary.

**Answer:**

(c) [4 pts] Draw four good test cases for the function `looped`, as box-and-arrow diagrams. You won’t be able to get complete coverage, so choose your test cases well.

**Answer:**
Now that you have some test cases, we recommend running `looped` on some of them by hand to help you understand why the algorithm works. This will be useful for the rest of the problem. There is an extra copy of the code for `looped` at the end of the exam. Feel free to remove it and keep it on the side for reference.

We can split the argument that `looped` implements its spec into two parts. First we show that it returns `false` if the chain has no loop, then we show that it returns `true` if the chain has a loop. This will involve constructing two arguments by induction. Make sure for each argument that you clearly state the property \( P(n) \) that you are trying to prove and the induction hypothesis. Argue both the base case and the induction step. We expect readable sentences, which may be clarified by the use of diagrams.

**Hint:** it may be useful to read through the rest of the problem before starting on the following parts. Remember that `looped` can only be called when its Requires clause is satisfied.

(d) [10 pts] Suppose that \( c_1 \) is the head of a chain of \( n \) nodes that does not contain a loop. Show that in this case `looped` returns `false`.

(Hint: Use induction on \( n \). Argue as the base case that the function works correctly for \( n \leq 2 \), then argue the induction step considering \( n > 2 \). You will need to make use of the Requires clause.)

**Answer:**
We are trying to show that `looped` returns `false` for all non-looped chains. Our property \( P(n) \) is that it returns `false` for all non-looped chains of length \( n \) nodes. If \( n \) is 0, 1 or 2 then either we fall into the `false` case immediately or else \( !r_2 \) is nil and the recursive call to `looped` returns `false` immediately. Now suppose we have \( n > 2 \). Our induction hypothesis is \( P(n-1) \), that it returns `false` for all non-looped chains of length \( n-1 \). If control falls into the second arm of the case the code returns `false` as desired. Now consider execution of the first arm. Because the chain has no loop and \( c_2 \) is reachable from \( c_1 \), we have \( r_1<>r_2 \). Therefore the result is `looped` applied to \( !r_1 \), which must be the head of a chain of length \( n-1 \). By the induction hypothesis this call correctly returns `false`.

A common error that we saw was to argue the base case successfully but then fail to make the inductive step. Many people gave hand-waving arguments about what “eventually” happens as the algorithm executes. The whole point of induction is that you don’t have to say anything about what happens “eventually”; instead, you break the problem down into one small, simple step at a time.

(e) [10 pts] Now we will start working on the case where the chain contains a loop. Start by proving a useful lemma: If the chain that starts at \( c_1 \) is a loop that includes \( c_1 \), then `looped` returns `true`.

(Hint: In this case it must also be possible to reach \( c_1 \) from \( c_2 \). Let \( n \) be the smallest non-zero number of edges that must be traversed to get from \( c_2 \) to \( c_1 \). Use induction on \( n \), with the base case \( n = 1 \).)

**Answer:**
\( P(n) \) is this: for all chains of length \( n \) that start at \( c_1 \) and have a loop that includes \( c_1 \), `looped` returns `true`. Because \( c_1 \) and \( c_2 \) are part of a loop, the first case arm will be executed for such a chain. Consider the base case \( n = 1 \). Then we will have \( r_1=r_2 \) and the function returns `true`. Now assume it takes \( n \) steps to get from \( c_2 \) to \( c_1 \) where \( n > 1 \). Then \( !r_1 \) advances by one node from \( c_1 \) but \( !r_2 \) advances by two nodes from \( c_2 \). So the number of steps from \( !r_2 \) to \( !r_1 \) must be \( n-1 \). The Requires clause on `looped` is trivially satisfied because all nodes in the loop are reachable from one another. Therefore by the induction hypothesis \( P(n-1) \), `looped` returns `true`.

Using the lemma of part 5(e) we can finish the argument for the case where the chain contains a loop but that loop does not necessarily contains \( c_1 \). Suppose that there is a loop reachable from \( c_1 \). Then there is some minimal number of edges that must be followed to get to a node in the loop. Call this distance \( n \). Note that \( n \) may be zero if \( c_1 \) is in the loop. We show by induction on \( n \) that `looped` returns `true` for all \( n \geq 0 \). Our base case is \( n = 0 \).
(f) [3 pts] Explain briefly why $\text{looped'}$ returns true in the base case $n = 0$.

Answer:

If $n = 0$ then the loop includes $c_1$, so by the lemma we just proved $\text{looped'}$ returns true.

The rest of the proof, for free (the easy part!): Now let’s consider the inductive step. Suppose that $n > 0$, so $c_1$ is not in the loop. Our inductive hypothesis is that for loops reachable in $n - 1$ steps the function returns true. Since $c_1$ is not in the loop, we must have $r_1 <> r_2$ and therefore the result comes from the recursive call. But the distance from $!r_1$ to the first loop node must be $n - 1$. Therefore by the inductive hypothesis the function returns true.

From part 5(d) and the argument just given, the function $\text{looped'}$ works correctly, and therefore so does $\text{looped}$.