Final Examination
Computer Science 312, Cornell University 14 May 2002

There are 5 problems on this exam. It is 16 pages long (many are blank); make sure you have the whole exam. You will have 150 minutes in which to work on the problems. You will likely find some problems easier than others; read all problems before beginning to work, and use your time wisely. The prelim is worth 100 points total. The point breakdown for the parts of each problem is printed with the problem. Some of the problems have several parts, so make sure you do all of them!

This is an closed-book examination; you **may not** use outside materials, calculators, computers, etc.

Do all written work on the exam itself. If you are running low on space, write on the back of the exam sheets and be sure to write (OVER) on the front side. It is to your advantage to show your work — we will award partial credit for incorrect solutions that are headed in the right direction. If you feel rushed, try to write a brief statement that captures key ideas relevant to the solution of the problem.

If you finish in the last ten minutes of the exam, please remain in your seat until the end of the exam as a courtesy to your fellow students.

Name and NetID ______________________________

Problem	Points	Score
1	18	
2	17	
3	15	
4	13	
5	37	
Total	100	

NetID ______________________________

The Final Exam of 312

In the Elder Days, before the rise of Man, before the Host of Valinor broke Thangorodrim and overthrew Morgoth, before Numenor and the Silmarils, a great evil waited, brooding. To this no other evil can be compared; even Morgoth, the great Enemy of whom Sauron was a merest lieutenant, was not worthy to clip its toenails.

And in the land of Corn'ell there dwelt the students, a noble and proud people, whose grades and scores shone bright with the fires of the stars. They had answered the great questions of SAT, and proven their valor before the Admissions Committee. So great was their might and splendor that even the Great Roster of Courses had to do homage unto Hrunding the Hrunpronouncable, the greatest of the student-kings. But in their pride and folly, the students turned toward the data structures of CS and the lure of functional abstraction.

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.

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

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

d. ____ The function $f(n) = n^2 + n + 1$ is $O(3n^3)$

e. ____ The SML expression `fn(f) => f(f)` is well-typed.

f. ____ The SML expression `fn x => fn y => 1` has a polymorphic type.

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).

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

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

2. Programming Languages [17 pts]
   (parts a–c)

   The following defines (most of) an implementation of the Fibonacci function, without using the keyword `fun`:

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

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

   (b) [7 pts] There are three ...'s to be filled in to complete the code. Provide code for each of them. You may not use the keywords `fun` or `and`.

   $\ldots^1$ =

   $\ldots^2$ =

   $\ldots^3$ =

Consider the following function:

```
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.

And so it came to pass that after the fall of Gil-Galad, the class passed to the unknowing hands of Mortal Men. It was there that Functor son of Closure became the first steward of the class. The class was passed down this noble line, through the Four Great Kings, the Six Not-So-Great Kings, the Twenty-Three Mediocre Kings, the Forty-Two Kings Who Nobody Remembers, the Eight Silly Kings, the Two Postmodern Existential Kings, and lastly to the Five Most Recent Kings While There Were Still Kings.

It was in the time of the last of the Kings that portentous and momentous events took shape. It was then that the Cracker Jack Decoder Rings of Power were forged by the great elven-smiths under the guidance of Sauron and with the use of the Y combinator. But the elven-smiths were deceived; the rings were not simple trinkets to be distributed at his newphew's grandson's bar mitzvah. In secret the Dark Prof was stirring again, and was bending the events of Middle-Earth to his desires. The Class would rise once again, and the number that was its sign was 312...

3. Streams [15 pts]
(parts a–b)

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

```
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(b(i-1)) for i > 0 and (a0, b0) = f(b).
   * The values ai, bi are evaluated lazily. *)
  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:

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), ...] *)
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 ... :

```
fun stream_pair(a: 'a stream, b: 'b stream) =
  Stream.make((a,b), ...)
```

...Thus the deception was complete. At length did the students hearken to his lectures, and were dazzled by his promises of representation invariants and polymorphic functions. Under the sections of the TAs, who were once great students themselves but were now bound in the service of the Dark Prof, the students received instruction in the great structures of data and the interpretation of programs. With this dark guidance the students-coders reached the heights of their functional craft, and it was in this time that the Six Great Problem Sets were forged.

Yet as the weeks passed, they became more and more beholden to the Dark Prof. They build great dictionaries of the compressed tongue, clearing the once-grand forest of red-black trees. They raised a great army to force the inclusion of objects in functional languages. By the reign of Huffman son of Puffman, they had blown the house down, and their hearts had turned to the dark, save for a few.

But not all the lands of CS had fallen under the sway of the Structure and the Signature. The Paths of Dijkstra, though not trodden by mortal feet since the days of the graph, still connected the shortest way across the breadth of Middle-Earth. Regexp son of FedEx still reigned in the most regular of the high kingdoms of Automata...

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.

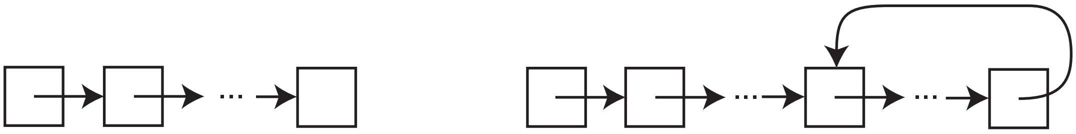
Suppose that when you first see the light it is green and and when you last see it is green. Write a regular expression that describes the possible strings that you might have jotted down.

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

(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?

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:

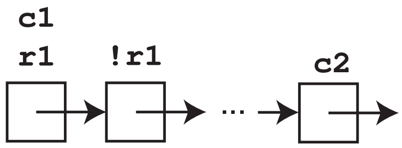
```
datatype chain = nil | node of chain ref

(*  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.

(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.

(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.

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 `c1` 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.)

(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 `c1` is a loop *that includes* `c1`, then `looped'` returns `true`.
(*Hint*: In this case it must also be possible to reach `c1` from `c2`. Let n be the smallest non-zero number of edges that must be traversed to get from `c2` to `c1`. Use induction on n, with the base case $n = 1$.)

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 `c1`. Suppose that there is a loop reachable from `c1`. 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 `c1` 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 `looped'` returns `true` in the base case $n = 0$.

The rest of the proof, for free (the easy part!): Now let's consider the inductive step. Suppose that $n > 0$, so `c1` is not in the loop. Our inductive hypothesis is that for loops reachable in $n - 1$ steps the function returns true. Since `c1` is not in the loop, we must have `r1<>r2` and therefore the result comes from the recursive call. But the distance from `!r1` 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 `looped'` works correctly, and therefore so does `looped`.

Extra copy of code for `looped` (May be removed)

```
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
```

Here is related a part of the tale of Frito son of Chee-tos, and his quest to pass the One Class

"I cannot read the fiery letters," said Frito.

"No," said the wizard, "but I can. The letters are ASCII, of a seven-bit encoding, but the language is that of LaTeX, which I will not utter here.

"But this in the Common Tongue is what is said, close enough:

One Test to weigh them all, One Test we made them
One Test to bring them all and on the curve grade them

"It is only two lines of a verse long known in CS-lore:

Homework for students who want to have fun
Prelims for those whose grades could fall
Consulting for hackers whose code does not run
Grades from the Dark Prof in his Dark Hall

In the Hall of Kimball where the students try
One Test to weigh them all, One Test we made them
One Test to bring them all and on the curve grade them
In the Hall of Kimball where the students try

He paused, and then said slowly in a deep voice: "This is the Final Exam, the One Test to rule them all. This is the 312 that was devised many semesters ago."

Frito sat silent and motionless. Fear seemed to stretch out to engulf him. "This class!" he stammered. "How on earth did I come to take it?"

The wizard spoke in a firm tone. "You must find the courage within yourself, to take the class all the way to the Final Exam where you will pass it and Middle-Earth will be saved..."