FOR INTERNAL SCRUTINY (date of this version: 12/5/2010)

UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

FUNCTIONAL PROGRAMMING AND SPECIFICATION SAMPLE EXAM

Thursday 1st April 2010

00:00 to 00:00

Year 3 Courses

Convener: ITO-Will-Determine
External Examiners: ITO-Will-Determine

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.

All questions carry equal weight.

FOR INTERNAL SCRUTINY (date of this version: 12/5/2010)

1. **Your answers to this question should be split into three separate files; see below for details.**

   Consider binary trees defined as follows:

   ```
   datatype 'a tree = empty | tip of 'a | node of 'a tree * 'a tree
   ```

   The *deepest* tips are the ones that are furthest from the root of the tree. So in

   ```
   node(node(node(tip 2, empty),
             tip 1),
        node(empty,
             node(tip 3, tip 2)))
   ```

   the deepest tips are the ones underlined. Note that all of the deepest tips are at the same depth.

   (a) **Save your answer to this sub-question in a file `q1a.sml` and submit that file when you are finished using the command: `examsubmit q1a.sml`**

   We want a function `deepest : 'a tree -> 'a list` that returns a list containing the labels of the deepest tips in a tree, in the order that they appear. In the example above, the result would be `[2,3,2]`.
   One algorithm for `deepest` works by computing the depth of the deepest tip and then using this result in a function that returns the list of all tip labels at a given depth. Implement this. [*8 marks*]

   (b) **Save your answer to this sub-question in a file `q1b.sml` and submit that file when you are finished using the command: `examsubmit q1b.sml`**

   Another algorithm for `deepest` works by traversing the tree, deciding at each node which recursive applications of `deepest` are needed in the result on the basis of the depth of that subtree. Implement this. [*8 marks*]

   (c) **Save your answer to this sub-question in a file `q1c.sml` and submit that file when you are finished using the command: `examsubmit q1c.sml`**

   Implement `shallowest : 'a tree -> 'a list`, which returns a list containing the labels of the *shallowest* tips in a tree. In the example above, the result would be `[1]`. (Note: `shallowest` t should be `[]` only if there are no tips in t.) [*9 marks*]

2. **Your answers to this question should be split into two separate files; see below for details.**

Sets of integers may be represented using lists of "intervals": the interval $[a, b]$ for $a \leq b$ represents the set of integers between a and b inclusive, where $[a, a]$ represents the set $\{a\}$. A list containing several intervals represents the union of the sets represented by the intervals. If the intervals do not overlap or "touch" then this representation is space-efficient; if they are kept in ascending order then manipulation of sets can be made time-efficient. We call a list of intervals *valid* if it satisfies these conditions.

Here is an example of sets represented this way, and four non-examples. (We represent the interval $[a, b]$ in ML as the pair `(a,b):int*int`.)

`[(1,3),(7,7),(10,11)]` is valid and represents $\{1, 2, 3, 7, 10, 11\}$.
`[(2,1),(5,6)]` is invalid: $[2, 1]$ isn't a valid interval.
`[(1,4),(3,6)]` is invalid: intervals overlap.
`[(1,4),(5,6)]` is invalid: intervals "touch".
`[(3,4),(1,1)]` is invalid: intervals aren't in ascending order.

When implementing functions using this representation, one may assume that sets provided as input are valid, and sets produced as results must be valid.

The most complicated function to implement is *insertion* of an integer into a set; here it is. (You will find this code in a file named `sets.sml`.)

```
local
    fun insert'(z,nil) = [(z,z)]
      | insert'(z,(a,b)::l) =
          if z<a then (z,z)::(a,b)::l
          else if z<=b then (a,b)::l
          else (a,b)::insert'(z,l)
    fun fix nil = nil
      | fix [(a,b)] = [(a,b)]
      | fix((a,b)::(c,d)::l) =
          if b+1=c then fix((a,d)::l) else (a,b)::(fix((c,d)::l))
in
    fun insert(z,l) = fix(insert'(z,l))
end
```

(a) **Save your answer to this sub-question in a file `q2a.sml` and submit that file when you are finished using the command: `examsubmit q2a.sml`**

Implement the following.

i. The *empty set* [*1 mark*]

ii. *Set membership*: checking if an integer is in a set. [*5 marks*]

iii. *Deletion* of an integer from a set. [*7 marks*]

(b) **Save your answer to this sub-question in a file `q2b.sml` and submit that file when you are finished using the command: `examsubmit q2b.sml`**

The list-of-intervals representation of sets may be used for sets over domains other than integers. Give an SML functor that implements sets over any given linearly-ordered element type, by modifying the functions in your answer to (a) and the definition of `insert` above. (Concentrate on *discrete* domains like integers, disregarding domains such as the real numbers.) Include both input and output signatures, with the output signature ascribed transparently [*9 marks*]

or for full marks, opaquely. [*+3 marks*]

3. **Save all of your answers to this question in a file `q3.sml` and submit that single file when you are finished using the command:** `examsubmit q3.sml`

   (a) Define an ML function `ncompose : ('a -> 'a) * int -> ('a -> 'a)` that takes a function f and a non-negative integer n and returns the n-fold composition of f with itself. If $n = 0$, `ncompose` should return the identity function.

   For instance, `ncompose((fn x => x+7), 4)` returns a function that is equivalent to the function `(fn x => x+28)` and `ncompose((fn x => x+7), 0)` returns `(fn x => x)`. [*5 marks*]

   (b) Define a function `lcompose : ('a -> 'a) list -> ('a -> 'a)` that takes a list of functions and returns the sequential composition of the functions in the list.

   For instance, `lcompose [fn x => 2*x, fn x => x-1]` should return a function that is equivalent to the function `fn x => 2*(x-1)`. When the list is empty, `lcompose` should return the identity function.

   Your solution should use the `foldr` function; recall that it is defined as

   ```
   fun foldr f e nil = e
     | foldr f e (h::t) = f(h,foldr f e t)
   ```

   with type `('a * 'b -> 'b) -> 'b -> 'a list -> 'b`. [*5 marks*]

   (c) The function `createGate` has type

   ```
   (string -> string) * int * string * 'a -> (string -> 'a option)
   ```

   `createGate(`f`,`n`,`s`,`v`)` returns a function that expects a string password. When called with the correct password, this function returns `SOME(`v`)`; otherwise it returns `NONE`. The correct password is determined by applying the function f to the string s, n times, where n is non-negative.

   For instance, suppose `rotate : string -> string` is a function that rotates a string one position to the right. Then evaluation of

   ```
   val gate = createGate(rotate, 3, "abcdefg", "secret")
   ```

   will yield `gate : string -> string option` which expects the password `"efgabcd"`. `gate "efgabcd"` will return `SOME("secret")` whereas `gate "wrong"` will return `NONE`.

   Define `createGate`. [*8 marks*]

   (d) Suppose that, in the scenario of part (c), you have discovered the function f and the string s but not the number n. Define a function `crack` with type

   ```
   (string -> 'a option) -> (string -> string) * string -> 'a
   ```

   that searches until it finds the correct n and uses it to penetrate the gate. For instance, `crack gate (rotate,"abcdefg")` should yield `"secret"`. [*7 marks*]