UNIVERSITY OF EDINBURGH

FACULTY OF SCIENCE AND ENGINEERING

M.Sc./DIPLOMA IN ARTIFICIAL INTELLIGENCE
M.Sc./DIPLOMA IN COGNITIVE SCIENCE
M.Sc./DIPLOMA IN COMPUTER SCIENCE
M.Sc./DIPLOMA IN INFORMATICS

FUNCTIONAL PROGRAMMING AND SPECIFICATION

Friday 0 2002

??:?? to ??:??

Examiners:
N. Chater (External)
N. Jennings (External)
E. Hancock (External)
K. Hawick (External)
R.N. Ibbett (Convener)
J. Oberlander (Co-Convener)

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.

All questions carry equal weight.

1. Recall the definition of `'a option`:

```
datatype 'a option = NONE | SOME of 'a
```

Consider the following functions:

```
fun mapPartial f [] = []
  | mapPartial f (h::t) = case f h of NONE => mapPartial f t
                                    | SOME v => v::(mapPartial f t)

fun take(_,0) = []
  | take([],_) = raise Subscript
  | take(h::t,n) = h::take(t,n-1)
```

(a) Sometimes these functions are composed as follows:

```
take(mapPartial f s, n)
```

which can be inefficient since list elements are computed by `mapPartial` only to be discarded by `take`. In such circumstances it would be more efficient to use a function `takeMapPartial` with the type

```
('a -> 'b option) -> 'a list -> int -> 'b list
```

Implement `takeMapPartial`. [*10 marks*]

Answer:

```
fun takeMapPartial _ _ 0 = []
  | takeMapPartial _ [] _ = raise Subscript
  | takeMapPartial f (h::t) n =
        case f h of NONE => takeMapPartial f t n
                  | SOME v => v::(takeMapPartial f t (n-1))
```

(b) Give an example of `f`, `s` and `n` for which `take(mapPartial f s, n)` and `takeMapPartial f s n` would differ. [*5 marks*]

Answer: A difference can only arise from the unnecessary application of `f` to a value for which it would fail to terminate or raise an exception. So let

```
fun f 0 = f 0  | f n = SOME n
```

then `takeMapPartial f [1,2,3,0] 3 = [1,2,3]` while evaluation of the expression `take(mapPartial f [1,2,3,0], 3)` would fail to terminate.

(c) Some SML functions raise exceptions instead of explicitly indicating failure with the value `NONE` of the `option` datatype. Consider now `robustMap`, a robust version of the `map` function which traps these exceptions and continues to process the remainder of the list, with the same type as `map` itself:

```
('a -> 'b) -> 'a list -> 'b list
```

Provide *two* implementations of `robustMap`, one which uses `mapPartial` and one which does not. [*10 marks*]

Answer:

```
fun robustMap f =
  let
    fun robustF x = SOME(f x) handle _ => NONE
  in
    mapPartial robustF
  end

fun robustMap f [] = []
  | robustMap f (h::t) =
      (f h)::(robustMap f t) handle _ => robustMap f t
```

2. A *multiqueue* is a collection of queues with a service discipline where the customer at the front of the longest queue is served first. A new customer might choose to join any of the existing queues by number or start another one (by joining queue $n+1$ out of n, say). One can see how many customers are in each queue; there are zero customers in queue $n+1$ of n. Attempting either to identify or to serve the front customer when there are no customers are both exceptional cases. The following SML signature describes multiqueues:

```
signature MULTIQUEUE =
  sig
    type multiqueue
    type customer
    val empty: multiqueue
    val join: multiqueue * int * customer -> multiqueue
    val customers: multiqueue * int -> int
    exception Front
    exception Serve
    val front: multiqueue -> customer
    val serve: multiqueue -> multiqueue
  end
```

Implement an SML functor which produces a multiqueue structure for a given customer type. (One way of representing a multiqueue is as a list of lists. If you choose this representation then you may use functions such as `length`, `nth`, `hd` and `tl` if you require them.) [*25 marks*]

Answer: Five marks for each of the four functions plus five marks for the functor heading. The `serve` function below does not preserve the order of queues; this was not required by the question.

```
functor Multiqueue(type C):>MULTIQUEUE where type customer=C =
  struct
    type customer = C
    type multiqueue = (customer list) list
    val empty = []
    fun join([],1,c) = [[c]]
      | join([],n,c) = [] :: join([],n-1,c)
      | join(h::t,1,c) = (h@[c])::t
      | join(h::t,n,c) = h::join(t,n-1,c)
    fun customers(m,n) = length(nth(m,n)) handle Nth => 0
    exception Front
    exception Serve
    fun insert(x,[]) = [x]
      | insert(x,h::t) =
        if (length x) > (length h) then x::h::t else h::insert(x,t)
    fun sort [] = []
      | sort(h::t) = insert(h,sort t)
    fun front m = ((hd o hd) (sort m)) handle Hd => raise Front
    fun serve m =
        case sort m of [] => raise Serve
```

```
                    | h::t=> ((tl h)::t handle Tl => raise Serve)
end
```

3. Consider the following SML signature for a simple pattern matcher:

```
signature MATCH =
  sig
    type pattern = string list
    val matches : pattern * int list -> bool
  end
```

A pattern is a list of variables, represented as strings. When matching a list of variables against a list of integers, variables must be matched consistently. So, for instance, `["X","Y","X"]` matches `[1,1,1]` and `[2,1,2]` but not `[1,2,2]`.

(a) Turn the above signature into an EML specification of the problem. [*8 marks*]

Answer:

```
signature MATCH =
  sig
    type pattern = string list
    val matches : pattern * int list -> bool
    axiom forall (vl,nl) =>
            matches(vl,nl) iff (exists f => map f vl = nl)
  end
```

(b) Give an SML structure that matches your EML signature. [*8 marks*]

Answer:

```
structure Match:MATCH =
  struct
    type pattern = string list
    fun matcheswith f [] [] = true
      | matcheswith f [] (n::nl) = false
      | matcheswith f (v::vl) [] = false
      | matcheswith f (v::vl) (n::nl) =
          (f v = n) andalso (matcheswith f vl nl)
          handle Match =>
          matcheswith (fn x => if x=v then n else f x) vl nl
    fun matches(vl,nl) = matcheswith (fn x=>raise Match) vl nl
  end
```

(c) Explain why your structure satisfies your specification. You need not give a formal proof. [*9 marks*]

Full marks will be given for an explanation that is essentially an outline of a proof.

The idea of the implementation is to construct a "witness" for `f` in the specification. If such an `f` can be constructed than we have a match; if this is impossible because of an inconsistency then there is no match.

Suppose `matches(vl,nl)=true`. Then by inspection of `matcheswith`, this will involve evaluation of an expression of the form `matcheswith f [] []`. This `f` is the witness we seek. We can show by induction on the structure of the lists that `(matcheswith f vl sl) iff (map f vl = sl)`. Then by inspection of `matcheswith` we see that `f` is built up by adding associations when a variable is first encountered, and associations are never removed, so

```
(matcheswith f vl sl) iff (matcheswith (fn x=>raise Match) vl sl)
                        = matches(vl,nl)
```

Suppose `matches(vl,nl)=false`. Then either the lengths of `vl` and `nl` are different or else there is some variable that is repeated in `vl` with the corresponding integers in `nl` being different; that can be seen from the definition of `matcheswith`. In that case there can be no witness `f`.