

**Possible solutions for
Written examination
18 June 2003 ***

Updated 2003-06-22: More cases added to solution 2.5

The suggested solutions below are not the only possible ones.

Question 1 (25 %): Standard ML

Question 1.1

```
> val isSorted = fn : int list -> bool
> val res1 = false : bool
  val 'a isSortedGen = fn : ('a -> int) -> 'a list -> bool
> val res2 = true : bool
  val 'a makeIntv = fn : 'a -> 'a list -> ('a * 'a) list
> val makeIntervals = fn : int list -> (int * int) list
> val res3 = [(0, 2), (2, 5), (5, 8)] : (int * int) list
```

The application `isSorted xs` returns `true` if the list `xs` of numbers appears in strictly increasing order; otherwise `false`. (The overloading resolution defaults to `int`, but with a suitable type constraint the function could work on `string` or `real` also; this is not an important point in the question).

The application `isSortedGen f xs` returns `true` if the list of integers resulting from applying `f` to the elements of `xs` appears in strictly increasing order; otherwise `false`.

The application `makeIntervals xs` first checks that the elements of `xs` appear in strictly increasing order; if not, it raises exception `Fail`. If they do, it creates a list of integer pairs which represent the intervals between the elements of the integer list `xs`.

Question 1.2

```
fun checkNonEmpty (t1, t2) = t1 < t2

fun checkAllNonEmpty intvs = List.all checkNonEmpty intvs;
```

Question 1.3

```
fun totalLength intvs =
  List.foldl (fn ((t1,t2), res) => res+(t2-t1)) 0 intvs
```

Question 1.4

```
fun checkActivities [] = true
  | checkActivities [intv1] = true
  | checkActivities ((t11, t12) :: (t21, t22) :: rest) =
    t12 <= t21 andalso checkActivities ((t21, t22) :: rest)
```

Question 1.5

```
fun move k intvs = List.map (fn (t1, t2) => (t1+k, t2+k)) intvs
```

Question 1.6

```
fun belongs2 t intvs =
  List.exists (fn (t1,t2) => t1 <= t andalso t < t2) intvs
```

Question 2 (25 %): Grammar and abstract syntax**Question 2.1**

```

Drawing ::= white
         | black
         | Drawing - Drawing
         | Drawing | Drawing
         | n *- Drawing
         | n *| Drawing
         | < Drawing
         | > Drawing
         | ( Drawing )
         | let x = Drawing in Drawing end
         | x

```

Question 2.2

```

%left DASH          /* lowest precedence */
%left BAR
%nonassoc STARDASH STARBAR
%nonassoc LEFT RIGHT /* highest precedence */

```

Question 2.3 and 2.4

```

Main:
    Drawing EOF          { $1 }
;

Drawing:
    WHITE                { Fill White }
    | BLACK              { Fill Black }
    | Drawing DASH Drawing { Col2 ($1, $3) }
    | Drawing BAR Drawing { Row2 ($1, $3) }
    | LEFT Drawing       { RotL $2 }
    | RIGHT Drawing      { RotR $2 }
    | INT STARDASH Drawing { RepC ($1, $3) }
    | INT STARBAR Drawing { RepR ($1, $3) }
    | LPAR Drawing RPAR  { $2 }
    | LET ID EQUALS Drawing IN Drawing END { Let($2, $4, $6) }
    | ID                 { Var $1 }
;

```

Question 2.5

Here's a super deluxe simplifier for expressions that represent drawings. Even so I'm not sure I have not missed some cases. The function `simp` performs the actual simplifications, and `simplify` repeats the application of `simp` until no more simplifications are made.

To solve the exam question, it would suffice to do approximately half of `simp`; there was no requirement to implement a function like `simplify`.

```

fun simp (RepC(1, d))      = simp d
  | simp (RepC(n, d)) =
    (case simp d of
      Fill c      => Fill c
    | Row2(d1, d2) => Row2(d1, d2)
    | RepC(n2, d2) => RepC(n*n2, d2)
    | simpd       => RepC(n, simpd))
  | simp (RepR(n, d)) =
    (case simp d of
      Fill c      => Fill c
    | Col2(d1, d2) => Col2(d1, d2)
    | RepR(n2, d2) => RepR(n*n2, d2)
    | simpd       => RepR(n, simpd))
  | simp (Col2(d1, d2)) =
    let val d1s = simp d1
        val d2s = simp d2
    in
      if d1s = d2s then
        d1s
      else
        Col2(d1s, d2s)
    end
  | simp (Row2(d1, d2)) =
    let val d1s = simp d1
        val d2s = simp d2
    in
      if d1s = d2s then
        d1s
      else
        Row2(d1s, d2s)
    end
  | simp (RotR d) =
    (case simp d of
      Fill c      => Fill c
    | RotL d1     => d1
    | RotR (RotR d) => RotL d
    | Col2 (d1, d2) => Row2(RotR d2, RotR d1)
    | Row2 (d1, d2) => Col2(RotR d1, RotR d2)
    | simpd       => RotR simpd)
  | simp (RotL d) =
    (case simp d of
      Fill c      => Fill c
    | RotR d1     => d1
    | RotL (RotL d) => RotR d
    | Col2(d1, d2) => Row2(RotL d1, RotL d2)
    | Row2(d1, d2) => Col2(RotL d2, RotL d1)
    | simpd       => RotL simpd)
  | simp (Let(x, d1, d2)) = Let(x, simp d1, simp d2)
  | simp d = d;

fun simplify d =
  let val res = simp d
  in
    if res=d then res else simplify res
  end

```

Question 3 (25 %)

Question 3.1

First, manually execute the code p below on the stack machine. Show the stack contents after every instruction, and draw the corresponding drawing:

```
FILL 1          Fill(1)
FILL 0          Fill(1) : Fill(0)
ROW2           Row(Fill(1), Fill(0))
FILL 0         Row(Fill(1), Fill(0)) : Fill(0)
COL2          Col(Row(Fill(1), Fill(0)), Fill(0))
STOP          -
```

The lower half of the resulting drawing is black; the upper half's left half is white and its right half is black. The result is a black square with a white square in the upper left-hand quarter.

Question 3.2

```
FILL 0
FILL 1
COL2
STOP
```

Question 3.3

```
fun board n =
  [FILL 1, FILL 0, COL2, FILL 0, FILL 1, COL2, ROW2, REPC n, REPR n, STOP];
```

Question 3.4

```
case VAR:
  s[sp+1] = s[sp-p[pc++]];
  sp++;
  break;
```

Question 4 (25 %): Compilation**Question 4.1**

```

fun comp41 e =
  case e of
    Fill c =>
      [FILL (case c of Black => 0 | White => 1)]
  | Col2(d1, d2) =>
      comp41 d1 @ comp41 d2 @ [COL2]
  | Row2(d1, d2) =>
      comp41 d1 @ comp41 d2 @ [ROW2]
  | RepC(n, d) =>
      comp41 d @ [REPC n]
  | RepR(n, d) =>
      comp41 d @ [REPR n]
  | RotL d =>
      comp41 d @ [ROTL]
  | RotR d =>
      comp41 d @ [ROTR]
  | Let(x, dx, dbody) => raise Fail "not implemented"
  | Var x => raise Fail "not implemented"

fun compile41 e = comp41 e @ [STOP];

```

Question 4.2

```

FILL 1, FILL 0, ROW2,           // let pair = white | black
VAR 0, VAR 1, ROTL, ROTL, COL2, // let square = pair - < < pair
VAR 0, REPC 4, REPR 4,         // 4 * | (4 *- square)
SWAP, POP, SWAP, POP,          // drop variables pair and square
STOP                             // stop and draw

```

Question 4.3

The compilation function `comp` takes two arguments: a compile-time environment `cenv` and an expression `e` to compile. The compile-time environment is a list of `rtvalues` that describe the values on the stack at runtime. The idea is that `Bound "x"` stands for a variable "x", and `Intrm` stands for an intermediate result on the stack. See lecture slide 2-22.

```

datatype rtvalue =
  Bound of string
  | Intrm

fun getIndex []      x = raise Fail "Unknown variable"
  | getIndex (y::r) x = if y=x then 0 else 1 + getIndex r x;

fun comp cenv e =
  case e of
    Fill c =>
      [FILL (case c of Black => 0 | White => 1)]
  | Col2(d1, d2) =>
      comp cenv d1 @ comp (Intrm :: cenv) d2 @ [COL2]
  | Row2(d1, d2) =>
      comp cenv d1 @ comp (Intrm :: cenv) d2 @ [ROW2]
  | RepC(n, d) =>
      comp cenv d @ [REPC n]
  | RepR(n, d) =>
      comp cenv d @ [REPR n]
  | RotL d =>
      comp cenv d @ [ROTL]
  | RotR d =>
      comp cenv d @ [ROTR]
  | Let(x, dx, dbody) =>
      comp cenv dx @ comp (Bound x :: cenv) dbody @ [SWAP, POP]
  | Var x =>
      [VAR (getIndex cenv (Bound x))];

fun compile e = comp [] e @ [STOP];

```