

**Written examination**  
**18 June 2003 \*\***

This examination comprises 9 pages. Please check immediately that you have a complete set of questions. There are four questions, all of whose subquestions should be satisfactorily answered to get full marks. You may use any books, lecture notes, exercises, pocket calculators, and so on at the examination, but not computers that can run Standard ML.

## Question 1 (25 %): Standard ML

### Question 1.1

Here are four Standard ML functions on lists, and three declarations of variables:

```
fun isSorted [] = true
  | isSorted [t1] = true
  | isSorted (t1 :: t2 :: rest) = t1 < t2 andalso isSorted (t2::rest);
val res1 = isSorted [4, 5, ~9]

fun isSortedGen f [] = true
  | isSortedGen f [t1] = true
  | isSortedGen f (t1 :: t2 :: rest) =
    f t1 < f t2 andalso isSortedGen f (t2::rest);
val res2 = isSortedGen (fn i => i*i) [4, 5, ~9]

fun makeIntv start [] = []
  | makeIntv start [t] = [(start, t)]
  | makeIntv start (t :: rest) = (start, t) :: makeIntv t rest;
fun makeIntervals ts =
  if isSorted ts then makeIntv 0 ts
  else raise Fail "times not sorted";
val res3 = makeIntervals [2, 5, 8];
```

For each of the three functions `isSorted`, `isSortedGen` and `makeIntervals`, show the most general (most polymorphic) type of the function, and briefly explain what the function does. For each variable (`res1`, `res2`, `res3`) show the value of the variable.

### Question 1.2

In this question, and in those below, the duration of an activity (such as making coffee, or writing a report) is given by an interval. An interval is represented by a pair  $(t_1, t_2)$  of integers, where  $t_1$  is the starting time of the activity and  $t_2$  is the ending time. Thus the pair  $(t_1, t_2)$  represents an activity that lasts from time  $t_1$  until but not including time  $t_2$ .

(a) Write a function `checkNonEmpty : int * int -> bool` to check that a given interval  $(t_1, t_2)$  is non-empty, that is, that  $t_1 < t_2$ .

(b) A collection of activities is represented by a list of intervals.

Write a function `checkAllNonEmpty : (int * int) list -> bool` to check that all intervals in a list of activities are non-empty.

**Question 1.3**

The length of an activity with interval  $(t_1, t_2)$  is the difference  $t_2 - t_1$ . Write a function `totalLength : (int * int) list -> int` to compute the total length of the activities, that is, the sum of the interval lengths.

**Question 1.4**

The activities in a list should be ordered and non-overlapping: the ending time of one activity must be less than or equal to the starting time of the next activity. That is, if an activity in the interval  $(t_{11}, t_{12})$  is followed by an activity in the interval  $(t_{21}, t_{22})$  in the list, then it must hold that  $t_{12} \leq t_{21}$ .

Write a function `checkActivities : (int * int) list -> bool` that checks this property.

**Question 1.5**

Write a function `move : int -> (int * int) list -> (int * int) list` that postpones all activities by  $k$  time units. That is, `move k [(t11, t12), (t21, t22), ..., (tn1, tn2)]` should give the result `[(t11+k, t12+k), (t21+k, t22+k), ..., (tn1+k, tn2+k)]`.

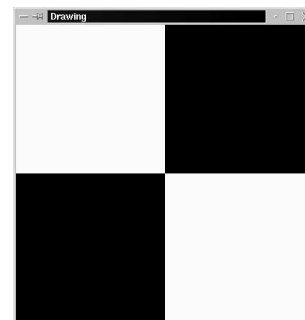
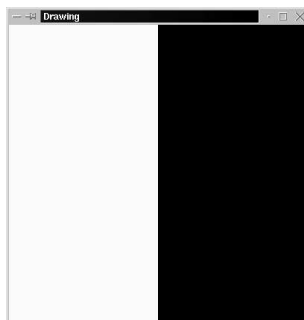
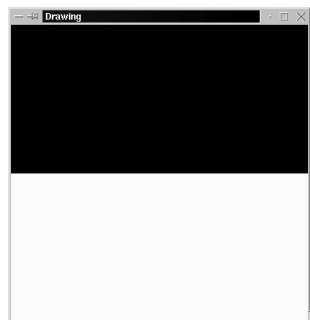
**Question 1.6**

A point  $t$  in time belongs to an interval  $(t_1, t_2)$  if  $t_1 \leq t$  and  $t < t_2$ .

Write a function `belongs : int -> (int * int) list -> bool` that returns `true` if  $t$  belongs to one of the intervals in the list, and `false` otherwise.

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

This and the following questions concern an expression language for producing simple black and white rectangular drawings, such as these three:



The form and meaning of drawing expressions are explained by the table below. Let  $d$ ,  $d_1$  and  $d_2$  be drawing expressions,  $n > 0$  an integer, and  $x$  a variable name:

Expression	Meaning
<code>white</code>	A completely white drawing
<code>black</code>	A completely black drawing
$d_1 - d_2$	Column: Make drawing whose top half is $d_1$ and whose bottom half is $d_2$
$d_1   d_2$	Row: Make drawing whose left half is $d_1$ and whose right half is $d_2$
$n * - d$	Column repeat: Make drawing from $n$ equal-size copies of $d$ on top of each other
$n *   d$	Row repeat: Make drawing from $n$ equal-size copies of $d$ put beside each other
$< d$	Rotate left: Make drawing which is $d$ rotated counterclockwise by 90 degrees
$> d$	Rotate right: Make drawing which is $d$ rotated clockwise by 90 degrees
$(d)$	Make the drawing $d$
<code>let <math>x = d_1</math> in <math>d_2</math> end</code>	Bind the drawing $d_1$ to $x$ when making drawing $d_2$
$x$	Make the drawing bound to variable $x$ (which must be <code>let</code> -bound)

The column operator ‘-’ is meant to resemble the horizontal dividing line between drawings put on top of each other, and the row operator ‘|’ is meant to resemble the vertical dividing line between drawings put beside each other.

Some example drawing expressions:

- The expression `black - white` produces the leftmost drawing shown above: a completely black drawing stacked on top of a completely white one.
- The expression `white | black` produces the middle drawing shown above: a completely white drawing next to a completely black one.
- The expression `< (white | black)` would also produce the leftmost drawing shown above: the subexpression within parentheses creates a drawing like the middle one, and the left rotation operator ‘<’ rotates it left by 90 degrees.
- Similarly, the expression `> (black | white)` would also produce the leftmost drawing shown above: the subexpression within parentheses creates a drawing whose left half is black and whose right half is white, and the right rotation operator ‘>’ rotates it right by 90 degrees so black gets on top.
- The expression `let pair = white | black in pair - << pair end` produces the rightmost drawing shown above. The subexpression `white | black` creates a drawing like the middle one, which gets bound to the variable `pair`. The subexpression `<< pair` creates a drawing by rotating `pair` twice to the left, so black and white swap place. Finally, the `pair` drawing is stacked on top of this rotated drawing using the column operator ‘-’. The result is equivalent to `(white | black) - (black | white)` and also to `(white - black) | (black - white)`.

A drawing does not itself determine how large it is; it just fills the space given to it. The column composition (-) and the row composition (|) divide the drawing space equally between the two subdrawings. Thus `white | white`, for example, produces the same result as `white`: each of the two white subdrawings will have exactly half the size of the single white drawing.

Here is an abstract syntax for drawings:

```
datatype color = Black | White

datatype drawing =
  Fill of color (* A filled rectangle *)
| Col2 of drawing * drawing (* Drawing d1 on top of d2 *)
| Row2 of drawing * drawing (* Drawing d1 to left of d2 *)
| RepC of int * drawing (* Column of n copies of drawing *)
| RepR of int * drawing (* Row of n copies of drawing *)
| RotL of drawing (* Rotate left (counterclockwise) *)
| RotR of drawing (* Rotate right (clockwise) *)
| Let of string * drawing * drawing (* Let-bind drawing in drawing *)
| Var of string (* Variable denoting drawing *)
```

**Question 2.1**

Write an informal grammar for drawings. It must be able to generate all the expression forms shown in the table on page 3. A single nonterminal symbol suffices.

**Question 2.2**

Write the operator precedence and associativity part of the `mosmlyac` parser specification (`.grm` file). The rotation operators '`<`' and '`>`' should have higher precedence (bind more strongly) than '`*-`' and '`*|`', which have higher precedence than '`|`', which has higher precedence than '`-`'. The latter two operators associate to the left.

You may assume that the following token declarations are available in the parser specification already, where `INT` represents integer constants  $n$ , and `ID` represents variable names  $x$ :

```
%token <int> INT
%token <string> ID
%token WHITE BLACK
%token DASH BAR
%token STARDASH STARBAR
%token LEFT RIGHT
%token LPAR RPAR
%token LET EQUALS IN END
%token EOF
```

**Question 2.3**

Write the rule part of a mosmlyac parser specification for drawings. It should have this form:

```

Main:
    Drawing EOF          { ... }
;

Drawing:
    ...                 { ... }
    | ...               { ... }
    ...
;
    
```

**Question 2.4**

Extend the parser specification from question 2.3 with semantic actions, so that each rule generates a Standard ML value corresponding to the abstract syntax shown above.

**Question 2.5**

Many different drawing expressions have the same meaning. For example, rotating a completely white drawing has no effect, so `< white` is the same as `white`.

Similarly, `< > d`, which means `< (> d)`, is the same as `d` itself: rotating right and then left has no effect.

More subtly, `< (d1 - d2)` is the same as `(< d1) | (< d2)`: by left-rotating a column that has `d1` on top of `d2` we obtain a row that has a left-rotated `d1` to the left of a left-rotated `d2`.

Write an SML function `simplify : drawing -> drawing` that simplifies the abstract syntax representing a drawing by eliminating useless rotations, by pushing rotations inside other operators, and what else you can think of.

You do not have to implement more than 7 kinds of simplifications; implementing all possible simplifications takes a good deal of thought and time.

### Question 3 (25 %): Stack machines

This question concerns a simple stack machine for interpreting drawing commands. The stack machine state has a program  $p$ , a program counter  $pc$ , a stack  $s$  holding representations of drawings, and a stack top pointer  $sp$ . The instructions of the abstract machine, and their effect on the stack, are described by this table:

Instruction	Stack before	Stack after	Effect
FILL $c$	$s$	$s : \text{Fill}(c)$	Make filled rectangular drawing in color $c$ (0=black, 1=white)
COL2	$s : d_1 : d_2$	$s : \text{Col}(d_1, d_2)$	Make drawing which is $d_1$ on top of $d_2$
ROW2	$s : d_1 : d_2$	$s : \text{Row}(d_1, d_2)$	Make drawing which is $d_1$ to left of $d_2$
REPC $n$	$s : d$	$s : \text{RepC}(n, d)$	Make drawing which is column of $n$ copies of $d$
REPR $n$	$s : d$	$s : \text{RepR}(n, d)$	Make drawing which is row of $n$ copies of $d$
ROTL	$s : d$	$s : d_{\text{left}}$	Make drawing which is $d$ rotated left 90 degrees
ROTR	$s : d$	$s : d_{\text{right}}$	Make drawing which is $d$ rotated right 90 degrees
SWAP	$s : d_1 : d_2$	$s : d_2 : d_1$	Swap drawings on stack top
POP	$s : d$	$s$	Pop drawing from stack top
DUP	$s : d$	$s : d : d$	Duplicate drawing on stack top
VAR $i$	$s$	$s : s[sp - i]$	Access the variable $i$ positions below stack top
STOP	$s : d$	—	Halt the machine and show the drawing $d$

The stack machine may be implemented in Java as shown on the next page. The array  $p$  contains the program code. Execution starts at  $p[0]$ . The stack  $s$  contains Drawing objects, as defined below.

The stack machine uses this Java representation of drawings. A Drawing can be drawn with a specified upper left-hand corner  $(x_0, y_0)$  and a specified width  $w$  and height  $h$ . Also, a drawing can be rotated left and right:

```
abstract class Drawing {
    public abstract void draw(Graphics g, int x0, int y0, int w, int h);
    public abstract Drawing rotateLeft();
    public abstract Drawing rotateRight();
}
class Fill extends Drawing {
    private final int color;
    public Fill(int color) { this.color = color; }

    public void draw(Graphics g, int x0, int y0, int w, int h) {
        switch (color) {
            case 0: g.setColor(Color.black); break;
            case 1: g.setColor(Color.white); break;
            default: g.setColor(Color.yellow); break;
        }
        g.fillRect(x0, y0, w, h);
    }
    public Drawing rotateLeft() { return this; }
    public Drawing rotateRight() { return this; }
}

class Col extends Drawing {
    private final Drawing d1, d2;
    public Col(Drawing d1, Drawing d2) { this.d1 = d1; this.d2 = d2; }
    public void draw(Graphics g, int x0, int y0, int w, int h) {
        int h2 = (h+1)/2;
        d1.draw(g, x0, y0, w, h2);
        d2.draw(g, x0, y0+h2, w, h-h2);
    }
    public Drawing rotateLeft() {
        return new Row(d1.rotateLeft(), d2.rotateLeft());
    }
    public Drawing rotateRight() {
        return new Row(d2.rotateRight(), d1.rotateRight());
    }
}

class Row extends Drawing { ... } // Similar to Col
class RepC extends Drawing { ... } // Quite similar to Col
class RepR extends Drawing { ... } // Quite similar to Row
// End of Java classes for representing drawings
```

This is a possible Java implementation of the abstract machine for making drawings; it represents drawings using the classes from the preceding page:

```

/* Abstract stack machine for creating drawings: */
/* p[] is program, pc is program counter      */
/* s[] is stack, sp is stack top pointer      */

static void execcode(int[] p, Drawing[] s, int pc, int sp) {
    for (;;) {
        switch (p[pc++]) {
            case FILL:
                s[sp+1] = new Fill(p[pc++]); sp++; break;
            case COL2:
                s[sp-1] = new Col(s[sp-1], s[sp]);
                sp--; break;
            case ROW2:
                s[sp-1] = new Row(s[sp-1], s[sp]);
                sp--; break;
            case REPC:
                s[sp] = new RepC(p[pc++], s[sp]); break;
            case REPR:
                s[sp] = new RepR(p[pc++], s[sp]); break;
            case ROTL:
                s[sp] = s[sp].rotateLeft(); break;
            case ROTR:
                s[sp] = s[sp].rotateRight(); break;
            case SWAP:
                { Drawing tmp = s[sp]; s[sp] = s[sp-1]; s[sp-1] = tmp; } break;
            case POP:
                sp--; break;
            case DUP:
                s[sp+1] = s[sp]; sp++; break;
            case VAR:
                // question 3.4
                break;
            case STOP:
                showDrawing(s[sp]);           // Calls s[sp].draw(...)
                return;
            default:
                throw new RuntimeException("Illegal instruction " + p[pc-1] + " at " + (pc-1));
        }
    }
}

```

**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:

- 0: FILL 1
- 2: FILL 0
- 4: ROW2
- 5: FILL 0
- 7: COL2
- 8: STOP

**Question 3.2**

Write stack machine code to create the leftmost drawing shown at the beginning of question 2.

**Question 3.3**

The instructions of the drawing machine may be encoded in a Standard ML datatype as follows:

```
datatype instr =
  FILL of int                               (* 0 = black, 1 = white *)
  | COL2 | ROW2
  | REPC of int | REPR of int
  | ROTL | ROTR
  | SWAP | POP | DUP
  | VAR of int
  | STOP;
```

Write an SML function `board : int -> instr list` that generates a drawing machine program to create a chessboard with side length  $2n$ . That is, `board n` should generate code to make a checkered drawing with  $2n$  columns and  $2n$  rows.

For instance, executing the code generated by `board 4` should create the chessboard drawing shown in question 4.2 below.

Hint: First generate a drawing like the rightmost one in question 2, then use that drawing as a building block. You do not need to use the `DUP` or `VAR i` instructions (but you may if you want to).

**Question 3.4**

The Java implementation of the stack machine above does not show the implementation of the `VAR i` instruction. That instruction should read the contents of the  $i$ 'th stack element below the stack top and push that value onto the stack (incrementing the stack top pointer).

Since the stack top is the 0'th element below the stack top, `VAR 0` should be equivalent to `DUP`.

Show the necessary modifications to the Java implementation of the stack abstract machine on page 7.

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

**Question 4.1**

Consider first the abstract syntax for drawings (question 2 above) *without* variables and `let`-bindings. Write a compilation function `compile : drawing -> instr list` that compiles from the drawing abstract syntax, without variables and `let`-bindings, to a list of instructions for the abstract machine (question 3 above).

For instance,

```
< (4 * | (white | black))
```

could be compiled to

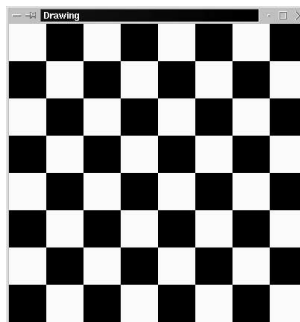
```
[FILL 1, FILL 0, ROW2, REPR 4, ROTL, STOP] : instr list
```

**Question 4.2**

This expression

```
let pair = white | black
in let square = pair - < < pair
  in 4 * | (4 *- square) end
end
```

creates a drawing of a standard chessboard:



Show a sequence of drawing machine instructions that correspond to the above expression. Use the `VAR i` instruction.

**Question 4.3**

Extend the compilation function from question 4.1 to compile the full abstract syntax, including variables and `let`-bindings. For instance,

```
let pair = white|black in pair - < < pair end
```

could be compiled to

```
[FILL 1, FILL 0, ROW2, VAR 0, VAR 1, ROTL, ROTL, COL2, SWAP, POP, STOP]
```