

**Possible solutions for
Written examination
29 June 2005**

Version 1 of 2004-06-29

Question 1 (30 %): Standard ML

Question 1.1

```
fun cellName (c, r) = letter c ^ Int.toString (r+1)
```

A fancier version that works also for column number 26 and greater (not required in the question) is this:

```
fun columnName c =
  if c < 26 then letter c
  else columnName(c div 26 - 1) ^ letter(c mod 26)
fun cellName (c, r) = columnName c ^ Int.toString (r+1)
```

Question 1.2

```
fun union ([], ys) = ys : set
  | union (x1::xr, ys) =
    if member x1 ys then
      union(xr, ys)
    else
      x1 :: union(xr, ys);

fun rectangle ((c1,r1), (c2,r2)) =
  let fun fromto i j = if i > j then [] else i :: fromto (i+1) j
      fun combine [] ys = []
        | combine (x1::xr) ys = map (fn y => (x1, y)) ys @ combine xr ys
      val cs = fromto c1 c2
      val rs = fromto r1 r2
    in combine cs rs end
```

Here's another solution for rectangle, using the `List.tabulate` function:

```
fun rectangle ((c1,r1), (c2,r2)) : set =
  List.concat(List.tabulate (c2-c1+1,
    fn c => List.tabulate(r2-r1+1,
      fn r => (c1+c, r1+r)))));
```

Question 1.3

```
fun showRef1 (Ref(ck, col, rk, row)) =
  let fun dollar Rel = ""
      | dollar Abs = "$"
    in
      dollar ck ^ letter col ^ dollar rk ^ Int.toString(row+1)
    end;
```

Question 1.4

```
fun move Rel delta value = value+delta
  | move Abs delta value = value

fun moveRef (dc, dr) (Ref(ck, col, rk, row)) : cellref =
  Ref(ck, move ck dc col, rk, move rk dr row)
```

Question 2 (25 %): Grammar and abstract syntax

Question 2.1

```

Formula ::=
    = Expr

Expr ::=
    Number
    | CellRef
    | CellRef:CellRef
    | Expr - Expr
    | Expr + Expr
    | Expr * Expr
    | f(ExprList)
    | ( Expr )

ExprList ::=
    <empty>
    | ExprList1

ExprList1 ::=
    Expr
    | Expr, ExprList1
    
```

Question 2.2

```

%left PLUS MINUS          /* lowest precedence */
%left TIMES                /* highest precedence */
    
```

Questions 2.3 and 2.4

```

Formula:
    EQUALS Expr EOF      { $2 }
;

Expr:
    Number                { Number $1 }
    | CELLREF              { Cell $1 }
    | CELLREF COLON CELLREF { Area($1, $3) }
    | Expr MINUS Expr      { Fun("-", [$1, $3]) }
    | Expr PLUS Expr       { Fun("+", [$1, $3]) }
    | Expr TIMES Expr      { Fun("*", [$1, $3]) }
    | NAME LPAR ExprList RPAR { Fun("$1, $3) }
    | LPAR Expr RPAR       { $2 }
;

ExprList :
    /* empty */          { [] }
    | ExprList1          { $1 }
;

ExprList1 :
    Expr                  { [$1] }
    | Expr SEMICOLON ExprList1 { $1 :: $3 }
;

Number:
    REAL                  { $1 }
    | MINUS REAL          { ~ $2 }
    
```

;

Questions 2.5 and 2.6

```

rule Token = parse
  [ '\t' '\n' '\r' ] { Token lexbuf }
| '=' { EQUALS }
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| ':' { COLON }
| ';' { SEMICOLON }
| '(' { LPAR }
| ')' { RPAR }
| '$'?'[A-Z]+'$?'[0-9]+'
  { CELLREF (convertRef (getLexeme lexbuf)) }
| [a-zA-Z][a-zA-Z0-9]*
  { NAME (getLexeme lexbuf) }
| [0-9]+(\.[0-9]+)?
  { case Real.fromString (getLexeme lexbuf) of
      NONE => lexerError lexbuf "internal error"
      | SOME i => REAL i
    }
| eof { EOF }
| _ { lexerError lexbuf "Illegal symbol in input" }
;

```

Question 3

Question 3.1

```

fun showExpr e =
  case e of
    Number n          => Real.toString n
  | Fun(f, [])        => f ^ "()"
  | Fun(f, e1::er)    =>
      let fun commaShow ex = ", " ^ showExpr ex
          in f ^ "(" ^ showExpr e1 ^ String.concat(List.map commaShow er) ^ ")" end
  | Cell cref         => showRef cref
  | Area (cref1, cref2) => showRef cref1 ^ ":" ^ showRef cref2;

```

Question 3.2

```

fun moveExpr (dc, dr) e : expr =
  case e of
    Number _      => e
  | Fun(f, es)    => Fun(f, List.map (moveExpr (dc, dr)) es)
  | Cell cref     => Cell(moveRef (dc, dr) cref)
  | Area (cref1, cref2) => Area(moveRef (dc, dr) cref1, moveRef (dc, dr) cref2) ;

```

Question 3.3

```

exception Shape;

fun check (e : expr) : shape =
  case e of
    Number _      => Single
  | Fun(ope, [e1, e2]) =>
      if ope="-" orelse ope="+" orelse ope="*" then
        case (check e1, check e2) of
          (Single, Single) => Single
        | _                => raise Shape
      else
        raise Shape
  | Fun("SUM", [e1]) =>
      (check e1; Single)
  | Fun("SIN", [e1]) =>
      (case check e1 of
        Single => Single
       | _     => raise Shape)
  | Fun _      => raise Shape
  | Cell _     => Single
  | Area _     => Multi

```

Question 3.4

```

exception Eval;

fun eval (sheet : expr list list) (e : expr) : real =
  case e of
    Number n      => n
  | Fun("-", [e1, e2]) => eval sheet e1 - eval sheet e2
  | Fun("+", [e1, e2]) => eval sheet e1 + eval sheet e2
  | Fun("*", [e1, e2]) => eval sheet e1 * eval sheet e2
  | Fun("SUM", [e1]) =>
    (case e1 of
      Area (Ref(_, c1, _, r1), Ref(_, c2, _, r2)) =>
        let val cells = rectangle ((c1, r1), (c2, r2))
            fun getval (col, row) = eval sheet (sheet sub (col, row))
        in
          List.foldr (op+) 0.0 (List.map getval cells)
        end
      | _ => raise Eval)
  | Fun("SIN", [e1]) =>
    Math.sin(eval sheet e1)
  | Fun _ => raise Eval
  | Cell (Ref(_, col, _, row)) => eval sheet (sheet sub (col, row))
  | Area _ => raise Eval

```