

**Possible solutions for
Trial written examination
23 November 2009**

Version 1.0 of 2009-12-06

Question 1**Question 1.1**

Some example strings that are recognized: *ab, abbb, abbbb*. In general, precisely those strings consisting of an *a* followed by an odd number of *b*'s are recognized.

Question 1.2**Question 1.3**

The DFA has three states *S1, S2* and *S3*, where *S1* is the starting state and *S3* is the only accepting state. There is a transition from *S1* on *a* to *S2*; a transition from *S2* on *b* to *S3*, and a transition from *S3* on *b* back to *S2*.

Question 2**Question 2.1**

```
let cellName (c, r) = letter c + string (r+1);;
```

Question 2.2

```
let rec union xs ys =
  match xs with
  | [] -> ys
  | cell::xr -> if mem cell ys then union xr ys
                 else cell :: union xr ys;;

let rec rectaux c r1 r2 =
  if r1>r2 then [] else (c,r1) :: rectaux c (r1+1) r2
let rec rectangle (c1,r1) (c2,r2) =
  if c1>c2 then []
  else rectaux c1 r1 r2 @ rectangle (c1+1,r1) (c2,r2);;
```

Note the use of auxiliary function `rectaux`; you can always introduce such auxiliary functions if helpful.

Question 2.3

```
let showRef (Ref(absCol, col, absRow, row)) =
  match (absCol, absRow) with
  | (Rel, Rel) -> letter col + string row
  | (Rel, Abs) -> letter col + "$" + string row
  | (Abs, Rel) -> "$" + letter col + string row
  | (Abs, Abs) -> "$" + letter col + "$" + string row;;
```

Question 2.4

```
let moveRef (dc, dr) (Ref(absCol, col, absRow, row)) =
  Ref(absCol, (match absCol with | Abs -> col | Rel -> col+dc),
      absRow, (match absRow with | Abs -> row | Rel -> row+dr));;
```

or, maybe more transparently, because more similar to showRef:

```
let moveRef2 (dc, dr) (Ref(absCol, col, absRow, row)) =
  match (absCol, absRow) with
  | (Rel, Rel) -> Ref(absCol, col+dc, absRow, row+dr)
  | (Rel, Abs) -> Ref(absCol, col+dc, absRow, row)
  | (Abs, Rel) -> Ref(absCol, col, absRow, row+dr)
  | (Abs, Abs) -> Ref(absCol, col, absRow, row)
```

Question 3

Question 3.1

```
Formula ::=
  = Expr

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

ExprList ::=
  <empty>
  | ExprList1

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

Question 3.2

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

Questions 3.3 and 3.4

```
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:
  FLOAT                               { $1                }
| MINUS FLOAT                         { -$2               }
;

```

Questions 3.5 and 3.6

```

rule Token = parse
| [' '\t' '\n' '\r'] { Token lexbuf }
| '='                { EQUALS      }
| '+'                { PLUS        }
| '-'                { MINUS       }
| '*'                { TIMES       }
| ':'                { COLON       }
| ';'                { SEMICOLON   }
| '('                { LPAR        }
| ')'                { RPAR        }
| '$'?'[A-Z]+'$'?'[0-9]'+
                        { CELLREF (convertRef (lexemeAsString lexbuf)) }
| ['a-z'A-Z'] ['a-z'A-Z'0-9']*
                        { NAME (lexemeAsString lexbuf) }
| [0-9]+'(.'[0-9]'+)?
                        { System.Double.Parse (lexemeAsString lexbuf) }
| eof                  { EOF      }
| _                    { lexerError lexbuf "Illegal symbol in input" }

```

Question 4**Question 4.1**

```

let rec moveExpr (dc,dr) e : expr =
  match e with
  | Number _           -> e
  | Fun(f, es)         -> Fun(f, List.map (moveExpr (dc,dr)) es)
  | Cell cellref       -> Cell (moveRef (dc,dr) cellref)
  | Area(cellref1, cellref2) -> Area (moveRef (dc,dr) cellref1,
                                       moveRef (dc,dr) cellref2);;

```

Question 4.2

```

let rec check e =
  match e with
  | Number _           -> Single
  | Fun("+", [e1; e2]) -> if check e1 = Single && check e2 = Single then Single
                          else failwith "Error in +"
  | Fun("*", [e1; e2]) -> if check e1 = Single && check e2 = Single then Single
                          else failwith "Error in *"
  | Fun("SUM", [e1])   -> if check e1 = Multi then Single
                          else failwith "Error in SUM"
  | Fun("SIN", [e1])   -> if check e1 = Single then Single
                          else failwith "Error in SIN"
  | Fun(_, _)         -> failwith "Unknown function or error in function"
  | Cell cellref      -> Single
  | Area(cellref1, cellref2) -> Multi;;

```

Question 4.3

```

let getCell sheet (c,r) = List.nth (List.nth sheet r) c;;

let rec eval sheet e =
  match e with
  | Number x           -> x
  | Fun("+", [e1; e2]) -> eval sheet e1 + eval sheet e2
  | Fun("*", [e1; e2]) -> eval sheet e1 * eval sheet e2
  | Fun("SIN", [e1])   -> sin(eval sheet e1)
  | Fun("SUM", [Area(Ref(_,c1,_,r1),Ref(_,c2,_,r2))]) ->
    let cellsAddr = rectangle (c1,r2) (c2,r2)
    let exprs = List.map (getCell sheet) cellsAddr
    let vals = List.map (eval sheet) exprs
    in List.fold (fun x y -> x+y) 0.0 vals
  | Fun(_, _)         -> failwith "Error in function"
  | Cell(Ref(_,c,_,r)) -> eval sheet (getCell sheet (c,r))
  | Area(cellref1, cellref2) -> failwith "Expression cannot be Area";;

```