Written examination
29 June 2005

This examination comprises 7 pages. Please check immediately that you have a complete set of questions. There are three 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 or that are connected to any kind of network.

As a rule, if one question asks you to define a particular function, then you may use that function in subsequent questions, regardless of whether you were able to define it.

Question 1 (30 %): Standard ML

A spreadsheet is a rectangular grid of cells, where each cell may contain a number (constant) or a formula. Columns are named by letters A, B, . . . , Z, AA, AB, . . . , and rows are named by numbers 1, 2, . . . , so cells are named A1, A2, . . . , B1, B2, . . . and so on, as shown in figure 1.

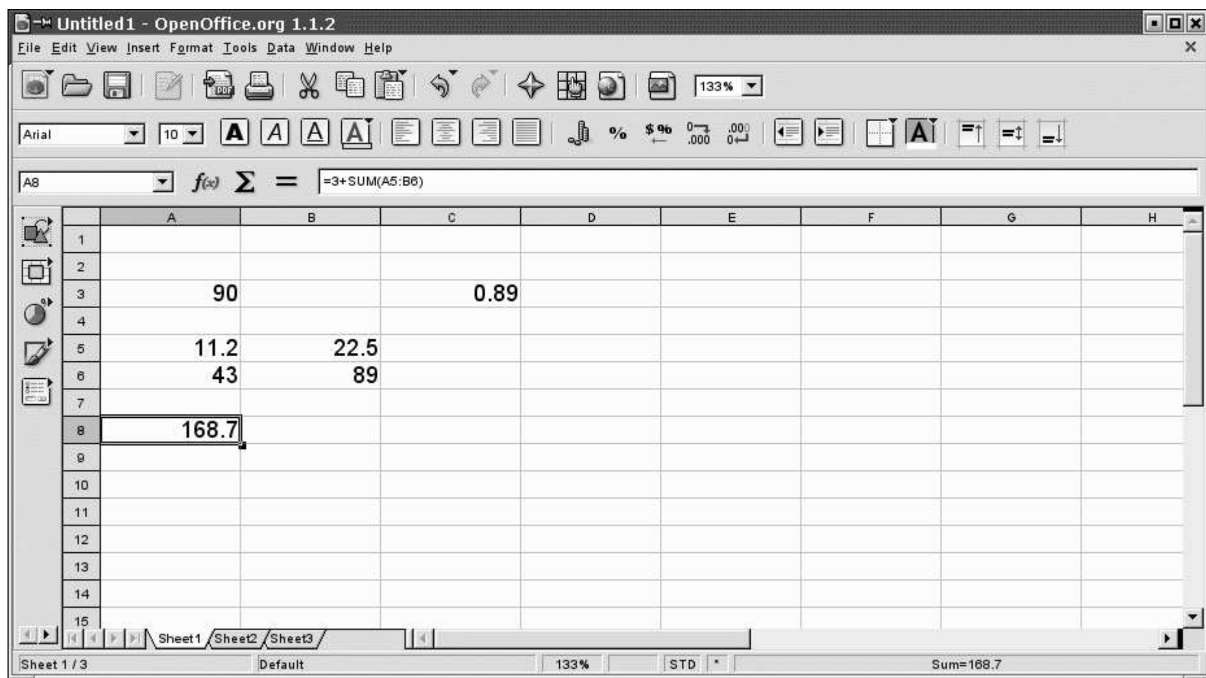

Figure 1: Spreadsheet with numbers and formulas.

Cell A3 contains the number 90, and cells A5, A6, B5 and B6 contain the numbers shown in the figure.

Cell A8, which is shown framed because it has focus, contains the spreadsheet formula `=3+SUM(A5:B6)`. This formula can be seen in the formula window (next to Σ =) above the cell grid. The cell itself just shows the value 168.7 of that formula.

Cell C3 has the value 0.89. In fact, it contains the formula `=SIN(A3)`, but this could be seen only by moving focus to cell C3.

The questions of this examination concern the representation and use of the formulas contained in spreadsheet cells. All other properties of spreadsheets – icons, menus and other gadgets – are irrelevant.

Question 1.1

Internally in a spreadsheet, the *address* of a cell is a pair of integers (column, row), where (0, 0) is the address of the top left cell. For instance, cell A5 has address (0, 4), and cell F1 has address (5,0).

Write a Standard ML function `cellName : int * int -> string` to convert a cell address `(column, row)` into a cell name. The function needs to work only for the column numbers 0–25. You may use the function `Int.toString : int -> string` and you may assume the existence of a function `letter : int -> string` that behaves as shown in this table:

Call	Result
`letter(0)`	`"A"`
`letter(1)`	`"B"`
...	...
`letter(25)`	`"Z"`

Question 1.2

For certain checks on a spreadsheet, one needs to work on sets of cell addresses. Let such a set be represented by a value of type `(int * int) list`. Then the empty set of cell addresses is represented by value `empty`, a one-element set containing a cell address can be created by function `singleton`, and function `member` can be used to test whether a given cell address is a member of a given set:

```
val empty = []
fun singleton x = [x]
fun member x []       = false
  | member x (x1::xr) = x=x1 orelse member x xr;
```

Define the function `union` such that `union(xs, ys)` is a set containing all the members of `xs` and all the members of `ys`. The resulting list must contain no duplicates, provided neither `xs` nor `ys` contains any duplicates. The function must have this type:

```
val union : (int * int) list * (int * int) list -> (int * int) list
```

Define the function `rectangle` so that `rectangle` $((c_1, r_1), (c_2, r_2))$ is a set containing the cell addresses of all cells in the rectangle whose top left corner is (c_1, r_1) and whose bottom right corner is (c_2, r_2). You may assume that $c_1 \leq c_2$ and $r_1 \leq r_2$. The type of the function should be:

```
val rectangle : (int * int) * (int * int) -> (int * int) list
```

The order of the elements of the result list does not matter. For instance, the call `rectangle ((0,4),(1,5))` should return a four-element list such as `[(0, 4), (0, 5), (1, 4), (1, 5)]`, but may return any list that contains the same four elements in some other order.

Question 1.3

A spreadsheet cell may contain a spreadsheet formula. The structure of spreadsheet formulas is explained in more detail in question 2 below.

In this question the only thing that matters is that a spreadsheet formula may contain *cell references* that refer to other spreadsheet cells. Such a reference is a cell name whose column name and row name may be decorated with a dollar sign (`$`). The `$`-decoration does not affect what cell is referred to in the given formula, but determines how the column name and row name are adjusted when the formula is copied to another cell.

Namely, a cell reference of the form `A3` is a *relative reference*, so that if a formula containing this reference is copied down one row, then the cell reference gets adjusted to `A4`. If the formula is copied right one column, then the cell reference gets adjusted to `B3`. If the formula is copied down one row and right one column, then the cell reference gets adjusted to `B4`.

A cell reference of the form `$A$3` is an *absolute reference*, which does not get adjusted regardless how the formula containing the reference is copied.

A cell reference of the form `$A3` is *column absolute*, so only the row number gets adjusted when the formula is copied. Conversely, `A$3` is *row absolute* and only the column number gets adjusted when the formula is copied.

In summary:

Cell reference	**Kind**	**Effect when copying the formula**
`A3`	Relative	Both column and row are adjusted
`$A$3`	Absolute	Neither column nor row is adjusted
`$A3`	Column absolute	Only row is adjusted
`A$3`	Row absolute	Only column is adjusted

Cell references may be represented like this:

```
datatype kind =
    Rel
  | Abs
datatype cellref =
  Ref of kind * int * kind * int              (* (column,  row) *)
```

For instance, cell reference `$A3` is represented as `Ref(Abs, 0, Rel, 2)`, and cell reference `A$3` is represented as `Ref(Rel, 0, Abs, 2)`.

Write a function `showRef : cellref -> string` that converts a cell reference to the corresponding cell name: a string containing column name and row name with `$`-decorations where required.

Question 1.4

Define a function `moveRef : int * int -> cellref -> cellref` such that `moveRef (dc, dr) cref` returns the cell reference that results when copying `cref` right by `dc` columns and down by `dr` rows.

For instance, `moveRef (1,1) (Ref(Abs, 0, Rel, 2))` should give `Ref(Abs, 0, Rel, 3)`. This corresponds to copying a formula containing cell reference `$A3` right by one column and down by one row, so that the cell reference becomes `$A4` . Note that `dc` and `dr` may be negative, zero or positive; negative movement means ‘left’ or ‘up’.

Question 2 (40 %): Grammar and abstract syntax

This question and the following ones concern the expression language used in spreadsheet formulas.

A spreadsheet formula has the form `=`e where e is a spreadsheet expression. A spreadsheet expression is either a number, a cell reference, an area reference, an operator with two operand expressions, a function application, or an expression within parentheses.

The form and meaning of expressions are explained by the table below. Let $e, e_1, e_2, \ldots, e_n$ be expressions.

Expression	Meaning	Examples
d	The floating-point number d	`3.14   23   -42`
cr	Cell reference (column c row r)	`A3  $A3  A$3  $A$3  AA52`
cr`:`cr	Area reference (top left cell, bottom right cell)	`A5:B6  $A5:F7`
e_1 `+` e_2	The sum of e_1 and e_2	`3.15+A12`
e_1 `-` e_2	The difference of e_1 and e_2	`3.15-A12`
e_1 `*` e_2	The product of e_1 and e_2	`3.15*A12`
f`(`e_1`;  ...;` e_n`)`	The function f applied to cells $e_1, \ldots, e_n$	`SIN(A2)  SUM(A5:B6)`
`(`e`)`	The value of e	`(3+SUM(A5:B6))`

In a function call f`(`e_1`;  ...;` e_n`)` we have $n \geq 0$; in particular, the number of argument expressions may be zero. Here are some example formulas and their meaning:

- The formula `=A3` has the same value as cell A3.
- The formula `=$A$3` has the same value as cell A3. The only difference from the preceding formula is that this formula will remain `=$A$3` even if copied to another cell (as explained in question 1.3).
- The formula `=A3+5` computes the sum of the value of cell A3 and the number 5.
- The formula `=SUM(A5:B6)` computes the sum of the values of cells A5, A6, B5 and B6, where `A5:B6` is an area reference.
- The formula `=3+SUM(A5:B6)` computes the sum of the number 3 and the values of cells A5, A6, B5 and B6.
- The formula `=3+SUM(A5:B6)*4` computes the sum of the number 3 and four times the sum of the values of cells A5, A6, B5 and B6.
- The formula `=SIN(SUM(A5:B6)*3.1415)` computes the sine of 3.1415 times the sum of the values of cells A5, A6, B5 and B6.
- The formula `=1-2-3` computes $(1 - 2) - 3$, so the result is -4.

Here is an abstract syntax for spreadsheet expressions (from a file called `Absyn.sml`):

```
datatype expr =
    Number of real                    (* Floating-point constant           *)
  | Fun of string * expr list         (* Function application              *)
  | Cell of cellref                   (* Cell reference                    *)
  | Area of cellref * cellref         (* Rectangle (topleft, bottomright) *)
```

Expressions of the form e_1`+`e_2 are represented as function applications `Fun("+", [`e_1`, `e_2`])` in this abstract syntax, and similarly for the other infix operators e_1`-`e_2 and e_1`*`e_2.

Question 2.1

Write a context-free grammar for expressions. It must be able to generate all the expression forms shown in the table on page 4 and in the examples below it.

Question 2.2

Assume that the following token declarations are available in a `mosmlyac` parser specification (`.grm` file) for spreadsheet formulas and expressions. The token `REAL` represents a floating-point constant, and `CELLREF` represents a cell reference of type `cellref` shown in question 1.3:

```
%token <real> REAL
%token <string> NAME
%token <Absyn.cellref> CELLREF
%token EQUALS PLUS MINUS TIMES COLON SEMICOLON LPAR RPAR
%token EOF
```

Write the operator precedence and associativity part of the same parser specification. The multiplication operator '`*`' should have higher precedence (bind more strongly) than '`+`' and '`-`'. All these operators associate to the left.

Question 2.3

Write the rule part of the parser specification for formulas and expressions. You may leave the semantic actions `{ ... }` empty in this question. The rule part could have this form:

```
Formula:
    ...                                 { ... }
;

Expr:
    ...                                 { ... }
;

... rules for more nonterminal symbols ...
```

Question 2.4

Extend the parser specification from question 2.3 with semantic actions within `{ ... }`, so that the rules for `Formula` and `Expr` construct values of type `expr`, the abstract syntax type defined on page 4.

Question 2.5

As indicated by the parser's token declarations in question 2.2, the tokens of a spreadsheet formula are the following:

Token name	Meaning	Examples
`REAL`	A non-negative floating-point constant	`3.14` `3` `213`
`NAME`	A function name	`SIN` `SUM` `LOG10`
`CELLREF`	A cell reference	`A3` `$A$3` `$A3` `A$3` `UB40`
`EQUALS`	Equals sign	`=`
`PLUS`	Plus sign	`+`
`MINUS`	Minus sign	`-`
`TIMES`	Multiplication sign	`*`
`COLON`	Colon	`:`
`SEMICOLON`	Semicolon	`;`
`LPAR`	Left parenthesis	`(`
`RPAR`	Right parenthesis	`)`

Write the `rule` part of a `mosmllex` lexer specification for spreadsheet formulas, that is, the left hand side of a declaration of this form:

```
rule Token = parse
    ...                               { ... }
  | ...
```

Question 2.6

Write the 'semantic actions' `{ ... }` of the lexer specification from question 2.5.

You may assume that there is a function `convertRef : string -> cellref` that takes as argument a well-formed cell reference string such as `"$A3"` and returns a cell reference such as `Ref(Abs, 0, Rel, 2)` of type `cellref` (from question 1.3).

Question 3 (30 %): Manipulation and checking of expressions

This question concerns manipulation and checking of expressions, using the abstract syntax type `expr` as defined on page 4.

Question 3.1

Write a function `show : expr -> string` that displays a spreadsheet formula as a string. You may write all function applications in prefix form, for instance `+(1.0,$A3)` instead of `1.0+$A3`.

Question 3.2

Write a function `moveExpr : int * int -> expr -> expr` for adjusting any cell references contained in an expression, by column and row offsets. That is, `moveExpr (dc, dr) e` should return a new expression in which all cell references have been moved according to the offsets `dc` and `dr`, as explained in question 1.3.

For instance, `moveExpr (1,1) (Cell (Ref(Rel, 0, Abs, 3)))` should return `Cell(Ref(Rel, 1, Abs, 3))`.

Question 3.3

In an expression, functions must be applied to arguments of the correct *shape*. The shape of an argument is either `Single`, representing a single cell or number, or `Multi`, representing a rectangular area of the spreadsheet.

```
datatype shape =
    Single
  | Multi
```

For example, the subtraction, addition and multiplication functions (`-`, `+`, `*`) must be applied to two arguments of shape `Single`, and will produce a result of shape `Single`. Function `SUM` must be applied to one argument of type `Multi` and produces a result of shape `Single`. Function `SIN` must be applied to one argument of type `Single` and produces a result of shape `Single`.

You need not consider any other functions besides `+`, `-`, `*`, `SUM` and `SIN`.

Write a function `check : expr -> shape` to find the result shape of a formula. The function must throw exception `Shape` if a function is applied to the wrong shape of argument anywhere in the formula.

Question 3.4

A spreadsheet can be represented as a list of lists of expressions, where the inner lists correspond to the rows of the sheet.

Write an ML function `eval : expr list list -> expr -> real` such that `eval sheet e` computes the result of evaluating expression `e`.

Note that the value of the expression stored in a spreadsheet cell must be a `real`. Hence the expression cannot be just an area reference, which would be multi-values, but may well be just a cell reference.

The `eval` function must throw exception `Eval` with a string argument if evaluation of expression `e` goes wrong; for instance, if it attempts to apply function "`*`" to less than two arguments or more than two arguments.

Hint: If `sheet` has type `expr list list`, then expression `List.nth(List.nth(sheet, r), c)` can be used to get the cell in column `c` of row `r` in the list `sheet`.

Hint: As experienced spreadsheet users know, a spreadsheet may continue a circularity: a formula that depends on itself, perhaps indirectly. Your `eval` function can ignore this problem.