

Written examination

9 June 2004

This examination comprises 6 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 or that are connected to any kind of network.

Question 1 (25 %): Standard ML

Question 1.1

Here are four Standard ML functions, and six declarations of variables:

```
fun add k [] = []
  | add k (x::xr) = (k + x) :: add k xr

fun mul k [] = []
  | mul k (x::xr) = (k * x) :: mul k xr

fun fromTo m n = if m > n then [] else m :: fromTo (m+1) n

fun tabulate(m, f) = if m < 0 then [] else f m :: tabulate(m-1, f)

val res1 = fromTo 2 5
val res2 = mul 3 res1
val res3 = fromTo 5 2
val res4 = fromTo 1 0
val res5 = (res3 = res4)
val res6 = tabulate(5, fn x => x * x)
```

For each of the four functions `add`, `mul`, `fromTo`, and `tabulate`, show the type of the function and briefly explain what the function does.

For each of the six variables, show its type and value.

Question 1.2

Define each of the above functions `add` and `mul` using the higher order function

`map : ('a -> 'b) -> 'a list -> 'b list` from the Standard ML `List` structure.

Define the function `fromTo` using `tabulate` from above.

Question 1.3

This question and the next ones concern regularly spaced integer sequences such as $\langle 7, 14, 21, 28 \rangle$ or $\langle -2, 0, 2, 4, 6 \rangle$ or $\langle 12, 9, 6 \rangle$ where the difference between two successive elements is the same in the entire sequence. In the example sequences the difference is 7 and 2 and -3 respectively.

Note in particular that the empty sequence $\langle \rangle$, and any one-element sequence such as $\langle 1 \rangle$ or $\langle 9 \rangle$, and any constant sequence such as $\langle 13, 13, 13 \rangle$ is regularly spaced.

The general form of a regularly spaced integer sequence is $\langle a, a + i, a + 2i, \dots, a + (k - 1)i \rangle$ where a and i and k are integers, $k \geq 0$. Therefore any regularly spaced integer sequence can be represented by a triple (a, i, k) of the start value a , the increment i , and the number of elements k . Conversely, any triple (a, i, k) with $k \geq 0$ represents a regularly spaced integer sequence. Some examples:

Sequence	Representation (a, i, k)
$\langle 7, 14, 21, 28 \rangle$	is represented by $(7, 7, 4)$
$\langle -2, 0, 2, 4, 6 \rangle$	is represented by $(-2, 2, 5)$
$\langle 12, 9, 6 \rangle$	is represented by $(12, -3, 3)$
$\langle \rangle$	can be represented by $(0, 0, 0)$ and $(1, 1, 0)$ and by any triple of form $(a, i, 0)$
$\langle 1 \rangle$	can be represented by $(1, 1, 1)$ or $(1, 17, 1)$ or by any triple of form $(1, i, 1)$
$\langle 9 \rangle$	can be represented by $(9, 1, 1)$ or $(9, 17, 1)$ or by any triple of form $(9, i, 1)$
$\langle 1, 2, 3, 4, 5, 6 \rangle$	is represented by $(1, 1, 6)$
$\langle 13, 13, 13 \rangle$	is represented by $(13, 0, 3)$

Write an SML function `sequence : int * int * int -> int list` such that the application `sequence (a, i, k)` produces the regularly spaced sequence represented by the triple. The result should be an SML integer list.

Question 1.4

Define an SML function `isRegSeq : int list -> bool` that checks whether a given list is a regularly spaced integer sequence as described above. For example, `isRegSeq []` and `isRegSeq [9]` and `isRegSeq [7, 14, 21, 28]` should be true, whereas `isRegSeq [7, 14, 20]` should be false.

Question 1.5

Define an SML function `getRegSeq : int list -> int * int * int` such that `getRegSeq xs` either returns a triple (a, i, k) that represents the sequence `xs`, or else throws the exception `Fail "not regular"`.

Thus when `getRegSeq xs` does not throw an exception but returns (a, i, k) , it should hold that `sequence (a, i, k) = xs`.

Question 2 (25 %): Grammar and abstract syntax

This and the following questions concern a small expression language in which an expression evaluates to either a single integer, or an integer sequence (but not necessarily a regularly spaced integer sequence).

The form and meaning of expressions are explained by the table below. Let e , e_1 and e_2 be expressions, and let m and n be signed integer constants such as 5 or 0 or -17:

Expression	Meaning
m	The integer m
$[]$	The empty sequence $\langle \rangle$
$[m : n]$	The sequence $\langle m, m + 1, \dots, n \rangle$, empty if $m > n$
$e_1 + e_2$	The sum of e_1 and e_2 , where at most one of e_1 and e_2 may be a sequence
$e_1 * e_2$	The product of e_1 and e_2 , where at most one of e_1 and e_2 may be a sequence
$e_1 ++ e_2$	The concatenation of e_1 and e_2 ; both e_1 and e_2 must be sequences
$\min e$	The minimal value in the sequence e ; or 1073741823 if the sequence is empty
$\max e$	The maximal value in the sequence e ; or -1073741824 if the sequence is empty
$\text{rev } e$	The reversal of e , which must be a sequence
(e)	The value of e

In an addition $e_1 + e_2$ or multiplication $e_1 * e_2$, the result is an integer if both operands evaluate to integers; the result is a sequence if one of the operands evaluates to a sequence; and the expression is illegal if both operands evaluate to a sequence.

In a concatenation $e_1 ++ e_2$ both operands must evaluate to a sequence, and then the result is a sequence.

In $\max e$ and $\min e$ and $\text{rev } e$, the operand e must evaluate to a sequence.

Some example expressions and their values:

- The expression $[2 : 5]$ produces the sequence $\langle 2, 3, 4, 5 \rangle$.
- The expression $[5 : 2]$ produces the empty sequence $\langle \rangle$.
- The expression $3 + [2 : 5]$ produces the sequence $\langle 5, 6, 7, 8 \rangle$.
- The expression $3 + 2 * [2 : 5]$ produces the sequence $\langle 7, 9, 11, 13 \rangle$.
- The expression $3 + 2 * [2 : 5] ++ [2 : 4]$ produces the sequence $\langle 7, 9, 11, 13, 2, 3, 4 \rangle$.
- The expression $\max (3 + 2 * [2 : 5])$ produces the integer 13.
- The expression $\min (3 + 2 * [2 : 5])$ produces the integer 7.
- The expression $\text{rev } (3 + 2 * [2 : 5])$ produces the sequence $\langle 13, 11, 9, 7 \rangle$.

Here is an abstract syntax for expressions:

```

datatype expr =
  CstI of int                (* Integer constant          *)
| FromTo of int * int        (* Sequence (from, to)      *)
| Empty                      (* Empty sequence           *)
| Add of expr * expr         (* Sum of ints or int and seq *)
| Mul of expr * expr         (* Product                   *)
| Append of expr * expr      (* Concatenation of sequences *)
| Max of expr                (* Maximum of sequence       *)
| Min of expr                (* Minimum of sequence       *)
| Rev of expr                (* Reverse of sequence        *)

```

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 3. A single nonterminal symbol suffices.

Question 2.2

Write the operator precedence and associativity part of the `mosmlyac` parser specification (`.grm` file). The multiplication operator `*` should have higher precedence (bind more strongly) than `+`, which has higher precedence than `++`. All operators associate to the left.

You may assume that the following token declarations are available in the parser specification already, where `INT` represents non-negative integer constants m and n :

```
%token <int> INT
%token MAX MIN REV
%token MINUS PLUS TIMES PLUSPLUS COLON LBRACKET RBRACKET LPAR RPAR
%token EOF
```

Question 2.3

Write the rule part of a `mosmlyac` parser specification for expressions. Hint: To make sure that `min e1 + e2` is parsed as `(min e1) + e2`, not `min (e1 + e2)`, use a separate nonterminal `AtExpr` for the atomic expressions such as `5`, `-2`, `[]`, `[m:n]` and `(e)`.

The rule part should have this form:

```
Main:
    Expr EOF          { ... }
;
Expr:
    AtExpr           { ... }
    | MIN AtExpr     { ... }
    ...
;
AtExpr:
    ...             { ... }
    | ...           { ... }
    ...
;
Int:
    ...            { ... }
    | ...          { ... }
;
```

Question 2.4

Extend the parser specification from question 2.3 with semantic actions, so that each rule generates a Standard ML value of type `expr`, the abstract syntax type shown on page 3.

Question 3 (25 %): Evaluation and type checking

This question concerns evaluation and type checking of expressions as defined on page 3, using the abstract syntax type `expr`.

Question 3.1

Evaluation of an expression should produce either an integer or an integer sequence. The two kinds of results can be represented by the type `value`:

```
datatype value =  
  Int of int  
  | Seq of int list
```

Write an ML function `eval : expr -> value` such that `eval e` computes the result — which must be an integer or an integer sequence — described by expression `e`. Note that the expression language has no variables and therefore the only argument to the `eval` function is the expression `e`; there is no need for an environment.

The `eval` function must throw exception `Eval` with a string argument if evaluation of expression `e` goes wrong; for instance, if it attempts to add two integer sequences to each other. Assume that the exception `Eval` is declared as follows:

```
exception Eval of string
```

Question 3.2

Write a type checker for expressions as defined on page 3. The type of an expression is either ‘integer’ or ‘integer sequence’, represented by `TypI` and `TypS` below:

```
datatype typ =  
  TypI  
  | TypS
```

Write an ML function `check : expr -> typ` such that `check e` finds the type of expression `e`.

The `check` function must throw exception `Type` with a string argument if the expression `e` is ill-typed; for instance, if it attempts to add two integer sequences to each other. Assume that exception `Type` is declared like this:

```
exception Type of string
```

The intention is that if `check e` evaluates to a type without throwing exception `Type`, then `eval e` will evaluate to a value without throwing exception `Eval`.

Question 4 (25 %): Simplification

This question concerns simplification of expressions as defined on page 3, using the abstract syntax type `expr`.

Many expressions have the same value. Often a complex expression can be simplified to obtain a simpler one that has the same value. For instance, the expression `0 + [5:9]` has the same value as the simpler expression `[5:9]`, and the expression `3 * max [17:101]` has the same value as the simpler expression `303`.

This table lists some possible simplifications:

<code>0 + e</code>	can be simplified to	<code>e</code>	
<code>1 * e</code>	can be simplified to	<code>e</code>	
<code>k + [m:n]</code>	can be simplified to	<code>[k+m:k+n]</code>	when <code>k</code> an integer constant
<code>[m:n]</code>	can be simplified to	<code>[]</code>	when <code>m > n</code>
<code>max []</code>	can be simplified to	<code>-1073741824</code>	
<code>min []</code>	can be simplified to	<code>1073741823</code>	
<code>min [m:n]</code>	can be simplified to	<code>m</code>	when <code>m <= n</code>
<code>max [m:n]</code>	can be simplified to	<code>n</code>	when <code>m <= n</code>
<code>rev []</code>	can be simplified to	<code>[]</code>	
<code>rev (rev e)</code>	can be simplified to	<code>e</code>	

Note: Some of the above simplification rules assume that integer arithmetics does not silently wrap around on overflow (as in Java/C/C++), but raises an exception, as in Standard ML and in C# checked contexts. You can ignore this subtlety below.

Question 4.1

Propose four more simplifications not listed in the table above. Briefly explain for each simplification why it is correct.

Question 4.2

Write a function `simplify : expr -> expr` so that `simplify e` is an expression that has the same value as `e`, but simpler than `e` when possible. Your simplifier should implement all the simplifications shown in the table above, and should be able to simplify the expression `3 * max [17:101]` to the expression `303`.

Question 4.3

Some simplifications may look plausible, but turn out to be wrong upon closer inspection. For each of the following wrong simplifications, give an example to show that the simplification is wrong. To give an example, it is sufficient to choose suitable integer constants to insert instead of `k`, `m` and `n`, and show that the ‘simplified’ expression has a different value than the original one.

<code>k * [m:n]</code>	can be ‘simplified’ to	<code>[k*m:k*n]</code>	when <code>k</code> is an integer constant
<code>min [m:n]</code>	can be ‘simplified’ to	<code>m</code>	
<code>max [m:n]</code>	can be ‘simplified’ to	<code>n</code>	
<code>rev [m:n]</code>	can be ‘simplified’ to	<code>[n:m]</code>	