

Written examination

19 June 2002 *

There are three questions all of whose subquestions should be satisfactorily answered to get full marks.

Question 1 (30 %)

Question 1.1

Note: When a subquestion says ‘Define a function $f(\dots)$ so that ...’ and a later question says ‘Use function $f(\dots)$ from above ...’, you can always take the function f for granted when answering the later subquestion, even if you were unable to answer the former one, and you can still get full marks for the later one. This holds for questions 2 and 3 also.

Here are five Standard ML functions on lists:

```

fun sum [] = 0
  | sum (x::xr) = x + sum xr
fun sumlist [] = []
  | sumlist (xs::xsr) = sum xs :: sumlist xsr
fun sumsumlist [] = 0
  | sumsumlist (xs::xsr) = sum xs + sumsumlist xsr
fun map f [] = []
  | map f (x::xr) = f x :: map f xr
fun maplist g [] = []
  | maplist g (xs::xsr) = (map g xs) :: maplist g xsr

```

For each function, show the most general (most polymorphic) type of the function, and briefly explain what the function does.

Question 1.2

Write an SML function `doublelist : int list list -> int list list` that takes a list of integer lists and multiplies each element of the inner lists by 2. For instance, `doublelist [[4, 7], [11]]` should give `[[8, 14], [22]]`.

Question 1.3

An integer list $[x_1, x_2, \dots, x_n]$ is *sorted* if the list has 0 or 1 elements, or if $x_1 \leq x_2 \leq \dots \leq x_n$. Define a function `sorted : int list -> bool` so that `sorted xs` returns true if `xs` is sorted, and false otherwise.

Define a function `sortedlists : int list list -> bool` so that `sortedlists xss` returns true if all lists in `xss` are sorted.

Question 1.4

Define a function `merge : int list -> int list -> int list` so that `merge(xs, ys)` merges the lists `xs` and `ys`. The resulting list must contain all the elements of `xs` and `ys`, and if both `xs` and `ys` are sorted, then `merge(xs, ys)` must be sorted also.

Question 1.5

Define a function `mergelists : int list list -> int list` so that `mergelists xss` merges all the inner lists in `xss`, in some order.

One possibility is to define `mergelists xss` so that if `xss` is $[xs_1, xs_2, \dots, xs_{n-1}, xs_n]$, then `mergelists xss` is equivalent to `merge(xs1, merge(xs2, ..., merge(xsn-1, xsn...))`.

Question 1.6

First, define a function `singletons : 'a list -> 'a list list` that splits a list of values into a list of one-element lists each containing one of those values. For instance, `singletons [7, 13, 9]` should give `[[7], [13], [9]]`.

Secondly, use `singletons` and `mergelists` to define a sorting function `sort : int list -> int list` in a single line of Standard ML code.

Question 2 (30 %)

In an electronic calendar system, or in an accounting system, one often needs to specify recurring appointment or event dates such as these:

```
every day
every Friday
first day of the month
tenth day of the month
first Tuesday of the month
second Tuesday of the month
last day of the month
last Friday of the month
second last day of the month
```

and so on. Such appointment dates can be described using this abstract syntax:

```
datatype day =
    Day | Mon | Tue | Wed | Thu | Fri | Sat | Sun;

datatype apptdate =
    Every of day
  | First of int * day
  | Last of int * day
```

In type `day`, the intention is that `Day` means any day, whereas each of `Mon`, `Tue`, ..., `Sun` indicates a particular day of the week.

In type `apptdate`, the intention is that `First(i, Day)` means the *i*'th day of the month, and for instance `First(i, Mon)` means the *i*'th Monday of the month, and similarly for the other weekdays. Correspondingly, `Last(i, Day)` is the *i*'th last day of the month, and `Last(i, Mon)` is the *i*'th last Monday of the month.

Thus, the above nine appointment dates can be described as follows in abstract syntax:

```
Every Day
Every Fri
First (1, Day)
First (10, Day)
First (1, Tue)
First (2, Tue)
Last (1, Day)
Last (1, Fri)
Last (2, Day)
```

Question 2.1

Write an informal grammar for appointment dates. It must recognize all of the above nine example appointment dates.

You may assume that there are terminal symbols (tokens) `FIRST` (representing 'first'), `LAST` (representing 'last'), `NTH` (representing 'second', 'third', and so on), `DAY` (representing 'day'), `MONDAY` (representing 'Monday'), ..., `SUNDAY`, `EVERY`, `OF`, `THE`, and `MONTH`.

Meaningless phrases such as `first last day of the month` should not be derivable from the grammar.

Hint: The question can be answered using only four nonterminal symbols.

Question 2.2

Write the grammar part of a `mosmlyac` parser specification for recurring appointment dates above. You may assume that the following token declarations are available in the parser specification already:

```
%token FIRST LAST
%token DAY MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY SATURDAY SUNDAY
%token EVERY OF THE MONTH
%token <int> NTH
%token EOF
```

That is, there are individual tokens for the keywords ‘first’ and ‘last’, and for ‘day’, ‘Monday’, ‘Tuesday’, ..., ‘Sunday’, as well as ‘every’, ‘of’, ‘the’, and ‘month’.

Moreover, all the ordinal numbers are represented by the token class `NTH`, so that ‘second’ is represented by `NTH(2)`, and ‘third’ is represented by `NTH(3)`, and so on.

It suffices to write the rule part of the `.grm` file, which begins like this:

```
Main:
    Date EOF                { ... }
;

Date:
    ...                    { ... }
| ...                    { ... }
...
...
```

Question 2.3

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

Question 2.4

The ordinal numbers `first`, `second`, `third`, `fourth`, `fifth` and so on are often written `1st`, `2nd`, `3rd`, `4th`, `5th` and so on. (However, there is no such shorthand for `last`, as far as I know). In an implementation of appointment dates it would be good to accept also these abbreviations, but this question concerns only the form of the abbreviations.

- We need to consider only the ordinal numbers between 1 and 99.
- Ordinal numbers ending in 1 must have suffix `st`, with the exception of `11th`.
- Ordinal numbers ending in 2 must have suffix `nd`, with the exception of `12th`.
- Ordinal numbers ending in 3 must have suffix `rd`, with the exception of `13th`.
- Ordinal numbers ending in 0, 4, 5, 6, 7, 8 or 9 have suffix `th`.

Write a regular expression that represents the well-formed ordinal numbers between 1 and 99. That is, it should accept `3rd` and `13th` and `23rd`, but not `13rd` or `23th`.

You may use abbreviated regular expressions such as `[4 – 9]` instead of the more verbose `(4|5|6|7|8|9)`.

Question 3 (40 %)

This question concerns a simple stack machine for manipulating lists, such as lists of integers. The stack machine has a program p , a program counter pc , a stack s holding integers and lists, 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
CST i	$s \Rightarrow$	$i : s$	Push integer constant i
ADD	$i_2 : i_1 : s \Rightarrow$	$(i_1 + i_2) : s$	Add integers
SUB	$i_2 : i_1 : s \Rightarrow$	$(i_1 - i_2) : s$	Subtract integers
DUP	$v : s \Rightarrow$	$v : v : s$	Duplicate
SWAP	$v_2 : v_1 : s \Rightarrow$	$v_1 : v_2 : s$	Swap
POP	$v : s \Rightarrow$	s	Pop
GOTO a	$s \Rightarrow$	s	Jump to a
IFNZRO a	$v : s \Rightarrow$	s	Jump to a if $v \neq 0$
CALL a	$v : s \Rightarrow$	$v : r : s$	Call function at a , pushing return address r
RET	$v : r : s \Rightarrow$	$v : s$	Return: jump to r
PRINT	$v : s \Rightarrow$	$v : s$	Print v and keep it on stack
STOP	$s \Rightarrow$	—	Halt the machine
MKNIL	$s \Rightarrow$	$\text{Nil} : s$	Push Nil, that is, the empty list
MKCONS	$t : i : s \Rightarrow$	$\text{Cons}(i, t) : s$	Push cons node $\text{Cons}(i, t)$
LISTCASE a	$\text{Nil} : s \Rightarrow$	s	If Nil, do not jump
LISTCASE a	$\text{Cons}(i, t) : s \Rightarrow$	$t : i : s$	If Cons, unpack components and jump to a

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 contains Integer objects as well as List objects, as defined below. The initial stack $s[0..k-1]$ contains the k command line arguments (which must be integers).

The stack machine uses a standard Java representation of lists of integers:

```

abstract class List { }

class Nil extends List {
    public String toString() {
        return "Nil";
    }
}

class Cons extends List {
    public final Integer i;
    public final List tail;

    public Cons(Integer i, List tail) {
        this.i = i; this.tail = tail;
    }

    public String toString() {
        return "Cons(" + i + ", " + tail + ")";
    }
}
    
```

```

/* Abstract stack machine for manipulating lists */
/* p[] is program, pc is program counter      */
/* s[] is stack,  sp is stack top pointer     */

static int execcode(int[] p, Object[] s, int pc, int sp) {
  for (;;) {
    switch (p[pc++]) {
      case CST:
        s[sp+1] = new Integer(p[pc++]); sp++; break;
      case ADD:
        s[sp-1] = new Integer(((Integer)s[sp-1]).intValue()
                               + ((Integer)s[sp]).intValue());
        sp--; break;
      case SUB:
        s[sp-1] = new Integer(((Integer)s[sp-1]).intValue()
                               - ((Integer)s[sp]).intValue());
        sp--; break;
      case DUP:
        s[sp+1] = s[sp]; sp++; break;
      case SWAP:
        { Object tmp = s[sp]; s[sp] = s[sp-1]; s[sp-1] = tmp; } break;
      case POP:
        sp--; break;
      case GOTO:
        pc = p[pc]; break;
      case IFNZRO:
        pc = (((Integer)s[sp--]).intValue() != 0 ? p[pc] : pc+1); break;
      case CALL:
        s[sp+1] = s[sp]; s[sp] = new Integer(pc+1); sp++;
        pc = p[pc]; break;
      case RET:
        pc = ((Integer)s[sp-1]).intValue();
        s[sp-1] = s[sp]; sp--; break;
      case PRINT:
        System.out.println(s[sp] + " "); break;
      case MKNIL:
        s[sp+1] = new Nil(); sp++; break;
      case MKCONS:
        s[sp-1] = new Cons((Integer)s[sp-1], (List)s[sp]);
        sp--; break;
      case LISTCASE:
        if (s[sp] instanceof Cons) {
          Cons cons = (Cons)s[sp];
          s[sp] = cons.i; s[sp+1] = cons.tail;
          sp++;
          pc = p[pc];
        } else {
          sp--;
          pc++;
        }
        break;
      case STOP:
        return sp;
      default:
        throw new RuntimeException("Illegal instruction " + p[pc-1] + " at " + (pc-1));
    }
  }
}

```

Question 3.1

First, manually execute the code below on the stack machine. Show the stack contents after every instruction, and show what is printed on the console:

```
0: CST 100
2: CST 11
4: ADD
5: MKNIL
6: MKCONS
7: PRINT
8: STOP
```

Secondly, write stack machine code to create and print the list `Cons(111, Cons(222, Nil))`.

Question 3.2

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

```
datatype instr =
  CST of int | ADD | SUB
  | DUP | SWAP
  | GOTO of int | IFNZRO of int | CALL of int | RET
  | PRINT
  | MKNIL | MKCONS | LISTCASE of int
  | STOP
```

Write a Standard ML function `mklist : int -> instr list` that for a given integer $n \geq 0$ generates stack machine code (in the form of SML abstract syntax) to build and print a list with n cons nodes, of this form:

$$\text{Cons}(2n, \text{Cons}(2n-2, \dots, \text{Cons}(2, \text{Nil})\dots))$$

For instance, if $n = 3$ then `Cons(6, Cons(4, Cons(2, Nil)))` should be printed.

Question 3.3

The machine instructions `CALL a` and `RET` can be used to implement simple functions that take zero or more arguments (on the stack top) and return one result (on the stack top).

What does this stack machine program compute, given that the integer 42 is on the stack top when execution is started at instruction address 0?

```
0: CALL 4
2: PRINT
3: STOP
4: DUP
5: DUP
6: MKNIL
7: MKCONS
8: MKCONS
9: MKCONS
10: RET
```

Show the contents of the stack after each instruction, in the order in which the instructions are executed.

Question 3.4

Write a stack machine program that, given that $n \geq 0$ is on the stack top, builds and prints an n -element list of this form:

$$\text{Cons}(n, \text{Cons}(n-1, \dots, \text{Cons}(1, \text{Nil})\dots))$$

Hint: you will need to use functions, implemented using `CALL` and `RET`. You are welcome to use symbolic labels instead of absolute jump addresses in those instructions that take an address argument (`GOTO`, `IFNZRO`, `CALL`).

Question 3.5

The instruction `LISTCASE a` can be used to test whether the list on the stack top is `Nil` or `Cons(...)`. If the stack top element is `Nil`, execution simply continues with the next instruction. If the stack top element is `Cons(i, t)`, then the integer i and list tail t are unpacked on the stack, and execution continues at address a .

Thus the expression `case xs of Nil => e1 | Cons(i, t) => e2` can be translated to the instruction sequence `LISTCASE lab1, <e1>, GOTO lab2, lab1: <e2>, lab2:`.

Assume that the list `Cons(111, Cons(222, Nil))` is on the stack top when the function at address 21 below is called (by `CALL 21`). What is on the stack top, and is therefore printed, when control reaches instruction 28? What is printed if the empty list `Nil` is on the stack top when `CALL 21` is executed?

```
21: LISTCASE 27, CST 999, GOTO 28,
27: POP
28: PRINT
```

Question 3.6

Write a stack machine function that, given a list on the stack top, computes the sum of the list elements. Hint: you need to use `LISTCASE`, `CALL`, and `RET`.

Question 3.7

The stack machine as presented above is very limited and hard to use. One reason is that there is no way to access the stack elements below the topmost two elements: there is no way to index into the stack. Extend the stack machine with a new instruction `GETVAR i` that reads the contents of the i 'th stack element below the stack top and pushes that value onto the stack (incrementing the stack top pointer).

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

First show the necessary modifications to the Java implementation of the stack abstract machine on page 6.

Then use the new `GETVAR` instruction to write a stack machine function that can create a list that contains n copies of v , like this:

$$\text{Cons}(v, \text{Cons}(v, \dots, \text{Cons}(v, \text{Nil})\dots))$$

The function should be called with a stack of form $n : v : s$, that is, with n on the stack top and v below it, and should return with a stack of the form $w : s$ where w is an n -element list all of whose elements are v , as shown above.

Question 3.8

Write a stack machine program that, given that $n \geq 0$ is on the stack top, builds and prints an n -element list of this form:

$$\text{Cons}(2n, \text{Cons}(2n-2, \dots, \text{Cons}(2, \text{Nil})\dots))$$

Hint: This can be done even without the new `GETVAR i` instruction from Question 3.7.