Uppsala University
MN Functional Programming
Period 2, Autumn 2003
Exam 1
Friday 12 December 2003, from 14:00 to 19:00

Global Instructions

Read these instructions, as well as the actual questions, very carefully **before** attempting to solve the problems. Especially pay attention to **stressed** words (in boldface). The questions have been engineered to have many short and elegant answers. If you get into some lengthy or difficult reasoning, you are probably on the wrong track and might benefit from re-reading the question.

This question set is double-sided. To the extent possible, write your answers into the gaps. The provided space is really sufficient each time. Write your name onto **every** sheet. This is an exam with **closed** books and notes. An English-Swedish dictionary may be available at the front desk. Normally, the instructor will come and answer questions between 15:30 and 16:30.

To save time, program in a **non-defensive** style. Provide a specification (at least the names of the argument components, a signature, a pre-condition, a post-condition involving **all** the names of the argument components, and **useful** examples) for **every** function you construct. Each specification **must** be suitable for justifying your function or for constructing another function. Provide a justification outline (the chosen variant) for **every recursive** function you construct. You need **not** provide any other justifications, but the given ones **must** correspond to your function. For instance, each clause should **not** be redundant with previous clauses. Failure to provide such a specification or justification outline for at least one function of a sub-question will result in zero points for that entire sub-question, even if the function is actually correct. If you cannot comply with a requirement of a sub-question, such as the presence or absence of recursion, the indicated variant, or the number of new functions, then **explicitly** lift that requirement and proceed without it.

You may **only** use the directives and functions of the **standard** library of SML, as well as `Math.sqrt`. Do **not** use higher-order functions, **except** where explicitly requested. The instructor's solutions to the questions only involve `=`, `>`, `+`, `-`, `*`, `::`, `@`, `abstype`, `andalso`, `as`, `fn`, `foldl`, `foldr`, `fun`, `hd`, `if…then…else`, `infix`, `let…in…end`, `list`, `map`, `Math.sqrt`, `of`, `op`, `orelse`, `tl`, and `val`. Exact SML syntax is not required. Layout is unimportant, but please be considerate.

Unless otherwise posted, the instructor is **only** interested in correct SML functions. Any attempts at efficient functions are purely at your own risk, namely the risk of missing out on correctness or of losing time.

The 4 credit points for this exam are awarded if your exam points are in the interval [50%,100%]. Furthermore, a *very-good* (VG) pass grade is earned for the interval [85%,100%], while a *good* (G) pass grade is earned for the interval [50%,84%]. Otherwise, an "*underkänd*" (U) fail grade is earned.

For official use (do not write below this line):

Q1	Q2	Q3	Exam
/ 26	/ 16	/ 46	/ 88

Question 1 Methodology and Recursion (26 points)

A *segment* of a list is a prefix of a suffix of that list.

For example, the lists [], [4,5], and [2,1,4,5,3] are segments of [2,1,4,5,3].

A *plateau* of a list is a segment thereof with all-equal elements but different previous and next elements, if any.

For example, the list [3,3] is a plateau of [1,3,3,2,2,2,5], but its segments [3,3,2] and [3] are not plateaus thereof.

Using the concepts defined above, answer the following sub-questions:

(9 points) a. Construct a function compressing a list of binary digits (0 or 1) by encoding its plateaus of 1s as their lengths. Use **recursion**. Use the indicated **variant**. Use **no** new functions.

```
function compress L
signature:
pre:
post:

example: compress [0,0,1,1,1,1,0,1,0,0,1,1,1] = [0,0,4,0,1,0,0,3]
fun

variant: the number of elements of L
```

(17 points) b. Construct another function for the specification above of `compress`. Use **recursion**. Use the indicated **variant**. Use **at most one** new function (and declare it on the next page).

```
fun

variant: the number of plateaus of L
```

If you needed a new function, then give a **most general** specification and construct it here:

function
signature:
pre:
post:

example:

```
fun
```

variant:

Background for Questions 2 and 3

A table with m rows and n columns is called an $m \times n$ table. If A is a table, then a_{ij} denotes the element in its i^{th} row and j^{th} column. The *pairwise product* of two rows/columns $(x_1,x_2,\ldots,x_p)$ and $(y_1,y_2,\ldots,y_p)$ is the number $x_1y_1 + x_2y_2 + \ldots + x_py_p$. The *product* of an $m \times n$ table A by an $n \times p$ table B, denoted $A \cdot B$, is an $m \times p$ table C whose element c_{ij} is the pairwise product of the i^{th} row of A and the j^{th} column of B. The *flip* of an $m \times n$ table A, denoted A', is the $n \times m$ table obtained by converting the rows of A into columns. The *norm* of an $m \times n$ table A, denoted $|A|$, is the square-root of the sum of the squares of its elements. For example,

$$\begin{bmatrix} 2 & 3 \\ -1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & -2 & 1 \\ 3 & 8 & -6 \end{bmatrix} = \begin{bmatrix} 19 & 20 & -16 \\ 7 & 34 & -25 \end{bmatrix}, \text{ and } \begin{bmatrix} 5 & -2 & 1 \\ 3 & 8 & -6 \end{bmatrix}' = \begin{bmatrix} 5 & 3 \\ -2 & 8 \\ 1 & -6 \end{bmatrix}, \text{ and } \left| \begin{bmatrix} 2 & 3 \\ -1 & 4 \end{bmatrix} \right| = \sqrt{2^2 + 3^2 + (-1)^2 + 4^2} = \sqrt{30}.$$

Question 2 Specification of an ADT (16 points)

Using the concepts and notations in the background above, specify the following functions for an SML abstract datatype (ADT) — called `'a tab` — for polymorphic tables:

(4 points) c. A function `tab2list`, which converts a table into a list by appending its rows from top to bottom.

function `tab2list A`
signature:
pre:
post:
example:

(6 points) d. A curried function `list2tab`, which, given a pair `(m,n)` of positive integers and a list `L` of `mn` elements, converts `L` into the `m×n` table `A` such that `tab2list A = L`.

```
function list2tab (m,n) L
signature:
pre:
post:
example:
```

(2 points) e. A function `norm`, which returns the norm of a real-number table.

```
function norm A
signature:
pre:
post:
```

(4 points) f. An infix function `times`, which returns the product of two integer tables, assuming it is defined.

```
function A times B
signature:
pre:
post:
```

Question 3 A Realisation of the ADT (46 points)

Realise the `'a tab` ADT, using a representation that is based on lists of lists: an $m \times n$ table A, with $m > 0$ and $n > 0$, is to be represented by `TAB` `[[`a_{11}`,…,`a_{1n}`],...,[`a_{m1}`,…,`a_{mn}`]]`. The **representation invariant** is that there are $m > 0$ element lists, all of the same length $n > 0$. Answer the following sub-questions:

(1 point) g. Declare the realisation of the `'a tab` ADT.

```
abstype 'a tab =
with (* here comes the code of the other sub-questions *) end
```

(3 points) h. Realise `tab2list`. Use **no** recursion. Use one or more of the **standard** higher-order functions `map`, `foldr`, and `foldl`. Use **no** new functions.

```
fun
```

(12 points) i. Realise `list2tab`. Use **recursion**. Use the indicated **variant**. Use **at most one** new function.

```
fun
```

```
variant: m
```

If you needed a new function, then give a **most general** specification and construct it here:

function

signature:

pre:

post:

example:

fun

variant:

(6 points) j. Realise `norm`. Use **no** recursion. Use one or more of the **standard** higher-order functions `map`, `foldr`, and `foldl`. Use **no** new functions. You **may** use other functions from the ADT.

fun

(17 points) k. Specify and implement an **infix** function `pairwise` for the pairwise product of two integer rows/ columns of the same length, represented as integer lists. Use **recursion**. Use **no** new functions.

function `X pairwise Y`

signature:

pre:

post:

example:

fun

variant:

Is this function tail-recursive or not? Why / Why not?

If not, then specify a **descending** generalisation (which introduces an accumulator), called `pairwise''`, of `pairwise` and construct it using **tail-recursion**:

function

signature:

pre:

post:

example:

fun

variant:

Non-recursively re-realise the `pairwise` function, calling it `pairwise'` now though it has the same specification as `pairwise`, using only `pairwise''`:

val pairwise' =

(7 points) l. To realise `times`, let us declare a new, similar **infix** function `mult` that takes the flip of the second table so as to get more convenient access to its columns.

example: (TAB [[2],[~1]]) times (TAB [[3,4]]) = (TAB [[6,8],[~3,~4]])

infix times

fun A times B = A mult (flip B)

assuming we also have a function `flip` that returns the flip of a table. Give a **most general** specification of `mult` and implement it here. Use **recursion**. Use the indicated **variant**. Use the `pairwise'` function. Use the **standard** higher-order function `map` to avoid declaring another new function.

function A mult B

signature:

pre:

post:

example: (TAB [[2],[~1]]) mult (TAB [[3],[4]])
= (TAB [[6,8],[~3,~4]])

fun

variant: the number of rows of A