Uppsala University
MN Functional Programming
Period 2, Autumn 2003
Exam 2
Tuesday 13 January 2004, from 08:00 to 13:00

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 10:00 and 11:00.

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. Do **not** use higher-order functions, **except** where explicitly requested. The instructor's solutions to the questions only involve `=`, `<`, `>`, `+`, `-`, `*`, `::`, `abstype…with…end`, `as`, `fn`, `fun`, `hd`, `if…then…else`, `int`, `infix`, `let…in…end`, `list`, `nil`, `of`, `rec`, 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
/ 18	/ 14	/ 48	/ 80

Question 1 Methodology and Recursion (18 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 [2,2] is a plateau of [4,2,2,3,3,3,1], but its segments [2,2,3] and [2] are not plateaus thereof.

The *compression* of a list L is a list of (x_i,c_i) pairs, such that the i^{th} plateau of L has c_i elements equal to x_i.

For example, the compression of [4,2,2,2,2,3,3,3,4,4] is [(4,1),(2,4),(3,3),(4,2)].

Using the concept names above, answer the following sub-questions.

(2 points) a. Specify a function `compress` returning the compression of an arbitrary list.

```
function compress L
sig:
pre:
post:
ex: compress [4,2,2,2,2,3,3,3,4,4] = [(4,1),(2,4),(3,3),(4,2)]
```

(5 points) b. Construct a first SML function for `compress`. Use **recursion**. Use the indicated **variant**. Use **no** new functions.

```
fun
```

```
variant: the number of elements of L
```

(11 points) c. Construct another SML function for `compress`. Use **recursion**. Use the indicated **variant**. Use **at most one** new function (the space for it is provided 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

sig:

pre:

post:

ex:

fun

variant:

Question 2 Specification of a `nat` ADT (14 points)

A *prime number* is a positive integer having exactly one positive divisor other than 1.

Given a positive integer n such that $n > 1$, its *prime factorisation* is n rewritten as a product of prime numbers.

For example, $2 = 2$, $3 = 3$, $4 = 2 \cdot 2 = 2^2$, $5 = 5$, $6 = 2 \cdot 3$, $7 = 7$, $8 = 2 \cdot 2 \cdot 2 = 2^3$, $9 = 3 \cdot 3 = 3^2$, $10 = 2 \cdot 5$, $11 = 11$, $12 = 2 \cdot 2 \cdot 3 = 2^2 \cdot 3$, etc.

A positive integer n such that $n > 1$ can thus be rewritten as a product $p_1^{a_1} \cdot \cdots \cdot p_q^{a_q}$ where the p_i are prime numbers — called the *prime factors* of n — and the powers a_i are positive integers.

Additionally, we arbitrarily define the prime factorisation of 0 to be 0^1, and the one of 1 to be 1^1, although 0 and 1 are not prime numbers, so that **every** natural number (non-negative integer) has a prime factorisation. The prime factorisation of any natural number is **unique**.

Without reading Question 3, specify the following functions for an SML abstract datatype (ADT) — called `nat` — for natural numbers (that is, non-negative integers):

(2 points) d. The function `intToNat` converts a non-negative integer into a natural number.

function `intToNat i`

sig:

pre:

post:

(2 points) e. The function `natToInt` converts a natural number into an integer.

function `natToInt n`

sig:

pre:

post:

(3 points) f. The function `primeFactors` returns the non-decreasing list of integer prime factors of a natural number.

```
function primeFactors n
sig:
pre:
post:
ex: primeFactors (intToNat 12) = [2,2,3]
```

(3 points) g. The infix function `plus`, which returns the sum of two natural numbers.

```
function a plus b
sig:
pre:
post:
ex:
```

(4 points) h. The infix function `times` returns the product of two natural numbers that are larger than 1.

```
function a times b
sig:
pre:
post:
ex:
```

Question 3 Realisation of the `nat` ADT (48 points)

Realise the `nat` ADT, using a representation that is based on the prime factorisation. The natural number with the prime factorisation ${p_1}^{a_1} \cdot \cdots \cdot {p_q}^{a_q}$ is to be represented by `PF` $[(p_1, a_1), \ldots, (p_q, a_q)]$. The **representation invariant** is that the prime factors are strictly increasing from left to right across the list, which must be non-empty, and that all the powers are positive. Answer the following sub-questions.

(2 points) i. Declare the realisation of the `nat` ADT.

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

(11 points) j. Realise the `intToNat` function. Assume there are SML functions for the following two specifications:

```
function candidates i
sig: int → int list
pre: i > 1
post: the candidate prime factors of i, in increasing order
ex: candidates 10 = [2,3,5,7] ;  candidates 11 = [2,3,5,7,11]
function divGen i j
sig: int → int → int * int
pre: j > 1
post: (q,d) where q is maximal such that j^q divides i, and d = i / j^q
ex: divGen 40 2 = (3,5) ;  divGen 9 2 = (0,9)
```

Indeed, $40 = 2^3 \cdot 5$ and $9 = 2^0 \cdot 9$. Note that the first component of `divGen i j` is `0` if and only if `i` is not divisible by `j`. Use **at most one** new function. Use **no** recursion for `intToNat`. Use the **idea** of reducing the given integer by successively trying all its candidate prime factors, in increasing order.

ex: `intToNat 180 = PF [(2,2),(3,2),(5,1)]`

fun `intToNat`

If you needed a new function, then specify it and construct it here:

function

sig:

pre:

post:

ex:

fun

variant:

(15 points) k. Realise the `natToInt` function. Use **recursion**. Use **no** new functions, not even exponentiation.

ex: `natToInt (PF [(2,2),(3,2),(5,1)]) = 180`

fun `natToInt`

variant:

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

If not, then specify a descending generalisation (which introduces an accumulator) of `natToInt` and construct a **tail-recursive** SML function for it here:

```
function
sig:
pre:
post:
ex:
fun
```

```
variant:
```

Non-recursively re-realise the `natToInt` function. Use **only** the generalisation that you have specified.

```
fun
```

(7 points) l. Realise the `primeFactors` function. Use **recursion**. Use **no** new functions.

```
val rec primeFactors =
```

```
variant:
```

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

If not, can we apply the descending generalisation (accumulator-introduction) technique to specify a generalisation that can be implemented using **tail-recursion**? Why / Why not?

(1 point) m. Realise the `plus` function. Use **no** recursion. Use **no** new functions other than from the ADT.

```
ex: PF [(2,2),(3,1)] plus PF [(3,1),(5,1)] = PF [(3,3)]
infix plus
fun
```

(12 points) n. Realise the `times` function. Use **no** recursion for `times`. Examine and use **only** the new function `mmm` below, which works on prime-factor lists without the `PF` value constructor.

```
ex: PF [(2,2),(3,1)] times PF [(3,1),(5,1)] = PF [(2,2),(3,2),(5,1)]
infix times
fun
```

```
function mmm a b
sig:
pre:
post:
```

```
ex: mmm [(2,2),(3,1)] [(3,1),(5,1)] = [(2,2),(3,2),(5,1)]
fun mmm
```

```
variant:
```

You may draw pictures or take scratch notes from here on!