Uppsala University
ITP1 Programkonstruktion, del 2
Period 3, Spring 2002
Exam 1
Monday 11 March 2002, from 9:00 to 14:00

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 concise and elegant answers, so if you get into some messy reasoning, you are probably on the wrong track and would 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 amply sufficient each time. Write your name onto **every** sheet. This is an exam with **closed** books and notes. An English-Swedish dictionary is available at the front desk. Normally, the instructor will come and answer questions between 11:00 and 12:00.

Provide a specification (with 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** program you construct, **such that** this specification would be suitable for justifying your program or constructing another program. Provide a justification outline (the chosen induction parameter and well-founded relation) for **every** recursive program you construct. You often need **not** provide any other justifications, but the given ones **must** correspond to your program: for instance, each case should **not** be redundant with the other cases. 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 program is actually correct.

You may **only** use the functions and directives of the **standard** library of SML. For instance, the instructor's solutions to the questions below only involve `=`, `<`, `>`, `+`, `-`, `*`, `::`, `@`, `if…then…else…`, `foldl`, and `infix`. Do **not** use higher-order functions, except where explicitly requested. Layout is unimportant, but please be considerate.

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

The 2.2 credit points for this exam are awarded for VT02 if the sum of your exam points and bonus points is in the interval [55,100]. Furthermore, a "med beröm godkänd" (5) grade is earned if this sum is in [85,100], while an "icke utan beröm godkänd" (4) grade is earned if this sum is in [70,84], and a "godkänd" (3) grade is earned if this sum is in [55,69]. In all other cases, an "underkänd" (U) grade is earned.

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

Q1	Q2	Q3	Exam
/ 14	/ 55	/ 11	/ 80

Cover Story

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 instance, $2 = 2$, $3 = 3$, $4 = 2 * 2 = 2^2$, $5 = 5$, $6 = 2 * 3$, $7 = 7$, $8 = 2 * 2 * 2 = 2^3$, $9 = 3 * 3 = 3^2$, $10 = 2 * 5$, $11 = 11$, $12 = 2 * 2 * 3 = 2^2 * 3$, etc.

A positive integer n such that $n > 1$ can thus be rewritten as a product $p_1{}^{a_1} * \cdots * p_q{}^{a_q}$

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 has a prime factorisation.

The prime factorisation of any natural number is unique.

Question 1 Specification of an ADT (14 points)

Specify an SML abstract datatype (ADT) — called `nat` — for natural numbers, with the following functions:

a. An infix function `plus`, which returns the sum of two natural numbers:

   function

   pre:

   post:

   Example:

b. An infix function `times`, which returns the product of two natural numbers that are larger than 1:

   function

   pre:

   post:

   Example:

c. A function `natToInt`, which converts a natural number into an integer:

   function

   pre:

   post:

d. A function `intToNat`, which converts an integer into a natural number:

   function

   pre:

   post:

e. A function `primeFactors`, which returns the non-decreasing list of prime factors of a natural number:

   function

   pre:

   post:

   Example: `primeFactors (intToNat 12) = [2,2,3]`

Question 2 A First Realisation of the ADT (55 points)

Realise the `nat` ADT, using a representation that is based on prime factorisation. The natural number with prime factorisation $p_1^{a_1} * \cdots * p_q^{aq}$ is to be represented by `PF` $[(p_1, a_1), \ldots, (p_q, a_q)]$. You must protect the **representation invariant** 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:

a. Declare the realisation of the `nat` ADT:

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

b. Realise the `times` function, using **at most one** other function.

Hint: Introduce a help function that works on the 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)]`

```
fun
```

If you introduced a (recursive) help function, then specify and implement it here:

function

pre:

post:

Example:

```
fun
```

by induction on:

well-founded relation:

In what sense is the given example of `times` **not** really an example?

c. Realise the `natToInt` function, using **recursion** and introducing **no** new functions, say exponentiation:

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

fun

by induction on:

well-founded relation:

Is this function tail-recursive or not? Why?

If not, then specify a generalisation of `natToInt` and implement it using **tail-recursion**:

function

pre:

post:

Example:

fun

by induction on:

well-founded relation:

as well as re-realise the `natToInt` function using only the generalisation you have specified:

fun

d. Assuming that you are **given** help functions for the following two specifications:

function `candidates i : int → int list`

pre: `i ≥ 2`

post: `the candidate prime factors of i, in increasing order`

Examples: `candidates 10 = [2,3,5,7] ; candidates 11 = [2,3,5,7,11]`

and:

function `divGen i j : int → int → int * int`

pre: `j > 0`

post: `(the largest number q such that i` **mod** j^q `= 0, i` **div** j^q`)`

Examples: `divGen 40 2 = (3,5) ; divGen 9 2 = (0,9)`

realise the `intToNat` function, using the **idea** of reducing the given integer by successively trying all its candidate prime factors, and introducing **at most one** new function:

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

fun

If you introduced a (recursive) help function, then specify and implement it here:

function

pre:

post:

Example:

fun

by induction on:

well-founded relation:

e. Realise the `plus` function, using **no** recursion and using **no** new functions outside the ADT:

Example: `PF [(2,2),(3,1)] plus PF [(3,1),(5,1)] = PF [(3,3)]`

fun

f. Realise the `primeFactors` function, using **recursion** and introducing **no** new functions:

val rec

by induction on:

well-founded relation:

Is this function tail-recursive or not? Why?

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

Question 3 Another Realisation of the ADT (11 points)

Realise the `nat` ADT again, using a representation that is based on integers. The natural number *n* is to be represented by the expression `NAT` *n*. You have to protect the **representation invariant** that the involved integer is non-negative. Answer the following sub-questions:

a. Declare the realisation of the `nat` ADT:

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

b. Realise the `plus`, `natToInt`, and `intToNat` functions, using **no** recursion and introducing **no** new functions:

```
fun
fun
fun
```

c. Assuming you already have a realisation of the `primeFactors` function, use it to realise the `times` function, using **no** recursion but one or more of the **standard** higher-order functions `map`, `foldr`, and `foldl` as well as introducing **no** new functions:

```
fun
```

You may draw pictures or take scratch notes below this line!