Uppsala University
DVP1 Programmeringsmetodik
Periods 1 & 2 of Fall 2001
Exam 1
Monday 17 December 2001, 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 to answer questions between 11:00 and 12:00.

All components of function arguments should be named and used in **at least** the post-condition. Failure to provide a specification (signature, pre-condition, and post-condition) **or** — if the program is recursive — a justification skeleton (the chosen induction parameter and well-founded relation) for **at least** one function of a sub-question will result in a zero grade for that entire sub-question, **even** if the program is actually correct. You need **not** hand in any other justifications, but the provided ones **must** correspond to your program.

You may **only** use the functions of the **standard** library of SML. For instance, my solutions to the questions below only involve +, –, *map*, and *foldl*. Layout is unimportant, but please be considerate.

Unless otherwise posted, I am **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 four credit points for this exam are awarded for HT01 if the sum of your exam points and bonus points is in the interval 55..100. Furthermore, a very-good grade is earned if this sum is in the interval 75..100, while a good grade is earned if this sum is in the interval 55..74.

Cover Story

Towards axiomatising natural-number arithmetic the way Euclid axiomatised geometry around 300 BCE, the Italian mathematician Giuseppe Peano proposed, in 1889, the following alternative representation of natural numbers: the number 0 is represented by the constant `Zero`, the number 1 is represented by `S(Zero)`, where `S` can be seen as a successor function, the number 2 is represented by `S(S(Zero))`, and so on. In other words, the positive integer *n* is represented by `S(`*p*`)`, where *p* is the Peano representation of *n*–1.

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

Q1	Q2	Q3	Q4	Q5	Exam
/ 8	/ 30	/ 10	/ 25	/ 7	/ 80

Question 1 Abstract Datatypes (8 points)

Design the skeleton of an SML abstract datatype (ADT) for natural numbers, using Peano's representation and exporting, at this moment, only the following types and constructors:

a. A datatype `nat`.

b. A value `zero`, which denotes the natural number 0.

```
    value zero :
    description:
    val zero
```

c. A function `next`, which increases a natural number by 1. Example: `next Zero = S(Zero)`

```
    function next n :        →
    pre:
    post:
    fun next
```

d. In what sense is the example in sub-question c **not** really an example?

Question 2 Recursion (30 points)

Add the following functionalities to the ADT, by programming **recursively**, in a **non-defensive** style, and **without** calling any new auxiliary functions, **except** maybe the other functions of **this** question:

e. The predicate `lessEq` returns `true` if and only if its 'first' natural number is less than or equal to its 'second' natural number. Example: `lessEq S(Zero) S(S(Zero)) = true`

```
    function lessEq n1 n2 :              →
    pre:
    post:
    val rec lessEq =




    by induction on:
    well-founded relation:
```

f. The function `plus` returns the sum of two natural numbers.

Example: `S(Zero) plus S(S(Zero)) = S(S(S(Zero)))`

```
function n1 plus n2 :            →
pre:
post:
fun       plus




by induction on:
well-founded relation:
```

g. The function `minus` returns the difference of two natural numbers, **assuming** the result is a natural number. Example: `S(S(S(S(S(Zero))))) minus S(S(Zero)) = S(S(S(Zero)))`

```
function n1 minus n2 :            →
pre:
post:
fun       minus




by induction on:
well-founded relation:
```

h. The function `times` returns the product of two natural numbers.

Example: `S(S(S(Zero))) times S(S(Zero)) = S(S(S(S(S(S(Zero))))))`

```
function n1 times n2 :            →
pre:
post:
fun       times




by induction on:
well-founded relation:
```

i. The function `divMod` returns the quotient and the remainder of the division of two natural numbers, **assuming** the result is mathematically defined.

Example: `S(S(S(S(S(Zero))))) divMod S(S(Zero)) = ( S(S(Zero)) , S(Zero) )`

```
function n1 divMod n2 :              →
pre:
post:
fun        divMod




by induction on:
well-founded relation:
```

Question 3 Type conversion (10 points)

Add the following functionalities to the ADT, by programming **recursively**, in a **non-defensive** style:

j. The function `natToInt` converts a natural number into an integer.

Example: `natToInt S(S(S(S(S(Zero))))) = 5`

```
function natToInt n :        →
pre:
post:
fun natToInt


by induction on:
well-founded relation:
```

k. The function `intToNat` converts a non-negative integer into a natural number.

Example: `intToNat 5 = S(S(S(S(S(Zero)))))`

```
function intToNat i :        →
pre:
post:
fun intToNat


by induction on:
well-founded relation:
```

Question 4 Tail Recursion (25 points)

Answer the following sub-questions:

l. Which of your programs in Question 2 are tail-recursive and which are not? Why?

`lessEq:`

`plus:`

`minus:`

`times:`

`divMod:`

m. Using your programs in Question 2, reduce the expression `S(S(Zero)) times S(S(S(Zero)))`, temporarily considering `plus` to be a primitive. Discuss this reduction.

n. Construct a **tail-recursive** generalisation of `times`, called `times''`, after justifying its existence.

```
function times''        :                    →
pre:
post:
fun times''


by induction on:
well-founded relation:
```

o. Construct a new, **non-recursive** function for `times`, called `times′`, in terms of `times′′`.

```
fun        times′
```

p. Reduce the expression `S(S(Zero)) times′ S(S(S(Zero)))`, temporarily considering `plus` to be a primitive. Discuss this reduction.

Question 5 Higher-Order Functions (7 points)

Add the following functionality to the ADT, by programming **non-recursively**, in a **non-defensive** style, and **by** using one or more of the **standard** higher-order functions `map`, `foldr`, and `foldl`:

q. The function `sigmaSqr` returns the sum of the squares of the elements of a list of natural numbers.

Example: `sigmaSqr [S(Zero), S(S(Zero))] = S(S(S(S(S(Zero)))))`

```
function sigmaSqr N :             →
pre:
post:
fun sigmaSqr
```