Uppsala University
DVP1 Programmeringsmetodik
Periods 1 & 2 of Fall 2001
Exam 2
Wednesday 16 January 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. Unfortunately, the instructor cannot come and answer questions, due to a clash with his teaching schedule. However, Dr Mikael Pettersson will be there at 10:00 and 12:00 (noon) for an exam he is giving at the same time and the same place, so he will try and answer any questions you may have.

Provide a specification (with at least the names of the argument components, a signature, a pre-condition, and a post-condition involving **all** the names of the argument components) for **every** program you construct, **such that** this specification would be suitable for justifying your program or constructing another program. Provide a justification skeleton (the chosen induction parameter and well-founded relation) for **every** recursive program you construct. You 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 skeleton 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 of the **standard** library of SML. For instance, the instructor's solutions to the questions below only involve `=`, `<>`, `<`, `>`, `::`, `+`, `@`, `andalso`, `if…then…else…`, and `orelse`. 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 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.

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

Q1	Q2	Q3	Exam
/ 20	/ 25	/ 35	/ 80

Question 1 Type Inference, Reduction, Currying (20 pts)

Considering the following SML declarations:

```
fun mystery x = ((fn y => y-1) x) * ((fn y => y+1) x) + 1
fun triple g z = (1+1) * g z
fun min (x,y) = if x>y then x else y
fun max (v,w) = min (min v, min w)
```

answer the following sub-questions:

a. Complete the following specifications, simplifying the post-conditions as much as possible:

```
function mystery n :        →
pre: (none)
post:
function triple f x :        →
pre: f is defined on x
post:
```

b. Give the SML **value** declaration that is equivalent to the SML **function** declaration of `triple` above:

```
val triple =
```

Step-by-step show what happens with the SML declaration `val strange = triple mystery`:

Step-by-step normalise the SML expression `strange 4`:

c. Give the signatures of `min` and `max`:

```
function min (a,b) :
function max (a,b) :
```

Give **curried** signatures of `min` and `max`, calling them `minC` and `maxC`, **assuming** `maxC` uses `minC`:

```
function minC
function maxC
```

Question 2 Methodology and Recursion (25 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 definitions above, answer the following sub-questions, by programming in a **non-defensive** style:

a. Construct a **recursive** program compressing a list of binary digits (0 or 1) by encoding plateaus of 1s as their sums, **without** introducing any help functions:

```
function compress L :          →
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 compress
```

```
by simple induction on: the number of elements of L
well-founded relation: <
```

b. Construct another **recursive** program for `compress`, taking a different methodological decision:

```
fun compress
```

```
by simple induction on: the number of plateaus of L
well-founded relation: <
```

If you need a **recursive** help function (you are allowed **at most one**!), then construct it here:

```
function          :          →
pre:

post:

Examples:

fun





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

Question 3 Recursion versus Tail-Recursion (35 points)

A *binary tree* is either the empty binary tree, or a non-empty binary tree with a root element as well as left and right binary trees as subtrees. The *prefix traversal* of a binary tree *B* is a list where the root of *B* appears just before the prefix traversal of its left subtree followed by the prefix traversal of its right subtree.

Using the definitions above, answer the following sub-questions, by programming in a **non-defensive** style:

a. Declare a new SML datatype for binary trees of elements of **any** type, for use in the other sub-questions:

```
datatype
```

b. Construct a **recursive** program computing the prefix traversal of a binary tree, **without** any help functions:

```
function prefix BT :          →
pre:
post:
fun prefix



by simple induction on: the number of elements of BT
well-founded relation: <
```

Is this program tail-recursive or not? Why?

c. Construct a **recursive** program for the generalisation of `prefix` that computes the concatenation of the prefix traversals of the elements of a **list** of binary trees, **without** any help functions and with **at most one** recursive call per clause:

```
function prefix′        :                    →
pre:
post:
fun prefix′




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

Is this program tail-recursive or not? Why?

Construct a **non-recursive** program for `prefix`, using **only** `prefix′`:

```
fun prefix
```

Is this program more or less efficient than the other one for `prefix`? Why?

d. Construct a **recursive higher-order** function for binary trees, performing what `foldr` and `foldl` do for lists, namely (in this case) applying a 3-ary function *f* to the root of the tree and the two results of applying *f* to its two subtrees, given a start element *e*:

```
function fold f e BT :                       →
pre:
post:
fun fold




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

Construct a **non-recursive** program computing the maximum element of a binary tree of **natural numbers**, using `fold`:

```
function max BT :            →
pre:
post:
val max =
```

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