Uppsala University
ITP1 Programkonstruktion, del 1
Periods 1 & 2 of Fall 2001
Exam 1
Friday 7 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.

Write your answers onto these sheets, **including** their backsides: the provided space is sufficient. Write your Swedish person number onto **every** sheet. Do **not** write your name onto any 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.

Comment **every** SML function with its specification, indicating the signature, pre-condition, and post-condition of the function, and (when useful) listing some examples and counter-examples of its usage. **None** of the functions in this exam should have side-effects or raise exceptions, so you need never mention that there are none. **All** components of the argument of a function should be named and used in **at least** the post-condition. Failure to comply with this directive for at least one function for a sub-question will result in a zero grade for that sub-question, even if the function is actually correct.

Comment **every** recursive SML function with the induction parameter and the well-founded relation you used during the analysis phase of programming. Failure to comply with this directive for at least one recursive function for a sub-question will result in a zero grade for that sub-question, even if the function is actually correct. You need **not** hand in any other information about any analysis phase.

You may **only** use the functions of the standard library of SML. For instance, my solutions to the questions only involve =, <, <=, +, –, as well as *if-then-else*. 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.

The two 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 “med beröm godkänd” (5) grade is earned if this sum is in the interval 85..100, while an “icke utan beröm godkänd” (4) grade is earned if this sum is in the interval 70..84, and a “godkänd” (3) grade is earned if this sum is in the interval 55..69. In all other cases, an “underkänd” (U) grade is earned.

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

Q1	Q2	Q3	Q4	Q5	Q6	Bonus	Total
/ 10	/ 15	/ 20	/ 15	/ 10	/ 10	/ 20	/ 100

Question 1 Reduction and Currying (10 points)

Consider the SML function declaration

```
fun fromTo i j = if i>j then [] else i::fromTo (i+1) j
```

for a function that returns the list of integers from `i` to `j`, both inclusive.

Answer the following sub-questions:

(a) What is the signature of `fromTo`?

(b) Which SML **value** declaration is equivalent to the SML function declaration above?

(c) Considering (b), show the reductions **and** bindings effected by the following 2 SML value declarations:

```
val f = fromTo 1
```

```
val xs = f 2
```

Question 2 Reverse Engineering (15 points)

Infer specifications, **if any**, for the following five SML declarations of pre-condition-**free** functions:

(a)

```
fun f (x,[]) = []
  | f (x,y::ys) = (x+y)::f(x-1,ys)
```

(b)

```
fun g [] = []
  | g ((x,y)::s) = (x,y)::(y,x)::(g s)
```

(c)

```
fun h [] = []
  | h (x::xs) = x::(h xs)@[x]
```

(d)

```
fun k e [] = []
  | k e (x::xs) = if e=x then k e xs else x::k e xs
```

(e)

```
fun m x:int  = x div 2
  | m x:real = x  /  2
```

Question 3 The Usefulness of Pre-Conditions (20 points)

An integer list $[x_1, x_2, ..., x_n]$ is *non-decreasing* if its elements satisfy the inequalities

$$x_1 \le x_2 \le \ldots \le x_{n-1} \le x_n$$

Using **recursion**, **non**-defensively construct SML functions for **four** of the following five requirements.

You **must** exploit the fact that a given list is expected to be non-decreasing.

(a) Function *count* returns the number of occurrences of an integer in a non-decreasing integer list.

Example: *count* (3,[0,1,1,1,3,3,7]) = 2.

(b) Function *insert* returns the non-decreasing integer list obtained by inserting an integer into such a list.

Example: *insert* (3,[0,1,1,1,3,3,7]) = [0,1,1,1,3,3,3,7].

(c) Function *intersect* returns the non-decreasing integer list that is the intersection of two such lists.

Example: *intersect* [0,1,1,1,3,3,7] [1,1,7,7,9] = [1,1,7].

(d) Function *union* returns the non-decreasing integer list that is the union of two such lists.

Example: *union* [0,1,1,1,3,3,7] [1,1,7,7,9] = [0,1,1,1,1,1,3,3,7,7,7,9].

(e) Function *minus* returns the non-decreasing list of integers that are in a first but not in a second such list.

Example: *minus* [0,1,1,1,3,3,7] [1,1,7,7,9] = [0,1,3,3].

Question 4 The Choices of Methodology (15 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 the list [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 **recursion**, **non**-defensively construct two different SML functions, both named *compress*, for computing the compression of a list, following two methodological variants:

(a) Give the **common** specification for the two functions, using the definitions above.

(b) Construct a function by induction on the number of **elements** of the list that is to be compressed.

(c) Construct a function by induction on the number of **plateaus** of the list that is to be compressed.

Question 5 The Usefulness of Reuse (10 points)

Given a list *L*, the function *reducePlateau* returns a copy of *L* where all the plateaus of *L* have been reduced to one element. For example, *reducePlateau* [ ] = [ ] and *reducePlateau* [1,2,2,2,7,3,3] = [1,2,7,3]. As a counter-example, *reducePlateau* [1,2,2,2,7,3,3] ≠ [3,7,2,1].

Using **recursion**, **non**-defensively construct — by induction on the number of plateaus in *L* (and **not** on any other expression) — an SML function for the requirement above.

Hint: Is there a function elsewhere in this exam that you can (adapt and) reuse?

Question 6 The Usefulness of Generalisation (10 points)

Consider the following SML function declarations:

```
fun f 0 = 1
  | f n = n * f (n-1)
fun g 0 y = y
  | g x y = g (x-1) (x*y)
```

Answer the following sub-questions:

(a) What is the most likely specification of `f`?

(b) **Step-by-step** reduce the following SML expressions, **ignoring** the details about anonymous functions:

`f 4`

`g 4 1`

(c) What differences do you observe in these reductions? Discuss them!

(d) Write a new, **non**-recursive SML function for `f`, in terms of `g`, establishing `g` as a generalisation of `f`.