

# TENTAMEN I CD5100 FUNCTIONAL PROGRAMMING

Friday Oct. 27th, 2006, 14:30 – 19:30

## SUGGESTED SOLUTIONS

---

---

### UPPGIFT 1 (4 POÄNG)

We define a function “capwords” to carry out the requested functionality. A locally defined function `f` converts a string into starting with a capital letter followed by small letters. `f` is then simply mapped over the list of strings. Also in the definition of `f`, `toLower` is mapped over the tail of the string.

```
capwords = map f
  where
    f [] = []
    f (x:xs) = toUpper x : map toLower xs
```

### UPPGIFT 2 (4 POÄNG)

A simple IO action-loop which reads `s`, applies the composed function `unwords . capwords . words` to `s`, and prints the result.

```
capinput =
  do s <- getLine
     case s of [] -> return()
               otherwise -> do putStrLn ((unwords . capwords . words) s)
                               capinput
```

### UPPGIFT 3 (3 POÄNG)

The result is 17. `repeat 17` is an infinite list: due to Haskell’s lazy evaluation an answer is returned, despite that the argument is infinite, since only the needed elements are calculated. This is an illustration of how the evaluation proceeds:

```
select (repeat 17) 1 => select (17 : (repeat 17)) 1
                    => select (repeat 17) 0
                    => select (17 : (repeat 17)) 0
                    => 17
```

### UPPGIFT 4 (4 POÄNG)

One can use the mathematical fact that, whenever  $n \geq 0$ , holds that  $x^n = x^{\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil}$ , where  $\lfloor x \rfloor$  is the closest whole number  $\leq x$  and  $\lceil x \rceil$  the closest whole number  $\geq x$ . It holds that  $\text{div } n \ 2 = \lfloor n/2 \rfloor$ , and  $\lceil n/2 \rceil$  equals  $\text{div } n \ 2$  when  $n$  is even and  $\text{div } n \ 2 + 1$  when  $n$  is odd. Once functions for these are defined, it is easy to give the solution:

```
div2 n = div n 2
div2up n = let k = div2 n
           in if k * 2 == n then k else k + 1

power x 0 = 1
power x 1 = x
power x n = power x (div2 n) * power x (div2up n)
```

**UPPGIFT 5** (7 POÄNG)

a) It is possible to solve the assignment with a directly recursive function. However, we instead define a function `insert`, which inserts an element into a sorted list such that the result also is sorted, and we then use `foldr` with `insert` to define insertion sort in a simple and elegant way:

```
insert x [] = [x]
insert x (y:ys) | x <= y = x:(y:ys)
                | otherwise = y:insert x ys
```

```
isort xs = foldr insert [] xs
```

b) `isort` is a function taking a list and returning a list of the same type, i.e., if it has a type then this type must be of the form `[a] -> [a]`. Are there any limitations on the type `a`? Yes, this is the type for the elements which `insert` inserts in a list, and these elements are compared with the `<=`-operator which has type `Ord a => a -> a -> Bool`. Thus, `isort` must have the type `Ord a => [a] -> [a]`. There are no further limitations on `a`.

**UPPGIFT 6** (5 POÄNG)

a)

```
data Primitive = Zero | One | Many deriving Eq, Show
```

b) The definition of `+` should be self-evident. For `negate` holds that we set negative results to zero (`Zero`) since our number system cannot represent negative numbers. For the same reason, we let `fromInteger` map negative numbers of type `Integer` to `Zero`.

```
Instance Num Primitive where
```

```
Zero + Zero = Zero
Zero + One = One
Zero + Many = Many
One + Zero = One
Many + Zero = Many
One + One = Many
Many + One = Many
One + Many = Many
Many + Many = Many
```

```
negate _ = Zero
```

```
fromInteger 0 = Zero
fromInteger 1 = One
fromInteger n | n < 0 = Zero
              | otherwise = Many
```

**UPPGIFT 7** (3 POÄNG)

There are two different chains of reductions (in each step, the redex is underlined):

$$\begin{array}{l} (\lambda y.z) \underline{((\lambda x.x x) x)} \rightarrow (\lambda y.z) \underline{(x x)} \rightarrow z \\ (\lambda y.z) \underline{((\lambda x.x x) x)} \rightarrow z \end{array}$$

In both cases, the result is `z`.