RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHR- UND FORSCHUNGSGEBIET INFORMATIK II
RWTH Aachen · D-52056 Aachen · GERMANY
http://www-i2.informatik.rwth-aachen.de/lufgi2.html

Prof. Dr. Jürgen Giesl
Darius Dlugosz

Exercises *Functional Programming* – Sheet 1

Solutions will be collected until Thursday, May 3, 2001.

Exercise 1 (2 points)

Consider a language of expressions for representing integers defined by the following syntax rules:

(i) $zero$ is an expression

(ii) if e, e_1 and e_2 are expressions, then so are $succ(e)$, $pred(e)$ and $add(e_1, e_2)$

An evaluator reduces expressions in this language by applying the following rules repeatedly until no longer possible:

$$succ(pred(e)) = e$$

$$pred(succ(e)) = e$$

$$add(zero, e_2) = e_2$$

$$add(succ(e_1), e_2) = succ(add(e_1, e_2))$$

$$add(pred(e_1), e_2) = pred(add(e_1, e_2))$$

Simplify the expression

(a) $succ(pred(succ(pred(pred(zero)))))$

(b) $add(succ(pred(zero)), zero)$

In how many ways can the reduction rules be applied to this expression?
Do they always lead to same final result?
Which evaluation is strict and which is non-strict?

(c) Suppose we have an additional reduction rule: $pred(zero) = zero$.

Simplify the expression in (a) and also answer the previous questions.

(d) Suppose we have an additional reduction rule: $double(e) = add(e, e)$.

Simplify the expression $double(pred(succ(zero)))$ using strict, non strict, and lazy evaluation.

Exercise 2 (1 point)

(a) Suppose f and g have the following types:

$$f :: Int \to Int$$
$$g :: Int \to (Int \to Int)$$

Let h be defined by

$$h :: ...$$
$$h\ x\ y = f\ (g\ x\ y)$$

Fill in the correct type assignment for h.

(b) Give examples of functions with the following types:

(i) $(Int \to Int) \to Int$
(ii) $(Int \to Int) \to (Int \to Int)$
(iii) $(Int \to [Int]) \to (Int \to Int)$

(c) Suppose we curry the arguments of some function $f :: (Int, Int, Int) \to Int$, so that we can write $f\ a\ b\ c$ rather than $f\ (a, b, c)$. What is the type of the curried version?

Exercise 3 (1 point)

Which of the following equations between two lists are correct? Why?

(a) $[] : xs = xs$

(b) $[] : xs = [[], xs]$

(c) $xs : [] = xs$

(d) $xs : [] = [xs]$

(e) $x : y = [x, y]$

(f) $(x : xs) + +ys = x : (xs + +ys)$

The operation ++ concatenates two lists. For example: $[1, 2, 3] + +[2, 3] = [1, 2, 3, 2, 3]$.

Exercise 4 (2 points)

(a) Give a Haskell implementation of the following function only using the operation $+$, comparing operations $==, >, etc.$ and the function $minus$ defined in (a).

   (i) $minus :: Int \to Int \to Int$, which subtracts two natural numbers.
   In particular $minus\ x\ y = 0$ holds if $x < y$.

   (ii) $gcd :: Int \to Int \to Int$, which computes the greatest common divisor of two natural numbers.

   (iii) $lcm :: Int \to Int \to Int$, which computes the least common multiple of two natural numbers.

(a) Using pattern matching declarations give the definitions of the boolean functions $and, or, xor :: (Bool, Bool) \to Bool$.

Implement the functions ha and fa, corresponding to half-adder and full-adder, in Haskell notation using your defined boolean functions

$$ha :: (Bool, Bool) \to (Bool, Bool)$$
$$fa :: (Bool, Bool, Bool) \to (Bool, Bool)$$

Exercise 5 (4 points)

The Fibonacci numbers $f_0, f_1, ...$ are defined by the rule that $f_0 = 0, f_1 = 1$ and $f_{n+2} = f_n + f_{n+1}$ for all $n \geq 0$.

(a) Give the simplest possible Haskell implementation of the function fib that takes an integer n and returns f_n.

(b) Modify the implementation from (a) so that the result is computed in linear time.

(c) Are local declarations and/or auxiliary functions necessary for the implementation from (b)? Why?

Exercise 6: (2 points)

The function $length$ which computes the length of a list of integers can be implemented as follows:

$$
\begin{aligned}
& length :: [Int] \to Int \\
& length\ xs = len\ 0\ xs \\
& \quad where\ len\ n\ [] = n \\
& \quad\quad\quad len\ n\ (\_ : xs) = len\ (n+1)\ xs
\end{aligned}
$$

This function uses the function len that has an additional parameter to accumulate the result. The function len is tail-recursive, because the recursive function calls in the right-hand side of the definition are not inside the arguments of other functions.

Using the technique of tail-recursion give a Haskell implementation of the following functions:

(a) *prod*, which computes the product of integers of a list

(b) fac, which computes the factorial of a natural number

(c) $reverse$, which reverses a list