**McMaster University**
**Department of Computing and Software**
**Dr. W. Kahl**

**SFWR ENG 3E03**
**Exercise Sheet 4**
**Solution Hints**

# Design and Selection of Programming Languages

5 October 2006

**Exercise 4.1**

Assume the following Haskell definitions:

$size = 10$
$square\ n = n * n$

Add a definition for $cube$ with the obvious meaning, and manually perform single-stepped expression evaluation for the expression "$cube\ size - cube\ (size - 2)$".

**Solution Hints**

$cube\ n = n * square\ n$

Then:

$cube\ size - cube\ (size - 2)$
$= (size * square\ size) - cube\ (size - 2)$ −− unfolding cube definition
$= (10 * square\ 10) - cube\ (10 - 2)$ −− unfolding size definition
$= (10 * (10 * 10)) - cube\ (10 - 2)$ −− unfolding square definition
$= (10 * 100) - cube\ (10 - 2)$ −− multiplication
$= 1000 - cube\ (10 - 2)$ −− multiplication
$= 1000 - (10 - 2) * square\ (10 - 2)$ −− unfolding cube definition
$= 1000 - 8 * square\ 8$ −− subtraction
$= 1000 - 8 * (8 * 8)$ −− unfolding square definition
$= 1000 - 8 * 64$ −− multiplication
$= 1000 - 512$ −− multiplication
$= 488$ −− subtraction

---

**Exercise 4.2**

Haskell has predefined types $Float$ for single-precision floating point numbers (which we ignore in the following) and $Double$ for double-precision floating point numbers.

Standard mathematical functions like

$sqrt, sin, atan :: Double \rightarrow Double$

and $pi :: Double$ are also available; $x$ ^ $k$ stands for $x^k$ if $k$ is natural; $x ** q$ can be used for $x^q$ where both $x$ and $q$ are of type $Double$.

Define the following Haskell functions, with the meanings obvious from their names:

(a) $sphereVolume :: Double \rightarrow Double$

(b) $sphereSurface :: Double \rightarrow Double$

(c) $centuryToPicosecond :: Integer \rightarrow Integer$

Try the last one in C or Java, too; test both, and compare the results

**Solution Hints**
Introduce auxiliary constants or functions at least for (c)!

```
sphereVolume :: Double -> Double
sphereVolume r = 4/3 * pi * r ^ 3

sphereSurface :: Double -> Double
sphereSurface r = 4 * pi * r^2

centuryToPicosecond :: Integer -> Integer
centuryToPicosecond c  = c * daysPerCentury * 24 * 3600 * 10 ^ 12

daysPerCentury, daysPerYear, leapYearsPerCentury :: Integer
daysPerCentury =  100 * daysPerYear + leapYearsPerCentury

leapYearsPerCentury = 24
daysPerYear         = 365
```

(This does not take leap-seconds into account.)

In C or Java, some extra effort would be required to make this work with some integral type, since:

```
Main> centuryToPicosecond 1
3155673600000000000000
Main> 2 ^ 64
18446744073709551616
```

---

**Exercise 4.3**

Define the following Haskell functions:

(a) *stutter* :: [ *a* ] → [ *a* ]

duplicates each element of its argument lists, e.g.: *stutter* [1,2,3] = [1,1,2,2,3,3]

**Solution Hints**

```
stutter :: [a] -> [a]
stutter []     = []
stutter (x:xs) = x : x : stutter xs
```

---

(b) *splits* :: [ *a* ] → [ ( [ *a* ] , [ *a* ] ) ]

delivers for each argument list all possibilities to segment it into non-empty prefix and suffix, e.g.:

*splits* [1,2,3] = [ ( [1] , [2,3] ) , ( [1,2] , [3] ) ]

(The order is irrelevant.)

**Solution Hints**

```
–– most “natural”:
splits [] = []
```

```
splits [x] = []
splits (x:xs) = ([x],xs) : map (pupd1 (x:)) (splits xs)
          --  = ([x],xs) : [ (x:pre, suff) | (pre,suff) <- splits
xs ]

pupd1 f (x,y) = (f x, y)

-- much less efficient:
splits' [] = []
splits' (x : xs) = spl [x] xs
 where
  spl ys [] = []
  spl ys (xs@(x : xs')) = (ys, xs) : spl (ys ++ [x]) xs'

-- roughly equally inefficient:
splits" xs = map (flip splitAt xs) [1 .. length xs - 1]
```

---

(c) *rotations* :: [*a*] → [[*a*]]

delivers for each argument list all different results of rotations, each result only once, e.g.:

*rotations* [1,2,3] = [[1,2,3], [3,1,2], [2,3,1]]

(The order is irrelevant.)

**Solution Hints**

```
rotations :: [a] -> [[a]]
rotations xs = xs : map (uncurry (flip (++))) (splits xs)
         --  = xs : [ suff ++ pre  |  (pre, suff) <- splits xs ]

rotations' xs = r [] xs
 where
  r ys [] = [ys]
  r ys xs@(x : xs') = (xs ++ ys) : r (ys ++ [x]) xs'
```

---

(d) *permutations* :: [*a*] → [[*a*]]

delivers for each argument list all different results of permutations, each result only once, e.g.:

*permutations* [1,2,3] = [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]

(The order is irrelevant.)

**Solution Hints**

```
permutations :: [a] -> [[a]]
permutations [] = [[]]
permutations xs =
  concat [ map (y:) (permutations ys)  |  (y : ys) <- rotations
xs ]

permutations' [] = [[]]
```

```
permutations' xs = concatMap permAux (rotations xs)
  where
   permAux (y : ys) = map (y:) (permutations ys)
```

---

**Exercise 4.4 — Defining Haskell Functions (40% of Midterm 1, 2003)**

Define the follwing Haskell functions (the solutions are independent of each other):

(a) *polynomial* :: [*Double*] → *Double* → *Double*

such that for coefficients $c_0, c_1, c_2, \ldots, c_n$ and any $x$ the following holds:

$$\mathit{polynomial}\ [c_0, c_1, c_2, \ldots, c_n]\ x = c_0 + c_1 \cdot x + c_2 \cdot x^2 + \cdots + c_n \cdot x^n$$

e.g.: *polynomial* [3,4,5] 100.0 = 50403.0

**Hint:** Use Horner's rule:

$$c_0 + c_1 \cdot x + c_2 \cdot x^2 + \cdots + c_n \cdot x^n \;=\; c_0 + x \cdot (c_1 + x \cdot (c_2 + \cdots + x \cdot (c_n) \cdots))$$

**Solution Hints**

```
polynomial, polynomial1, polynomial2, polynomial3 :: [Double] -> Double -> Double
polynomial [] x = 0
polynomial (c : cs) x = c + x * polynomial cs x

polynomial1 cs x = foldr (\ c r -> c + x * r) 0 cs

polynomial2 cs x = foldr (\ c -> (c +) . (x *)) 0 cs

polynomial3 cs x = foldr ((. (x *)) . (+)) 0 cs
```

If we swap the argument order, we can easily abstract away *cs*. The "λ-lifting" of the argument to *foldr* however leads to rather unreadable code, presented here as a puzzle: Do the transformations leading there yourself!

```
polynomial4 :: Double -> [Double] -> Double
polynomial4 x = foldr ((. (x *)) . (+)) 0
```

---

(b) *findJump* :: *Integer* → [*Integer*] → (*Integer* , *Integer*)

takes an integer *d* and a list and returns the first pair of **adjacent** elements of the list such that the values of these two elements are farther than *d* apart, e.g.,

*findJump* 3 [2,3,4,2,5,3,6,2,3,5,4,1,6] = (6,2)

If the list contains no such values, an error is produced.

**Solution Hints**

```
findJump :: Integer -> [Integer] -> (Integer, Integer)
findJump d [] = error "findJump: empty list"
findJump d [x] = error "findJump: singleton list"
```

```
findJump d (x : xs≅(y : ys)) = if abs (x − y) > d
                               then (x, y)
                               else findJump d xs
```

---

(c) $suffixes :: [a] \to [[a]]$

delivers for each argument list all its suffixes, e.g.:

$suffixes\ [1,2,3,4] = [[1,2,3,4],[2,3,4],[3,4],[4],[\,]]$

(The order is irrelevant.)

**Solution Hints**

```
suffixes :: [a] → [[a]]
suffixes [] = [[]]
suffixes xs≅(y : ys) = xs : suffixes ys
```

---

(d) $diagonal :: [[a]] \to [a]$

interprets its argument as a matrix (represented as in Exercise 2.1), which may be assumed to be square, and returns the main diagonal of that matrix, e.g.:

$diagonal\ [[1,2,3],[4,5,6],[7,8,9]] = [1,5,9]$

**Solution Hints**

```
diagonal, diagonal' :: [[a]] → [a]
diagonal [] = []
diagonal ([] : xss) = error "not square"
diagonal ((x:xs) : xss) = x : diagonal (map tail xss)

diagonal' = zipWith ((head .) ∘ drop) [0..]
```

Discuss the use of *head* in the variant *diagonal'*!

---

(e) $isSquare :: [[a]] \to Bool$

determines whether its argument corresponds to a list-of-lists representation (as in Exercise 2.1) of a ***square*** matrix.

**Solution Hints**

The following works only for finite lists of finite lists:

```
isSquare, isSquare' :: [[a]] → Bool
isSquare xs = all (((length xs) ≡) ∘ length) xs

isSquare' xs = all ((length xs) ≡) (map length xs)
```

(It is undecidable whether an inifinite list of lists has only infinite element lists.)

---

**Exercise 4.5 — Haskell Evaluation (30% of Midterm 1, 2003)**

Assume the following Haskell definitions to be given:

```
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)

concat           = foldr (++) []

(||)             :: Bool -> Bool -> Bool  -- Boolean disjunction: or
True  || _       =  True
False || b       =  b

any p = foldr ((||) . p) False

gen f (x,s) = x : gen f (f x s)

foo k n = (k + n, n + 2)
```

Simulate Haskell evaluation for the following expressions (write down the sequence of intermediate expressions):

(a) `foldr (*) 1 [6,7]`

(b) `any (> 0) (gen foo (0,1))`

**Solution Hints**

foldr (*) 1 [6,7]
= 6 * (foldr (*) 1 [7])
= 6 * (7 * (foldr (*) 1 []))
= 6 * (7 * 1)
= 6 * 7 -- X
= 42

any (> 0) (gen foo (0,1))
= foldr ((||) . (> 0)) False (gen foo (0,1))
= foldr ((||) . (> 0)) False (0 : gen foo (foo 0 1))
= ((||) . (> 0)) 0 (foldr ((||) . (> 0)) False (gen foo (foo 0 1)))
= (||) ((> 0) 0) (foldr ((||) . (> 0)) False (gen foo (foo 0 1)))
= (||) (0 > 0) (foldr ((||) . (> 0)) False (gen foo (foo 0 1))) -- X
= (||) False (foldr ((||) . (> 0)) False (gen foo (foo 0 1)))
= foldr ((||) . (> 0)) False (gen foo (foo 0 1))
= foldr ((||) . (> 0)) False (gen foo (0 + 1, 1 + 2))
= foldr ((||) . (> 0)) False ((0 + 1) : gen foo (foo (0 + 1) (1 + 2)))
= ((||) . (> 0)) (0 + 1) (foldr ((||) . (> 0)) False (gen foo (foo (0 + 1) (1 + 2))))
= (||) ((> 0) (0 + 1)) (foldr ((||) . (> 0)) False (gen foo (foo (0 + 1) (1 + 2))))
= (||) ((0 + 1) > 0) (foldr ((||) . (> 0)) False (gen foo (foo (0 + 1) (1 + 2)))) -- X
= (||) (1 > 0) (foldr ((||) . (> 0)) False (gen foo (foo 1 (1 + 2))))
= (||) True (foldr ((||) . (> 0)) False (gen foo (foo 1 (1 + 2))))
= True

**Exercise 4.6 — Defining Haskell Functions (20% of Midterm 1, 2004)**

Define the follwing Haskell functions (the solutions are independent of each other):

(a) $sum :: [Integer] \rightarrow Integer$

such that $sum\ xs$ evaluates to the sum of all elements of the list $xs$.

(b) $all :: (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool$

such that $all\ p\ xs$ evaluates to **True** if $p$ considered as a predicate holds for all elements of $x$, and to **False** if there is at least one element in $xs$ for which $p$ does not hold.

E.g., $all\ (>1)\ [2..10] = \mathbf{True}$

(c) $selMod :: Integer \rightarrow [Integer] \rightarrow [Integer]$

such that $selMod\ k\ xs$ selects from the list $xs$ all those elements that are equivalent to $k$ modulo $k+1$, e.g.,

$selMod\ 2\ [2, 3, 8, 1, 2, 5] = [2, 8, 2, 5]$

(d) $sources :: Eq\ a \Rightarrow [(a, a)] \rightarrow [a]$

such that $sources\ ps$ returns the *sources* of the graph $ps$.

Here, the list $ps$ of pairs is considered as representing a simple graph by representing each edge from node $x$ to node $y$ by the pair $(x, y)$.

The *context* "$Eq\ a \Rightarrow$" just means that you may use the equality test for elements of type $a$, i.e., $(==) :: a \rightarrow a \rightarrow Bool$.

Example: $sources\ [(2,3), (3,4), (1,4), (1,5), (2,5)] = [2,1]$

(The order is irrelevant.)

**Solution Hints**

$sum = foldl\ (+)\ 0$

$sum = foldr\ (+)\ 0$

$sum\ [\,] = 0$
$sum\ (x : xs) = x + sum\ xs$

$all = foldr\ (\&\&)\ \mathbf{True}$

$all\ p\ [\,] = \mathbf{True}$
$all\ p\ (x : xs) = p\ x\ \&\&\ all\ p\ xs$

$selMod :: Integer \rightarrow [Integer] \rightarrow [Integer]$
$selMod\ k\ xs = [\ x \mid x \leftarrow xs\ ,\ x\ \text{`}mod\text{`}\ (k+1) \equiv k\ ]$

$sources, sources' :: Eq\ a \Rightarrow [(a, a)] \rightarrow [a]$

$$
\begin{aligned}
&\mathit{sources}\ \mathit{ps} = \mathbf{let}\ (\mathit{srcs}, \mathit{trgs}) = \mathit{unzip}\ \mathit{ps} \\
&\qquad\quad \mathbf{in}\ \mathit{filter}\ (\text{`}\mathit{notElem}\text{`}\ \mathit{trgs})\ \mathit{srcs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sources'}\ \mathit{ps} = \mathbf{let}\ \mathit{trgs} = [\ \mathit{snd}\ p \mid p \leftarrow \mathit{ps}\ ] \\
&\qquad\quad \mathbf{in} = [\ x \mid (x, y) \leftarrow \mathit{ps}, x\ \text{`}\mathit{notElem}\text{`}\ \mathit{trgs}\ ]
\end{aligned}
$$