# CSc 372

# Comparative Programming Languages

# 16 : Haskell — Exercises


Department of Computer Science
University of Arizona

collberg@gmail.com





Christian Collberg

# List Prefix

- Write a recursive function `begin xs ys` that returns true if `xs` is a prefix of `ys`. Both lists are lists of integers. Include the type signature.

```
> begin [] []
True
> begin [1] []
False
> begin [1,2] [1,2,3,4]
True
> begin [1,2] [1,1,2,3,4]
False
> begin [1,2,3,4] [1,2]
```

# List Containment

- Write a recursive function `subsequence xs ys` that returns true if `xs` occurs anywhere within `ys`. Both lists are lists of integers. Include the type signature.
- Hint: reuse `begin` from the previous exercise.

```
> subsequence [] []
True
> subsequence [1] []
False
> subsequence [1] [0,1,0]
True
> subsequence [1,2,3] [0,1,0,1,2,3,5]
True
```

# Mystery

- Consider the following function:

```
mystery :: [a] -> [[a]]
mystery [] = [[]]
mystery (x:xs) = sets ++ (map (x:) sets)
                 where sets = mystery xs
```

- What would `mystery [1,2]` return? `mystery [1,2,3]`?
- What does the funtion compute?

# foldr

- Explain what the following expressions involving `foldr` do:
  1. `foldr (:)  [] xs`
  2. `foldr (:)  xs ys`
  3. `foldr ( y ys -> ys ++ [y]) [] xs`

# shorter

- Define a function `shorter xs ys` that returns the shorter of two lists.

```
> shorter [1,2] [1]
[1]
> shorter [1,2] [1,2,3]
[1,2]
```

# stripEmpty

- Write function `stripEmpty xs` that removes all empty strings from `xs`, a list of strings.

```
> stripEmpty ["", "Hello", "", "", "World!"]
["Hello","World!"]
> stripEmpty [""]
[]
> stripeEmpty []
[]
```

# merge

- Write function `merge xs ys` that takes two ordered lists `xs` and `ys` and returns an ordered list containing the elements from xs and ys, without duplicates

```
> merge [1,2] [3,4]
[1,2,3,4]
> merge [1,2,3] [3,4]
[1,2,3,4]
> merge [1,2] [1,2,4]
[1,2,4]
```

# Function Composition

- Rewrite the expression

  ```
  map f (map g xs)
  ```

  so that only a single call to map is used

# Reduce

- Let the Haskell function reduce be defined by

```
reduce f []     v = v
reduce f (x:xs) v = f x (reduce f xs v)
```

- Reconstruct the Haskell functions length, append, filter, and map using reduce. More precisely, complete the following schemata (in the simplest possible way):

```
mylength xs     = reduce ___ xs ___
myappend xs ys = reduce ___ xs ___
myfilter p xs  = reduce ___ xs ___
mymap f xs     = reduce ___ xs ___
```

# 372 Midterm 2004 – Problem 1

- Write a non-recursive function

  `invert :: [Bool] -> [Bool]`

  that turns all `True` values into `False`, and `False` values into `True`. Example:

  ```
  > invert [True,False]
  [False,True]
  ```

# 372 Midterm 2004 – Problem 2

- Write a non-recursive function `count p xs` that takes a predicate `p` and a list `xs` of elements (of arbitrary type) as arguments and returns the number of elements in the list that satisfies `p`:

  ```
  > count even [1,2,3,4,5]
  2
  ```

- Ideally, you should define the function using composition of higher-order functions from the standard prelude!

# 372 Midterm 2004 – Problem 3

- Write a non-recursive function `blend xs ys` that takes two lists of elements (of arbitrary type) as argument, and returns a list where the elements have been taken alternatingly from `xs` and `ys`:

```
> blend [1,2,3] [4,5,6]
[1,4,2,5,3,6]
```

  You can assume that `xs` and `ys` are of the same length.

# 372 Midterm 2004 – Problem 4

- Write a function `adjpairs` that takes a list as argument and returns the list of all pairs of adjacent elements. Examples:

```
> adjpairs []
[]
> adjpairs [1]
[]
> adjpairs [1,2]
[(1,2)]
> adjpairs [1,2,3]
[(1,2),(2,3)]
> adjpairs [1,2,3,4,5,6]
[(1,2), (2,3), (3,4), (4,5), (5,6)]
```

- Give both a recursive and a non-recursive solution!

# 372 Midterm 2004 – Problem 5

- Write a non-recursive function `section f c xs` that extracts a sublist of the list `xs` starting at position `f` and which is `c` elements long. Use 0-based indexing. Assume that `xs` has at least `f+c` elements. Examples:

```
> section 0 1 [1,2,3,4,5]
[1]
> section 0 3 [1,2,3,4,5]
[1,2,3]
> section 1 3 [1,2,3,4,5]
[2,3,4]
> section 4 1 [1,2,3,4,5]
[5]
```

# 372 Midterm 2004 – Problem 6

- Given these Haskell function definitions

```
duh :: [Int] -> Int -> [[Int]]
duh xs a = duh' xs a []

duh' [] _ [] = []
duh' [] _ xs = [xs]
duh' (x:xs) a ys
     | a == x          = nut ys (duh' xs a [])
     | otherwise       = duh' xs a (ys ++ [x])

nut [] xs = xs
nut xs ys = xs : ys
```

# 372 Midterm 2004 – Problem 6. . .

answer these questions:

1. What is the result of `nut [] [[1,2]]`?
2. What is the result of `nut [2] [[1,2]]`?
3. What is the most general type of `nut`?
4. What is the result of `duh [1,2,3] 1`?
5. What is the result of `duh [1,2,3,1,4] 1`?

# 372 Midterm 2004 – Problem 7

What are the results of these Haskell expressions?

1. ```
   filter p [[1],[1,2],[1,2,3],[1,2,3,4]]
       where p xs = length xs > 2
   ```
2. ```
   filter (not . even . length) xs
       where xs = [[1],[1,2],[1,2,3],[1,2,3,4]]
   ```
3. ```
   foldr (\ xs i -> length xs + i) 0 xs
       where xs = [[1],[1,2],[1,2,3],[1,2,3,4]]
   ```
4. ```
   iterate id 1
   ```
5. ```
   (fst. head . zip [1,2,3]) [4,5,6]
   ```
   ```

# 372 Final 2004 – Problem 1

- Given these Haskell function definitions

```
mystery :: [a] -> [[a]]
mystery xs = [take n xs,drop n xs]
             where n  = h  xs

h :: [a] -> Int
h [] = 0
h [_] = 0
h (_:_:xs) = 1 + h xs
```

what does the expression

```
mystery [1,2,3,4,5]
```

return?

# 372 Final 2004 – Problem 2

1. What is *referential transparency*? Illustrate with an Icon procedure and a Haskell function.
2. Haskell is a *lazy* language. What does this mean?