Advanced Programming
Andrew Black and Tim Sheard

Lecture 4
Intro to Haskell
Assignment #2

- You are to write a number of functions in Haskell
  - `explode:: String -> [Char]`
  - `digitValue:: [Char] -> [Int]`
  - `reverse:: [Int] -> [Int]`
  - `pairedWithPowersOf10:: [Int] -> [(Int,Int)]`
  - `pairWiseProduct ::[(Int,Int)] -> [Int]`
  - `sum :: [Int] -> Int`
  - `toInt :: String -> Int`

- Write 5 functions (sum & reverse are primitives, and don’t need to be written).
- Write each one using pattern matching and recursion.
- Create a number of tests for each function
  - `test1 = explode "tim" == ['t', 'i', 'm']`
- Turn in an elegant program that communicates your solution well.
List operations

• In Haskell we use library functions all the time.
  - import Char
  - import List

• The list library functions are particularly useful.
  - These library functions often take a function as an argument
  map :: (a -> b) -> [a] -> [b]
  find :: (a -> Bool) -> [a] -> Maybe a
  filter :: (a -> Bool) -> [a] -> [a]
  all :: (a -> Bool) -> [a] -> Bool
  any :: (a -> Bool) -> [a] -> Bool
  foldr :: (a -> b -> b) -> b -> [a] -> b

• It is worth studying these functions closely
map captures a pattern

- Add one to every element of a list

  \[
  \text{addone \hspace{1mm} \begin{cases}
  [\hspace{1mm} & \text{if } x = \begin{cases}
  [\hspace{1mm} & \text{if } x = \begin{cases}
  \] \\
  [\hspace{1mm} & \text{if } x > \begin{cases}
  \] \\
\end{cases}
\end{cases}
\end{cases}
\end{cases}
\]
\]

  \[
  \begin{align*}
  \text{addone} \hspace{1mm} & (x : xs) = (x + 1) : \text{addone} \hspace{1mm} xs \\
\end{align*}
\]

  Main> addone [2,4,6]
  [3,5,7]

- Turn a list of ints into a list of strings

  \[
  \text{stringy :: \hspace{1mm} [Int] \rightarrow [String]}
  \]

  \[
  \begin{align*}
  \text{stringy} & \hspace{1mm} [] = [] \\
  \text{stringy} & \hspace{1mm} (x : xs) = (\text{show} \hspace{1mm} x) : \text{stringy} \hspace{1mm} xs \\
\end{align*}
\]

  Main> stringy [2,5,9]
  ["2","5","9"]
More map patterns

• Negate every element of a list

  \[
  \text{negL} \; [] = [] \\
  \text{negL} \; (x:xs) = (\text{not} \; x) : \text{negL} \; xs
  \]

Main> negL [True, 3 > 4]
[False, True]
The pattern

addone [] = []
addone (x:xs) = (x+1) : addone xs
stringy [] = []
stringy (x : xs) = (show x) : stringy xs
negL [] = []
negL (x:xs) = (not x) : negL xs

map f [] = []
map f (x:xs) = (f x):(map f xs)

map (\ x -> (x+1)) [2,3,4] → [3,4,5]

map show [2,5,7] → ["2","5","7"] list

map not [True, 3 > 4] → [False,True] : bool list
Maybe type

data Maybe a = Nothing | Just a

Used when an answer maybe be missing
Anonymous functions

• **Study:** (\ x -> (x+1))
  - It is an anonymous function. A function without a name.
  - It has one parameter “x”
  - It adds one to its parameter, and returns the result.
    (\ x -> (x+1)) 4 \rightarrow 5 :: Int

• Any non-recursive function can be written anonymously.
  - (\ x -> x == 5)
    - Tests if its parameter is equal to 5
      map (\ x -> x==5) [1,4,5,3,5] \rightarrow
        [False,False,True,False,True] :: [Bool]

  - (\ x -> \ y -> (x,y))
    - Has two parameters
    - Returns a pair

  - (\ (x,y) -> (not y, x+3))
    - What is the type of this function?
find

• Used for searching a list.
  \[
  \text{find} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Maybe } a
  \]

• Uses a function as a parameter to determine if the search is successful.

• E.g. Is there an even element in a list?
  \[
  \text{find even } [1,3,5] \rightarrow \\
  \text{Nothing :: Maybe Int}
  \]
  \[
  \text{find even } [1,3,4] \rightarrow \\
  \text{Just 4 :: Maybe Int}
  \]
find and anonymous functions

find (\ x -> x == "Tim")

["Tom", "Jane"] →

Nothing :: Maybe String

find (\ x -> even x && x>10)

[2,4,5,12] →

Just 12 :: Maybe Int
filter

Filter keeps some elements, and throws away others.

\[
\text{filter} :: (a \to \text{Bool}) \to [a] \to [a]
\]

It uses a function \(p\) as a parameter to decide which elements to keep \(p \ x \ = \ True\), and which to throw away \(p \ x \ = \ False\)

\[
\text{filter even \ [1,2,3,4,5,6] } \rightarrow \\
[2,4,6] :: \ [\text{Int}]
\]
filter and anonymous functions

people =
   [("tim",22),("john",18),("jane",25),("tim",8 )];

filter (\ (nm,age) -> nm /= "tim" || age>10)
  people;

[("tim",22),("john",18),("jane",25)]
:: [(String,Int)]
any

- “any” is like “find” in that it searches a list
  - But rather than the element that completes the search it is only interested in if such an element exists.
  - \( \text{any} :: (a \to \text{Bool}) \to [a] \to \text{Bool} \)

- Uses a function as a parameter to determine if the search is successful.

\[
\text{any even } [2,3,5] \Rightarrow \text{True} :: \text{Bool}
\]

- Note that even if only 1 element in the list causes the function to be true, any returns true.
all

- all tests elements in a list for a property. It returns true only if every element has that property.

  all :: (a -> Bool) -> [a] -> Bool

- Uses a function as a parameter to perform the test.

  all even [2,4,5] -> False :: Bool

- Any and all are related functions. They are duals.

  \[ \neg \text{all } p\ \text{xs} = \text{any } (\neg \ . \ p)\ \text{xs} \]
foldr captures a pattern

• Add up every element in a list.
  \[ \text{sum} \; [] = 0 \]
  \[ \text{sum} \; (x:xs) = x + (\text{sum} \; xs) \]

• Compute the maximum element in a list of natural numbers (Integers \( \geq 0 \)).
  \[ \text{maximum} \; [] = 0 \]
  \[ \text{maximum} \; (x:xs) = \max x (\text{maximum} \; xs) \]

• Compute if every element in a list of boolean is true.
  \[ \text{allTrue} \; [] = \text{True} \]
  \[ \text{allTrue} \; (x:xs) = x \; && \; (\text{allTrue} \; xs) \]
Pattern

\[
\text{sum} \ [\ ] = 0 \\
\text{sum} (x:xs) = x + (\text{sum} \ xs)
\]

\[
\text{maximum} \ [\ ] = 0 \\
\text{maximum} (x:xs) = \text{max} \ x \ (\text{maximum} \ xs)
\]

\[
\text{allTrue} \ [\ ] = \text{True} \\
\text{allTrue} (x:xs) = x \ &\& (\text{allTrue} \ xs)
\]

\[
\text{foldr} \ \text{acc} \ \text{base} \ [\ ] = \text{base} \\
\text{foldr} \ \text{acc} \ \text{base} \ (x:xs) \\
\hspace{1em} = \text{acc} \ x \ (\text{foldr} \ \text{acc} \ \text{base} \ xs)
\]
See the pattern in use.

\[
\text{sum } \[] = 0 \\
\text{sum } (x:xs) = x + (\text{sum } xs);
\]

\[
\text{sum } xs = \text{foldr } (+) 0 xs
\]

\[
\text{maximum } \[] = 0 \\
\text{maximum } (x:xs) = \text{max } x (\text{maximum } xs)
\]

\[
\text{maximum } xs = \text{foldr } \text{max } 0 xs
\]

\[
\text{allTrue } \[] = \text{true} \\
\text{allTrue } (x:xs) = x \&\& (\text{allTrue } xs)
\]

\[
\text{allTrue } xs = \text{foldr } (\&\&) \text{ True } xs;
\]
More on lists

- Nub – remove duplicate elements
- Sort – put list in ascending order
- Concat – merge together a list of lists
- Take and drop – manipulate prefixes of lists
- Comprehensions – special syntax for whole list at a time operations
- Zip – combine two lists
Nub, sort, and concat

Main> :t nub
nub :: Eq a => [a] -> [a]
Main> nub [1,2,3,4,1,6]
[1,2,3,4,6]

Main> :t sort
sort :: Ord a => [a] -> [a]
Main> sort [1,5,2,4,1,7]
[1,1,2,4,5,7]

Main> :t concat
concat :: [[a]] -> [a]
Main> concat [[2,3],[[]],[5]]
[2,3,5]
A detour: Hamming’s problem

• 1 is in the set
• If x is in the set, then 2x, 3x, and 5x are in the set.

• Break it into 2 parts
  1. Adding (2, 3, and 5) multiples to those already there
  2. Repeating until no new elements are added
Representing sets – detour #1

• We will represent sets of integers as lists stored in normal form.

• A normal form is a list in ascending order with no duplicates

```
norm :: [Int] -> [Int]
norm x = sort (nub x)
```
Taking one step – detour #2

```haskell
onestep xs = norm(xs ++ lessThan100)
where multiples x = [2*x, 3*x, 5*x]
    products = concat(map multiples xs)
    lessThan100 = filter (< 100) products
```
Keep applying – detour #3

Main> onestep [1]  
[1,2,3,5]  
Main> onestep (onestep [1])  
[1,2,3,4,5,6,9,10,15,25]  
Main> onestep(onestep (onestep [1]))  
[1,2,3,4,5,6,8,9,10,12,15,18,20,25,27,30,45,50,75]  
Main> onestep(onestep(onestep (onestep [1])))  
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,36,40,45,50,54,60,75,81,90]  

Main> onestep(onestep(onestep(onestep (onestep [1]))))  
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,40,45,48,50,54,60,72,75,80,81,90]  
Main> onestep(onestep(onestep(onestep(onestep (onestep [1])))))  
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,40,45,48,50,54,60,64,72,75,80,81,90,96]  
Main> onestep(onestep(onestep(onestep(onestep(onestep (onestep [1]))))))  
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,40,45,48,50,54,60,64,72,75,80,81,90,96]
Fixpoint – detour #4

fixpoint :: Eq a => (a -> a) -> a -> a
fixpoint f init = if init==next
    then next
    else fixpoint f next

    where next = f init
Parameterizing over 100 – detour #5

\[
\text{onestep} :: \text{Int} \rightarrow \text{[Int]} \rightarrow \text{[Int]}
\]
\[
\text{onestep } \text{max} \ \text{xs} = \text{norm} (\text{xs} ++ \text{lessThanMax})
\]
\[
\text{where multiples } \text{x} = [2*\text{x}, 3*\text{x}, 5*\text{x}]
\]
\[
\text{products} = \text{concat} (\text{map multiples} \ \text{xs})
\]
\[
\text{lessThanMax} = \text{filter} (< \text{max}) \ \text{products}
\]

\[
\text{hamming } \text{max} = \text{fixpoint} \ (\text{onestep } \text{max}) \ [1]
\]
More on lists - take and drop

- Two useful functions on lists
  
  Main> :t take
  take :: Int -> [a] -> [a]
  Main> take 4 [1,2,3,4,5,6,7]
  [1,2,3,4]

  Main> :t drop
  drop :: Int -> [a] -> [a]
  Main> drop 4 [1,2,3,4,5,6,7]
  [5,6,7]
Zip

- Zip combines two lists into a pair of lists

  Main> :t zip
  zip :: [a] -> [b] -> [(a,b)]

  Main> zip [1,2,3,4] ["a","c","tom","z"]
  [(1,"a"),(2,"c"),(3,"tom"),(4,"z")]

  Main> zip [1,2,3] [True,False]
  [(1,True),(2,False)]

  Main> zip [1,2,3] [1,2,3,4,5,6]
  [(1,1),(2,2),(3,3)]
zipWith

- zipWith applies a binary function to elements from two lists.
  
  \[ \text{zipWith } f \text{ xs ys} = \text{map } (\lambda (x,y) \to f \ x \ y) \ (\text{zip } \text{xs } \text{ys}) \]

Main> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

Main> zipWith (+) [1,2,3] [7,2,9]
[8,4,12]
List Comprehensions

- Generators

  Main> [1..10]
  [1,2,3,4,5,6,7,8,9,10]

  Main> [1,5..25]
  [1,5,9,13,17,21,25]

  Main> take 12 [1,3..]
  [1,3,5,7,9,11,13,15,17,19,21,23]
More comprehensions

• Manipulators

Main> [ i * 2 | i <- [2..8]]
[4,6,8,10,12,14,16]

Main> [ i * 2 | i <- [2..8], even i ]
[4,8,12,16]

Main> [(i,j) | i <- [2..4] , j <- [1..3]]
[(2,1),(2,2),(2,3),(3,1),(3,2),(3,3),
 (4,1),(4,2),(4,3)]
Hamming’s problem revisited

\[ \text{onestep' :: Int -> [Int] -> [Int]} \]
\[ \text{onestep' max xs = norm(lessThanMax)} \]
\[ \quad \text{where lessThanMax = [ j | i <- xs} \]
\[ \quad \quad , j <- [i,2*i,3*i,5*i] \]
\[ \quad \quad , j <= \text{max} ] \]

\[ \text{hamming' max = fixpoint (onestep' max) [1]} \]
HUnit

• HUnit is a unit testing framework for Haskell
• Download it from http://hunit.sourceforge.net.
• Unzip the file and put it somewhere accessible, say C:\HUnit
• Update the path directory of Hugs to point to this directory
• :set -P; c: \HUnit

Be sure an use a semicolon, or the old path entries will be lost
import HUnit

explode :: String -> [Char]
explode x = x

test1 = testCase
  (assertEqual "explode empty, "
   (explode "")
   [])

test2 = testCase
  (assertEqual "Into chars, "
   (explode "abc")
   ['a','b','c'])
test3 = testCase([\n'] @=? explode "\n")
tests = TestList [test1, test2, test3]
Running tests

Main> runTestTT tests
Cases: 3  Tried: 3  Errors: 0  Failures: 0

bad = TestCase
  (assertEqual "explode reverses?, "
   (explode "ab")
   "ba")

Main> runTestTT bad
### Failure:
explode reverses?,
expected: "ab"
  but got: "ba"
Cases: 1  Tried: 1  Errors: 0  Failures: 1
Assignment #4

- You are to write a number of functions in Haskell
  - explode :: String -> [Char]
  - digitValue :: [Char] -> [Int]
  - reverse :: [Int] -> [Int]
  - pairedWithPowersOf10 :: [Int] -> [(Int, Int)]
  - pairWiseProduct :: [(Int, Int)] -> [Int]
  - sum :: [Int] -> Int
  - toInt :: String -> Int

- Write 5 functions (sum & reverse are primitives, and don’t need to be written).
- Write each function using only list operations
- Use Hunit to create 3 tests for each function
- Turn in an elegant program that communicates your solution well.