FOLDS IN HASKELL

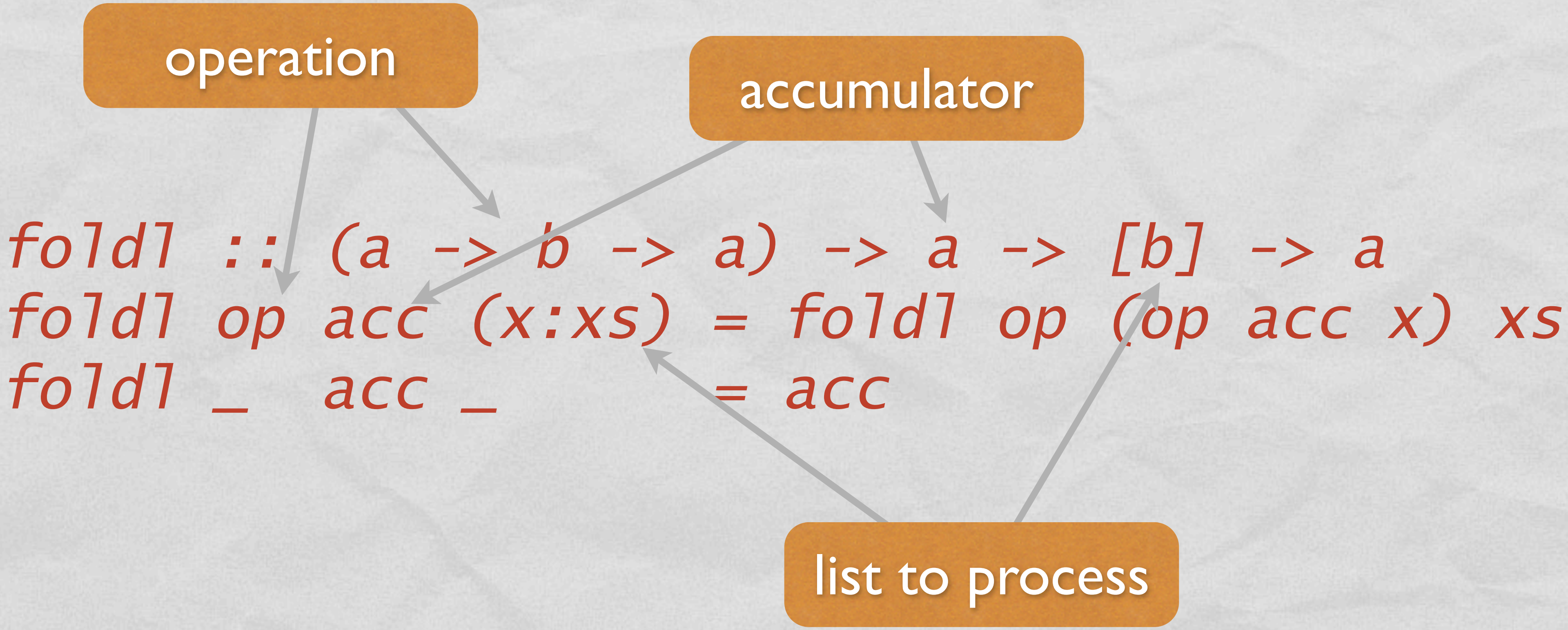
Curt Clifton
Rose-Hulman Institute of Technology

SVN Update *HaskellFoldsInClass* folder, open *fold.hs*

EXAMPLE: ADLER-32

- Concatenates two 16-bit checksums
  - First is the sum of all the input bytes, plus 1
  - Second is the running total of the intermediate values of the first checksum
  - Both are modulo 65521

LEFT FOLD



operation

accumulator

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl op acc (x:xs) = foldl op (op acc x) xs
foldl _  acc _      = acc
```

list to process

Q1

ADLER-32 WITH FOLDL

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl op acc (x:xs) = foldl op (op acc x) xs
foldl _  acc _      = acc
```

```
adler32_v3 :: String -> Int
adler32_v3 xs = let (chSum1,chSum2) = foldl procByte (1,0) xs
                in (chSum2 `shiftL` 16) .|. chSum1
    where procByte (chSum1,chSum2) x =
            let chSum1' = (chSum1 + (ord x .&. 0xff))
            in (chSum1' `mod` base, (chSum1' + chSum2) `mod` base)
```

RIGHT FOLD

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op acc (x:xs) = op x (foldr op acc xs)
foldr _  acc []     = acc
```

Consider: *foldr (+) 0 [1..3]*

Input: *1 : (2 : (3 : []))*
Result: *1 + (2 + (3 + 0))*

Q3

THE POWER OF FOLDR

```
-- filter using foldr
myFilter :: (c -> Bool) -> [c] -> [c]
myFilter pred xs = foldr op [] xs
    where op x acc | pred x    = x : acc
                   | otherwise = acc

-- map using foldr
myMap :: (c -> d) -> [c] -> [d]
myMap f xs = foldr op [] xs
    where op x acc = (f x) : acc

-- append using foldr
append :: [c] -> [c] -> [c]
append xs ys = foldr (:) ys xs
```

Try to match types here to types in foldr’s signature

FOLDL VS. FOLDR

- any :: (a -> Bool) -> [a] -> Bool
- any odd [2,4,6] == False
- any odd [2,5,6] == True
- any odd [] == False

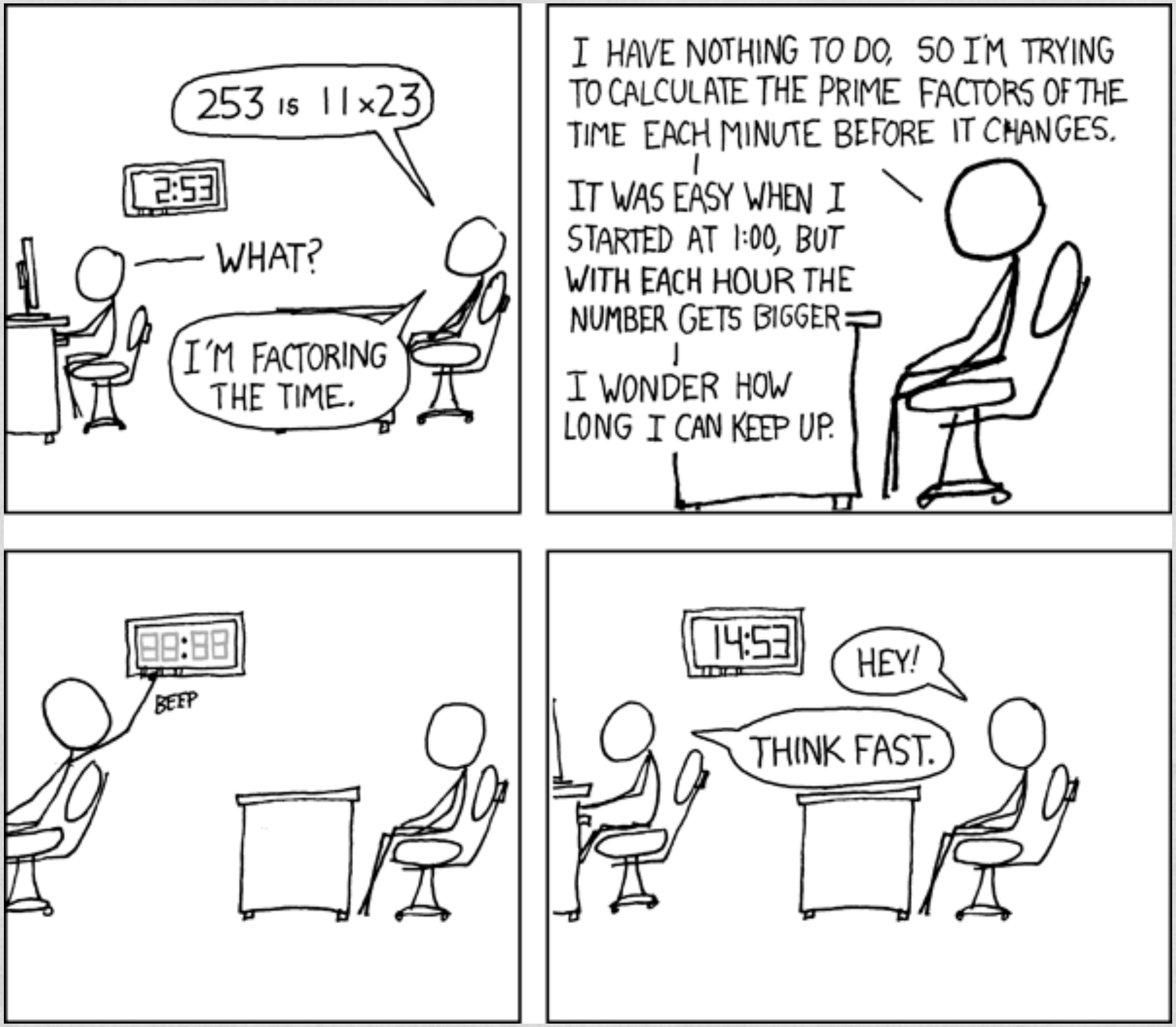
```
any p xs = foldr op False xs
    where op x acc | p x = True
                   | otherwise = acc
any p xs = foldl op False xs
    where op acc x | p x = True
                   | otherwise = acc
```

SPACE LEAKS

- *foldl* generates big thunks
  - take lots of space to store and evaluate
  - can use *foldl'* for strict (non-lazy) version
- *foldr may* generate big thunks…
  - …but most applications don't if they leave right-side unchanged or ignore it

Q4

FACTORING THE TIME


253 IS 11×23
2:53
WHAT?
I'M FACTORING THE TIME.
I HAVE NOTHING TO DO, SO I'M TRYING TO CALCULATE THE PRIME FACTORS OF THE TIME EACH MINUTE BEFORE IT CHANGES.
IT WAS EASY WHEN I STARTED AT 1:00, BUT WITH EACH HOUR THE NUMBER GETS BIGGER.
I WONDER HOW LONG I CAN KEEP UP.
88:88
BEEP
14:53
HEY!
THINK FAST.

I occasionally do this with mile markers on the highway.

MISCELLANY

LAMBDAS

- Problem: defining simple function arguments to library functions can require verbose helpers
- Solution: lambdas
- Example expression: **(\x y -> abs(x-y) < 5)**
- Example use: **nubBy (\x y -> abs(x-y) < 5) [1..20]**

CURRIED FUNCTIONS

- Curried functions take a single argument and return functions taking subsequent arguments
- All functions automatically curried
- Allows "partial application"

Mmm, curry

CURRIED FUNCTIONS

- Curried functions take a single argument and return functions taking subsequent arguments
- All functions automatically curried
- Allows “partial application”

```
ghci> :module +Data.Char
ghci> :t dropWhile
dropWhile :: (a -> Bool) -> [a] -> [a]
ghci> :t dropWhile isSpace
dropWhile isSpace :: [Char] -> [Char]
ghci> let lTrim = dropWhile isSpace
ghci> let m = ["dog", "  cat", "   raptor  "]
ghci> map lTrim m
["dog","cat","raptor  "]
```

SECTIONS

- Can partially apply infix operators on either side
- E.g., (==2), (>2), (2*)

```
ghci> :t (2^)
(2^) :: (Num t, Integral b) => b -> t
ghci> :t (^2)
(^2) :: (Num a) => a -> a
ghci> map (^2) [1..4]
[1,4,9,16]
ghci> map (2^) [1..4]
[2,4,8,16]
```

AS-PATTERNS

- Problem: sometimes we need to pattern match, but want to refer to the whole value in the definition

- Solution: as-patterns

- Example: **xs@(_:_)**, matches non-empty list, binds **xs** to whole list

- Application:
```
sufs xs@(_:xs’) = xs : sufs xs’
sufs _ = []
```

```
sufs “whale” == [“whale”, “hale”, “ale”, “le”, “e”]
```

DOT NOTATION

- Problem: often we can compose library functions, but nested parens get ugly

  -
    ```
    capCount s = length (filter p (words s))
                 where p w = isUpper (head w)
    ```

- Solution: dot notation composes functions right-to-left

  -
    ```
    capCount = length . filter (isUpper . head) . words
    ```

HASKELL STYLE GUIDELINES

- *map*, *filter*, *take*, and company are your friends
- Prefer compositions of library functions over folds
- Prefer folds over custom tail recursion
- Use recursion when you must
- Avoid anonymous lambdas