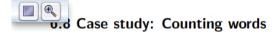
Script generated by TTT

Title: Nipkow: Info2 (21.11.2014)

Date: Fri Nov 21 07:30:32 GMT 2014

Duration: 86:39 min

Pages: 108



Input: A string, e.g. "never say never again"

Output: A string listing the words in alphabetical order, together with their frequency,

e.g. "again: 1\nnever: 2\nsay: 1\n"



Input: A string, e.g. "never say never again"

Case study: Counting words

Input: A string, e.g. "never say never again"

Output: A string listing the words in alphabetical order, together with their frequency,

e.g. "again: 1\nnever: 2\nsay: 1\n"

Function putStr yields

again: 1 never: 2 say: 1



v.d Case study: Counting words

Input: A string, e.g. "never say never again"

 $\begin{tabular}{ll} \textbf{Output:} A string listing the words in alphabetical order, together \\ \end{tabular}$

with their frequency,

e.g. "again: 1\nnever: 2\nsay: 1\n"

Function putStr yields

again: 1 never: 2 say: 1

Design principle:

Solve problem in a sequence of small steps transforming the input gradually into the output



v.d Case study: Counting words

Input: A string, e.g. "never say never again"

 $\begin{tabular}{ll} \textbf{Output:} A string listing the words in alphabetical order, together \\ \end{tabular}$

with their frequency,

e.g. "again: 1\nnever: 2\nsay: 1\n"

Function putStr yields

again: 1 never: 2 say: 1

Design principle:

Solve problem in a sequence of small steps transforming the input gradually into the output

Unix pipes!



Step 1: Break input into words

"never say never again"



["never", "say", "never", "again"]



Step 1: Break input into words

"never say never again"

["never", "say", "never", "again"]

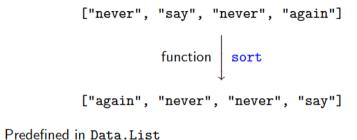
Predefined in Prelude



Step 2: Sort words

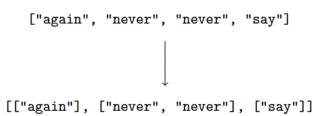


Step 2: Sort words





Step 3: Group equal words together





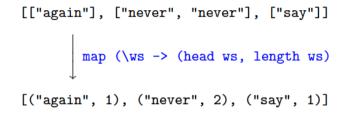
Step 3: Group equal words together



Step 4: Count each group



Step 4: Count each group





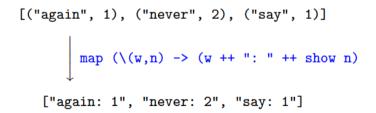
Step 5: Format each group



Step 5: Format each group

```
[("again", 1), ("never", 2), ("say", 1)]

["again: 1", "never: 2", "say: 1"]
```





Step 6: Combine the lines

Step 6: Combine the lines

Predefined in Prelude



The solution



The solution

```
countWords :: String -> String
countWords =
  unlines
  . map (\(w,n) -> w ++ ": " ++ show n)
  . map (\ws -> (head ws, length ws))
  . group
  . sort
  . words
```

```
countWords :: String -> String
countWords =
  unlines
  . map (\(w,n) -> w ++ ": " ++ show n)
  . map (\ws -> (head ws, length ws))
  . group
  . sort
  . words
```



Merging maps



Merging maps

Can we merge two consecutive maps?

```
map f . map g =
```

Can we merge two consecutive maps?

```
map f . map g = map (f.g)
```



Merging maps



The optimized solution

Can we merge two consecutive maps?

```
map f . map g = map (f.g)
```

```
countWords :: String -> String
countWords =
  unlines
  . map (\ws -> head ws ++ ": " ++ show(length ws))
  . group
  . sort
  . words
```



Proving map f . map g = map (f.g)

First we prove (why?)

$$map f (map g xs) = map (f.g) xs$$



Proving map f . map g = map (f.g)

First we prove (why?)

$$map f (map g xs) = map (f.g) xs$$

by induction on xs:

- Base case:
- map f (map g []) = []
 map (f.g) [] = []
- Induction step:

- = f (g x) : map f (map g xs)
- = f(g x) : map(f.g) xs -- by IH
- map (f.g) (x:xs)
- = f(g x) : map(f.g) xs



Proving map f . map g = map (f.g)

First we prove (why?)

$$map f (map g xs) = map (f.g) xs$$

by induction on xs:

- Base case:
 - map f (map g []) = []
 map (f.g) [] = []
- Induction step:

- = f (g x) : map f (map g xs)
- = f(g x) : map(f.g) xs -- by IH
- map (f.g) (x:xs)
- = f(g x) : map(f.g) xs

$$\implies$$
 (map f . map g) xs = map f (map g xs) = map (f.g) xs



Proving map f . map g = map (f.g)

First we prove (why?)

$$map f (map g xs) = map (f.g) xs$$

by induction on xs:

Base case:

Induction step:

- = f(g x) : map f(map g xs)
- = f(g x) : map(f.g) xs -- by IH
- map (f.g) (x:xs)
- = f(g x) : map(f.g) xs

$$\implies$$
 (map f . map g) xs = map f (map g xs) = map (f.g) xs

$$\implies$$
 (map f . map g) = map (f.g) by extensionality





7. Type Classes

Remember: type classes enable overloading



Remember: type classes enable overloading

Example

```
elem ::
elem x = any (== x)
```



Remember: type classes enable overloading

Example

```
elem :: Eq a => a -> [a] -> Bool
elem x = any (== x)
where Eq is the class of all types with ==
```



In general:

Type classes are collections of types that implement some fixed set of functions



In general:

Type classes are collections of types that implement some fixed set of functions

Haskell type classes are analogous to Java interfaces: a set of function names with their types



In general:

Type classes are collections of types that implement some fixed set of functions

Haskell type classes are analogous to Java interfaces: a set of function names with their types

Example

```
class Eq a where
  (==) :: a -> a -> Bool
```



In general:

Type classes are collections of types that implement some fixed set of functions

Haskell type classes are analogous to Java interfaces: a set of function names with their types

Example

```
class Eq a where
  (==) :: a -> a -> Bool
```

Note: the type of (==) outside the class context is Eq a => a -> a -> Bool



The general form of a class declaration:

```
class C a where
  f1 :: T1
    ...
  fn :: Tn
```



The general form of a class declaration:

```
class C a where
  f1 :: T1
    ...
  fn :: Tn
```

where the Ti may involve the type variable a

Type classes support generic programming:



The general form of a class declaration:

```
class C a where
  f1 :: T1
    ...
  fn :: Tn
```

where the Ti may involve the type variable \underline{a}

Type classes support generic programming: Code that works not just for one type but for a whole class of types,



Instance

A type T is an instance of a class C if T supports all the functions of C.



The general form of a class declaration:

class C a where f1 :: T1

fn :: Tn

where the Ti may involve the type variable a

Type classes support generic programming: Code that works not just for one type but for a whole class of types, all types that implement the functions of the class.



Instance

A type T is an instance of a class C if T supports all the functions of C.



Instance

A type T is an instance of a class C if T supports all the functions of C. Then we write C T.



Instance

A type T is an instance of a class C if T supports all the functions of C. Then we write C T.

Example

Type Int is an instance of class Eq, i.e., Eq Int



Instance



Instance

A type T is an instance of a class C if T supports all the functions of C. Then we write C T.

Example

```
Type Int is an instance of class Eq, i.e., Eq Int
Therefore elem :: Int -> [Int] -> Bool
```

A type T is an instance of a class C if T supports all the functions of C. Then we write C T.

Example

```
Type Int is an instance of class Eq, i.e., Eq Int
Therefore elem :: Int -> [Int] -> Bool
```

Warning Terminology clash:



Instance



Instance

A type T is an instance of a class C if T supports all the functions of C. Then we write C T.

Example

```
Type Int is an instance of class Eq, i.e., Eq Int
Therefore elem :: Int -> [Int] -> Bool
```

Warning Terminology clash:

Type T_1 is an *instance* of type T_2

A type T is an instance of a class C if T supports all the functions of C. Then we write C T.

Example

```
Type Int is an instance of class Eq, i.e., Eq Int
Therefore elem :: Int -> [Int] -> Bool
```

Warning Terminology clash:

```
Type T_1 is an instance of type T_2 if T_1 is the result of replacing type variables in T_2.
```



Instance



instance

A type T is an instance of a class C if T supports all the functions of C. Then we write C T.

Example

```
Type Int is an instance of class Eq, i.e., Eq Int
Therefore elem :: Int -> [Int] -> Bool
```

Warning Terminology clash:

```
Type T_1 is an instance of type T_2 if T_1 is the result of replacing type variables in T_2. For example (Bool, Int) is an instance of (a,b).
```



The instance statement makes a type an instance of a class.



instance



instance

The instance statement makes a type an instance of a class.

Example

The instance statement makes a type an instance of a class.

Example



Instances can be constrained:

Example



Instances can be constrained:

Example

Possibly with multiple constraints:

Example

instance (Eq a, Eq b) => Eq (a,b) where
$$(x1,y1) == (x2,y2) = x1 == x2 && y1 == y2$$



The general form of the instance statement:

```
instance (context) => C T where
  definitions
```



The general form of the instance statement:

```
instance (context) => C T where
  definitions
```

```
T is a type context is a list of assumptions C_i T_i definitions are definitions of the functions of class C
```



Subclasses



Subclasses

Example

```
class Eq a => Ord a where
  (<=), (<) :: a -> a -> Bool
```

Example

```
class Eq a => Ord a where
  (<=), (<) :: a -> a -> Bool
```

Class Ord inherits all the operations of class Eq



Subclasses

Subclasses

Example

```
class Eq a => Ord a where
  (<=), (<) :: a -> a -> Bool
```

Class Ord inherits all the operations of class Eq

Because Bool is already an instance of Eq, we can now make it an instance of Ord:

Example

```
class Eq a => Ord a where
  (<=), (<) :: a -> a -> Bool
```

Class Ord inherits all the operations of class Eq

Because Bool is already an instance of Eq, we can now make it an instance of Ord:

```
instance Ord Bool where
b1 <= b2 = not b1 || b2
b1 < b2 = b1 <= b2 && not(b1 == b2)</pre>
```



From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```



From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default definition:
  x /= y = not(x==y)
```



From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default definition:
  x /= y = not(x==y)

class Eq a => Ord a where
  (<=), (<), (>=), (>) :: a -> a -> Bool
```



From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default definition:
  x /= y = not(x==y)

class Eq a => Ord a where
  (<=), (<), (>=), (>) :: a -> a -> Bool
  -- default definitions:
  x < y = x <= y && x /= y
  x > y = y < x
  x >= y = y <= x</pre>
```



From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default definition:
  x /= y = not(x==y)

class Eq a => Ord a where
  (<=), (<), (>=), (>) :: a -> a -> Bool
  -- default definitions:
  x < y = x <= y && x /= y
  x > y = y < x
  x >= y = y <= x</pre>
class Show a where
  show :: a -> String
```



From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default definition:
  x /= y = not(x==y)

class Eq a => Ord a where
  (<=), (<), (>=), (>) :: a -> a -> Bool
  -- default definitions:
  x < y = x <= y && x /= y
  x > y = y < x
  x >= y = y <= x</pre>
class Show a where
  show :: a -> String
```







So far: no really new types, just compositions of existing types

Example: type String = [Char]

Now: data defines new types



So far: no really new types,

just compositions of existing types

Example: type String = [Char]

Now: data defines new types

Introduction by example: From enumerated types



8.1 data by example



Bool

From the Prelude:

```
data Bool = False | True
```

(

Bool

From the Prelude:

```
data Bool = False | True
```

not :: Bool -> Bool
not False = True
not True = False

Bool

deriving

From the Prelude:

```
data Bool = False | True

not :: Bool -> Bool
not False = True
not True = False

(&&) :: Bool -> Bool -> Bool
False && q = False
True && q = q
```



Bool

deriving

From the Prelude:

```
data Bool = False | True

not :: Bool -> Bool

not False = True

not True = False

(&&) :: Bool -> Bool -> Bool

False && q = False

True && q = q

(||) :: Bool -> Bool -> Bool

False || q = q

True || q = True
```



deriving

deriving

show False = "False"

Better: let Haskell write the code for you:



deriving



instance Eq Bool where

True == True = True
False == False = True
== = False

instance Show Bool where
 show True = "True"

Better: let Haskell write the code for you:

 $\hbox{\tt deriving supports many more classes: Ord, Read, } \ldots$

Warning

Do not forget to make your data types instances of Show

Otherwise Haskell cannot even print values of your type



Warning

Do not forget to make your data types instances of Show

Otherwise Haskell cannot even print values of your type

Warning

QuickCheck does not automatically work for data types



Warning

Do not forget to make your data types instances of Show

Otherwise Haskell cannot even print values of your type

Warning

QuickCheck does not automatically work for data types

You have to write your own test data generator. Later.



Season



Shape

type Radius = Float
type Width = Float
type Height = Float



Season

Shape

next Autumn = Winter
next Winter = Spring

```
type Radius = Float
type Width = Float
type Height = Float
```

Shape



Shape



type Radius = Float

type Width = Float

type Height = Float

Shape

Shape

```
data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
Some values of type Shape: Circle 1.0
                         Rect 0.9 1.1
                        Circle (-2.0)
```

```
type Radius = Float
type Width = Float
type Height = Float
data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
Some values of type Shape: Circle 1.0
                        Rect 0.9 1.1
                        Circle (-2.0)
```

area :: Shape -> Float



Shape

```
E
```

Shape

```
type Radius = Float
type Width = Float
type Height = Float
data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
Some values of type Shape: Circle 1.0
                        Rect 0.9 1.1
                        Circle (-2.0)
area :: Shape -> Float
area (Circle r) = pi * r^2
```

```
type Radius = Float
type Width = Float
type Height = Float
data Shape = Circle Radius | Rect Width Height
            deriving (Eq, Show)
Some values of type Shape: Circle 1.0
                        Rect 0.9 1.1
                       Circle (-2.0)
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect w h) = w * h
```



Maybe

Maybe

From the Prelude:

Some values of type Maybe: Nothing :: Maybe a

From the Prelude:

Some values of type Maybe: Nothing :: Maybe a

Just True :: Maybe Bool

Just "?" ::



Maybe

Maybe

From the Prelude:

Some values of type Maybe: Nothing :: Maybe a

Just True :: Maybe Bool
Just "?" :: Maybe String

From the Prelude:

Some values of type Maybe: Nothing :: Maybe a

Just True :: Maybe Bool
Just "?" :: Maybe String

lookup :: Eq a \Rightarrow a \Rightarrow [(a,b)] \Rightarrow Maybe b



Maybe

Maybe

From the Prelude:

Some values of type Maybe: Nothing :: Maybe a

Just True :: Maybe Bool
Just "?" :: Maybe String

lookup :: Eq a \Rightarrow a \Rightarrow [(a,b)] \Rightarrow Maybe b lookup key [] \Rightarrow

From the Prelude:

Some values of type Maybe: Nothing :: Maybe a

Just True :: Maybe Bool
Just "?" :: Maybe String

lookup :: Eq a => a -> [(a,b)] -> Maybe b

lookup key [] = Nothing



Maybe

Maybe

From the Prelude:

Some values of type Maybe: Nothing :: Maybe a

Just True :: Maybe Bool
Just "?" :: Maybe String

From the Prelude:

Some values of type Maybe: Nothing :: Maybe a

Just True :: Maybe Bool
Just "?" :: Maybe String

lookup :: Eq a \Rightarrow a \Rightarrow [(a,b)] \Rightarrow Maybe b

lookup key [] = Nothing
lookup key ((x,y):xys)
 | key == x = Just y
 | otherwise =



Maybe

Nat

From the Prelude:

Some values of type Maybe: Nothing :: Maybe a

Just True :: Maybe Bool
Just "?" :: Maybe String

```
lookup :: Eq a \Rightarrow a \Rightarrow [(a,b)] \Rightarrow Maybe b
```

| otherwise = lookup key xys

```
Natural numbers:
```



Nat

Natural numbers:

Some values of type Nat: Zero

Suc Zero

Suc (Suc Zero)

Nat

Natural numbers:

Some values of type Nat: Zero

Suc Zero

Suc (Suc Zero)

:

add :: Nat -> Nat -> Nat

Nat

Natural numbers:



Nat

Natural numbers:



Nat

Natural numbers:

From the Prelude:

```
data [a] = [] | (:) a [a]
```

Lists



Lists

From the Prelude: