Title: Nipkow: Info2 (21.11.2014)
Date: Fri Nov 21 07:30:32 GMT 2014
Duration: 86:39 min
Pages: 108

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"
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 putStrLn 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"

\[
\text{function} \rightarrow \text{words}
\]

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

Predefined in Prelude
Step 2: Sort words

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

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

Predefined in Data.List

Step 3: Group equal words together

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

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

Predefined in Data.List
Step 4: Count each group

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

\[
\downarrow
\]

\[["again", 1], ["never", 2], ["say", 1]\]

Step 5: Format each group

\[["again", 1], ["never", 2], ["say", 1]\]

\[
\downarrow
\]

\[\text{map } \langle w, n \rangle \to (w \times " \times \text{show } n)\]

\[["again: 1", "never: 2", "say: 1"]\]
Step 6: Combine the lines

"again: 1", "never: 2", "say: 1"

\[\text{unlines}\]

"again: 1\nnever: 2\nsay: 1\n"

Predefined in Prelude

The solution

countWords :: String -> String
countWords =
  unlines
  . map (\(w,n) -> w ++ ": " ++ show n)
  . map (\ws -> (head ws, length ws))
  . group
  . sort
  . words
Can we merge two consecutive `maps`?

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

Can we merge two consecutive `maps`?

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

**The optimized solution**

```haskell
countWords :: String -> String
countWords =
  unlines
  . map (\ws -> head ws ++ ": " ++ show(length ws))
  . group
  . sort
  . words
```
Proving \( \text{map } f \cdot \text{map } g = \text{map } (f \cdot g) \)

First we prove (why?)

\[
\text{map } f \ (\text{map } g \ x) = \text{map } (f \cdot g) \ x
\]

by induction on \( x \):

- Base case:
  \[
  \text{map } f \ (\text{map } g \ [] ) = [] \\
  \text{map } (f \cdot g) \ [] = []
  \]

- Induction step:
  \[
  \text{map } f \ (\text{map } g \ (x:xs)) \\
  = f \ (g \ x) : \text{map } f \ (\text{map } g \ xs) \\
  = f \ (g \ x) : \text{map } (f \cdot g) \ xs \quad \text{-- by IH} \\
  \text{map } (f \cdot g) \ (x:xs) \\
  = f \ (g \ x) : \text{map } (f \cdot g) \ x
  \]

\[
\Rightarrow (\text{map } f \cdot \text{map } g) \ x = \text{map } f \ (\text{map } g \ x) = \text{map } (f \cdot g) \ x
\]
7. Type Classes

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

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

where the $T_i$ 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.*

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:

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

where the $T_i$ 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.*

---

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 $\text{Int}$ is an instance of class $\text{Eq}$, i.e., $\text{Eq Int}$
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 $\text{Int}$ is an instance of class $\text{Eq}$, i.e., $\text{Eq\ Int}$
Therefore $\text{elem :: Int -> [Int] -> Bool}$

Warning Terminology clash:
Type $T_1$ is an instance of type $T_2$
if $T_1$ is the result of replacing type variables in $T_2$. 

Example
Type $\text{Int}$ is an instance of class $\text{Eq}$, i.e., $\text{Eq\ Int}$
Therefore $\text{elem :: Int -> [Int] -> Bool}$

Warning Terminology clash:
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 $\text{Int}$ is an instance of class $\text{Eq}$, i.e., $\text{Eq \ Int}$
Therefore $\text{elem :: Int} \to [\text{Int}] \to \text{Bool}$

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 $(\text{Bool, Int})$ is an instance of $(a, b)$.

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

Example
instance $\text{Eq \ Bool}$ where
  $\text{True} \ == \ True$ = True
  $\text{False} \ == \ False$ = True
  _   == _   = False

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

Example
instance $\text{Eq \ Bool}$ where
  $\text{True} \ == \ True$ = True
  $\text{False} \ == \ False$ = True
  _   == _   = False
Instances can be constrained:

Example

```
instance Eq a => Eq [a] where
  []   == []   = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _   == _    = False
```

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
```

$T$ is a type

`context` is a list of assumptions $C_i, T_i$

`definitions` are definitions of the functions of class $C$
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)
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default definition:
  x /= y = not(x==y)

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

class Show a where
  show :: a -> String

8. Algebraic data Types

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

From the Prelude:

data Bool = False | True

From the Prelude:

data Bool = False | True

not :: Bool -> Bool
not False = True
not True = False
From the Prelude:

```haskell
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

instance Eq Bool where
    True == True  = True
    False == False = True
    _ == _ = False
```
instance Eq Bool where
  True == True  = True
  False == False = True
  _ == _        = False

instance Show Bool where
  show True    = "True"
  show False   = "False"

Better: let Haskell write the code for you:

data Bool = False | True
  deriving (Eq, Show)

Warning
Do not forget to make your data types instances of Show

Otherwise Haskell cannot even print values of your type

deriving supports many more classes: Ord, Read, …
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

data Season = Spring | Summer | Autumn | Winter
  deriving (Eq, Show)

type Radius = Float
type Width = Float
type Height = Float
data Season = Spring | Summer | Autumn | Winter
deriving (Eq, Show)

next :: Season -> Season
next Spring = Summer
next Summer = Autumn
next Autumn = Winter
next Winter = Spring

data Shape = Circle Radius | Rect Width Height
deriving (Eq, Show)

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
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
From the Prelude:

```haskell
data Maybe a = Nothing | Just a
deriving (Eq, Show)
```

Some values of type Maybe:

- Nothing :: Maybe a
- Just True :: Maybe Bool
- Just "?" :: Maybe String

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```
From the Prelude:

```haskell
data Maybe a = Nothing | Just a
  deriving (Eq, Show)

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
lookup key [(x,y):ys] | key == x = Just y
  | otherwise = lookup key ys
```
**Maybe**

From the Prelude:

```haskell
data Maybe a = Nothing | Just a
  deriving (Eq, Show)
```

Some values of type `Maybe`:

```haskell
Nothing :: Maybe a
Just True :: Maybe Bool
Just "?" :: Maybe String
```

```haskell
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup key [] = Nothing
lookup key ((x,y):ys)
  | key == x = Just y
  | otherwise = lookup key ys
```

**Nat**

Natural numbers:

```haskell
data Nat = Zero | Suc Nat
  deriving (Eq, Show)
```

Some values of type `Nat`:

```haskell
Zero
Suc Zero
Suc (Suc Zero)
```

```haskell
add :: Nat -> Nat -> Nat
```
**Natural numbers:**

```haskell
data Nat = Zero | Suc Nat
  deriving (Eq, Show)
```

Some values of type Nat:

- Zero
- Suc Zero
- Suc (Suc Zero)

```
add :: Nat -> Nat -> Nat
add Zero n = n
add (Suc m) n = Suc (add m n)
```

---

**Lists**

From the Prelude:

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

From the Prelude:

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