Haskell

Higher Order Functions

Why Are They Useful?

- **<u>***</u>** Common programming idioms can be encoded as functions within the language itself.
- **<u># Domain specific languages</u>** can be defined as collections of higher-order functions.
- **<u>**Algebraic properties</u>** of higher-order functions can be used to reason about programs.

Introduction

A function is called <u>higher-order</u> if it takes a function as an argument or returns a function as a result.

twice ::
$$(a \rightarrow a) \rightarrow a \rightarrow a$$

twice f x = f (f x)

twice is higher-order because it takes a function as its first argument.

The Map Function

The higher-order library function called \underline{map} applies a function to every element of a list.

$$\texttt{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

Examples

$$sum1 x = x+1$$

The map function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = [f x | x \leftarrow xs]
```

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

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

The Filter Function

The higher-order library function <u>filter</u> selects every element from a list that satisfies a predicate.

```
\texttt{filter} \, :: \, \, (\texttt{a} \, \rightarrow \, \texttt{Bool}) \, \rightarrow \, [\texttt{a}] \, \rightarrow \, [\texttt{a}]
```

For example:

```
> filter even [1..10]
[2,4,6,8,10]
```

Esercizio

- Si consideri una matrice di interi data da una lista di liste
- Si scriva la funzione matQuad che eleva al quadrato ogni elemento della matrice:

6

filter can be defined using a list comprehension:

```
filter p xs = [x \mid x \leftarrow xs, p x]
```

Alternatively, it can be defined using recursion:

The Foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

$$f [] = v$$

$$f (x:xs) = x \oplus f xs$$

f maps the empty list to some value v, and any non-empty list to some function ⊕ applied to its head and f of its tail.

The higher-order library function <u>foldr</u> (fold right) encapsulates this simple pattern of recursion, with

For example:

```
sum = foldr (+) 0
product = foldr (*) 1
or = foldr (||) False
and = foldr (&&) True
```

the function \oplus and the value v as arguments.

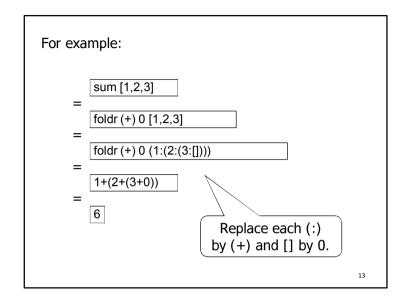
11

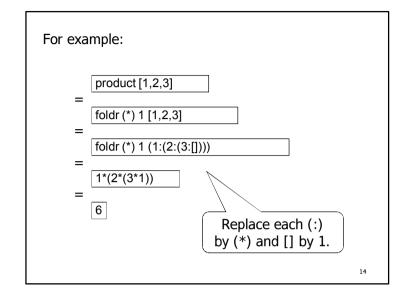
For example: sum [] = 0V = 0 \oplus = + sum (x:xs) = x + sum xsproduct [] = 1 V = 1product (x:xs) = x * product xs ⊕ = * v = True and [] = True *&*3 = ⊕ and (x:xs) = x && and xs

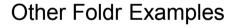
Foldr itself can be defined using recursion:

```
foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

However, it is best to think of foldr non-recursively, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.







Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the length function:

```
length :: [a] \rightarrow Int
length [] = 0
length (_:xs) = 1 + length xs
```

15

Why Is Foldr Useful?

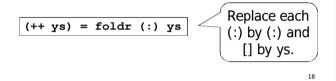
- **#**Some recursive functions on lists, such as sum, are <u>simpler</u> to define using foldr.
- #Properties of functions defined using foldr can be proved using algebraic properties of foldr, such as fusion and the banana split rule.
- **%**Advanced program <u>optimisations</u> can be simpler if foldr is used in place of explicit recursion.

19

Hence, we have:

reverse = foldr (
$$\lambda x xs \rightarrow xs ++ [x]$$
) []

Finally, we note that the append function (++) has a particularly compact definition using foldr:



Left Fold

• There is also a left fold:

The difference is that in **fold1** the value is accumulated "on the left".

foldr
$$\oplus$$
 a $[\mathbf{x}_1, ..., \mathbf{x}_n] = \mathbf{x}_1 \oplus (\mathbf{x}_2 \oplus (... (\mathbf{x}_n \oplus \mathbf{a})))$
foldl \oplus a $[\mathbf{x}_1, ..., \mathbf{x}_n] = (((\mathbf{a} \oplus \mathbf{x}_1) \oplus \mathbf{x}_2) ... \oplus \mathbf{x}_n)$

Exercise

- Count the number of vowels in a string using folds
- Useful function: elem x xs checks if x is in list xs

21

Function composition

• It composes functions in a readable manner

٧S

(f.g.h.k)(x)

· Note that usually functions associate to the left

>not even 2 ERROR

>(not.even) 2 False

23

Function composition

The library function (.) returns the <u>composition</u> of two functions as a single function.

For example:

odd :: Int \rightarrow Bool odd = not . even

22

Example

• Given a list of numbers, create a list of all negated absolute values using map

• Useful functions: abs and negate

The library function <u>all</u> decides if every element of a list satisfies a given predicate.

```
all :: (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool
all p xs = and [p x | x \leftarrow xs]
```

For example:

```
> all even [2,4,6,8,10]
True
```

25

27

The library function <u>takeWhile</u> selects elements from a list while a predicate holds of all the elements.

```
takeWhile :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]
takeWhile p [] = []
takeWhile p (x:xs)

| p x = x : takeWhile p xs
| otherwise = []
```

For example:

```
> takeWhile (/= ' ') "abc def"
"abc"
```

Dually, the library function <u>any</u> decides if at least one element of a list satisfies a predicate.

```
any :: (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool
any p xs = or [p x \mid x \leftarrow xs]
```

For example:

```
> any (== ' ') "abc def"
True
```

26

Dually, the function <u>dropWhile</u> removes elements while a predicate holds of all the elements.

```
dropWhile :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]

dropWhile p [] = []

dropWhile p (x:xs)

| p x = dropWhile p xs

| otherwise = x:xs
```

For example:

```
> dropWhile (== ' ') " abc"
"abc"
```

Exercises

- Express the comprehension
 [f x | x ← xs, p x]
 using the functions map and filter.
- Write a function zipWith'

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
```

with the following behavior

29

Create a password strength checker

- Create a password strength checker using higher-order functions
- A strong password has
 - at least 15 characters
 - uppercase letters
 - lowercase letters
 - numbers

```
Prelude> :t strong
strong :: String -> Bool
Prelude> strong "sup3rL33Tpassw0rd"
True
```

Esercizio (estratto dal compito 29 giu 2016)

- si scriva una funzione di ordine superiore maxf :: Ord a => (t -> a) -> [t] -> t
- che prende come parametri una funzione f e una lista xs e fornisce l'elemento x della lista xs che massimizza la funzione f (ossia il valore x per cui f(x) è massimo).

30

 The function remdups removes adjacent duplicates from a list. For example,

 Define remdups using foldr. Give another definition using foldl.

Un po' più difficili ...

• Redefine map f and filter p using foldr.

These slides were adapted from the material of the book

Graham Hutton, Programming in Haskell, Cambridge University Press, 2nd edition, 2016

