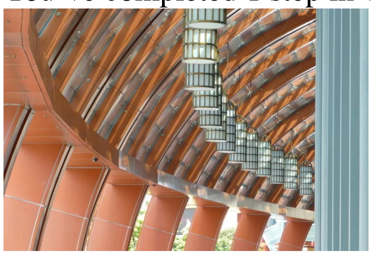
To do

- To do
- Activity
- Progress

3.2
You've completed 1 step in week 3



Recursive Functions on Lists

10 comments

Computing with lists

- There are two approaches to working with lists:
    - Write functions to do what you want, using recursive definitions that traverse the list structure.
    - Write combinations of the standard list processing functions.
- The second approach is preferred, but the standard list processing functions do need to be defined, and those definitions use the first approach (recursive definitions).
- We'll cover both methods.

Recursion on lists

- A list is built from the empty list [] and the function $cons\ :: \ a \rightarrow [a] \rightarrow [a]$. In Haskell, the function $cons$ is actually written as the operator (:) , in other words *:* is pronounced as `cons`.
- Every list must be either
    - [] or
    - $(x : xs)$ for some x (the head of the list) and xs (the tail)

where $(x : xs)$ is pronounced as $x\ cons\ xs$

- The recursive definition follows the structure of the data:
    - Base case of the recursion is [].
    - Recursion (or induction) case is $(x : xs)$.

Some examples of recursion on lists

Recursive definition of *length*

The length of a list can be computed recursively as follows:

```
length :: [a] -> Int              -- function type
length [] = 0                     -- base case
length (x:xs) = 1 + length xs     -- recursion case
```

Recursive definition of *filter*

- *filter* is given a *predicate* (a function that gives a Boolean result) and a list, and returns a list of the elements that satisfy the predicate.

```
filter :: (a->Bool) -> [a] -> [a]
```

Filtering is useful for the "generate and test" programming paradigm.

```
filter (<5) [3,9,2,12,6,4] -- > [3,2,4]
```

The library definition for `filter` is shown below. This relies on guards, which we will investigate properly next week.

```
filter :: (a -> Bool) -> [a] -> [a]
filter pred []    = []
filter pred (x:xs)
  | pred x          = x : filter pred xs
  | otherwise       = filter pred xs
```

Computations over lists

- Many computations that would be for/while loops in an imperative language are naturally expressed as list computations in a functional language.

- There are some common cases:

  - Perform a computation on each element of a list: *map*
  - Iterate over a list, from left to right: $foldl$
  - Iterate over a list, from right to left: $foldr$
- It's good practice to use these three functions when applicable
- And there are some related functions that we'll see later

Function composition

- We can express a large computation by "chaining together" a sequence of functions that perform smaller computations

1. Start with an argument of type *a*
2. Apply a function $g :: a \to b$ to it, getting an intermediate result of type *b*
3. Then apply a function $f :: b \to c$ to the intermediate result, getting the final result of type *c*

- The entire computation (first g, then f) is written as $f \circ g$.
- This is traditional mathematical notation; just remember that in $f \circ g$, the functions are used in right to left order.

- Haskell uses `.` as the function composition operator

  ```
  (.) :: (b->c) -> (a->b) -> a -> c
  (f . g) x = f (g x)
  ```

Performing an operation on every element of a list: *map*

- *map* applies a function to every element of a list

  ```
  map f [x0,x1,x2] -- > [f x0, f x1, f x2]
  ```

Composition of maps

- *map* is one of the most commonly used tools in your functional toolkit

- A common style is to define a set of simple computations using map, and to compose them.

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

This theorem is frequently used, in both directions.

Recursive definition of *map*

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Folding a list (reduction)

- An iteration over a list to produce a singleton value is called a *fold*
- There are several variations: folding from the left, folding from the right, several variations having to do with “initialisation”, and some more advanced variations.
- Folds may look tricky at first, but they are extremely powerful, and they are used a lot! And they aren’t actually very complicated.

Left fold: *foldl*

- foldl is *fold from the left*
- Think of it as an iteration across a list, going left to right.
- A typical application is $foldl\ f\ z\ xs$
- The $z :: b$ is an initial value
- The $xs :: [a]$ argument is a list of values which we combine systematically using the supplied function f
- A useful intuition: think of the $z :: b$ argument as an “accumulator”.

- The function f takes the current value of the accumulator and a list element, and gives the new value of the accumulator.

  ```
  foldl :: (b->a->b) -> b -> [a] -> b
  ```

Examples of *foldl* with function notation

$$\begin{aligned}
\texttt{foldl f z []} &\rightsquigarrow & z \\
\texttt{foldl f z [x0]} &\rightsquigarrow & f\ z\ x0 \\
\texttt{foldl f z [x0, x1]} &\rightsquigarrow & f\ (f\ z\ x0)\ x1 \\
\texttt{foldl f z [x0, x1, x2]} &\rightsquigarrow & f\ (f\ (f\ z\ x0)\ x1)\ x2
\end{aligned}$$

Examples of *foldl* with infix notation

In this example, + denotes an arbitrary operator for f; it isn't supposed to mean specifically addition.

```
foldl (+) z []          -- > z
foldl (+) z [x0]        -- > z + x0
foldl (+) z [x0,x1]     -- > (z + x0) + x1
foldl (+) z [x0,x1,x2]  -- > ((z + x0) + x1) + x2
```

Recursive definition of *foldl*

```
foldl        :: (b -> a -> b) -> b -> [a] -> b
foldl f z0 xs0 = lgo z0 xs0
             where
                lgo z []     =  z
                lgo z (x:xs) = lgo (f z x) xs
```

Right fold: *foldr*

- Similar to $foldl$, but it works from right to left

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Examples of *foldr* with function notation

$$
\begin{aligned}
\texttt{foldr f z []} &\rightsquigarrow & z\\
\texttt{foldr f z [x0]} &\rightsquigarrow & f\ x0\ z\\
\texttt{foldr f z [x0, x1]} &\rightsquigarrow & f\ x0\ (f\ x1\ z)\\
\texttt{foldr f z [x0, x1, x2]} &\rightsquigarrow & f\ x0\ (f\ x1\ (f\ x2\ z))
\end{aligned}
$$

Examples of *foldr* with operator notation

```
foldr (+) z []          -- > z
foldr (+) z [x0]        -- > x0 + z
foldr (+) z [x0,x1]     -- > x0 + (x1 + z)
foldr (+) z [x0,x1,x2]  -- > x0 + (x1 + (x2 + z))
```

Recursive definition of *foldr*

```
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr k z = go
          where
            go []     = z
            go (y:ys) = y `k` go ys
```

Relationship between *foldr* and list structure

We have seen that a list `[x0,x1,x2]` can also be written as

```
    x0 :  x1 : x2 : []
```

Folding *cons* (:) over a list using the empty list [] as accumulator gives:

```
foldr (:)  [] [x0,x1,x2]
  -- >
  x0 :  x1 : x2 : []
```

This is identical to constructing the list using (:) and [] ! We can formalise this relationship as follows:

$$foldr\ cons\ []\ xs \ = \ xs$$

Some applications of folds

```
sum xs = foldr (+) 0 xs
product xs = foldr (*) 1 xs
```

We can actually “factor out” the xs that appears at the right of each side of the equation, and write:

```
sum      = foldr (+) 0
product  = foldr (*) 1
```

(This is sometimes called “point free” style because you’re programming solely with the functions; the data isn’t mentioned directly.)

10 comments
Step complete
Welcome to Week 3video
Functional Maps and Folds versus Imperative Loopsvideo

Share this article:

-
-
-
-

Contact FutureLearn for Support