Session 7

Defining functions over lists: recursion

Implementing functions: recursion

- In sessions 4 and 5, we've seen list comprehensions and a range of library functions on lists
- We can use these features to produce powerful and concise programs, but sometimes you need to write other functions
- In sessions 2 and 3, we defined functions over user-defined types using *pattern matching*
- In this session, we'll examine how several standard functions are implemented, combining *pattern matching* with a new feature: *recursion*
- *Aim:* by the end of this session, you should be able to write your own recursive functions
- In the following two sessions, we'll be exploring other applications of recursion

Lists as algebraic types

Pattern matching on tuples

Recall that user-defined data types can have multiple alternatives containing data:

```
data Shape
    = Circle Double
    | Rectangle Double Double
```

We can choose between alternatives and access the data they contain by using separate equations, each with a *pattern* as an argument:

```
area :: Shape -> Double
area (Circle r) = pi*r*r
area (Rectangle w h) = w*h
```

But what about lists?

Cons: add element to the front of a list

There is a infix operator '`:`' (pronounced "cons") that constructs a new list by adding an element at the front.

interpreter

```
1:[2,3,4]
'a':"bcd"
[]
3:[]
2:3:[]
1:2:3:[]
'h':'e':'l':'l':'o':[]
:t (:)
```

Primitive lists

There are two kinds of lists:

- `[]` is a list, the empty list.
- If x_1 is an element and $[x_2, \ldots, x_n]$ is a list, then there is another list

$$x_1 : [x_2, \ldots, x_n] = [x_1, x_2, \ldots, x_n]$$

Indeed

$$[x_1, x_2, \ldots, x_n] = x_1 : x_2 : \ldots : x_n : []$$

(because '`:`' is right associative)

Defining a function on lists

So when defining a function on lists, there will be two cases.

Testing whether a list is empty (a `Prelude` function):

Prelude

```
null :: [a] -> Bool
null [] = True
null (x:xs) = False
```

As only the first matching clause is used, the following is equivalent:

Prelude

```
null [] = True
null xs = False
```

We can also use an "anonymous" variable "_":

Prelude

```
null [] = True
null _ = False
```

More standard functions

Head and tail of a list:

Prelude
```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs
```

Note that only the '`:`' case is covered.

interpreter
```
head [1,2,3,4]
head (1:[2,3,4])
tail [1,2,3,4]
tail (1:[2,3,4])
head []
tail []
```

Recursive definitions

Revision: using other functions

If we want to write a function

```
maxOfThree :: Int -> Int -> Int -> Int
maxOfThree x y z = ???
```

We ask: are there any functions (possibly not yet existing) of **`x`**, **`y`** and **`z`** that might help?

```
maxOfThree :: Int -> Int -> Int -> Int
maxOfThree x y z = max (max x y) z
```

In top-down design, we use functions before we've written them. This is fine as long as we know what the functions do.

Using the same function (recursion)

Problem: determine the length of a list.

First step: write the type and left-hand sides:

Prelude
```
length :: [a] -> Int
length [] = ???
length (x:xs) = ???
```

Do the easy thing first:

Prelude
```
length [] = 0
```

Now ask: what would be useful? *Answer*: the length of **`xs`**!

Prelude
```
length (x:xs) = 1 + length xs
```

Why does it work?

A recursive function:

Prelude
```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Expanding an example step by step:

```
length [6,7,8]  ⇝  1 + length [7,8]
                ⇝  1 + (1 + length [8])
                ⇝  1 + (1 + (1 + length []))
                ⇝  1 + (1 + (1 + 0))
                ⇝  3
```

The key: the argument of `length` on the right-hand-side is a *smaller* list.

How to write a recursive function on lists

1. Write the type and the left-hand sides:

   Prelude
   ```
   product :: [Int] -> Int
   product [] = ???
   product (x:xs) = ???
   ```

2. Do the easy case (empty lists):

   Prelude
   ```
   product [] = 1
   ```

3. Ask: given `x`, `xs` and `product xs`, how can I construct `product (x:xs)`?

   Prelude
   ```
   product (x:xs) = x * product xs
   ```

Singleton lists

Some functions treat the case of a singleton list specially:

Prelude
```
last :: [a] -> a
last [x] = x
last (x:xs) = last xs

init :: [a] -> [a]
init [x] = []
init (x:xs) = x : init xs
```

- The pattern `[x]` matches a list with exactly one element.
- The order of the clauses matters here, because `x:xs` matches lists with one or more elements.

Functions on two lists

Zipping two lists

First, write out the type and the cases:

Prelude
```
zip :: [a] -> [b] -> [(a,b)]
zip [] [] = ???
zip [] (y:ys) = ???
zip (x:xs) [] = ???
zip (x:xs) (y:ys) = ???
```

Do the easy cases first:

Prelude
```
zip [] [] = []
zip [] (y:ys) = []
zip (x:xs) [] = []
```

Now, how to build the right-hand side of the last case?

Zipping lists, continued

We know the result will be a list with first element `(x,y)`. The tail will be another `zip`:

Prelude
```
zip :: [a] -> [b] -> [(a,b)]
zip [] [] = []
zip [] (y:ys) = []
zip (x:xs) [] = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

An equivalent version:

Prelude
```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y):zip xs ys
zip _ _ = []
```

Concatenating two lists

First write down the type:

Prelude
```
(++) :: [a] -> [a] -> [a]
```

Key insight: we only need to consider the two cases of the first argument:

Prelude
```
[] ++ ys = ???
(x:xs) ++ ys = ???
```

Do the easy case first:

Prelude
```
[] ++ ys = ys
```

In other case, the result will be a list with first element `x`:

Prelude
```
(x:xs) ++ ys = x:(xs ++ ys)
```

Patterns and guards

Combining patterns and guards

Testing whether a value is in a list:

Prelude
```
elem :: Eq a => a -> [a] -> Bool
elem x [] = False
elem x (y:ys)
  | x == y = True
  | otherwise = elem x ys
```

Repeated variables are *not* allowed:

Prelude
```
elem x (x:ys) = True        -- ILLEGAL
elem x (y:ys) = elem x ys
```

Alternative version:

Prelude
```
elem x [] = False
elem x (y:ys) = x == y || elem x ys
```

Another example: `take`

Write down the type and the cases we consider:

Prelude
```
take :: Int -> [a] -> [a]
take n [] = ???
take n (x:xs)
  | n > 0 = ???
  | otherwise = ???
```

Do the easy cases:

Prelude
```
take n [] = []
take n (x:xs)
  | n > 0 = ???
  | otherwise = []
```

Take, continued

We know the result is a non-empty list with first element `x`:

Prelude
```
take n [] = []
take n (x:xs)
  | n > 0 = x:???
  | otherwise = []
```

Now we want `n-1` more elements (if present):

Prelude
```
take n [] = []
take n (x:xs)
  | n > 0 = x:take (n-1) xs
  | otherwise = []
```

General selection functions

General structure of functions that pick some elements from a list:

```
pick :: [SomeType] -> [SomeType]
pick [] = []
pick (x:xs)
  | ... = ...
  | otherwise = ...
```

Possible lists for the right-hand sides yield different behaviours:

- `[]` – discard **x** and **xs**
- `[x]` – keep **x** but discard **xs**
- `xs` – discard **x** and return the rest without further selection
- `x:xs` – return the input list without further selection
- `pick xs` – discard **x** and the selection of the rest
- `x : pick xs` – keep **x** and the selection of the rest

Generalizing selecting elements from a list

Generalize from

```
letters :: [Char] -> [Char]
letters [] = []
letters (c:cs)
  | isAlpha c = c : letters cs
  | otherwise = letters cs
```

to take a function returning a **Bool** as a parameter:

Prelude

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

Splitting lists

Getting letters from the start of a list:

```
takeLetters :: [Char] -> [Char]
takeLetters [] = []
takeLetters (c:cs)
  | isAlpha c = c : takeLetters cs
  | otherwise = []
```

Generalizing over the predicate:

Prelude

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
  | p x       = x : takeWhile p xs
  | otherwise = []
```

Getting the rest

The rest of the list after initial letters:

```
dropLetters :: [Char] -> [Char]
dropLetters [] = []
dropLetters (c:cs)
  | isAlpha c = dropLetters cs
  | otherwise = c:cs
```

Generalizing over the predicate:

Prelude

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs)
  | p x = dropWhile p xs
  | otherwise = x:xs
```

Other higher-order list functions

Mapping a function over a list

Generalize from

```
ordAll :: [Char] -> [Int]
ordAll [] = []
ordAll (c:cs) = ord c : ordAll cs

doubleAll :: [Int] -> [Int]
doubleAll [] = []
doubleAll (n:ns) = double n : doubleAll ns
  where double x = 2*x
```

to a function that takes a function as a parameter (a *higher-order function*):

Prelude

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

interpreter

```
map ord "Hello world"
```

Folding lists to summary values

Another common pattern of recursion over lists:

```
sum :: [Int] -> Int
sum [] = 0
sum (n:ns) = n + sum ns

product :: [Int] -> Int
```

```
product [] = 1
product (n:ns) = n * product ns

concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss
```

Generalized folding

Special case:

```
sum :: [Int] -> Int
sum [] = 0
sum (n:ns) = n + sum ns
```

Generalization:

Prelude

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Examples:

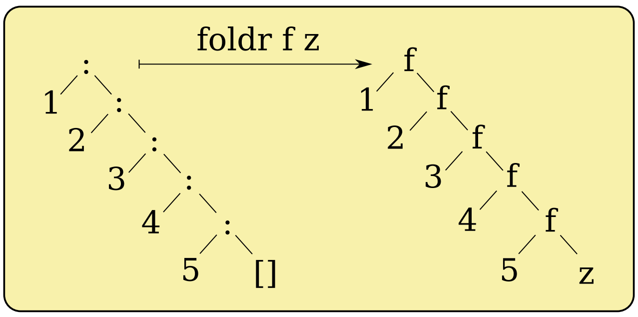
interpreter

```
foldr (+) 0 [1..5]
foldr (*) 1 [1..5]
foldr (++) [] [[1], [2,3], [], [4,5]]
```

Using `foldr`

The application `foldr f z xs` replaces

- `:` by `f`, and
- `[]` by `z`.



The previous functions become

```
sum ns = foldr (+) 0 ns
product ns = foldr (*) 1 ns
concat xss = foldr (++) [] xss
```

Folding non-empty lists

Sometimes there is no answer in the empty list case:

Prelude

```
maximum :: [Int] -> Int
maximum [x] = x
maximum (x:xs) = max x (maximum xs)
```

Generalization:

Prelude

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Examples:

interpreter

```
foldr1 max [3,1,4,5,9]
foldr1 min [3,1,4,5,9]
foldr1 max []
```

Next week: recursive data types

- In sessions 2 and 3, we defined new types with `data`, and defined functions over them using pattern matching.
- We can represent many structures using recursive data types, *e.g.*

```
data Tree
    = Leaf Int
    | Branch Tree Tree
```

- Functions on these types are typically recursive, *e.g.*

```
sumTree :: Tree -> Int
sumTree (Leaf n) = n
sumTree (Branch l r) = sumTree l + sumTree r
```

Exercises

1. Consider the following definitions

```
foo :: Eq a => [a] -> [a]
foo [] = []
foo (x:xs) = x : bar x xs

bar :: Eq a => a -> [a] -> [a]
bar x [] = []
bar x (y:ys)
  | x == y = bar x ys
  | otherwise = y : bar y ys
```

What does the function **foo** do?

2. One way to develop a higher-order function is to write a specific version and then generalize it.

   (a) Define a recursive function

   ```
   removeFirstDigit :: [Char] -> [Char]
   ```

   (using pattern matching over lists) that removes the first element of the list that is a digit, but keeps the rest. (You will need to import `Data.Char`.) Test your function on the strings `"r2d2"` and `"2d2"`.

   (b) Generalize the above to a recursive higher-order function

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

   that removes the first element of the list that does not satisfy the property, but keeps the rest.

3. (a) Define a function

   ```
   addLists :: Num a => [a] -> [a] -> [a]
   ```

   that adds corresponding elements of two lists of the same type, with the extra elements of the longer list added at the end, e.g.

   - `addLists [1,2,3] [1,3,5,7,9] = [2,5,8,7,9]`

   (b) Generalize the previous function to a higher-order function that takes the combining function as an argument:

   ```
   longZip :: (a -> a -> a) -> [a] -> [a] -> [a]
   ```

   For example, `addLists` should be equivalent to `longZip (+)`.

4. (a) Write a function

   ```
   merge :: Ord a => [a] -> [a] -> [a]
   ```

   that takes two lists, assumed to be ordered, and produces an ordered list obtained by merging these two lists.

   (b) Give recursive definitions of functions (from session 4)

   ```
   odds :: [a] -> [a]
   evens :: [a] -> [a]
   ```

   that return lists consisting of every second element of the original list. For example,

   - `odds "Haskell"= "Hsel"`
   - `evens "Haskell"= "akl"`

   A particularly neat way to do this is to define each of these functions using the other.

   (c) One idea for sorting (and an efficient one too) is to split a list into roughly equal halves, sort those, and merge the results. (That's for lists of at least two elements – shorter lists are easier.) Use the functions from the previous two parts to implement

   ```
   mergeSort :: Ord a => [a] -> [a]
   ```