Informatics 1
Functional Programming Lecture 6, continued

Even more fun with recursion

Don Sannella
University of Edinburgh
Part I

Select, take, and drop
Select, take, and drop

Prelude> "words" !! 3
'd'

Prelude> take 3 "words"
"wor"

Prelude> drop 3 "words"
"ds"

Select the item in the nth position of a list. Isn't used nearly as often as you'd expect.

Return the first n items in a list.

Return all except the first n items in a list.

These are all built-in functions in Haskell. They work on lists of any type, not just strings.
Select, take, and drop (comprehensions)

\[\text{selectComp :: } [a] \to \text{Int} \to a \quad -- \quad (!!)\]
\[\text{selectComp } xs \ i \ = \ \text{the } [ \ x \mid (j,x) \leftarrow \text{zip } [0..] \ xs, \ j == i \ ]\]
\[\text{where}\]
\[\text{the } [x] \ = \ x\]

\[\text{takeComp :: } \text{Int} \to [a] \to [a]\]
\[\text{takeComp } i \ xs \ = \ [ \ x \mid (j,x) \leftarrow \text{zip } [0..] \ xs, \ j < i \ ]\]

\[\text{dropComp :: } \text{Int} \to [a] \to [a]\]
\[\text{dropComp } i \ xs \ = \ [ \ x \mid (j,x) \leftarrow \text{zip } [0..] \ xs, \ j >= i \ ]\]

All of these can be defined using the same trick as in the comprehension definition of search, with \text{zip } [0..] \ xs. Using different names to avoid conflict with the built-in functions.
How take works (comprehension)

```
takeComp :: Int -> [a] -> [a]
takeComp i xs = [ x | (j,x) <- zip [0..] xs, j < i ]

take 3 "words"
=
[ x | (j,x) <- zip [0..] "words", j < 3 ]
=
[ x | (j,x) <- [(0,'w'),(1,'o'),(2,'r'),(3,'d'),(4,'s')], j < 3 ]
=
['w'|0<3]++['o'|1<3]++['r'|2<3]++['d'|3<3]++['s'|4<3]
=
['w']++['o']++['r']++[]++[]
= "wor"
```
Lists

Every list can be written using only (: ) and [].

\[ [1,2,3] = 1 : (2 : (3 : [])) \]

"list" = [‘l’,’i’,’s’,’t’]
     = ‘l’ : (‘i’ : (‘s’ : (‘t’ : [])))

A recursive definition: A list is either

- null, written [], or

- constructed, written \( x : xs \),
  with head \( x \) (an element), and tail \( xs \) (a list).

Remember the definition of lists ...
Natural numbers

Every natural number can be written using only \((+1)\) and 0.

\[
3 = ((0 + 1) + 1) + 1
\]

A recursive definition: A natural number is either

- **zero**, written 0, or

- **successor**, written \(n+1\)
  with **predecessor** \(n\) (a natural number).

... we can do something similar to define NATURAL NUMBERS (positive integers). Then we can use this to define functions using recursion on natural numbers. In Haskell, we use \(n\) and \(n-1\) rather than \(n+1\) and \(n\), once we have dealt with \(n=0\).
Select, take, and drop (recursion)

\[(!!) :: [a] \to \text{Int} \to a\]
\[(x:xs) !! 0 = x\]
\[(x:xs) !! i = xs !! (i-1)\]

\[\text{take} :: \text{Int} \to [a] \to [a]\]
\[\text{take} 0 \ \text{xs} \quad = \quad []\]
\[\text{take} \ i \ \text{[]} \quad = \quad []\]
\[\text{take} \ i \ (x:xs) \quad = \quad x : \text{take} \ (i-1) \ \text{xs}\]

\[\text{drop} :: \text{Int} \to [a] \to [a]\]
\[\text{drop} \ 0 \ \text{xs} \quad = \quad \text{xs}\]
\[\text{drop} \ i \ \text{[]} \quad = \quad []\]
\[\text{drop} \ i \ (x:xs) \quad = \quad \text{drop} \ (i-1) \ \text{xs}\]

These definitions do simultaneous recursion on \(i\) and \(xs\).
That's why we need two base cases.
Pattern matching and conditionals (squares)

Pattern matching

\[
\begin{align*}
\text{squares} :: [\text{Int}] & \rightarrow [\text{Int}] \\
\text{squares} [] & = [] \\
\text{squares} (x:xs) & = x^2 : \text{squares} \, xs
\end{align*}
\]

Conditionals with binding

\[
\begin{align*}
\text{squares} :: [\text{Int}] & \rightarrow [\text{Int}] \\
\text{squares} \, ws & = \\
& \quad \text{if} \ \text{null} \ ws \ \text{then} \\
& \quad [] \\
& \quad \text{else} \\
& \quad \quad \text{let} \\
& \quad & \quad x = \text{head} \ ws \\
& \quad & \quad xs = \text{tail} \ ws \\
& \quad & \quad \text{in} \\
& \quad & \quad x^2 : \text{squares} \, xs
\end{align*}
\]
Pattern matching and conditionals (take)

Pattern matching

\[
\text{take} :: \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
\text{take } 0 \ \text{xs} \quad = \quad [] \\
\text{take } i \ [\] \quad = \quad [] \\
\text{take } i \ (x:x\text{s}) \quad = \quad x \ : \ \text{take } (i-1) \ \text{x}\text{s}
\]

Conditionals with binding

\[
\text{take} :: \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
\text{take } i \ \text{ws} \\
\quad \text{if } i == 0 || \ \text{null } \text{ws} \ \text{then} \\
\quad \quad [] \\
\quad \text{else} \\
\quad \quad \text{let} \\
\quad \quad \quad x \ = \ \text{head } \text{ws} \\
\quad \quad \quad x\text{s} \ = \ \text{tail } \text{ws} \\
\quad \quad \text{in} \\
\quad \quad x \ : \ \text{take } (i-1) \ \text{x}\text{s}
\]
Pattern matching and guards (take)

Pattern matching

\[
\text{take} :: \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]}
\]
\[
\text{take } 0 \; \text{xs} \; = \; []
\]
\[
\text{take } i \; [] \; = \; []
\]
\[
\text{take } i \; (x:xs) \; = \; x : \text{take} \; (i-1) \; xs
\]

Guards

\[
\text{take} :: \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]}
\]
\[
\text{take } 0 \; \text{xs} \; = \; []
\]
\[
\text{take } i \; [] \; = \; []
\]
\[
\text{take } i \; (x:xs) \; | \; i > 0 \; = \; x : \text{take} \; (i-1) \; xs
\]
How take works (recursion)

```haskell
take :: Int -> [a] -> [a]
take 0 xs = []
take i [] = []
take i (x:xs) = x : take (i-1) xs

take 3 "words"
=
  'w' : take 2 "ords"
=
  'w' : ('o' : take 1 "rds")
=
  'w' : ('o' : ('r' : take 0 "ds"))
=
  'w' : ('o' : ('r' : []))
=
  "wor"
```
The infinite case

take :: Int -> [a] -> [a]
take 0 xs = []
take i [] = []
take i (x:xs) = x : take (i-1) xs

takeComp :: Int -> [a] -> [a]
takeComp i xs = [ x | (j,x) <- zip [0..] xs, j < i ]

Prelude> take 3 [10..]
[10,11,12]

Prelude> takeComp 3 [10..]
[10,11,12  -- computation goes on forever!

The comprehension version of take (takeComp) only works for finite lists - keeps going forever for infinite lists. Using recursion, the cost is proportional to the number of elements taken. Using comprehension, the cost is proportional to the length of the list. (Same for drop and select.)
The infinite case explained

Function takeComp is equivalent to takeCompRec.

```
    takeCompRec :: Int -> [a] -> [a]
    takeCompRec i xs = helper 0 i xs
        where
            helper j i [] = []
            helper j i (x:xs) | j < i = x : helper (j+1) i xs
                             | otherwise = helper (j+1) i xs
```

takeCompRec 3 [10..]
=
   helper 0 3 [10..]
=
   10 : helper 1 3 [11..]
=
   10 : (11 : helper 2 3 [12..])
=
   10 : (11 : (12 : helper 3 3 [13..]))
=
   10 : (11 : (12 : helper 4 3 [14..]))
=
   ...
```
Part II

Arithmetic

Optional material: arithmetic functions defined using recursion.
Arithmetic (recursion)

\[(+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[m + 0 = m \]
\[m + n = (m + (n-1)) + 1 \]

\[(*) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[m * 0 = 0 \]
\[m * n = (m * (n-1)) + m \]

\[(^) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[m ^ 0 = 1 \]
\[m ^ n = (m ^ (n-1)) * m \]
How arithmetic works (recursion)

\[
(+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
m + 0 = m \\
m + n = (m + (n-1)) + 1
\]

\[
2 + 3 = (2 + 2) + 1 = ((2 + 1) + 1) + 1 = (((2 + 0) + 1) + 1) + 1 = (2 + 1) + 1 = 5
\]
Giuseppe Peano (1858–1932)

The definition of the natural numbers is named the *Peano axioms* in his honour. Made key contributions to the modern treatment of mathematical induction.