Informatics 1
Functional Programming Lecture 5

Select, Take, Drop

Don Sannella
University of Edinburgh
Part I

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

> "words" !! 3
'd'

> take 3 "words"
"wor"

> drop 3 "words"
"ds"
Select, take, and drop (comprehensions)

```haskell
selectComp :: [a] -> Int -> a -- (!!)
selectComp xs i = the [ x | (j,x) <- zip [0..] xs, j == i ]
  where
    the [x] = x

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

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

How take works (comprehension)

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

takeComp 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).
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).
Select, take, and drop (recursion)

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

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

drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop i [] = []
drop i (x:xs) = drop (i-1) xs
Pattern matching and conditionals (squares)

Pattern matching

squares :: [Int] -> [Int]
squares [] = []
squares (x:xs) = x*x : squares xs

Conditionals with binding

squares :: [Int] -> [Int]
squares ws =
  if null ws then
    []
  else
    let
      x = head ws
      xs = tail ws
    in
      x*x : squares xs
Pattern matching and conditionals (take)

Pattern matching

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

Conditionals with binding

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

Pattern matching

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

Guards

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

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

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

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

\[
> \text{take} \ 3 \ [[10..] \\
[10,11,12]
\]

\[
> \text{takeComp} \ 3 \ [[10..] \\
[10,11,12] \quad -- \text{computation goes on forever}!
\]
The infinite case explained

**Function** `takeComp` **is equivalent to** `takeCompRec`.

```haskell
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
```

```haskell
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
Arithmetic (recursion)

(+) :: Int -> Int -> Int
m + 0   =  m
m + n   =  (m + (n-1)) + 1

(*) :: Int -> Int -> Int
m * 0   =  0
m * n   =  (m * (n-1)) + m

(^) :: Int -> Int -> Int
m ^ 0   =  1
m ^ n   =  (m ^ (n-1)) * m
How arithmetic works (recursion)

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

\[
2 + 3 =
\]
\[
(2 + 2) + 1 =
\]
\[
((2 + 1) + 1) + 1 =
\]
\[
(((2 + 0) + 1) + 1) + 1 =
\]
\[
((2 + 1) + 1) + 1 =
\]
\[
5
\]
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.