Functional Programming

2022-2023

Sjaak Smetsers

Lists

Lecture 3

Outline

- List notation
- Compositional programming
- Case study: DNA analysis
- List constructors
- List design pattern
- Some list operations
- List comprehensions

```
(:) :: a→[a]→[a]
```

List notation

- lists are central to functional programming (cf LISP!)
- enclosed in square brackets, comma-separated: `[1,2,3]`, `[ ]`
- variable-length sequences of elements of the same type
- the type of lists with elements of type `T` is `[T]`
- strings are just lists of characters: `['N','a','n','c','y']`

```
type String = [Char]
```

  but with special syntax `"Nancy"`
- list elements can be any type:

```
[1,2,3] :: [Integer]
[[1,2],[ ],[3]] :: [[Integer]]
[(+),(*)] :: [Integer → Integer → Integer]
[(/2),(+ (-2))] :: [Integer → Integer]
```

Some library functions

- exploring the library `Data.List`

```
import Data.List
```

- see also `https://www.haskell.org/hoogle/`

```
length :: [a]⟶Int
```
e.g.
```
length [1,2,3] = 3
```

```
reverse :: [a]⟶[a]
```
e.g.
```
reverse "billy" = "yllib"
```

```
concat :: [[a]]⟶[a]
```
e.g.
```
concat [[1,2],[ ],[3]] = [1,2,3]
```

```
map :: (a⟶b)⟶[a]⟶[b]
```
e.g.
```
map (+ 1) [1,2,3] = [2,3,4]
```

```
filter :: (a⟶Bool)⟶[a]⟶[a]
```
e.g.
```
filter (< 6) [2,7,6,5]=[2,5]
```

remark: some operations actually have more generic types, but here we pretend that you can only use them on lists.

More library functions

- `lines :: String⟶[String]` e.g.
  `lines "a\nbc\nd" = ["a","bc","d"]`
- `unlines :: [String]⟶String` e.g.
  `unlines ["a","bc","d"] = "a\nbc\nd\n"`
- `tails :: [a]⟶[[a]]` e.g.
  `tails "will" = ["will","ill","ll","l",""]`
- `take :: Int⟶[a]⟶[a]` e.g.
  `take 3 "Mike Wheeler" = "Mik"`
- `sort :: (Ord a)⟹[a]⟶[a]` e.g.
  `sort "demogorgon" = "deggmnooor"`
- `group :: (Eq a)⟹[a]⟶[[a]]` e.g.
  `group "jim hopper" = ["j","i","m"," ","h","o","pp","e","r"]`
- and many more . . .

How to solve it?

- write down the type (what's the input?, what's the output?)
- can you solve it using existing vocabulary?
- use function application
- some exercises: given a string (a list of characters)
  - remove newlines
  - count the number of lines
  - flip text upside down
  - flip text from left to right
  - determine the list of all segments

```
lines    :: String⟶[String]
unlines  :: [String]⟶String
tails    :: [a]⟶[[a]]
take     :: Int⟶[a]⟶[a]
sort     :: (Ord a)⟹[a]⟶[a]
group    :: (Eq a)⟹[a]⟶[[a]]

reverse  :: [a]⟶[a]
concat   :: [[a]]⟶[a]
map      :: (a⟶b)⟶([a]⟶[b])
```

Solutions

- remove newlines

```
unwrap :: String⟶String
unwrap s = concat (lines s)
```

- count the number of lines

```
countLines :: String⟶Int
countLines s = length (lines s)
```

- flip text upside down

```
upsideDown :: String⟶String
upsideDown s = unlines (reverse (lines s))
```

- flip text from left to right

```
leftRight :: String⟶String
leftRight s = unlines (map reverse (lines s))
```

Solutions continued

- determine the list of all prefixes (actually, also defined in the library: `inits`)

```
suffixes, prefixes :: String⟶[String]
suffixes s = tails s
prefixes s = map reverse (tails (reverse s))
```

- determine the list of all segments

```
segments :: String⟶[String]
segments s = concat (map prefixes (suffixes s))
```

Case study: DNA analysis

You are working on the evolutionary history of rodents. To this end you analyse the genome of various species, considering genome size, proportions of non-repetitive DNA, and repetitive DNA.

In particular, you are interested in finding repeated segments of length *m* in a DNA sequence of length *n*.

ATGTAAAGGGTCCAATGACTGGAATTACTTCACAGCCCTGACACTGTGGAGAGATGGATA
CCAGTTTAATTATGGACAGCTGAGAATAATCTGGGAGTCACCTGCTCTGGATATTTATCA
TCATTATGAACTTCTCACCAGGCTGTGGCCCGAACACTTTCATAACGTCCCATTTGTGTT
GGGCAGACATTATGATCTATACAGAACCTGCCTTGTCTGTCACCTTTGAATTTATGGATA
GACAATTTAATAT
GTGTTCCTGGCAGCAAAGATAATCATGGAGAGTGGAGAGAAACTAACCTTACCATTGATA
GGGAAACTCTTGAAGTGTCAACTTCTCCATATTAAATCCAAGGACCAGCAGAGACGAGAA
AATGAAAAGAAGATGGTAGAAGAAAGAACAAAATCAGAAAAAGACAAAGGAAAAGGGAAG
TCTCCAAAGGAGAAGAAAGTTGCCAGTGCCAAGCCTGGGAAGGGAAAGAGCAAGGACCAG
. . .

Think big

- representation of DNA data

```
data Base = A | C | G | T
    deriving (Eq,Ord,Show)
type DNA = [Base]
dna :: DNA
dna = [A,T,G,T,A,A,A,G,G,G,T,C,C,A,A,T,G,A]
```

- think in terms of entire lists, not individual list elements
- think in terms of transformations
- algorithmic idea:
  - step 1: generate all *m*-segments
  - step 2: identify repeated *m*-segments

Step 1: generate all m-segments (m = 3)

```
   ATGTAAAGGGTCCAATGA
⇩ tails
   [ATGTAAAGGGTCCAATGA,TGTAAAGGGTCCAATGA,
     ...
    CCAATGA,CAATGA,AATGA,ATGA,TGA,GA,A,]
⇩ map (take 3)
   [ATG,TGT,GTA,TAA,AAA,AAG,AGG,GGG,GGT,GTC,
    TCC,CCA,CAA,AAT,ATG,TGA,GA,A,]
```

Step 2: identify repeated m-segments

```
    [ATG,TGT,GTA,TAA,AAA,AAG,AGG,GGG,GGT,GTC,
     TCC,CCA,CAA,AAT,ATG,TGA,GA,A,]
⇩ sort
    [,A,AAA,AAG,AAT,AGG,ATG,ATG,CAA,CCA,GA,
      GGG,GGT,GTA,GTC,TAA,TCC,TGA,TGT]
⇩ group
    [[ ],[A],[AAA],[AAG],[AAT],[AGG],[ATG,ATG],[CAA],
     [CCA],[GA],[GGG],[GGT],[GTA],[GTC],[TAA],
     [TCC],[TGA],[TGT]]
⇩ filter (\s -> length s > 1)
    [[ATG,ATG]]
```

Solution

- chain transformations

```
repeatedSegments :: Int→DNA→[[DNA]]
repeatedSegments m dna
= filter (\x→length x > 1) (group (sort (map (take m) (tails dna))))
```

- using composition (more in week 5)

```
g ∘ f = \x → g (f x)
```

```
repeatedSegments :: Int→DNA→[[DNA]]
repeatedSegments m
= filter (\x→length x > 1) ∘ group ∘ sort ∘ map (take m) ∘ tails
```

- transformational programming at work

List constructors

- a list is either
  - empty, written `[ ]`
  - or consists of an element `x` followed by a list `xs`, written `x : xs`
- every (finite) list can be built up from `[ ]` using `:`
- e.g. `[1,2,3]` = `1 : (2 : (3 : [ ]))` = `1 : 2 : 3 : [ ]`
- `[ ]` and `:` are called *constructors*

Type of list constructors

- nil: the empty list

  `[] :: [a]`

- cons: function for prefixing an element onto a list

  `(:) :: a→[a]→[a]`

- `[]` and `:` are polymorphic!
- puzzle: is `[]:[]` well-typed? what's the type? what about `[]:([]:[])` and `([]:[]):[]`?

Pattern matching

- to define a function over lists, it suffices to consider the two cases `[]` and `:`
- e.g. to test if list is empty

```
null :: [a]⟶Bool
null []     = True
null (_:_)  = False
```

- (subtle: why is this different from `( ==[])`?)
- e.g. to return first element of non-empty list

```
head :: [a]⟶a
head (x:_) = x
```

Case analysis

- cases can also be analysed using a *case-expression*

```
null :: [a]⟶Bool
null xs = case xs of
            []    ⟶ True
            (_:_) ⟶ False
```

- declaration style: equations using patterns; expression style: case-expression using patterns

Recursive definitions

- definitions by pattern-matching can be recursive too
- natural thing to do as the type is also recursively defined
- eg sum of a list of integers

```
sum :: [Integer]⟶Integer
sum []     = 0
sum (x:xs) = x + sum xs
```

- eg length of a list of elements

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

Computation = reduction

```
[2,3,4] = 2 : 3 : 4 : []
```

```
sum :: [Integer]⟶Integer
sum []       = 0
sum (x:xs) = x + sum xs
```

```
sum [2,3,4]
= 2 + sum [3,4]
= 2 + (3 + sum [4])
= 2 + (3 + (4 + sum []))
= 2 + (3 + (4 + 0))
= 2 + (3 + 4)
= 2 + 7
= 9
```

List design pattern

- remember: every type comes with a pattern of definition
- task: define a function `f :: [P]⟶S`
- step 1: solve the problem for the empty list

```
f [] = ...
```

- step 2: solve the problem for non-empty lists;
- assume that you already have the solution for `xs` at hand; extend the intermediate solution to a solution for `x:xs`

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

- you have to program only a step
- put on your problem-solving glasses

Some list operations

solution for xs

- append: `[1,2,3] ++ [4,5] = [1,2,3,4,5]`

```
(++) :: [a]→[a]→[a]
[] ++ ys     = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

- concatenation: `concat [[1,2],[],[3]] = [1,2,3]`

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

- reverse: `reverse [1,2,3] = [3,2,1]`

```
reverse :: [a]→[a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Computation = reduction

```
[1,2,3] = 1 : 2 : 3 : []
```

```
(++) :: [a]⟶[a]⟶[a]
[] ++ ys     = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
[1,2,3] ++ [4,5]
= 1:([2,3] ++ [4,5])
= 1:(2:([3] ++ [4,5]))
= 1:(2:(3:([] ++ [4,5])))
= 1:(2:(3:([4,5])))
≡ [1,2,3,4,5]
```

More list operations

- is a list ordered?

```
ordered :: (Ord a)⇒[a]⟶Bool
ordered [] = True
ordered [_] = True
ordered (x1:x2:xs) = x1 ≤ x2 && ordered (x2:xs)
```

we distinguish three cases

- zip: eg `zip [1,2,3] "abc" = [(1,’a’),(2,’b’),(3,’c’)]`

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

we pattern match on both arguments

List comprehensions

- two useful operators on lists: `map` and `filter`
- list comprehensions provide a convenient syntax for expressions involving `map`, `filter`, and `concat`
- useful for constructing new lists from existing lists

Map

- applies given function to every element of given list
- eg `map square [1,2,3] = [1,4,9]`
- eg `map succ "HAL" = "IBM"`
- definition

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

- eg `sum (map square [1..10])`

Computation = reduction

```
even :: Int → Bool
```

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

```
  map even [1,2,3,4]
= even 1:map even [2,3,4]
= False:map even [2,3,4]
= False:even 2:map even [3,4]
= False:True:map even [3,4]
= False:True:even 3:map even [4]
= False:True:False:map even [4]
= False:True:False:even 4:map even []
= False:True:False:True:map even []
= False:True:False:True:[]
≡ [False,True,False,True]
```

Filter

- returns sub-list of the argument whose elements satisfy given predicate
- eg `filter isDigit "more4u2say" = "42"`
- eg `(sum ∘ map square ∘ filter odd) [1..5] = 35`
- definition

```
filter :: (a⟶Bool)⟶[a]⟶[a]
filter _p []    = []
filter p (x:xs)
    | p x       = x:filter p xs
    | otherwise = filter p xs
```

List comprehensions

- special convenient syntax for list-generating expressions
- Map: alter every element `x` from `xs` (by applying `f`)

```
[ f x | x ← xs ]
```

- Filter: select element `x` from `xs` with predicate `pred`

```
[ f x | x ← xs, pred x ]
```

- Cartesian product: combine each element `x` from `xs` with each element `y` from `ys` (nested loops)

```
[ g x y | x ← xs, y ← ys ]
```

- Example: `sum [square x | x ← [1..5], odd x]`

Comprehensions

- formally, a comprehension $[e \mid Qs]$ for expression e and non-empty comma-separated sequence of qualifiers Qs
- qualifier may be
  - *generator* of the form $x \leftarrow xs$ or
  - *guard* i.e. a Boolean expression

Examples of comprehensions

- eg primes up to a given bound

```
primes, divisors :: Integer⟶[Integer]
primes m   = [n | n ⟵ [1..m], divisors n == [1,n]]
divisors n = [d | d ⟵ [1..n], n `mod` d == 0]
```

- eg database query

```
overdue =
   [(name,addr) | (key,name,addr) ⟵ customers,
                  (key',date) ⟵ invoices, key == key',
                  date < today]
```

- eg Quicksort

```
quicksort :: (Ord a)⟹[a]⟶[a]
quicksort []     = []
quicksort (x:xs) = quicksort [a | a ⟵ xs, a < x]
                      ++ [x] ++
                   quicksort [a | a ⟵ xs, x ≤ a]
```

Another point of view

- list comprehension is ‘really’ a form of nested loop
- eg `[f b | a ← x, b ← g a, p b]` is related to

```
List<Integer> res = new LinkedList<>();
for ( int a : x ) {
  for ( int b: g(a) ) {
    if ( p(b) ) {
      res.add(f(b));
    }
  }
}
return res;
```

Zip

- traversing two lists `xs`, `ys` simultaneously: combine each element `x` from `xs` pairwise with element `y` from `ys`
- example: define a function `odds` that returns all elements of the input lists that are located at odd positions.
- eg:
  - `odds [1,2,3,4,5] = [1,3,5]`
  - `odds "Stannis Baratheon" = "SansBrten"`
- Only basic/library functions and/or list comprehensions,
- Solution: `odds l = [ x | (i,x) ← zip [1..] l, odd i ]`
- In general: `[ …x…y… | (x,y) ← zip xs ys ]`

patterns are allowed

Summary: How to solve it?

- write down the type (what's the input?, what's the output?)
- can you solve the problem using existing vocabulary?
- if not, define new vocabulary
- use the list design pattern
  - remember: you only have to solve a step
- can you solve the step using existing vocabulary?
- if not, define new vocabulary (identify a sub-problem)
- solve the sub-problem in the same manner