Intro to FP
Part 2: Lists and other Algebraic Datatypes

Jeremy Gibbons
University of Oxford

John Hughes 2023

2.1 List notation

- lists are central to functional programming (cf LISP!)
- sequences of elements of the same type
- enclosed in square brackets, comma-separated: $[1,2,3]$, $[\,]$
- the type of lists with elements of type A is $[A]$
- strings are just lists of characters: `['H','e','l','l','o']`

  type *String* = [*Char*]

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

$$\begin{array}{ll} [1,2,3] & :: [\mathit{Integer}] \\ [[1,2],[\,],[3]] & :: [[\mathit{Integer}]] \\ [(+),(\times)] & :: [\mathit{Integer} \to \mathit{Integer} \to \mathit{Integer}] \end{array}$$

2.2 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 $:$
- eg $[1, 2, 3] = 1 : (2 : (3 : [\,])) = 1 : 2 : 3 : [\,]$

2.2 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 $:$ *in a unique way*
- eg $[1, 2, 3] = 1 : (2 : (3 : [\,])) = 1 : 2 : 3 : [\,]$
- $[\,]$ and $:$ are called *constructors*

2.3 Pattern matching

- constructors are exhaustive
- to define function over lists, it suffices to consider the two cases [ ] and :
- eg to test if list is empty

$$
\begin{aligned}
&null :: [Integer] \to Bool \\
&null\ [\,] \quad\ = True \\
&null\ (x : xs) = False
\end{aligned}
$$

- eg to return first element of non-empty list

$$
\begin{aligned}
&head :: [Integer] \to a \\
&head\ (x : xs) = x
\end{aligned}
$$

2.3 Case analysis

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

$$
\begin{aligned}
&\mathit{null} :: [\mathit{Integer}] \to \mathit{Bool} \\
&\mathit{null}\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
&\qquad\qquad [\,] \qquad\ \to \mathit{True} \\
&\qquad\qquad (x : \mathit{xs}') \to \mathit{False}
\end{aligned}
$$

- *declaration style*: equation using patterns;
  expression style: **case**-expression using patterns

2.3 Recursive definitions

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

$$
\begin{aligned}
&sum :: [Integer] \to Integer \\
&sum\ [\,] \quad\ = 0 \\
&sum\ (x : xs) = x + sum\ xs
\end{aligned}
$$

- eg length of a list

$$
\begin{aligned}
&length :: [Integer] \to Int \\
&length\ [\,] \quad\ = 0 \\
&length\ (x : xs) = 1 + length\ xs
\end{aligned}
$$

2.3 List definition pattern

- remember: every type comes with a definition pattern
- *task:* define a function $f :: [P] \to S$
- *step 1:* solve the problem for the empty list

$$f\,[\,] = \ldots$$

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

$$\begin{aligned} f\,[\,] &= \ldots \\ f\,(x : xs) &= \ldots\, x \ldots\, xs \ldots\, f\,xs \ldots \end{aligned}$$

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

2.4 Some list operations

- append: $[1,2,3] +\!\!+ [4,5] = [1,2,3,4,5]$

$$
\begin{aligned}
&(+\!\!+) :: [Integer] \to [Integer] \to [Integer] \\
&[\,] \quad\quad +\!\!+ ys = ys \\
&(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)
\end{aligned}
$$

- concatenation: $concat\,[[1,2],[\,],[3]] = [1,2,3]$

$$
\begin{aligned}
&concat :: [[Integer]] \to [Integer] \\
&concat\,[\,] \quad\quad\quad = [\,] \\
&concat\,(xs : xss) = xs +\!\!+ concat\,xss
\end{aligned}
$$

- reverse: $reverse\,[1,2,3] = [3,2,1]$

$$
\begin{aligned}
&reverse :: [Integer] \to [Integer] \\
&reverse\,[\,] \quad\quad = [\,] \\
&reverse\,(x : xs) = reverse\,xs +\!\!+ [x]
\end{aligned}
$$

(exercise: complexity? improve!)

2.5 List comprehensions

- two useful operators on lists: *map* and *filter*
- list comprehensions provide a convenient syntax for expressions involving *map*, *filter*, *concat*
- analogous to a database query language
- useful for constructing new lists from old lists

2.5 Map

- applies given function to every element of given list
- eg $map\ square\ [1,2,3] = [1,4,9]$
- a higher-order function
- definition

$$
\begin{aligned}
&map :: (Integer \to Integer) \to [Integer] \to [Integer] \\
&map\ f\ [\,] \quad\quad = [\,] \\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}
$$

- another eg: $sum\ (map\ square\ [1\,..\,10])$
- (special syntax $[m\,..\,n]$ for enumerations)

2.5 Filter

- returns sublist of the argument whose elements satisfy given predicate
- eg $(sum \circ map\ square \circ filter\ odd)\ [1..5] = 35$
- definition

$$
\begin{aligned}
&filter :: (Integer \to Bool) \to [Integer] \to [Integer] \\
&filter\ p\ [\,] = [\,] \\
&filter\ p\ (x : xs) \\
&\quad \mid p\ x \qquad\quad = x : filter\ p\ xs \\
&\quad \mid otherwise = \quad filter\ p\ xs
\end{aligned}
$$

- another higher-order function

2.5 Comprehensions

- special convenient syntax for list-generating expressions
- eg $sum\,[\,square\,x \mid x \leftarrow [\,1\,..\,5\,],\,odd\,x\,]$
- 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* (a boolean expression)

2.5 Examples of comprehensions

- eg primes up to a given bound

$$
\begin{aligned}
&primes, divisors :: Integer \to [Integer] \\
&primes\ m = [n \mid n \leftarrow [1..m], divisors\ n == [1, n]] \\
&divisors\ n = [d \mid d \leftarrow [1..n], n\ `mod`\ d == 0]
\end{aligned}
$$

- eg database query

$$
\begin{aligned}
&overdue = \\
&\quad [(name, addr) \mid (key, name, addr) \leftarrow customers, \\
&\qquad\qquad (key', date, amount) \leftarrow invoices, key == key', \\
&\qquad\qquad date < today]
\end{aligned}
$$

- eg Quicksort

$$
\begin{aligned}
&quicksort :: [Integer] \to [Integer] \\
&quicksort\ [\,] \quad\quad = [\,] \\
&quicksort\ (x : xs) = quicksort\ [y \mid y \leftarrow xs, y < x] \\
&\qquad\qquad\qquad\quad +\!\!+ [x] +\!\!+ \\
&\qquad\qquad\qquad\quad quicksort\ [y \mid y \leftarrow xs, y \geqslant x]
\end{aligned}
$$

2.5 Another point of view

- list comprehension is 'really' a form of nested loop
- eg $[f\ a\ b \mid a \leftarrow x, b \leftarrow g\ a, p\ b]$ is related to

```
foreach (int a in x)
  foreach (int b in g a)
    if (p b)
      yield (f a b)
```

2.6 Polymorphism (a sneak preview)

- most of those list functions not specific to $[Integer]$
- how to capture generality, maintaining safety? polymorphism!

$$
\begin{array}{ll}
[\,] & :: [a] \\
(:) & :: a \to [a] \to [a] \\
null & :: [a] \to Bool \\
length & :: [a] \to Int \\
(+\!\!+) & :: [a] \to [a] \to [a] \\
concat & :: [[a]] \to [a] \\
reverse & :: [a] \to [a] \\
map & :: (a \to b) \to [a] \to [b] \\
filter & :: (a \to Bool) \to [a] \to [a] \\
quicksort & :: (Ord\ a) \Rightarrow [a] \to [a]
\end{array}
$$

-- these are all *parametric* polymorphism

-- this is *constrained* polymorphism

- covered properly tomorrow...

2.7 Compositional programming

- exploring the library *Data.List*

  import *Data.List*

- $concat :: [[a]] \to [a]$ eg $concat\ [[1,2],[\,],[3]] = [1,2,3]$
- $length :: [a] \to Int$ eg $length\ [1,2,3] = 3$
- $reverse :: [a] \to [a]$ eg $reverse$ `"oxford"` $=$ `"drofxo"`
- $map :: (a \to b) \to [a] \to [b]$ eg $map\ (+1)\ [1,2,3] = [2,3,4]$
- $lines :: String \to [String]$ eg $lines$ `"a\nbc\nd"` $= [$`"a"`$,$`"bc"`$,$`"d"`$]$
- $unlines :: [String] \to String$ eg $unlines\ [$`"a"`$,$`"bc"`$,$`"d"`$] =$ `"a\nbc\nd\n"`
- $tails :: [a] \to [[a]]$ eg
  $tails$ `"oxford"` $= [$`"oxford"`$,$`"xford"`$,$`"ford"`$,$`"ord"`$,$`"rd"`$,$`"d"`$,$`""`$]$

2.7 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 and function composition

- 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 substrings

2.7 Solutions

- remove newlines

$$unwrap :: String \to String$$
$$unwrap = concat \circ lines$$

- count the number of lines

$$countLines :: String \to Int$$
$$countLines = length \circ lines$$

- flip text upside down

$$upsideDown :: String \to String$$
$$upsideDown = unlines \circ reverse \circ lines$$

- flip text from left to right

$$leftRight :: String \to String$$
$$leftRight = unlines \circ map\ reverse \circ lines$$

2.7 Solutions continued

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

$$
\begin{aligned}
&\mathit{suffixes}, \mathit{prefixes} :: \mathit{String} \to [\mathit{String}] \\
&\mathit{suffixes} = \mathit{tails} \\
&\mathit{prefixes} = \mathit{map}\ \mathit{reverse} \circ \mathit{tails} \circ \mathit{reverse}
\end{aligned}
$$

- determine the list of all substrings

$$
\begin{aligned}
&\mathit{substrings} :: \mathit{String} \to [\mathit{String}] \\
&\mathit{substrings} = \mathit{concat} \circ \mathit{map}\ \mathit{prefixes} \circ \mathit{suffixes}
\end{aligned}
$$

2.8 Fold, scan, and unfold

- many recursive definitions on lists share a *pattern* of computation
- capture that pattern as a function
  (abstraction, conciseness, general properties, familiarity, . . .)
- *map* and *filter* are two common patterns
- folds and unfolds capture many more

2.8 Fold right

- consider following pattern of definition

$$\begin{aligned} h\ [\,] &= e \\ h\ (x : xs) &= x \mathbin{`op`} h\ xs \end{aligned}$$

(simple variant of list definition pattern: xs is only used in the recursive call)

- then

$$h\ (x : (y : (z : [\,]))) = x \mathbin{`op`} (y \mathbin{`op`} (z \mathbin{`op`} e))$$

- h replaces constructors by functions
- capture pattern as *foldr*

$$\begin{aligned} &foldr :: (a \to b \to b) \to b \to [a] \to b \\ &foldr\ op\ e\ [\,] = e \\ &foldr\ op\ e\ (x : xs) = x \mathbin{`op`} foldr\ op\ e\ xs \end{aligned}$$

2.8 Examples of fold right

- many examples:

$$
\begin{aligned}
sum &= foldr\ (+)\ 0 \\
id &= foldr\ (:)\ [\,] \\
length &= foldr\ (\lambda x\ n \to 1 + n)\ 0 \\
map\ f &= foldr\ ((:) \circ f)\ [\,] \\
concat &= foldr\ (+\!\!+)\ [\,] \\
reverse &= foldr\ snoc\ [\,]\ \textbf{where}\ snoc\ x\ ys = ys +\!\!+ [x] \\
xs +\!\!+ ys &= foldr\ (:)\ ys\ xs
\end{aligned}
$$

- right-to-left computation
- operator may be associative ($+$, $+\!\!+$) but need not (:, *snoc*)

2.8 Sorting

- given

$$
\begin{aligned}
&\mathit{insert} :: (\mathit{Ord}\ a) \Rightarrow a \to [a] \to [a] \\
&\mathit{insert}\ x\ [\,] \qquad\quad = [x] \\
&\mathit{insert}\ x\ (y : ys) \\
&\quad \mid x \leqslant y \qquad\quad = x : y : ys \\
&\quad \mid \mathit{otherwise} = y : \mathit{insert}\ x\ ys
\end{aligned}
$$

- we have

$$
\begin{aligned}
&\mathit{insertSort} :: (\mathit{Ord}\ a) \Rightarrow [a] \to [a] \\
&\mathit{insertSort} = \mathit{foldr}\ \mathit{insert}\ [\,]
\end{aligned}
$$

2.8 Fold left

- not every list function is a *foldr*
- eg $decimal\,[1,2,3] = 123$
- efficient algorithm using *Horner's rule*:

$$decimal\,[x,y,z] = 10 \times (10 \times (10 \times 0 + x) + y) + z$$

- left-to-right computation — hence *foldl*

$$foldl\ op\ e\ [x,y,z] = ((e \mathbin{`op`} x) \mathbin{`op`} y) \mathbin{`op`} z$$

- definition

$$\begin{aligned} &foldl :: (b \to a \to b) \to b \to [a] \to b \\ &foldl\ op\ e\ [\,] \quad\ = e \\ &foldl\ op\ e\ (x : xs) = foldl\ op\ (e \mathbin{`op`} x)\ xs \end{aligned}$$

then $decimal = foldl\ (\lambda n\ d \to 10 \times n + d)\ 0$

2.8 Accumulating parameter

- recall *reverse* program

$$\begin{aligned} &reverse :: [a] \to [a] \\ &reverse = foldr\ (\lambda x\ ys \to ys +\!\!+ [x])\ [\,] \end{aligned}$$

- another definition

$$\begin{aligned} &reverse' :: [a] \to [a] \\ &reverse' = foldl\ (flip\ (:))\ [\,] \end{aligned}$$

- (now what is complexity?)
- second argument of *foldl* is an *accumulating parameter*

2.8 Fold over non-empty lists

- consider computing maximum element in a list of numbers
- $maximum\ (x : xs) = x\ `max`\ maximum\ xs$ suggests *foldr*
- but what is the starting value e?
- e should be $maximum\ [\,]$; also require $x\ `max`\ e = x$
- want to choose e to be smallest possible value
- could restrict to context *Bounded*, with $e = minBound$
- alternatively, don't use on empty list!

- capture pattern via two folds on non-empty lists

$$
\begin{aligned}
&\mathit{foldr1} :: (a \to a \to a) \to [a] \to a \\
&\mathit{foldr1}\ \mathit{op}\ (x : \mathit{xs}) \\
&\quad \mid \mathit{null}\ \mathit{xs} \quad\ = x \\
&\quad \mid \mathit{otherwise} = x\ \text{`}\mathit{op}\text{`}\ \mathit{foldr1}\ \mathit{op}\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldl1} :: (a \to a \to a) \to [a] \to a \\
&\mathit{foldl1}\ \mathit{op}\ (x : \mathit{xs}) = \mathit{foldl}\ \mathit{op}\ x\ \mathit{xs}
\end{aligned}
$$

- eg

$$
\begin{aligned}
&\mathit{foldr1}\ \mathit{op}\ [x, y, z] = x\ \text{`}\mathit{op}\text{`}\ (y\ \text{`}\mathit{op}\text{`}\ z) \\
&\mathit{foldl1}\ \mathit{op}\ [x, y, z] \ = (x\ \text{`}\mathit{op}\text{`}\ y)\ \text{`}\mathit{op}\text{`}\ z
\end{aligned}
$$

- now $\mathit{maximum} = \mathit{foldr1}\ \mathit{max} = \mathit{foldl1}\ \mathit{max}$
- we have defined our own customized loop scheme!

2.8 Scan

- sometimes convenient to apply *foldl* to every initial segment of a list

$$scanl\ op\ e\ [x, y, z] = [e, e\ `op`\ x, (e\ `op`\ x)\ `op`\ y, ((e\ `op`\ x)\ `op`\ y)\ `op`\ z]$$

- eg $scanl\ (+)\ 0$ computes *prefix sums*
- eg $scanl\ (\times)\ 1\ [1\,..\,n]$ computes first $n+1$ factorials
- start with specification

$$\begin{aligned} &scanl :: (b \to a \to b) \to b \to [a] \to [b] \\ &scanl\ op\ e = map\ (foldl\ op\ e) \circ inits \end{aligned}$$

$$\begin{aligned} &inits :: [a] \to [[a]] \\ &inits\ [\,] \quad\ = [[\,]] \\ &inits\ (x : xs) = [\,] : map\ (x:)\ (inits\ xs) \end{aligned}$$

(exercise: *inits* matches pattern for *foldr*)

2.8 Efficient scan

- inefficient, as quadratically many applications of *op*
- straightforward to synthesize more efficient implementation

$$\begin{aligned} &scanl\ op\ e\ [\,] &&= [\,e\,] \\ &scanl\ op\ e\ (x : xs) &&= e : scanl\ op\ (op\ e\ x)\ xs \end{aligned}$$

- dually,

$$\begin{aligned} &scanr :: (a \to b \to b) \to b \to [\,a\,] \to [\,b\,] \\ &scanr\ op\ e = map\ (foldr\ op\ e) \circ tails \end{aligned}$$

$$\begin{aligned} &tails :: [\,a\,] \to [\,[\,a\,]\,] \\ &tails\ [\,] &&= [\,[\,]\,] \\ &tails\ (x : xs) &&= (x : xs) : tails\ xs \end{aligned}$$

has more efficient implementation

$$scanr\ op\ e = foldr\ (\lambda x\ ys \to op\ x\ (head\ ys) : ys)\ [\,e\,]$$

2.8 Duality: unfold

- so far we have focused on *consumers*
- *producers* are important too
- producers (unfolds) are *dual* to consumers (folds)
- common pattern

$$unfoldr :: (b \to Bool) \to (b \to a) \to (b \to b) \to b \to [a]$$
$$unfoldr\ p\ f\ g\ z = \textbf{if}\ p\ z\ \textbf{then}\ [\,]\ \textbf{else}\ f\ z : unfoldr\ p\ f\ g\ (g\ z)$$

- *unfoldr* is *dual* to *foldr* (in what way...?)
- relation to OO iterators?

2.8 Examples of unfold

- $[m..n]$ aka *enumFromTo m n*

$$\begin{aligned} &\mathit{enumFromTo} :: (\mathit{Num}\ a, \mathit{Ord}\ a) \Rightarrow a \to a \to [a] \\ &\mathit{enumFromTo}\ m\ n = \mathit{unfoldr}\ (>n)\ \mathit{id}\ (+1)\ m \end{aligned}$$

- *map* can also be expressed as an unfold

$$\begin{aligned} &\mathit{map} :: (a \to b) \to ([a] \to [b]) \\ &\mathit{map}\ f = \mathit{unfoldr}\ \mathit{null}\ \mathit{head}\ \mathit{tail} \end{aligned}$$

- another sorting algorithm, selection sort

$$\begin{aligned} &\mathit{selectsort} = \mathit{unfoldr}\ \mathit{null}\ \mathit{minimum}\ \mathit{deleteMin} \\ &\quad \textbf{where}\ \mathit{deleteMin} = \ldots \quad \text{-- exercise!} \end{aligned}$$

2.9 Algebraic datatypes

- eg people with names and ages

```
type Name = String
type Age = Integer
data Person = P Name Age
```

- then $P :: Name \to Age \to Person$
- such *constructor functions* do not simplify, they are in normal form
- so they can be used in pattern-matching:

```
showPerson :: Person → String
showPerson (P n a) = "Name: " ++ n ++ ", Age: " ++ show a
```

2.9 Sum types

- datatypes can have multiple *variants*

  data *Maybe a* = *Just a* | *Nothing*

- eg *Just* 13 :: *Maybe Int*; so *Just* :: *a* → *Maybe a* and *Nothing* :: *Maybe a*
- useful for modelling exceptions

  ```
  safeHead :: [a] → Maybe a
  safeHead [ ]      = Nothing
  safeHead (x : xs) = Just x
  ```

- the library definition of unfold:

  ```
  unfoldr :: (b → Maybe (a, b)) → (b → [a])
  unfoldr f z = case f z of
                  Nothing    → [ ]
                  Just (x, z') → x : unfoldr f z'
  ```

2.10 Arithmetic expressions

- algebraic datatypes may be recursive too
- eg datatype of arithmetic expressions

$$\textbf{data}\ \mathit{Expr} = \mathit{Lit}\ \mathit{Integer} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr}$$

- an arithmetic expressions is either a literal,
  or two expressions added together, or two multiplied

2.10 Constructing expressions

- constructing expressions

$$
\begin{aligned}
&expr1, expr2 :: Expr\\
&expr1 = Add\ (Mul\ (Lit\ 4)\ (Lit\ 7))\ (Lit\ 11)\\
&expr2 = Mul\ (Add\ (Lit\ 4)\ (Lit\ 7))\ (Lit\ 11)
\end{aligned}
$$

- note the difference between *syntax*

$$
\begin{aligned}
&\text{Main}\rangle\ Add\ (Lit\ 4)\ (Mul\ (Lit\ 7)\ (Lit\ 11))\\
&Add\ (Lit\ 4)\ (Mul\ (Lit\ 7)\ (Lit\ 11))
\end{aligned}
$$

- and *semantics*

$$
\begin{aligned}
&\text{Main}\rangle\ 4 + 7 \times 11\\
&81
\end{aligned}
$$

2.10 *Expr* definition pattern

- recursive definitions by pattern-matching

$$
\begin{aligned}
&\mathit{evaluate} :: \mathit{Expr} \to \mathit{Integer} \\
&\mathit{evaluate}\ (\mathit{Lit}\ i) &&= i \\
&\mathit{evaluate}\ (\mathit{Add}\ e1\ e2) &&= \mathit{evaluate}\ e1 + \mathit{evaluate}\ e2 \\
&\mathit{evaluate}\ (\mathit{Mul}\ e1\ e2) &&= \mathit{evaluate}\ e1 \times \mathit{evaluate}\ e2
\end{aligned}
$$

- the evaluator essentially replaces syntax (*Add* and *Mul*) by semantics ($+$ and $\times$)

2.10 *Expr* definition pattern

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: Expr \to S$
- *step 1:* solve the problem for literals

$$f\,(Lit\ n) = \ldots n \ldots$$

- *step 2:* solve the problem for addition;
  assume that you already have the solution for x and y at hand; *extend* the intermediate solution to a solution for $Add\ x\ y$

$$\begin{aligned} f\,(Lit\ n) &= \ldots n \ldots \\ f\,(Add\ x\ y) &= \ldots x \ldots y \ldots f\,x \ldots f\,y \ldots \end{aligned}$$

  you have to program only a *step*
- *step 3:* do the same for $Mul\ x\ y$

$$\begin{aligned} f\,(Lit\ n) &= \ldots n \ldots \\ f\,(Add\ x\ y) &= \ldots x \ldots y \ldots f\,x \ldots f\,y \ldots \\ f\,(Mul\ x\ y) &= \ldots x \ldots y \ldots f\,x \ldots f\,y \ldots \end{aligned}$$

2.10 Naturals

- *Peano* definition of natural numbers (non-negative integers)

$$\textbf{data}\ Nat = Zero \mid Succ\ Nat$$

- every natural is either *Zero* or the *Succ*essor of a natural
- eg $Succ\ (Succ\ (Succ\ Zero))$ corresponds to 3
- extraction

$$\begin{aligned} &nat2int :: Nat \to Integer \\ &nat2int\ Zero \quad\ = 0 \\ &nat2int\ (Succ\ n) = 1 + nat2int\ n \end{aligned}$$

- addition

$$\begin{aligned} &plus :: Nat \to Nat \to Nat \\ &plus\ Zero \quad\ n = n \\ &plus\ (Succ\ m)\ n = Succ\ (plus\ m\ n) \end{aligned}$$

(does this look familiar?)

2.10 Peano definition pattern

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: Nat \to S$
- *step 1:* solve the problem for *Zero*

$$f\ Zero = \ldots$$

- *step 2:* solve the problem for *Succ n*;
  assume that you already have the solution for n at hand; *extend* the intermediate solution to a solution for *Succ n*

$$\begin{aligned} f\ Zero &= \ldots \\ f\ (Succ\ n) &= \ldots\ n \ldots\ f\ n \ldots \end{aligned}$$

  you have to program only a *step*
- put on your problem-solving glasses
- (exercise: multiplication, exponentiation)

2.10 Lists

- built-in type of lists is not special (has only special syntax)

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

- eg $[1,2,3]$ or $1:2:3:[\,]$ corresponds to $\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil}))$
- recursive definitions by pattern-matching

$$\begin{aligned}&\mathit{mapList} :: (a \to b) \to (\mathit{List}\ a \to \mathit{List}\ b)\\ &\mathit{mapList}\ f\ \mathit{Nil} = \mathit{Nil}\\ &\mathit{mapList}\ f\ (\mathit{Cons}\ x\ xs) = \mathit{Cons}\ (f\ x)\ (\mathit{mapList}\ f\ xs)\end{aligned}$$

2.10 List definition pattern

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: List\ P \to S$
- *step 1:* solve the problem for the empty list

$$f\ Nil = \ldots$$

- *step 2:* solve the problem for the non-empty list;
  assume that you already have the solution for xs at hand; *extend* the intermediate solution to a solution for $Cons\ x\ xs$

$$\begin{aligned} f\ Nil &= \ldots \\ f\ (Cons\ x\ xs) &= \ldots\ x \ldots\ xs \ldots\ f\ xs \ldots \end{aligned}$$

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

2.10 Binary search trees

- definition

```
data BST k v = Leaf | Branch (BST k v) k v (BST k v)
```

- eg insertion

```
insert :: Ord k ⇒ k → v → BST k v → BST k v
insert k v Leaf = Branch Leaf k v Leaf                -- insert at leaf
insert k v (Branch l k' v' r)
  | k < k'      = Branch (insert k v l) k' v' r       -- insert to the left
  | k > k'      = Branch l k' v' (insert k v r)       -- insert to the right
  | otherwise   = Branch l k' v r                     -- overwrite the root
```

- binary search tree definition pattern?