Exercise: Positional List Splitting

- **take**: \( \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]} \)
  
  The `take` function, applied to a \( k : \text{Int} \) and a list \( xs \), returns the longest prefix of \( xs \) of elements that has no more than \( k \) elements.

- **drop**: \( \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]} \)
  
  The `drop` function, \( k : \text{Int} \) and a list \( xs \), returns the suffix remaining after `take k xs`.

**Laws**:

- \( \text{take k xs} + \text{drop k xs} = xs \)
- \( \text{length (take k xs)} \leq k \)

**Note**: `splitAt k xs = (take k xs, drop k xs)`

Guarded Definitions

\[ \text{sign} \ x \ |
\begin{align*}
  x > 0 &= 1 \\
  x == 0 &= 0 \\
  x < 0 &= -1
\end{align*} \]

\[ \text{choose} :: \text{Ord} \ a \Rightarrow (a,b) \rightarrow (a,b) \rightarrow b \]

\[ \begin{align*}
  | x > y &= v \\
  | x < y &= w \\
  | \text{otherwise} &= \text{error "I cannot decide!"}
\end{align*} \]

If no guard succeeds, the next pattern is tried:

- \( \text{take 0 _} = [] \)
- \( \text{take k _} | k < 0 = \text{error "take: negative argument"} \)
- \( \text{take k (x : xs)} = x : \text{take} (k - 1) \text{xs} \)

\[ \begin{align*}
  \text{take 2 [5,6,7]} &= \text{take 2 (5 : 6 : 7 : [])} \\
  &= 5 : \text{take} (2 - 1) (6 : 7 : []) \\
  &= 5 : \text{take} 1 (6 : 7 : []) \\
  &= 5 : 6 : \text{take} (1 - 1) (7 : []) \\
  &= 5 : 6 : \text{take} 0 (7 : []) \\
  &= 5 : 6 : [] = [5, 6]
\end{align*} \]

where Clauses

If an auxiliary definition is used only locally, it should be inside a **local definition**, e.g.:

\[ \text{commaWords} :: \text{[String]} \rightarrow \text{String} \]

\[ \text{commaWords} [] = [] \]

\[ \text{commaWords} (x : xs) = x + \text{commaWordsAux} \text{xs} \]

**where**

\[ \begin{align*}
  \text{commaWordsAux} [] &= [] \\
  \text{commaWordsAux} \text{xs} &= "," : \text{commaWords} \text{xs}
\end{align*} \]

where clauses are visible **only** within their enclosing clause, here “commaWords (x : xs) = …”

where clauses are visible within all guards:

\[ \begin{align*}
  f \ x \ y | y > z &= ... \\
  | y == z &= ... \\
  | y < z &= ...
\end{align*} \]

**where** \( z = x * x \)

**let Expressions**

Local definitions can also be part of expressions:

\[ \begin{align*}
  f \ k \ n &= \text{let} \ m = k \mod n \\
  &\quad \text{in if} \ m == 0 \\
  &\quad \text{then} \ n \\
  &\quad \text{else} \ f \ n \ m
\end{align*} \]

\[ \begin{align*}
  h \ x \ y &= \text{let} \ x2 = x * x \\
  &\quad \text{y2} = y * y \\
  &\quad \text{in} \ sqrt (x2 + y2)
\end{align*} \]

Definitions can use **pattern bindings**:

\[ \begin{align*}
  g \ k \ n &= \text{let} \ (d, m) = \text{divMod} \ k \ n \\
  &\quad \text{in if} \ d == 0 \\
  &\quad \text{then} \ [m] \\
  &\quad \text{else} \ g \ d \ n ++ [m]
\end{align*} \]

Guards, let, and where bindings, and case cases are all **layout sensitive**.
Figuring out the choice between `let` and `where` is a matter of style:

- `let` bindings in an `expression` is an expression.
- `fname patterns guarded RHSs where bindings` is a clause that is part of a definition.
- `(where clauses can also modify case cases)`

Frequently, the choice between `let` and `where` is a matter of style:

- `where` clauses result in a top-down presentation.
- `let` expressions lend themselves also to bottom-up presentations.

### 1. Let or Where?

- `let bindings in expression` is an `expression`.
- `fname patterns guarded RHSs where bindings` is a clause that is part of a `definition`.
- `(where clauses can also modify case cases)`

### 2. Case Expressions

**Case Expressions**

```
sign x = case compare x 0 of
  GT -> 1
  EQ -> 0
  LT -> -1
```

The prelude datatype `Ordering` has three elements and is used mostly as result type of the prelude function `compare`:

```
data Ordering = LT | EQ | GT
```

```haskell
compare :: Ord a ⇒ a → a → Ordering
```

Another example:

```
choose (x, v) (y, w) = case compare x y of
  GT → v
  LT → w
  EQ → error "I cannot decide!"
```

### 3. If...Then...Else... and Case Expressions

The type `Bool` can be considered as a two-element enumeration type:

```
data Bool = False | True
```

Conditional expressions are “syntactic sugar” for `case` expressions over `Bool`:

```
if condition then expr1 else expr2 ≡ case condition of
  True → expr1
  False → expr2
```

Two ways of defining functions:

- **Pattern Matching**
  ```haskell
  not True = False
  not False = True
  ```

- **Case**
  ```haskell
  not b = case b of
    True → False
    False → True
  ```

### 4. Case Expressions are “Anonymous” Pattern Matching

```
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ case xs of
  [] → []
  _ → "", : commaWordsAux xs
```

```
commaWordsAux [] = []
commaWordsAux xs = "," : commaWords xs
```

Every use of a `case` expression can be transformed into the use of an auxiliary function defined by pattern matching:

```
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ commaWordsAux xs
```

```
commaWordsAux [] = []
commaWordsAux xs = "," : commaWords xs
```
Some Prelude Functions — Elementary List Access

- **head**: \([a] \rightarrow a\)
  - head \((x:_):\) = \(x\)

- **last**: \([a] \rightarrow a\)
  - last \([x]\) = \(x\)
  - last \((_:xs)\) = last \(xs\)

- **tail**: \([a] \rightarrow [a]\)
  - tail \((_:xs)\) = \(xs\)

- **init**: \([a] \rightarrow [a]\)
  - init \([x]\) = \([\]\)
  - init \((x:xs)\) = \(x: init \(xs\)\)

- **null**: \([a] \rightarrow Bool\)
  - null \([\]\) = \(True\)
  - null \((_: _)\) = \(False\)

Some Prelude Functions — List Indexing

- **length**: \([a] \rightarrow Int\)
  - length = foldl’ \((\n_ \rightarrow n + 1)\) 0

- **\((!!)\)**: \([b] \rightarrow Int \rightarrow b\)
  - \((x:_)\) !! 0 = \(x\)
  - \((_:xs)\) !! \(n \mid n>0 = xs !! (n-1)\)
  - \((_:_)\) !! _ = error "PreludeList.!!: negative index"
  - \([\]\) !! _ = error "PreludeList.!!: index too large"

Some Prelude Functions — Positional List Splitting

- **take**: \(Int \rightarrow [a] \rightarrow [a]\)
  - take \(0\) _ = \([\]\)
  - take _ \([\]\) = \([\]\)
  - take \(n\) \((x:xs) \mid n>0 = x : take \((n-1)\) \(xs\)
  - take _ _ = error "take: negative argument"

- **drop**: \(Int \rightarrow [a] \rightarrow [a]\)
  - drop \(0\) \(xs\) = \(xs\)
  - drop _ \([\]\) = \([\]\)
  - drop \(n\) \((_:xs) \mid n>0 = drop \((n-1)\) \(xs\)
  - drop _ _ = error "drop: negative argument"

- **splitAt**: \(Int \rightarrow [a] \rightarrow ([a], [a])\)
  - splitAt \(0\) \(xs\) = \(([], xs)\)
  - splitAt _ \([\]\) = \(([], [])\)
  - splitAt \(n\) \((x:xs) \mid n>0 = (x:xs’, xs")\)
    where \((xs’, xs") = splitAt \((n-1)\) \(xs\)
  - splitAt _ _ = error "splitAt: negative argument"

Some Prelude Functions — Concatenation, Iteration

- **\((++)\)**: \([a] \rightarrow [a] \rightarrow [a]\)
  - \([\]\) ++ \(ys\) = \(ys\)
  - \((x:xs)\) ++ \(ys\) = \(x : (xs ++ ys)\)

- **concat**: \([[a]] \rightarrow [a]\)
  - concat = foldr \((++)\) \([\]\)

- **iterate**: \((a \rightarrow a) \rightarrow a \rightarrow [a]\)
  - iterate \(f\) \(x\) = \(x : iterate \(f\) \((f\) \(x)\)

- **repeat**: \(a \rightarrow [a]\)
  - repeat \(x\) = \(xs\) where \(xs = x:xs\)
  - repeat _ _ = for understanding

- **replicate**: \(Int \rightarrow a \rightarrow [a]\)
  - replicate \(n\) \(x\) = \(take\) \(n\) \((repeat\) \(x)\)

- **cycle**: \([a] \rightarrow [a]\)
  - cycle \(xs\) = \(xs’\) where \(xs’ = xs ++ xs’\)
Separation of Concerns: Generation and Consumption

replicate 3 '!' = take 3 (repeat '!' )
= take 3 ('!' : repeat '!' )
= '!' : take (3 - 1) (repeat '!' )
= '!' : take 2 (repeat '!' )
= '!' : take 2 ('!' : repeat '!' )
= '!' : '!' : take 0 (repeat '!' )
= "!!!

Exercise: Splitting with Predicates

• takeWhile :: (a -> Bool) -> [a] -> [a]

takeWhile, applied to a predicate p and a list xs, returns the longest prefix (possibly empty) of xs of elements that satisfy p.

• dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p xs returns the suffix remaining after takeWhile p xs.

Laws:
• all p (takeWhile p xs) = True

— if p is total (on xs).

Note: span p xs = (takeWhile p xs, dropWhile p xs)

Exercise: zipWith

• zip :: [a] -> [b] -> [(a, b)]

zip takes two lists and returns a list of corresponding pairs. If one input list is short, excess elements of the longer list are discarded.

• zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith generalises zip by zipping with the function given as the first argument, instead of a tupling function. For example, zipWith (+) is applied to two lists to produce the list of corresponding sums.

• diagonal :: [[a]] -> [a]

interprets its argument as a matrix, which may be assumed to be square, and returns the main diagonal of that matrix, e.g.:

diagonal [[1,2,3],[4,5,6],[7,8,9]] = [1,5,9]

Some Prelude Functions — List Splitting with Predicates

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

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

span, break :: (a -> Bool) -> [a] -> ([a],[a])
span p [] = ([],[])
span p xs@(x:xs') |
  | p x = let (ys,zs) = span p xs' in (x:ys,zs)
  | otherwise = ([],xs)

break p = span (not . p)
Consider matching of the third clause against `dropWhile (< 5) [1,2,3]`:

- \( p = ( < 5) \)
- \( xs = [1,2,3] \)
- \( x = 1 \)
- \( xs' = [2,3] \)
- \( p x = ( < 5) 1 = 1 < 5 = \text{True} \)

Therefore: \( dropWhile (< 5) [1,2,3] = dropWhile (< 5) [2,3] \)

Consider matching of the third clause against `dropWhile (< 5) [5,4,3]`:

- \( p = ( < 5) \)
- \( xs = [5,4,3] \)
- \( x = 5 \)
- \( xs' = [4,3] \)
- \( p x = ( < 5) 5 = 5 < 5 = \text{False} \)

Therefore: \( dropWhile (< 5) [5,4,3] = [5,4,3] \)

Many functions taking lists as arguments can be defined via \textit{structural induction}:

- \( length :: [a] \rightarrow \text{Int} \)
  \( length [] = 0 \)
  \( length (x : xs) = 1 + \text{length } xs \)

- \( concat :: [[a]] \rightarrow [a] \)
  \( concat [] = [] \)
  \( concat (xs : xss) = xs + \text{concat } xss \)

- \( (+) :: [a] \rightarrow [a] \rightarrow [a] \)
  \( [] + ys = ys \)
  \( (x : xs) + ys = x : (xs + ys) \)

- \( sum :: \text{Num } a \Rightarrow [a] \rightarrow a \)
  \( sum [] = 0 \)
  \( sum (x : xs) = x + \text{sum } xs \)

- \( elem :: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool} \)
  \( x \cdot \text{elem'} [] = \text{False} \)
  \( x \cdot \text{elem'} (y : ys) = x \equiv y \ || \ x \cdot \text{elem'} ys \)

(All these functions are in the standard prelude.)
Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via structural induction:

\[ \text{length} : [a] \rightarrow \text{Int} \]
\[ \text{length} = \text{foldr} \ (\text{const} \ (1 +)) \ 0 \]

\[ (+) : [a] \rightarrow [a] \rightarrow [a] \]
\[ xs + ys = \text{foldr} \ (::) \ ys \ xs \]

\[ \text{elem} :: \text{Eq} \ a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool} \]
\[ \text{elem} \ x = \text{foldr} \ (\lambda y r \rightarrow x \equiv y || r) \ \text{False} \]

\[ \text{concat} : [[a]] \rightarrow [a] \]
\[ \text{concat} = \text{foldr} \ (+) \ [] \]

\[ \text{sum} : \text{Num} a \Rightarrow [a] \rightarrow a \]
\[ \text{sum} = \text{foldr} \ (+) \ 0 \]

\[ \text{product} : \text{Num} a \Rightarrow [a] \rightarrow a \]
\[ \text{product} = \text{foldr} \ (∗) \ 1 \]

(All these functions are in the standard prelude.)

foldr

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

foldrl

foldrl :: (a -> a -> a) -> [a] -> a
foldrl (⊗) [x] = x
foldrl (⊗) (xs) = x ⊗ (foldrl (⊗) xs)

List Folding

foldr abstracts structural induction over lists!

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

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

foldl

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

foldll1 :: (a -> a -> a) -> [a] -> a
foldll1 f (xs) = foldll1 f (f x xs)
Lambda-Abstraction

Named functions:
\[ add1 \ x = x + 1 \]
\[ recip \ x = 1 / x \]
\[ square \ x = x \ast x \]

Anonymous functions:
\[ (+ \ 1) \]
\[ (1 /) \]
\[ \lambda \ x \rightarrow x \ast x \]
\[ \backslash \ x \rightarrow x \ast x \]

In “\( \lambda \ x \rightarrow \text{body} \)”, the variable \( x \) is **bound**.

**Typing rule:**

If, assuming \( x : : a \), we can get \( \text{body} : : b \), then \( (\lambda \ x \rightarrow \text{body}) : : a \rightarrow b \)

**Evaluation rule**: \( \beta \)-**reduction** uses substitution:

\[ (\lambda \ x \rightarrow \text{body}) \ \text{arg} \rightarrow \text{body}[x \mapsto \text{arg}] \]

---

Enumeration Type Definitions

**data** **Bool** = **False** | **True** deriving (Eq, Ord, Read, Show)
**data** **Ordering** = **LT** | **EQ** | **GT** deriving (Eq, Ord, Read, Show)

**data** **Suit** = **Diamonds** | **Hearts** | **Spades** | **Clubs** deriving (Eq, Ord)

Pattern matching:

\[ \text{not} \ \text{False} = \text{True} \]
\[ \text{not} \ \text{True} = \text{False} \]

\[ \text{lexicalCombineOrdering} : : \text{Ordering} \rightarrow \text{Ordering} \rightarrow \text{Ordering} \]
\[ \text{lexicalCombineOrdering} \ \text{LT} \ _ = \ \text{LT} \]
\[ \text{lexicalCombineOrdering} \ \text{EQ} \ x = x \]
\[ \text{lexicalCombineOrdering} \ \text{GT} \ _ = \ \text{GT} \]

---

Simple data Type Definitions

**data** **Point** = **Pt** **Int** **Int** deriving (Eq) —— screen coordinates

This defines at the same time a **data constructor**:

\[ \text{Pt} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Point} \]

Pattern matching:

\[ \text{addPt} \ (\text{Pt} \ x1 \ y1) \ (\text{Pt} \ x2 \ y2) = \text{Pt} \ (x1 + x2) \ (y1 + y2) \]

---

Multi-Constructor data Type Definitions

**data** **Transport** = **Feet** | **Bike** | **Train** **Int** —— price in cent

This defines at the same time **data constructors**:

\[ \text{Feet} :: \text{Transport} \]
\[ \text{Bike} :: \text{Transport} \]
\[ \text{Train} :: \text{Int} \rightarrow \text{Transport} \]

Pattern matching:

\[ \text{cost} \ \text{Feet} = 0 \]
\[ \text{cost} \ \text{Bike} = 0 \]
\[ \text{cost} \ (\text{Train} \ \text{Int}) = \text{Int} \]
**Token Type**

\[\text{data} \quad \text{Token} = \text{Number Integer} \mid \text{Sep Char} \mid \text{Ident String} \quad \text{deriving} \quad (\text{Show})\]

**Constructors:**

- \(\text{Number} :: \text{Integer} \to \text{Token}\)
- \(\text{Sep} :: \text{Char} \to \text{Token}\)
- \(\text{Ident} :: \text{String} \to \text{Token}\)

**Pattern Matching:**

- \(\text{showToken} \ (\text{Number} \ n) = "\text{Number} " + \text{show} \ n\)
- \(\text{showToken} \ (\text{Sep} \ c) = "\text{Sep} " + \text{show} \ c\)
- \(\text{showToken} \ (\text{Ident} \ s) = "\text{Ident} " + \text{show} \ s\)

(Defining this as \(\text{show} :: \text{Token} \to \text{String}\) is the effect of \(\text{deriving} \ (\text{Show})\).)

---

**Lexical Analysis — Haskell Example**

```haskell
module SimpleLexer where
import Char

data Token
    = Number Integer
    | Sep Char
    | Ident String
    deriving (Show)

data simpleLexer :: String -> [Token]
simpleLexer c = lexNumber c
    | lexIdent c
    | lexSep c
    | otherwise = error "simpleLexer:illegal character" ++ take 20 (c:cs)
simpleLexer [] = []

lexNumber cs = lexDigit cs
    | lexAlpha cs
    | lexSep cs
    | otherwise = error "simpleLexer:illegal character" ++ take 20 (c:cs)

lexDigit c = lexNumber (c:cs)
lexAlpha c = lexIdent (c:cs)
lexSep c = lexIdent cs
lexDigit c = lexNumber (c:cs)
lexAlpha c = lexIdent (c:cs)
lexSep c = lexIdent cs

isSep c = c "el" 
```

---

**Abstract Syntax Example — Haskell**

**Abstract Syntax Example — Haskell**

```haskell
data Op = MkOp String
    deriving Show

data Expr
    = Var String
    | Num Integer
    | Bin Expr Op Expr
    deriving Show

expr1 = Bin
    ( Bin ( Var "a")
    ( MkOp "+")
    ( Var "b") )
    ( MkOp "*" )
    ( Var "c")
```

```haskell
plus x y = Bin x (MkOp "+") y
mult x y = Bin x (MkOp "*") y

expr2 = ( Var "a" 'plus' Var "b") 'mult' Var "c"
```
**Showing Expr**

```haskell
data Op = MkOp String
deriving Show

showOp :: Op -> String
showOp (MkOp s) = s

data Expr
  = Var String
  | Num Integer
  | Bin Expr Op Expr

showExpr :: Expr -> String
showExpr (Var v) = v
showExpr (Num n) = show n
showExpr (Bin e1 op e2) =
  ("(" : showExpr e1 ++ showOp op ++ showExpr e2 ++ ")")
```

**Some Prelude Functions — Text Processing**

```haskell
lines :: String -> [String]
lines ""     = []
lines s     = let (l,s') = break ("\n"==) s
             in l : case s' of []     -> []
                 (_:""") -> lines s

words :: String -> [String]
words s     = case dropWhile isSpace s of
              ""     -> []
              s'     -> w : words s"
            where (w,s") = break isSpace s'

unlines :: [String] -> String
unlines = foldr (\ l r -> l ++ '\n' : r) []

unwords :: [String] -> String
unwords [] = ""
unwords [w] = w
unwords (w:ws) = w ++ ' ' : unwords ws
```

**Exercise: Text Processing**

- **lines :: String -> [String]**
  
  `lines` breaks a string up into a list of strings at newline characters. The resulting strings do not contain newlines.

- **words :: String -> [String]**
  
  `words` breaks a string up into a list of words, which were delimited by white space.

- **unlines :: [String] -> String**
  
  `unlines` is an inverse operation to `lines`. It joins lines, after appending a terminating newline to each.

- **unwords :: [String] -> String**
  
  `unwords` is an inverse operation to `words`. It joins words with separating spaces.