

Haskell

- **functional** — programs are function definitions; functions are “**first-class citizens**”
- **pure** (referentially transparent) — “**no side-effects**”
- **non-strict** (lazy) — arguments are evaluated only when needed
- **statically strongly typed** — all type errors caught at compile-time
- **type classes** — safe overloading
- Standardised language version: **Haskell 98**
- Several compilers and interpreters available
- Comprehensive web site: <http://haskell.org/>

Simple Expression Evaluation

The Haskell interpreters `hugs`, `ghci`, and `hi` accept any expression at their prompt and print (after the first ENTER) the value resulting from *evaluation* of that expression.

```
Prelude> 4*(5+6)-2
42
```

Expression evaluation proceeds by applying rules to subexpressions:

$4 * (5+6) - 2$	[subtraction & mult. impossible]
=	(addition)
$4 * 11 - 2$	[subtraction impossible]
=	(multiplication)
$44 - 2$	
=	(subtraction)
42	

Simple Expression Evaluation — Explanation

- Arguments to a function or operation are **evaluated only when needed**.
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position i is always needed. then f is called **strict in its i -th argument**
- Therefore: If f is strict in its i -th argument, then the i -th argument has to be evaluated whenever a result is needed from f .
- Simpler: A one-argument function f is strict iff $f \text{ undefined} = \text{undefined}$.
 - **Constant functions are non-strict:** $(\text{const } 5) \text{ undefined} = 5$
 - Checking a list for emptiness is **strict:** $\text{null } \text{undefined} = \text{undefined}$
 - **List construction is non-strict:** $\text{null } (\text{undefined} : \text{undefined}) = \text{False}$
 - **Standard arithmetic operators are strict in both arguments:**
 $0 * \text{undefined} = \text{undefined}$

Simple Expression Evaluation

The Haskell interpreters `hugs`, `ghci`, and `hi` accept any expression at their prompt and print (after the first ENTER) the value resulting from *evaluation* of that expression.

```
Prelude> 4*(5+6)-2
42
```

Expression evaluation proceeds by applying rules to subexpressions:

$4 * (5+6) - 2$	[subtraction & mult. impossible]
=	(addition)
$4 * 11 - 2$	[subtraction impossible]
=	(multiplication)
$44 - 2$	
=	(subtraction)
42	

Unfolding Definitions

Assume the following definitions to be in scope:

```
answer = 42
magic = 7
```

Expression evaluation will **unfold** (or **expand**) definitions:

```
Prelude> (answer - 1) * (magic * answer - 23)
11111

      (answer - 1) * (magic * answer - 23)
= (42 - 1) * (magic * 42 - 23)           (answer)
= 41 * (magic * 42 - 23)                 (subtraction)
= 41 * (7 * 42 - 23)                     (magic)
= 41 * (294 - 23)                         (multiplication)
= 41 * 271                                 (subtraction)
= 11111                                    (multiplication)
```

How did I find those numbers?

Easy!

```
Prelude> [ n | n <- [1 .. 400] , 11111 `mod` n == 0 ]
[1,41,271]
```

This is a **list comprehension**:

- return all n
- where n is taken from then list $[1 \dots 400]$
- and a result is returned only if n divides 11111.

Conditional Expressions

```
Prelude> if 11111 `mod` 41 == 0 then 11111 `div` 41
else 5
271
```

The pattern is:

if *condition* **then** *expression1* **else** *expression2*

- If the condition evaluates to **True**, the conditional expression evaluates to the value of *expression1*.
- If the condition evaluates to **False**, the conditional expression evaluates to the value of *expression2*.

Therefore: “if _ then _ else” is strict in the condition.

In C: (*condition* ? *expression1* : *expression2*)

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
fact 3
= if 3 == 0 then 1 else 3 * fact (3-1)
= if False then 1 else 3 * fact (3-1)
= 3 * fact (3-1)
= 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
= 3 * if 2 == 0 then 1 else 2 * fact (2-1)
= 3 * if False then 1 else 2 * fact (2-1)
= 3 * 2 * fact (2-1)
= 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
= 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
= 3 * 2 * if False then 1 else 1 * fact (1-1)
= 3 * 2 * 1 * fact (1-1)
= 3 * 2 * 1 * if (1-1) == 0 then 1 else (1-1) * fact ((1-1)-1)
= 3 * 2 * 1 * if 0 == 0 then 1 else 0 * fact (0-1)
= 3 * 2 * 1 * if True then 1 else 0 * fact (0-1)
= 3 * 2 * 1 * 1
= 3 * 2 * 1
= 3 * 2
= 6
```

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

```
fact 3
= 3 * fact (3-1)           (fact n)
= 3 * fact 2              (determining which fact rule matches)
= 3 * (2 * fact (2-1))    (fact n)
= 3 * (2 * fact 1)       (determining which fact rule matches)
= 3 * (2 * (1 * fact (1-1))) (fact n)
= 3 * (2 * (1 * fact 0)) (determining which fact rule matches)
= 3 * (2 * (1 * 1))      (fact 0)
= 3 * (2 * 1)            (multiplication)
= 3 * 2                  (multiplication)
= 6                      (multiplication)
```

Lists

- **List display:** between square brackets explicitly listing all elements, separated by commas:

```
[1,4,9,16,25]
```

- **Enumeration lists:** denoted by ellipsis “..” inside square brackets; defined by beginning (and end, if applicable):

```
[1 .. 10] = [1,2,3,4,5,6,7,8,9,10]
```

```
[1,3 .. 10] = [1,3,5,7,9]
```

```
[1,3 .. 11] = [1,3,5,7,9,11]
```

```
[11,9 .. 1] = [11,9,7,5,3,1]
```

```
[11 .. 1] = []
```

```
[1 .. ] = [1,2,3,4,5,6,7,8,9,10, ...] -- infinite list
```

```
[1,3 .. ] = [1,3,5,7,9,11, ...] -- infinite list
```

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

- either the **empty list**: `[]`,
- or **non-empty**, and **constructed** from a **head** `x` and a **tail** `xs` (read: “`xes`”)

`x : xs` — read: “`x cons xes`”.

“:” is used as *infix list constructor*:

```
3 : [] = [3]
```

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

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

As an infix operator, “:” *associates to the right*:

$$x : y : ys = x : (y : ys)$$

Example:

$$1 : 2 : [3,4] = 1 : (2 : [3, 4]) = 1 : [2, 3, 4] = [1, 2, 3, 4]$$

Cons is Not Associative

The convention that “:” *associates to the right* allows to save parentheses in certain circumstances.

However, “:” is **not** associative:

- A *list of integers*:

$$1 : (2 : [3,4]) = 1 : 2 : [3,4] = [1, 2, 3, 4]$$

- $(1 : 2) : [3,4]$ is **nonsense**, since 2 is not a list!

- A *list of lists of integers*:

$$[2] : [[3,4,5], [6,7]] = [[2],[3,4,5],[6,7]]$$

- Another *list of lists of integers*:

$$(1 : [2]) : [[3,4,5], [6,7]] = [[1,2],[3,4,5],[6,7]]$$

- $1 : ([2] : [[3,4,5], [6,7]])$ is **nonsense** again!

Reason: 1 and [2] cannot be members of the same list (*type error*).

List Comprehensions

General shape:

$$[\textit{term} \mid \textit{generator} \{ , \textit{generator_or_constraint} \}^*]$$

Examples:

$$[n * n \mid n \leftarrow [1 .. 5]] = [1, 4, 9, 16, 25]$$

$$[n * n \mid n \leftarrow [1 .. 10], \textit{even} \ n] = [4, 16, 36, 64, 100]$$

$$[m * n \mid m \leftarrow [1, 3, 5], n \leftarrow [2, 4, 6]] = [2, 4, 6, 6, 12, 18, 10, 20, 30]$$

Note:

- The left generator “generates slower”.
- Haskell code fragments will frequently be presented like above in a form that is more readable than plain typewriter text — in that case, the “comes from” arrow “<-” in generators turns into “←”

Important Points

- Execution of Haskell programs **is** expression evaluation
- Defining functions in Haskell is more like defining functions in mathematics than like defining procedures in C or classes and methods in Java
- One Haskell function may be defined by several “equations” — the first that matches is used.
- Lists are an easy-to-use datastructure with lots of language and library support.

For this reason, lists are heavily used especially in beginners’ material.

In many cases, advanced Haskell programmers will use other datastructures, for example *FiniteMaps* instead of association lists.

Overview — Next Part

- Types
- Operations on Functions
- Functions over List

The Type Language

Haskell has a full-fledged **type language**, with

- Simple predefined datatypes: `Bool`, `Char`, `Integer`, ...
- Predefined **type constructors**: lists, tuples, functions, ...
- Type synonyms
- User-defined datatypes and type constructors
- Type variables — to express **parametric polymorphism**
- ...

Simple Predefined Datatypes

Bool	truth values	False, True
Char	“Unicode” characters	(in GHC: ISO-10646)
Integer	integers	arbitrary precision
Int	“machine integers”	≥ 32 bits
Float	real floating point	single precision
Double	real floating point	double precision
Complex Float	complex floating point	single precision
Complex Double	complex floating point	double precision

List Types

If t is a type, then the **list type** $[t]$ is the type of **lists** with elements of type t .

```
answer :: Integer
answer = 42

limit :: Int
limit = 100
```

Then:

```
• [ 1, 2, 3, answer ] :: [Integer]
• [ 1 .. limit ] :: [Int]
• [ [ 1 .. limit ], [ 2 .. limit ] ] :: [[Int]]
• [ 'h', 'e', 'l', 'l', 'o' ] :: [Char]
• "hello" :: [Char]
• [ "hello", "world" ] :: [[Char]]
• [ ["first", "line"], ["second", "line"] ] :: [[[Char]]]
```

Product Types (Pairs)

If t and u are types, then the **product type** (t, u) is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- $(\text{answer}, \text{limit}) :: (\text{Integer}, \text{Int})$
- $(\text{limit}, \text{answer}) :: (\text{Int}, \text{Integer})$
- $(\text{"???"}, \text{answer}) :: ([\text{Char}], \text{Integer})$
- $(\text{"???"}, (\text{limit}, \text{answer})) :: ([\text{Char}], (\text{Int}, \text{Integer}))$
- $(\text{"???"}, 'X') :: ([\text{Char}], \text{Char})$
- $(\text{limit}, (\text{"???"}, 'X')) :: (\text{Int}, ([\text{Char}], \text{Char}))$
- $(\text{True}, [(\text{"X"}, \text{limit}), (\text{"Y"}, 5)]) :: (\text{Bool}, [[\text{Char}], \text{Int}])$

Tuple Types

If $n \neq 1$ is a natural number and t_1, \dots, t_n are types, then the **tuple type** (t_1, \dots, t_n) is the type of n -**tuples** with the i th component of type t_i .

Examples:

- $(\text{answer}, 'c', \text{limit}) :: (\text{Integer}, \text{Char}, \text{Int})$
- $(\text{answer}, 'c', \text{limit}, \text{"all"}) :: (\text{Integer}, \text{Char}, \text{Int}, [\text{Char}])$
- $() :: ()$

— there is exactly one **zero-tuple**.

The type $()$ of zero-tuples is also called the **unit type**.

Simple Type Synonyms

If t is a type not containing any type variables, and $Name$ is an identifier with a capital first letter, then

`type Name = t`

defines $Name$ as a **type synonym** for t , i.e., $Name$ can now be used interchangeably with t .

Examples:

```
type String = [Char]           -- predefined
type Point = (Double, Double) -- (1.5, 2.7)
type Triangle = (Point, Point, Point)
type CharEntity = (Char, String) -- ('Ã¼', "&uuml;")
type Dictionary = [(String,String)] -- [("day", "jour")]
```

Type Variables and Polymorphic Types

- Identifiers with lower-case first letter can be used as type variables.
- Type variables can be used like other types in the construction of types, e.g.:

```
[(a,b)]
(Bool, (a, Int))
[(String, [(key, val)])]
```

- A type containing at least one type variable is called **polymorphic**
- Polymorphic types can be instantiated by instantiating type variables with types, e.g.:

```
[(a,b)]   => [(Char,b)]
[(a,b)]   => [(Char,Int)]
[(a,b)]   => [(a,[(String,Int)])]
[(a,b)]   => [(a,[(String,c)])]
```

Typing of List Construction

- The empty list can be used at any list type: `[] :: [a]`
- If an element $x :: a$ and a list $xs :: [a]$ are given, then

$$(x : xs) :: [a]$$

Examples:

```
2 :: Int
[] :: [Int]
[2] = 2 : [] :: [Int]
[[3,4,5], [6,7]] :: [[Int]]
[2] : [[3,4,5], [6,7]] :: [[Int]]
1 : ([2] : [[3,4,5], [6,7]]) -- cannot be typed!
```

Function Types and Function Application

If t and u are types, then the **function type** $t \rightarrow u$ is the type of all **functions** accepting arguments of type t and producing results of type u (mathematically: $t \rightarrow u$).

Then:

- If a function $f :: a \rightarrow b$ and an argument $x :: a$ are given, then we have $(f x) :: b$.
- If a function $f :: a \rightarrow b$ is given and we know that $(f x) :: b$, then the argument x is used at type a .
- If an argument $x :: a$ is given and we know that $(f x) :: b$, then the function f is used at type $a \rightarrow b$.

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x
```

```
fst ('c', False) :: Char
```

```
["hello", fst (x, 17)] => x :: String
```

```
f p = limit + fst p      => p :: (Int,a)
                        f :: (Int,a) -> Int
```

```
g h = fst (h "") : [limit]
    => h :: String -> (Int,a)
```

Let's Play the Evaluation Game Again — 1

```
h1 :: String -> (Int, String)
h1 str = (length str, ' ' : str)
```

```
g h = fst (h "") : [limit]
```

Then:

```
g h1
= fst (h1 "") : [limit]
= fst (length "", ' ' : "") : [limit]
= length "" : [limit]
= 0 : [limit]
= [0, 100]
```

Let's Play the Evaluation Game Again — 2

```
h2 :: String -> (Int, Char)
h2 str = (sum (map ord (notOccCaps str)), head str)
```

```
notOccCaps :: String -> String
notOccCaps str = filter ('notElem' str) ['A' .. 'Z']
```

```
g h = fst (h "") : [limit]
```

Then:

```
g h2
= fst (h2 "") : [limit]
= fst (sum (map ord (notOccCaps "")), head "") : [limit]
= sum (map ord (notOccCaps "")) : [limit]
= ...
= 2015 : [limit]
= [2015, 100]
```

Higher-Order Functions

```
g h = fst (h "") : [limit]
```

Functional Programming: Functions are first-class citizens

- Functions can be **arguments of other functions**: `g h2`
- Functions can be **components of data structures**: `(7,h1), [h1, h2]`
- Functions can be **results of function application**: `succ . succ`

A **first-order function** accepts only non-functional values as arguments.

A **higher-order function** expects functions as arguments.

`g` is a second-order function: it expects first-order functions like `h1, h2` as arguments.

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x
```

```
fst ('c', False) :: Char
```

```
["hello", fst (x, 17)] => x :: String
```

```
f p = limit + fst p      => p :: (Int,a)
                        f :: (Int,a) -> Int
```

```
g h = fst (h "") : [limit]
=> h :: String -> (Int,a)
   g :: (String -> (Int,a)) -> [Int]
```

Curried Functions

- **Function application associates to the left**, i.e.,

$$f\ x\ y = (f\ x)\ y$$

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., “they accept their arguments one at a time”:

```
cylVol r h = (pi :: Double) * r * r * h
```

Since the right-hand side, `r`, and `h` obviously all have type `Double`, we have;

```
(cylVol r) :: Double -> Double
cylVol     :: Double -> (Double -> Double)
```

- **Function type construction associates to the right**, i.e.,

$$a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)$$

“Partial Application”

Let values with the following types be given:

```
f :: a -> b -> c
x :: a
y :: b
```

The type of `f` is the function type `a -> (b -> c)`, with

- argument type `a`,
- result type `b -> c`.

Therefore, we can apply `f` to `x` and obtain:

```
(f x) :: b -> c
```

The application of a “two-argument function” to a single argument is a “one-argument function”, which can then be applied to a second argument:

```
(f x) y :: c = f x y
```

Partial Application — Example

```
g :: (String -> (Int, a)) -> [Int]
g h = fst (h "") : [limit]
```

```
k :: Int -> String -> (Int, String)
k n str = (n * (length str + 1), unwords (replicate n str))
```

```
g (k 3)
= fst (k 3 "") : [limit]
= fst (3 * (length "" + 1), unwords (replicate 3 "")) : [limit]
= (3 * (length "" + 1)) : [limit]
= (3 * (0 + 1)) : [limit]
= (3 * 1) : [limit]
= 3 : [limit]
= [3, 100]
```

Operations on Functions

```

id :: a -> a           -- identity function
id x = x

(.) :: (b -> c) -> (a -> b) -> (a -> c) -- function composition
(f . g) x = f (g x)

flip :: (a -> b -> c) -> (b -> a -> c)  -- argument swapping
flip f x y = f y x

curry :: ((a,b) -> c) -> (a -> b -> c)  -- currying
curry g x y = g (x,y)

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y) = f x y

```

Exercise (*necessary!*): Copy only the definitions to a sheet of paper, and then infer the types yourself!

Operator Sections

- Infix operators are turned into functions by surrounding them with parentheses:

```
(+) 2 3 = 2 + 3
```

- This is necessary in type declarations:

```

(+)  :: Int -> Int -> Int    -- not the “natural” type of (+)
(:)  :: a  -> [a] -> [a]
(++) :: [a] -> [a] -> [a]

```

- It is also possible to supply only one argument (which has to be an atomic expression):

```

(2 +) 3    = 2 + 3    = (+ 3) 2
(8.3 /) 2.5 = 8.3 / 2.5 = (/ 2.5) 8.3
(7 :) []   = 7 : []   = (: []) 7
((2^17) :) (16:[]) = (2^17) : 16 : [] = (: (16:[])) (2^17)

```

Turning Functions into Infix Operators

Surrounding a function name by **backquotes** turns it into an infix operator.

Frequently used examples (not the “natural” types throughout):

```

div, mod, max, min :: Int -> Int -> Int
elem :: Int -> [Int] -> Bool

```

```

12 `div` 7      = 1
12 `mod` 7      = 5
12 `max` 7      = 12
12 `min` 7      = 7
12 `elem` [1 .. 10] = False

```

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

```

null    :: [a] -> Bool
null [] = True
null (x : xs) = False

```

```

head    :: [a] -> a
head (x : xs) = x

```

```

tail    :: [a] -> [a]
tail (x : xs) = xs

```

(head and tail are **partial functions** — both are undefined on the empty list.)

Defining Functions Over Lists by Structural Induction

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

```
length      :: [a] → Int      concat :: [[a]] → [a]
length []   = 0               concat []   = []
length (x : xs) = 1 + length xs concat (xs : xss) = xs ++ concat xss
```

```
(++)      :: [a] → [a] → [a]  sum []      = 0
[] ++ ys = ys                 sum (x : xs) = x + sum xs
(x : xs) ++ ys = x : (xs ++ ys)

product []      = 1
product (x : xs) = x * product xs
```

```
x 'elem' []      = False
x 'elem' (y : ys)
  = x == y || x 'elem' ys
```

(All these functions are in the standard prelude.)

Unfolding Definitions

A simple definition:

```
limit = 10 ^ 2
```

Expanding this definition:

```
4 * (limit + 1)
= 4 * ((10 ^ 2) + 1)
= ...
```

Another definition:

```
concat = foldr (++) []
```

Expanding this definition:

```
concat [[1,2,3],[4,5]]
= (foldr (++) []) [[1,2,3],[4,5]]
= ...
```

Guarded Definitions

```
sign x | x > 0 = 1
       | x == 0 = 0
       | x < 0 = -1

choose :: Ord a => (a, b) → (a, b) → b
choose (x, v) (y, w)
  | x > y   = v
  | x < y   = w
  | otherwise = error "I cannot decide!"
```

If no guard succeeds, the next pattern is tried:

```
take_while p (x : xs) | p x = x : take_while p xs
take_while p xs      = []
```

```
take_while (< 5) [1, 2, 3]
= take_while (< 5) (1 : 2 : 3 : [])
= 1 : take_while (< 5) (2 : 3 : [])
= 1 : 2 : take_while (< 5) (3 : [])
= 1 : 2 : 3 : take_while (< 5) []
= 1 : 2 : 3 : []
= [1, 2, 3]
```

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

```
take_while p (x : xs) | p x = x : take_while p xs
take_while p xs      = []
```

```
take_while (< 5) [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6]
= take_while (< 5) (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : take_while (< 5) (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : take_while (< 5) (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : take_while (< 5) (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : take_while (< 5) (3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : 3 : take_while (< 5) (4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : 3 : 4 : take_while (< 5) (3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : take_while (< 5) (4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : take_while (< 5) (5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : []
= [1, 2, 3, 2, 3, 4, 3, 4]
```

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
```

```
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!"
```

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*:

<pre>if condition then expr1 else expr2</pre>	≡	<pre>case condition of True -> expr1 False -> expr2</pre>
---	---	---

Two ways of defining functions:

Pattern Matching

```
not True = False
not False = True
```

case

```
not b = case b of
  True -> False
  False -> True
```

case Expressions are “Anonymous” Pattern Matching

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

where Clauses

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

```
commaWords :: [String] -> String
commaWords [] = []
commaWords (x : xs) = x ++ commaWordsAux xs
  where
    commaWordsAux [] = []
    commaWordsAux xs = ", " : commaWords xs
```

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

where clauses are visible within all guards:

```
f x y | y > z = ...
      | y == z = ...
      | y < z = ...
  where z = x * x
```

let Expressions

Local definitions can also be part of expressions:

```
f k n = let m = k `mod` n
        in if m == 0
           then n
           else f n m
```

```
h x y = let x2 = x * x
        y2 = y * y
        in sqrt (x2 + y2)
```

Definitions can use **pattern bindings**:

```
g k n = let (d,m) = divMod k n
        in if d == 0
           then [m]
           else g d n ++ [m]
```

Guards, let and where bindings, and case cases all are **layout sensitive!**

let or where?

- let *bindings* in *expression* is an **expression**
- *fname patterns guardedRHSs* 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

Some Prelude Functions — Elementary List Access

```
head :: [a] -> a
head (x:_) = x

last :: [a] -> a
last [x] = x
last (_:xs) = last xs

tail :: [a] -> [a]
tail (_:xs) = xs

init :: [a] -> [a]
init [x] = []
init (x:xs) = x : init xs

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

Some Prelude Functions — List Indexing

```
length :: [a] -> Int
length = foldl' (\n _ -> n + 1) 0

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

Some Prelude Functions — Positional List Splitting

```

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

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

splitAt   :: Int -> [a] -> ([a], [a])
splitAt 0 xs = ([],xs)
splitAt _ [] = ([],[])
splitAt n (x:xs) | n>0 = (x:xs',xs'')
    where (xs',xs'') = splitAt (n-1) xs
splitAt _ _ = error "splitAt: negative argument"

```

Some Prelude Functions — Concatenation, Iteration

```

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

concat :: [[a]] -> [a]
concat = foldr (++) []

iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

repeat :: a -> [a]
repeat x = xs where xs = x:xs
{- repeat x = x : repeat x -}      -- for understanding

replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)

cycle :: [a] -> [a]
cycle xs = xs' where xs' = xs ++ xs'

```

Separation of Concerns: Generation and Consumption

```

replicate 3 '!'
= take 3 (repeat '!')           -- replicate
= take 3 ('!' : repeat '!')    -- repeat
= '!' : take (3 - 1) (repeat '!') -- take (iii)
= '!' : take 2 (repeat '!')    -- subtraction
= '!' : take 2 ('!' : repeat '!') -- repeat
= '!' : '!' : take (2 - 1) (repeat '!') -- take (iii)
= '!' : '!' : take 1 (repeat '!') -- subtraction
= '!' : '!' : take 1 ('!' : repeat '!') -- repeat
= '!' : '!' : '!' : take (1 - 1) (repeat '!') -- take (iii)
= '!' : '!' : '!' : take 0 (repeat '!') -- subtraction
= '!' : '!' : '!' : []        -- take (i)
= "!!!"

```

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference
- **Operator precedence rules:** juxtaposition as operator, “associate to the left/right”
- **Argument passing:** not by value or reference, but by name
- **Powerful datatypes** with simple interface: *Integer*, lists, lists of lists of ...
- **Non-local control** (evaluation on demand): modularity (e.g., generate / prune)

Exercise: Splitting with Predicates

- $takeWhile :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$
 $takeWhile$, applied to a predicate p and a list xs , returns the longest prefix (possibly empty) of xs of elements that satisfy p .
- $dropWhile :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$
 $dropWhile p xs$ returns the suffix remaining after $takeWhile p xs$.

Laws:

- $takeWhile p xs ++ dropWhile p xs = xs$
- $all p (takeWhile p xs) = True$
- $null (dropWhile p xs) \parallel p (head (dropWhile p xs))$

— if p is total (on xs).

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

as-Patterns

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

Consider matching of the third clause against $dropWhile (< 5) [1,2,3]$:

- $p = (< 5)$
- $xs = [1,2,3]$
- $x = 1$
- $xs' = [2,3]$
- $p x = (< 5) 1 = 1 < 5 = True$

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

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

```
break p        = span (not . p)
```

as-Patterns — 2

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

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

- $p = (< 5)$
- $xs = [5,4,3]$
- $x = 5$
- $xs' = [4,3]$
- $p x = (< 5) 5 = 5 < 5 = False$

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

map and filter

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

```
filter :: (a -> Bool) -> ([a] -> [a])
filter p [] = []
filter p (x : xs) = if p x then x : rest else rest
  where rest = filter p xs
```

These functions could also be defined via list comprehension:

```
map f xs = [ f x | x <- xs ]
filter p xs = [ x | x <- xs, p x ]
```

Examples:

```
map (7 *) [1 .. 6] = [7, 14, 21, 28, 35, 42]
filter even [1 .. 6] = [2, 4, 6]
```

foldr1

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 (⊗) [x] = x
foldr1 (⊗) (x:xs) = x ⊗ (foldr1 (⊗) xs)
```

```
foldr1 (⊗) [x1, x2, x3, x4, x5 ]
= x1 ⊗ (foldr1 (⊗) [x2, x3, x4, x5 ])
= x1 ⊗ (x2 ⊗ (foldr1 (⊗) [x3, x4, x5 ]))
= x1 ⊗ (x2 ⊗ (x3 ⊗ (foldr1 (⊗) [x4, x5 ])))
= x1 ⊗ (x2 ⊗ (x3 ⊗ (x4 ⊗ (foldr1 (⊗) [x5 ]))))
= x1 ⊗ (x2 ⊗ (x3 ⊗ (x4 ⊗ x5 )))
```

foldr

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

```
foldr (⊗) z [x1, x2, x3, x4, x5 ]
= x1 ⊗ (foldr (⊗) z [x2, x3, x4, x5 ])
= x1 ⊗ (x2 ⊗ (foldr (⊗) z [x3, x4, x5 ]))
= x1 ⊗ (x2 ⊗ (x3 ⊗ (foldr (⊗) z [x4, x5 ])))
= x1 ⊗ (x2 ⊗ (x3 ⊗ (x4 ⊗ (foldr (⊗) z [x5 ]))))
= x1 ⊗ (x2 ⊗ (x3 ⊗ (x4 ⊗ (x5 ⊗ (foldr (⊗) z [])))))
= x1 ⊗ (x2 ⊗ (x3 ⊗ (x4 ⊗ (x5 ⊗ z))))
```

Exercise: zipWith

- $zip :: [a] \rightarrow [b] \rightarrow [(a, b)]$
 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.
- $zipWith :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$
 $zipWith$ 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.
- $diagonal :: [[a]] \rightarrow [a]$
interprets its argument as a matrix, which may be assumed to be square, and returns the main diagonal of that matrix, e.g.:
 $diagonal [[1,2,3],[4,5,6],[7,8,9]] = [1,5,9]$

Defining Functions Over Lists by Structural Induction

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

$length :: [a] \rightarrow Int$	$concat :: [[a]] \rightarrow [a]$
$length [] = 0$	$concat [] = []$
$length (x : xs) = 1 + length xs$	$concat (xs : xss) = xs ++ concat xss$
$(++) :: [a] \rightarrow [a] \rightarrow [a]$	$sum :: Num a \Rightarrow [a] \rightarrow a$
$[] ++ ys = ys$	$sum [] = 0$
$(x : xs) ++ ys = x : (xs ++ ys)$	$sum (x : xs) = x + sum xs$
$elem :: Eq a \Rightarrow a \rightarrow [a] \rightarrow Bool$	$product :: Num a \Rightarrow [a] \rightarrow a$
$x \text{ 'elem' } [] = \text{False}$	$product [] = 1$
$x \text{ 'elem' } (y : ys)$	$product (x : xs) = x * product xs$
$= x \equiv y \parallel x \text{ 'elem' } ys$	

(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**:

$length :: [a] \rightarrow Int$	$concat :: [[a]] \rightarrow [a]$
$length = foldr (const (1+)) 0$	$concat = foldr (++) []$
$(++) :: [a] \rightarrow [a] \rightarrow [a]$	$sum :: Num a \Rightarrow [a] \rightarrow a$
$xs ++ ys = foldr (:) ys xs$	$sum = foldr (+) 0$
$elem :: Eq a \Rightarrow a \rightarrow [a] \rightarrow Bool$	$product :: Num a \Rightarrow [a] \rightarrow a$
$elem x = foldr (\lambda y r \rightarrow x \equiv y \parallel r) \text{False}$	$product = foldr (*) 1$

(All these functions are in the standard prelude.)

List Folding

foldr abstracts structural induction over lists!

$foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
$foldr f z [] = z$
$foldr f z (x:xs) = f x (foldr f z xs)$
$foldr1 :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
$foldr1 f [x] = x$
$foldr1 f (x:xs) = f x (foldr1 f xs)$
$foldl :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$
$foldl f z [] = z$
$foldl f z (x:xs) = foldl f (f z x) xs$
$foldl1 :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
$foldl1 f (x:xs) = foldl f x xs$

Lambda-Abstraction

Named functions:

```
add1 x = x + 1

recip x = 1 / x

square x = x * x
```

Anonymous functions:

```
(+ 1)

(1 /)

\ x -> x * x

\ x -> x * x
```

In “ $\lambda x \rightarrow body$ ”, the variable x is **bound**.

Typing rule:

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

Evaluation rule: Sym beta-reduction uses substitution:

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

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.

Some Prelude Functions — Text Processing

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

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

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:

```
not False = True
not True  = False
```

```
lexicalCombineOrdering :: Ordering → Ordering → Ordering
lexicalCombineOrdering LT _ = LT
lexicalCombineOrdering EQ x = x
lexicalCombineOrdering GT _ = GT
```

Simple data Type Definitions

```
data Point = Pt Int Int    deriving (Eq)    -- screen coordinates
data Transport = Feet
                | Bike
                | Train Int    -- price in cent
```

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

```
Pt :: Int → Int → Point
Feet :: Transport
Bike :: Transport
Train :: Int → Transport
```

Pattern matching:

```
addPt (Pt x1 y1) (Pt x2 y2) = Pt (x1 + x2) (y1 + y2)

cost Feet = 0
cost Bike = 0
cost (Train Int) = Int
```

Simple Polymorphic data Type Definitions

The prelude **type constructors** *Maybe*, *Either*, *Complex* are defined as follows:

data *Maybe* *a* = *Nothing* | *Just* *a* **deriving** (*Eq*, *Ord*, *Read*, *Show*)

data *Either* *a* *b* = *Left* *a* | *Right* *b*

data *Complex* *r* = *r* :+: *r* **deriving** (*Eq*, *Read*, *Show*)

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

Nothing :: *Maybe* *a*

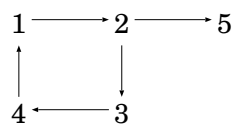
Just :: *a* → *Maybe* *a*

Left :: *a* → *Either* *a* *b*

Right :: *b* → *Either* *a* *b*

(*:+*) :: *r* → *r* → *Complex* *r*

Edge Graphs

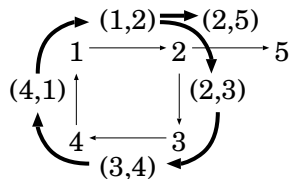


type *Graph* *a* = [(*a*, *a*)]

Define *edgeGraph* :: *Eq* *a* ⇒ *Graph* *a* → *Graph* (*a*, *a*) such that *edgeGraph* *g* returns the *edge graph* of *g*. This edge graph has edges of *g* as nodes, and has an edge from *e1* to *e2* iff the end node of *e1* is equal to the start node of *e2* (as edges in *g*).

edgeGraph :: *Eq* *a* ⇒ *Graph* *a* → *Graph* (*a*, *a*)

```
edgeGraph g =
  [(e1,e2)
  | e1@(_,x) <- g
  , e2@(y,_) <- g
  , x == y]
```



Side Effects and Haskell

- Haskell is **pure**:
 - Evaluating expressions has **no side-effects**
 - Expressions are evaluated only for obtaining their **values**
- But sometimes we want our programs to affect the real world (printing, controlling a robot, drawing a picture, etc).

How do we reconcile these two aspects?

In Haskell, certain “pure values” are “worldly actions” that can be *performed*

- **Types:** An expression with type *IO a* has as its value a **computation** (in the *IO-monad*) that can be understood as returning a value of type *a*.

Alternative explanation: An expression with type *IO a* has possible *actions* associated with its execution, while returning a value of type *a*

- **Syntax:** The **do** syntax sequences several actions (using layout)

The do Syntax

- *readFile* *"/etc/passwd"* :: *IO String* is an action.
- We use the **do** syntax to bind the result of that action to the variable *s*, and sequence this action with other actions that depend on *s*:

main = **do**

```
s ← readFile "/etc/passwd"
putStrLn $ "/etc/passwd has " ++ shows (length s) " characters"
let logins = map (takeWhile (':' ≠)) $ lines s
putStrLn $ "There are " ++ shows (length logins) " logins"
let funny = filter (all (notElem "AEIOUaeiou")) logins
putStrLn $ "Funny logins:" ++ concatMap (',' :) funny
```

- Inside **do**, one may write **let** without **in**.

When IO Actions are Performed

An expression with type `IO a` has as its value a **computation** that, when performed, may return a value of type `a`.

- A value of type `IO a` is an **action**, but it is still a *value*: it will **only** have an **effect** when it is **performed**.
- In Haskell, a program's value is the value of the variable `main` in the module `Main`.
That value has to have type `IO a`.
It will be **performed** upon execution of the program.
- In Hugs and GHCi, you can type any expression to the prompt.
If the expression has type `IO a` it will be performed; otherwise its value will be printed on the display.

Predefined IO Actions

-- get one character from keyboard
`getChar :: IO Char`

-- write one character to terminal
`putChar :: Char → IO ()`

-- write a string to terminal (without/with adding a newline)
`putStr, putStrLn :: String → IO ()`

-- get a whole line from keyboard
`getLine :: IO String`

-- read a file as a `String`
`readFile :: FilePath → IO String`

-- write a `String` to a file
`writeFile :: FilePath → String → IO ()`

IO Example

module `Main (main) where` -- this is the default module header

```
main = do
  line ← getLine
  let ws = words line
  case ws of
    [] → return ()
    _ → do
      putStrLn ("You entered " ++ show (length ws) ++ " words")
      main
```

Compile and run:

```
ghc --make -o WC WC.hs
./WC
```

Another IO Example

import qualified `System` -- `Cat.hs`

```
main = do
  args ← System.getArgs
  putStrLn (shows (length args) " arguments")
  let (flags, files) = span ("-"≡) ∘ take 1 args
  print flags
  mapM (λ file → readFile file >>= putStrLn) files
```

Compile and run:

```
ghc --make -o Cat Cat.hs
./Cat -flag1 -q -v -flag4 file1 qwerty -what file4
```

Recursive Actions

`getLine` can be defined recursively in terms of simpler actions:

```
getLine :: IO String
getLine =
  do c <- getChar           -- get a character
     if c == '\n'          -- if it's a newline
     then return ""        -- then return empty string
     else do l <- getLine  -- otherwise get rest of
                        -- line recursively,
                        return (c:l) -- and return entire line
```

The function `return :: a → IO a` takes a value of type `a`, and turns it into an action of type `IO a`, which does nothing but return the value.

Exercise 4.1 (a)

```
module StackMachine where
import System.IO
```

We define a type of transition functions that define state transitions triggered by *inputs* and also producing *outputs*:

```
type Transition state input output = (state, input) → (state, output)
```

Define a Haskell function

```
process :: Transition state input output → state → [input] → [output]
```

that calculates the list of outputs produced by a transition function given a starting state and a list of inputs.

```
process tr s [] = []
process tr s (input : inputs) = let
  (s', output) = tr (s, input)
in output : process tr s' inputs
```

Exercise 4.1 (b)

Using `process` from (b) and prelude functions, the function

```
runprocess :: Transition state String String → state → IO ()
runprocess tr s = do
  hSetBuffering stdout LineBuffering
  interact (unlines ∘ process tr s ∘ lines)
```

turns a transition with `String` inputs and outputs into a runnable program.

Try: `runprocess id 0`

Define a transition function

```
countEcho :: Transition Integer String String
```

that keeps a counter as its state and otherwise just reproduces the input prefixed with line numbers as output.

Try: `runprocess countEcho 0`

```
countEcho (count, input) = (count', shows count' ('' : input))
  where count' = succ count
```

Exercise 4.1 (c)

Define a transition function

```
trAdd :: Transition Integer String String
```

that uses the prelude functions `read` and `show` to add the `Integer` reading of the input to the accumulating state, and outputs that state as a string.

Try: `runprocess trAdd 0`

```
trAdd (s, input) = (s', show s')
  where
    n = read input
    s' = s + n
```

Exercise 4.1 (d)

Define a transition function

polish :: *Transition* [*Integer*] *String* *String*

that implements a reverse Polish notation calculator by pushing number inputs on the stack, always outputting the top of the stack (if present), and interpreting +, −, *, / as taking their arguments from the stack and pushing the result back onto the stack.

Try: *runprocess polish []*

```
polish ( n : m : ks, "+" ) = ( k : ks, show k ) where k = m + n
polish ( n : m : ks, "-" ) = ( k : ks, show k ) where k = m - n
polish ( n : m : ks, "*" ) = ( k : ks, show k ) where k = m * n
polish ( n : m : ks, "/" ) = ( k : ks, show k ) where k = m `div` n
polish ( ks      , input ) = ( k : ks, show k ) where k = read input
```

Exercise 7.1

```
let k = 3 * 4
> k = 12
let n = 3 * (k + 2)
> n = 42
n - k
> 30
n + m
> error: Variable "m" undefined!
```