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/

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)
```


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
```


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 .. 400]`
- and a result is returned only if *n* divides `11111`

Expanding Function Definitions

```
perimeter :: Double -> Double
perimeter r = 2 * r * pi

square :: Integer -> Integer
square x = x * x
```

```
   perimeter (1 + 2)
 = 2 * (1 + 2) * pi
 = 2 * 3 * pi
 = 6 * pi
 = 18.84955592153876

   square (1 + 2)
 = (1 + 2) * (1 + 2)
 = 3 * 3
 = 9
```


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)
```


Simple Expression Evaluation — Explanation

- Arguments to a fuction 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 $f\ \mathit{undefined} = \mathit{undefined}$.
  - **Constant functions** are **non-strict**: $(\mathit{const}\ 5)\ \mathit{undefined} = 5$
  - Checking a list for emptyness is **strict:** $\mathit{null}\ \mathit{undefined} = \mathit{undefined}$
  - **List construction** is **non-strict**: $\mathit{null}\ (\mathit{undefined} : \mathit{undefined}) = \mathbf{False}$
  - **Standard arithmetic operators** are **strict in both arguments**: $0 * \mathit{undefined} = \mathit{undefined}$

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*.
- If the condition does not evaluate to anything, the conditional expression also does not evaluate to anything.

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
```


List Construction

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

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: “`x`es”)

`x : xs` — read: “`x` *cons* `x`es”.

“:” 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]

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
```


Cons is Not Associative

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

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:

[*term* | *generator* {, *generator_or_constraint* }*]

Examples:

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

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

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

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 *FiniteMap*s instead of association lists.

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"	$\geq$ 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:

- `(answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`
- `("???", answer) :: ([Char], Integer)`
- `("???", (limit, answer)) :: ([Char], (Int, Integer))`
- `("???", ’X’) :: ([Char], Char)`
- `(limit, ("???", ’X’)) :: (Int, ([Char], Char))`
- `(True, [("X",limit),("Y",5)]) :: (Bool, [([Char], 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 `,` … `,` t_n `)` is the type of n-**tuples** with the ith component of type t_i.

Examples:

- `(answer, ’c’, limit) :: (Integer, Char, Int)`
- `(answer, ’c’, limit, "all") :: (Integer, Char, Int, [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)]`	$\Rightarrow$	`[(Char,b)]`
`[(a,b)]`	$\Rightarrow$	`[(Char,Int)]`
`[(a,b)]`	$\Rightarrow$	`[(a,[(String,Int)])]`
`[(a,b)]`	$\Rightarrow$	`[(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`->`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 -> b` and an argument `x :: a` are given, then we have `(f x) :: b`.
- If a function `f :: a -> 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 -> 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.

`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]
```

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)
```


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 -> b -> c  =  a -> (b -> c)
```


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]
```


"Partial Application"

Let values with the following types be given:

$f :: a \rightarrow b \rightarrow c$
$x :: a$
$y :: b$

The type of f is the function type $a \rightarrow (b \rightarrow c)$, with

- argument type a,
- result type $b \rightarrow c$.

Therefore, we can apply f to x and obtain:

$(f\ x) :: b \rightarrow c$

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:

$(f\ x)\ y :: c \quad = \quad f\ x\ y$

Partial Application — Example

$g :: (String \to (Int, a)) \to [Int]$
$g\ h = fst\ (h\ "") : [limit]$

$k :: Int \to String \to (Int, String)$
$k\ n\ str = (n * (length\ str + 1), unwords\ (replicate\ n\ str))$

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

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!

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**:

$null :: [a] \to Bool$
$null\ [\,] = \mathbf{True}$
$null\ (x : xs) = \mathbf{False}$

$head :: [a] \to a$
$head\ (x : xs) = x$

$tail :: [a] \to [a]$
$tail\ (x : xs) = xs$

(`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
length []           =  0
length (x : xs)     =  1 + length xs

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

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

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

sum []         = 0
sum (x : xs)   = x + sum xs

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

(All these functions are in the standard prelude.)

Exercise: Positional List Splitting

- *take* :: *Int* → [*a*] → [*a*]

  take, applied to a *k* :: *Int* and a list *xs*, returns the longest prefix of *xs* of elements that has no more than *k* elements.

- *drop* :: *Int* → [*a*] → [*a*]

  drop k xs returns the suffix remaining after *take k xs*.

Laws:

- *take k xs* ++ *drop k xs* = *xs*
- *length* (*take k xs*) ≤ *k*

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

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 0 _ = []
take k _ | k < 0 = error "take: negative argument"
take k [] = []
take k (x : xs) =  x : take (k − 1) xs
```

```
take 2 [5, 6, 7]              = take 2 (5 : 6 : 7 : [])
= 5 : take (2 − 1) (6 : 7 : [])
= 5 : take 1 (6 : 7 : [])
= 5 : 6 : take (1 − 1) (7 : [])
= 5 : 6 : take 0 (7 : [])
= 5 : 6 : []              = [5, 6]
```