Distributed and Parallel Technology

Introducing Haskell

Hans-Wolfgang Loidl
http://www.macs.hw.ac.uk/~hwloidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

Semester 2 2016/17

[0]No proprietary software has been used in producing these slides

Haskell: yet another Functional Language

Haskell is a polymorphicly-typed, lazy, purely-functional language.
Hence Haskell is similar to SML, e.g.

Function definitions	
SML:	*Haskell:*
fun fac 1 = 1 |	fac 1 = 1
fac n = n*fac (n-1);	fac n = n*fac (n-1)

Evaluation

fac 5 $\Rightarrow$ *5 * 4 * 3 * 2 * 1*

Characteristics of Functional Languages

Like other modern functional languages e.g. F# or Racket, Haskell includes *advanced features:*

- Sophisticated polymorphic type system, with type inference.

  $length :: [a] \rightarrow Int$

- Pattern matching.

  $length :: [a] \rightarrow Int$
  $length\ [] = 0$
  $length\ (x : xs) = 1 + length\ xs$

- Higher-order functions.

  $map\ (*2)\ [1, 2, 3, 4]$

- Data abstraction.

  data *MyList a* = *Nil*
  | *Cons a* (*MyList a*)

- Garbage-collected storage management.

Characteristics of Functional Languages (cont'd)

These features are also found in modern object-oriented languages.
The distinctive feature of *pure* functional languages is their *referential transparency*.

Definition (Stoy, 1977)

The only thing that matters about an expression is its value, and *any subexpression can be replaced by any other equal in value*. Moreover, the value of an expression is, within certain limits, the same wherever it occurs.

Implications:

- Two expressions are *equal* if they have the same value, e.g. sin(6) = sin(1+5).
- Value-based equality enables *equational reasoning*, where one expression is substituted by another of equal value, e.g. f(x) + f(x) = 2*f(x)
- Scope matters: if x = 6, then sin(x) = sin(6) = sin(1+5)

Referentially Opaque Notations

Example

English:

> *"Eric the Red was so called because of his red beard"*

Cannot substitute "Eric Jarlsson" for "Eric the Red" and retain the meaning.

Procedural programming languages: `x = x + 1`

Exercise

*In C, does replacing the sum of two function calls: f(x) + f(x), by 2*f(x) preserve the meaning of the program? If not, give an example function, f that should not be substituted.*

Referentially Transparent Notations

Most of mathematics is referentially transparent.

- integer algebras: $2x + x = 12$.
- relational algebra: $R \cup S = S \cup R$
  As *SQL* is based on the relational algebra, a large subset of it is referentially transparent, except for parts that change the database, e.g. UPDATE, DELETE etc.
- Mathematical logics: $P(x) \wedge Q(x)$.
  As a result, a large part of most logic, or deductive, languages is referentially transparent, except for features like CUT, ASSERT and RETRACT.
- *Purely functional languages*

Consequences of Referential Transparency

Equational reasoning

- Proofs of correctness are much easier than reasoning about state as in procedural languages.
- Used to *transform* programs, e.g. to transform simple specifications into efficient programs.

Freedom from execution order.

- Meaning of program is not dependent on execution order: so programs are shorter as the programmer doesn't have to specify an execution order.
- *Lazy evaluation:* Most languages have a *strict* evaluation order, e.g. evaluate the parameters to a function left-to-right before calling it.
  A lazy language only evaluates an expression when, and if, it's needed.
- *Parallel/distributed evaluation*. Often there are many expressions that can be evaluated at a time, because we know that the order of evaluation doesn't change the meaning, the sub-expressions can be evaluated in parallel (Wegner 1978)

Elimination of *side effects* (unexpected actions on a global state).

Downside of referential transparency

Interaction with state outside the program, e.g. reading from a file, updating a database is harder.
⇒ language needs special constructs for dealing with stateful objects, and Haskell uses *Monads*

No straightforward language support for in-place operations
⇒ libraries are provided that achieve in-place operations on certain data structures using a monadic style of programming.

Differences between SML and Haskell

Where SML is an imperative language with a large purely-functional subset, Haskell is (almost entirely) purely-functional, guaranteeing **referential transparency**.

Where SML is *strict* with a defined evaluation order, Haskell is *lazy*.

Other minor differences: different module systems, Haskell has parametric polymorphism (C#'s and Java's generics are based on it), etc.

Preliminaries

Basic types in Haskell:

- *Bool*: boolean values: *True* und *False*
- *Char*: characters
- *String*: strings (as list of characters)
- *Int*: fixed precision integers
- *Integer*: arbitrary precision integers
- *Float*: single-precision floating point numbers

Preliminaries

Compound types:

- *Lists*: $[\cdot]$, e.g. $[Int]$ list of (fixed precision) integers;
- *Tupels*: $(\cdot, \cdots)$, e.g. $(Bool, Int)$ tupel of boolean values and integers;
- *Records*: $\cdot\ \{\cdot, \cdots\}$, e.g. $Bl\ \{\ b :: Bool,\ i :: Int\ \}$ a record of boolean values and integers;
- *Functions*: $a \rightarrow b$, e.g. $Int \rightarrow Bool$

Typesynonyms can be defined like this:

type *IntList* $=$ $[Int]$

Haskell Types & Values

Note that all Haskell values are *first-class*: they may be passed as arguments to functions, returned as results, placed in data structures.

Example

Example Haskell values and types:

$$
\begin{array}{ll}
5 & :: Integer \\
'a' & :: Char \\
True & :: Bool \\
inc & :: Integer \rightarrow Integer \\
[1,2,3] & :: [Integer] \\
('b', 4) & :: (Char, Integer)
\end{array}
$$

N.B: The "::" can be read "has type."

Function Definitions

Functions are normally defined by a series of equations. Giving type signatures for functions is optional, but highly recommended.

$$inc :: Integer \to Integer$$
$$inc\ n = n + 1$$

To indicate that an expression `e1` evaluates to another expression `e2`, we write

Evaluation

`e1` ⇒`e2`

Evaluation

inc (inc 3) ⇒*5*

User-defined Types

Data constructors are introduced with the keyword **data**.
Nullary data constructors, or enumerated types:

$$\textbf{data}\ Bool = False \mid True$$
$$\textbf{data}\ Color = Red \mid Green \mid Blue$$
$$\textbf{data}\ Day = Mon \mid Tue \mid Wed \mid Thu \mid Fri \mid Sat \mid Sun$$

Pattern matching over such data types is frequently used to make a case distinction over a user-defined (algebraic) data-type:

$$\begin{aligned} next &:: Day \to Day \\ next\ Mon &= Tue \\ next\ Tue &= Wed \\ next\ Wed &= Thu \\ next\ Thu &= Fri \\ next\ Fri &= Sat \\ next\ Sat &= Sun \\ next\ Sun &= Mon \end{aligned}$$

User-defined Types

A recursive data constructor:

$$\textbf{data}\ Tree\ a = Leaf\ a \mid Branch\ (Tree\ a)\ (Tree\ a)$$

$$\begin{aligned} fringe &:: Tree\ a \to [a] \\ fringe\ (Leaf\ x) &= [x] \\ fringe\ (Branch\ left\ right) &= fringe\ left\ +\!+\ fringe\ right \end{aligned}$$

Here ++ is the infix operator that concatenates two lists.
N.B: type constructor names must be capitalised.
N.B: This is the same declaration in *SML:*

$$\begin{aligned} \textbf{datatype}\ 'a\ binTree &= leaf \\ &\mid node\ \textbf{of}\ 'a * 'a\ binTree * 'a\ binTree; \end{aligned}$$

Type Synonyms

Type synonyms are names for commonly used types, rather than new types, and defined with the keyword **type**:

$$\textbf{type}\ String = [Char]$$
$$\textbf{type}\ Person = (Name, Address)$$
$$\textbf{type}\ Name = String$$

$$\textbf{data}\ Address = None \mid Addr\ String$$

Syntactic support is provided for strings, e.g. `"bye"` ⇒`['b','y','e']`, and list operations operations can be applied to them, e.g. `length "bye"` ⇒`3`.

Pattern Matching

A pattern may contain a **wildcard**, e.g. to chose just the first `n` elements of a list,

Evaluation

take 2 [1,2,3] ⇒*[1,2]*

$$
\begin{aligned}
&take\ 0\ \_ &&= [] \\
&take\ \_\ [] &&= [] \\
&take\ n\ (x : xs) &&= x\ :\ take\ (n-1)\ xs
\end{aligned}
$$

Guarded Patterns

A pattern may contain a **guard**: a condition that must be true for the pattern to match, e.g.

$$
\begin{aligned}
sign\ x\ &| x > 0 &&= 1 \\
&| x == 0 &&= 0 \\
&| x < 0 &&= -1
\end{aligned}
$$

Lists

Constructed from cons (`:`) and nil (`[]`), e.g.

$1 : []$
$1 : 2 : 3 : []$
$'b' :' y' :' e' : []$

having types [*Integer*], [Integer] and [Char]. Lists are commonly abbreviated:

$[1]$
$[1, 2, 3]$
$['b','y','e']$

A list can be indexed with the `!!` operator:

Evaluation

$[1, 2, 3]$!! $0 \Rightarrow 1$
$['b','y','e']$!! $2 \Rightarrow 'e'$

List Comprehensions

"*List comprehensions*" are a short-hand notation for defining lists, with notation similar to set comprehension in mathematics. They have been introduced as ZF-comprehensions in Miranda, a pre-cursor of Haskell, and are also supported in Python.
Example: List of square values of all even numbers in a given list of integers *xs*:

$$sq\_even\ xs\ =\ [x * x \mid x\ \leftarrow\ xs,\ even\ x]$$ $$sq\_even\ xs\ =\ [\underline{x * x} \mid x\ \leftarrow\ xs,\ e$$

The expression $\underline{x * x}$ is the body of the list comprehension. It defines the value of each list element.
The expression $\underline{x \leftarrow xs}$ is the generator and binds the elements in *xs* to the new variable *x*, one at a time.
The condition $\underline{even\ x}$ determines, whether the value of *x* should be used in computing the next list element.

List comprehension example

List comprehensions enable list functions to be expressed concisely:

$$
\begin{aligned}
&\mathit{quicksort}\ [] \ = \ [] \\
&\mathit{quicksort}\ (x : xs) \ = \\
&\qquad \mathit{quicksort}\ [y \mid y \leftarrow xs,\ y < x\ ] \ ++ \\
&\qquad [x] \ ++ \\
&\qquad \mathit{quicksort}\ [y \mid y \leftarrow xs,\ y >= x]
\end{aligned}
$$

Polymorphic Functions

A *polymorphic function* (generic method in Java or C#) can operate on values of many types, e.g.

$$
\begin{aligned}
&\mathit{length} && :: [a] \rightarrow \mathit{Integer} \\
&\mathit{length}\ [] && = 0 \\
&\mathit{length}\ (x : xs) && = 1 \ + \ \mathit{length}\ xs
\end{aligned}
$$

Evaluation

$$
\begin{aligned}
&\mathit{length}\ [1,2,3] \Rightarrow 3 \\
&\mathit{length}\ ['b','y','e'] \Rightarrow 3 \\
&\mathit{length}\ [[1],[2]] \Rightarrow 2
\end{aligned}
$$

N.B: `a` is a type variable, that can stand for any type.

Local Definitions

Haskell, like SML has a mutually recursive **let** binding, e.g.

$$
\begin{aligned}
&\textbf{let}\ \ y && = a * b \\
&\quad\ \ f\ x && = (x + y)/y \\
&\textbf{in}\ \ \ f\ c \ + \ f\ d
\end{aligned}
$$

The Haskell **where** binding scopes over several guarded equations:

$$
\begin{aligned}
&f\ x\ y \ \ \mid y > z \ \ = ... \\
&\qquad\ \mid y == z = ... \\
&\qquad\ \mid y < z \ \ = ... \\
&\qquad\qquad\qquad \textbf{where}\ z \ = \ x * x
\end{aligned}
$$

Layout

Haskell lays out equations in columns to disambiguate between multiple equations, e.g. could previous definition be:

$$
\begin{aligned}
&\textbf{let}\ \ y \ = \ a * b\ f \\
&\quad\ \ x \ = \ (x + y)/y \\
&\textbf{in}\ \ \ f\ c \ + \ f\ d
\end{aligned}
$$

Key rule: declarations start to the right of **where** or **let**.

Curried & Infix Functions

Currying: a function requiring n arguments can be applied to fewer arguments to get another function, e.g.

$$\begin{aligned} &add\ x\ y = x + y \\ &inc :: Integer \to Integer \\ &inc = add\ 1 \end{aligned}$$

Example

Define the sum over list of squares over all even numbers:

$$sqs\_even :: [Integer] \to Integer$$

$$\begin{aligned} sqs\_even\ [] &= 0 \\ sqs\_even\ (x : xs) \mid even\ x &= x^2 + sqs\_even\ xs \\ \mid otherwise &= sqs\_even\ xs \end{aligned}$$

Higher Order Functions

Functions are first class values and can be *passed as arguments to other functions and returned as the result* of a function.
Many useful higher-order functions are defined in the Prelude and libraries, including most of those from your SML course, e.g.
filter takes a list and a boolean function and produces a list containing only those elements that return `True`

$$filter :: (a \to Bool) \to [a] \to [a]$$

$$\begin{aligned} filter\ p\ [] &= [] \\ filter\ p\ (x : xs) \mid p\ x &= x : filter\ p\ xs \\ \mid otherwise &= filter\ p\ xs \end{aligned}$$

Evaluation

filter even [1,2,3]⇒[2]

The higher-order function `map`

map applies a function *f* to every element of a list

$$\begin{aligned} &map :: (a \to b) \to [a] \to [b] \\ &map\ f\ [] = [] \\ &map\ f\ (x : xs) = (f\ x) : map\ f\ xs \end{aligned}$$

Evaluation

map inc [2,3]
⇒(inc 2) : map inc [3]
⇒(inc 2) : (inc 3) : map inc []
⇒(inc 2) : (inc 3) : []
...
⇒[3,4]

In general:

Evaluation

$$map\ f\ [x_0, x_1, \ldots, x_n] \Longrightarrow [f\ x_0, \ldots, f\ x_n]$$

map example

Example: sum over list of squares over all even numbers:

$$\begin{aligned} &sqs\_even :: [Integer] \to Integer \\ &sqs\_even\ xs = sum\ (map\ (\lambda x \to x * x)\ (filter\ even\ xs)) \end{aligned}$$

The higher-order function *foldr*

foldr applies a binary function to every element of a list:

$$\begin{aligned} &foldr :: (a \to b \to b) \to b \to [a] \to b \\ &foldr\ f\ z\ [] \quad\ = z \\ &foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs) \end{aligned}$$

Example:

Evaluation

foldr add 0 [2,3]
⇒*add 2 (foldr add 0 [3])*
⇒*add 2 (add 3 (foldr add 0 []))*
...
⇒*5*

In general: *foldr* replaces every : in the list by an f, and the [] by an v:

Evaluation

$foldr \oplus v (x_0 : \ldots : x_n : []) \Longrightarrow x_0 \oplus \ldots \oplus (x_n \oplus v)$

zip converts a pair of lists into a list of pairs:

$$zip :: [a] \to [b] \to [(a, b)]$$

Evaluation

zip [1,2,3] [9,8,7] ⇒*[(1,9),(2,8),(3,7)]*

`zipWith` takes a pair of lists and a binary function and produces a list containing the result of applying the function to each 'matching' pair:

Evaluation

$zipWith \oplus (x_0 : \ldots : x_n : []) (y_0 : \ldots : y_n : [])$
$\Longrightarrow (x_0 \oplus y_0) : \ldots : (x_n \oplus y_n)$

Example

```
dotProduct xs ys = sum (zipWith (*) xs ys)
```


Lambda Abstractions

Functions may be defined "anonymously" via a lambda abstraction (`fn` in SML). For example definitions like

$$\begin{aligned} &inc\ x \quad = x + 1 \\ &add\ x\ y = x + y \end{aligned}$$

are really shorthand for:

$$\begin{aligned} &inc = \backslash x \to x + 1 \\ &add = \backslash x\ y \to x + y \end{aligned}$$

Lambda Expressions	
SML:	*Haskell:*
`fn x => ...`	`\ x -> ...`

Infix Operators

Infix operators are really just functions, and can also be defined using equations, e.g. list concatenation:

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

and function composition:

$$\begin{aligned} &(.) :: (b \to c) \to (a \to b) \to (a \to c) \\ &f\ .\ g = \backslash x \to f\ (g\ x) \end{aligned}$$

Lexically, infix operators consist entirely of "symbols," as opposed to normal identifiers which are alphanumeric.

Function composition	
SML:	*Haskell:*
`f o g`	`f . g`

Sections

Since operators are just functions, they can be curried, e.g. (parentheses mandatory)

$$(x+) = \backslash y \rightarrow x + y$$
$$(+y) = \backslash x \rightarrow x + y$$
$$(+) = \backslash x\ y \rightarrow x + y$$

Example

The sum over list of squares of all even numbers:

$$sqs\_even :: [Integer] \rightarrow Integer$$
$$sqs\_even = foldr\ (+)\ 0\ .\ map\ (\lambda x \rightarrow x * x)\ .\ filter\ even$$

Lazy Functions

Most programming languages have *strict* semantics: the arguments to a function are evaluated before the evaluating the function body. This sometimes wastes work, e.g.

$$f\ True\ y = 1$$
$$f\ False\ y = y$$

It may even cause a program to fail when it could complete, e.g.

Evaluation

$$f\ True\ (1/0) \Rightarrow ?$$

It may even cause a program to fail when it could complete, e.g.

Evaluation

$$f\ True\ (1/0) \Rightarrow 1$$

"Infinite" Data Structures

As an example of lazy evaluation, consider the following function:

$$foo\ x\ y\ z = \textbf{if}\ \underline{x < 0}\ \textbf{then}\ abs\ x \textbf{ else } x + y$$
$$foo\ x\ y\ z = \textbf{if}\ x < 0\ \textbf{then}\ \underline{abs\ x} \textbf{ else } x + y$$
$$foo\ x\ y\ z = \textbf{if}\ x < 0\ \textbf{then}\ abs\ x \textbf{ else } \underline{x + y}$$

Evaluation order:

- Evaluating the conditional requires the evaluation of $\underline{x < 0}$ which in turn requires the evaluation of the argument x.
- If $x < 0$ is True, the value of $\underline{abs\ x}$ will be returned; in this case *neither* y *nor* z will be evaluated.
- If $x < 0$ id False, the value of $\underline{x + y}$ will be returned; this requires the evaluation of y.
- z won't be evaluated in either of these two cases.
- In particular, the expression $foo\ 1\ 2\ (1\ `div`\ 0)$ is well-defined.

Haskell functions are *lazy*: the evaluation of the arguments is delayed, and the body of the function is evaluated and only if the argument is

"Infinite" Data Structures

Data constructors are also lazy (they're just a special kind of function), e.g. list construction (`:`)

Non-strict constructors permit the definition of (conceptually) infinite data structures. Here is an infinite list of ones:

$$ones = 1 : ones$$

More interesting examples, successive integers, and all squares (using infix exponentiation):

$$numsFrom\ n = n : numsFrom\ (n + 1)$$
$$squares = map\ (\hat{}2)\ (numsFrom\ 0)$$

Any program only constructs a finite part of an infinite sequence, e.g.

Evaluation

take 5 squares ⇒*[0,1,4,9,16]*

Infinite Data-structures

The prelude function for selecting the n-the element of a list:

$$[] \,!!\, \_ = \mathit{error}\ \text{"}\mathit{Empty\ list}\text{"}$$
$$(x : \_) \,!!\, 0 = x$$
$$(\_ : xs) \,!!\, n = xs \,!!\, (n-1)$$

Here is the evaluation order of [0..]!!2:

Evaluation

[0..]!!2	$\Longrightarrow$	*is the list empty?*
(0 : [1..])!!2	$\Longrightarrow$	*is the index 0?*
[1..]!!1	$\Longrightarrow$	*is the list empty?*
(1 : [2..])!!1	$\Longrightarrow$	*is the index 0?*
[2..]!!0	$\Longrightarrow$	*is the list empty?*
(2 : [3..])!!0	$\Longrightarrow$	*is the index 0?*
2		

Example Sieve of Erathostenes

Compute a list of all prime numbers by,

1. starting with a list of all natural numbers,
2. take the first number in the list and cancel out all its multiples,
3. repeat Step 2 forever.

```
-- iteratively remove all multiples of identified prime numbers
sieve :: [Integer] -> [Integer]
sieve (p:xs) = p : sieve (removeMults p xs)

-- remove all multiples of a given number
removeMults :: Integer -> [Integer] -> [Integer]
removeMults p xs = [ x | x <- xs, not (x `rem` p == 0) ]

-- define an infinite list of prime numbers
primes :: [Integer]
primes = sieve [2..]
```


An example of an infinite data structure

The goal is to create a list of *Hamming numbers*, i.e. numbers of the form $2^i 3^j 5^k$ for $i, j, k \in \mathbb{N}$

```
hamming = 1 : map (2*) hamming `union`
              map (3*) hamming `union`
              map (5*) hamming

union a@(x:xs) b@(y:ys) = case compare x y of
            LT -> x : union  xs  b
            EQ -> x : union  xs  ys
            GT -> y : union  a   ys
```

Note, that `hamming` is an infinite list, defined as a cyclic data structure, where the computation of one element depends on prior elements in the list.

[0] Solution from Rosetta code

Normal Forms

Normal forms are defined in terms of reducable expressions, or **redexes**, i.e. expressions that can be simplified e.g. `(+) 3 4` is reducable, but `7` is not.
Strict languages like SML reduce expressions to *Normal Form (NF)*, i.e. until no redexes exist (they are "fully evaluated"). Example NF expressions:

$$5$$
$$[4, 5, 6]$$
$$\backslash x \rightarrow x + 1$$

Lazy languages like Haskell reduce expressions to *Weak Head Normal Form (WHNF)*, i.e. until no top level redexes exist. Example WHNF expressions:

$$(:)\ 2\ [2+1] \quad \textit{usually written as} \quad 2 : [2+1]$$

Examples

Example *non*-WHNF expressions:

$(+)\,4\,1$
$(\backslash x \rightarrow x + 1)\,3$

Example WHNF expressions that are *not in NF*:

$(*)\,((+)\,4\,1)$
$\backslash x \rightarrow 5 + 1$
$(3 + 1) : [4, 5]$
$(2^2) : (map\,(^2)\,[4, 6])$

N.B: In a parallel non-strict language threads, by default, reduce expressions to WHNF.

In addition to the *parametric* polymorphism already discussed, e.g.

$length :: [a] \rightarrow Int$

Haskell also supports *ad hoc* polymorphism or overloading, e.g.

- 1, 2, etc. represent both fixed and arbitrary precision integers.
- Operators such as + are defined to work on many different kinds of numbers.
- Equality operator (==) works on numbers and other types.

Note that these overloaded behaviors are different for each type and may be an error, whereas in parametric polymorphism the type truly does not matter, e.g. `length` works for lists of any type.

Declaring Classes and Instances

It is useful to define equality for many types, e.g. `String, Char, Int`, etc, but not all, e.g. functions.
A Haskell class declaration, with a single method:

```
class Eq a where
(==) :: a -> a -> Bool
```

Example instance declarations, `integerEq` and `floatEq` are primitive functions:

```
instance Eq Integer where
  x == y = x `integerEq` y
instance Eq Float where
  x == y = x `floatEq` y
instance (Eq a) => Eq (Tree a) where
  Leaf a         ==Leaf b         = a == b
  (Branch l1 r1) ==(Branch l2 r2) = (l1 == l2) && (r1 == r2)
  _              ==_              = False
```

Number Classes

Haskell has a rich set of numeric types and classes that inherit in the obvious ways. The root of the numeric class hierarchy is `Num`.

- `Integer` in class `Integral`: Arbitrary-precision integers
- `Int` in class `Integral`: Fixed-precision integers
- `Float` in class `RealFloat`: Real floating-point, single precision

Input/Output

To preserve referential transparency, stateful interaction in Haskell (e.g. I/O) is performed in a *Monad*.
Input/Output actions occur in the IO Monad, e.g.

$$\begin{aligned}
&\mathit{getChar} :: \mathit{IO\ Char}\\
&\mathit{putChar} :: \mathit{Char} \to \mathit{IO}\,()\\
&\mathit{getArgs} :: \mathit{IO}\,[\mathit{String}]\\
&\mathit{putStr}, \mathit{putStrLn} :: \mathit{String} \to \mathit{IO}\,()\\
&\mathit{print} :: \mathit{Show\ a} \Rightarrow a \to \mathit{IO}\,()
\end{aligned}$$

Every Haskell program has a `main :: IO ()` function, e.g.

$$\mathit{main} = \mathit{putStr}\ \text{"}\mathit{Hello}\text{"}$$

A `do` statement performs a sequence of actions, e.g.

$$\begin{aligned}
&\mathit{main} :: \mathit{IO}\,()\\
&\mathit{main} = \mathit{do}\ c \leftarrow \mathit{getChar}\\
&\qquad\qquad \mathit{putChar}\ c
\end{aligned}$$

Useful I/O

Many useful IO functions are in the system module and must be imported, see below.

$$\mathit{show} :: (\mathit{Show\ a}) \Rightarrow a \to \mathit{String}$$

converts values of most types into a `String`, e.g.

Evaluation

show 3 $\Rightarrow$ *"3"*

```
read :: (Read a) => String -> a
```

parses a value of most types from a `String`.
A program returning the sum of its command line arguments:

$$\begin{aligned}
&\mathit{main} = \mathit{do}\ \ \mathit{args} \leftarrow \mathit{getArgs}\\
&\qquad\qquad\ \ \mathit{let}\ x = \mathit{read}\,(\mathit{args}!!0)\\
&\qquad\qquad\ \ \mathit{let}\ y = \mathit{read}\,(\mathit{args}!!1)\\
&\qquad\qquad\ \ \mathit{putStrLn}\,(\mathit{show}\,(x+y))
\end{aligned}$$

How to read monadic code

Monadic code enforces a step-by-step execution of commands, operating on a hidden state that is specific to the monad
$\Rightarrow$ this is exactly the programming model you are used to from other languages.

In functional languages, monadic code is a special case, and typically used when interacting with the outside world. We need to distinguish between monadic and purely functional code. This distinction is made in the type, e.g.

$$\mathit{readFile} :: \mathit{FilePath} \to \mathit{IO\ String}$$

Read this as: "the `readFile` function takes a file-name, as a full file-path, as argument and *performs an action in the IO monad* which returns the file contents as a string."

NB: Calling `readFile` doesn't give you the string contents, rather it performs an *action*

Another example of monadic code

Read a file that contains one number at each line, and compute the sum over all numbers.

```
myAction :: String -> IO Int  -- define an IO-action, that takes
myAction fn =
 do                           -- this starts a block of monadic action
    str <- readFile fn        -- perform an action, reading from file
    let lns = lines str       -- split the file contents by lines
    let nums = map read lns   -- turn each line into an integer va
    let res = sum nums        -- compute the sum over all integer valu
    print res                 -- print the sum
    return res                -- return the sum
```

NB: the $\leftarrow$ operator (written `<-`) *binds* the result from executing monadic code to a variable.
The `let` constructs *assigns* the value of a (purely functional) computation to a variable.

Case Study: Caesar Cipher

As a case study of a slightly larger program we examine the “Caesar Cipher” implementation that we have seen in F21CN Computer Network Security. It is one of the examples in the textbook "Programming in Haskell", by Graham Hutton (p42ff).

The code is available here:
http://www.macs.hw.ac.uk/ hwloidl/Courses/F21CN/Labs/caesar.hs

See the comments in this file on installing required packages and running the program.

Example: Caesar Cipher

To summarise:

> *to encrypt a plaintext message M, take every letter in M and shift it by e elements to the right to obtain the encrypted letter; to decrypt a ciphertext, take every letter and shift it by $d = -e$ elements to the left*

As an example, using $e = 3$ as key, the letter *A* is encrypted as a *D*, *B* as an *E* etc.

```
Plain:    ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher:   DEFGHIJKLMNOPQRSTUVWXYZABC
```

Encrypting a concrete text, works as follows:

```
Plaintext:  the quick brown fox jumps over the lazy dog
Ciphertext: WKH TXLFN EURZQ IRA MXPSV RYHU WKH ODCB GRJ
```

More formally we have the following functions for en-/de-cryption:

$$E_e(x) = x + e \mod 26$$

$$D_e(x) = x - e \mod 26$$

Characteristics of Caesar’s Cipher

Note the following:

- The sets of plain- and cipher-text are only latin characters. We cannot encrypt punctuation symbols etc.
- The en- and de-cryption algorithms are the same. They only differ in the choice of the key.
- The key strength is not tunable: shifting by 4 letters is no more safe than shifting by 3.
- This is an example of a *symmetric or shared-key cryptosystem*.

Exercise

Implement an en-/de-cryption function based on the Caesar cipher. Implement a function that tries to crack a Caesar cipher, ie. that retrieves plaintext from ciphertext for an unknown key.

Program header and import statements

```
module Caesar where

import Data.Char
import Math.Algebra.Field.Base
import Data.List
import Test.QuickCheck
```

The `import` statement makes all definitions from a different module available in this file.

Helper functions

```
-- convert a character to an integer, starting with 0 for 'a' etc
let2int :: Char -> Int
let2int c = ord c - ord 'a'

-- convert an index, starting with 0, to a character, e.g 'a'
int2let :: Int -> Char
int2let n = chr (ord 'a' + n)

-- shift a character c by n slots to the right
shift :: Int -> Char -> Char
shift n c | isLower c = int2let (((let2int c) + n) `mod` 26)
          | otherwise = c
```

The `shift` function is the basic operation that we need in the Caesar Cipher: it shifts a character by given number of steps through the alphabet.

The encoding function

```
-- top-level string encoding function
encode :: Int -> String -> String
encode n cs = [ shift n c | c <- cs ]

percent :: Int -> Int -> Float
percent n m = (fromIntegral n / fromIntegral m)*100

-- compute frequencies of letters 'a'..'z' in a given string
freqs :: String -> [Float]
freqs cs = [percent (count c cs) n | c <- ['a'..'z'] ]
           where n = lowers cs
```

The function `freqs` determines the frequencies of all letters of the alphabet in the given text `cs`.

The decoding function

```
-- chi-square function for computing distance between 2 frequency lis
chisqr :: [Float] -> [Float] -> Float
chisqr os es = sum [((o-e)^2)/e | (o,e) <- zip os es]

-- table of frequencies of letters 'a'..'z'
table :: [Float]
table = [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0,
         6.1, 7.0, 0.2, 0.8, 4.0, 2.4,
         6.7, 7.5, 1.9, 0.1,  6.0, 6.3, 9.1,
         2.8, 1.0, 2.4, 0.2, 2.0, 0.1]
```

The `chisqr` function formalises the intuition of matching two curves and returning a value that represents a "distance" between the two curves.

The decoding function

```
-- top-level decoding function
crack :: String -> String
crack cs = encode (-factor) cs
           where factor = head (positions (minimum chitab) chital
                 chitab = [ chisqr (rotate n table') table
                          | n <- [0..25] ]
                 table' = freqs cs
```

In the `crack` function, we try all possible shift values, and match the curve for each value with the known frequency of letters, taken from an English dictionary.

More helper functions

```
-- rotate a list by n slots to the left; take, drop are Prelude functi
rotate :: Int -> [a] -> [a]
rotate n xs = drop n xs ++ take n xs

-- the number of lower case letters in a given string
lowers :: String -> Int
lowers cs = length [ c | c <- cs, isLower c]

-- count the number of occurrences of c in cs
count :: Char -> String -> Int
count c cs = length [ c' | c' <- cs, c==c']

-- find list of positions of x in the list xs
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [ i' | (x', i') <- zip xs [0..n], x==x' ]
                 where n = length xs - 1
```

These are the helper functions, needed by `crack`.

Case Study

To test the program, start the Haskell interpreter `ghci`:

```
# ghci -package HaskellForMaths -package QuickCheck caesar.hs
```

Now, inside the interpreter try these commands:

```
#> let s1 = "a completely random text string"
#> let c1 = encode 3 s1
#> c1
#> let d1 = crack c1
#> d1
#> let s2 = "unusal random string"
#> let c2 = encode 7 s2
#> c2
#> let d2 = crack c2
#> d2
```

Main learning & reference material

- A list of learning resources is on the Haskell Wiki: `http://www.haskell.org/haskellwiki/Learning_Haskell`
- An excellent site for learning Haskell is: `https://www.fpcomplete.com/school`
- A very recent MOOC on Haskell has been developed at the Univ. of Glasgow: `https://www.futurelearn.com/courses/functional-programming-haskell/`
- the Haskell Wiki has a lot of resources: `http://www.haskell.org/haskellwiki/Haskell`
- the language specification is available as Haskell Report: `http://www.haskell.org/onlinereport/haskell2010/`
- a list of all prelude functions is available: `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html`
- Hackage is a list of (lost of) third-party libraries: `http://hackage.haskell.org/`

Uses of Haskell

Rapid prototyping: code is much shorter

Symbolic applications development, e.g. natural language processors, chess games.

Computational finance for market predictions based on mathematical models.

High-level programming concepts: Generics in Java/C# are instances of polymorphism

Computation language for parallel, distributed, mobile or Grid languages

Used by major companies and institutions, such as Facebook, AT&T, and NASA.

Exercise: Trees

- Implement a function `depth :: Tree a -> Int` to compute the depth of the tree, ie. the longest path from the root of the tree to one of the leaves.
- Using the `depth` function, write a function `isBalanced :: Tree a -> Bool` that checks, whether a tree is balanced, ie. the lengths of any 2 paths in the tree does not differ by more than 1.
- Write a Haskell function `mkTree :: [a] -> Tree a` that constructs a balanced binary tree, containing the elements of a given list.
  Hint: Check the prelude function `splitAt`.

For a full list of Haskell exercises see: `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/tutorial0.html`

HERIOT WATT UNIVERSITY