AC21007: Haskell Lecture 2
List functions, function polymorphism, non-strict semantics

František Farka

Recapitulation

Haskell

- purely functional
- non-strict (also lazy) semantics
- (strong) static typing

Recapitulation (cont.)

- Data types (`Bool`, `Int`, `String`, ...) and data values (`True`, `False`, ..., -1, 0, 1, ..., "Hello World!", ...) begin with an upper case letter
- Function and variable identifiers (power, neg, b, n) begin with a lower case letter
- Variables in Haskell cannot be updated
- Function definition:
  - a set of equations, LHS is a pattern, RHS is an expression
  - value matches only itself (True matches True)
  - variable matches any value ... and binds the variable to the matched value

Recapitulation (cont.)

- An example: logic and

```
myAnd :: Bool -> Bool -> Bool
myAnd True  True   = True
myAnd True  False  = False
myAnd False True   = False
myAnd False False  = False
```

- Recall:
  - value matches only itself (True matches True)
  - variable matches any value ... and binds the variable to the matched value

Recapitulation (cont.)

- An example: logic and

```
myAnd :: Bool -> Bool -> Bool
myAnd True  True   = True
myAnd a     b      = False
```

- Recall:
  - value matches only itself (True matches True)
  - variable matches any value ... and binds the variable to the matched value

Recapitulation (cont.)

- An example: logic and

```
myAnd :: Bool -> Bool -> Bool
myAnd True  True   = True
myAnd _     _      = False
```

- Recall:
  - value matches only itself (True matches True)
  - variable matches any value ... and binds the variable to the matched value
- New:
  - '_' matches any value, no binding created

List Datatype

- data type `[Int]` – a list where each element is of the type `Int`
- list values created by *constructors*
  - `[]` – constructs an empty list, and
  - `(:)` – (*cons*) from a value and list of values constructs a new list, prepends the value
- These are lists:

```
[]
(1 : [])
(2 : (5 : (3 : [])))
```

- There is a special syntax:

```
[1]
[2, 5, 3]
```

List Datatype (cont.)

- data type `[Bool]` – each element is of the type `Bool`
- yet again, constructors `[]` and `(:)`
- these are lists of booleans:

```
[]
True : (False : (True : []))
[False, True, True, False]
```

Programming with list datatypes

- The `sum` function computes the sum of a list of integers:

```
sum :: [Int] -> Int
sum []           = 0
sum (x : xs)     = x + (sum xs)
```

- The `all` function determines whether all the elements of a list of booleans are `True`:

```
all :: [Bool] -> Bool
all []           = True
all (True : xs) = all xs
all _            = False
```

- New patterns: list values can be matched against list constructors: `[]` matches itself and `(:)` matches a non-empty list, while matching both the patterns for the first element and for the rest of the list

Programming with list datatypes (cont.)

- The `lengthInt` function computes the length of a list of integers:

```
lengthInt :: [Int] -> Int
lengthInt []       = 0
lengthInt (_ : xs) = 1 + lengthInt xs
```

- The `lengthBool` function computes the length of a list of integers:

```
lengthBool :: [Bool] -> Int
lengthBool []       = 0
lengthBool (_ : xs) = 1 + lengthBool xs
```

- The source code is nearly the same . . . can we abstract over Int and Bool?

List Datatype - [a]

- Haskell has *type variables* – identifiers beginning with a lowercase letter
- Data type `[a]` – a list where each element is of type `a`
- Exactly two constructors:
  - `[] :: [a]`
  - `(:) :: a -> [a] -> [a]`
- A type with type variables is *polymorphic*, it is instantiated to a *monomorphic* type
- A polymorphic `length` function:

```
length :: [a] -> Int
length []       = 0
length (_ : xs) = 1 + length xs
```

List Datatype [a] - some functions

- `head` - access the first element:

```
head :: [a] -> a

head (x : _) = x
```

- `tail` - access the rest of a list:

```
tail :: [a] -> [a]

tail (_ : xs) = xs
```

- What about a head of an empty list `head []`?

```
Error:  Non-exhaustive patterns in function head
```

List Datatype [a] - some functions

- `head` - access the first element:

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

- `tail` - access the rest of a list:

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

- What is the RHS? We don't know anything about the type `a`.

List Datatype [a] - some functions

- `head` - access the first element:

```
head :: [a] -> a
head []      = error "Empty list"
head (x : _) = x
```

- `tail` - access the rest of a list:

```
tail :: [a] -> [a]
tail []       = error "Empty list"
tail (_ : xs) = xs
```

- Haskell has special functions for run-time errors:
  - `error :: String -> a`
    prints a specified error and terminates evaluation (program)
  - `undefined :: a`
    print a generic error and terminates evaluation

Syntactic intermezzo – functions and operators

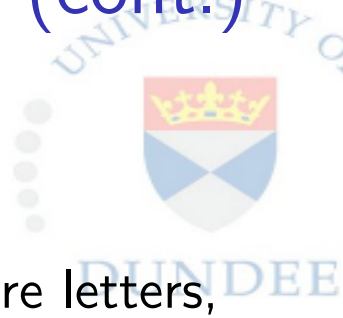
- Sometimes we do not want functions (e.g. `power`, `sum`) but operators (e.g. `*`, `++`)
- Consider the following list index function:

```
at :: [a] -> Int -> a
at (x : _)      0    = x
at (_ : xs)     1    = at (i - 1) xs
at []           _    = error "out of bound"

    -- usage:   at [1,2,3] 1     ==> 2
```

- We can use an operator:

```
(!!) :: [a] -> Int -> a
xs !! i = at xs i

    -- usage:   [1,2,3] !! 1    ==> 2
```

Syntactic intermezzo – functions and operators (cont.)



- Function identifiers
  - consist of a lowercase letter followed by zero or more letters, digits, underscores, and single quotes
  - prefix application (e.g. `at [1,2,3] 0`)
- Operators
  - consist of symbols – `%!#$%&*+./<=>?^|-~`
  - infix application (e.g. `[1,2,3] !! 0`)
- Special syntax for using an operator in the prefix notation

```
(!!) [1,2,3] 2
```

- Special syntax for using a function in the infix notation

```
[1,2,3] `at` 2
```

Next time

- Monday the the 25th of January, 2-3PM, Dalhousie 3G05 LT2
- Non-strict semantics
- More list functions
- Tuples
- First-class functions
- Folds over lists