Concepts of Programming Languages

A Brief Intro to Programming in Haskell

Lecturer: Gabriele Keller
Tutor: Liam O’Connor
University of New South Wales
School of Computer Sciences & Engineering
Sydney, Australia

COMP 3161/9161 Week 1

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

Summary

- Function definitions have the form:

```
<functionName> <arg1> <arg2> …   = <expr>
f x y = 2 * x + y
```

- We can add optional type annotations:

```
<functionName> :: <arg1Type> -> <arg2> … -> <resultType>
f ::  Double -> Double -> Double
f x y = 2 * x + y
```

- Haskell has the usual basic types (type names start with upper case letter!
  - `Char, Float, Int, Double, …`
  - `Bool` with the elements `True False`
  - `String`

Summary

- The list type:

```
[1,2,3]            :: [Int]
['a', 'b', 'c']    :: [Char]
[[1,2], [3], []]   :: [[Int]]
```

- The list notation [1,2,3] is syntactic sugar for

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

- `:` and `[]` are the *data constructors* for lists

Data Constructors: Lists

- Data constructors are used to build values of non-basic type
- Lists have two data constructors, already predefined in `Prelude` module
  - `(:) :: a -> [a] -> [a]`

    right associative infix constructor which takes a data item and a list as argument
  - `[]  :: [a]`

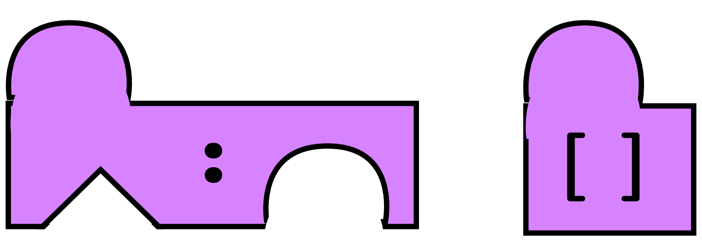
    the empty list constructor
- Lists are *polymorphic*: *a* is a *type variable* - can be any type!
  - type names start with upper case letter
  - type variables with lower case

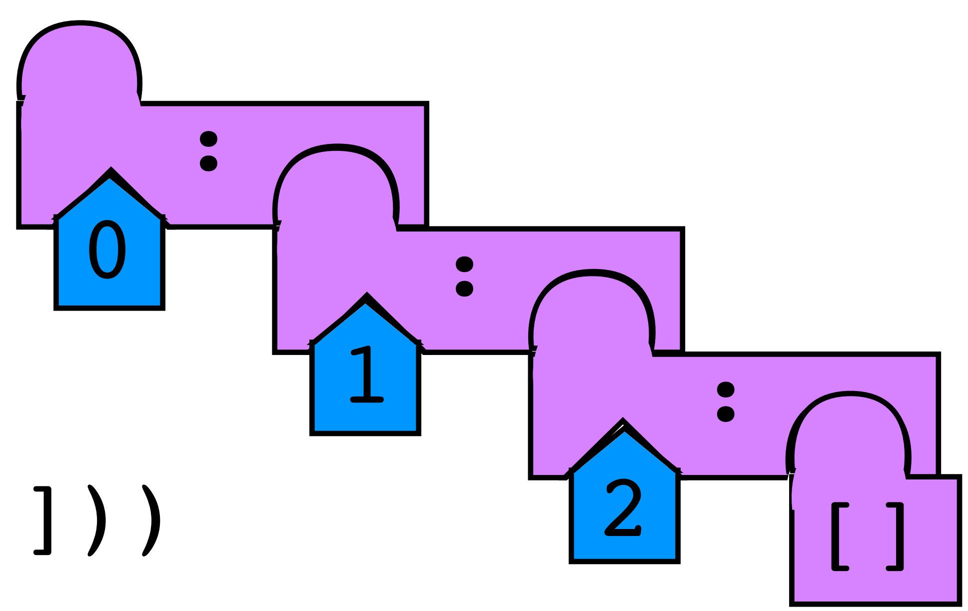
Data Constructors: Lists

- Data constructors are used to build values of non-basic type
- Lists, which are predefined in `Prelude` module have two data constructor:
  - `(:) :: a -> [a] -> [a]`

    right associative infix constructor which takes a data item and a list as argument
  - `[] :: [a]`

    the empty list constructor





`0:1:2:[]` `0:(1:(2:[]))`

Data Constructors

- Data constructors are a special class of functions
  - they consume zero or more arguments
  - return a new value
- In contrast to most functions, you can always take the result apart and retrieve the arguments:
  - `4 + 5   = 9`   `9 = ? + ?`
  - `1 : [2] = [1,2]`   `[1, 2] = ? : ?`

Pattern matching

- Function definitions can match on argument patterns:

```
<functionName> <arg1> <arg2> …    = <expr>

<functionName> <argPattern> <argPattern> …    = <expr>
```

- Patterns are a mix of data constructors, constants values and variable names

```
[]

(1 : xs)

(x : y : xs)

((x : y) : xs)
```

~~(x : x : xs)~~

Pattern matching

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

```
length :: [a] -> Int
length xs =  case xs of
   []      -> 0
   (y:ys) -> 1 + length ys
```

Lists

- Since lists are such a central data structure, there is some additional syntactic sugar to make using them more convenient
  - `0:1:2:[]` can be written as `[0, 1, 2]` or any mix of the styles, like `0:[1,2]` `0:1:[2]` (but not `[0,1]:[2]`!!)
  - Strings are lists of characters
    - `"Hello"`
    - `['H','e','l','l','o']`
    - `'H':'e':'l':'l':'o': []`

type synonym defined in the Prelude module, `type` is similar to `typedef` in C

```
type String = [Char]
```

Back to the lexer

- We defined a new data type to represent tokens, as there is no suitable predefined data type

```
data Token
  = OpenP
  | CloseP
  | Plus
  | Minus
```

`Token` is the name we choose for the new type

`OpenP`, `CloseP` etc are the names we choose for the elements of the new type. They are the type constructors of the type `Token`

Back to the lexer

- What about numbers?

```
(( 2 - 3) + 5()
```

```haskell
data Token
  = OpenP
  | CloseP
  | Plus
  | Minus
  | IntLit Int
```

```haskell
OpenP  :: Token
IntLit :: Int -> Token
```

Lists are everywhere

- Lists are homogeneous - all elements have to have the same type
  - `[1,2,3]` :: [Int]
  - `[“hello”, “world”]` :: [[Char]] or [String]
  - `[‘a’, 5, 6]` type error - Char!
- Many useful higher-oder list functions predefined in Prelude (have a look at the module)
  - `map :: (a -> b) -> [a] -> [b]`
    - `map f [`x_1`, `x_2`, `x_3`, `x_4`]` is `[f `x_1`, f `x_2`, f `x_3`, f `x_4`]`
  - `foldr :: :: (a -> b -> b) -> b -> [a] -> b`
    - `foldr (+) n [`x_1`, `x_2`, `x_3`, `x_4`]` is $x_1+(x_2+(x_3+(x_4+n)))$
  - `foldl :: (a -> b -> a) -> a -> [b] -> a`
    - `foldl (+) n [`x_1`, `x_2`, `x_3`, `x_4`]` is $(((n+x_1)+x_2)+x_3)+x_4$
  - `break :: (a -> Bool) -> [a] -> ([a], [a])`
    - `break (isUpper) "hELlo"` is `(“h”,"ELlo")` –`isUpper from Data.Char`

Some other examples

- Days of the week:

```
data Day
  = Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
  | Sunday

— sample function
isWeekday :: Day -> Bool
isWeekday Saturday = False
isWeekday Sunday   = False
isWeekday day      = True   — variable ‘day’ matches any day
```

Some other examples

- Shapes:

```
data Shape
  = Square    Double        — square has length as argument
  | Rectangle Double Double — rectangle has height & width
  | Circle    Double        — circle has radius as argument


— calculate are of shape
areaOfShape :: Shape -> Double
areaOfShape (Square len) = len * len
areaOfShape (Rectangle height width) = height * width
areaOfShape (Circle radius) = radius * pi * pi — ‘pi’ is predefined
```

Type Classes

- What could be the type of the function (==), which compares two data items for equality?
  - `(==) :: a -> a -> Bool`   no, that's too general!
  - `(==) :: Eq a => a -> a -> Bool`
  - if `a` is a member of *type class* `Eq`, then `(==)` can compare two values of this type for equality
  - when we define a new data type, we can include it into the class using `deriving`

```
data Token
  = OpenP
  | CloseP
  | Plus
  | Minus
  | IntLit Int
  deriving (Eq, Show)
```