



EDAF95/EDAN40: Functional Programming Language Overview

Jacek Malec

Dept. of Computer Science, Lund University, Sweden

March 22nd, 2023



Typing

- Haskell is strongly and statically typed
- Type declarations optional
- Type checking uses type inferencing
- Enforced convention:
 - Type names: begin with an uppercase letter
 - Variable/expression names: begin with a lowercase letter



Operators and functions

2 + 3

and [True, True, False]



Operators and functions

2 + 3

and [True, True, False]

Function from operator:

add1 = (+)

Operator from function:

2 `add1` 3



Curried functions

Parentheses avoided:

f a b

is to be read

((f a) b)

and is different from

f (a, b)

f (a b)

Functions always take only one argument!



Partial application, cntd.

```
inc1 = add1 1
```

or

```
inc2 = (+1)
```

An expression is itself often its best name!



Composition of functions

```
doublePlusOne = inc2.(2*)
```

or

```
doublePlusOne = (+1).(2*)
```



Lambda expressions

```
incAll = map (\i->i+1)
```



Pattern-based definitions

```
count :: Int -> String
```

```
count 1 = "one"  
count 2 = "two"  
count _ = "many"
```

One function can have many equations.



Guards

```
oddOrEven :: Int -> String
```

```
oddOrEven i
| odd i      = "odd"
| even i     = "even"
| otherwise  = "strange"
```



Local definitions

```
isPythagorean1 a b c =
  (sq a) + (sq b) == (sq c)
  where sq x = x*x

isPythagorean2 a b c =
  let sq x = x*x
  in (sq a) + (sq b) == (sq c)
```



Indentation

- indentation denotes continuation, unless brace notation is used
- some keywords (let, where, do, of) begin *layout blocks*

```
module Main where{main=do{putStrLn "Look at me";putStrLn  
"I'm writing all my code on four lines";dice};dice=do{input<-getLine;  
let{val::Int;val=read input};putStrLn$ "What a "++if val<5 then  
"small number"else "not-so-small number"};}
```

More about it later.



Polymorphic types

The type of (.)

$(f.g) \ x = f \ (g \ x)$

is

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

Type variables begin with a lowercase letter.



Tuples

Fixed number of elements, may be of different types.

Pairs:

```
(4, "four") :: (Int, String)
```

Triples, quadruples, etc. - analogously.



Lists

Arbitrary number of elements of the same type

[1,2,3,4] ,

[1..10] ,

[1,3..10] ,

[2..] :: [Int]



Strings

A special case of lists

String = [Char]

with a special syntax

"Common Lisp" = ['C', 'o', 'm', 'm', 'o', 'n', ' ', 'L', 'i', 's', 'p']



Functions on lists

The archetypical pattern:

```
length1 :: [a] -> Int
```

```
length1 [] = 0
```

```
length1 (x:xs) = 1 + (length1 xs)
```



Some standard list functions

filter:

```
filter even [1..]
```

map:

```
map doublePlusOne [1..3]
```

fold (foldr, foldl):

```
sum = foldr (+) 0
```

```
length2 = foldr (\i->(\j->j+1)) 0
```

zip, zipWith:

```
indexed aList = zip [0..] aList
```

```
indexed2 = zip [0..]
```



List comprehensions

```
allIntPairs = [ (i,j) | i<-[0..], j<-[0..i]]
```

```
eExp x =     runningSums [ (x^i)/(fac i) | i<-[0..]]
```



Infinite lists

```
ones1 = 1:ones1
```

```
ones2 = [1,1..]
```

```
sieve1 (n:ns) = n: sieve1 (filter (\x-> x `mod` n > 0) ns)
```

```
sieve2 (n:ns) = n: sieve2 [ x | x <- ns, x `mod` n > 0 ]
```

```
eExp x = runningSums [ (x^i)/(fac i) | i<-[0..]]
```



Type synonyms

```
type Name = String
```



Enumerated types

```
data Color =  
    Red | Green | Blue |  
    Yellow | Black | White
```



Algebraic datatypes

```
data Price =  
    Euro Int Int | Dollar Int Int
```

Enumerated types generalized!



Pattern matching revisited

```
complement :: Color -> Color
```

```
complement Red = Green
```

```
complement Green = Red
```

```
complement Blue = Yellow
```

```
...
```

The pattern cases correspond to alternative constructor functions of the data type.



Recursive type definitions

```
data IntTree =  
    IntEmpty | IntNode Int IntTree IntTree
```



Recursive type definitions

```
data IntTree =  
    IntEmpty | IntNode Int IntTree IntTree
```

or a polymorphic version:

```
data Tree a =  
    Empty | Node a (Tree a) (Tree a)
```



Qualified types

The type of:

`elem x xs = any (==x) xs`

is

`(Eq a) => a -> [a] -> Bool`



Type classes

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
    x == y = not (x /= y)
```

Somewhat like Java interfaces!



Class instances

```
instance Eq Bool where
    True == True    = True
    False == False  = True
    _ == _          = False
```



Subclassing

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) ...
```



Input/output

The abstract datatype `IO` a of I/O actions

```
putChar :: Char -> IO ()  
getChar :: IO Char
```



Do-notation

The abstract datatype `IO` a of I/O *actions*

```
greeting :: IO ()
```

```
greeting = do
    putStrLn "Enter your name"
    name <- getLine
    putStrLn ("You " ++ name ++ ", me Haskell!")
```



Another I/O function

```
getLine :: IO String
```

```
getLine = do c <- getChar
            if c == '\n'
                then return ""
                else do l <- getLine
                        return (c:l)
```



Other

- Modules
- Comments
- Literate Haskell