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 names: begin with a lowercase letter

Operators and functions

2 + 3
and [True, True, False]

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.

\[ \text{inc1} = \text{add1 1} \]

or

\[ \text{inc2} = (+1) \]

An expression is itself often its best name!

Composition of functions

\[ \text{doublePlusOne} = \text{inc2}(2\ast) \]

or

\[ \text{doublePlusOne} = (+1)(2\ast) \]

Lambda expressions

\[ \text{incAll} = \text{map } (\lambda i \rightarrow i+1) \]
Pattern-based definitions

```haskell
count :: Int -> String
count 1 = "one"
count 2 = "two"
count _ = "many"
```

One function can have many equations.

```haskell
oddOrEven :: Int -> String
oddOrEven i
  | odd i    = "odd"
  | even i   = "even"
  | otherwise = "strange"
```

Guards

```haskell
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)
```

Local definitions

```
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"};}
```

Indentation

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

More about it later.
Polymorphic types

The type of \((f \cdot g)\)
\[(f \cdot g)\ x = f\ (g\ x)\]
is
\[(b \to c) \to (a \to b) \to a \to c\]

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:

```haskell
length1 :: [a] -> Int
length1 [] = 0
length1 (x:xs) = 1 + (length1 xs)
```

### Some standard list functions

**filter:**

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

**map:**

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

**fold (foldr, foldl):**

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

**zip, zipWith:**

```haskell
indexed aList = zip [0..] aList
indexed2 = zip [0..]
```

### List comprehensions

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

### Infinite lists

```haskell
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

```haskell```
type Name = String
```

Enumerated types

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

Algebraic datatypes

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

Pattern matching revisited

```haskell```
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

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**

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

**Subclassing**

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

**Input/output**

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

```haskell
putChar :: char -> IO ()
getChar :: IO char
```

**Do-notation**

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

```haskell
greeting :: IO ()
greeting = do
    putStrLn "Enter your name"
    name <- getLine
    putStrLn ("You " ++ name ++ ", me Haskell!")
```
Another I/O function

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

Modules
Comments
Literate Haskell