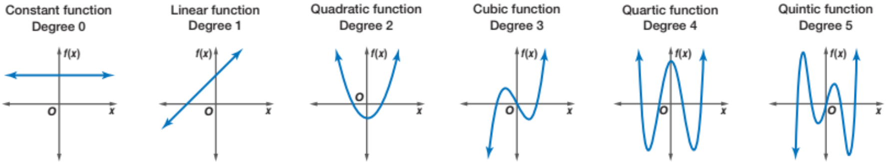
Basic Function Definitions

Stephen A. Edwards

Columbia University

Fall 2019



Patterns

You can define a function with patterns

Patterns may include literals, variables, and _ “wildcard”

```
badCount :: Integral a => a -> String
badCount 1 = "One"
badCount 2 = "Two"
badCount _ = "Many"
```

Patterns are tested in order; put specific first:

```
factorial :: (Eq a, Num a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Pattern Matching May Fail

```
Prelude> :{
Prelude| foo 'a' = "Alpha"
Prelude| foo 'b' = "Bravo"
Prelude| foo 'c' = "Charlie"
Prelude| :}
Prelude> :t foo
foo :: Char -> [Char]
Prelude> foo 'a'
"Alpha"
Prelude> foo 'd'
"*** Exception: <interactive>:(23,1)-(25,19): Non-exhaustive
                                            patterns in function foo
```

Let the Compiler Check for Missing Cases

Much better to get a compile-time error than a runtime error:

```
Prelude> :set -Wall
Prelude> :{
Prelude| foo 'a' = "Alpha"
Prelude| foo 'b' = "Bravo"
Prelude| :}

<interactive>:32:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for 'foo':
        Patterns not matched: p where p is not one of {'b', 'a'}
```

```
Prelude> :set -Wincomplete-patterns
```

Pattern Matching on Tuples

A tuple in a pattern lets you dismantle the tuple. E.g., to implement *fst*,

```
Prelude> fst' (x,_) = x
Prelude> :t fst'
fst' :: (a, b) -> a
Prelude> fst' (42,28)
42
Prelude> fst' ("hello",42)
"hello"
```

```
Prelude> addv (x1,y1) (x2,y2) = (x1 + x2, y1 + y2)
Prelude> :t addv
addv :: (Num a, Num b) => (a, b) -> (a, b) -> (a, b)
Prelude> addv (1,10) (7,3)
(8,13)
```

Patterns in List Comprehensions

Usually, where you can bind a name, you can use a pattern, e.g., in a list comprehension:

```
Prelude> :set +m
Prelude> pts = [ (a,b,c) | c <- [1..20], b <- [1..c], a <- [1..b],
Prelude|                   a^2 + b^2 == c^2 ]
Prelude> pts
[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17),(12,16,20)]

Prelude> perimeters = [ a + b + c | (a,b,c) <- pts ]

Prelude> perimeters
[12,24,30,36,40,48]
```

Pattern Matching On Lists

You can use `:` and `[,,,]`-style expressions in patterns

Like *fst*, *head* is implemented with pattern-matching

```
Prelude> :{
Prelude| head' (x:_) = x
Prelude| head' [] = error "empty list"
Prelude| :}

Prelude> :t head'
head' :: [p] -> p

Prelude> head' "Hello"
'H'
```

Pattern Matching On Lists

```
Prelude> :{
Prelude| dumbLength [] = "empty"
Prelude| dumbLength [_] = "singleton"
Prelude| dumbLength [_,_] = "pair"
Prelude| dumbLength [_,_,_] = "triple"
Prelude| dumbLength _ = "four or more"
Prelude| :}

Prelude> :t dumbLength
dumbLength :: [a] -> [Char]
Prelude> dumbLength []
"empty"
Prelude> dumbLength [1,2,3]
"triple"
Prelude> dumbLength (replicate 10 ' ')
"four or more"
```

List Pattern Matching Is Useful on Strings

```
Prelude> :{
Prelude| notin ('i':'n':xs) = xs
Prelude| notin xs = "in" ++ xs
Prelude| :}

Prelude> notin "inconceivable!"
"conceivable!"
Prelude> notin "credible"
"incredible"
```

Pattern Matching On Lists with Recursion

```
Prelude> :{
Prelude| length' [] = 0
Prelude| length' (_:xs) = 1 + length' xs
Prelude| :}
Prelude> :t length'
length' :: Num p => [a] -> p
Prelude> length' "Hello"
5
```

```
Prelude> :{
Prelude| sum' [] = 0
Prelude| sum' (x:xs) = x + sum' xs
Prelude| :}
Prelude> sum' [1,20,300,4000]
4321
```

The “As Pattern” Names Bigger Parts

Syntax: <name>@<pattern>

```
Prelude> :{
Prelude| initial "" = "Nothing"
Prelude| initial all@(x:_) = "The first letter of " ++ all ++
Prelude|                     " is " ++ [x]
Prelude| :}

Prelude> :t initial
initial :: [Char] -> [Char]
Prelude> initial ""
"Nothing"
Prelude> initial "Stephen"
"The first letter of Stephen is S"
```

Guards: Boolean constraints

Patterns match structure; guards (Boolean expressions after a |) match value

```
Prelude> :{
Prelude| heightEval h
Prelude|   | h < 150 = "You're short"
Prelude|   | h < 180 = "You're average"
Prelude|   | otherwise = "You're tall"     -- otherwise = True
Prelude| :}

Prelude> heightEval 149
"You're short"
Prelude> heightEval 150
"You're average"
Prelude> heightEval 180
"You're tall"
```

Filter: Keep List Elements That Satisfy a Predicate

odd and *filter* are Standard Prelude functions

```
odd n = n `rem` 2 == 1

filter :: (a -> Bool) -> [a] -> [a]
filter p []                   = []
filter p (x:xs) | p x         = x : filter p xs
                | otherwise   = filter p xs
```

```
Prelude> filter odd [1..10]
[1,3,5,7,9]
```

Compare: Returns LT, EQ, or GT

Another Standard Prelude function

```
x `compare` y
  | x < y     = LT
  | x == y    = EQ
  | otherwise = GT
```

```
Prelude> :t compare
compare :: Ord a => a -> a -> Ordering
Prelude> compare 5 3
GT
Prelude> compare 5 5
EQ
Prelude> compare 5 7
LT
Prelude> 41 `compare` 42
LT
```

Where: Defining Local Names

```
triangle :: Int -> Int -> Int -> String
triangle a b c
   | a + b < c  || b + c < a  || a + c < b  = "Impossible"
   | a + b == c || a + c == b || b + c == a = "Flat"
   | right                                  = "Right"
   | acute                                  = "Acute"
   | otherwise                              = "Obtuse"
   where
     right = aa + bb == cc || aa + cc == bb || bb + cc == aa
     acute = aa + bb > cc  && aa + cc > bb  && bb + cc > aa
     sqr x = x * x
     (aa, bb, cc) = (sqr a, sqr b, sqr c)
```

Order of the *where* clauses does not matter

Indentation of the *where* clauses must be consistent

Where blocks are attached to declarations

The Primes Example

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

`[2..]`	The infinite list [2,3,4,...]
`where filterPrime`	*Where* clause defining *filterPrime*
`(p:xs)`	Pattern matching on head and tail of list
`p : filterPrime ...`	Recursive function application
`[x | x <- xs, x ‘mod‘ p /= 0]`	List comprehension: everything in xs not divisible by `p`

Haskell Layout Syntax

Internally, the Haskell compiler intreprets

```haskell
a = b + c
   where
     b = 3
     c = 2
```

as

```haskell
a = b + c where { b = 3 ; c = 2 }
```

The only effect of layout is to insert `{ ; }` tokens.

Manually inserting `{ ; }` overrides the layout rules

Haskell Layout Syntax

- Layout blocks begin after *let*, *where*, *do*, and *of* unless there's a {
- The first token after the keyword sets the indentation of the block
- Every following line at that indentation gets a leading ;
- Every line indented more is part of the previous line
- The block ends (an implicit }) when anything is indented less

```
a = b + c where
  b = 2
  c = 3
```

```
a = b + c
  where b = 3
            + 2
        c = 3
```

```
a = b + c where b = 2
                c = 3
```

```
a = b + c
  where b = 3
        + 2   -- No
        c = 3
```

```
a = b + c
  where b = 2
        c = 3
```

```
a = b + c
  where b = 2
      c = 3 -- No
```

Let Bindings: Naming Things In an Expression

`let` <bindings> `in` <expression>

```
cylinder :: RealFloat a => a -> a -> a
cylinder r h = let sideArea = 2 * pi * r * h
                   topArea = pi * r^2
               in sideArea + 2 * topArea
```

This example can be written "more mathematically" with *where*

```
cylinder :: RealFloat a => a -> a -> a
cylinder r h = sideArea + 2 * topArea
  where sideArea = 2 * pi * r * h
        topArea = pi * r^2
```

Semantically equivalent; *let...in* is an expression; *where* only comes after bindings. Only *where* works across guards.

let...in Is an Expression and More Local

A contrived example:

```
f a = a + let a = 3 in a
```

This is the "add 3" function. The scope of `a = 3` is limited to the *let...in*

let bindings are recursive. E.g.,

```
let a = a + 1 in a
```

does not terminate because all the a's refer to the same thing: `a + 1`

This can be used to define infinite lists

```
Prelude> take 5 (let x = 1 : 2 : x in x)
[1,2,1,2,1]
```

but is mostly for defining recursive functions. There's no non-recursive *let*

Let Can Also Be Used in List Comprehensions

```
Prelude> handshakes n = [ handshake | a <- [1..n-1], b <- [a+1..n],
Prelude|                                let handshake = (a,b) ]
Prelude> handshakes 3
[(1,2),(1,3),(2,3)]
```

Its scope includes everything after the *let* and the result expression

case...of Is a Pattern-Matching Expression

Defining a function with patterns is syntactic sugar for *case...of*

```
badCount 1 = "One"
badCount 2 = "Two"
badCount _ = "Many"
```

is equivalent to

```
badCount x = case x of
     1 -> "One"
     2 -> "Two"
     _ -> "Many"
```

But, like *let*, *case...of* is an expression and may be used as such:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of []  -> "empty"
                                               [x] -> "a singleton"
                                               _   -> "two or more"
```