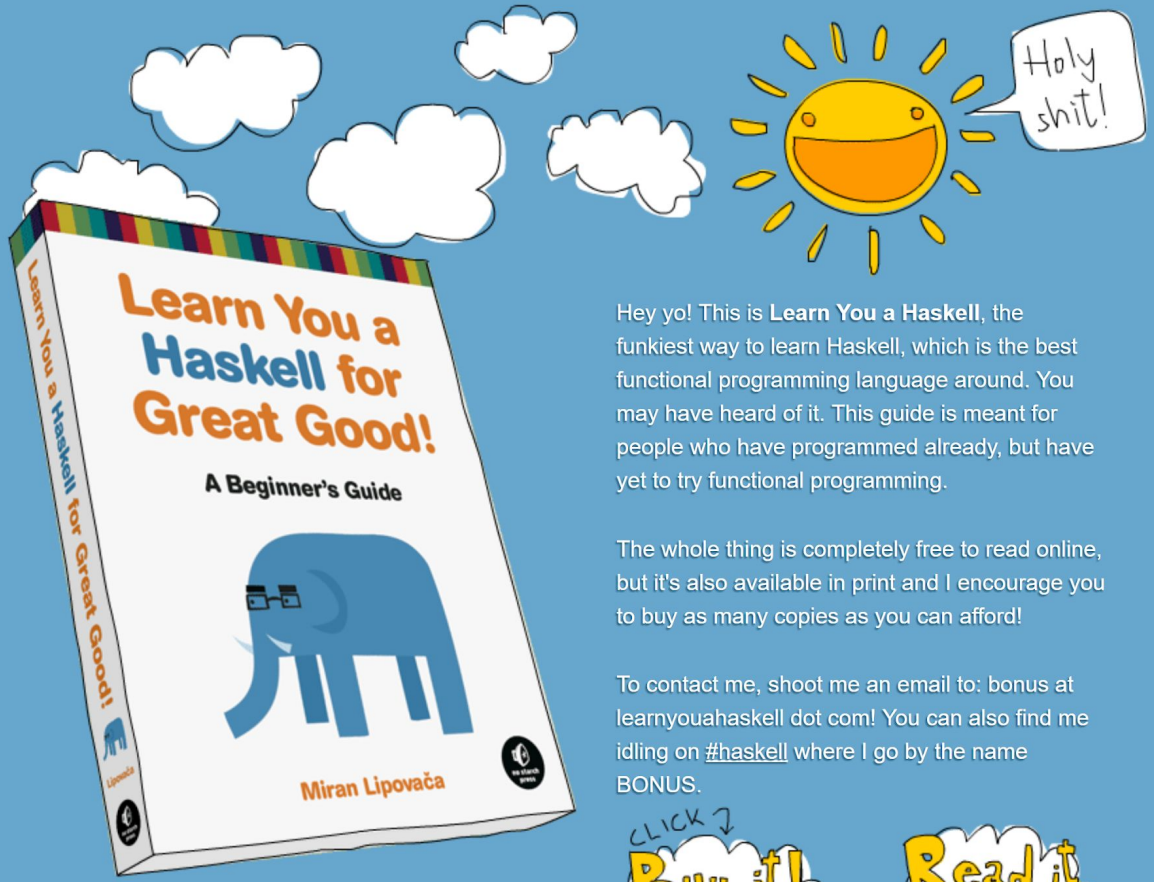
EECS 368

<= 4/13/2018

Paul Kline
pauliankline@gmail.com
Office 2050Eaton
M: 4-5pm
W: 4-5pm
F: 2-3pm



Hey yo! This is **Learn You a Haskell**, the funkiest way to learn Haskell, which is the best functional programming language around. You may have heard of it. This guide is meant for people who have programmed already, but have yet to try functional programming.

The whole thing is completely free to read online, but it's also available in print and I encourage you to buy as many copies as you can afford!

To contact me, shoot me an email to: bonus at learnyouahaskell dot com! You can also find me idling on #haskell where I go by the name BONUS.

CLICK

Buy it! You know you want to!

Read it online! For free!

Learnyouahaskell.com

The Glasgow Haskell Compiler

A powerful combination of an optimizing compiler and a Hugs-style interactive environment

Also on the web Dowload Haskell compiler

https://www.haskell.org/platform/windows.html

Starting GHCi

GHCi can be started using `% ghci`

```
andy$ ghci                                          :? for help
GHCi, version 6.10.1:
http://www.haskell.org/ghc/  Loading package
ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> ⟶ ghci>
```

:set prompt "ghci> "

GHCi can evaluate expressions

```
andy$ ghci
> 2 + 3 * 4
14
> (2+3)*4   20
> sqrt (3^2 + 4^2)   5.0
```

> 5 * (-3)

* is infix function

The Standard Prelude

List 可以是不同的长度，但必须是相同的类型。

The library file Prelude.hs provides a large number of standard functions. In addition to the familiar numeric functions such as + and *, the library also provides many useful functions on lists.

函数调用：函数名，空格，空格分隔的参数

- Select the first element of a list:

```
> head [1,2,3,4,5]
1
```

head [] //error

Select the last element of a lis
> last [1,2,3,4,5] 5

Remove the last element from a list

>init [1,2,3,4,5] [1,2,3,4]

- Remove the first element from a list:

```
> tail [1,2,3,4,5]
[2,3,4,5]
```

head, tail, init, last 不能用在空 list!

- Select the nth element of a list:

index begins 0

```
> [1,2,3,4,5] !! 2
3
```

- Select the first n elements of a list:

```
> take 3 [1,2,3,4,5]
[1,2,3]
```

cycle [1,2,3]

返回一个 list, 循环输出 1,2,3

[1,2,3,1,2,3....]

repeat 返回一个仅包含该值的无限list repeat 5
> replicate 3 10 [10,10,10] [5,5,5...]

- Remove the first n elements from a list:

```
> drop 3 [1,2,3,4,5]
[4,5]
```

他大小!

>[3,2,1] > [2,1,0]

True

- Calculate the length of a list:

```
> length [1,2,3,4,5]
5
```

- Calculate the sum of a list of numbers:

```
> sum [1,2,3,4,5]
 15
```

> minimum [1,2,3]

1

> maximum

- Calculate the product of a list of numbers:

```
> product [1,2,3,4,5]
120
```

- Append two lists:

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

- Reverse a list:

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

>4 'elem' [3,4,5,6]
True

Add elements in front:

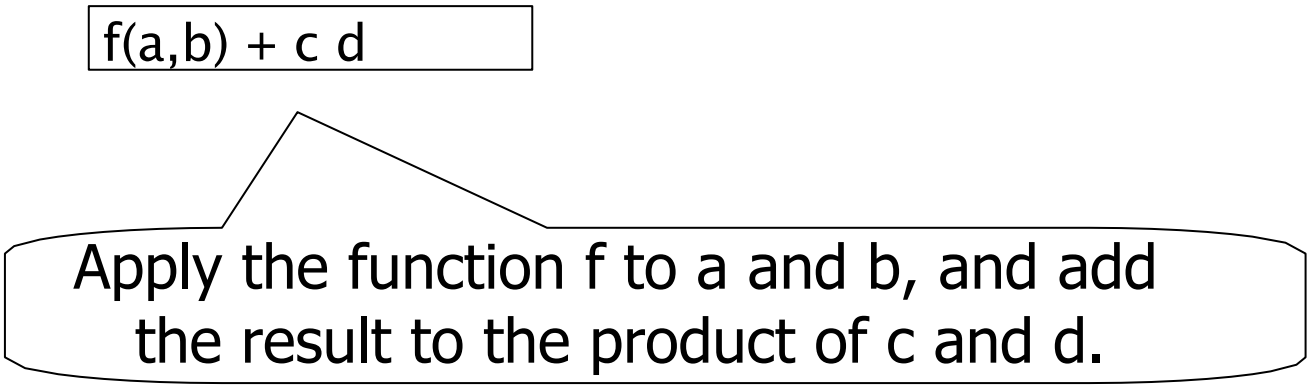
'A' : "Small Cat"

5 : [1,2,3]

1 : 2 : 3 : [] = [1,2,3]

Function Application

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.



In Haskell, function application is denoted using space, and multiplication is denoted using *.

函数声明：函数名，空格，...

```
f a b + c*d
```

As previously, but in Haskell syntax.

Moreover, function application is assumed to have higher priority than all other operators.

```
f a + b
```

高优先级

Means (f a) + b, rather than f (a + b).

Examples

Mathematics	Haskell
f(x)	f x
f(x,y)	f x y
f(g(x))	f (g x)
f(x,g(y))	f x (g y)
f(x)g(y)	f x * g y

Haskell Scripts

As well as the functions in the standard prelude, you can also define your own functions;

New functions are defined within a script, a text file comprising a sequence of definitions;

By convention, Haskell scripts usually have a `.hs` suffix on their filename.

My First Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running Hugs.
Start an editor, type in the following two function definitions, and save the script as Test.hs:

```
double x  = x + x
quadruple x = double (double x)
```

Leaving the editor open, in another window start up Hugs with the new script

```
andy$ ghci Test.hs
```

Now both Prelude.hs and Test.hs are loaded, and functions from both scripts can be used:

```
andy$ ghci
> quadruple 10
40
> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

Leaving GHC open, type the following two definitions, and resave:

```
factorial n = product [1..n]

average ns  = sum ns `div` length ns
```

Note:

div is enclosed in back quotes, not forward; x `f` y is just syntactic sugar for f x y.

GHC does not automatically detect that the script has been changed, so a reload command must be executed before the new definitions can be used:

```
> :r
[1 of 1] Compiling Main ( Test.hs, interpreted
)  Ok, modules loaded: Main.
> factorial 10
3628800
> average [1,2,3,4,5]
3
```

Naming Requirements

- Function and argument names must begin with a lower-case letter. For example:

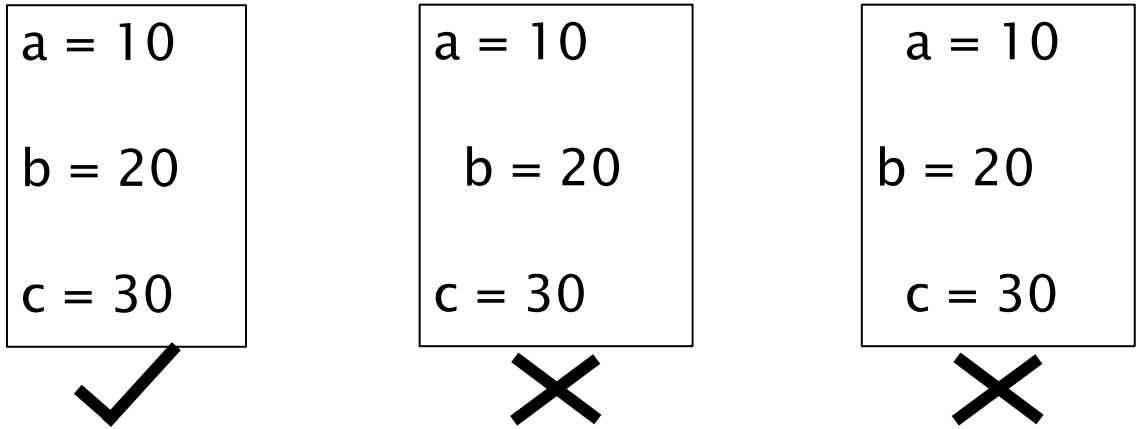
myFun	fun1	arg_2 x'	

- By convention, list arguments usually have an s suffix on their name. For example:

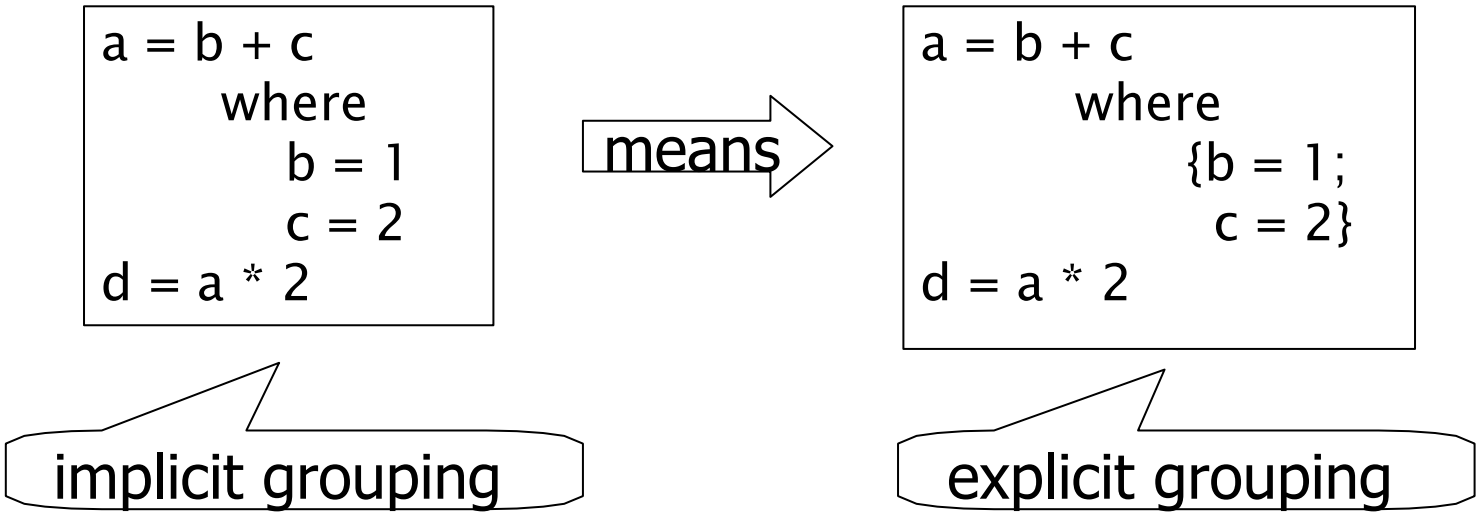
xs	ns	nss

The Layout Rule

In a sequence of definitions, each definition must begin in precisely the same column:



The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.



Exercises

(1) What are the errors?

(2) What is the result?

avoid explicit syntax

10 / 5 = 2

```
N = a `div` length xs  where
        a = 10
        xs = [1,2,3,4,5]
```

last [1,2,3,4] = 4

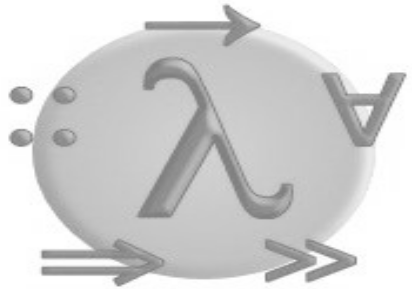
(3) Show how the library function last that selects the last element of a list can be defined using the functions introduced in this lecture.

(4) Can you think of another possible definition?

(5) Similarly, show how the library function init that removes the last element from a list can be defined in two different ways.

init [1,2,3,4] = [1,2,3]

PROGRAMMING IN HASKELL



Chapter 3 - Types and Classes

What is a Type?

A type is a name for a collection of related values. For example, in Haskell the basic type

Bool

contains the two logical values:

False True

Type Errors

statically typed

Applying a function to one or more arguments of the wrong type is called a type error.

```
> 1 + False
Error
```

1 is a number and False is a logical value, but + requires two numbers.

Types in Haskell

- If evaluating an expression e would produce a value of type t, then e has type t, written

```
e :: t
```

- Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference.

自动类型推导

- All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.
- In Hugs, the :type command calculates the type of an expression, without evaluating it:

```
> not False
True
> :type not False
not False :: Bool
```

Basic Types

Haskell has a number of basic types, including:

凡是类型首字母必须大写

Bool	- logical values
Char	- single characters
String	- strings of characters
Int	- fixed-precision integers 有界的
Integer	- arbitrary-precision integers 无界的
Float	- floating-point numbers

List Types

A list is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d']   :: [Char]
```

In general:

[t] is the type of lists with elements of type t.

Note:

- The type of a list says nothing about its length:

```
[False,True]  :: [Bool]

[False,True,False] :: [Bool]
```

Error : [1, True]

[2, 'a']

- The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

Tuple Types

对类型明确 二元数组 三元数组

A tuple is a sequence of values of different types:

```
(False,True)     :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

In general:

(t1,t2,…,tn) is the type of n-tuples whose ith components have type ti for any i in 1…n.

Note:

- The type of a tuple encodes its size:

```
(False,True)      :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

- The type of the components is unrestricted:

```
('a',(False,'b')) :: (Char,(Bool,Char))
 (True,['a','b'])                 :: (Bool,[Char])
```

Function Types

A function is a mapping from values of one type to values of another type:

```
not     :: Bool → Bool


isDigit :: Char → Bool
```

In general:

values of type t1 to values to type t2.
t1 → t2 is the type of functions that map

Note:

- The arrow → is typed at the keyboard as ->.

- The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

```
add :: (Int,Int) → Int
add (x,y) = x+y

zeroto  :: Int → [Int]
zeroto n  = [0..n]
```

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

```
add' :: Int → (Int → Int)
add' x y = x+y
```

add' takes an integer x and returns a function add' x. In turn, this function takes an integer y and returns the result x+y.

Note:

- add and add' produce the same final result, but add takes its two arguments at the same time, whereas add' takes them one at a time:

```
add  :: (Int,Int) → Int

add' :: Int → (Int → Int)
```

- Functions that take their arguments one at a time are called curried functions, celebrating the work of Haskell Curry on such functions.

- Functions with more than two arguments can be curried by returning nested functions:

```
mult     :: Int → (Int → (Int → Int))
mult x y z = x * y * z
```

mult takes an integer x and returns a function mult x, which in turn takes an integer y and returns a function mult x y, which finally takes an integer z and returns the result x*y*z.

Why is Currying Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function.

For example:

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]
drop 5 :: [Int] → [Int]
```

```
--lookup :: Database -> Key -> Value


let mylookup = lookup mydb


mylookup "Ryan"
mylookup "Jim"
mylookup "Pam"
```

Currying Conventions

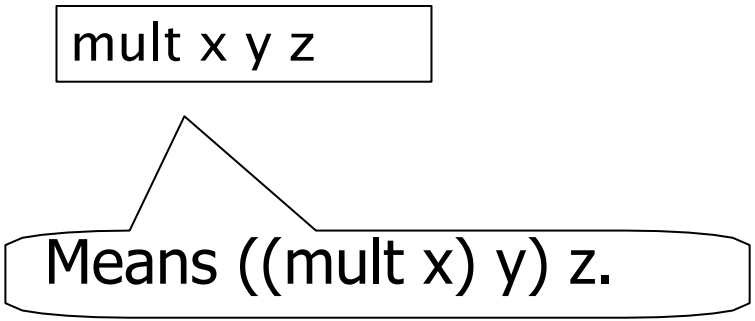
To avoid excess parentheses when using curried functions, two simple conventions are adopted:

- The arrow → associates to the right.

Int → Int → Int → Int

Means Int → (Int → (Int → Int)).

- As a consequence, it is then natural for function application to associate to the left.



Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

```
length :: [a] → Int
```

for any type a, length takes a list of values of type a and returns an integer.

Note:

- Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]  2

> length [1,2,3,4]
4
```

a = Bool

a = Int

- Type variables must begin with a **lower-case** letter, and are usually named a, b, c, etc.

- Many of the functions defined in the standard prelude are polymorphic. For example:

```
fst :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip :: [a] → [b] → [(a,b)]

id  :: a → a
```

Overloaded Functions

Num：包含实数和整数

Integral：仅包含整数。

A polymorphic function is called overloaded if its type contains one or more class constraints.

```
sum :: Num a ⇒ [a] → a
```

:info Num

for any numeric type a, sum takes a list of values of type a and returns a value of type a.

Eq：等于/不等

Ord：比大小

Ord 是 Eq 家族一员

“The difference between polymorphism and overloading is that polymorphism basically means that there is one algorithm for different types of operands whereas overloading means that for one symbol there are different implementations.”

https://mail.haskell.org/pipermail/beginners/2010-December/006057.html

Note:

- Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> sum [1,2,3]
6

> sum [1.1,2.2,3.3]
6.6

> sum ['a','b','c']  ERROR
```

a = Int

a = Float

Char is not a numeric type

EECS 368
Programming Language Paradigms

Dr. Andy Gill

Department of Electrical Engineering & Computer Science
University of Kansas

April 3, 2017

KU

Strings

A **String** is a sequence of characters enclosed in double quotes. Internally and explicitly, however, strings are represented as lists of characters.

```
"Hello" :: String
```

This means the same as `['H','e','l','l','o']`.

```
> "Hello" == ['H','e','l','l','o']
True
```

KU

Because strings are just special kinds of lists, any polymorphic function that operates on lists can also be applied to strings. For example:

```
length :: [a] -> Int
drop   :: Int -> [a] -> [a]
head   :: [a] -> a
```

```
> length "Hello"
5
> drop 2 "Hello"
"llo"
> head "Hello"
'H'
```

The Zip Function

A useful library function is zip, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] -> [b] -> [(a,b)]
```

```
> zip [a,b,c] [1,2,3,4]
[(a,1),(b,2),(c,3)]
```

Using zip we can define a function returns the list of all pairs of adjacent elements from a list:

```
pairs    :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

```
> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

Factorial

```
factorial  :: Int -> Int
factorial n = product [1..n]
```

factorial maps any integer n to the product of the integers between 1 and n.

Expressions are evaluated by a stepwise process of applying functions to their arguments.

```
factorial 4
```

=

Expressions are evaluated by a stepwise process of applying functions to their arguments.

```
factorial 4
```

=

```
product [1..4]
```

=

Expressions are evaluated by a stepwise process of applying functions to their arguments.

```
factorial 4
```

=

```
product [1..4]
```

=

```
product [1,2,3,4]
```

=

Expressions are evaluated by a stepwise process of applying functions to their arguments.

```
factorial 4
```

=

```
product [1..4]
```

=

```
product [1,2,3,4]
```

=

```
1*2*3*4
```

=

Expressions are evaluated by a stepwise process of applying functions to their arguments.

```
factorial 4
```

=

```
product [1..4]
```

=

```
product [1,2,3,4]
```

=

```
1*2*3*4
```

=

```
24
```

Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called recursive.

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

factorial maps 0 to 1, and any other positive integer to the product of itself and the factorial of its predecessor.

KU

```
factorial 3
```

=

```
factorial 3
```

=

```
3 * factorial 2
```

=

```
factorial 3
```

=

```
3 * factorial 2
```

=

```
3 * (2 * factorial 1)
```

=

```
factorial 3
```

=

```
3 * factorial 2
```

=

```
3 * (2 * factorial 1)
```

=

```
3 * (2 * (1 * factorial 0))
```

=

```
factorial 3
```

=

```
3 * factorial 2
```

=

```
3 * (2 * factorial 1)
```

=

```
3 * (2 * (1 * factorial 0))
```

=

```
3 * (2 * (1 * 1))
```

=

```
factorial 3
```
=
```
3 * factorial 2
```
=
```
3 * (2 * factorial 1)
```
=
```
3 * (2 * (1 * factorial 0))
```
=
```
3 * (2 * (1 * 1))
```
=
```
3 * (2 * 1)
```
=

```
factorial 3
```
=
```
3 * factorial 2
```
=
```
3 * (2 * factorial 1)
```
=
```
3 * (2 * (1 * factorial 0))
```
=
```
3 * (2 * (1 * 1))
```
=
```
3 * (2 * 1)
```
=
```
3 * 2
```
=

```
factorial 3
```
=
```
3 * factorial 2
```
=
```
3 * (2 * factorial 1)
```
=
```
3 * (2 * (1 * factorial 0))
```
=
```
3 * (2 * (1 * 1))
```
=
```
3 * (2 * 1)
```
=
```
3 * 2
```
=
```
6
```

- `factorial 0 = 1` is appropriate because 1 is the identity for multiplication:

$$1 * x = x = x * 1$$

- The recursive definition diverges on integers < 0 because the base case is never reached:

```
> factorial (-1)
Error: Control stack overflow
```

Why is Recursion Useful?

- Some functions, such as factorial, are simpler to define in terms of other functions.
- As we shall see, however, many functions can naturally be defined in terms of themselves.
- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

KU

Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on lists.

```
product          :: [Int] -> Int
product []        = 1
product (n:ns) = n * product ns
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

```
product [2,3,4]
```

=

```
product [2,3,4]
```

=

```
2 * product [3,4]
```

=

```
product [2,3,4]
```

=

```
2 * product [3,4]
```

=

```
2 * (3 * product [4])
```

=

```
product [2,3,4]
```

=

```
2 * product [3,4]
```

=

```
2 * (3 * product [4])
```

=

```
2 * (3 * (4 * product []))
```

=

```
product [2,3,4]
```
=
```
2 * product [3,4]
```
=
```
2 * (3 * product [4])
```
=
```
2 * (3 * (4 * product []))
```
=
```
2 * (3 * (4 * 1))
```
=

```
product [2,3,4]
```

=

```
2 * product [3,4]
```

=

```
2 * (3 * product [4])
```

=

```
2 * (3 * (4 * product []))
```

=

```
2 * (3 * (4 * 1))
```

=

```
24
```

Using the same pattern of recursion as in product we can define the length function on lists.

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

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.

```
length [2,3,4]
```
=
```
1 + length [2,3]
```
=
```
1 + (1 + length [3])
```
=
```
1 + (1 + (1 + length []))
```
=
```
1 + (1 + (1 + 0))
```
=
```
3
```

Using a similar pattern of recursion we can define the reverse function on lists.

```
reverse        :: [a] -> [a]
reverse []      = []
reverse (x:xs) = reverse xs ++ [x]
```

reverse maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.

```
reverse [1,2,3]
```
=
```
reverse [2,3] ++ [1]
```
=
```
(reverse [3] ++ [2]) ++ [1]
```
=
```
((reverse [] ++ [3]) ++ [2]) ++ [1]
```
=
```
(([] ++ [3]) ++ [2]) ++ [1]
```
=
```
[3,2,1]
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. Consider `zip`.

```
zip             :: [a] -> [b] -> [(a,b)]
zip []     _     =
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. Consider `zip`.

```
zip              :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip _      []     =
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. Consider `zip`.

```
zip                 :: [a] -> [b] -> [(a,b)]
zip []      _        = []
zip _       []       = []
zip (x:xs) (y:ys) =
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. Consider `zip`.

```
zip                 :: [a] -> [b] -> [(a,b)]
zip []     _         = []
zip _      []        = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Remove the first n elements from a list:

```
drop           :: Int -> [a] -> [a]
drop 0      xs = xs
drop n []      = []
drop n (_:xs) = drop (n-1) xs
```

Appending two lists:

```
(++)          :: [a] -> [a] -> [a]
[]      ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

KU

Quicksort

The quicksort algorithm for sorting a list of integers can be specified by the following two rules:

- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail values $\leq$ the head, sorting the tail values $>$ the head, and then appending the resulting lists on either side of the head value.

KU

Using recursion, this specification can be translated directly into an implementation:

```
qsort          :: [Int] -> [Int]
qsort []        = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a <- xs, a <= x]
      larger  = [b | b <- xs, b > x]
```

This is probably the simplest implementation of quicksort in any programming language!

KU

EECS 368
Programming Language Paradigms

Dr. Andy Gill

Department of Electrical Engineering & Computer Science
University of Kansas

April 5, 2017

KU

Set Comprehensions

In mathematics, the comprehension notation can be used to construct new sets from old sets.

$$\{x^2 | x \in \{1 \ldots 5\}\}$$

Set Comprehensions

In mathematics, the comprehension notation can be used to construct new sets from old sets.

$$\{x^2 | x \in \{1 \ldots 5\}\}$$

The set $\{1, 4, 9, 16, 25\}$, or all numbers x^2 such that x is an element of the set $\{1 \ldots 5\}$.

KU

List Comprehensions

In Haskell, the list comprehensions can be used to construct new lists from old lists.

```
[ x ^ 2 | x <- [1..5]]
```

输出函数　　限制条件(predicate)，由逗号分隔

```
[ x^2 | x <- [1..5], x>2, x<5]
= [ x^2 | x <- [1..5], x>2 && x<5]
```

List Comprehensions

In Haskell, the list comprehensions can be used to construct new lists from old lists.

```
[ x ^ 2 | x <- [1..5]]
```

The set `[1,4,9,16,25]`, or all numbers `x^2` such that `x` is an element of the set `[1..5]`.

- The expression `x <- [1..5]` is called a generator, as it states how to generate values for `x`.
- Comprehensions can have multiple generators, separated by commas. For example:

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

KU

- Changing the order of the generators changes the order of the elements in the final list:

```
> [(x,y) | y <- [4,5], x <- [1,2,3] ]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

- These are the for loops of Haskell.
- Multiple generators are like nested loops, with later generators as more deeply nested loops whose variables change value more frequently.

KU

Dependent Generators

Later generators can depend on the variables that are introduced by earlier generators.

```
[(x,y) | x <- [1..3], y <- [x..3]]
```

Dependent Generators

Later generators can depend on the variables that are introduced by earlier generators.

```
[(x,y) | x <- [1..3], y <- [x..3]]
```

The list `[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]`, or all pairs of numbers `(x,y)` such that `x` and `y` are elements of the list `[1..3]` and `y >= x`.

KU

Using a dependent generator we can define the library function that concatenates a list of lists:

```
concat     :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

```
> concat [[1,2,3],[4,5],[6]]
```

Using a dependent generator we can define the library function that concatenates a list of lists:

```
concat     :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

```
> concat [[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

KU

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x <- [1..10], even x]
```

KU

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x <- [1..10], even x]
```

The list `[2,4,6,8,10]` of all numbers `x` such that `x` is an element of the list `[1..10]` and `x` is even.

Using a guard we can define a function that maps a positive integer to its list of factors:

```
factors  :: Int -> [Int]
factors n = [ x | x <- [1..n]
                , n `mod` x == 0
                ]
```

```
> factors 15
[1,3,5,15]
```

A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

```
prime  :: Int -> Bool
prime n = factors n == [1,n]
```

```
> prime 15
False
> prime 7
True
```

Using a guard we can now define a function that returns the list of all primes up to a given limit:

```
primes  :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

Infinite Lists

```
> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,25,26,27,
28,29,30,31,32,33,34,35,36,37,38,39,
40,41,42,43,44,45,46,47,48,49,50,51,
52,53,54,55,56,57,58,59,60,61,62,63,
64,65,66,67,68,69,70,71,72,73,74,75,
76,77,78,79,80,81,82,83,84,85,86,87,
88,89,90,91,92,93,94,95,96,97,98,99,
100,101,102,103,104,105,106,107,108,
109,110,111,112,113,114,115,116,...
```

KU

Infinite Comprehensions

We can actually define a list of all primes.

```
primes  :: [Int]
primes = [x | x <- [2..], prime x]
```

```
> primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,
43,47,53,59,61,67,71,73,79,83,89,97,
101,103,107,109,113,127,131,137,139,
149,151,157,163,167,173,179,181,...
```

KU

EECS 368
Programming Language Paradigms

Dr. Andy Gill

Department of Electrical Engineering & Computer Science
University of Kansas

April 7, 2017

KU

Introduction

A function is called higher-order if it takes a function as an argument or returns a function as a result.

```
twice     :: (a -> a) -> a -> a
twice f x = f (f x)

intmap :: (Int -> Int) -> [Int] -> [Int]
intmap f xs = [ f x | x <- xs ]
```

`twice` and `intmap` are higher-order because they both take a function as one of their arguments.

KU

Why Are They Useful?

- Common programming idioms can be encoded as functions within the language itself.
- Domain specific languages can be defined as collections of higher-order functions.
- Algebraic properties of higher-order functions can be used to reason about programs.

KU

The Map Function

The higher-order library function called map applies a function to every element of a list.

```
map :: (a -> b) -> [a] -> [b]
```

For example:

```
> map (+1) [1,3,5,7]
[2,4,6,8]
```

The map function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = [f x | x <- xs]
```

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

KU

The Filter Function

The higher-order library function `filter` selects every element from a list that satisfies a predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
```

For example:

```
> filter even [1..10]
[2,4,6,8,10]
```

KU

The map function can be defined in a particularly simple manner using a list comprehension:

```
filter p xs = [x | x <- xs, p x]
```

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

```
filter p []       = []
filter p (x:xs)
   | p x          = x : filter p xs
   | otherwise    = filter p xs
```

KU

The Foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

`f` maps the empty list to some value v, and any non-empty list to some function $\oplus$ applied to its head and `f` of its tail.

For example:

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

and []     = True
and (x:xs) = x && and xs

product []     = 1
product (x:xs) = x * product xs
```

The higher-order library function `foldr` (fold right) encapsulates this simple pattern of recursion, with the function $\oplus$ and the value v as arguments.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
sum     xs = foldr (+) 0 xs
product xs = foldr (*) 1 xs
or      xs = foldr (||) False xs
and     xs = foldr (&&) True xs
```

所有遍历 List 中元素按此回传一个值的操作

Foldr itself can be defined using recursion:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

One way to think about `foldr` is non-recursively, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.

For example:

```
sum [1,2,3]
```

=

```
foldr (+) 0 [1,2,3]
```

=

```
foldr (+) 0 (1:(2:(3:[])))
```

{ Replace all the `(:)` with `(+)`, and the `[]` with `0`. }

=

```
1+(2+(3+0))
```

=

```
6
```

For example:

```
product [1,2,3]
```

=

```
foldr (*) 1 [1,2,3]
```

=

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

{ Replace all the `(:)` with `(*)`, and the `[]` with `1`. }

=

```
1*(2*(3*1))
```

=

```
6
```

Other Foldr Examples

Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the length function:

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

For example:

```
length [1,2,3]
```

=

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

{ Replace all the `(:)` with `(\ _ n -> 1 + n)`, and the `[]` with `0`. }

=

```
1+(1+(1+0))
```

=

```
3
```

Hence, we have:

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

Why Is `foldr` Useful?

- Some recursive functions on lists, such as sum, are simpler to define using `foldr`.
- Properties of functions defined using `foldr` can be proved using algebraic properties of `foldr`.
- Advanced program optimizations can be simpler if `foldr` is used in place of explicit recursion.

...

KU

Why Is `foldr` Useful?

- Some recursive functions on lists, such as sum, are simpler to define using `foldr`.
- Properties of functions defined using `foldr` can be proved using algebraic properties of `foldr`.
- Advanced program optimizations can be simpler if `foldr` is used in place of explicit recursion.
- Real functional programmers just use recursion.
- Real functional programmers do use higher-order functions.

KU

Other Library Functions

The library function `(.)` returns the composition of two functions as a single function.

```
(.)    :: (b -> c) -> (a -> b) -> (a -> c)
f . g  = \ x -> f (g x)
```

For example:

```
odd :: Int -> Bool
odd  = not . even
```

The library function `all` decides if every element of a list satisfies a given predicate.

```
all      :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]
```

For example:

```
> all even [2,4,6,8,10]
True
```

Dually, the library function any decides if at least one element of a list satisfies a predicate.

```
any      :: (a -> Bool) -> [a] -> Bool
any p xs = or [p x | x <- xs]
```

For example:

```
> any isSpace "abc def"
True
```

The library function takeWhile selects elements from a list while a predicate holds of all the elements.

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

For example:

```
> takeWhile isAlpha "abc def"
"abc"
```

Also a dropWhile function.

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

Finally, data-structures can also contain functions.

```
data Move = Move (Int -> Int)


doMove :: Move -> Int -> Int
doMove (Move f) n = f n
```

Polymorphic arguments can also be functions.

```
funs :: [Int -> Int]
funs = [ succ,
         (+ 1),
         (* 100),
         \ x -> x * 2]
```

Functions are values, too!

- You can pass them to functions.
- You can return them from functions.
- You can store them in data-structures.
- You can never *update* them, only pass them around, and apply them.

KU

EECS 368
Programming Language Paradigms

Dr. Andy Gill

Department of Electrical Engineering & Computer Science
University of Kansas

April 14, 2017

KU

Type Declarations

In Haskell, a new name for an existing type can be defined using a type declaration.

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

`String` is a synonym for the type `[Char]`.

Type declarations can be used to make other types easier to read. For example, given

```
type Pos = (Int,Int)
```

we can define

```
origin     :: Pos
origin      = (0,0)

left       :: Pos -> Pos
left (x,y) = (x-1,y)
```

Like function definitions, type declarations can also have parameters. For example, given

```
type Pair a = (a,a)
```

we can define

```
mult       :: Pair Int -> Int
mult (m,n) = m*n

copy       :: a -> Pair a
copy x      = (x,x)
```

Type declarations can be nested:

```
type Pos   = (Int,Int)     -- GOOD

type Trans = Pos -> Pos    -- GOOD
```

However, they cannot be recursive:

```
type Tree = (Int,[Tree])   -- BAD
```

Data Declarations

A completely new type can be defined by specifying its values using a data declaration.

```
data Bool = False | True
```

`Bool` is a new type, with two new values `False` and `True`.

Note:

- The two values False and True are called the constructors for the type Bool.
- Type and constructor names must begin with an upper-case letter.
- Data declarations are similar to context free grammars. The former specifies the values of a type, the latter the sentences of a language.

KU

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers      :: [Answer]
answers       = [Yes,No,Unknown]

flip         :: Answer -> Answer
flip Yes      = No
flip No       = Yes
flip Unknown  = Unknown
```

KU

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square          :: Float -> Shape
square n         = Rect n n
area            :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

KU

Note:

- Shape has values of the form Circle r where r is a float, and Rect x y where x and y are floats.
- Circle and Rect can be viewed as functions that construct values of type Shape:

```
-- Not a definition
Circle :: Float -> Shape
Rect   :: Float -> Float -> Shape
```

KU

Not surprisingly, data declarations themselves can also have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

we can define:

```
safediv    :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead   :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

KU

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be recursive.

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors `Zero :: Nat` and `Succ :: Nat -> Nat`.

Note:

- A value of type `Nat` is either `Zero`, or of the form `Succ n` where `n :: Nat`. That is, `Nat` contains the following infinite sequence of values:

```
Zero
```

```
Succ Zero
```

```
Succ (Succ Zero)
```

⋮

Note:

- We can think of values of type Nat as natural numbers, where `Zero` represents `0`, and `Succ` represents the successor function `1+`.
- For example, the value

```
Succ (Succ (Succ Zero))
```

  represents the natural number

```
1 + (1 + (1 + 0))
```

Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int           :: Nat -> Int
nat2int Zero       = 0
nat2int (Succ n) = 1 + nat2int n

int2nat           :: Int -> Nat
int2nat 0          = Zero
int2nat n          = Succ (int2nat (n - 1))
```

Two naturals can be added by converting them to integers, adding, and then converting back:

```
add     :: Nat -> Nat -> Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

The recursive definition for add corresponds to the laws

$$0 + n = n$$

and

$$(1 + m) + n = 1 + (m + n)$$

KU

Using recursion, an expression tree can be defined using:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

One example of such a tree written in Haskell is

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

KU

Using recursion, it is now easy to define functions that process expressions. For example:

```
size            :: Expr -> Int
size (Val n)     = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval            :: Expr -> Int
eval (Val n)     = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Note:

- The three constructors have types:

```
-- Not a definition
Val :: Int -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr
```

Using recursion, a binary tree can be defined using:

```
data Tree = Leaf Int
          | Node Tree Int Tree
```

One example of such a tree written in Haskell is

```
Node (Node (Leaf 1) 3 (Leaf 4))
     5
     (Node (Leaf 6) 7 (Leaf 9))
```

We can now define a function that decides if a given integer occurs in a binary tree:

```
occurs                    :: Int -> Tree -> Bool
occurs m (Leaf n)          = m==n
occurs m (Node l n r)      = m==n
                             || occurs m l
                             || occurs m r
```

In the worst case, when the integer does not occur, this function traverses the entire tree.

Search trees have the important property that when trying to find a value in a tree we can always decide which of the two sub-trees it may occur in:

```
occurs                    :: Int -> Tree -> Bool
occurs m (Leaf n)                 = m==n
occurs m (Node l n r)     | m==n  = True
                          | m<n   = occurs m l
                          | m>n   = occurs m r
```

This new definition is more efficient, because it only traverses one path down the tree.
What is the precondition for `Node`?

Finally consider the function `flatten` that returns the list of all the integers contained in a tree:

```
flatten                 :: Tree -> [Int]
flatten (Leaf n)         = [n]
flatten (Node l n r) = flatten l
                       ++ [n]
                       ++ flatten r
```

A tree is a search tree if it flattens to a list that is ordered.

KU