Topic 5:

Functional Languages

Q: What do communists and functional programmers have in common?

A: They hate classes!

Language Paradigms So Far

Recall:

- Imperative languages are tied to the Von Neumann architecture
- Many object–oriented languages are built upon imperative languages
  - As such, they share the same historical perspectives

$\Rightarrow$ Moving beyond that history requires adopting a different perspective on computation.

Mathematical Function Basics (1 / 3)

Hopefully just review from Discrete Math!

Definition: (Binary) Relation

A relation from set X to set Y is a subset of the Cartesian Product of X (the domain) and Y (the codomain).

Definition: Function

Mathematical Function Basics (2 / 3)

Let $f : X \to Y$ be a function, and $f(n) = p$ [$(n, p) \in f$].

- X is the _____ of f
- Y is the _____ of f
- f _____ X to Y
- p is the _____ of n
- n is the _____ of p
- f's _____ is the set of all images of X's elements

Note: A function's range need not equal its codomain.

Mathematical Function Basics (3 / 3)

Example(s): Consider the relation $\{(1,1),(2,8),(3,27)\}$

- Is this relation also a function?
- What is its domain?
- What is its codomain?
- What is its range?
- What is its mapping?

Function Definition v. Application

Definition: Function Definition

. . . . . .

Definition: Function Application

. . . . . .

Example(s):

Lambda Expressions (1 / 3)

A little history:

- Alonzo Church invented lambda calculus in the 1930s.
- In 1936, he published a version now known as untyped lambda calculus, which became the foundation of functional programming languages.
- However, because typing is useful in programming, functional languages opt for typed lambda calculus.

In computing, the utility of lambda calculus is as a mechanism to express computation via functions.

Lambda Expressions (2 / 3)

Definition: Lambda Expression

Several syntaxes are used, including:

Lambda Expressions (3 / 3)

An application of a lambda expression adds a constant as an argument:

Higher–Order Functions (1 / 3)

(a.k.a. Functional Forms, a.k.a. Functors)

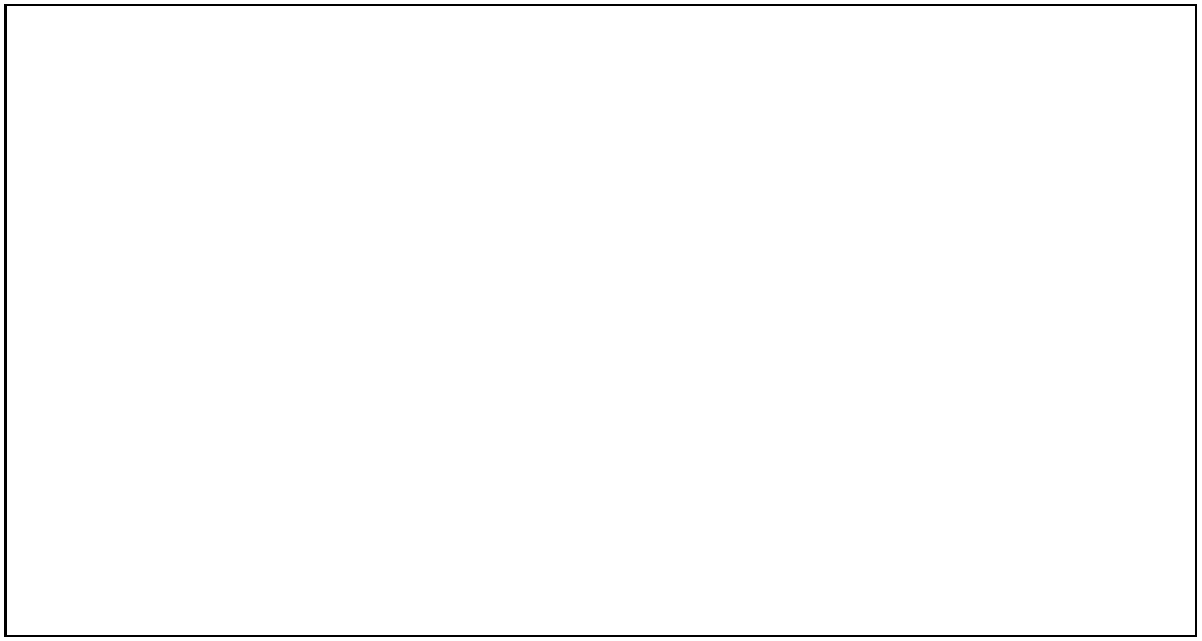
Higher–Order Functions allow functions to be passed as parameters, and to produce functions as results.

Example(s): (that you may have already seen/used!)

Higher–Order Functions (2 / 3)

Moving toward functional languages, here's an example that is probably familiar from Discrete Math:

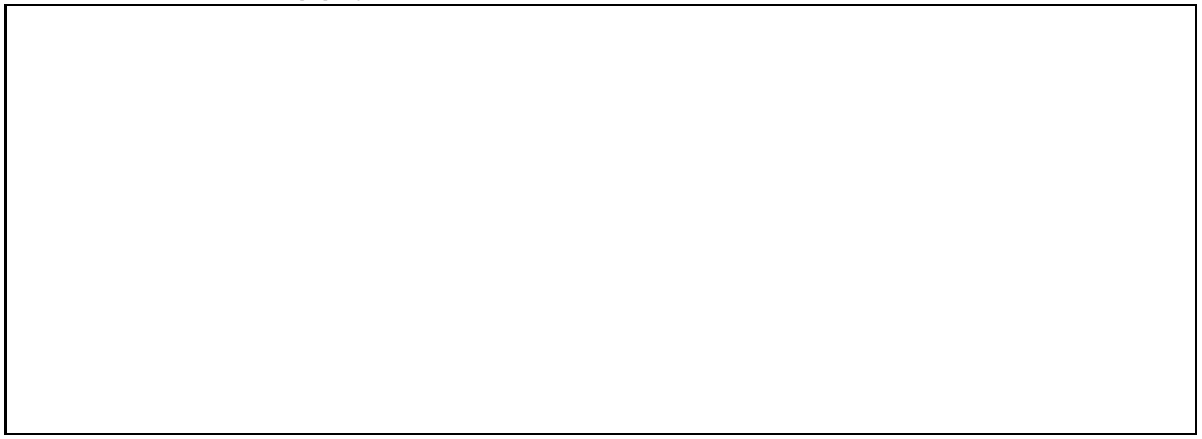
Example(s): Functional Composition



Higher–Order Functions (3 / 3)

Another HOF example, probably less–familiar:

Example(s): Apply–to–all (α)



“But, Imperative & OO Languages have Functions, too!”

That’s true. However:

Characteristics of Functional Languages

. . . include, but are not limited to:

- No (mutable!) variables
-
-
-
-
-
-

Side Effects (1 / 4)

Definition: (Functional) Side Effect

A more general definition of ‘side effect:’

Side Effects (2 / 4)

Example(s):

Side Effects (3 / 4)

Example(s):

Consider this C language program fragment:

```
int g = 2;
int inc() { return ++g; }
int main (int argc, char *argv[]) {



}
```


Side Effects (4 / 4)

Continuing from the last slide's example . . .

Question: When does the program fetch `g` from memory to compute the product in `main()`? Before or after the call to `inc()`?

Haskell

Named in honor of logician _____ .

Specifically, Curry worked on *combinatory logic*, which is based on *combinators*, a type of higher–order function.

- Was the basis for some early functional languages
- Can be viewed as a variant of λ–calculus

Origin of the Haskell Language

- A ‘language by committee’
- Built on a proprietary language named Miranda
- Version 1.0 was released in 1990
- Current version:

Haskell's Place in Functional Languages

Haskell:

- Is purely functional (i.e., not imperative, not OO)
-
-
-
-
-
-

No Language is Perfect!

Disadvantages of Haskell include:

-
-
-

Haskell Resources

- www.haskell.org — official home of Haskell
- hoogle.haskell.org — library search, etc.
- Class web site — links to free texts, sample code
- Compiler:
- REPL:

Notes on Using ghci

- Ctrl–D to exit! (or `:q`)
- Prompt: Now: **`ghci>`** Was: **`Prelude>`**
-
-
-
-
-
-

ghci Demo (1 / 3)

I'll do these examples 'live,' but here they are for reference:

```
ghci> :set +t
ghci> 1 == 1
True
it::Bool
ghci> 3 / 4
0.75
it :: Fractional a => a
```

(Believe it or not, a more complicated example's typing will be easier to explain . . .)

ghci Demo (2 / 3)

The function `recip` finds the reciprocal of its argument.

```
ghci> recip 2/3
0.16666666666666666   ← recip 2 = 1/2; (1/2)/3 = 1/6
it :: Fractional a => a
ghci> recip (2/3)
1.5
it :: Fractional a => a
ghci> :t recip
recip :: Fractional a => a -> a
```

(Yup, more complicated! But what does it mean?!?)

Aside: Components of a Type Signature

```
recip :: Fractional a => a -> a
```

`recip`

`Fractional`

`a`

`=>`

`->`

ghci Demo (3 / 3)

And so, our first type signature example . . .

```
ghci> 3 / 4
0.75
it :: Fractional a => a
```

. . . says that the result of 3 / 4 (0.75) is of type Fractional.

What if a function has multiple arguments?

```
ghci> logBase 2 16
4.0
ghci> :t logBase
logBase :: Floating a => a -> a -> a
```

In `a -> a -> a`, the first two 'a's represent the arguments and the third represents the result.

Types and Type Classes (1 / 3)

Some basic Haskell types:

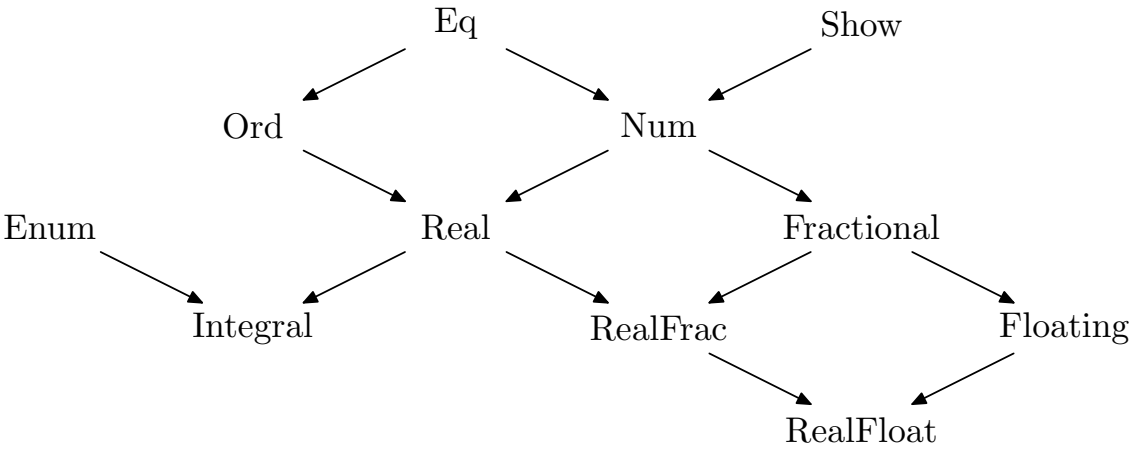
-
-
-
-

Types and Type Classes (2 / 3)

Haskell Type Classes $\neq$ Object–Oriented Classes!

Types and Type Classes (3 / 3)

This is the hierarchy of (most of) the type classes in the Prelude, Haskell's standard module (collection of declarations):



Source: Section 6.3 of the Haskell 2010 Report

Basic Math Functions (1 / 3)

- $+$, $-$, $*$, and **`negate`** (unary negation)
- $/$
- **`div`**, **`quot`**, **`mod`**, **`rem`**

Basic Math Functions (2 / 3)

Oddity #1: Unary negation in Haskell is a little weird.

```
3 - 2          ⇒
3 + -2         ⇒
-2 + 3         ⇒
3 + (-2)       ⇒
3 + negate 2   ⇒
```


Basic Math Functions (3 / 3)

Oddity #2: All math functions can be used as prefix operators, but symbolic functions need parentheses:

```
- 3 2          ⇒
(-) 3 2        ⇒
subtract 3 2   ⇒
```

Oddity #3: Using named math functions as infix operators requires backquotes:

```
div 18 7    ⇒
18 div 7    ⇒
18 `div` 7  ⇒
```

Lists

. . . are homogeneous and delimited with square brackets.

```
ghci> d = [0,1,2]
ghci> d
[0,1,2]
ghci> :t d
d :: Num a => [a]

ghci> v = ['a','e','i']
ghci> v
"aei"
ghci> :t v
v :: [Char]
```


Basic List Functions (1 / 4)

Haskell has lots of list functions! Here are the essentials:

- `length` — the quantity of elements in a list

- `null` — tests to see if a list is empty

Basic List Functions (2 / 4)

- `head` and `tail` — 1st element and list of the rest

- `last` and `init` — last element and list of all–but–last

Basic List Functions (3 / 4)

- `++` — performs list concatenation

- `:` — “cons”; prepend a value to a list

Basic List Functions (4 / 4)

- `!!` — access a list element by its (0–based) index

- `elem` — does a list contain an element?

Comparing List Contents

Basic operators: `==`, `<`, `>`, `<=`, `>=`, `/=` (not equals)

Standard lexicographic ordering applies:

```
"UA" > "ASU"     ⇒
"do" /= "doh"    ⇒
"do" < "doh"     ⇒
[1,2] == [2,1]   ⇒
[1,2] <= [2,1]   ⇒
```

Lists of Lists

We've already seen some examples; let's dig a bit deeper:

```
ghci> x = [[0],[1,2]]
ghci> :t x
x :: Num a => [[a]]

ghci> x ++ [3]
error: No instance for...
```


User–Defined Functions (1 / 3)

Defining simple functions is easy in Haskell:

```
ghci> pythag a b = sqrt(a^2 + b^2)
ghci> :t pythag
pythag :: Floating a => a -> a -> a


ghci> pythag 3 4
5.0
```

User–Defined Functions (2 / 3)

Prefer to choose a type yourself? Two options:

Option #1: Inline Type Annotation

Easy! Append `::` `<type>` to the one-line declaration:

```
ghci> pythag a b = sqrt(a^2 + b^2) :: Double
ghci> :t pythag
pythag :: Double -> Double -> Double
```


User–Defined Functions (3 / 3)

Option #2: Add a Function Type Signature Line

This is the typical way to declare functions in programs.

Format:
```
name :: type(s)
name = expression
```

The first line is the function signature; the second is the binding, and `::` is the type signature operator.

Here's the pythag function declared in this format:

```
pythag :: Double -> Double -> Double
pythag a b = sqrt(a^2 + b^2)
```

pythag2.hs

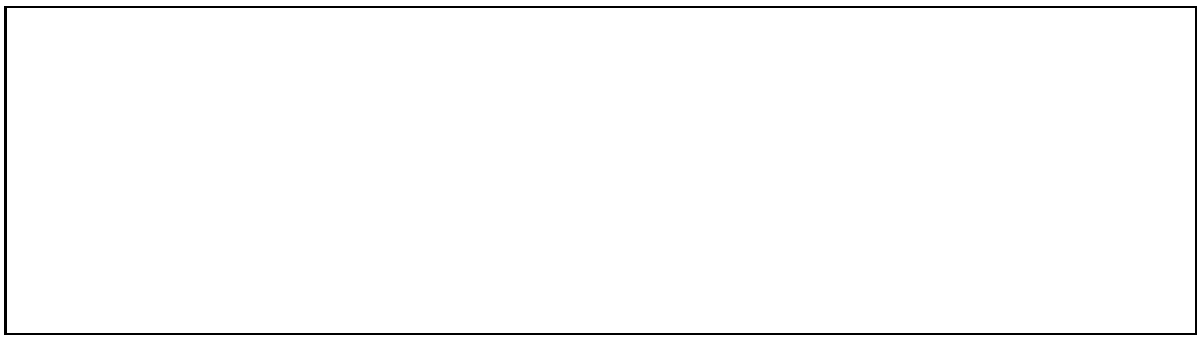
Selection Expressions (1 / 5)

(Why not 'statements'? Expressions produce values!)

Haskell has three; we'll cover two now and save the third for later.

(1) **`if - else`**

Example(s):



Selection Expressions (2 / 5)

(1) **`if - else`** (continued)

A few more details:

- Why did I name it **`myOdd`**? Haskell's Prelude already has a function named **`odd`**.

Selection Expressions (3 / 5)

(2) Guards

Remember piecewise functions from math? For example, the absolute value function:

$$|x| = \begin{cases} -x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

Haskell's guards allow us to write functions that mimic the syntax of piecewise functions.

Selection Expressions (4 / 5)

(2) Guards (continued)

Example(s): The guard version of absolute value.

guards.hs

Selection Expressions (5 / 5)

(2) Guards (concluded)

Notes:

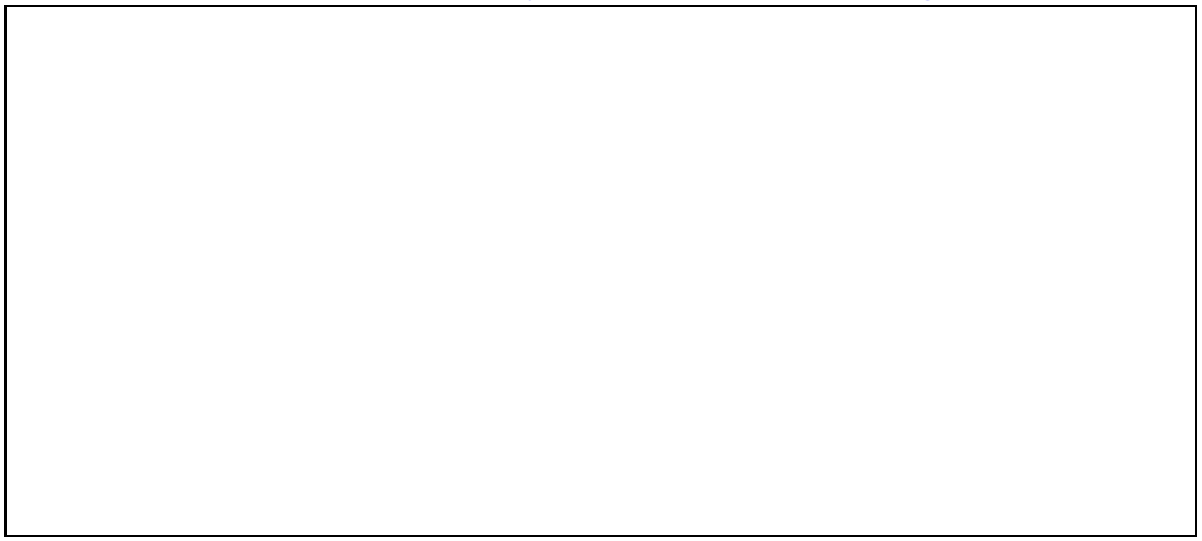
- Guards are attempted top–down. The first one found with a true guard has the result of its corresponding expression returned.

Recursion

Recall: We cannot iterate in a purely functional language.

No problem; we ♡ recursion!

Example(s): Factorial! (stay awake, there's a bug!)



Using trace to Help Find Logic Errors

This version is useful in functions; Haskell has others.

(1) `import Debug.Trace`

(2) Insert as the function's second line:

fact.hs

Aside: Using Multiply as a Prefix Operator

Reminder: Prefix symbol operators are tricky!

Legal or Error?

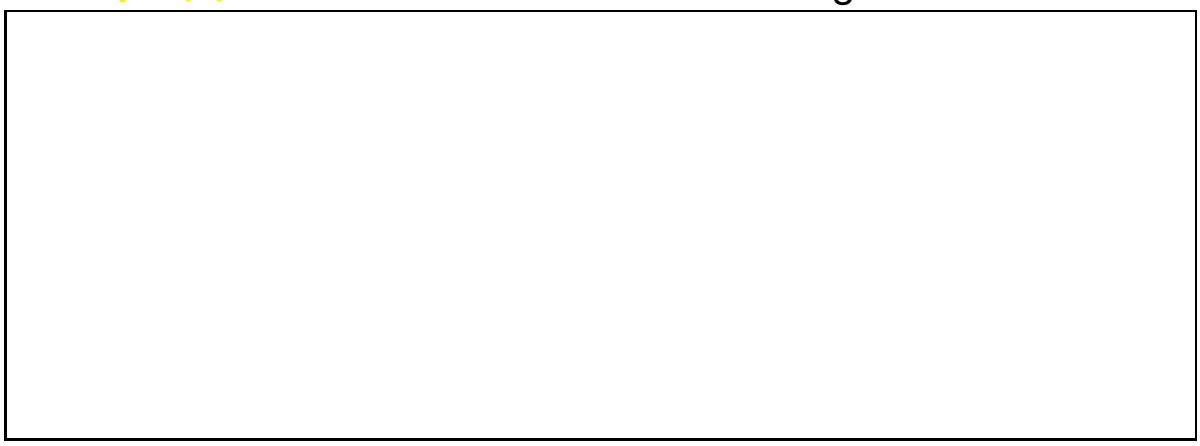
1. `* n fact(n-1)`

2. `(*) n fact(n-1)`

3. `(*) n (fact(n-1))`

Another Recursion Example

Example(s): Count the characters in a string.



Basic I/O (1 / 4)

We've seen `putStr` and `putStrLn` for printing strings.

Another: `print` (yes, I already used it in fact.hs . . .)

```
putStrLn "Hi"  ⇒ Hi
print "Hi"     ⇒ "Hi"
print 'H'      ⇒ 'H'
print 3.14     ⇒ 3.14
print [2,4,6]  ⇒ [2,4,6]
```

Basic I/O (2 / 4)

The function **`show`** sounds like an I/O function, but is not!

Its purpose:

`show 1.2` $\Rightarrow$

`show True` $\Rightarrow$

`show "Hi"` $\Rightarrow$

Basic I/O (3 / 4)

`show` pairs nicely with **`putStr`** and **`putStrLn`**
to display function results.

Example(s):

Basic I/O (4 / 4)

For keyboard input, **`getLine`** may be all you'll need.

```
putStr "What is your name? "


putStrLn ("Hi, " ++ name ++ "!")
```


Functional Languages and I/O

Known Problem: I/O actions produce side–effects.

Haskell's I/O Side–Effect Solution (1 / 5)

Part 1: Provide a new I/O type: `IO`

`IO`'s purpose:

Actions of type IO can be viewed as separate imperative programs that perform I/O actions and return resulting info/data to our programs.

Haskell's I/O Side–Effect Solution (2 / 5)

Part 2: Ignore return values from I/O functions

I/O functions are functions, and so do return values.
But, when we print to standard output, for example, there's nothing helpful for the I/O function to return to the caller.

Haskell's I/O Side–Effect Solution (3 / 5)

Part 3: Provide **do** notation, which:

(a) Allows I/O–using programs to be sequenced, similar to how we'd write them in imperative languages
(Actually: 'syntactic sugar' for the sequence operator (**>>**))

(b)

Haskell's I/O Side–Effect Solution (4 / 5)

Part 3: **do** notation, continued.

But that's not all: The 'cleaning' that **<–** provides isn't good enough for Haskell! It adds one more layer via **do** to protect our functional code from nasty I/O side–effects:

(c)

Haskell's I/O Side–Effect Solution (5 / 5)

A few notes to wrap this up:

- `a = ...` and `a <- ...` are different.
  - `=` gives a name to an expression
  - `<-` extracts a value from an IO monad

    (More about monads soon!)
- `do` must end with an I/O action
  - `return ()` makes a null I/O action for such situations.

    (Haskell's return is different from Java's!)

Now a Haskell Main Function Makes Sense!

```
main :: IO ()
main = do
    putStr "Name? "
    name <- getLine
    putStrLn("Hi, " ++ name ++ "!")
```

Text File I/O

We don't open or close files in Haskell, we just read & write.

Reading strings from a file: `readFile`

Returns a string containing the file's content

Writing a string to a new file: `writeFile`

Creates a file with the provided content

Appending strings to a file: `appendFile`

Adds provided content to the end of the file

I/O: Behind the Scenes

(1) Monads

-
- `IO` is an instance of class `Monad`.
- Others include `Maybe`, `Error`, and `List`.

(2) Bind (`>>=`)

-
- Bind allows:
  - Non–determinism
  - Return values to be skipped
  - State info to be maintained
  - Exception–handling

List Comprehension (1 / 2)

Remember set builder notation from discrete math?

Example: $\{ 3x + 1 \mid 0 < x \leq 9 \ \wedge$ odd(x) $\}$

describes the set $\{4, 10, 16, 22, 28\}$.

In functional languages, a basic list comprehension is the same idea: Apply a function to each of a collection of values and return a collection of results.

The same example in Haskell syntax:

List Comprehension (2 / 2)

Another example: Remember Cartesian Product of sets?

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

We can use list comprehension to generate a list of all of the tuples — the Cartesian Product:

Pattern Matching (1 / 4)

Remember **cons** (**:**)? Haskell uses the same syntax (**<value>:<list>**) to split a list into its 1st element and the remaining elements — the same idea as **head** and **tail**, but tighter syntax.

Example(s): Reversing the content of a list

Pattern Matching (2 / 4)

Here is a list size function using guards:

```
mySize1 :: [a] -> Int
mySize1 lst
  | null lst = 0
  | otherwise = 1 + mySize1 (tail lst)
```

And here is a version using patterns and the *wildcard*:

Pattern Matching (3 / 4)

In Haskell, a pattern can be:

- A literal (e.g., `6`, `'c'`, `False`) — Haskell will match exactly that value
- A name — will match any argument value
- The wildcard (`_`) — will also match any argument value, but we cannot refer to it later
- A list pattern (e.g., `[a,b,c]`, `(x:xs)`) — will match list components

. . . and a few others that we won't be covering.

Pattern Matching (4 / 4)

Here's a third version of MySize, using both a wildcard and a list comprehension:

```
mySize3 :: [a] -> Int
mySize3 lst = sum [1 | _ <- lst]
```

Higher–Order Functions (1 / 9)

(1) `map`

Remember **Apply-to-all** (α) from our introduction to functional languages? Haskell calls it **`map`**. The definition of **`map`** is worth examining:

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

We know almost all of the Haskell features that this definition uses! Even so, we will study it line by line.

Higher–Order Functions (2 / 9)

The first line introduces something new:

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

`(a -> b)` represents a function accepting values of type **`a`** and returning values of type **`b`**. (What are the types of **`a`** and **`b`**? We don't know, and we don't care!)

What are **`map`**'s arguments, and what does it return?

Higher–Order Functions (3 / 9)

Second line: `map _ [] = []`

Third line: `map f (x:xs) = f x : map f xs`

Higher–Order Functions (4 / 9)

Here are a few map examples:

```
ghci> map (*2) [1,2,3]
[2,4,6]

ghci> map (++ "?") ["who", "what", "how"]
["who?", "what?", "how?"]

ghci> threeOne x = x*3 + 1
ghci> map threeOne [1,3,5,7,9]
[4,10,16,22,28]

ghci> map threeOne [x | x <- [1..9], odd(x)]
[4,10,16,22,28]
```

Higher–Order Functions (5 / 9)

(2) `filter`

`filter` accepts a Boolean function and uses it to collect only the list elements for which the function evaluates to True.

```
ghci> filter (>=6) [1..9]
[6,7,8,9]

ghci> filter odd [1..9]
[1,3,5,7,9]

ghci> threeOne x = x*3 + 1
ghci> map threeOne
[x | x <- filter odd (filter (<10) [1..100])]
[4,10,16,22,28]
```


Higher–Order Functions (6 / 9)

(3) `zip` and `zipWith`

`zip` pairs into tuples members of two lists (like teeth in a zipper), until a list is exhausted. (**`zip`** is not an HOF!)

```
ghci> zip [1,2,3] [True, False]
[(1,True),(2,False)]
```

`zipWith` is like a merging of **`map`** and **`zip`**: Apply a function to the corresponding elements of a pair of lists.

```
ghci> zipWith (++) ["in","un"] ["sane","wise"]
["insane","unwise"]
```

Aside: Lambda Functions in Haskell (finally!)

Recall the general λ form: $(\lambda x \mapsto x^3)(2)$

In Haskell, the corresponding lambda syntax is:

```
\x -> x * x * x
```

Haskell calls it an anonymous function. They are useful in higher–order functions:

```
ghci> map (\x -> 3*x+1) [1,3,5,7,9]
[4,10,16,22,28]
```


Higher–Order Functions (7 / 9)

(4) `foldl` and `foldr`

`foldl` and **`foldr`** compute a value by 'folding' a list — they each apply the given function to the list members either left–to–right (**`foldl`**) or right–to–left (**`foldr`**). Why both left and right? Sometimes the direction matters.

Example: Consider totaling a list of numbers:

```
total [1,2,3]
```

We know how to do this iteratively — Start with a sum of 0, add each element into the total:

0+1 = 1 $\leadsto$ 1+2 = 3 $\leadsto$ 3+3 = 6

Higher–Order Functions (8 / 9)

(4) `foldl` and `foldr` (continued)

To do this functionally, we need:

- A binary function accepting the current sum & a list value
  ⇒ Lambda time! `\accum x -> accum + x`
- The current accumulation quantity (0)
- The remaining list of values to be totaled

```
ghci> foldl (\accum x -> accum+x) 0 [1,2,3]
6
ghci> foldr (\accum x -> accum+x) 0 [1,2,3]
6
```


Higher–Order Functions (9 / 9)

(4) `foldl` and `foldr` (concluded!)

Switching from plus to minus will highlight the L/R difference.

But first, an observation: `\accum x -> accum+x` is just addition; we don't need a lambda function at all! For addition and subtraction, we can use `+` and `-`.

```
ghci> foldl (-) 0 [1,2,3]
```

```
ghci> foldr (-) 0 [1,2,3]
```

Streamlining Function Calls (1 / 2)

Consider this simple example: Sorting in descending order.

```
ghci> import Data.List    (supplies sort)
ghci> descending lst = reverse (sort lst)
```

We can compress function call sequences in three ways:

(a) Infix Application (**$**)

(b) Functional Composition [Recall: $(f \circ g)(x) \equiv f(g(x))$]

Streamlining Function Calls (2 / 2)

(c) Pointfree Style (a.k.a. Tacit Programming)

Background: In topology, there are functions on spaces of "points." Points are values. Thus, "pointfree" means that we don't include the values (our function arguments) in our expressions.

Continuing our example, our functional composition form:

```
descending lst = (reverse . sort) lst
```

can be written without either `lst` argument:

```
descending = (reverse . sort)
```

and without `lst` we don't need the paranetheses:

```
descending = reverse . sort
```

Partial Application

Surprise: All Haskell functions are unary (take one argument)!

```
ghci> (*) 2 3
6
ghci> (*2) 3
6
```

That ought to look familar from prior HOF examples, e.g.:

```
ghci> map (*2) [1,2,3]
ghci> filter (>=6) [1..9]
```

Pointfree style depends on this unary function idea:

```
ghci> times10 = map (*10)
```


Curried Functions

Taking arguments one at a time is called currying.
(Yes, something else named after Haskell Brooks Curry.)

Now you're ready to learn the truth about function types:

```
ghci> :t (*)
(*) :: Num a => a -> a -> a
```

`->` is right–associative, so there's an implicit set of parens:

```
(*) :: Num a => a -> (a -> a)
```

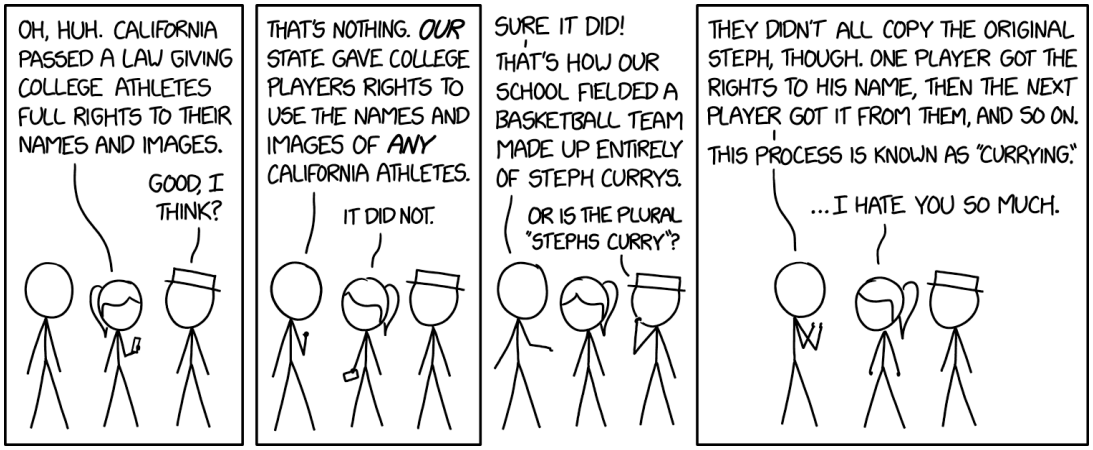
Which means that `(*)` actually takes one argument (the first `a`) and returns the function `(a -> a)`, which also accepts a single argument.

That's why the syntax of `(*2) 3` works!

There's a Lot More to Know about Haskell!

Such as creating types and type classes, applicative functors, monads beyond IO, contents of Haskell's modules, . . .

Hopefully, you now have enough of a functional foundation to learn those on your own, if you wish!



https://xkcd.com/2210