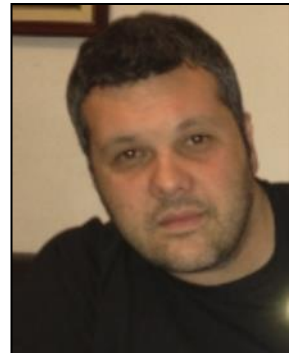


Uma Breve Introdução a Programação Funcional com Matemática Discreta

Fabio Galuppo, M.Sc.

<http://fabiogaluppo.com>

fabiogaluppo@acm.org



First year awarded:
2002

Number of MVP Awards:
11

Technical Expertise:
Visual C++

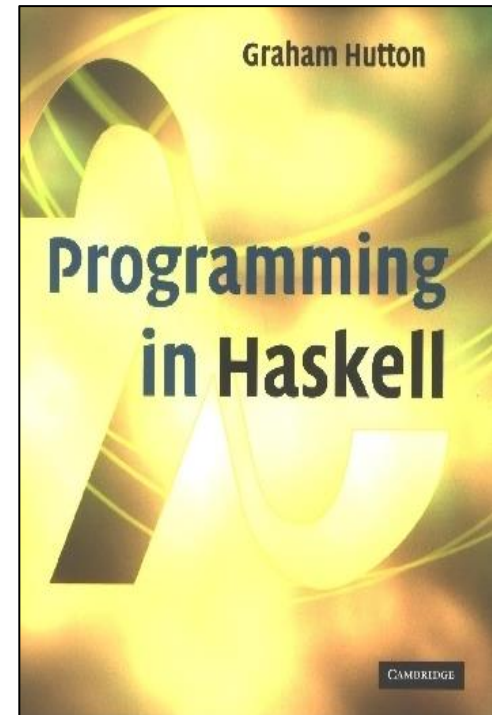
Technical Interests:
Visual C#, Visual F#

Programação Funcional

What is a Functional Language?

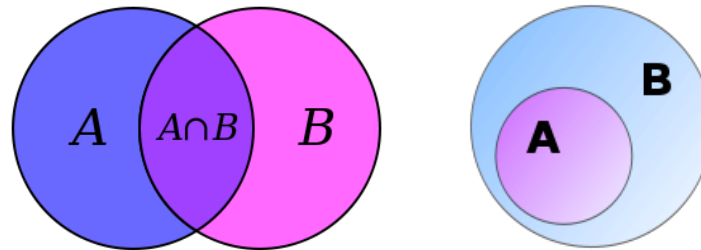
Opinions differ, and it is difficult to give a precise definition, but generally speaking:

- ⌘ Functional programming is style of programming in which the basic method of computation is the application of functions to arguments;
- ⌘ A functional language is one that supports and encourages the functional style.



Conjuntos

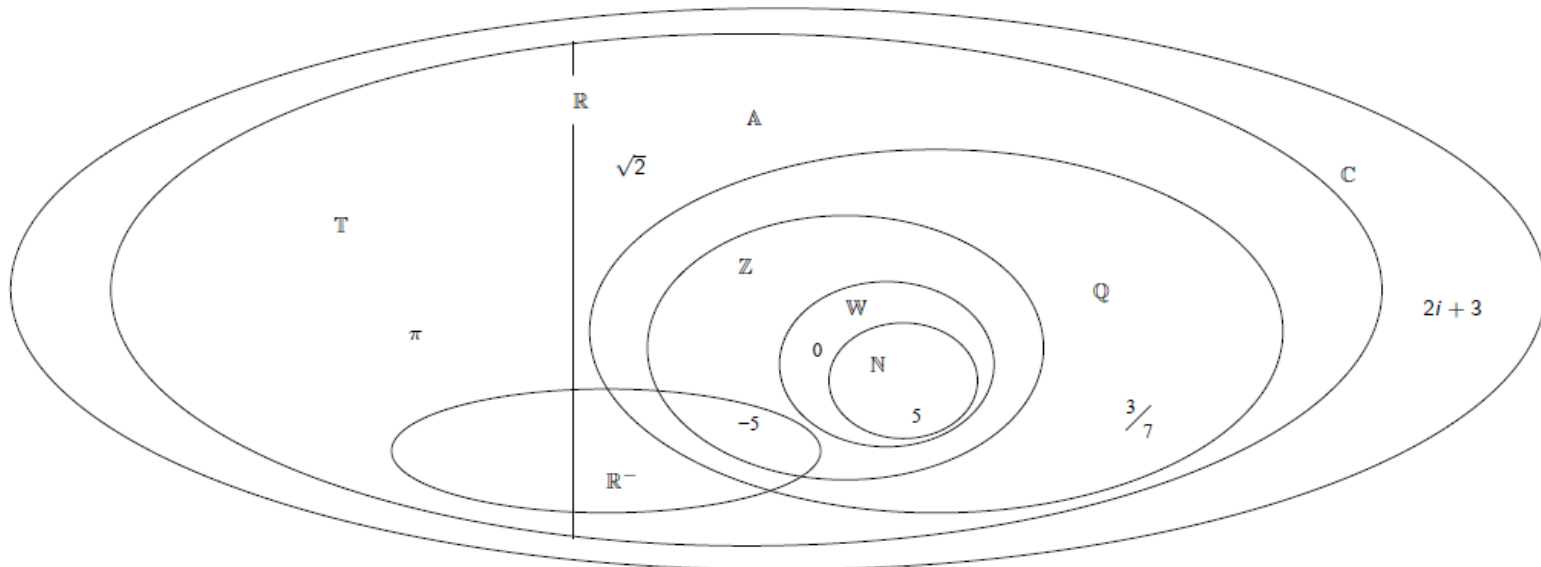
“A collection of well defined and distinct objects” [http://en.wikipedia.org/wiki/Set_\(mathematics\)](http://en.wikipedia.org/wiki/Set_(mathematics))



Things like 5 , $\frac{3}{7}$, $\sqrt{2}$, π , $2i + 3$ are elements.

Things like \mathbb{N} , \mathbb{Z} , \mathbb{Q} , \mathbb{R} , and \mathbb{C} are sets.

primitive terms
in set theory



Conjuntos

5 is a natural number (or the collection of natural numbers contains 5).

$$5 \in \mathbb{N}$$

All integers are rational numbers.

$$\mathbb{Z} \subseteq \mathbb{Q}$$

Merging the algebraic numbers and the transcendental numbers makes the real numbers.

$$\mathbb{R} = \mathbb{A} \cup \mathbb{T}$$

Negative integers are both negative and integers.

$$\mathbb{Z}^- = \mathbb{R}^- \cap \mathbb{Z}$$

Nothing is both transcendental and algebraic, or the collection of things both transcendental and algebraic is empty.

$$\mathbb{T} \cap \mathbb{A} = \emptyset$$

Adding 0 to the collection of natural numbers makes the collection of whole numbers.

$$\mathbb{W} = \{0\} \cup \mathbb{N}$$

The empty set: \emptyset or $\{\}$

Explicit listing of elements (sets are unordered, so order doesn't matter):

$$\begin{aligned} \{\text{Red, Green, Blue}\} &= \{\text{Green, Blue, Red}\} = \{\text{Blue, Red, Green}\} \\ &= \{\text{Blue, Green, Red}\} \end{aligned}$$

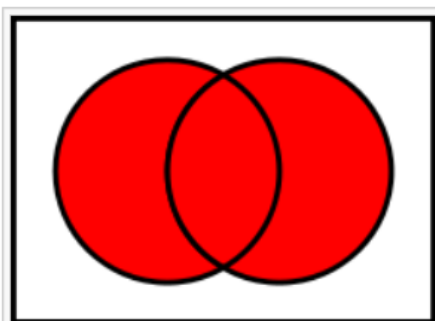
Membership of an element: $\text{Red} \in \{\text{Green, Red}\}$

Defining a set as a restriction on a larger set:

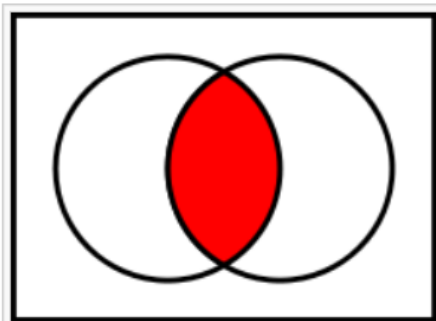
$$\mathbb{N} = \{x \in \mathbb{Z} \mid x > 0\} \leftarrow \text{a set-builder notation}$$

$$(1, 5] = \{x \in \mathbb{R} \mid 1 < x \leq 5\} \rightarrow \text{interval}$$

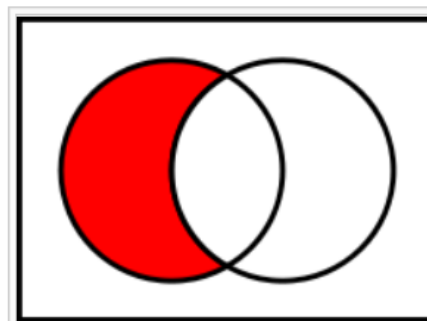
Operações entre Conjuntos



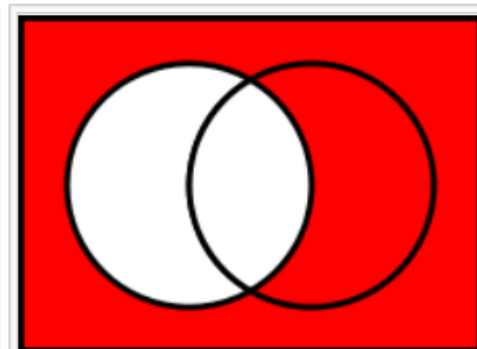
The **union** of A and B , denoted $A \cup B$



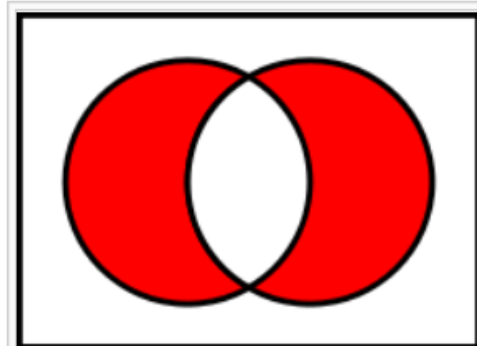
The **intersection** of A and B , denoted $A \cap B$.



The **relative complement** of B in A



The **complement** of A in U



The **symmetric difference** of A and B

$$A \Delta B = (A \setminus B) \cup (B \setminus A)$$

symmetric difference of $\{7,8,9,10\}$ and $\{9,10,11,12\}$ is the set $\{7,8,11,12\}$

$$\{1, 2\} \cup \{1, 2\} = \{1, 2\}.$$

$$\{1, 2\} \cup \{2, 3\} = \{1, 2, 3\}.$$

$$\{1, 2, 3\} \cup \{3, 4, 5\} = \{1, 2, 3, 4, 5\}$$

$$\{1, 2\} \cap \{1, 2\} = \{1, 2\}.$$

$$\{1, 2\} \cap \{2, 3\} = \{2\}.$$

$$\{1, 2\} \setminus \{1, 2\} = \emptyset.$$

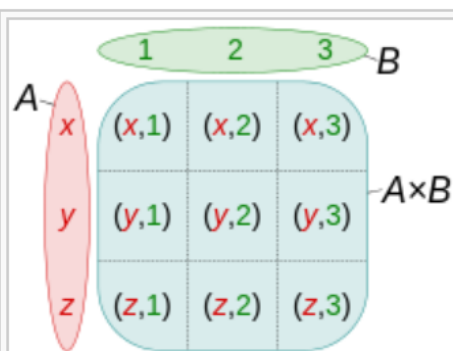
$$\{1, 2, 3, 4\} \setminus \{1, 3\} = \{2, 4\}.$$

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

$$A = \{1,2\}; B = \{3,4\}$$

$$A \times B = \{1,2\} \times \{3,4\} = \{(1,3), (1,4), (2,3), (2,4)\}$$

$$B \times A = \{3,4\} \times \{1,2\} = \{(3,1), (3,2), (4,1), (4,2)\}$$



Cartesian product $A \times B$ of the sets $A = \{x, y, z\}$ and $B = \{1, 2, 3\}$

Dualidade

$$E: (U \cap A) \cup (B \cap A) = A$$

$$E^*: (\emptyset \cup A) \cap (B \cup A) = A$$

Relações

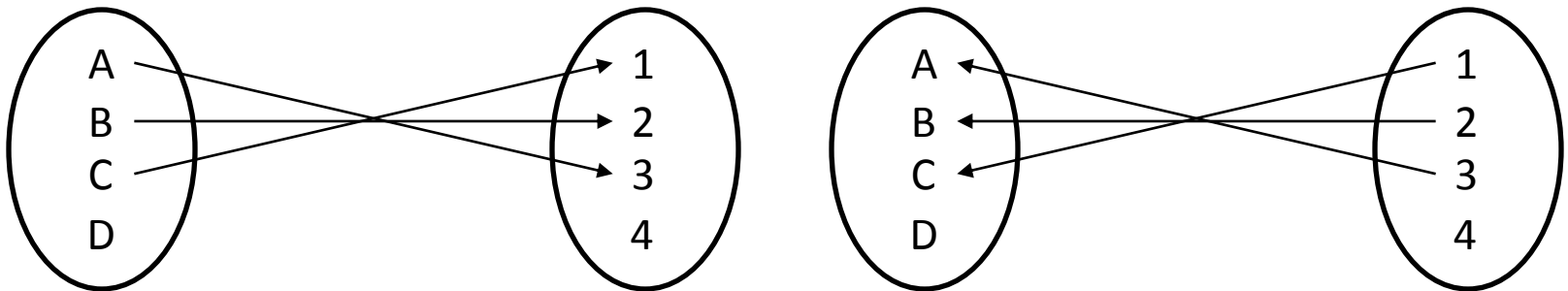
- Relação Binária de A para B é um subconjunto de $A \times B$

$A = \{\text{Mark Twain, Lewis Carroll, Charles Dickens, Stephen King}\}$

$B = \{\text{A Christmas Carol, Alice's Adventures in Wonderland, The Adventures of Tom Sawyer, The Left Hand of Darkness}\}$

$R = \{(\text{Mark Twain, The Adventures of Tom Sawyer}),$
 $(\text{Lewis Carroll, Alice's Adventures in Wonderland}),$
 $(\text{Charles Dickens, A Christmas Carol})\}$

$R^{-1} = \{(\text{The Adventures of Tom Sawyer, Mark Twain}),$
 $(\text{Alice's Adventures in Wonderland, Lewis Carroll}),$
 $(\text{A Christmas Carol, Charles Dickens})\}$



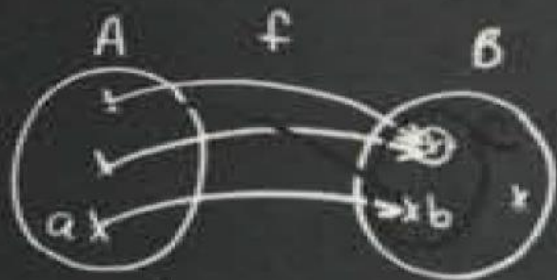
O domínio de R é $\{\text{Mark Twain, Lewis Carroll, Charles Dickens}\}$

A imagem de R é $\{\text{The Adventures of Tom Sawyer, Alice's Adventures in Wonderland, A Christmas Carol}\}$

Funções

Functions:

$f: A \rightarrow B$ means that f is a rule which assigns to each $a \in A$ an element $b \in B$



$A = \text{domain of } f$

$B = \text{range of } f$

$C = \text{Image of } f$
 $= f(A)$

$a \xrightarrow{f} b$
 $f(a) = b$

Onto Functions

Range = Image

Example:

$\rightarrow A = \{1, 2, 3\}$

$f(a) = 4a, a \in A$

$f(1) = 4, f(2) = 8$

$f(3) = 12$

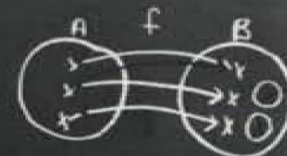
$B = \{4, 8, 12\}$

$f: A \rightarrow B$



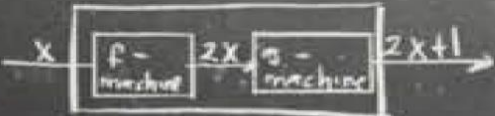
One-to-One Functions

$f(a_1) = f(a_2) \rightarrow a_1 = a_2$




Composição de Funções

Composition of functions



$q = g \circ f \rightarrow q(x) = g(f(x))$
 $f(x) = 2x, g(x) = x+1$
 $q(x) = 2x+1$
 $\text{dom } q = \text{dom } f$
 $a+b = b+a$
 $a \div b \neq b \div a$

$p = f \circ g$

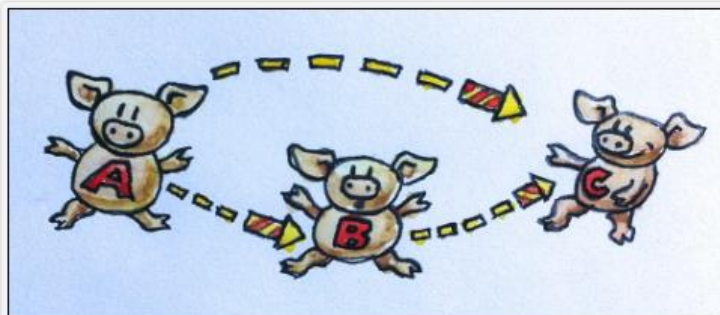


$p(x) = 2(x+1) = 2x+2$
 $q(x) = 2x+1$

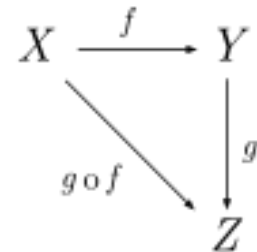
$$\begin{cases} f(x)g(x) = 2x(x+1) \\ f \circ g(x) = 2x+2 \\ g \circ f(x) = 2x+1 \end{cases}$$

<http://ocw.mit.edu/resources/res-18-006-calculus-revisited-single-variable-calculus-fall-2010/part-i-sets-functions-and-limits/lecture-2-functions/>

$$\frac{f: a \rightarrow b \quad g: b \rightarrow c}{g \circ f: a \rightarrow c}$$



In a category, if there is an arrow going from A to B and an arrow going from B to C then there must also be a direct arrow from A to C that is their composition. This diagram is not a full category because it's missing identity morphisms (see later).



<http://bartoszmilewski.com/2014/11/04/category-the-essence-of-composition/>

Funções Inversas

$$\log_2 8 = 3 \quad 2^3 = 8$$

$$(f \circ f^{-1})(x) = x$$

$$(f^{-1} \circ f)(y) = y$$

Inverse Functions
"Switch in Emphasis"

$$5 - 3 = 2 \quad 2 + 3 = 5$$

$$3 + \frac{(5-3)}{2} = 5$$

$$y = \log_b x ; b^y = x$$

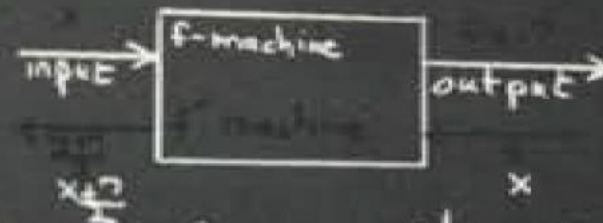
$$y = \sin^{-1} x ; x = \sin y$$

$$y = f(x) ; x = f^{-1}(y)$$

Example

$$\rightarrow y = 2x - 7 ; y = f(x) = 2x - 7$$

$$x = \frac{y+7}{2} ; x = f^{-1}(y) = \frac{y+7}{2}$$



$$f^{-1} \circ f = \text{Id}_{\text{Int}} \quad c \xrightarrow{f} 2c-7 \xrightarrow{f^{-1}} \frac{(2c-7)+7}{2} = c$$

$$f^{-1}(f(c)) = c$$

$$f(f^{-1}(d)) = d \quad d \xrightarrow{f^{-1}} \frac{d+7}{2} \xrightarrow{f} 2\left(\frac{d+7}{2}\right) - 7 = d$$

Eager versus Lazy

```
def something() = {  
  println("calling something")  
  1 // return value  
}
```

Now we are going to define two function that accept `Int` arguments that are exactly the same except that one takes the argument in a call-by-value style (`x: Int`) and the other in a call-by-name style (`x: => Int`).

```
def callByValue(x: Int) = {  
  println("x1=" + x)  
  println("x2=" + x)  
}  
  
def callByName(x: => Int) = {  
  println("x1=" + x)  
  println("x2=" + x)  
}
```

Now what happens when we call them with our side-effecting function?

```
scala> callByValue(something())  
calling something  
x1=1  
x2=1  
  
scala> callByName(something())  
calling something  
x1=1  
calling something  
x2=1
```

So you can see that in the call-by-value version, the side-effect of the passed-in function call (`something()`) only happened once. However, in the call-by-name version, the side-effect happened twice.

This is because call-by-value functions compute the passed-in expression's value before calling the function, thus the *same* value is accessed every time. However, call-by-name functions *recompute* the passed-in expression's value every time it is accessed.

Referential transparency

Referential transparency and **referential opacity** are properties of parts of [computer programs](#). An [expression](#) is said to be referentially transparent if it can be replaced with its [value](#) without changing the behavior of a program (in other words, yielding a program that has the same effects and output on the same input). The opposite term is referential opaqueness.

While in [mathematics](#) all function applications are referentially [transparent](#), in programming this is not always the case. The importance of referential transparency is that it allows the [programmer](#) and the [compiler](#) to reason about program behavior. This can help in proving [correctness](#), simplifying an [algorithm](#), assisting in modifying code without breaking it, or [optimizing](#) code by means of [memoization](#), [common subexpression elimination](#), [lazy evaluation](#), or [parallelization](#).

Referential transparency is one of the principles of [functional programming](#); only referentially transparent functions can be memoized (transformed into equivalent functions which cache results). Some [programming languages](#) provide means to guarantee referential transparency. Some functional programming languages enforce referential transparency for all functions.

As [referential transparency requires the same results for a given set of inputs at any point in time](#), a referentially transparent expression is therefore [deterministic](#).

head, tail, last, and init

head takes a list and returns its head. The head of a list is basically its first element.

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

tail takes a list and returns its tail. In other words, it chops off a list's head.

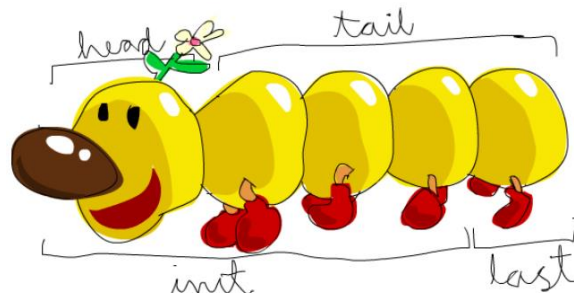
```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

last takes a list and returns its last element.

```
ghci> last [5,4,3,2,1]
1
```

init takes a list and returns everything except its last element.

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



High Order Functions

+ curried functions

Possui uma ou mais funções como entrada, ou uma função como saída, ou ambos.

Higher order functions

Haskell functions can take functions as parameters and return functions as return values. A function that does either of those is called a higher order function. Higher order functions aren't just a part of the Haskell experience, they pretty much are the Haskell experience. It turns out that if you want to define computations by defining what stuff *is* instead of defining steps that change some state and maybe looping them, higher order functions are indispensable. They're a really powerful way of solving problems and thinking about programs.



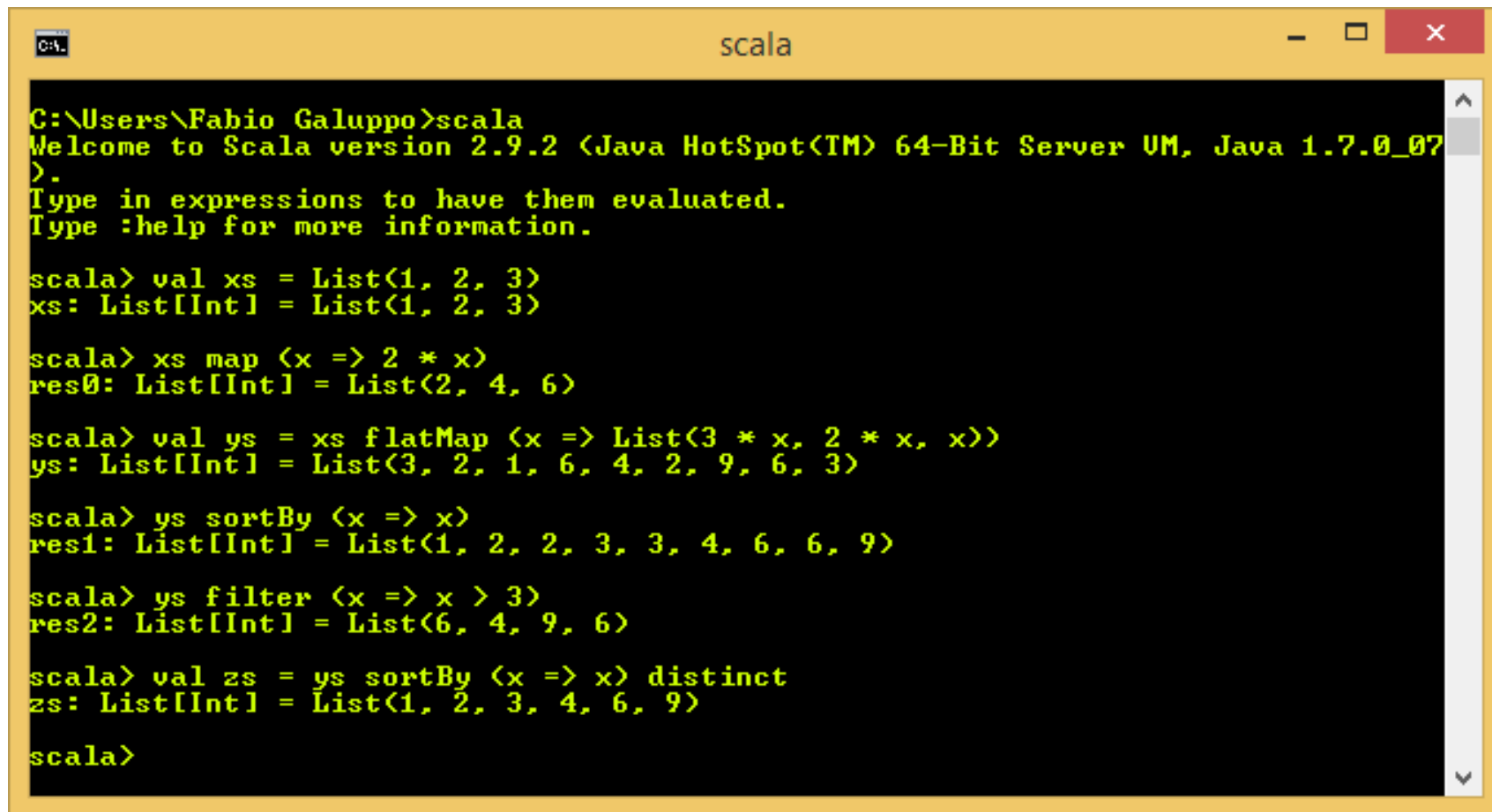
Curried functions

Every function in Haskell officially only takes one parameter. So how is it possible that we defined and used several functions that take more than one parameter so far? Well, it's a clever trick! All the functions that accepted *several parameters* so far have been **curried functions**. What does that mean? You'll understand it best on an example. Let's take our good friend, the `max` function. It looks like it takes two parameters and returns the one that's bigger. Doing `max 4 5` first creates a function that takes a parameter and returns either `4` or that parameter, depending on which is bigger. Then, `5` is applied to that function and that function produces our desired result. That sounds like a mouthful but it's actually a really cool concept. The following two calls are equivalent:

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

<http://learnyouahaskell.com/higher-order-functions>

map, flatMap, filter, sortBy, and distinct



```
C:\Users\Fabio Galuppo>scala
Welcome to Scala version 2.9.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_07).
Type in expressions to have them evaluated.
Type :help for more information.

scala> val xs = List(1, 2, 3)
xs: List[Int] = List(1, 2, 3)

scala> xs map (x => 2 * x)
res0: List[Int] = List(2, 4, 6)

scala> val ys = xs flatMap (x => List(3 * x, 2 * x, x))
ys: List[Int] = List(3, 2, 1, 6, 4, 2, 9, 6, 3)

scala> ys sortBy (x => x)
res1: List[Int] = List(1, 2, 2, 3, 3, 4, 6, 6, 9)

scala> ys filter (x => x > 3)
res2: List[Int] = List(6, 4, 9, 6)

scala> val zs = ys sortBy (x => x) distinct
zs: List[Int] = List(1, 2, 3, 4, 6, 9)

scala>
```

LINQ Where

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate) {
            if (source == null) throw Error.ArgumentNull("source");
            if (predicate == null) throw Error.ArgumentNull("predicate");
            if (source is Iterator<TSource>) return ((Iterator<TSource>)source).Where(predicate);
            if (source is TSource[]) return new WhereArrayIterator<TSource>((TSource[])source, predicate);
            if (source is List<TSource>) return new WhereListIterator<TSource>((List<TSource>)source, predicate);
            return new WhereEnumerableIterator<TSource>(source, predicate);
        }

        public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source, Func<TSource, int, bool> predicate) {
            if (source == null) throw Error.ArgumentNull("source");
            if (predicate == null) throw Error.ArgumentNull("predicate");
            return WhereIterator<TSource>(source, predicate);
        }

        static IEnumerable<TSource> WhereIterator<TSource>(IEnumerable<TSource> source, Func<TSource, int, bool> predicate) {
            int index = -1;
            foreach (TSource element in source) {
                checked { index++; }
                if (predicate(element, index)) yield return element;
            }
        }
    }
}
```


LINQ Where

```
{
    IEnumerable<TSource> source;
    Func<TSource, bool> predicate;
    IEnumerator<TSource> enumerator;

    public WhereEnumerableIterator(IEnumerable<TSource> source, Func<TSource, bool> predicate) {
        this.source = source;
        this.predicate = predicate;
    }

    public override Iterator<TSource> Clone() {
        return new WhereEnumerableIterator<TSource>(source, predicate);
    }

    public override void Dispose() {
        if (enumerator is IDisposable) ((IDisposable)enumerator).Dispose();
        enumerator = null;
        base.Dispose();
    }

    public override bool MoveNext() {
        switch (state) {
            case 1:
                enumerator = source.GetEnumerator();
                state = 2;
                goto case 2;
            case 2:
                while (enumerator.MoveNext()) {
                    TSource item = enumerator.Current;
                    if (predicate(item)) {
                        current = item;
                        return true;
                    }
                }
                Dispose();
                break;
        }
        return false;
    }
}
```

LINQ Where

```
abstract class Iterator<TSource> : IEnumerable<TSource>, IEnumerator<TSource>
{
    int threadId;
    internal int state;
    internal TSource current;

    public Iterator() {
        threadId = Thread.CurrentThread.ManagedThreadId;
    }

    public TSource Current {
        get { return current; }
    }

    public abstract Iterator<TSource> Clone();

    public virtual void Dispose() {
        current = default(TSource);
        state = -1;
    }

    public IEnumerator<TSource> GetEnumerator() {
        if (threadId == Thread.CurrentThread.ManagedThreadId && state == 0) {
            state = 1;
            return this;
        }
        Iterator<TSource> duplicate = Clone();
        duplicate.state = 1;
        return duplicate;
    }

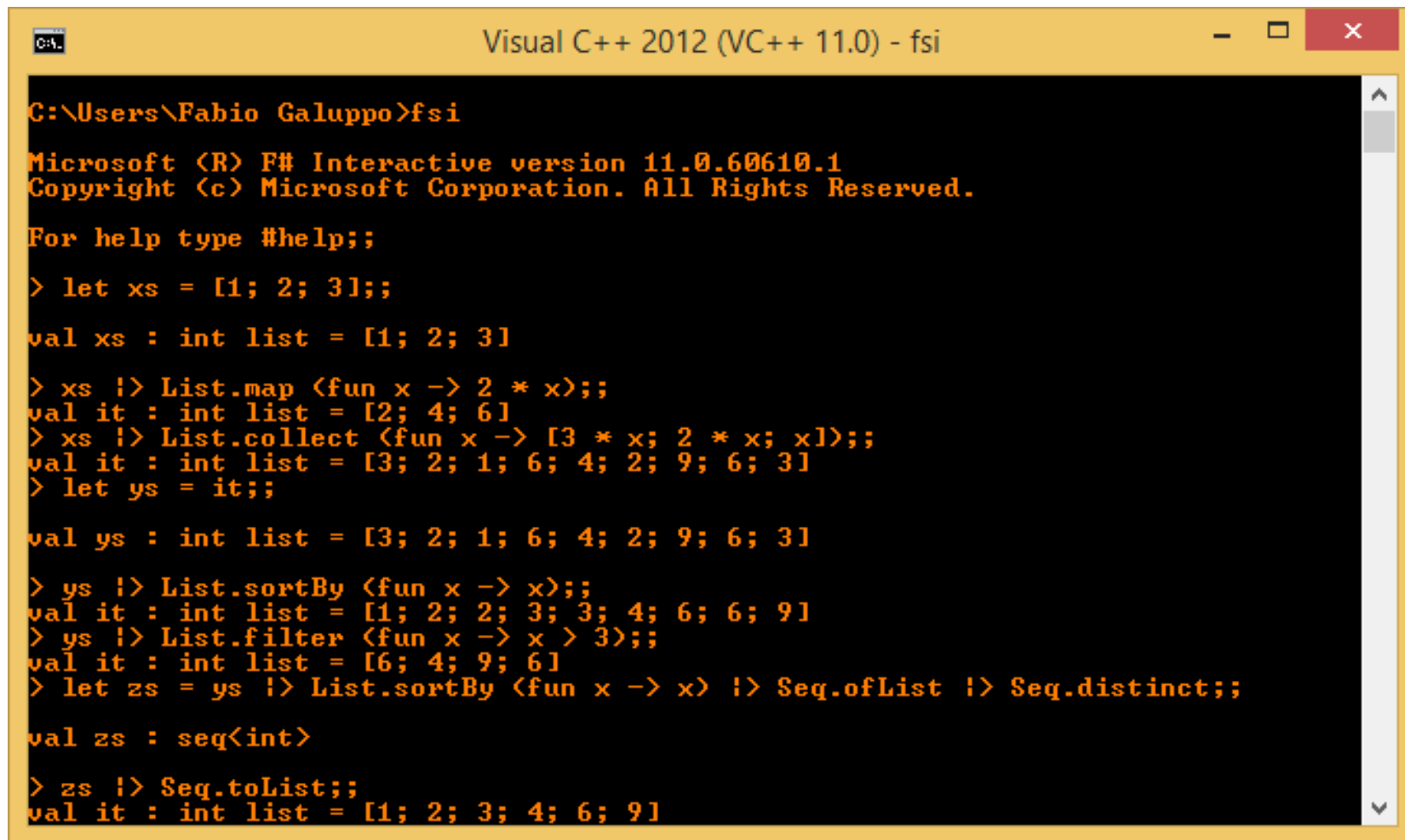
    public abstract bool MoveNext();

    public abstract IEnumerable<TResult> Select<TResult>(Func<TSource, TResult> selector);

    public abstract IEnumerable<TSource> Where(Func<TSource, bool> predicate);

    object IEnumerator.Current {
        get { return Current; }
    }
}
```

map, collect, filter, sortBy, and distinct

A screenshot of a Visual C++ 2012 (VC++ 11.0) console window titled "Visual C++ 2012 (VC++ 11.0) - fsi". The window has a yellow title bar and a black background with orange text. The console shows the following F# code and its output:

```
C:\Users\Fabio Galuppo>fsi
Microsoft (R) F# Interactive version 11.0.60610.1
Copyright (c) Microsoft Corporation. All Rights Reserved.
For help type #help;;
> let xs = [1; 2; 3];;
val xs : int list = [1; 2; 3]
> xs !> List.map <fun x -> 2 * x>;;
val it : int list = [2; 4; 6]
> xs !> List.collect <fun x -> [3 * x; 2 * x; x]>;;
val it : int list = [3; 2; 1; 6; 4; 2; 9; 6; 3]
> let ys = it;;
val ys : int list = [3; 2; 1; 6; 4; 2; 9; 6; 3]
> ys !> List.sortBy <fun x -> x>;;
val it : int list = [1; 2; 2; 3; 3; 4; 6; 6; 9]
> ys !> List.filter <fun x -> x > 3>;;
val it : int list = [6; 4; 9; 6]
> let zs = ys !> List.sortBy <fun x -> x> !> Seq.ofList !> Seq.distinct;;
val zs : seq<int>
> zs !> Seq.toList;;
val it : int list = [1; 2; 3; 4; 6; 9]
```

C#: Select, SelectMany, Where, OrderBy, and Distinct

Algebraic Data type

- Product types

- Tuples
- Records

("F# Rocks!", 123, 456.789) (*true, false*)

<https://msdn.microsoft.com/en-us/library/vstudio/dd233200.aspx>

```
F#
type Point = { x : float; y : float; z : float; }
type Customer = { First : string; Last : string; SSN : uint32; AccountNumber : uint32; }
```

<https://msdn.microsoft.com/en-us/library/vstudio/dd233184.aspx>

```
F#
let mypoint = { x = 1.0; y = 1.0; z = -1.0; }
```

- Sum types

- Also called, tagged unions or variant types

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

Discriminated union

F#

```
type Shape =
| Rectangle of width : float * length : float
| Circle of radius : float
| Prism of width : float * float * height : float
```

[https://msdn.microsoft.com/en-us/library/vstudio/dd233226\(v=vs.120\).aspx](https://msdn.microsoft.com/en-us/library/vstudio/dd233226(v=vs.120).aspx)

F#

```
let rect = Rectangle(length = 1.3, width = 10.0)
let circ = Circle (1.0)
let prism = Prism(5., 2.0, height = 3.0)
```

In C and C++, a tagged union can be created from untagged unions using a strict access discipline where the tag is always checked:

```
enum ShapeKind { Square, Rectangle, Circle };

struct Shape {
    int centerX;
    int centerY;
    enum ShapeKind kind;
    union {
        struct { int side; }; /* Square */
        struct { int length, height; }; /* Rectangle */
        struct { int radius; }; /* Circle */
    };
};

int getSquareSide(struct Shape* s) {
    assert(s->kind == Square);
    return s->side;
}

void setSquareSide(struct Shape* s, int side) {
    s->kind = Square;
    s->side = side;
}
```

http://en.wikipedia.org/wiki/Tagged_union

http://en.wikipedia.org/wiki/Algebraic_data_type

Pattern Matching

Pattern matching

This chapter will cover some of Haskell's cool syntactic constructs and we'll start with pattern matching. **Pattern matching consists of specifying patterns to which some data should conform and then checking to see if it does and deconstructing the data according to those patterns.**

When defining functions, you can define separate function bodies for different patterns. This leads to really neat code that's simple and readable. You can pattern match on any data type — numbers, characters, lists, tuples, etc. Let's make a really trivial function that checks if the number we supplied to it is a seven or not.

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

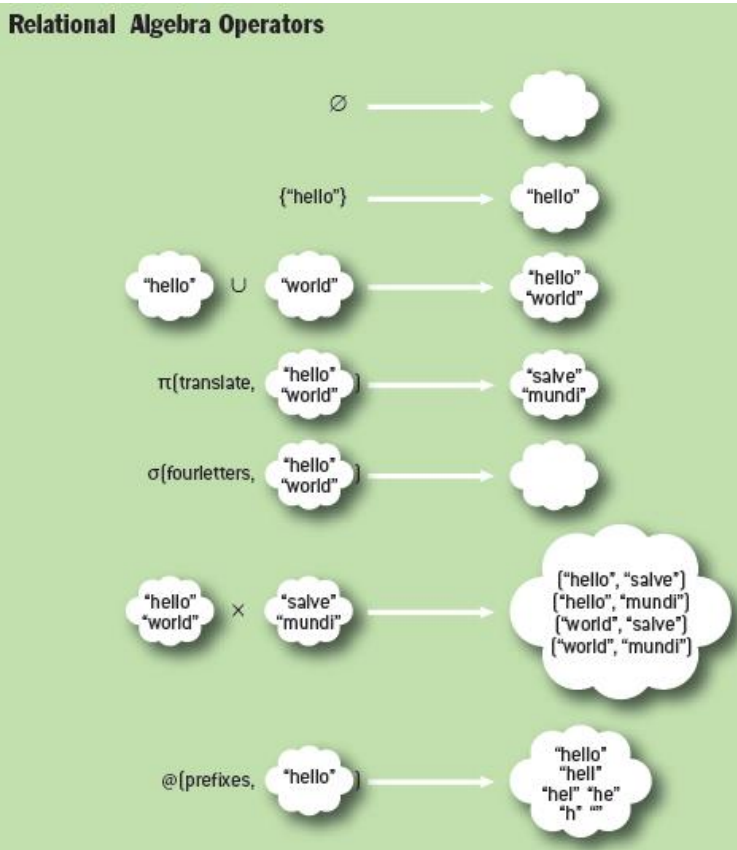
```
length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:xs) = 1 + length' xs
```

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y
tell (x:y:_) = "This list is long. The first two elements are: " ++ show x ++ " and " ++ show y
```



Relational Algebra

$\pi(x \Rightarrow x.NameOrNick, \sigma(x \Rightarrow Plays(x, VideoGames), Friends))$



The cross-apply operator @ is particularly powerful since it allows for correlated subqueries where you generate a second collection for each value from a first collection and flatten the results into a single collection $@(f, \{a, \dots, z\}) = f(a) \cup \dots \cup f(z)$. All other relational operators can be defined in terms of the cross-apply operator:

$xs \times ys = @(x \Rightarrow \pi(y \Rightarrow (x, y), ys), xs)$

$\pi(f, xs) = @(x \Rightarrow \{f(x)\}, xs)$

$\sigma(p, xs) = @(x \Rightarrow p(x) ? \{x\} : \emptyset, xs)$

projection
selection

```
//projection π
IEnumerable<T> Select<S,T>(IEnumerable<S> source,
                          Func<S,T> selector)

//CROSS-APPLY @
IEnumerable<T> SelectMany<S,T>(IEnumerable<S> source,
                              Func<S, IEnumerable<T>> selector)

//selection σ
IEnumerable<T> Where<T>(IEnumerable<T> source,
                      Func<T,bool> predicate)
```

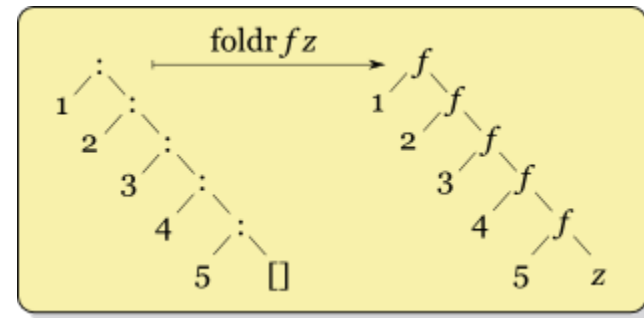
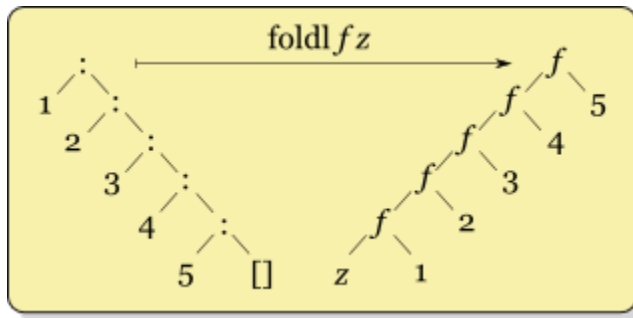
selection in "plain English":

`selection(p: predicate, xs: set(xs)) = cross-apply(x -> if p(x) then set(x) else emptySet, xs)`

The World According to LINQ

<http://queue.acm.org/detail.cfm?id=2024658>

foldl, and foldr



$xs = [1,2,3,4,5]$

$foldLeft\ acc\ xs \Rightarrow f\ (f\ (f\ (f\ (f\ acc\ 1)2)3)4)5$

$foldRight\ xs\ acc \Rightarrow f\ 1\ (f\ 2\ (f\ 3\ (f\ 4\ (f\ 5\ acc))))$

	Left Fold	Right Fold
C#	<code>ienum.Aggregate(acc, f)</code>	<code>ienum.Reverse().Aggregate(acc, f)</code>
C++	<code>std::accumulate(begin, end, acc, f)</code>	<code>std::accumulate(rbegin, rend, acc, f)</code>
F#	<code>List.fold f acc list</code>	<code>List.foldBack f list acc</code>
Haskell	<code>foldl f acc list</code>	<code>foldr f acc list</code>
Java	<code>stream.reduce(acc, f)</code>	☹
Scala	<code>list.foldLeft(acc)(f)</code>	<code>list.foldRight(acc)(f)</code>

Lifting

fmap :: Functor f => (a -> b) -> f a -> f b

fmap :: Functor f => (a -> b) -> (f a -> f b)

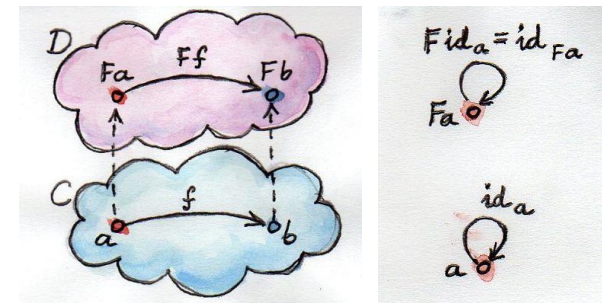
Before we go on to the rules that `fmap` should follow, let's think about the type of `fmap` once more. Its type is `fmap :: (a -> b) -> f a -> f b`. We're missing the class constraint `(Functor f) =>`, but we left it out here for brevity, because we're talking about functors anyway so we know what the `f` stands for. When we first learned about [curried functions](#), we said that all Haskell functions actually take one parameter. A function `a -> b -> c` actually takes just one parameter of type `a` and then returns a function `b -> c`, which takes one parameter and returns a `c`. That's how if we call a function with too few parameters (i.e. partially apply it), we get back a function that takes the number of parameters that we left out (if we're thinking about functions as taking several parameters again). So `a -> b -> c` can be written as `a -> (b -> c)`, to make the currying more apparent.

In the same vein, if we write `fmap :: (a -> b) -> (f a -> f b)`, we can think of `fmap` not as a function that takes one function and a functor and returns a functor, but as a function that takes a function and returns a new function that's just like the old one, only it takes a functor as a parameter and returns a functor as the result. It takes an `a -> b` function and returns a function `f a -> f b`. This is called *lifting* a function. Let's play around with that idea by using GHCi's `:t` command:

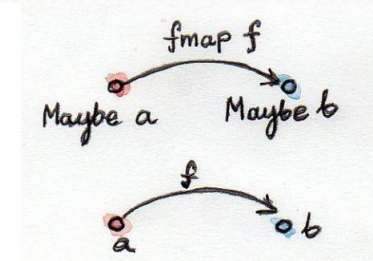
```
ghci> :t fmap (*2)
fmap (*2) :: (Num a, Functor f) => f a -> f a
ghci> :t fmap (replicate 3)
fmap (replicate 3) :: (Functor f) => f a -> f [a]
```



```
ghci> fmap (replicate 3) [1,2,3,4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]
ghci> fmap (replicate 3) (Just 4)
Just [4,4,4]
```



fmap :: (a -> b) -> (Maybe a -> Maybe b)



fmap :: (a -> b) -> Maybe a -> Maybe b

```
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)
```