Monoids

[mono = one
oid = oid]

- Monoids are an important pattern, and they crop up everywhere!
- Identifying monoids and putting them to use allows us to structure and abstract and to write better programs.
- Monoids are a special subset of things which can be combined (Microsoft: ICombinable or IAppendable)
- Mathemathically monoids are algebraic structures with a single binary operation ("combine") which satisfies the following properties:
  1. closure: the result is always of the same type as the inputs
  2. associative: order doesn't matter $(a+b)+c = a+(b+c)$
  3. identity: there exists a neutral element

(if 3. is false, the structure is a semigroup)

Examples:

(int, +, 0) (real): int + int = int

(1 + 2) + 3 = 1 + (2 + 3)

1 + 0 = 1

(int, *, 1) (real): int * int = int

(2 * 3) * 4 = 2 * (3 * 4)

1 * 2 = 2

(int, /, 1): int / int ≠ int !

(real, /, 1): (2/3)/4 ≠ 2/(3/4) !

(string, ++, "") : string ++ string = string

(str ++ str) ++ str = str ++ (str ++ str)

str ++ "" = str

(list, @, []) : list @ list = list

(list @ list) @ list = list @ (list @ list)

list @ [] = list

(option, <>, None): opt <> opt = opt

opt <> (opt <> opt) = (opt <> opt) <> opt

opt <> None = opt

Usefulness of monoids:

1. Closure: the binary operator can be extended to lists:

```
List.reduce (+) [1..10]
List.reduce (@) [[1..5]; [6...10]]
```

[1..5] @ [6...10] @ [ ] @ [ ] ...

2 Associativity: divide and conquer
incremental computations
parallelism

(beware commutation: $a \cdot b = b \cdot a$
vs: $a \cdot b \neq b \cdot a$)

3. Identity:

```
[] |> List.reduce (+)
vs. [] |> List.fold (+) 0
```

Option monoid:

```
let (<>) a b =
    match (a,b) with
    |(Some x, Some y) -> Some (x<>y)
    |(None, x) -> x
    |(x, None) -> x

let zero = None
```

Case study:

```
type Order = {
    Code : string
    Qty : int
    Price : float
}

let addOrders a b =                    ↙?!
    { Code = "total"; Qty = a.Qty + b.Qty;
      Price = a.Price + b.Price }

let zero = { Code = ""; Qty = 0; Price = 0.0 }
```

But this is kind of ugly and hacky!

Let's put a free monoid to use:

```
type Order = {
    Code : string list
    Qty : int list
    Price : float
}
```

now every member is properly monoidal in the context!

An even better option is to use an union:

```
type Product = { Code : string; Qty : int; Price : float }

type Order =
    | Product of Product
    | Total of float
    | Empty
```

Now we can easily define a monoid!

If we add the (+) operator and Zero as static members to the type, we can directly use functions like List.sum!:

```
type Order =
    ...
    static member (+) (x,y) = ...
    static member Zero = Empty

List.sum [o1; o2; o3]
```

Monoid homomorphisms

- Isomorphisms preserve both structure and information
- Homomorphisms only presere structure
- A monoid homomorphism transforms a monoid to a new, possibly very different monoid, while preserving "monoidness"

Example:

```
type Page = Page of string
let addPage (Page a) (Page b) = Page (a++b)
let wordCount str = ...
let count = Page.map wordCount (p1 <> p2 <> .. pn)
```

or

```
let counts = List.map wordCount [p1; p2; ...; pn]
let count = List.reduce (+) counts
```

First we convert (map) from one monoid (string) to a new monoid (int), and then we reduce to get the same result!

This is the essence of Googles MapReduce!

A word of warning: Not all (monoid) homomorphisms are created equal:

```
let mostFreqWord str = ...
let word = Page.map mostFreqWord (p1 <> p2 <> p3)
≠ List.reduce (++) (List.map mostFreqWord [p1;p2;p3]
```

The Law of monoid homomorphisms:

```
f a <> f b = f (a <> b)
```

Final thought:

We have already encountered a monoid without realizing it: the monad!

1. They have closure
2. join is associative
3. return is identity

A functor is a transform from one category to another. An endofunctor is to the same category.

"A monad is a monoid in the category of endofunctors". (Philip Wadler)

F# Type Providers

- Type Providers are a mechanism in F# for dynamically generating types at compile time from data sources.
- Examples are (SQL) databases, CSV, XML, JSON, OData, R ...
- Type Providers work by using reflexion and code quotations at compiletime
- Type providers are easy, not simple! They hide a lot of complexity, which may come back and bite you.
- For relational databases I recommend using Entity Framework Core instead.
- You can create your own TP:s using the TP SDK. It's an "interesting" exercise into the depths of the compiler and the IL / CLR.