# Functional Programming Concepts in Imperative Languages

Dave van Soest

April 14, 2014

Introduction

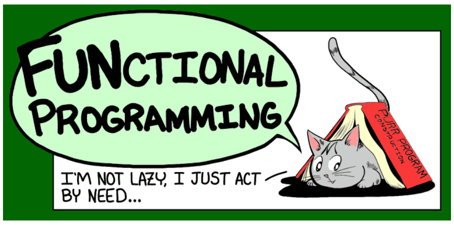
Functions

Collection operations

Tips & tricks

Questions & Discussion

## Introduction



## Goals

- Re-usable code patterns.
- Less code.
- Better readable.
- Higher performance.

## What is functional programming?

Functional programming is a declarative programming paradigm based on pure (i.e. side-effect-free) functions, which are composited and chained to create more complex functions.

Functional programming...

- can make program behavior easier to understand, more predictable and easier to prove correct;
- avoids program state and mutable data;
- reduces code size; (when using it in an applicable domain);
- doesn't tell a computer what to do, but rather how information can be computed from previous information;
- encourages modularity;
- is inherently optimized for parallel processing.

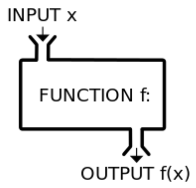
## Functional programming: Example

```
start(N) -> do_fib(0, 1, N).
do_fib(_, B, 1) -> B;
do_fib(A, B, N) -> do_fib(B, A + B, N - 1).
```

## Imperative languages

Imperative programming tells a computer what to do by using commands/statements that change program state.

## Functions



## Functors a.k.a. First-class Functions

A functor is a function object instance that can be referenced by a variable like a regular object, and which can be evaluated by invoking a method on the object.

- Almost all imperative programming languages support the concept of functor.
- There are big differences between languages w.r.t. functors:
  - Restrictions that apply to functors;
  - Support from the language itself;
  - Support from the standard libraries.

## Using functors to express functional concepts

- Consider using pure functions:
  - Side-effect-free.
  - Context independent.
- Next best thing: Side-effect-free functions, but context dependent.
- Never use functors with side-effects without a very good reason.
  - If you decide to, then try to keep the side-effects as local/close-by as possible.

## Collection operations

## Collection operations

- Lists/arrays.
- Hash tables/hash maps.
- Objects.

# Map a.k.a. transform a.k.a. ...

Transform each item in a collection.

```
var list = [3, 1, 2, 4];
var squared = _.map(list, function(n) {
    return n * n;
});
// squared === [9, 1, 4, 16]
```

# Map (ctd)

```
var keys = [’uno’, ’due’, ’tre’];
var iterator = function(k) {
    var o = {};
    o[k] = 1;
    return o;
};
var objs = _.map(keys, iterator);
// objs === [{uno: 1}, {due: 1}, {tre: 1}]
```

# Filter a.k.a. select a.k.a. ...

Select elements from a collection.

```
var list = [1, 2, 3, 4, 5, 6];
var isEven = function(n) {
    return n % 2 === 0;
};
var evenList = _.filter(list, isEven);
```

# Filter (ctd)

```
var list = [1, 2, 3, 4, 5, 6];
var isIndexEven = function(n, i) {
    return i % 2 === 0;
};
var output = _.filter(list, isIndexEven);
```

## Reduce a.k.a. fold a.k.a. aggregate a.k.a. ...

Aggregate data into a single output.

```
var list = [1, 2, 3, 4, 5];
var sum = function(imr, n) { return imr + n; };
var summedList = _.reduce(list, sum, 0);
```

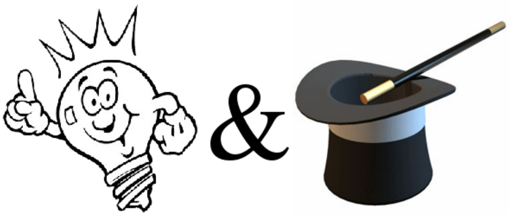
## Map-reduce

Map-reduce is a commonly used chain of functions for processing large data. It works perfectly in a distributed environment and supports data streams.

## Map-reduce (ctd)

```
var data = [
    {gender: 'm', age: 31}, {gender: 'f', age: 19},
    {gender: 'f', age: 29}, {gender: 'm', age: 37}
];

var weightedAverage = function(records, weightProp, valueProp, weights) {
    var mapped = _.map(records, function(rec) {
        return {
            weight: weights[weightProp][rec[weightProp]],
            value: rec[valueProp]
        };
    });

    var reduced = _.reduce(mapped, function(total, item) {
        return {
            weight: total.weight + item.weight,
            value: total.value + (item.value * item.weight)
        };
    }, {weight: 0, value: 0});

    return reduced.value / reduced.weight;
};

var output = weightedAverage(data, 'gender', 'age', {gender: {m: 1, f: 2}});
```

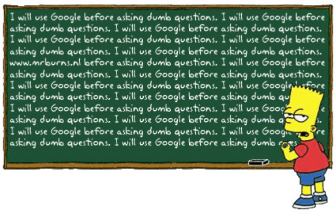
## Tips & tricks



## Tips & Tricks

- Prefer pure functions.
  - Otherwise use side-effect-free functions.
  - Otherwise don't use functional concepts, to avoid confusion.
- Use function argument binding, for quickly creating specific functions.
- Consider using function result caching/memoizing to speed-up computation.
  - Note: Only use result caching for pure functions!
- Use in-place operations if applicable to increase performance.

## Questions & Discussion


I will use Google before asking dumb questions. I will use Google before
asking dumb questions. I will use Google before asking dumb questions.
I will use Google before asking dumb questions. I will use Google before
asking dumb questions. I will use Google before asking dumb questions.
www.mrburns.nl before asking dumb questions. I will use Google before
asking dumb questions. I will use Google before asking dumb questions.
I will use Google before asking dumb questions. I will use Google
asking dumb questions. I will use Google before asking dumb qu
I will use Google before asking dumb questions. I will use Goo
asking dumb questions. I will use Google before asking dumb q
I will use Google before asking dumb questions. I will use Goog
asking dumb questions. I will use Google before asking dumb