

Functional Programming Concepts in Imperative Languages

Dave van Soest

April 14, 2014

Introduction

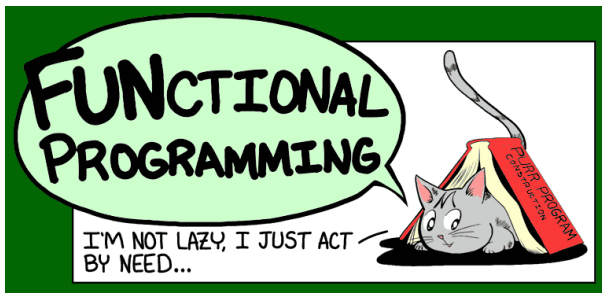
Functions

Collection operations

Tips & tricks

Questions & Discussion

Introduction



Goals

- Re-usable code patterns.

Goals

- Re-usable code patterns.
- Less code.

Goals

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

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.

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...

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;

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;

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);

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;

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;

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.

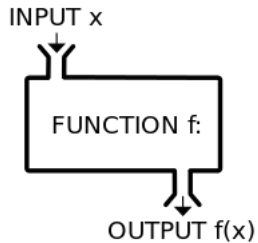
Functional programming: Example

```
1 start(N) -> do_fib(0, 1, N).  
2 do_fib(_, B, 1) -> B;  
3 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.

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.

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:

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;

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;

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**:

Using functors to express functional concepts

- Consider using pure functions:
 - Side-effect-free.

Using functors to express functional concepts

- Consider using pure functions:
 - Side-effect-free.
 - Context independent.

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.

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.

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.

Collection operations

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

Collection operations

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

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

Transform each item in a collection.

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

Transform each item in a collection.

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

Map (ctd)

```
1  var keys = ['uno', 'due', 'tre'];
2  var iterator = function(k) {
3    var o = {};
4    o[k] = 1;
5    return o;
6  };
7  var objs = _.map(keys, iterator);
8  // objs === [{uno: 1}, {due: 1}, {tre: 1}]
```

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

Select elements from a collection.

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

Select elements from a collection.

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

Filter (ctd)

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


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

Aggregate data into a single output.

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

Aggregate data into a single output.

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

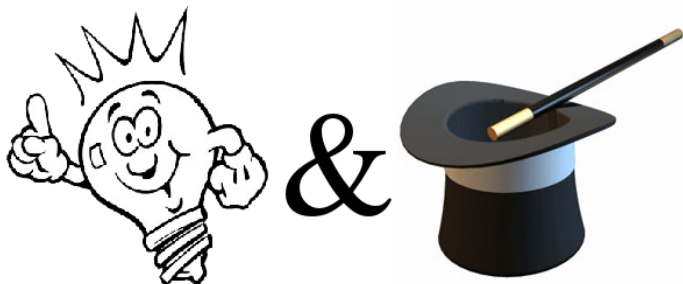
Map-reduce

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)

```
1  var data = [  
2    {gender: 'm', age: 31}, {gender: 'f', age: 19},  
3    {gender: 'f', age: 29}, {gender: 'm', age: 37}  
4  ];  
5  
6  var weightedAverage = function(records, weightProp, valueProp, weights) {  
7    var mapped = _.map(records, function(rec) {  
8      return {  
9        weight: weights[weightProp][rec[weightProp]],  
10       value: rec[valueProp]  
11     };  
12   });  
13  
14   var reduced = _.reduce(mapped, function(total, item) {  
15     return {  
16       weight: total.weight + item.weight,  
17       value: total.value + (item.value * item.weight)  
18     };  
19   }, {weight: 0, value: 0});  
20  
21   return reduced.value / reduced.weight;  
22 };  
23  
24 var output = weightedAverage(data, 'gender', 'age', {gender: {m: 1, f: 2}});
```

Tips & tricks



Tips & Tricks

- Prefer pure functions.

Tips & Tricks

- Prefer pure functions.
 - Otherwise use side-effect-free functions.

Tips & Tricks

- Prefer pure functions.
 - Otherwise use side-effect-free functions.
 - Otherwise don't use functional concepts, to avoid confusion.

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.

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.

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!

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

