Functional* UI JS

Dan Menssen | MN.js February 2015

(* Philosophically, at least)

Functional* UI JS

Dan Menssen | MN.js February 2015

OLSON

Solutions Architect

Functional* UI JS

Dan Menssen | MN.js February 2015

OLSON

Solutions Architect (?)

@menssen

dan@menssen.org

Functional programming

From Wikipedia, the free encyclopedia

For subroutine-oriented programming, see Procedural programming.

In computer science, **functional programming** is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. In the other well-known declarative programming paradigm, logic programming, relations are at the base of respective languages.[1]

Functional programming

From Wikipedia, the free encyclopedia

For subroutine-oriented programming, see Procedural programming.

In computer science, **functional programming** is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. In the other well-known declarative programming paradigm, logic programming, relations are at the base of respective languages.[1]

Functional programming

From Wikipedia, the free encyclopedia

For subroutine-oriented programming, see Procedural programming.

In computer science, **functional programming** is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. In the other well-known declarative programming paradigm, logic programming, relations are at the base of respective languages.[1]

(paradigms
procedural
OO
functional
logic)

Functional programming

From Wikipedia, the free encyclopedia

For subroutine-oriented programming, see Procedural programming.

In computer science, **functional programming** is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. In the other well-known declarative programming paradigm, logic programming, relations are at the base of respective languages.[1]

Functional programming

From Wikipedia, the free encyclopedia

For subroutine-oriented programming, see Procedural programming.

In computer science, **functional programming** is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. In the other well-known declarative programming paradigm, logic programming, relations are at the base of respective languages.[1]

(procedural)

```
var x = getUserInput()
function increment() {
    ++x
}
```

(OO)

```
class Incrementor {
  private x = 1
  function Incrementor() {
    this.x = getUserInput()
  }
  function execute() {
    ++this.x
  }
}
```

(logic)

????

$\forall x \exists y (x = a \supset y = a + 1)$

(functional)

```
function increment(x) {
  return x + 1
}

x = increment(
  getUserInput()
)

// f(x) = 3x² + 2x + 5
```

++X

this.x

(procedural)

```
var x = getUserInput()
x = x+2
x = ‘$’ + (string) x + ‘.00’
print x
```

(functional)

```
print(
  concat(
    ‘$’, add(2, getUserInput()), ‘.00’
  )
)
```

```
( procedural )
for (var i = 0; i < max; ++i) {
  doSomething(x)
}
```

```
( functional )
doSomethingForEveryElementInList(
  {1..max},
  doSomething
)
```

(why?
  testing - no mocks!
  easier to reason about - stay tuned)

```
doSomethingForEveryElementInList(
  {1..max},
  doSomething
)
```

```
doSomethingForEveryElementInList(
  {1..max},
  doSomething
)
```

```
[1, 2, 3].forEach(doSomething)
```

```
print(concat(‘$’, add(2, getUserInput())), ‘.00’))
```

```
print(concat(‘$’, add(2, getUserInput())), ‘.00’))


transformEveryElementInList(
  transformEveryElementInList(
    {1..max},
    doSomething
  ),
  doSomethingElse
)
```

```
print(concat(‘$’, add(2, getUserInput())), ‘.00’))


transformEveryElementInList(
  transformEveryElementInList(
    {1..max},
    doSomething
  ),
  doSomethingElse
)


[1, 2, 3].map(doSomething).map(doSomethingElse)
```

Three functions to transform an array into something else

Three functions to transform an array into something else

```
// transform into subset of itself

[1,2,3,4,5].filter(function(item) {
  return item > 3
})

// [4,5]
```

Three functions to transform an array into something else

```
// transform into a new list

[1,2,3,4,5].map(function(item) {
  return item + 1
})

// [2,3,4,5,6]
```

Three functions to transform an array into something else

```
// collapse

[1,2,3,4,5].reduce(function(last, item) {
  return item + lastResult
}, 0)

// 15

[ [1, 2], [3, 4], [5, 6]].reduce(function(last, item) {
  return last.concat(item)
}, [])

// [1,2,3,4,5]
```

http://github.com/menssen/jsmn-february-2015