[SOT118]
Functional programming - from LISP to JavaScript via Haskell

DJ Adams

May 31st, 2020

07:35 - 08:35 UTC

At the end of this session you'll understand why the reduce function exists in JavaScript, where it came from, and why it's so beautiful. And we'll start that journey of understanding in the 1950s with Lisp, and take a route via Haskell.

SAP Online Track
@sapOnlineTrack

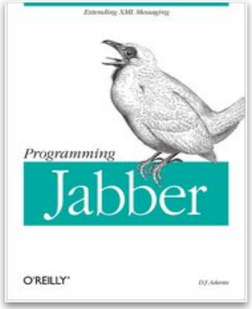
Supporting girls who code

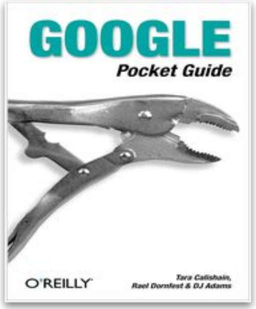
http://bit.ly/saponlinetrack-fundraiser

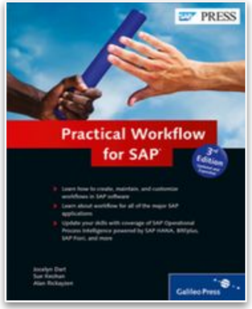
DJ Adams

Developer Advocate at SAP

Developer, Author, Speaker, Teacher
Live streaming: bit.ly/handsonsapdev
First SAP system: R/2 4.1d in 1987







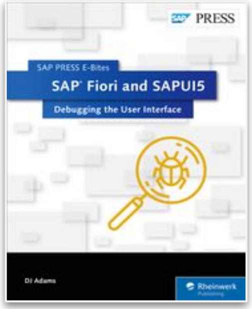
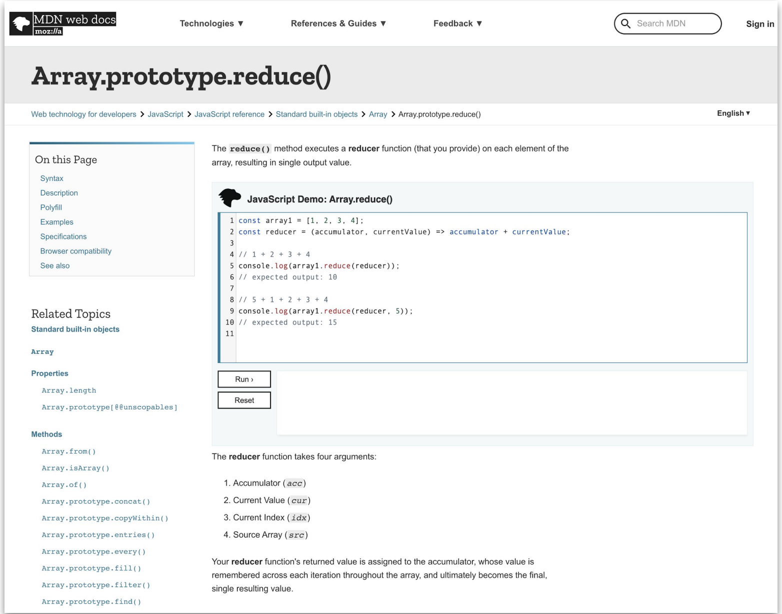



Image courtsey of Wikimedia



RedMonk Q120 Programming Language Rankings

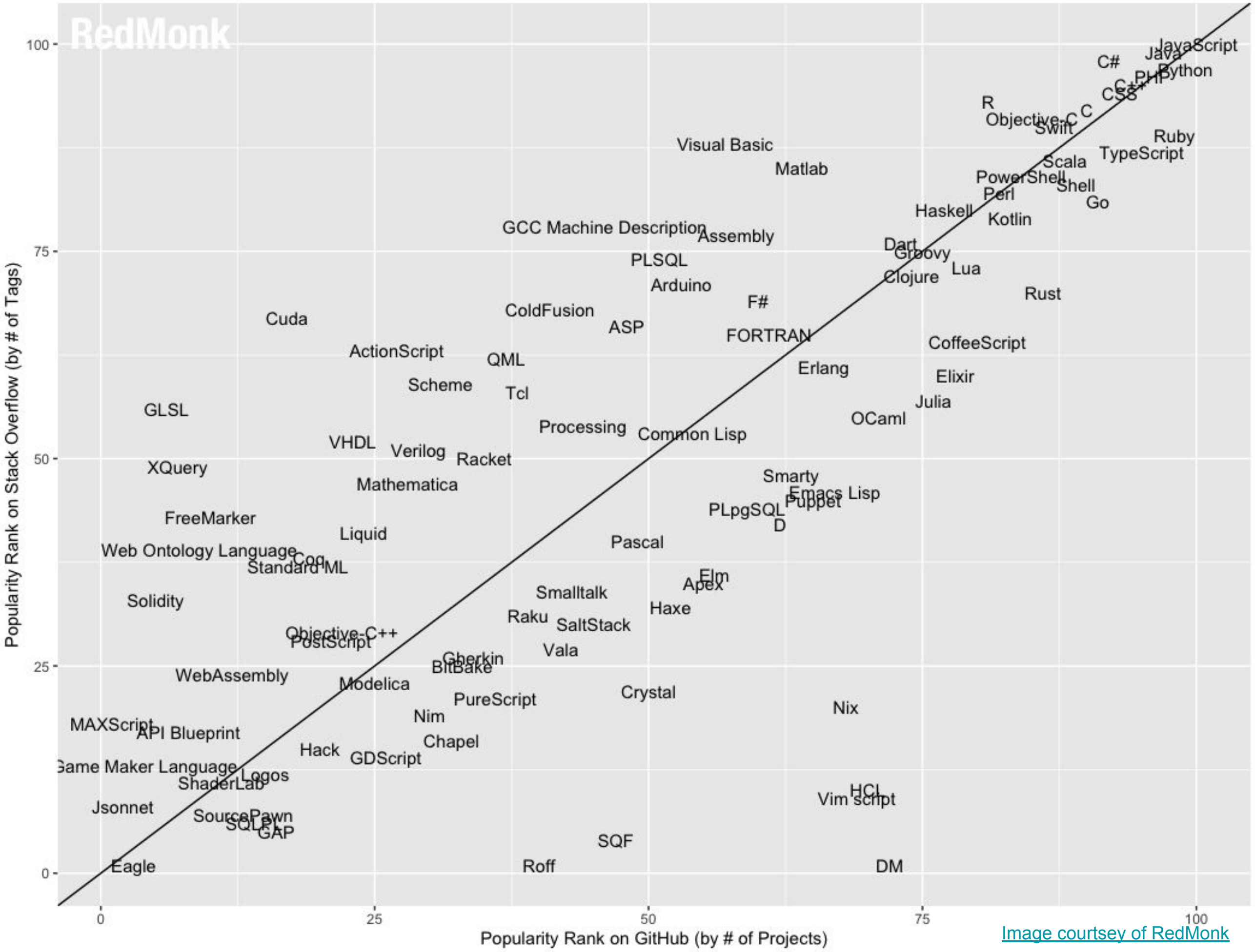
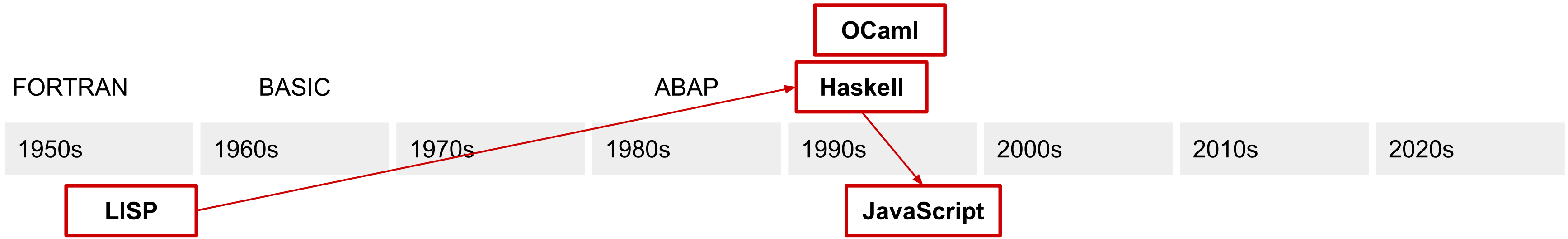


Image courtsey of RedMonk


OCaml
FORTRAN
BASIC
ABAP
Haskell
1950s
1960s
1970s
1980s
1990s
2000s
2010s
2020s
LISP
JavaScript

Part 1
Lists and recursion

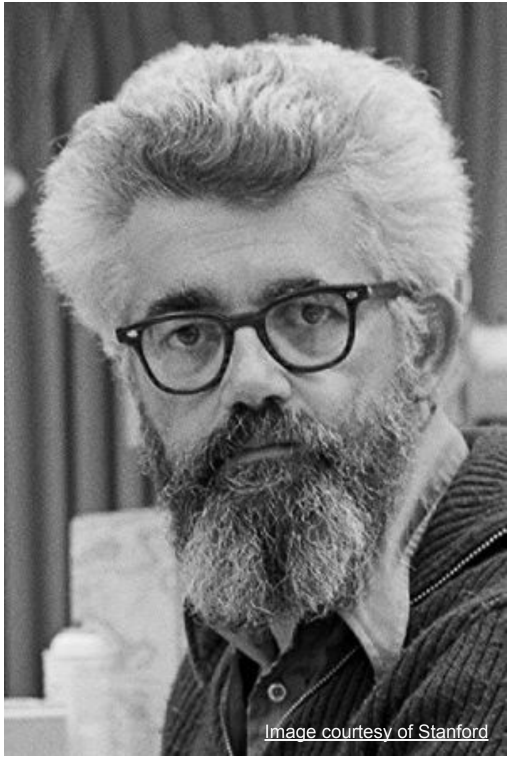
John McCarthy

Professor Emeritus of Computer Science at Stanford

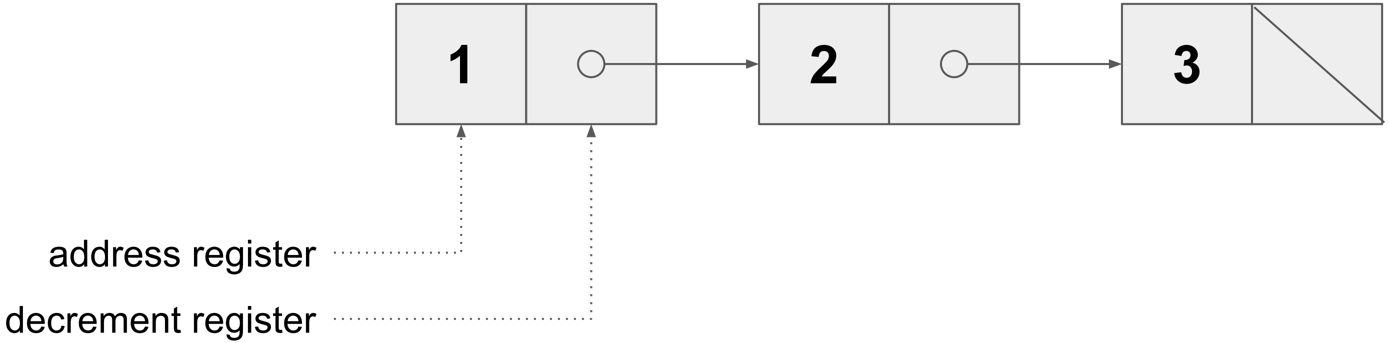
Coined the phrase "Artificial Intelligence"

Invented timesharing

Created LISP



LISP on the IBM 704 - car & cdr




TDX
13
my other car is a cdr

List Processing (LISP)

car : cdr

head : tail

first : rest

x : xs

LISP: "Iteration is a degenerate case of recursion"

```
(defun factorial (n)
    (if (= n 1)
    1
    (* n (factorial (- n 1)))))
```

LISP: "Iteration is a degenerate case of recursion"

```
(defun factorial (n)
    (if (= n 1)
    1
    (* n (factorial (- n 1)))))
```

```
(factorial 5)
120
```

LISP: "Iteration is a degenerate case of recursion"

```
(defun factorial (n)
    (if (= n 1)
    1
    (* n (factorial (- n 1)))))
```

```
(factorial 5)
(* 5 (factorial 4))
(* 5 (* 4 (factorial 3)))
(* 5 (* 4 (* 3 (factorial 2))))
(* 5 (* 4 (* 3 (* 2 (factorial 1)))))
(* 5 (* 4 (* 3 (* 2 1))))
120
```

Part 2
Pattern matching and abstraction

Haskell: Recursion & pattern matching

```haskell
take :: Int -> [a] -> [a]
take _ []     = []
take 0 _      = []
take n (x:xs) = x : take (n - 1) xs
```

```haskell
take 2 [1, 2, 3]
[1, 2]
```

Haskell: Factorial familiarity

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

```
factorial 5
120
```

Haskell: Factorial familiarity

```haskell
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

```
factorial 5
5 * (factorial 4)
5 * (4 * (factorial 3))
5 * (4 * (3 * (factorial 2)))
5 * (4 * (3 * (2 * (factorial 1))))
5 * (4 * (3 * (2 * (1 * (factorial 0)))))
5 * (4 * (3 * (2 * (1 * 1))))
120
```

Haskell: Wait, what?

```
take :: Int -> [a] -> [a]
take _ []     = []
take 0 _      = []
take n (x:xs) = x : take (n - 1) xs
```

```
take 2 [1, 2, 3]
1 : (take 1 [2, 3])
1 : 2 : (take 0 [3])
1 : 2 : []
[1, 2]
```

Haskell: List Processing (sum)

```haskell
sum :: Num a => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs
```

```
sum [1, 2, 3]
1 + sum [2, 3]
1 + 2 + sum [3]
1 + 2 + 3 + sum []
1 + 2 + 3 + 0
6
```

Haskell: List Processing (product)

```haskell
product :: Num a => [a] -> a
product []     = 1
product (x:xs) = x * product xs
```

```
product [1, 2, 3]
1 * product [2, 3]
1 * 2 * product [3]
1 * 2 * 3 * product []
1 * 2 * 3 * 1
6
```

Haskell: Comparing sum with product

```
sum :: Num a => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs
```

```
sum [1, 2, 3]
1 + sum [2, 3]
1 + 2 + sum [3]
1 + 2 + 3 + sum []
1 + 2 + 3 + 0
6
```

```
product :: Num a => [a] -> a
product []     = 1
product (x:xs) = x * product xs
```

```
product [1, 2, 3]
1 * product [2, 3]
1 * 2 * product [3]
1 * 2 * 3 * product []
1 * 2 * 3 * 1
6
```

Haskell: Getting to abstraction

```
sum []           = 0
sum (x:xs)       = x + sum xs

product []       = 1
product (x:xs) = x * product xs

and []           = True
and (x:xs)       = x && and xs
```

Haskell: foldr - one function to abstract them all

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)

foldr (+) 0 [1, 2, 3]
6
foldr (*) 1 [1, 2, 3]
6
foldr (&&) True [True, True, False]
False
```

Haskell: foldr - illustration

```
foldr (+) 0 [1, 2, 3]
(+) 1 (foldr (+) 0 [2, 3])
(+) 1 ((+) 2 (foldr (+) 0 [3]))
(+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))
(+) 1 ((+) 2 ((+) 3 0))
6
```

Haskell: Building on foldr with the power of currying

```
sum     = foldr (+)  0
product = foldr (*)  1
and     = foldr (&&) True
...
or      = foldr (||) False
```

OCaml: From fold_right to fold_left and accumulation

```
List.fold_right:

let rec fold_right op lst init = match lst with
  | []    -> init
  | x::xs -> op x (fold_right op xs init)

let sum lst = fold_right (+) lst 0

fold_right (+) [1;2;3] 0
1 + (2 + (3 + 0))
6
```

OCaml: From fold_right to fold_left and accumulation

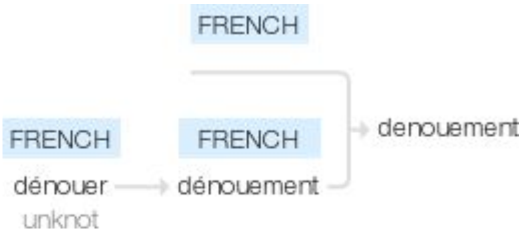
```
List.fold_left:

let rec fold_left op acc = function
  | []    -> acc
  | x::xs -> fold_left op (op acc x) xs

let sum = fold_left (+) 0

fold_left (+) 0 [1;2;3]
((0 + 1) + 2) + 3
6
```

Part 3
Denouement



Array.prototype.reduce

Syntax

```
arr.reduce(callback(accumulator, currentValue[, index[, array]]
)[, initialValue])
```

Syntax, err, reduced:

```
arr.reduce(callback(accumulator, currentValue)[, initialValue])
```

```
arr.reduce(fn(acc, x), init)
```

Array.prototype.reduce - example

```
arr.reduce(callback(accumulator, currentValue)[, initialValue])
```

```
[1, 2, 3].reduce((a, x) => a + x, 0)
((0 + 1) + 2) + 3
6

sum = (a, b) => a + b
[1, 2, 3].reduce(sum, 0)
((0 + 1) + 2) + 3
6
```

Array.prototype.reduce - the ultimate function

```
arr.reduce(callback(accumulator, currentValue)[, initialValue])

square = x => x * x
[1,2,3].map(square)
[1,4,9]

Array.prototype.ourmap = function(fn) {
    return this.reduce((a, x) => a.concat([fn(x)]), [])
}
[1,2,3].ourmap(square)
[1,4,9]
```

Array.prototype.reduce()

Related Topics
Standard built-in objects

`Array`

Properties

- `Array.length`
- `Array.prototype[@@unscopables]`

Methods

- `Array.from()`
- `Array.isArray()`
- `Array.of()`
- `Array.prototype.concat()`
- `Array.prototype.copyWithin()`
- `Array.prototype.entries()`
- `Array.prototype.every()`
- `Array.prototype.fill()`
- `Array.prototype.filter()`
- `Array.prototype.find()`

The `reduce()` method executes a **reducer** function (that you provide) on each element of the array, resulting in single output value.

JavaScript Demo: Array.reduce()

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15

```

Run ›
Reset

The **reducer** function takes four arguments:

1. Accumulator (`acc`)
2. Current Value (`cur`)
3. Current Index (`idx`)
4. Source Array (`src`)

Your **reducer** function's returned value is assigned to the accumulator, whose value is remembered across each iteration throughout the array, and ultimately becomes the final, single resulting value.

Reference material

LISP prehistory - Summer 1956 through Summer 1958
The origin of CAR and CDR in LISP
IBM 704 - Wikipedia
The RedMonk Programming Language Rankings: January 2020
Timeline of programming languages - Wikipedia
car & cdr - Programming in Emacs Lisp
Recursion and iteration
C9 Lectures: Dr. Erik Meijer - Functional Programming Fundamentals
Discovering the beauty of recursion and pattern matching
A Gentle Introduction to Haskell: Functions
Cornell CS 3110 - Higher-order Programming (in OCaml)
MDN web docs - Array.prototype.reduce()
Brief History of Haskell