
Neden fonksiyonel programlama? Çünkü günümüz işlemcilerinden maksimum verebilmek için algoritmaları hızlandırmak yerine concurrent yazılımlar yapmak daha etkili (Moore’s law), çünkü FP’nin temel amacı side-effect’leri yok etmek ve concurrency’nin en büyük problemi de bu (2 metod aynı anda aynı objeye erişiyor, vs.)

Hem Lodash gibi hem front-end’de hem Node.js’te sıklıkla kullandığımız kütüphaneler FP temelli, öte yandan asenkron işlemler için “bundan sonra bunu yap” demek FP’nin temelinde yatan prensiplerle çok yakından ilişkili (promise monad vs.).

Üstelik FP alışık olduğumuz programlama paradigmalarından tamamen farklı olduğu için son derece ufuk açıcı.

Tüm bu nedenlerden dolayı FP bir yazılımcının kendine katabileceği değerler sıralamasında İngilizcenin hemen ardında diyebiliriz

Konular: fonksiyonel programlama nedir, matematiksel temelleri, javascript’te fonksiyonlar ve özellikleri, map/reduce/filter gibi gündelik hayatta kullandığımız şeyler, bunların matematiksel temelleri, sonrasında pure functional programming’de daha çok yer bulan currying, monad, pattern matching ve memoization

## Fonksiyonel Programlama

**Programın her adımını matematiksel bir ifade olarak kabul eden programlama paradigması**

- Purity
  - Determinism
  - No Side-Effects
- Immutable Data
- Function Composition
  - Referential Transparency
- Functions as First-Class Citizens
- Higher-Order Functions



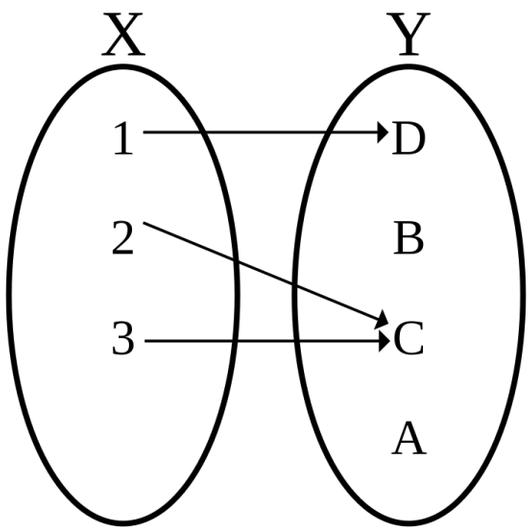

FP’de x = x + 1’in bir anlamı yok, çünkü matematikte böyle bir ifade yok. Her ifade matematiksel olduğuna göre, tamamiyle fonksiyonel programlamaya uygun biçimde yazılan programlar için başlı başına bir kanıttır (proof) denilir

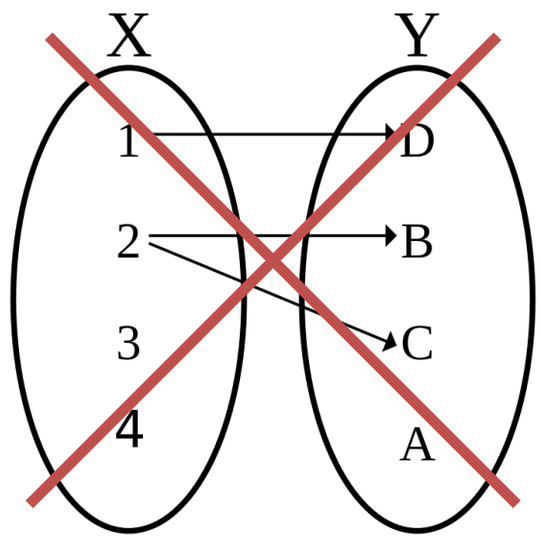
+ Lisp: 58, Erlang: 86, Haskell: 90, JS: 95, Scala: 2004, Clojure: 2007
+ Erlang: RabbitMQ, Whatsapp, ejabberd; Scala: Twitter, LinkedIn, Tumblr, Coursera; Clojure: Soundcloud; Node.js: Herkes!, Haskell/Lisp: Kimse :(

8

# Purity

$$f : X \mapsto Y$$






Matematiksel tanımlara dönecek olursak; fonksiyon, bir A kümesindeki değerleri bir B kümesindeki değerlere eşleyen ifadedir. A kümesindeki her değer için B kümesinde yalnızca bir tane karşılık bulunabilir (determinism).

f'nin ya da bir başka g fonksiyonunun X ya da Y kümesindeki değerleri etkilemesi/değiştirmesi, f(x)'in tanımını değiştirecektir, dolayısı ile böyle bir durumda bahsi geçen fonksiyonlar pure olmayacaktır (side-effects)

(WORKSHOP)

5 + w

# Immutable Data

```
def f(x: Int) = x + 5

var x = 5

f(x) // 10

x = 7

f(x) // 12

val y = 5

y = 12 // compile-time error!
```

```
val numbers = List(1, 2, 3)
// no push(), pop()
val newNumbers = numbers ++ List(4)
```



- Immutability: Bir değişkenin yalnızca 1 tanımının olması, üzerine yeni değer yazılamaması (benzer şekilde listelere yeni eleman eklenememesi, map'lerin value'larının değiştirilememesi)

JavaScript de dahil olmak üzere birçok dilde string'ler immutable'dır, karakter eklemek, çıkarmak ve değiştirmek yeni bir string oluşmasına sebep olur.

Örneğin Scala'da değişken tanımı var ya da val olarak yapılır. var olduğunda değişken mutable'dır, dolayısı ile IDE'ler gözden kaçmış bir şey olabileceğini düşünerek var kısmını kırmızıya boyar ve developer'ı uyarır.
val immutable için

Yeni JavaScript sürümü ile beraber herhangi bir değişken var x değil const x olarak tanımlandığı takdirde immutable olacak, ama üzerine atama yapılmaya çalışıldığında hata vermeden başarısız olduğu için çok da önemli değil bana göre.

(WORKSHOP)

4 + w

# Referential Transparency

$$f(x) = x + 2 \qquad g(x) = x!$$

$$5 * 5 = 25 \qquad 4 + 3 = 7$$

$$f(5 * 5) = f(25) \qquad g(4 + 3) = g(7)$$

$$(5 * 5) + 2 = 25 + 2 \qquad (4 + 3)! = 7!$$

$$27 = 27 \qquad 5040 = 5040$$

$$f(x * x) = (x * x) + 2 \qquad g(x + 3) = (x + 3)!$$



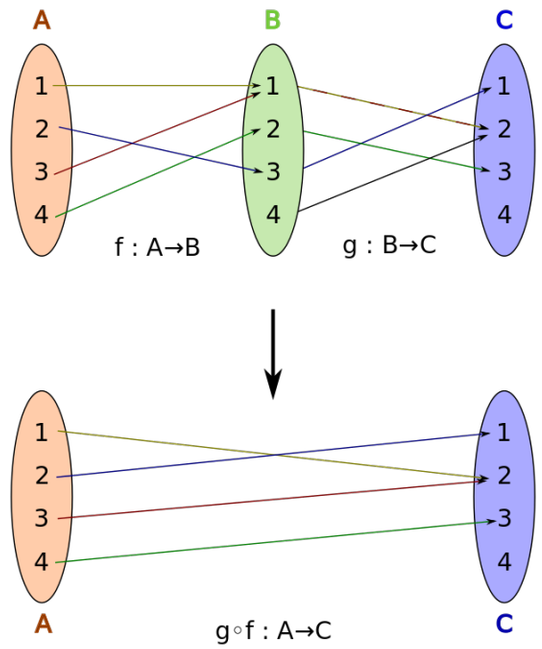
Bir sonraki slaytta bahsedeceğimiz composability ile biraz iç-içe bir konu. Bir fonksiyon çağrılırken bir başka matematiksel ifadenin kendisinin ya da sonuç değerinin parametre olarak verilmesinin, fonksiyonun sonucunu değiştirmemesi. Değeri yerine koyma ve expression'u evaluate etme olayı aslında. Matematikte tüm ifadeler referentially transparent'tır

Fonksiyonların birbirleri üzerine dağılma özelliğiyle karıştırılmamalı.

(WORKSHOP)

3 + w

# Function Composition




$$h(x) = g(f(x)) = (g \circ f)(x)$$

$$g(x) = 2x + 1 \quad f(x) = 3x + 4$$

$$h(x) = g(f(x)) = 2(3x + 4) + 1$$

$$h(x) = 6x + 9$$

$$f(5) = 3 \times 5 + 4 = 19$$

$$g(19) = 2 \times 19 + 1 = 39$$

$$h(5) = 6 \times 5 + 9 = 39$$



Birleşik fonksiyon olayı. Referential transparency sayesinde f:A->B ve g:B->C iki fonksiyon için şöyle bir fonksiyon yazılabilir: h(x) = fog(x) = f(g(x)) yani h(x) = f(g(x))

(WORKSHOP)

4 + w

# Functions as First-Class Citizens

```
function f(x) {
    ...
}

function foo() {
    var g = function(x) {
        ...
    };

    g(5);
}

function bar() {
    return function(x) {
        ...
    };
}

var someObject = {
    foo: 'bar',
    baz: function() {
        ...
    }
};

[1, 2, 3].map(function(x) {
    ...
});
```



- Functions as First-Class Citizen: Fonksiyonların herhangi bir scope’ta tanımlanabilmesi, başka fonksiyonlara parametre olarak geçilebilmesi, bir fonksiyonun dönüş değeri olarak dönülebilmesi.

Anonim fonksiyon ya da lambda expression olayına olanak sağlıyor ama beraberinde bir sonraki slaytta yer alacak olan closure kavramını da getiriyor. Higher-order function da yine fonksiyonların first-class citizen olması sayesinde varolabiliyor.

(WORKSHOP)

2 + w

# JavaScript'te Fonksiyonlar - Closure

```
var x = 5;

function f(x) {
    x = x + 6;
    return x;
}

console.log('f(10):', f(10)); // 16

function g(y) {
    x = x + 5;
    return y * x;
}

console.log('g(10):', g(10)); // 100
console.log('g(10):', g(10)); // 150
console.log('f(10):', f(10)); // 16
```



Her kod bloğu, tanımlı olduğu bloğun üst bloklarında tanımlanmş öğeleri görebilir ve değiştirebilir. Closure takibi yaparken süslü parantezleri takip etmek bir nebze kolaylık sağlayacaktır

Fonksiyonlara verilen parametreler, fonksiyonların tanımlı oldukları closure'daki aynı isimli değerlerin önüne geçer. Mesela örnekte f(x)'in içerisindeki x ile en üstteki x farklı şeyler, o nedenle f(x) için dış x'in bir anlamı yok.

g(y) fonksiyonunda görülebileceği gibi bu durum bir side-effect'e yol açabilir, o nedenle closure'daki değerler değiştirilirken çok dikkatli olunması gerek.

4

# JavaScript'te Fonksiyonlar - This Context

```
function f() {
    console.log(this.toString());
}
f(); // [object global]

var Foo = function() {
    this.x = 15;
    this.g = function() {
        console.log(this);
    };
};
var foo = new Foo();
console.log(foo); // { x: 15, g: [Function] }
foo.g(); // { x: 15, g: [Function] }

var bar = {
    x: 25,
    baz: function() {
        console.log(this);
    }
}
bar.baz(); // { x: 25, baz: [Function] }
```



this, bir nevi, fonksiyonun tanımlı olduğu kapsamı gösteren bir referans diyebiliriz. Aslında prototypal inheritance ile yakından ilişkisi var, ama bu apayrı ve hayli derin bir konu, o yüzden ona çok detaylı girmeyeceğiz.

Kontrolsüz side-effect'lere yol açmamak için değişkenleri this kapsamına tanımlayabiliriz ama bu side-effect'lerden tamamen kurtulacağımz anlamına gelmiyor. this'in kullanımı çok daha farklı side-effectlere yol açabilir, o nedenle halen dikkatli olmak gerekiyor.

Sahipsiz fonksiyonlarda this context'i tanımlı oldukları bloğun this'i ile aynı. En üst seviyede tanımlananlar için bu şey global objesi (front-end'de window)

(WORKSHOP)

3 + w

# Higher-Order Functions

```
var f = function(x) {
    return x * 2;
};
[ 1, 2, 3 ].map(f); // [ 2, 4, 6 ]

$('.button').on('click', function(event) {
    ...
});
```

$$\sum_{i=a}^{b} f(i) \qquad \prod_{i=a}^{b} f(i) \qquad \int_{a}^{b} f(x)dx \qquad \sum_{i=1}^{n} f(t_i)\Delta i$$



- Higher-Order Function: Parametre olarak başka fonksiyonları alan ve kullanan fonksiyonlar.

Örneğin matematikteki toplam ve çarpım (sigma ve pi) işlemleri birer higher-order function. Benzer şekilde limit ve integral de öyle.

İntegral'in x'in 0'a yakınsayan en ufak değeri dx için, verilen limitler içerisindeki f(dx) değerlerinin toplamı olduğunu hatırlayalım (örn. f(x) = x^2 için x: [a, b] aralığında parabolün altında kalan alanı verir)

3

# Map

```
map :: (a -> b) -> [a] -> [b]
```

```
function f(x) {
    return x * 2;
}

var numbers = [ 1, 2, 3 ];

var newNumbers = [];
for (var i = 0; i < numbers.length; i++) {
    var number = numbers[i];
    var newNumber = f(number);
    newNumbers.push(newNumber);
}

console.log('newNumbers:', newNumbers); // [ 2, 4, 6 ]
```



İlk higher-order function'ımız. İlk slaytta map'in yaptığı işi klasik programlama yöntemiyle yapma örneği var.

Tanım Haskell'den, (a'dan b'ye bir fonksiyon alıp, bu fonksiyonu a içeren bir listeye uygulayıp sonucunda bir b listesi üretmek)

Tanımdaki [] kısmı array'e benzer şekilde listeyi ifade ediyor (aslında list değil de monad)

https://en.wikipedia.org/wiki/Map_%28higher-order_function%29

3

# Map

```
function map(f, arr) {
    var ret = [];
    for (var i = 0; i < arr.length; i++) {
        var number = arr[i];
        var newNumber = f(number);
        ret.push(newNumber);
    }
    return ret;
}

function f(x) {
    return x * 2;
}

function g(x) {
    return x + 3;
}

var numbers = [ 1, 2, 3 ];

console.log('map(f, numbers):', map(f, numbers)); // [ 2, 4, 6 ]
console.log('map(g, numbers):', map(g, numbers)); // [ 4, 5, 6 ]
console.log('map(f, map(g, numbers)):', map(f, map(g, numbers))); // [ 8, 10, 12 ]
```



map’i bu şekilde genelleştirmek mümkün, ama en alttaki durumda okunabilirliğin azaldığını, composability’nin azaldığını görüyoruz

2

# Map

```
Array.prototype.map = function(f) {
    var ret = [];
    for (var i = 0; i < this.length; i++) {
        var val = this[i];
        var newVal = f(val);
        ret.push(newVal);
    }
    return ret;
};

function f(x) { return x * 2; }
function g(x) { return x + 3; }
function h(x) { return f(g(x)); }

var numbers = [ 1, 2, 3 ];

console.log('numbers.map(f):', numbers.map(f)); // [ 2, 4, 6 ]
console.log('numbers.map(g):', numbers.map(g)); // [ 4, 5, 6 ]
console.log('numbers.map(g).map(f):', numbers.map(g).map(f)); // [ 8, 10, 12 ]
console.log('numbers.map(h):', numbers.map(h)); // [ 8, 10, 12 ]
```



Haskell'ın akisne object-functional davranabildiğimiz map fonksiyonunu için Array class'ına tanımlayabiliriz, bu bize çok daha okunabilir ve rahat compose edilebilir bir map() fonksiyonu üretmemizi sağlar

Fonksiyonu Array'in prototype'ına tanımladığı için this referansı, üzerinden çağrıldığı array instance'ını gösterecektir

(WORKSHOP)

2 + w

# Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
Array.prototype.filter = function(p) {
    var ret = [];
    for (var i = 0; i < this.length; i++) {
        var val = this[i];
        var isTrue = p(val);
        if (isTrue) {
            ret.push(val);
        }
    }
    return ret;
};

function f(x) { return x % 2 == 0; }
function g(x) { return x < 3; }

var numbers = [ 1, 2, 3, 4 ];

console.log('numbers.filter(f):', numbers.filter(f)); // [ 2, 4 ]
console.log('numbers.filter(g):', numbers.filter(g)); // [ 1, 2 ]
console.log('numbers.filter(f).filter(g):', numbers.filter(f).filter(g)); // [ 2 ]
```



filter bir diğer higher-order function, bir predicate alıp, içerisinde bu testten geçen değerlerin olduğu yeni bir monad dönüyor

map'te olduğu gibi filter'ı da array'e ekleyebiliriz, ki bu bize hem map() hem de filter()'ı birlikte compose etme imkanı sunar

https://en.wikipedia.org/wiki/Filter_%28higher-order_function%29

(WORKSHOP)

2 + w

# Reduce

**foldl** :: (b -> a -> b) -> b -> [a] -> b

```
Array.prototype.reduce = function(f, initial) {
    var accumulator = initial;
    for (var i = 0; i < this.length; i++) {
        var val = this[i];
        accumulator = f(accumulator, val);
    }
    return accumulator;
};

function f(acc, x) { return acc + x; }
function g(acc, x) { return acc + ', ' + x; }

var numbers = [ 1, 2, 3 ];

console.log('numbers.reduce(f, 0):', numbers.reduce(f, 0)); // 6
console.log('numbers.reduce(g, \'A\'):', numbers.reduce(g, 'A')); // 'A, 1, 2, 3'
```



Diğer dillerdeki karşılığı foldLeft
Bir başlangıç değeri ile birleştirici bir fonksiyon alır, ilk adımda başlangıç değeri ve listedeki ilk elemanı kullanacak şekilde verilen fonksiyonu çağırır ve bir sonraki adıma o anki birleşim değerini vererek ilerler

Bir de diğer dillerde foldRight var, foldLeft her adımda bir sonraki adımda kullanılacak toplamsal değeri hesaplarken, foldRight son elemana gelene kadar her adımda o adımdaki toplam işlemini bir fonksiyon olarak stack'e ekler, dolayısı ile sınırlı stack derinliği olan dillerde patlar

https://en.wikipedia.org/wiki/Fold_%28higher-order_function%29

(WORKSHOP)

5 + w

# ForEach

```
foreach :: (a -> b) -> [a] -> ()
```

```
Array.prototype.forEach = function(f) {
    for (var i = 0; i < this.length; i++) {
        var val = this[i];
        f(val);
    }
};

function f(x) { console.log(x); }
function g(x) { return x + 3; }

var numbers = [ 1, 2, 3 ];

console.log('numbers.forEach(f):', numbers.forEach(f)); // undefined
console.log('numbers.forEach(g):', numbers.forEach(g)); // undefined
```



Aslında hep undefined dönen bir map() gibi de düşünülebilir

1

# Map, Filter, Reduce Workshop

- filter() fonksiyonunun yalnızca reduce() kullanılarak yeniden yazılması
- bir array'deki en ufak değeri bulan fonksiyon (Array.prototype.min)
- bir cümledeki tüm karakterlerin sayısını dökecek fonksiyonun for/while/do..while kullanılmadan yazılması
  - "Hello, world!" -> { H: 1, e: 1, l: 3, o: 2, ',': 1, ' ': 1, w: 1, r: 1, d: 1, '!': 1 }
- http://www.codewars.com/kata/parseint-reloaded
  - "twenty-eight" -> 28, "six hundred and sixty six" -> 666, "forty two" -> 42, "one million and one" -> 1000001
  - Template: https://gist.github.com/ygunayer/d52d09a670dfe3ba1f58



(WORKSHOP)

1 + w

# Daha Fazla Higher-Order Function

- Array.prototype.every(p) - ‘p’ tüm elemanlar için geçerli mi?
- Array.prototype.some(p) - ‘p’ en az 1 eleman için geçerli mi?

- Lodash - Hem front-end, hem back-end uyumlu bol özellikli utility kütüphanesi
  - take, takeWhile
  - drop, dropWhile
  - merge
  - memoize
  - curry
  - debounce
  - …
  - https://lodash.com/docs



(WORKSHOP)

Workshop esnasında lodash’in kolay compose edilememesinden ve bu nedenle de lodash-fp adında bir wrapper’ı yazıldığından, tüm bunlara alternatif olarak da Ramda.js’in kullanılabileceğinden bahsedilecek

1 + w

# Currying

$$f(x, y) = x^2 + y$$

$$f(2, y) = g(y) = 2^2 + y$$

$$f(2, 5) = g(5) = 2^2 + 5$$

$$f(2, 5) = f(2)(5)$$

```
function f(x, y) {
    return x * x + y;
}

function h(x) {
    return function(y) {
        return x * x + y;
    };
}

f(2, 5) == h(2)(5);
```



A technique of transforming a multi-argument function in such a way that it can be called as a chain of functions, each with a single argument.

Aslında workshop'larda currying'i zaten uyguladık (first-class-citizens, props @ map, gender @ filter…)

(map/reduce/filter kod örneklerinde ilgili kısımlar gösterilebilir)

3 + w

# Monad’lar

- “A monad is just a monoid in the category of endofunctors, what’s the problem?”

- Belirli bir tipte değer dönebilme özelliği olan kapsayıcı tip
  - maybe
  - either, try
  - list
  - promise



Maybe: İçi boş ya da dolu olabilen monad. Dolu ise verilen fonksiyonu çalıştırır, değilse çalıştırmaz
trait Maybe, object None, case class Some(value: …)
List: Her elemanın birer Maybe gibi davranan hücreler olan klasik bir linked list uygulaması gibi düşünülebilir. dolu ise hem bir değeri, hem de kendinden bir sonraki elemanın referansını barındıran klasik bir linked list uygulaması.
trait List, object EmptyList extends List, case class NoneEmptyList(value: …, next: List)
Either: Maybe gibi, fakat iki farklı tipten birini barındırır, try-catch veya tuple modellemesinde kullanılabilir
trait Either[T, R], case class Left(value: T), case class Right(value: R)
try-catch için: Either[T, Error], Left <- Success(value: T), Right <- Failure(err: Error)
Promise: Either üzerine modellenen bir başka yapı, asenkron bir işlem sonucunda kullanılır

(WORKSHOP)

3 + w

# Partial Functions & Pattern Matching

$$f(x) = |x|$$

$$f(x) = \begin{cases} -x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

```
Math.abs = function(x) {
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
};
```

```
def abs(x: Int): Int = x match {
    case n if n < 0 => -n
    case n if n >= 0 => n
}
```



Matematikte bir fonksiyon, bağlı olduğu değerlere bağlı farklı alt ifadeler için farklı sonuçlar üretebilir. Örnek: mutlak değer fonksiyonu. Bu üst ifadeye alt ifadeler bazında farklı davranışlar verme olayı pattern matching. Her dilde olan if, switch/case gibi şeyler aslında birer pattern matching uygulaması, sadece kullanılabilirlik açısından dilin içerisine yedirilmiş. %100 fonksiyonel bir dilde bunlar da compose edilebilir bir formata sokulabilir.

Partial function'lar ise tanımlı olduğu kümelerdeki her değer için bir karşılığı bulunmayan fonksiyonlara deniyor. Örneğin ilk slaytlardaki fonksiyon.

Dolayısı ile eğer bir pattern matching bloğu içerisinde kapsamadığımız bir blok varsa (yani bir fonksiyonun içinde bazı durumlarda bir şey return etmiyorsak) bu tarz fonksiyonlar partial function oluyor.

3

# More Pattern Matching

```
val numbers = 1 :: 2 :: 3 :: 4 :: 5 :: Nil

def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case head :: tail => head + sum(tail)
}
```

```
val someList: List[Any] = List(1, "Hello", 26.444f, List(4, 5, 6))

def foo(list: List[Any]): String = list match {
  case Nil => "Done!"
  case x :: xs => (x match {
    case n: Int        => "Int(" + n + ")"
    case s: String     => "String(" + s + ")"
    case f: Float      => "Float(" + f + ")"
    case l: List[Any] => "List(" + foo(l) + ")"
    case _             => "Other"
  }) + ", " + foo(xs)
}

println(foo(someList)) // Int(1), String(Hello), Float(26.444),
                       // List(Int(4), Int(5), Int(6), Done!), Done!
```



Scala’da List() class’ı direkt olarak bir monad.

head :: tail formatı list decomposition adında özel bir yöntem. List(1, 2) ile 1 :: 2 :: Nil aynı anlama geliyor. Pattern matching’de de böyle kullanılabiliyor. Adlarından da anlaşılabileceği gibi head burada listenin ilk elemanını, tail ise bu elemandan sonraki elemandan itibaren listenin geri kalanını veriyor. Tek elemanlı bir liste head :: tail ile parçalandığında tail kısmı haliyle Nil (ya da List()) oluyor

Bu syntax pattern matching’de kullanıldığı için boş bir liste bu formatta yakalanamazdı, case Nil => … bu nedenle kullanılıyor

Yakalanan eleman sayısı bu kadarla kısıtlı değil, x :: xs :: xss :: Nil gibi bir şey de olabilirdi

4

# Memoization

- Fonksiyonların sonuçlarını parametrelerine göre cache'leyerek performans artırma yöntemi

```
var cache = {};
var f = function(x) {
    if (x in cache) {
        return cache[x];
    }
    console.log('Calculated for', x);
    var result = x * 2;
    cache[x] = result;
    return result
};

console.log('f(2):', f(2)); // 'Calculated for 2' ... 4
console.log('f(2):', f(2)); // 4
```



Yalnızca pure fonksiyonlarda geçerli, aksi halde beklenmedik sonuçlara yol açabilir.
O nedenle buradaki side effect'e dikkat. İzleyiciye bu koddaki yanlışın (side-effect'in ta kendisi) ne olduğu sorulabilir.

(WORKSHOP)

3 + w

# Kaynak Önerileri

- JavaScript
  - fn.js - Pure Functional Programming in JavaScript
    - http://eliperelman.com/fn.js/
  - Ramda
    - http://fr.umio.us/why-ramda/
  - Functional Programming in 5 Minutes
    - http://slides.com/gsklee/functional-programming-in-5-minutes
  - Monads in JavaScript
    - https://curiosity-driven.org/monads-in-javascript
- Scala
  - Functional Programming Principles in Scala - Martin Odersky @ Coursera
    - https://www.coursera.org/course/progfun
  - Fehmi Can Sağlam (cimri.com)
    - http://fehmicansaglam.net/
    - http://fehmicansaglam.net/bora-gonul-ile-fonksiyonel-programlama-dilleri-uzerine-bir-soylesi/
- Haskell
  - Haskell.org
    - https://wiki.haskell.org/Functional_programming
    - https://wiki.haskell.org/Why_Haskell_just_works

# Kitap Önerileri

- JavaScript
  - Mostly Adequate Guide to Functional Programming
    - https://drboolean.gitbooks.io/mostly-adequate-guide/content/
- Haskell
  - Learn You a Haskell for Greater Good!
    - http://learnyouahaskell.com/chapters
  - Purely Functional Data Structures
    - http://www.amazon.com/Purely-Functional-Structures-Chris-Okasaki/dp/0521663504/
- Erlang
  - Learn You Some Erlang
    - http://learnyousomeerlang.com/content
- Lisp
  - Structure and Interpretation of Computer Programs
    - https://github.com/sarabander/sicp-pdf/raw/master/sicp.pdf
- Scala
  - Programming in Scala: A Comprehensive Step-by-Step Guide
    - http://www.amazon.com/Programming-Scala-Comprehensive-Step---Step/dp/0981531644/
  - Functional Programming in Scala
    - http://www.amazon.com/Functional-Programming-Scala-Paul-Chiusano/dp/1617290653/

# Sorular

"Haskell is Useless (!)" - Simon Peyton Jones
https://www.youtube.com/watch?v=iSmkqocn0oQ