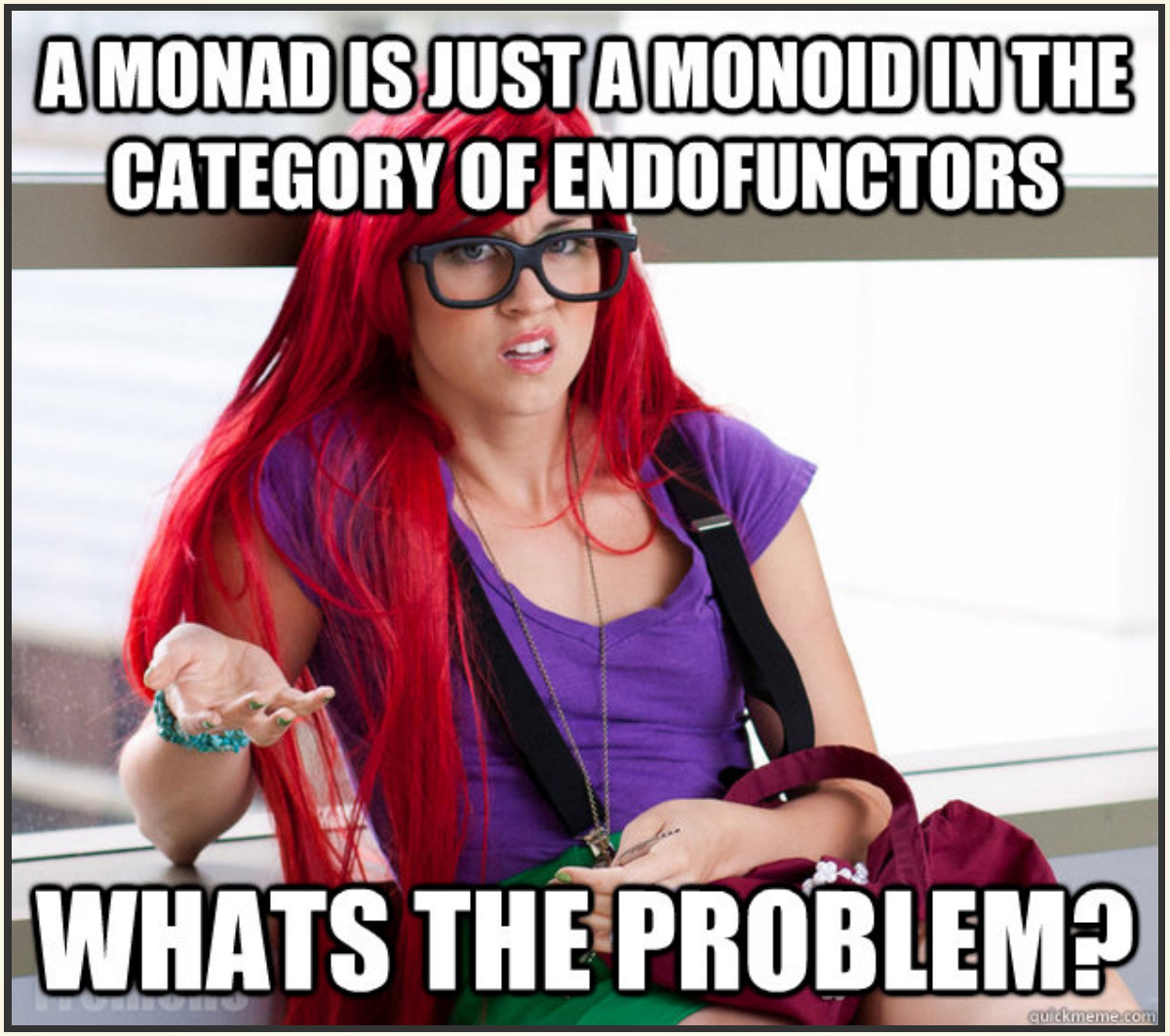
MONADEN

CARSTEN KÖNIG

HTTPS://CARSTENKOENIG.GITHUB.IO/DWX2014_MONADEN CODE

MONADE ... WAS IST DAS?



... darum geht es heute *nicht*

ES GEHT UM: SYNTAX

C#

```
IEnumerable<Outfit> Outfits()
{
    return
        from schuhe in Schuhe
        from hose in Hosen
        from hemd in Hemden
        select new Outfit {Schuhe = schuhe, Hose = hose, Hemd = hemd};
}
```

Monade: `IEnumerable`

ES GEHT UM: DSLS

F#

```
let fetchAsync(url:string) : Async<string> =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            return! webClient.AsyncDownloadString(uri)
        with
            | ex -> printfn "Fehler: %s" ex.Message
    }
```

Monade: `Async`

ES GEHT UM: KAPSELUNG

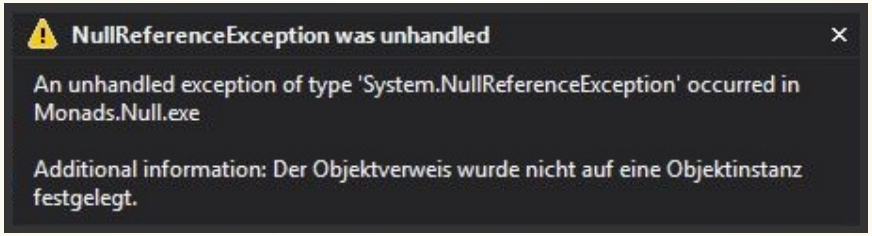
Haskell

```haskell
main :: IO ()
main = putStrLn "Hello, World!"
```

Monade: `IO`

MOTIVATION

NULLREFERENCEEXCEPTION



I call it my billion-dollar mistake.

Tony Hoare

GUARDS

```
var customer = GetCustomer(5);
if (customer != null)
{
  var orders = GetOrdersFor(customer);
  if (orders != null)
  {
    // ...
  }
}
return null;
```

EIN MUSTER...

```
var customer = GetCustomer(5);
if (customer != null)
{
  var orders = GetOrdersFor(customer);
  if (orders != null)
  { /* ... */ }
  else
    return null;
}
else
  return null;
```

REFACTOR

```
public tR Guard<tS, tR>(tS source, Func<tS, tR> f)
  where tS : class
  where tR : class
{
  if (source != null) return f(source);
  else return null;
}
```

```
return Guard(GetCustomer(5), customer =>
  Guard(GetOrdersFor(customer), order => {
    // ...
  }));
```

SYNTACTIC SUGAR

```
public static tR Guard<tS, tR>(this tS source, Func<tS, tR> f)
  where tS : class
  where tR : class
{
  if (source != null) return f(source);
  else return null;
}
```

```
return GetCustomer(5).Guard(customer =>
  GetOrdersFor(customer).Guard(order => {
    // ...
  }));
```

KÖNNEN WIR DAS ELEGANTER LÖSEN?

C# bekommt demnächst wahrscheinlich ein *monadic null checking*

```
GetCustomer(5)?.Orders()?. ...
```

MAYBE


```
Customer GetCustomer(int customerNumber)
```

Wenn eine Funktion **partiell** ist, dann sollte sie es auch *zugeben*

Sag es im Namen

```
Customer TryGetCustomer(int customerNumber)
```

aber: davon *merkt* der Compiler nichts

Try/Can Pattern

```
bool TryGetCustomer(int customerNumber, out Customer customer)
```

Aber: *imperatives* Handling

TYPEN - WAS SONST?

```
Maybe<Customer> TryGetCustomer(int customerNumber)
```

```
type Maybe<'a> =
  | Just of 'a
  | Nothing
```

Hinweis: tatsächlich würde man F# `Option<'a>` verwenden

für C# gibt es schöne Hilfsmittel in FSharpx.Core

DON'T LOOK SO DIFFERENT

```
var customer = TryGetCustomer(5);
if (customer.IsJust)
{
    var orders = TryGetOrdersFor(customer.Value);
    if (orders.IsJust)
    { /* ... berechne irgendwas vom Typ Rückgabe */ }
    else
        return Maybe.Nothing<Rückgabe>();
}
else
    return Maybe.Nothing<Rückgabe>();
```

REFACTOR (AGAIN)

```
return Bind(TryGetCustomer(5), customer =>
  Bind(TryGetOrdersFor(customer), order => {
    // ...
  }));
```

BIND

```
Maybe<tR> Bind<tS, tR>(Maybe<tS> source, Func<tS, Maybe<tR>> f)
```

```
{
  if (source.IsNothing) return Maybe.Nothing<tR>();
```

```
  else return f(source.Value);
}
```

SUGAR ... MORE SUGAR

C#: LINQ

```
return
from customer in TryGetCustomer(5)
from orders in TryGetOrdersFor(customer)
select ...
```

F#: COMPUTATION EXPRESSIONS

```
maybe {
    let! customer = tryGetCustomer 5
    let! orders   = tryGetOrders customer
    return ...
}
```

HASKELL: DO-NOTATION

```
beispiel :: Maybe Orders
beispiel = do
  customer <- tryGetCustomer 5
  orders   <- tryGetOrdersFor customer
  return orders
```

WAS MÜSSEN WIR DAFÜR TUN?

LINQ

`SelectMany` implementieren:

```
public static Maybe<tR> SelectMany<tS, tR>
  (this Maybe<tS> source, Func<tS, Maybe<tR>> selector)
{
    return Bind(source, selector);
}
```

LINQ

nochmal `SelectMany`:

```
public static Maybe<tR> SelectMany<tS, tCol, tR>
  (this Maybe<tS> source,
   Func<tS, Maybe<tCol>> collectionSelector,
   Func<tS, tCol, tR> resultSelector)
{
    if (source.IsNothing) return Nothing<tR>();
    var col = collectionSelector(source.Value);
    if (col.IsNothing) return Nothing<tR>();
    return resultSelector(source.Value, col.Value);
}
```

im wesentlichen `Bind` + `Tranformation/Select`

F#

Builder implementieren:

```
type MaybeBuilder() =
    member x.Bind(m, f)     = bind m f
    member x.Return(v)      = Just v

let maybe = MaybeBuilder()
```

HASKELL

`Monad` Typklasse instanzieren:

```haskell
instance Monad Maybe where
  return = Just
  Nothing >>= _ = Nothing
  Just a  >>= f = f a
```

LIST MONAD

ZIEL

nicht-deterministische Ergebnise repräsentieren

oder einfach

arbeiten mit Funktionen, die *mehrere mögliche Ergebnisse* haben können

WAS IST ZU TUN?

Listen `[a]` sollen zum Monaden werden

Gesucht: Implementation von

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

und

```
return :: a -> [a]
```

RETURN

```
return :: a -> [a]
```

Intuition: einen Wert in eine Liste **packen**

RETURN

Haskell:

```haskell
return :: a -> [a]
return x = [x]
```

F#:

```fsharp
let return (x : 'a) : 'a list = [x]
```

BIND

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

Intuition:

- jedes Element gibt wieder eine Liste
- zusammen also eine Liste von Listen
- diese gilt es **plattzumachen**

BIND

Haskell:

```
(>>=) :: [a] -> (a -> [b]) -> [b]
xs (>>=) f = concatMap f xs
```

F#:

```
let bind (xs : 'a list) (f : 'a -> 'b list) : 'b list =
  xs |> List.collect f
```

BEISPIEL

Permutation einer Liste/Aufzählung

Rezept:

- Nimm ein Element `x` heraus
- Nimm eine Permuation `xs` der restlichen Elemente
- `x : xs` ist eine Permutation
- Liste alle solchen Permuationen auf

F#

```
let ohne x =
    List.filter ((<>) x)

let rec permutationen = function
  | [] -> [[]]
  | xs -> [ for x in xs do
            let rest = xs |> ohne x
            for xs' in permutationen rest do
            yield (x::xs') ]
```

HASKELL

```haskell
import Data.List (delete)

perm :: (Eq a) => [a] -> [[a]]
perm [] = return []
perm xs = do
  x <- xs
  xs' <- perm $ delete x xs
  return $ x:xs'
```

C#/LINQ

```
IEnumerable<IEnumerable<T>> Permuationen<T>(IEnumerable<T> items)
{
    if (!items.Any()) return new[] {new T[] {}};

    return
        from h in items
        from t in Permuationen(items.Where(x => !Equals(x, h)))
        select h.Vor(t);
}
```

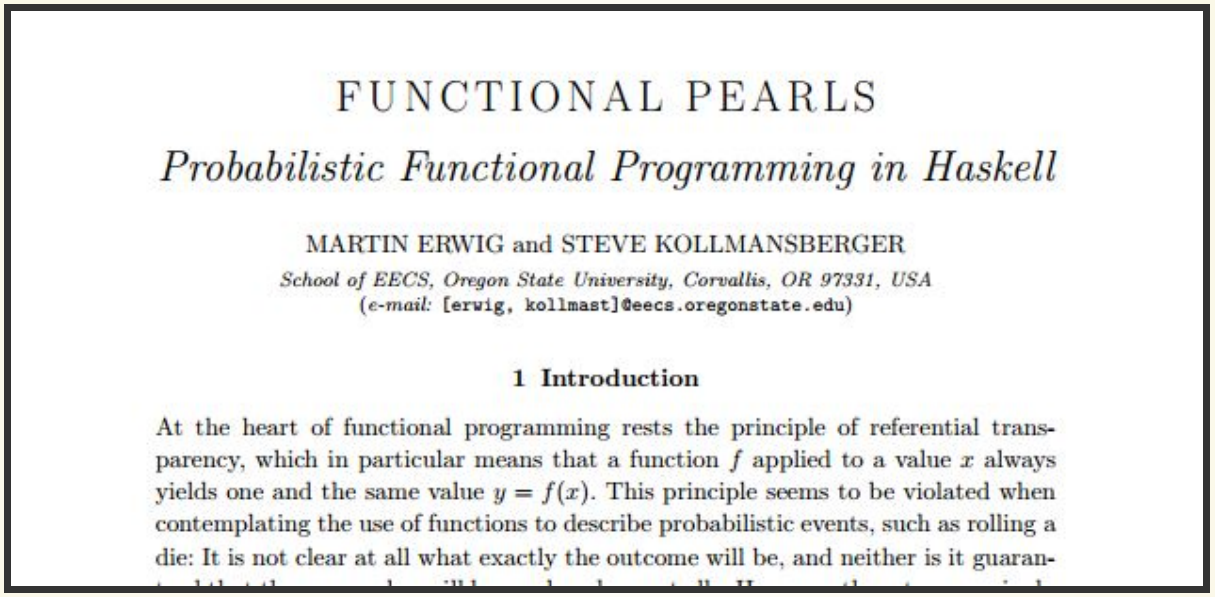
C#/LINQ

Hilfsfunktion

```
IEnumerable<T> Vor<T>(this T value, IEnumerable<T> rest)
{
    yield return value;
    foreach (var v in rest)
        yield return v;
}
```

WAHRSCHEINLICHKEITEN

INSPIERIERT DURCH



FUNCTIONAL PEARLS

Probabilistic Functional Programming in Haskell

MARTIN ERWIG and STEVE KOLLMANSBERGER

School of EECS, Oregon State University, Corvallis, OR 97331, USA
(*e-mail:* [erwig, kollmast]@eecs.oregonstate.edu)

1 Introduction

At the heart of functional programming rests the principle of referential transparency, which in particular means that a function f applied to a value x always yields one and the same value $y = f(x)$. This principle seems to be violated when contemplating the use of functions to describe probabilistic events, such as rolling a die: It is not clear at all what exactly the outcome will be, and neither is it guaran-

Functional Pearls: Probabilistic Functional Programming in Haskell

RISIKO

```
let ``Verluste beim Risiko (3 gegen 2)`` =
    vert {
        let! offensive = nWürfel 3
        let! defensive = nWürfel 2
        let defensivVerluste =
            List.zip (offensive |> List.sort |> List.tail)
                     (defensive |> List.sort)
            |> List.sumBy (fun (o,d) -> if d >= o then 0 else 1)
        return sprintf "%d:%d" (2-defensivVerluste) defensivVerluste
    } |> printVerteilung
```

```
>
"0:2": 37.17%
"1:1": 33.58%
"2:0": 29.26%
```

DARSTELLUNG

Eigentlich nur die *List*-Monade mit Wahrscheinlichkeiten

```
type Wahrscheinlichkeit = double
type Verteilung<'a> = ('a * Wahrscheinlichkeit) list
```

RETURN

Entspricht dem **sicheren Ereignis**

```
let sicher (a : 'a) : Verteilung<'a> =
    [a, 1.0]

let returnM (a : 'a) : Verteilung<'a> =
  sicher a
```

BIND

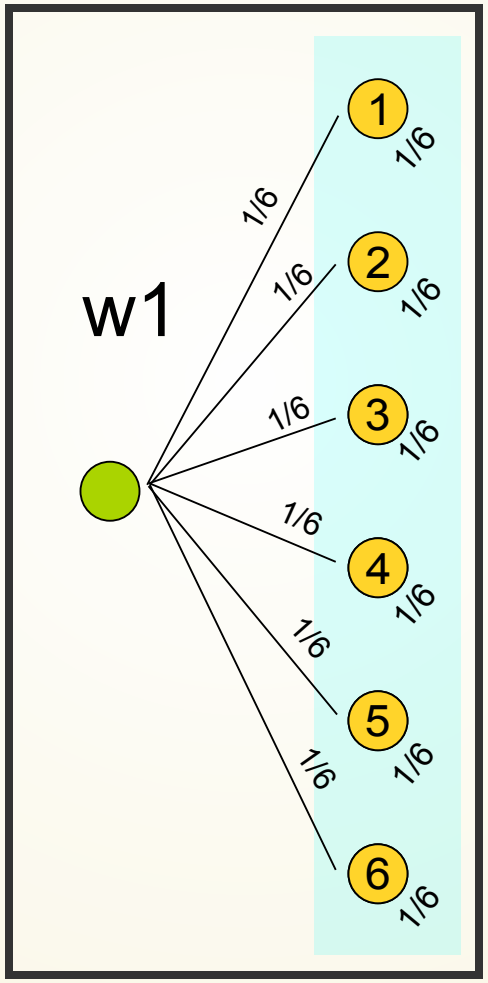
folge einem dynamisch erzeugten *Baumdiagramm* und multipliziere die Wahrscheinlichkeiten.

BEISPIEL

```
let ``Mensch ärgere dich (nicht?)`` =
    vert {
        let! w1 = würfel
        if w1 = 6 then return "ich komm raus" else
        let! w2 = würfel
        if w2 = 6 then return "ich komm raus" else
        let! w3 = würfel
        if w3 = 6 then return "ich komm raus" else
        return "grrr"
    }
```

```
val ( Mensch ärgere dich (nicht?) ) : Verteilung<string> =
  [("grrrr", 0.5787037037); ("ich komm raus", 0.4212962963)]
```

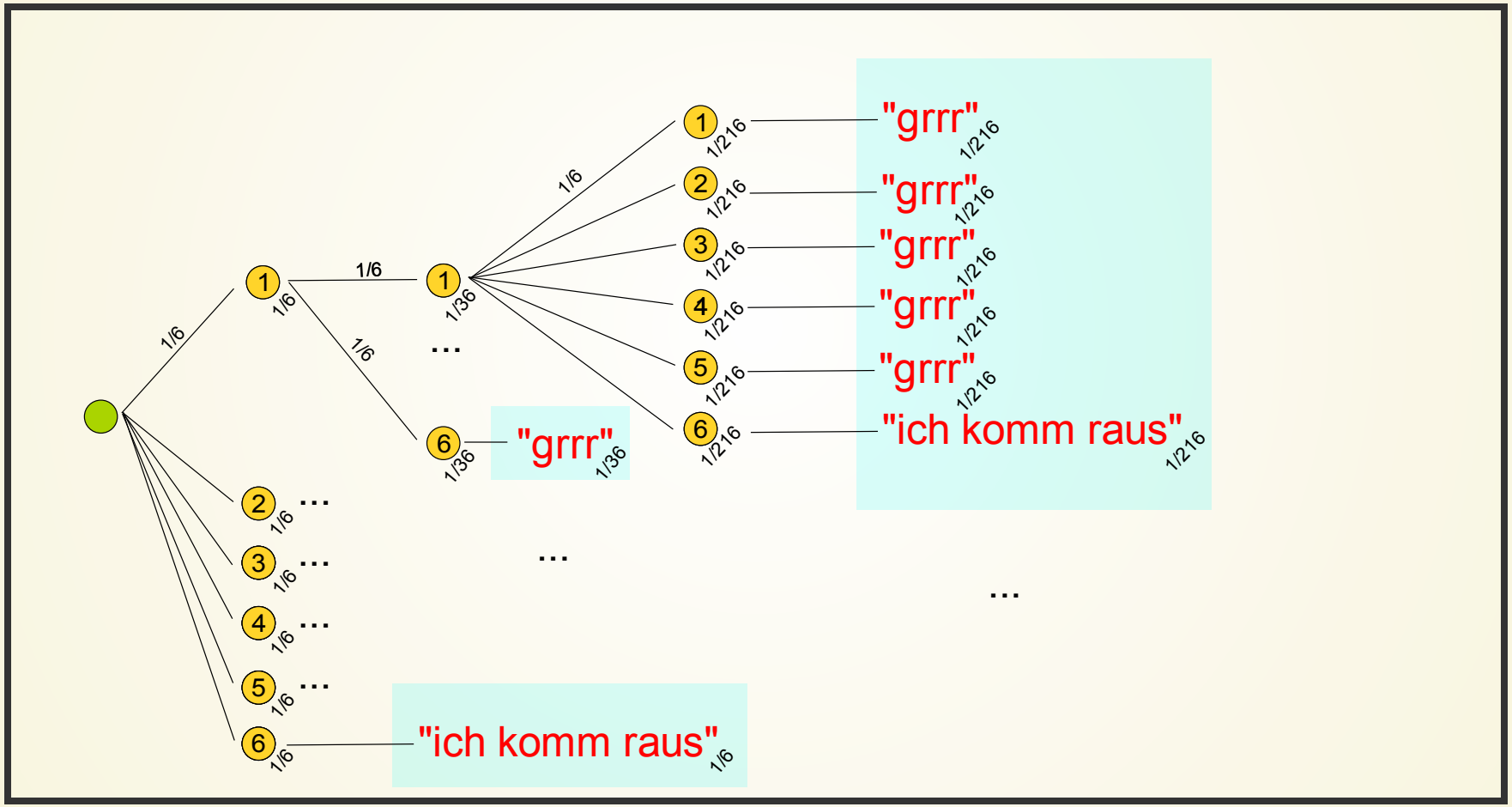
BAUMDIAGRAMM



erster Wurf

BAUMDIAGRAMM

BAUMDIAGRAMM



Ergebnis

BIND

```
let bind (v : Verteilung<'a>) (f : 'a -> Verteilung<'b>)
         : Verteilung<'b> =
    [ for (a,p) in v do
      for (b,p') in f a do
      yield (b, p*p')
    ] |> normalize
```

DETAIL

`normalize` fasst nur die Ergebnise additiv zusammen

```
let normalize (v : Verteilung<'a>) =
    let dict = new System.Collections.Generic.Dictionary<_,_>()
    let get a = if dict.ContainsKey a then dict.[a] else 0.0
    let add (a,p) = dict.[a] <- get a + p
    v |> List.iter add
    dict |> Seq.map (fun kvp -> (kvp.Key, kvp.Value))
         |> List.ofSeq
```

FSCHECK GENERATOR

FSCHECK?

- Testet **Eigenschaften** von Funktionen/Programmen automatisch.
- Eigenschaft: `prop : 'input -> bool`
- FsCheck generiert zufällige Eingaben und testet ob die Eigenschaft jeweils `true` liefert.

  Doku/Info https://fsharp.github.io/FsCheck/QuickStart.html

BEISPIEL

```
let ``zweimal List.rev ist Identität``(xs : int list) =
    List.rev (List.rev xs) = xs

Check.Quick ``zweimal List.rev ist Identität``
> Ok, passed 100 tests.
```

```
let ``List.rev ist Identität``(xs : int list) =
    List.rev xs = xs

Check.Quick ``List.rev ist Identität``
> Falsifiable, after 3 tests (6 shrinks) (StdGen (1233601158,295877135)):
> [1; 0]
```

GENERATOREN

Um die Testfälle zu generieren verwendet FsCheck *Generatoren*

```
type Würfel = internal W of int
let würfel n =
    if n < 1 || n > 6
    then failwith "ungültiger Wert"
    else W n

let würfelGen =
    gen {
        let! n = Gen.choose (1,6)
        return würfel n
    }
```

DEFINITION

vereinfacht: Generator ist eine Aktion, die einen zufälligen Wert liefert

```
type Gen<'a> = Gen of (unit -> 'a)

let sample (Gen g : Gen<'a>) : 'a =
    g()
```

MAKE IT MONADIC

Gesucht: Implementation von

```
bind : Gen<'a> -> ('a -> Gen<'b>) -> Gen<'b>
```

und

```
return : 'a -> Gen<'a>
```

RETURN

```
return : 'a -> Gen<'a> = 'a -> (unit -> 'a)
```

Intuition: zufällige Wahl aus einem Element ...

```
let returnM (a : 'a) : Gen<'a> =
    Gen <| fun () -> a
```

BIND

```
Gen<'a> -> ('a -> Gen<'b>) -> Gen<'b> =
(unit -> 'a) -> ('a -> (unit -> 'b)) -> (unit -> 'b)
```

F#:

```
let bind (m : Gen<'a>) (f : 'a -> Gen<'b>) : Gen<'b> =
    Gen <| fun () -> sample (sample m |> f)
```

ASYNC

MSDN EXAMPLE

```
let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length name
        with
            | ex -> printfn "%s" (ex.Message);
    }
```

SIMPLIFY

verkettete Continuations/Callback

CONTINUATION

```
type Cont<'a>  = 'a -> unit
```

- Nebeneffekt behaftete Aktion
- wird mit dem Ergebnis einer (asynchronen) Berechnung aufgerufen

CONTINUATION MONAD

```
type ContM<'a> = { run : Cont<'a> -> unit }

let mkCont (f : Cont<'a> -> unit) : ContM<'a> =
    { run = f }
```

- gib mir eine `'a` Continution `c`
- ich berechne ein `'a` und rufe `c` damit auf
- kann verzögert/asychron erfolgen

BEISPIEL

```
let delay (span : TimeSpan) : ContM<unit> =
fun f ->
    let timer = new Timer()
    timer.Interval <- int span.TotalMilliseconds
    timer.Tick.Add (fun _ -> timer.Dispose(); f())
    timer.Start()
|> mkCont
```

MAKE IT MONADIC

```
let returnM : 'a -> ContM<'a> = ...

let bind : ContM<'a> -> ('a -> ContM<'b>) -> ContM<'b> =
```

RETURN

```
let returnM (a : 'a) : ContM<'a> =
    mkCont (fun f -> f a)
```

rufe die übergebene Continuation sofort mit `a` auf

BIND

```
let bind : ContM<'a> -> ('a -> ContM<'b>) -> ContM<'b> =
```

```
=  (('a -> unit) -> unit)
-> ('a -> (('b -> unit) -> unit))
-> (('b -> unit) -> unit)
```

LET THE TYPES GUIDE YOU LUKE

BIND

```
let bind (f : 'a -> ContM<'b>) (m : ContM<'a>) : ContM<'b> =
  fun (c : Cont<'b>) ->
      m.run <| fun a -> (f a).run c
  |> mkCont
```

wenn eine Continuation `c` übergeben wird:

- berechne erst den Effekt von `m`
- nutze dessen Ergebnis um mit `f` ein `Cont<'b>` zu bekommen
- berechne diese und gib das Ergebnis an `c` weiter

EXAMPLE

```
let runWith (f : 'a -> 'b) (m : ContM<'a>) =
    m.run (f >> ignore)

let verzögertesHallo() =
    cont {
       do! delay <| TimeSpan.FromSeconds 2.0
       return "Hello, World!"
    } |> runWith MessageBox.Show
```

STATE

MOTIVATION

Berechnungen mit Zustand:

```
f :: zustand -> a -> (b, zustand)
g :: zustand -> b -> (c, zustand)
```

Komposition:

```
let (b, z1) = f z0 a
let (c, _)  = g z1 b
c
```

unschön - geht das mit Monaden besser?

TO MUCH FREEDOM

```
f :: zustand -> a -> (b, zustand)
```

- **Bisher:** immer ein generischer Parameter im Monaden
- **jetzt:** drei: `zustand`, `a` und `b`

`zustand` *ändert* sich nicht - aber was ist mit den anderen?

FUNCTIONAL TRICKSTER

ein wenig umformen:

```
f :: a -> (zustand -> (b, zustand))
```

hier ist ein Monade versteckt

Hint: `putStrLn :: String -> IO ()`

AND THE CANDIATE IS

Haskell:

```
data State s a = MkState { runState :: s -> (a, s) }
```

F#:

```
type State<'s,'a> = { runState : 's -> ('a*'s) }
let mkState f = { runState = f }
```

MAKE IT MONADIC

```
return :: a -> State s a
(>>=)  :: State s a -> (a -> State s b) -> State s b
```

RETURN

Haskell:

```haskell
return :: a -> State s a
return a =  MkState $ \s -> (a, s)
```

F#

```fsharp
let return (a : 'a) : State<'s, 'a> =
  mkState <| fun s -> (a, s)
```

reiche den Zustand weiter und gib den Wert zurück

BIND

Haskell

```
(>>=)  :: State s a -> (a -> State s b) -> State s b
State f >>= g =
    MkState $ \s -> let (a,s') = f s
                    in runState (g a) s'
```

F#

```
let bind (g : 'a -> State<'s,'b>) (f : State<'s,'a>) : State<'s,'b> =
    fun s ->
        let (a,s') = f.runState s
        (g a).runState s'
    |> mkState
```

KOMPOSITION

Problem war ja:

```
f :: a -> (State s b)
g :: b -> (State s c)
```

gesucht Operator `op` mit

```
f `op` g :: a -> (State s c)
```

IMPLEMENTATION

Haskell:

```haskell
op :: (a -> State s b) -> (b -> State s c) -> (a -> State s c)
op f g = \a -> f a >>= g
```

F#:

```fsharp
let op f g =
    fun a -> bind (f a) g
```

ÜBRIGENS...

HLint:

```
Error: Use >=>
Found:
  \ a -> f a >>= g
Why not:
  f Control.Monad.>=> g
```

das ist die *Kleisli* Komposition für Monaden ...

die ist in Haskell für alle Monaden definiert:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \x -> f x >>= g
```

in .NET ist das so leider nicht möglich

§ GESETZE §

MONADE SOLLTE ERFÜLLEN:

1. **Links-Identität:** `return a >>= f === f a`
2. **Rechts-Identität:** `m >>= return === m`
3. **Assoziativität:** `(m >>= f) >>= g === m >>= (\x -> f x >>= g)`

WARUM?

rein *mathematisch* sind diese Gesetze unbedingt nötig um sich Monade zu nennen.

siehe etwa: Category Theory & Programming

praktisch normale Programmiererintuition

RECHTS-IDENTITÄT

```
m >>= return === m
```

als Liste

```
[ y | x <- xs; y <- [x] ] === xs
```

LINQ / C#

```
return
   from x in xs
   from y in new[]{x}
   select y;
```

sollte natürlich das Gleiche wie einfach

```
return xs;
```

sein.

LINKS-IDENTITÄT

```
return a >>= f === f a
```

LINQ / C#

```
return
   from x in new []{a}
   from y in f x
   select y;
```

sollte natürlich das Gleiche wie einfach

```
return f(a);
```

sein.

ASSOZIATIVITÄT (LINKE SEITE)

```
(m >>= f) >>= g === m >>= (\x -> f x >>= g)
```

LINQ / C#

```
var zw1 =
   from x in xs
   from y in f(x)
   select y;
return
   from y in zw1
   from z in g(y)
   select y;
```

ASSOZIATIVITÄT (RECHTE SEITE)

```
(m >>= f) >>= g === m >>= (\x -> f x >>= g)
```

LINQ / C#

```
Func<X,IEnumerable<Z>> zw1 =
  x => from y in f(x)
       from z in g(y)
       return z;
return
  from x in xs
  from z in zw1(x)
  select z;
```

ASSOZIATIVITÄT

oder einfach

LINQ / C#

```
return
  from x in xs
  from y in f(x)
  from z in g(y)
  select z;
```

ASSOZIATIVITÄT KLEISLI

und mit *Kleisli*

```
(f >=> g) >=> h === f >=> (g >=> h)
```

sieht es sogar wie **Assoziativität** aus!