Developer Week 2016

EINFÜHRUNG IN F#

Carsten König

FRAGEN

sehr Willkommen

WAS IST ÜBERHAUPT FUNKTIONALE PROGRAMMIERUNG?

PROGRAMMIEREN MIT FUNKTIONEN

WAS SIND FUNKTIONEN?

MATHEMATISCHE FUNKTIONEN

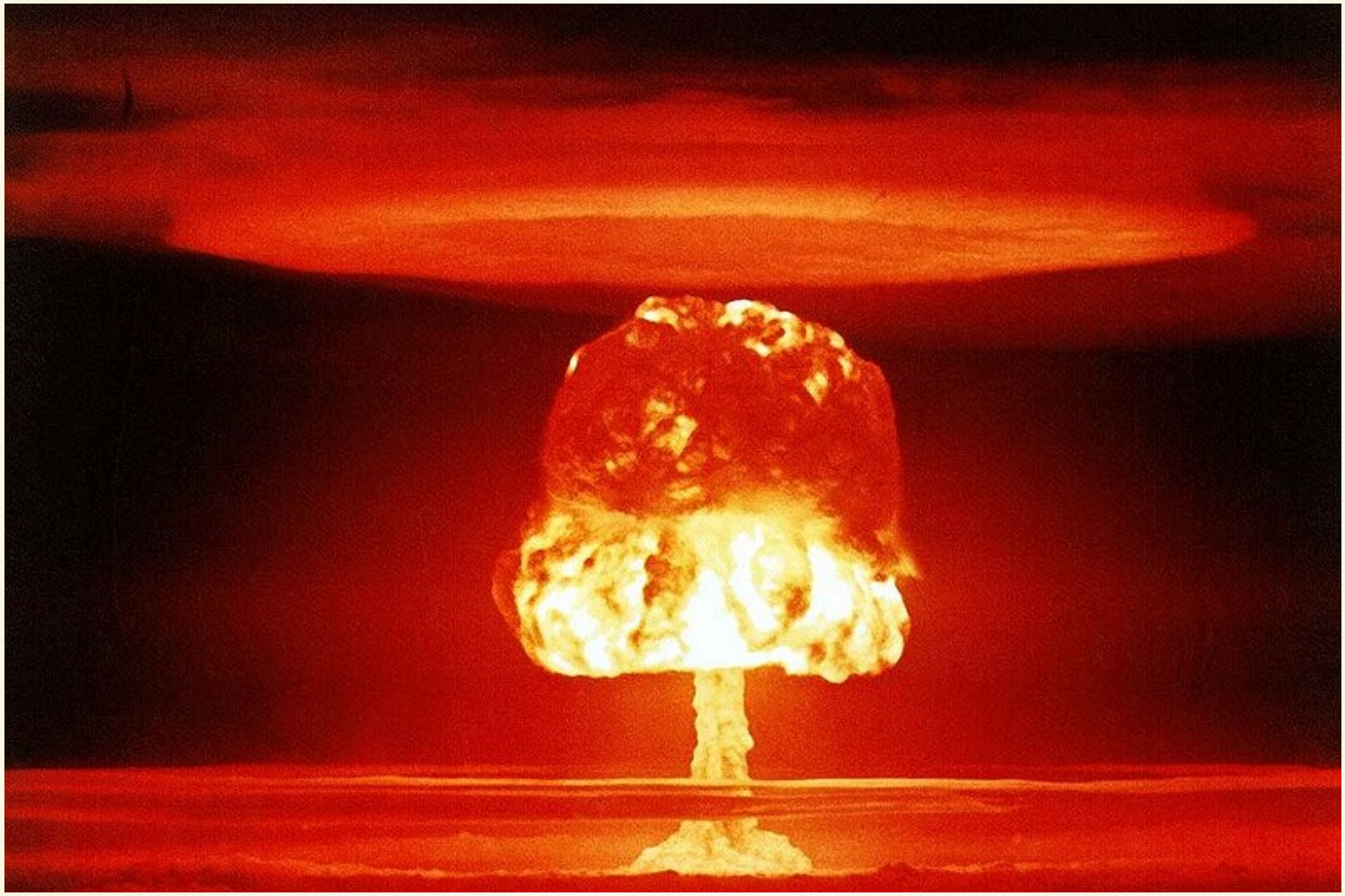
$$f : x \mapsto sinx$$

WESENTLICH

zu **jeder** Eingabe x liefert f **immer** genau **eine** Ausgabe $f(x)$

und bitte keine **Seiteneffekte**

ALSO NICHT ...



```
1: void Print(string text)
2: {
3:     Console.WriteLine(text);
4: }
```

```
1: void Print(string text)
2: {
3:     Console.WriteLine(text);
4: }
```

Seiteneffekt

```
1: DateTime Uhrzeit()
2: {
3:     return DateTime.Now;
4: }
```

```
1: DateTime Uhrzeit()
2: {
3:     return DateTime.Now;
4: }
```

keine Funktion + Seiteneffekt

```
1: int _globalInt = 0;
2:
3: void Incr ()
4: {
5:     _globalInt++;
6: }
```

```
1: int _globalInt = 0;
2:
3: void Incr ()
4: {
5:     _globalInt++;
6: }
```

verändert Daten (Seiteneffekt)

deswegen

UNVERÄNDERBARE DATENSTRUKTUREN / TYPEN

WARUM?

Referenzielle Transparenz

ein **Ausdruck** kann einfach durch seinen **Wert** ersetzen werden

VORTEILE

- (automatische) Optimierungen
- Memoization
- lazy evaluation
- einfacher parallelisierbar
- ...

F#

FUNKTIONEN

DEMO
FUNKTIONEN

```
1: let sag was zu =
2:     was + " " + zu
```

- Argumente getrennt durch Leerzeichen
- Blöcke durch einrückungen

```
> sag "Hallo" "Carsten"
val it : string = "Hallo Carsten"
```

- Anwendung ebenfalls durch Leerzeichen

ACHTUNG

Funktionsapplikation bindet am stärksten

```
1: let plus a b = a + b
2:
3: plus 4 3*4 // <- 28 nicht 16!
```

weil

`plus 4 3*4 = (plus 4 3)*4`

```
1: plus 4 (3*4) // 16
```

TYP-ANNOTATIONEN

hier nötig

```
1: let schreiWas (was : string) zu =
2:     was.ToUpper() + " " + zu
```

```
1: let schreiWas (was : string) (zu : string) : string  =
2:     was.ToUpper() + " " + zu
```

PARTIELLE APPLIKATION

```
1: let sagHalloZu = sag "Hallo"
```

Anwendung

```
> sagHalloZu "Carsten
val it : string = "Hallo Carsten"
```

PIPE-OPERATOR

```
> "Carsten" |> sagHalloZu
val it : string = "Hallo Carsten"
```

LISTEN

DEMO

LISTEN

ANLEGEN

```
1: let zahlen = [1;2;3;4]
2:
3: let mehrZahlen = [1..10]
```

ANNEINANDERFÜGEN

```
1: let zusammen = zahlen @ mehrZahlen
```

LIST-MODUL

enthält viele nützliche Funkionen - *entdecken* mit Intellisense lohnt sich

```
1: List.sum zahlen
2:
3: List.map (fun z -> z*2) zahlen
4:
5: mehrZahlen
6: |> List.filter (fun z -> z%2=0)
7: |> List.map (fun z -> z*2)
8: |> List.sum
```

`::` UND `[]`

- `[]` ist die leere Liste
- `1::[2;3] = [1;2;3]`
- tatsächlich `[1;2;3] = 1::2::3::[]`

PATTERN-MATCHING

```
1: let (kopf::rest) = [1;2;3;4]
2: // -> kopf = 1
3: // -> rest = [2;3;4]
```

MIT MATCH

```
1: let istLeer (xs : int list) =
2:     match xs with
3:     | [] -> "leer"
4:     | (kopf::rest) -> "nicht-leer - fängt mit " + string kopf + " an"
```

KATA
COINCHANGE

Aufgabe: Ein gegebener Betrag `betrag` (in Euro-Cent) soll in Euro-Münzen gewechselt werden.

Dabei soll die Anzahl der zurückgegebenen Münzen minimal sein (`betrag * 1ct`).

BEISPIEL

`237ct` werden in `200ct + 20ct + 10ct + 5ct + 2ct` gewechselt

IDEE

Gib die größte Münze zurück die noch in den Restbetrag passt.

STRATEGIE

- Münzen absteigend vorsortiert
- rekursiver Ansatz

DEMO
COINCHANGE

CODE

```
 1: type Münze = int
 2:
 3: let Euros : Münze list =
 4:     [ 200; 100; 50; 20; 10; 5; 2; 1 ]
 5:
 6: let wechsle betrag =
 7:     let rec wechsle' restBetrag münzen =
 8:         if restBetrag <= 0 then [] else
 9:         match münzen with
10:         | (münze::münzen') when münze <= restBetrag ->
11:             münze :: wechsle' (restBetrag-münze) münzen'
12:         | (_::münzen') -> wechsle' restBetrag münzen'
13:         | [] -> failwith "sollte nicht passieren"
14:     wechsle' betrag Euros
```

Beispiel

```
> wechsle 237;;
val it : int list = [200; 20; 10; 5; 2]
```

DISCLAIMER

die Lösung ist **nicht** die:

- *effizienteste*
- *kürzeste*
- *typischste*

BEISPIEL

```
> wechsle 100000000;;
Stack overflow
```

WIEDERHOLTES ABZIEHEN -> TEILEN MIT REST UND FOLD

```
 1: let wechsle betrag =
 2:     let reduziere (restBetrag, acc) münze =
 3:         let n = restBetrag / münze
 4:         let r = restBetrag % münze
 5:         (restBetrag - n * münze, (n,münze)::acc)
 6:     let flatten xs =
 7:         List.collect (fun (n,m) -> List.replicate n m) xs
 8:     List.fold reduziere (betrag, []) Münzen
 9:     |> snd
10:     |> List.rev
11:     |> flatten
```

KATA

TENNIS

Aufgabe: Punkte zählen beim Tennis

ZIEL

benutze Typen um
ungültige Zustände
nicht-darstellbar
zu machen.

DISCRIMINATED UNION

DISCRIMINATED UNION

deutsch: *Unterscheidungs-Union*

Manchmal auch:

- sum type
- disjoint union
- algebraic data type

DEFINITION

```
1: type Punkt =
2:     | Null
3:     | Fünfzehn
4:     | Dreißig
```

```
1: type NächsterPunkt =
2:     | Punkt of Punkt
3:     | Vierzig
```

KONSTRUKTION

```
1: let nächster30 = Punkt Dreißig
```

PATTERN MATCHING

```
1: let (Punkt p) = nächster30
2:
3: > val p : Punkt = Dreißig
```

BEISPIEL

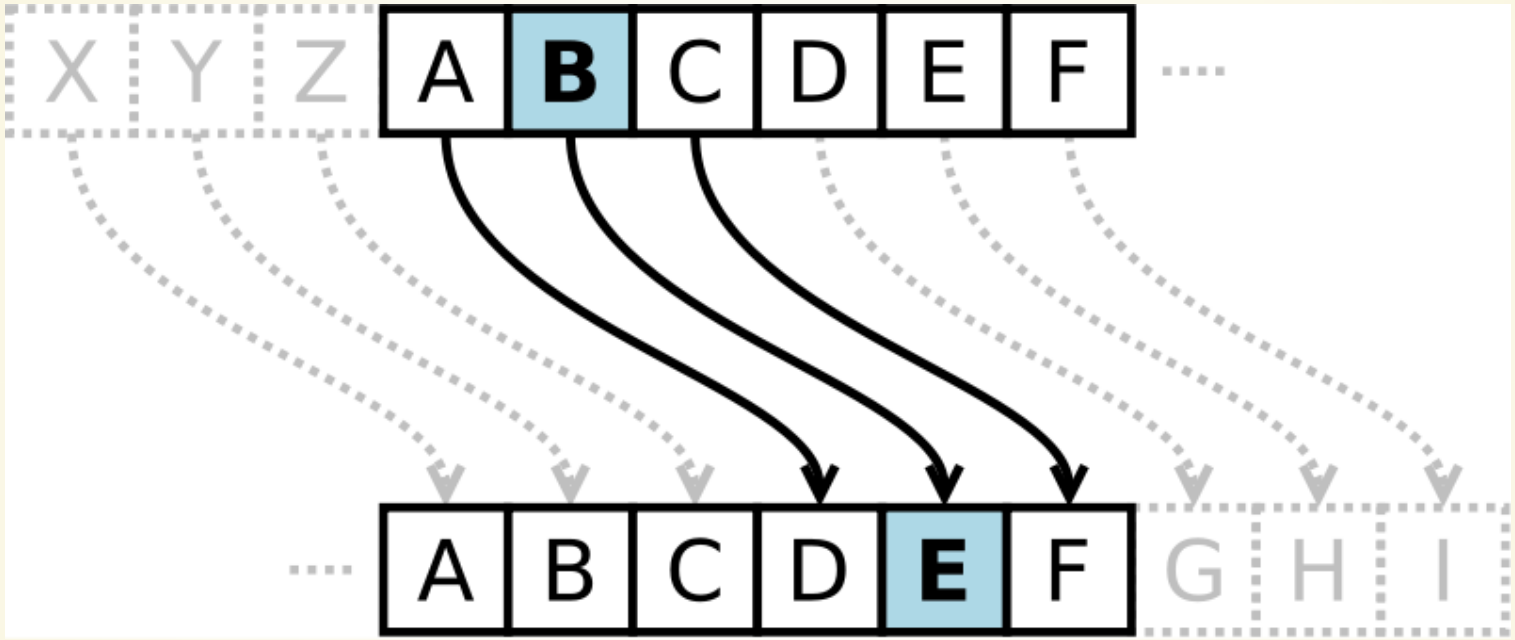
```
1: let nächsterPunkt punkt =
2:     match punkt with
3:     | Null     -> Punkt Fünfzehn
4:     | Fünfzehn -> Punkt Dreißig
5:     | Dreißig  -> Vierzig
```

DEMO

TENNIS

CAESAR UND DAS PHANTOM

Caesar-Verschlüsselung



PHANTOMTYPEN

```
1: type Nachricht<'label> =
2:     private | Nachricht of string
3:     override this.ToString() =
4:         match this with
5:         | Nachricht s -> s
6:
7: type Klartext   = private | Klartext
8: type Geheimtext = private | Geheimtext
```

NACHRICHT NORMALISIEREN

im Bereich `['A'..'Z']`

```
 1: let nachricht : string -> Nachricht<Klartext> =
 2:     let gültigeZeichen = System.Collections.Generic.HashSet ['A'..'Z']
 3:     fun s ->
 4:         if System.String.IsNullOrEmpty s
 5:             then ""
 6:             else s.ToUpper()
 7:         |> Seq.filter gültigeZeichen.Contains
 8:         |> Array.ofSeq
 9:         |> System.String
10:         |> Nachricht
```

EINZELNEN BUCHSTABEN VERSCHIEBEN

```
 1: type Schlüssel = int
 2:
 3: let private nmap f (Nachricht s) : Nachricht<'label> =
 4:     Seq.map f s
 5:     |> Array.ofSeq
 6:     |> System.String
 7:     |> Nachricht
 8:
 9: let private translate (key : Schlüssel) (c : char) =
10:     let mod' n d =
11:         let m = n % d
12:         if m < 0 then m+d else m
13:     char <| (mod' (int c - int 'A' + key) 26) + int 'A'
```

VER- UND ENTSCHLÜSSELN

```
1: let verschlüsseln (key : Schlüssel )
2:                   (n : Nachricht<Klartext>)
3:                   : Nachricht<Geheimtext> =
4:     nmap (translate key) n
5:
6: let entschlüsseln (key : Schlüssel)
7:                   (n : Nachricht<Geheimtext>)
8:                   : Nachricht<Klartext> =
9:     nmap (translate -key) n
```

```
> let msg = nachricht "Hello World";;
val msg : Nachricht<Caesar.Klartext> = HELLOWORLD

> let enc = verschlüsseln 4 msg;;
val enc : Nachricht<Geheimtext> = LIPPSASVPH

> Caesar.entschlüsseln 4 enc;;
val it : Caesar.Nachricht<Caesar.Klartext> = HELLOWORLD

> Caesar.entschlüsseln 4 msg;;
.. error FS0001: Type mismatch. Expecting a
   Nachricht<Geheimtext>
but given a
   Nachricht<Klartext>
```

UNITS OF MEASURE

WÄHRUNGSRECHNER

```
1: [<Measure>] type EUR
2: [<Measure>] type USD
3:
4: let wechselKurs = 1.13m<USD/EUR>
5:
6: let eur2usd (eur : decimal<EUR>) : decimal<USD> =
7:     eur * wechselKurs
```

erkennt Fehler:

```
1: let usd2eur (usd : decimal<USD>) : decimal<EUR> =
2:      usd * wechselKurs
```

The unit of measure 'EUR' does not match the unit of measure 'USD ^ 2/EUR'

ETWAS PHYSIK

```
1: [<Measure>] type km
2: [<Measure>] type m
3: [<Measure>] type h
4: [<Measure>] type s
5:
6: let kmh2ms (kmh : float<km/h>) : float<m/s> =
7:     kmh * 1000.0<m/km> / 3600.0<s/h>
8:
9: let g  = 9.81<m/s^2>
```

Senkrechter Wurf

```
1: let hMax (v0 : float<m/s>) : float<m> =
2:     v0*v0 / (2.0 * g)
```

Beispiel

```
1: hMax (kmh2ms 80.<km/h>)
2: > val it : float<m> = 25.16958005
```

ERWÄHNTES PAPER

- **P. Wadler** *Theorems for free*

VIELEN DANK!

Carsten König

- Twitter: CarstenK_dev
- Email: Carsten@gettingsharper.de
- Web: gettingsharper.de