# FP-Gilde Session 1

09.03.2022

# Expressions vs statements *

# Expressions

Alles was sich wie eine pure Funktion verhällt

- pure Funktion
- + - : x
- || &&
- ?:

# Statements

- if, else, while, for, do
- Geben keinen Wert zurück
- Können Seiteneffekte enthalten
- Nicht unterstützt durch pure FP-Sprachen

# Tailcall-Optimierung (I) *

- Ergebnis des letzten Funktionsaufrufs ist auch Ergebnis der gesamten Berechnung
- Kompilierter Code nutzt *jmp* anstelle von *call*
- Rekursionen ohne Stackoverflow

# Tailcall-Optimierung (II)

```haskell
recSum :: [Int] -> Int
recSum [] = 0
recSum (x:xs) = x + (recSum xs)
```

```haskell
recSum [1..100000000]
-- *** Exception: stack overflow
```

# Tailcall-Optimierung (III)

```haskell
optRecSum :: [Int] -> Int
optRecSum list = rec 0 list
    where
        rec acc [] = acc
        rec acc (x:xs) = rec (acc + x) xs
```

```haskell
optRecSum [1..100000000]
-- 5000000050000000
```

# Currying (I)

Wenn nicht alle Parameter geliefert werden, gibt ein Funktionsaufruf eine neue Funktion zurück. Die Funktion nimmt die restlichen Parameter an.

Genauer gibt es nur Funktionen mit einem Parameter. Jeder Aufruf erzeugt eine neue Funktion mit einem Parameter, bis alle Parameter vorliegen. Dann wird die letzte Funktion ausgeführt.

```
const curriedAdd3 = x => y => z => x + y + z

curriedAdd3(1) // neuen Funktion erwartet den Parameter y
curriedAdd3(1)(2) // neuen Funktion erwartet den Parameter z
curriedAdd3(1)(2)(3) // letzter notwendiger Parameter is vohanden;
                     // Addition wird ausgeführt
```

# Currying (II)

```
doubleMap = map (* 2)
doubleMap [5..8]
-- [10,12,14,16]
```

```
const map = fn => list => list.map(fn)
const multiply = x => y => x * y
const doubleMap = map(multiply(2))
```

# Pipe und compose

- Hilft bei der Verkettung von Funktionen
- Der vorherige Wert wird zum Parameter des nächsten Funktionsaufrufs

# Pipe

- Anfangswert wird weitergereicht. Der letzte Rückgabewert ist das Ergebnis

```
"Elixir rocks" |> String.upcase() |> String.split()
# ["ELIXIR", "ROCKS"]
```

```
words $ map toUpper $ "Haskell rocks"
```

# Compose

- Anfangswert ist unbekannt. Die Verkettung erzeugt eine neue Funktion, die den Anfangswert als Parameter hat.

```
let divisibleBy divisor nun = num % divisor = 0
let doubleEven = List.filter (divisibleBy 2) >> List.map ((*) 2)
```

```
divisibleBy divisor num = num `mod` divisor == 0
doubleEven = map (* 2) . filter (divisibleBy 2)
```

# Chaining (I)

```
var car =
    new Car(Color.Red)
        .Drive(10000)
        .Paint(Color.White);
```

```
let car =
    Car.create Red
        |> Car.drive 10000
        |> Car.paint White
```

# Chaining (II)

- Nur mit Objekten möglich
- Erweiterung braucht eine Änderung in der Klasse
- Chaining kann nicht immer weitergeführt werden

```
var car =
    new Car(Color.Red)
        .Drive(10000)
        .Paint(Color.White);
var owner = new Owner(car);
```

```
let owner =
    Car.create Red
        |> Car.drive 10000
        |> Car.paint White
        |> Owner.create //!!
```

# Monade (I)

- Ursprung in der Kategorientheorie
- Initial in Haskell eingeführt, um Seiteneffekte zu ermöglichen*
- Es gibt aber auch noch andere Nutzungsmöglichkeiten **

# Monade (II)

- Eine Monade kapselt Werte
- Eine Monade implemetiert map und bind

# map und bind

```
map :: (a -> b) -> M a -> M b
bind :: (a -> M b) -> M a -> M b
```

# Maybe

- Auch bekannt als Option oder Optional
- Kapselt optionale oder fehlende Werte
- `data Maybe a = Just a | Nothing`
- Ersetzt null/nil/undefined

# Maybe

```
repository
|> OwnerRepository.getOwner id // Option<Owner>
|> Option.bind (Owner.getCar) // Option<Car>
|> Option.map (Car.paint White) // Option<Car>
```

# Optional

```
repository
    .getOwner(id)
    .flatMap(Owner::getCar)
    .map(car -> car.paint(Color.WHITE))
```

```
Owner owner;

if(repository.getOwner(id).isPresent()) {
    owner = repository.getOwner(id).get()

    Car car;

    if(owner.getCar().isPresent()) {
        car = owner.getCar().get();
        return Optional.of(car.paint(Color.WHITE));
    }

    return Optional.empty();
}

return Optional.empty();
```

# Maybe (Vorteil)

Wenn aus

```
foo :: x -> y
```

```
foo :: x -> Maybe y
```

wird,
müssen aufrufende Funktionen sich anpassen

# Maybe (Kritik) *

Wenn aus

```
foo :: x -> y
foo :: Maybe x -> y
```

oder aus

```
foo :: x -> Maybe y
foo :: x -> y
```

wird,

müssen aufrufende Funktionen sich anpassen

# Either

- Auch bekannt als Result
- Unterscheidet zwischen zwei möglichen Ergebnissen
- IdR. Fehler oder valides Ergebnis
- `data Either l r = Left l | Right r`

# Either (Beispiel) *

```
type Request = {name:string; email:string}

let validateName request =
    if input.name = "" then Failure "Name must not be blank"
    else if input.name.Length > 50
        then Failure "Name must not be longer than 50 chars"
    else Success request

let validateEmail request =
    if input.email = ""
        then Failure "Email must not be blank"
    else Success request
```

```
let validateRequest = validateName >> bind validateEmail

{name=""; email=""} |> validateRequest
// Failure "Name must not be blank"

{name="Alice"; email=""} |> validateRequest
// Failure "Email must not be blank"

{name="Alice"; email="good"} |> validateRequest
// Success {name = "Alice"; email = "good";}
```

```
createMessage validationResult =
    match validationResult with
        Failure msg -> "Failed validation: " + msg
        Success -> "Valid request"
```

# Either (Nachteile)

- Kein Stacktrace
  - Daher nur für erwartete / bekannte Fehler einsetzen

# Quellen


1. https://fsharpforfunandprofit.com/posts/expressions-vs-statements/
2. https://eklitzke.org/how-tail-call-optimization-works
3. Hudak, Paul, et al. "A history of Haskell: being lazy with class." Proceedings of the third ACM SIGPLAN conference on History of programming languages. 2007.
4. Wadler, Philip. "Monads for functional programming." International School on Advanced Functional Programming (1995): 24-52.

5. https://youtu.be/YR5WdGrpoug
6. https://fsharpforfunandprofit.com/posts/recipe-part2/
7. https://fsharpforfunandprofit.com/posts/against-railway-oriented-programming/