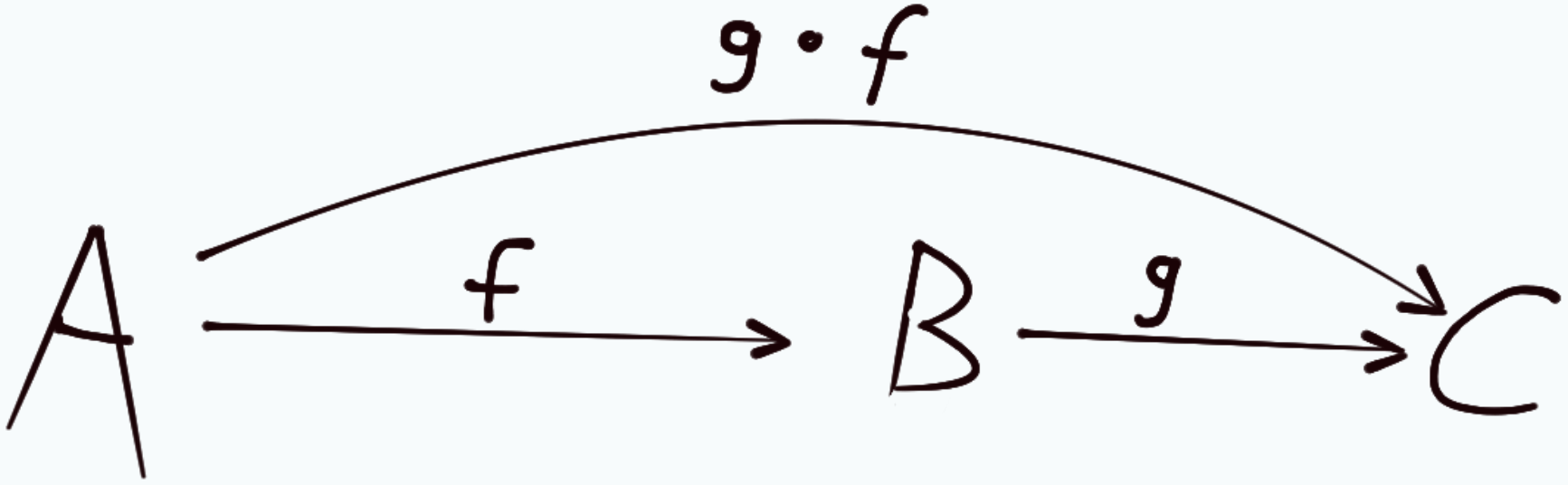
FUNKTIONALE PARSER IN F#

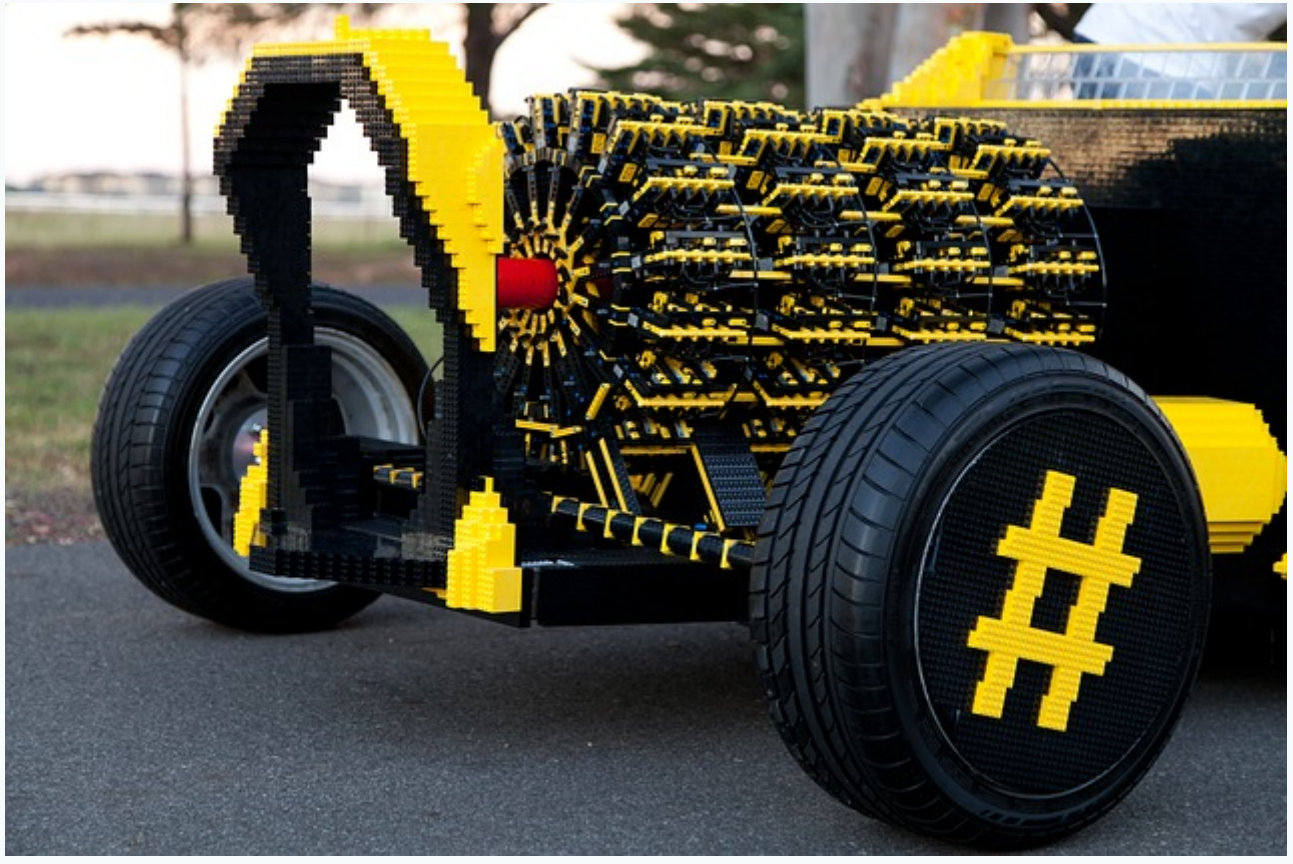
Carsten König

EINLEITUNG

FUNKTIONEN UND KOMPOSITION



DIE LEGO-IDEE

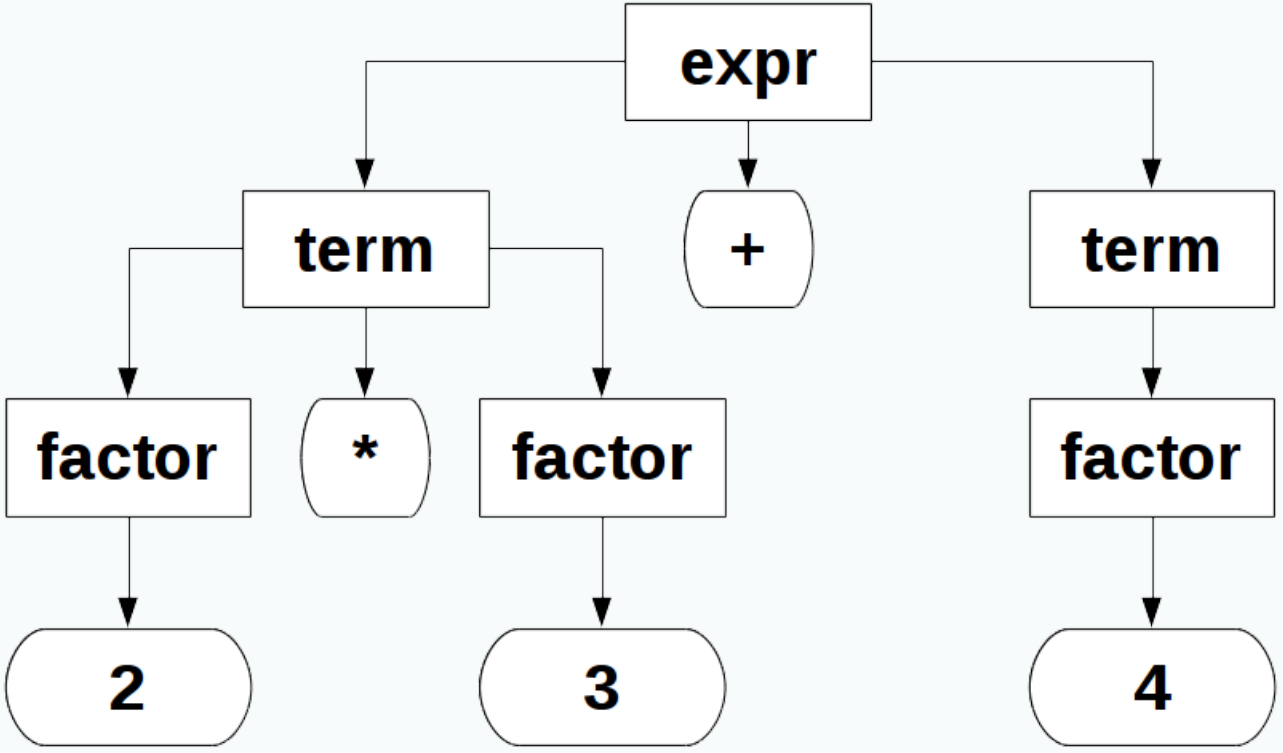


PARSER

WAS IST EIN PARSER?

ein **Parser** versucht eine *Eingabe* in eine für die Weiterverarbeitung geeignete *Ausgabe* umzuwandeln.

2*3+4

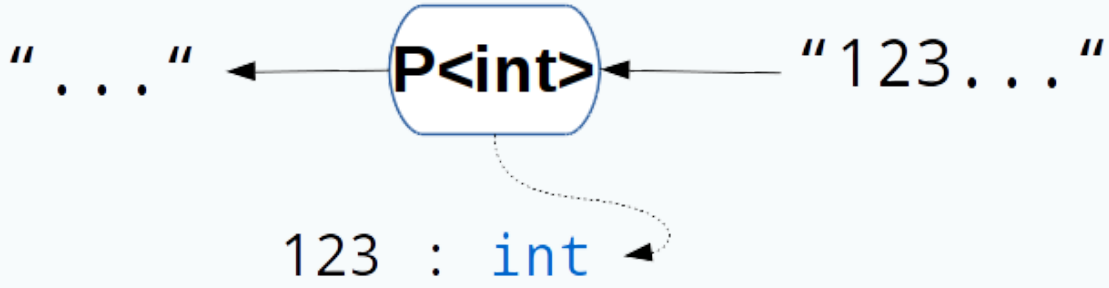


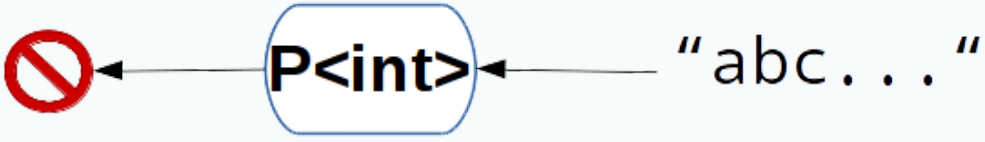
Syntaxbaum

DAZU

- *Parser* als **Daten** repräsentieren
- *Kombinatoren* als **Funktionen** zwischen diesen Daten

IDEE





DEFINITION Parser

Input-String → Output-Result

```
type Parser<'a> = string -> Result<'a>
```

Parser-WERTE

```
fun input -> ...
```

`Result`

eine von zwei Möglichkeiten:

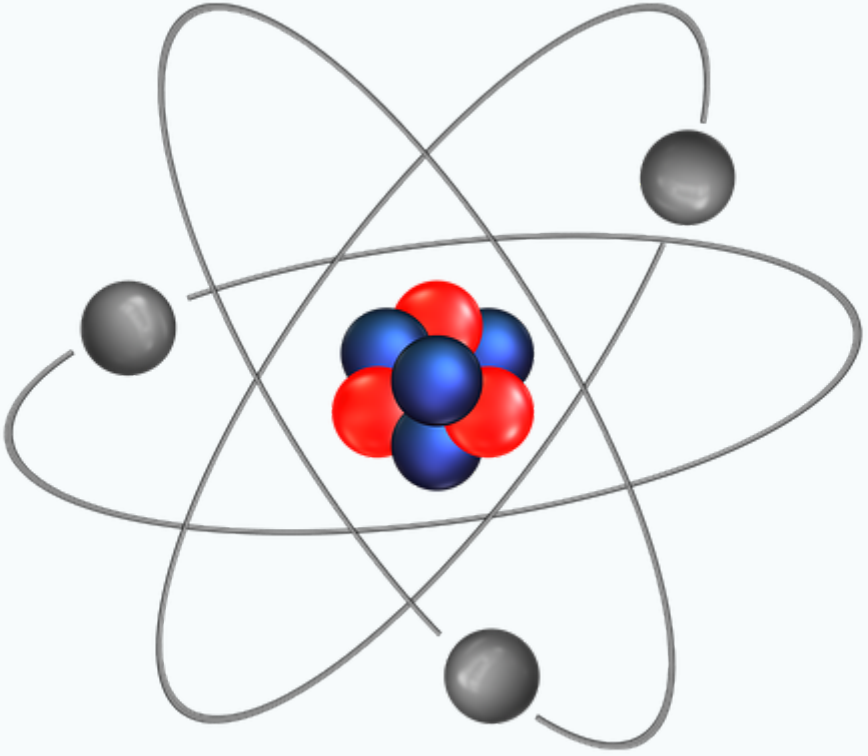
- Parser konnte Eingabe **nicht** erkennen
- Parser hat einen **Teil** der Eingabe erkannt und einen **Ergebniswert** berechnet

ALGEBRAISCHER DATENTYP

```
type Result<'a> =
    | Success of value:'a * remainingInput:string
    | Failure
```

PATTERN MATCHING

```
match result with
| Failure -> ...
| Success (value, remaining) -> ...
```



Run

```
let run (p : Parser<'a>) (input : string) : Result<'a> =
    p input
```

Fail

```
let fail () : Parser<'a> =
    fun _ -> Failure
```

schlägt immer fehl

Succeed

```
let succeed (withValue : 'a) : Parser<'a> =
    fun input -> Success (withValue, input)
```

gibt immer `withValue` zurück

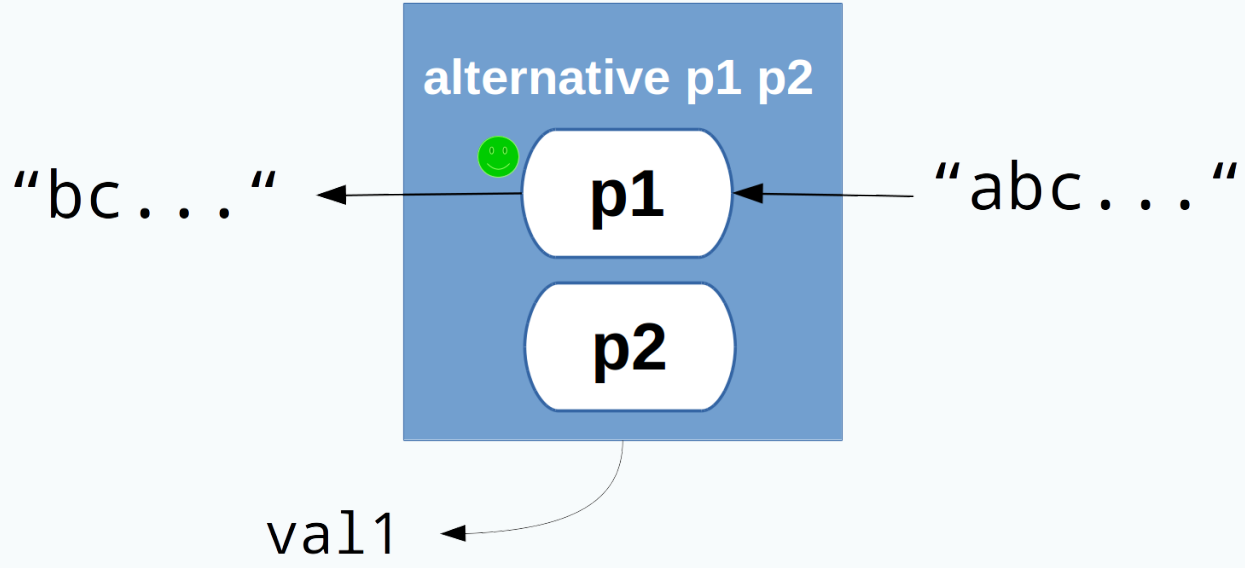
Character PARSER

entscheidet über ein Prädikat ob das erste Zeichen in der Eingabe erkannt wird

```
let character (isValid : char -> bool) : Parser<char> =
    fun input ->
        if input.Length = 0 || not (isValid input.[0])
        then Failure
        else Success (input.[0], input.[1..])
```

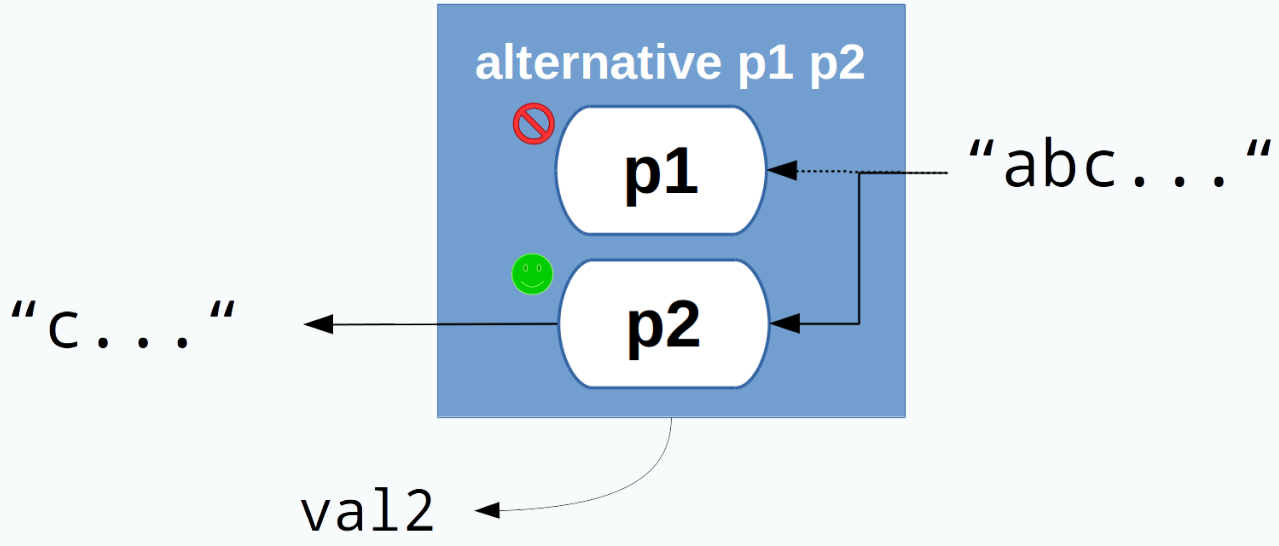
EINFACHE KOMBINATOREN

alternative



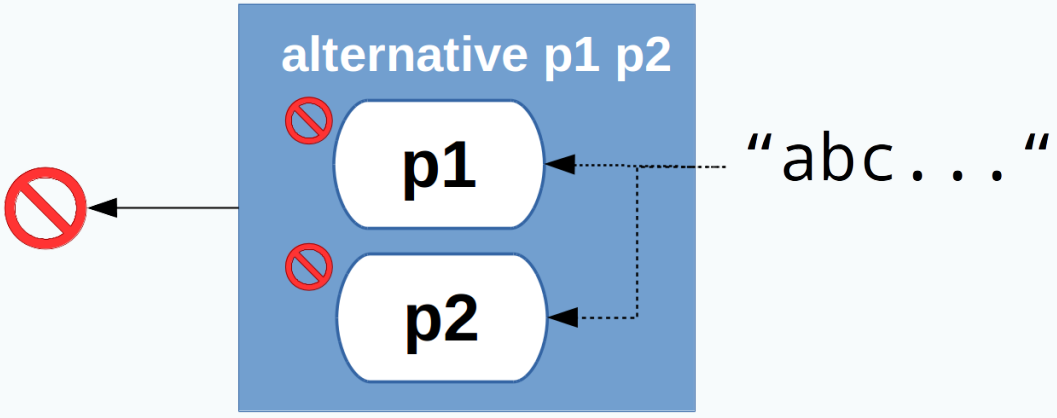
Parser 1 ok

alternative



Parser 1 fail, Parser 2 ok

alternative



beide fail

alternative

```
let alternative (either : Parser<'a>) (orElse : Parser<'a>) =
    fun input ->
        match run either input with
        | Success _ as res -> res
        | Failure          -> run orElse input
```

BEISPIEL

```
let binaryDigit = alternative
    (character (fun c -> c = '0'))
    (character (fun c -> c = '1'))
```

choice

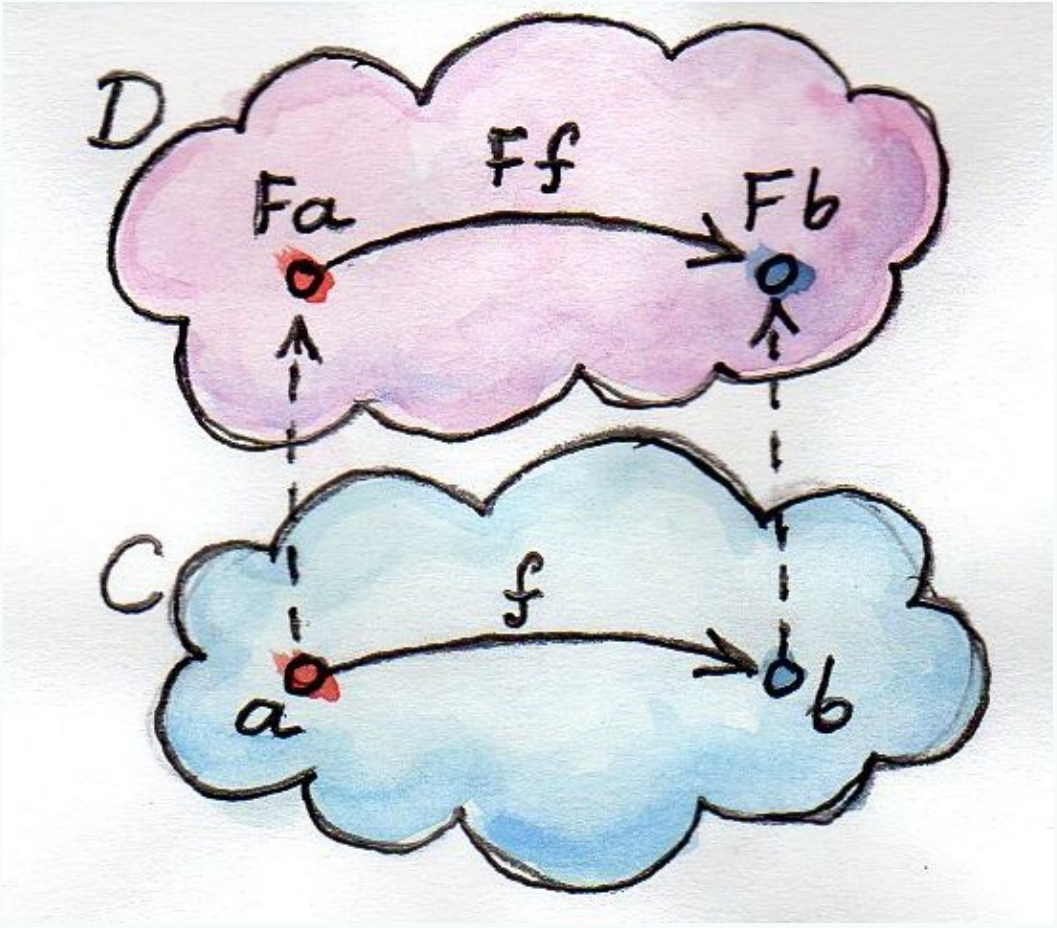
```
let choice (parsers : Parser<'a> seq) =
    Seq.reduce alternative parsers
```

Seq.reduce

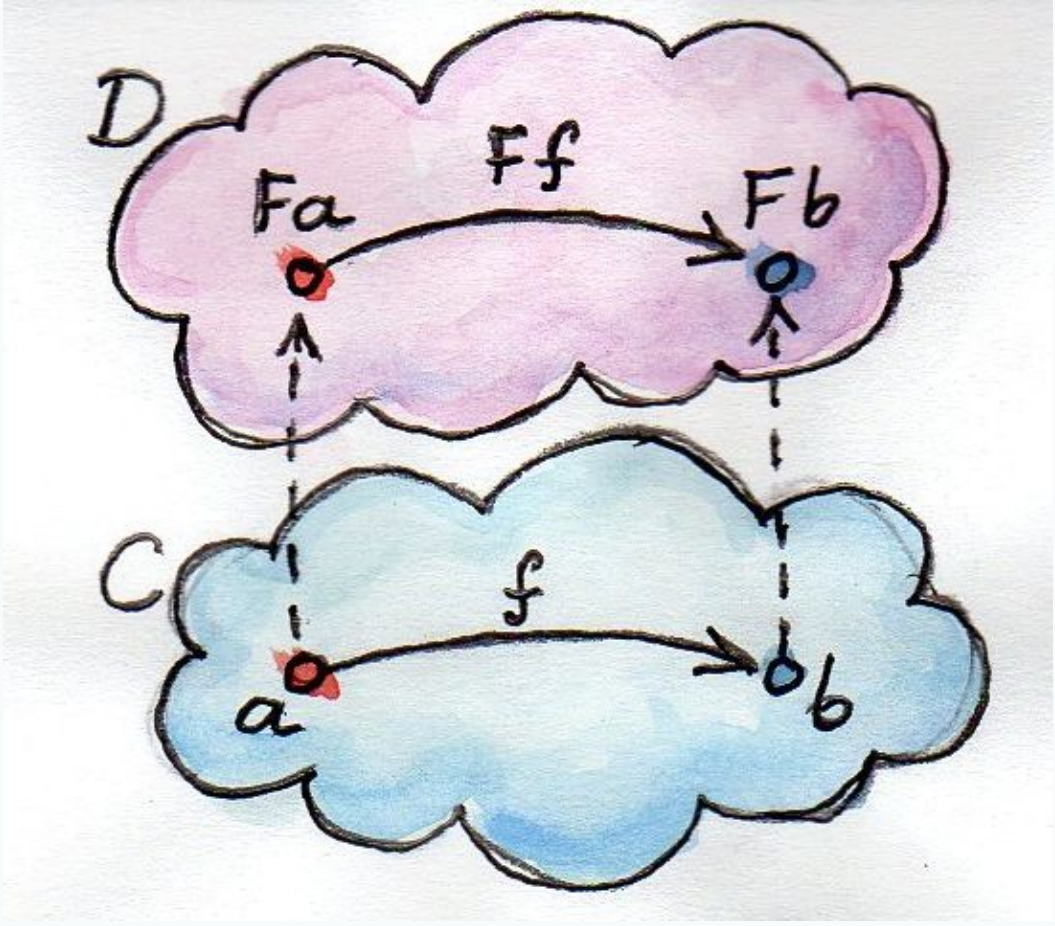
```
Seq.reduce op [x1; x2; x3; x4]
= Seq.reduce op [op x1 x2; x3; x4]
= Seq.reduce op [op (op x1 x2) x3; x4]
= Seq.reduce op [op (op (op x1 x2) x3) x4]
= op (op (op x1 x2) x3) x4
```

FUNKTOR

map : (f: 'a → 'b) → (F<'a> → F<'b>)

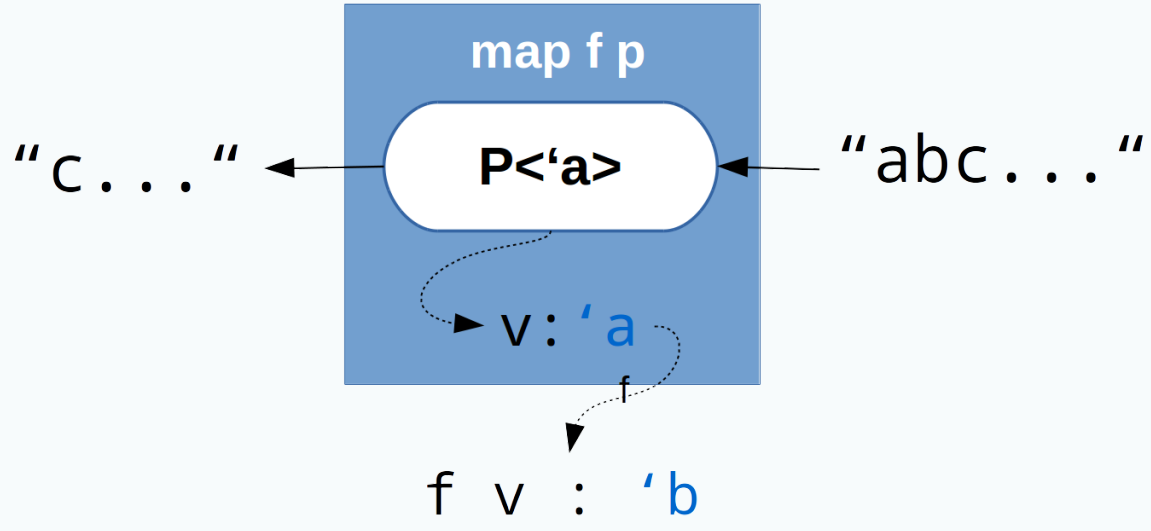


map : (f: 'a → 'b) → (Parser<'a> → Parser<'b>)



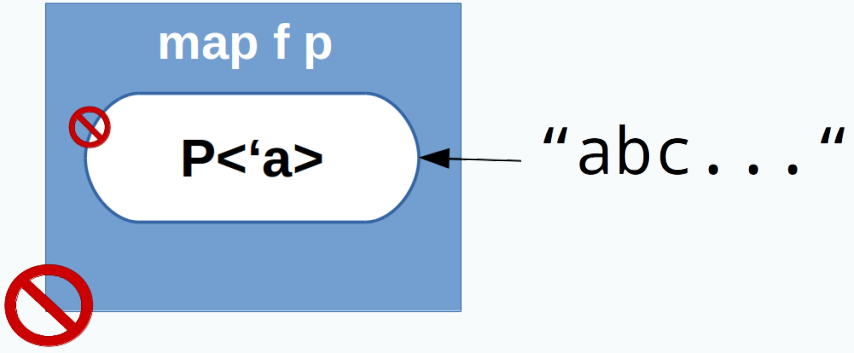
ERFOLG

```
f : 'a → 'b
p : Parser<'a>
```



FEHLSCHLAG

```
f : 'a → 'b
p : Parser<'a>
```



FUNKTOR-MAP

```
let map (f : 'a -> 'b) (p : Parser<'a>) : Parser<'b> =
    fun input ->
        match run p input with
        | Success (value, remaining) ->
            Success (f value, remaining)
        | Failure ->
            Failure
```

Result.Map

```
let map (f : 'a -> 'b) (r : Result<'a>) : Result<'b> =
    match r with
    | Success (value, remaining) -> Success (f value, remaining)
    | Failure                    -> Failure
```

kombiniert

```fsharp
let map (f : 'a -> 'b) (p : Parser<'a>) : Parser<'b> =
    fun input -> run p input |> Result.map f
```

```fsharp
let map (f : 'a -> 'b) (p : Parser<'a>) : Parser<'b> =
    run p >> Result.map
```

exactChar PARSER

```fsharp
let exactChar (expected : char) : Parser<unit> =
    character (fun c -> c = expected)
    |> map ignore
```

GESETZE

- `map id p ≡ p`
- `map (g << f) p ≡ map g (map f p)`

ANDERE “FUNKTOREN”

- `IEnumerable` / `Select`
- `Task`
- …

APPLICATIVE FUNCTORS

PURE

'a → P<'a>

APPLICATIVE MAP

P<'a → 'b> → P<'a> → P<'b>

pure

```
let pure a = succeed a
```

apMap

```
let apMap (pf : Parser<'a -> 'b>)
          (pa : Parser<'a>) : Parser<'b> =
    fun input ->
        match run pf input with
        | Failure -> Failure
        | Success (f, remainingF) ->
            match run pa remainingF with
            | Failure -> Failure
            | Success (a, remainingA) ->
                Success (f a, remainingA)

let (<*>) pf a = apMap pf a
```

“BUILDER”-PATTERN

BEISPIEL

```
let leftOf (ign : Parser<unit>) (p : Parser<'a>) =
    build (fun a _ -> a)
    |> withParser p
    |> andParser ign
```

```
let build = succeed
let withParser a pf = pf <*> a
let andParser = withParser
```

between

```
let between ( left : Parser<'l>
            , right : Parser<'r>
            )
            ( p : Parser<'a> ) =
    build (fun _ a _ -> a)
    |> withParser left
    |> andParser p
    |> andParser right
```

GESETZE

- `pure id <*> p ≡ p`
- `pure (<<) <*> u <*> v <*> w ≡ u <*> (v <*> w)`
- `pure f <*> pure x ≡ pure (f x)`
- `u <*> pure y ≡ pure (fun f -> f y) <*> u`

MONADEN

andThen

```
P<'a> → ('a → P<'b>) → P<'b>
```

KLEISLI / >=> - OPERATOR

```
('a → P<'b>) → ('b → P<'c>) → ('a → P<'c>)
```

```
let andThen (decide : 'a -> Parser<'b>)
            (p : Parser<'a>) : Parser<'b> =
    fun input ->
        match run p input with
        | Failure -> Failure
        | Success (valueA, remaining) ->
            decide valueA
            |> run remaining

let (>>=) p f = p |> andThen f
```

COMPUTATIONAL EXPRESSIONS

BEISPIEL

```
let rec many (p : Parser<'a>) : Parser<'a seq> =
    alternative (many1 p) (succeed Seq.empty)

and many1 (p : Parser<'a>) : Parser<'a seq> =
    parser {
        let! first = p
        let! rest = many p
        return seq { yield first; yield! rest }
    }
```

BEISPIEL `chainl1`

Ziel: `3 + 4 + 5 = (3 + 4) + 5 = 7 + 5 = 12`

```
<expr> ::= <operand>
         | <operand> operator <expr>
```

ChainLeft1

3 + 4 + 5 = (3 + 4) + 5 = 7 + 5 = 12

```
let chainLeft1 (operator : Parser<'a -> 'a -> 'a>)
               (operand : Parser<'a>) =
    let rec rest accum =
      choice
        [ parser {
            let! op = operator
            let! value = operand
            return! rest (op accum value)
          }
        , succeed accum
        ]
    operand |> andThen rest
```

BUILDER

```
type ParserBuilder() =
    member __.Bind(p,f) = p |> andThen f
    member __.Return(x) = succeed x
    member __.ReturnFrom x = x

let parser = ParserBuilder ()
```

GESETZE

- `pure a >>= f ≡ f a`
- `p >>= pure ≡ p`
- `(p >>= f) >>= g ≡ p >>= (fun x -> f x >>= g)`

BEISPIEL *RECHNER*

DEMO

```
$ dotnet run

> 5+5*5
30

> (5+5)*5
50

> 2-2-2
-2

> Hallo
parse error

>
```

DIE GRAMMATIK

```
<expr>   ::= <term>    | <term> ("+"|"-") <expr>
<term>   ::= <factor>  | <factor>  ("*"|"/") <term>
<factor> ::= zahl      | "(" <expr> ")"

zahl   = [0|1|..|9]+
```

LEERZEICHEN IGNORIEREN

```
let spaceP : Parser<unit> =
    Parser.character Char.IsWhiteSpace
    |> Parser.many
    |> Parser.map ignore

let trim p = Parser.leftOf spaceP p
```

ZAHL PARSEN

```
let zahlP : Parser<double> =
    Parser.character Char.IsDigit
    |> Parser.many1
    |> Parser.map (Seq.toArray >> String)
    |> Parser.tryMap System.Double.TryParse
    |> trim
```

OPERATOREN

```
let operatorP symbol op =
    Parser.exactChar symbol
    |> trim
    |> Parser.map (fun _ -> op)

let addOps : Parser<double -> double -> double> =
    Parser.choice [
        operatorP '+' (+)
        operatorP '-' (-)
    ]

let mulOps : Parser<double -> double -> double> =
    ...
```

PARSER ZWISCHEN KLAMMERN

```
let bracedP (p : Parser<'a>) : Parser<'a> =
    Parser.between
        ( Parser.exactChar '(' |> trim
        , Parser.exactChar ')' |> trim
        )
        p
```

EXPRESSION

```
let expressionP : Parser<double> =
    let (exprRef, setExpr) = Parser.createForwardRef()
    let factorP = Parser.alternative (bracedP exprRef) zahlP
    let termP   = factorP |> Parser.chainLeft1 mulOps
    let expP    = termP   |> Parser.chainLeft1 addOps
    setExpr expP
    exprRef
```

LINKS

REFERENZEN

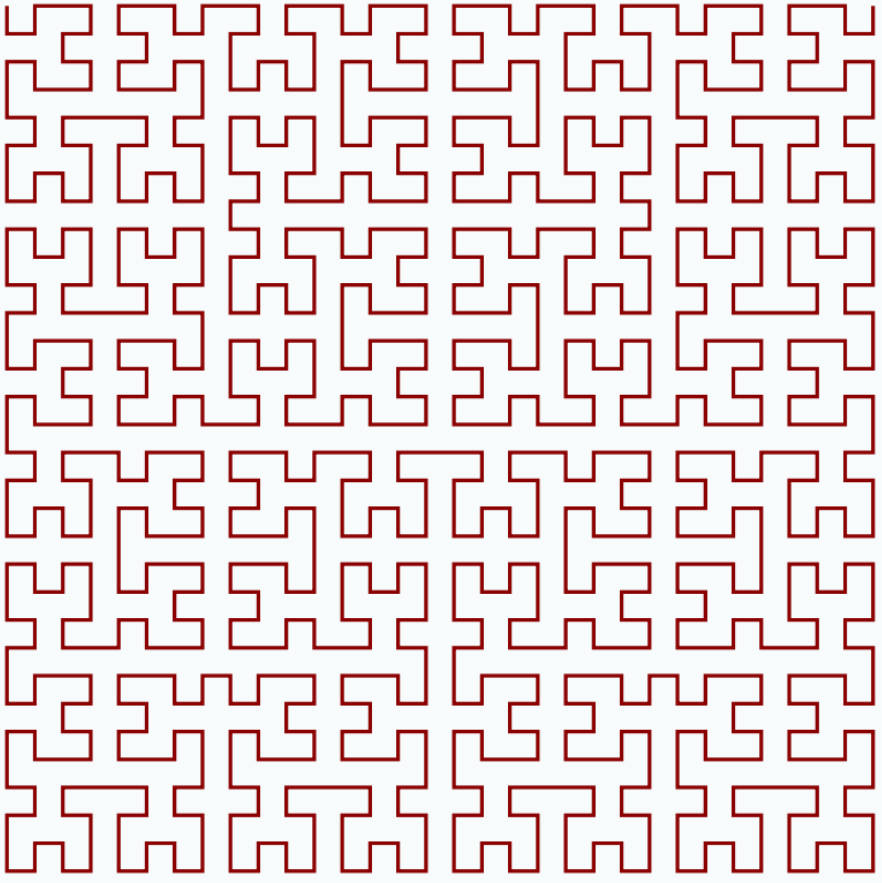
- G. Hutton, E. Meijer Monadic Parsing in Haskell
- G. Hutton, E. Meijer Monadic Parser Combinators

BIBLIOTHEKEN

- Sprache
- FParsec
- Liste mit anderen Implementationen

ANDERE BEISPIELE

DIAGRAMS



Hilbert Curve

DIAGRAMS

```
hilbert 0 = mempty
hilbert n = hilbert' (n-1) # reflectY <> vrule 1
         <> hilbert  (n-1) <> hrule 1
         <> hilbert  (n-1) <> vrule (-1)
         <> hilbert' (n-1) # reflectX
  where
    hilbert' m = hilbert m # rotateBy (1/4)
```

ELM - JSON DECODERS

```
type alias Info =
  { height : Float
  , age : Int
  }

infoDecoder : Decoder Info
infoDecoder =
  map2 Info
    (field "height" float)
    (field "age" int)
```

ANDERE

- Form / Validation
- Folds / Projections (Eventsourcing)

FRAGEN / ANTWORTEN?

VIELEN DANK

- **Slides/Demo**
  github.com/CarstenKoenig/DWX2019_Parser
- **Twitter** @CarstenK_Dev