# FUNKTIONALE PARSER IN C#

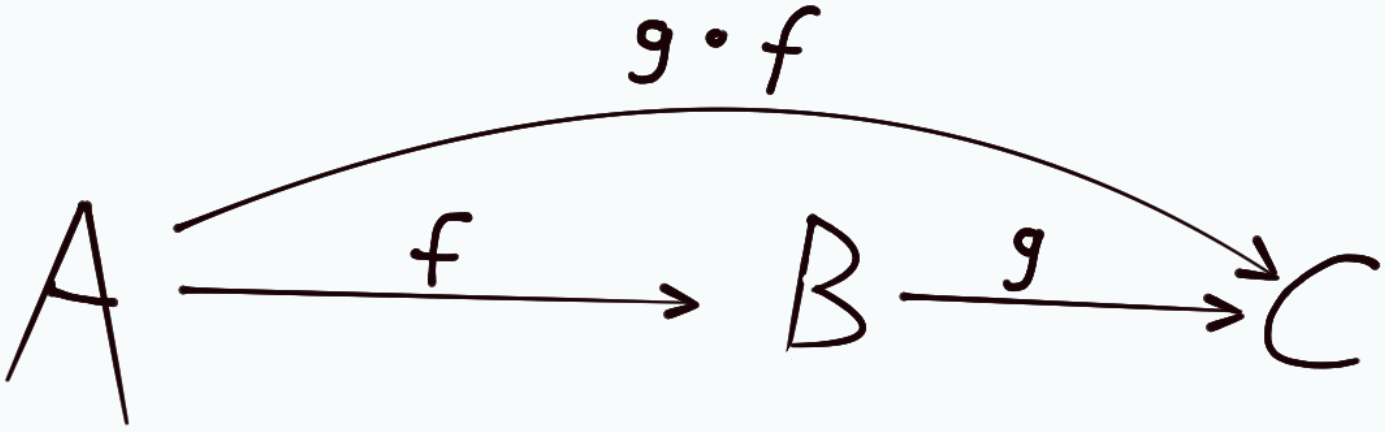
Carsten König

06. April 2018

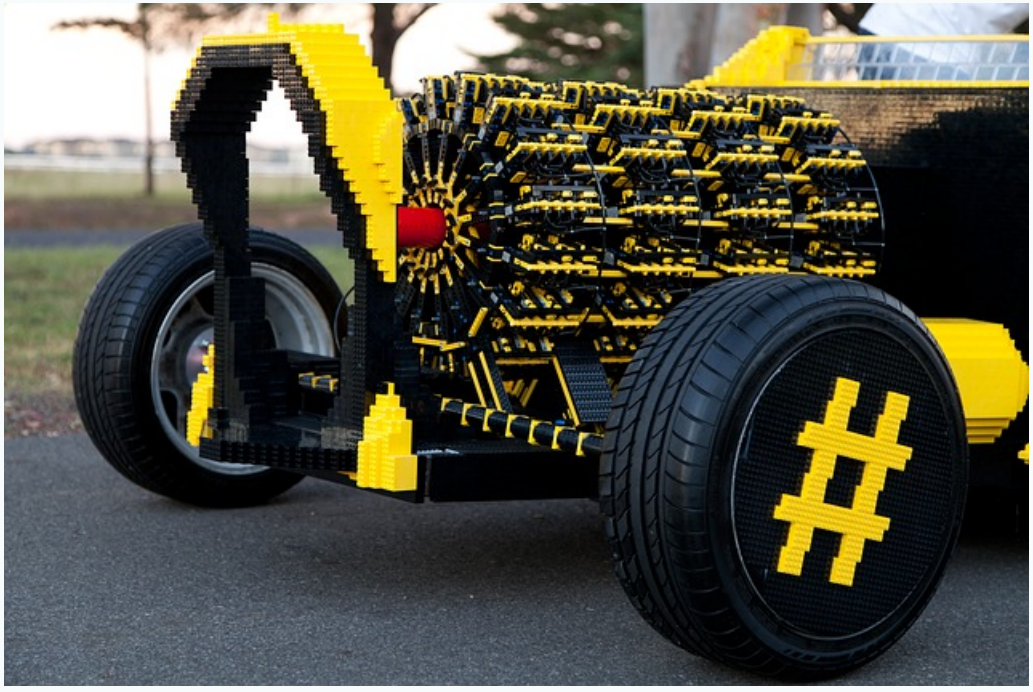
# EINLEITUNG


BAYREUTH
WÜRZBURG
NÜRNBERG
ANSBACH

# FUNKTIONEN UND KOMPOSITION



# DIE LEGO-IDEE

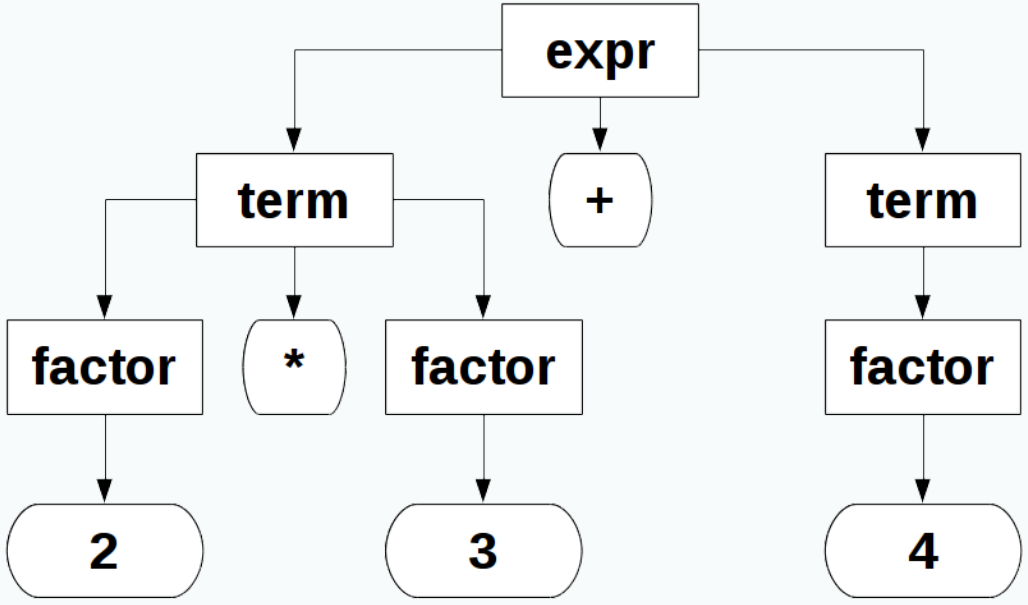


# PARSER

# WAS IST EIN PARSER?

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

# 2*3+4

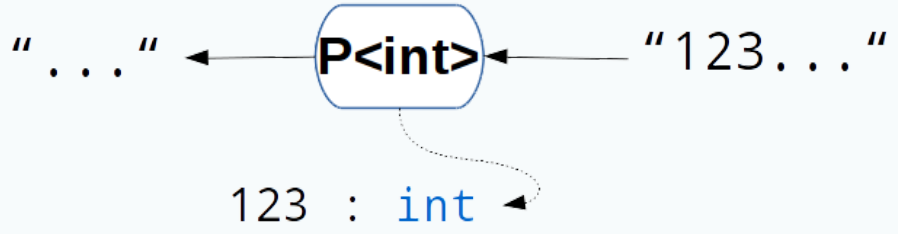


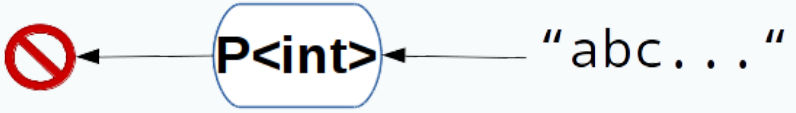

Syntaxbaum

# DAZU

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

# IDEE



# DEFINITION Parser

## Input-String → Output-ParseResult

```
delegate ParseResult<T> Parser<T>(string input)
```

# `ParserResult`

eine von zwei Möglichkeiten:

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

# ALGEBRAISCHER DATENTYP

## F#

```
type ParseResult<'a> =
   | Failure
   | Success of Value:'a * RemainingInput:string
```

# C#

```
abstract class ParseResult<T> { .. }

class FailureResult<T> : ParseResult<T>
{
}

class SuccessResult<T> : ParseResult<T>
{
  T Value { get; }
  string RemainingInput { get; }
}
```

# PATTERN MATCHING

## F#

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

# C# 7

```
switch (result)
{
   case FailureResult<T> fail:
     ...
     break;
   case SuccessResult<T> success:
     ... success.Value ...
     break;
}
```

# ALTERNATIVE *CHURCH-ENCODING*

**Was** machen wir mit den Datentyp?

# BEISPIEL Bool

## wird in *Entscheidungen* / `if` benutzt

```
if (boolValue)
   return thenBranch;
else
   return elseBranch;

// oder

boolValue ? thenBranch : elseBranch;
```

# Church-Encoding

```
delegate T Bool<T>(T thenBranch, T elseBranch);

T True<T>  (T thenBranch, T elseBranch) => thenBranch;
T False<T> (T thenBranch, T elseBranch) => elseBranch;

True ("Hallo", "Welt"); // = "Hallo"
False("Hallo", "Welt"); // = "Welt"
```

# BEISPIEL NAT

```
delegate T Nat<T>(Func<T,T> plus1, T zero);

T Null<T> (Func<T,T> plus1, T zero) => zero;
T Eins<T> (Func<T,T> plus1, T zero) => plus1(zero);
T Zwei<T> (Func<T,T> plus1, T zero) => plus1(plus1(zero));

Zwei (s => "*"+s, ""); // = "**"
Zwei (n => n+1, 0);    // = 2
```

# ParserResult

```
abstract class ParseResult<T>
{
   abstract Tres Match<Tres>(
      Func<T, string, Tres> withSuccess,
      Func<Tres> onError);
}
```

# FAILURE

```
class FailureResult<T> : ParseResult<T>
{
  override Tres Match<Tres> (
     Func<T, string, Tres> withSuccess,
     Func<Tres> onError)
  {
    return onError();
  }
}
```

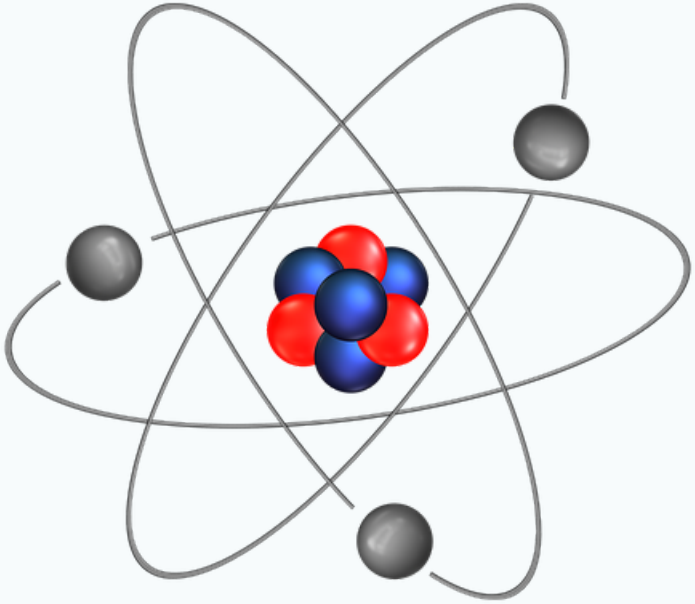
# SUCCESS

```
class SuccessResult<T> : ParseResult<T>
{
  T Value { get; }
  string RemainingInput { get; }

  override Tres Match<Tres>(
     Func<T, string, Tres> withSuccess,
     Func<Tres> onError)
  {
    return withSuccess(Value, RemainingInput);
  }
}
```

# ANWENDUNG

```
abstract class ParseResult<T>
{
  bool IsSuccess =>
     Match((val, rem) => true, () => false);
}
```



# Fail

- schlägt immer fehl

```
Parser<T> Fail<T>()
{
  return input => new FailureResult<T>();
}
```

# Succeed

- immer erfolgreich
- gibt *festen* Wert zurück
- *konsumiert* nichts vom Input

```
Parser<T> Succeed<T>(T withValue)
{
  return input => SuccessResult<T>(withValue, input);
}
```

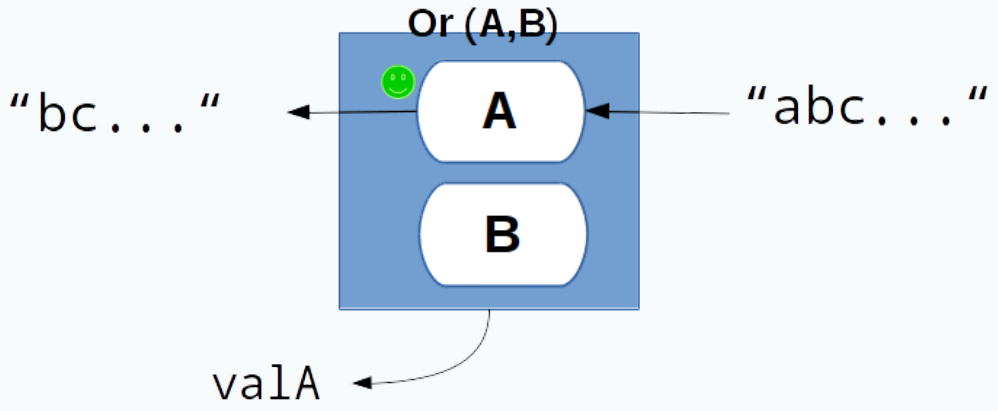
# Char PARSER

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

```
Parser<char> Char(Predicate<char> isValidChar)
{
  return input =>
    input.Length == 0 || !isValidChar(input[0])
    ? FailureResult<char>()
    : SuccessResult<char>(input[0], input.Substring(1));
}
```

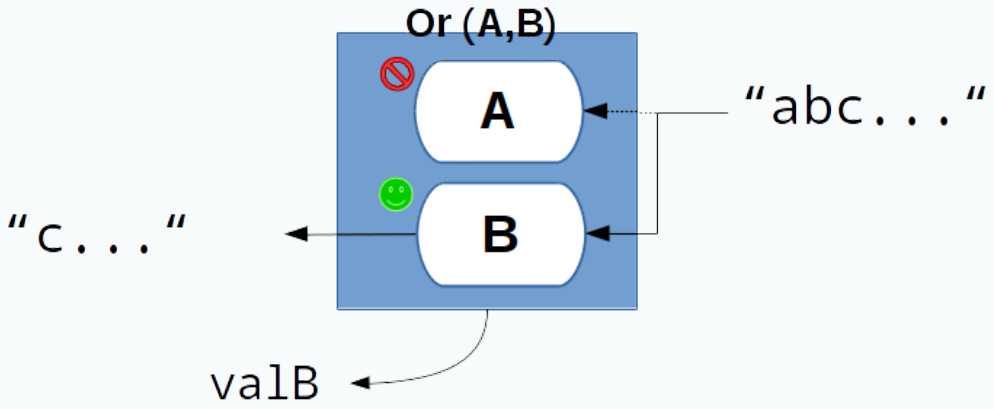
# PARSER KOMBINIEREN

# or



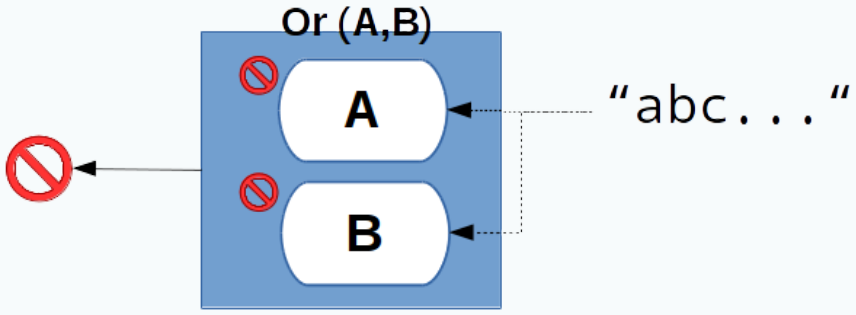

A ok

# or




A fail, B ok

# or




beide fail

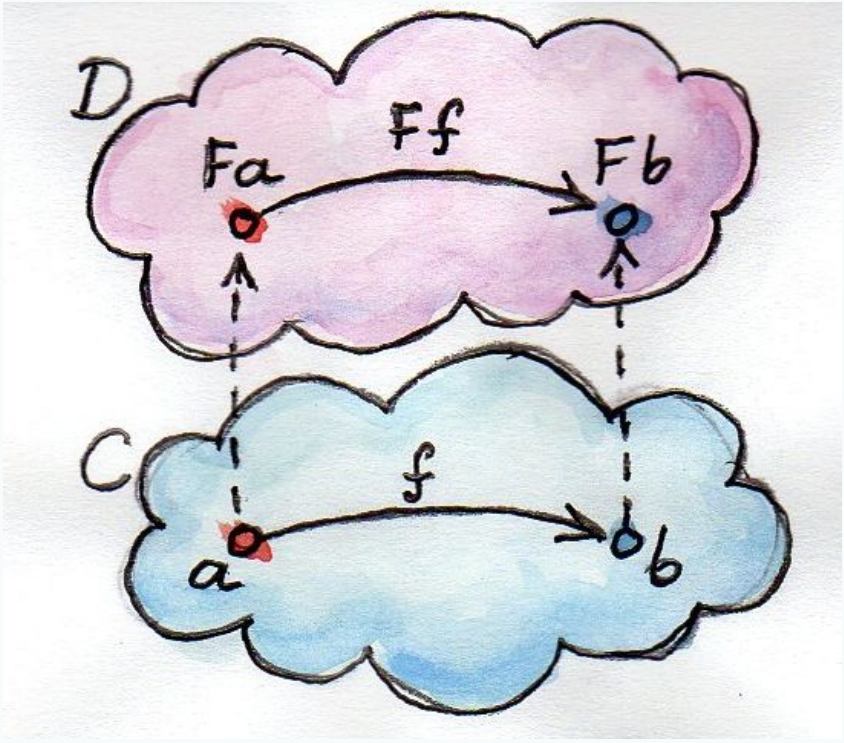
# or

```
Parser<T> Or<T>(this Parser<T> parserA, Parser<T> parserB)
{
  return input =>
    {
      switch (parserA(input))
      {
        case SuccessResult<T> success:
          return success;
        default: // Failed
          return parserB(input);
      }
    };
}
```

# many

```
Parser<IEnumerable<T>> Many<T>(this Parser<T> parser) {
  return input => {
      var results = new List<T>();
      var nextInput = input;
      while (true) {
        switch (parser(nextInput)) {
          case SuccessResult<T> res:
            results.Add(res.Value);
            nextInput = res.RemainingInput;
            break;
          default:
            return SuccessResult<IEnumerable<T>>
               (results, nextInput);
}}};}
```

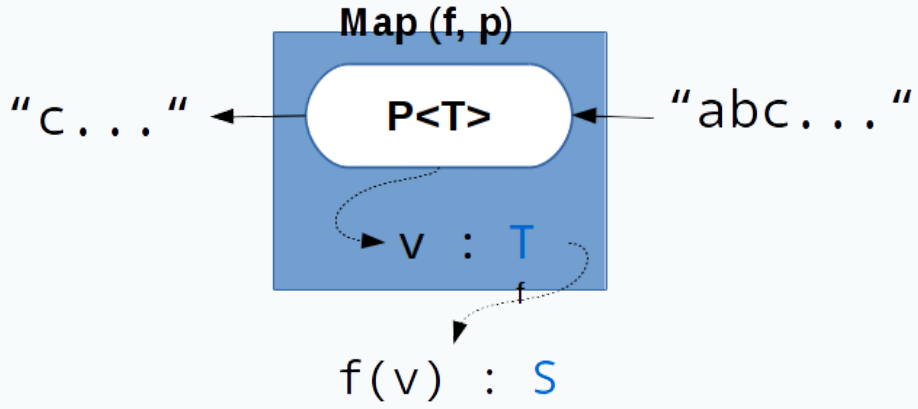
# FUNKTOR




map : (f: A → B) → (F<A> → F<B>)

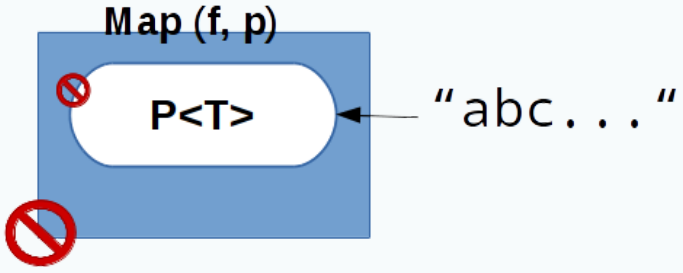
# ERFOLG

```
f : Func<T,S>
p : P<T>
```



# FEHLSCHLAG

```
f : Func<T,S>
p : P<T>
```



# C#

```
Parser<TRes> Map<T, TRes>(
   Parser<T> parser,
   Func<T, TRes> map)
{

  return input =>
     parser(input).Match(
        (valueT, remaining) =>
           new SuccessResult(map(valueT), remaining),
        () => new FailureResult<TRes>());
}
```

# ParseResult.Map

```
ParseResult<Tres> Map<Tres>(Func<T, Tres> map)
{
  return Match(
    (value, remaining) => map(value).Success(remaining),
    () => ParseResult.Fail<Tres>());
}
```

# kombiniert

```
Parser<TRes> Map<T, TRes>(
   this Parser<T> parser,
   Func<T, TRes> map)
{
  return input => parser(input).Map(map);
}
```

# GESETZE

- `p.map(x=>x)` ≡ `p`
- `p.map(x=>g(f(x)))` ≡ `p.map(f).map(g)`

# ANDERE “FUNKTOREN”

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

MONADEN

# andThen

P<A> → (a → P<B>) → P<B>

# andThen

```
Parser<TRes> AndThen<T, TRes>(
   this Parser<T> parser,
   Func<T, Parser<TRes>> getNext)
{
  return input =>
    parser(input).Match(
        (value, remaining) => getNext(value)(remaining),
        () => new FailureResult<TRes>());
}
```

LINQIFY

# BEISPIEL ChainLeft1

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

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

```
Parser<T> ChainLeft1<T>(
   this Parser<T> operandP,
   Parser<Func<T, T, T>> operatorP)
{
  Parser<T> rest(T accum) =>
    (from op in operatorP
      from val in operandP
      from more in rest(op(accum, val))
      select more)
    .Or(Succeed(accum));

  return operandP.AndThen(rest);
}
```

# SELECT

```
Parser<TResult> Select<TSource, TResult>(
   this Parser<TSource> source,
   Func<TSource, TResult> selector)
{
  return source.Map(selector);
}
```

# SELECT MANY

```
Parser<TResult> SelectMany<TSource, TCollection, TResult>(
  this Parser<TSource> source,
  Func<TSource, Parser<TCollection>> collectionSelector,
  Func<TSource, TCollection, TResult> resultSelector)
{
  return source
      .AndThen(val =>
        collectionSelector(val)
        .Map(col => resultSelector(val, col)));
}
```

# BEISPIEL *RECHNER*

# DEMO

```
$ dotnet run
input? 5+5*5
30
input? (5+5)*5
50
input? 2-2-2
-2
input? Hallo
parse error
input?
```

# DIE GRAMMATIK

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

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

# LEERZEICHEN IGNORIEREN

```
var spacesP = Parsers
    .Char(Char.IsWhiteSpace)
    .Many()
    .Map(_ => Unit.Value);
```

# ZAHL PARSEN

```
var zahlP =
  Parsers
    .Char(Char.IsDigit)
    .Many1()
    .Map(ToString)
    .TryMap<string, double>(Double.TryParse)
    .LeftOf(spacesP);
```

# OPERATOREN

```
Parser<Func<double, double, double>>
   OperatorP( char symbol,
              Func<double, double, double> operation )
      => Parsers.Char(symbol).Map(_ => operation);

var strichOperatorP =
  OperatorP('+', Add)
  .Or(OperatorP('-', Subtract))
  .LeftOf(spacesP);

var punktOperator = ...
```

# EXPRESSION

```
var expressionRef = Parsers.CreateForwardRef<double>();

var factorP = expressionRef.Parser
  .Between(Parsers.Char('('), Parsers.Char(')'))
  .Or(zahlP);

var termP = factorP
   .ChainLeft1(punktOperator);

expressionRef.SetRef(
   termP
   .ChainLeft1(strichOperator));
```

# LINKS

# REFERENZEN

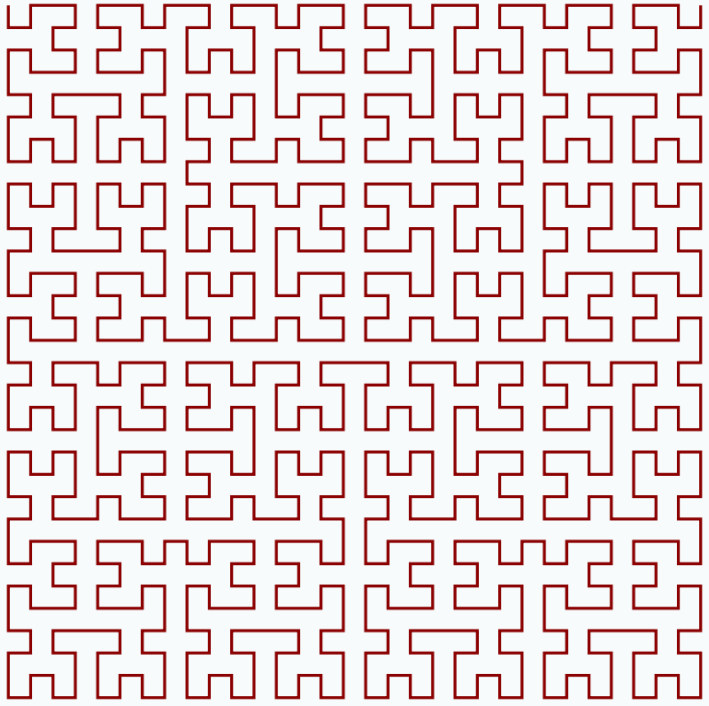

- G. Hutton, E. Meijer Monadic Parsing in Haskell
- G. Hutton, E. Meijer Monadic Parser Combinators

# 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/DDF2019
- **Twitter** @CarstenK_Dev