FUNKTIONALES C# PRAKTISCH DEMONSTRIERT

AM BEIPSPIEL PARSER-KOMBINATOREN

Carsten König

- @CarstenK_dev
- Github github.com/CarstenKoenig/MDD_2017

AGENDA

- Funktionales C#
- Was sind Parser
- *funktionale* Parser-Kombinatoren in C#
- Beispiel Taschenrechner `2 * 3 + 4` -> `= 10`

FUNKTIONALES IN C#

- C# *war* schon immer eine funktionale Sprache
- funktionale Aspekte werden *ständig* ausgebaut

GESCHICHTE

C# 1 DELEGATE

```
1: delegate int Arith(int a, int b);
2:
3: class Impl
4: {
5:   public static int Plus(int a, int b)
6:   { return a + b; }
7: }
8:
9: Arith plus = Impl.Plus;
```

GESCHICHTE 2

- C# 2:
  - annonyme Delegaten
  - *method group conversions*
- C# 3
  - Lambdas
  - *query expressions*
  - LINQ

GESCHICHTE 3

- C# 6
  - *null conditional operators*
- C# 7
  - lokale Funktionen
  - *pattern matching*
  - Tuples
  - *deconstruction*

PARSER

WAS IST EIN PARSER?

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

BEISPIEL

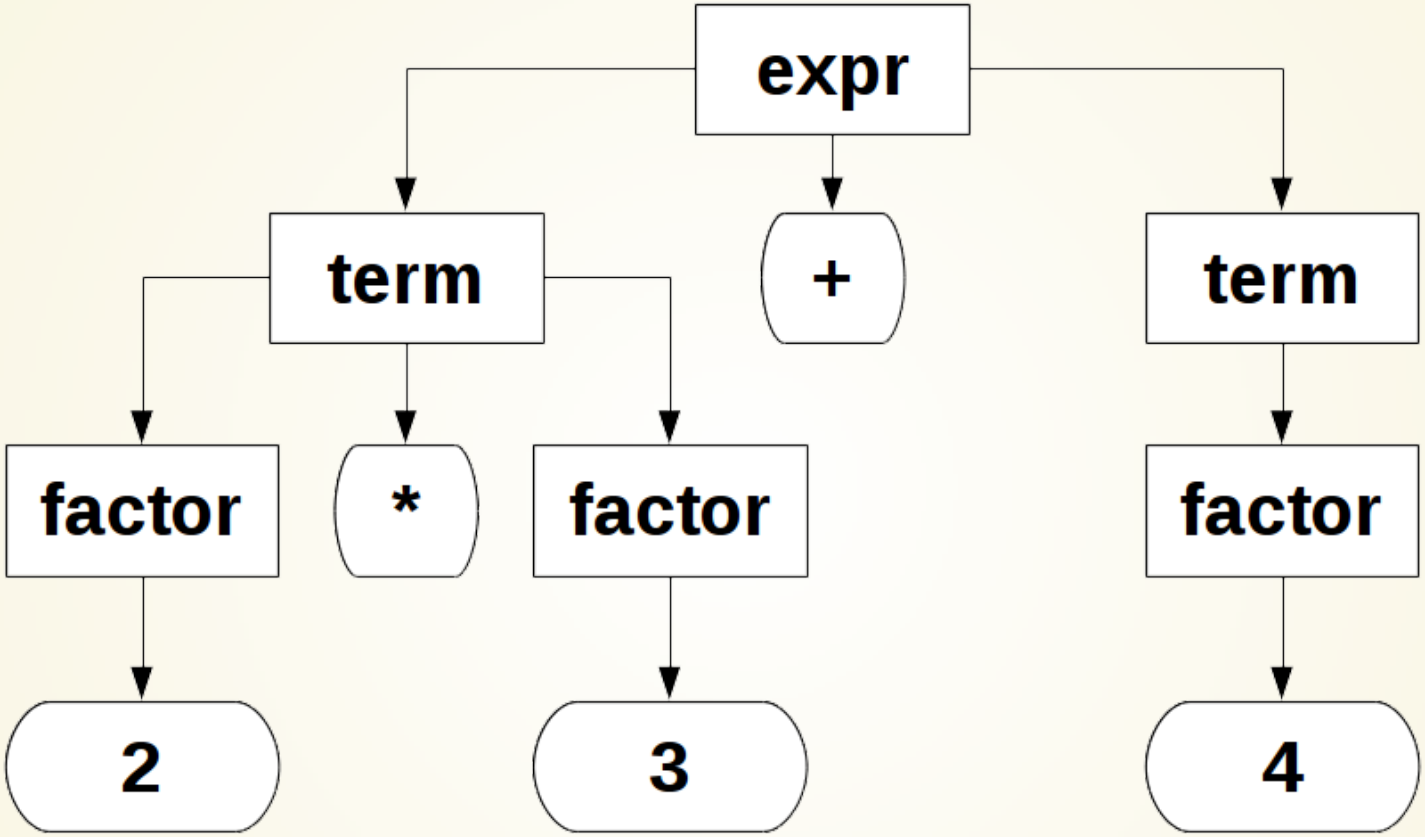
Eingabe: **Quelltext**

Ausgabe: **Syntaxbaum**

VON

$2 * 3 + 4$

2 * 3 + 4



MANUELL

einfache Parser direkt durch Funktionen implementieren (siehe wikipedia)

```
1:  void term(void) {
2:      factor();
3:      while (sym == times || sym == slash) {
4:          nextsym();
5:          factor();
6:      }
7:  }
8:
9:  void expression(void) {
10:     if (sym == plus || sym == minus)
11:         nextsym();
12:     term();
13:     while (sym == plus || sym == minus) {
14:         nextsym();
15:         term();
16:     }
17: }
```

COMPILER-COMPILER (GRAMMATIK)

übersetzen ein Grammatik wie

```
1: expr   ::= expr addop term | term
2: term   ::= term mulop factor | factor
3: factor ::= Zahl | ( expr )
   addop  ::= + | -
4: mulop  ::= * | /
5:
```

in eine *state-machine* in der *Zielsprache*

COMPILER-COMPILER

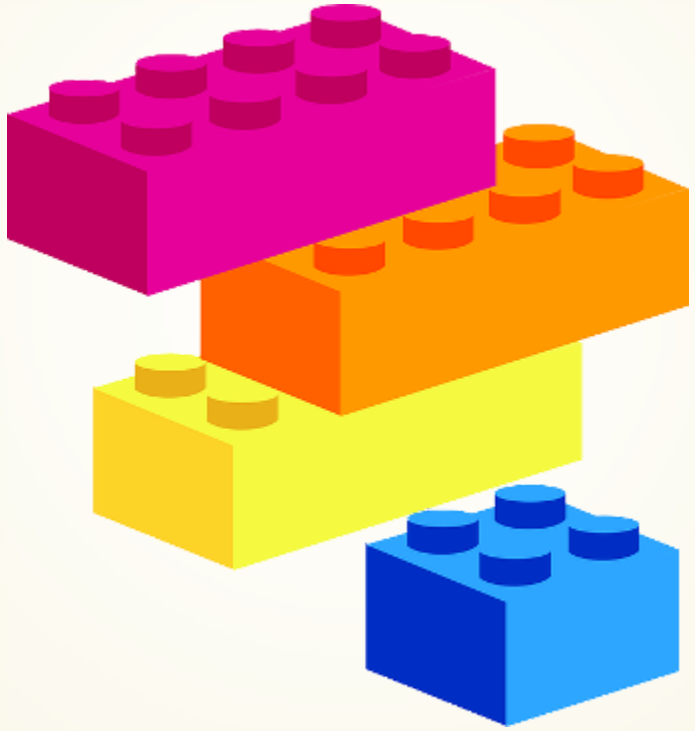
- YACC
- ANTLR

FUNKTIONALER PARSER

Umsetzung als parser combinator Bibliothek

FUNKTIONALER PARSER

KOMBINATOR?



Funktionen die *Parser* zu neuen *Parsern* zusammensetzen

DAZU

- Parser als Datenstruktur darstellen
- *Kombinatoren* als **Funktionen** dieser Datenstrukturen schreiben

PARSER

BAUSTEINE

WAS IST EIN PARSER?

verarbeitet **Eingabetoken** und *berechnet* einen **Ausgabewert**

SOLL KOMBINBAR SEIN

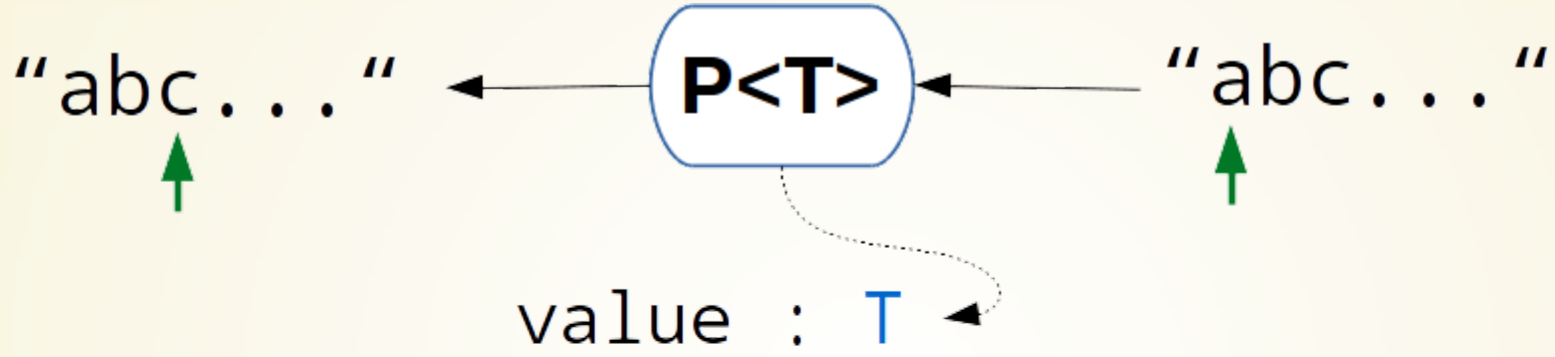
$\Longrightarrow$ verarbeitet/*konsumiert* vielleicht nur einen Teil der Eingabe

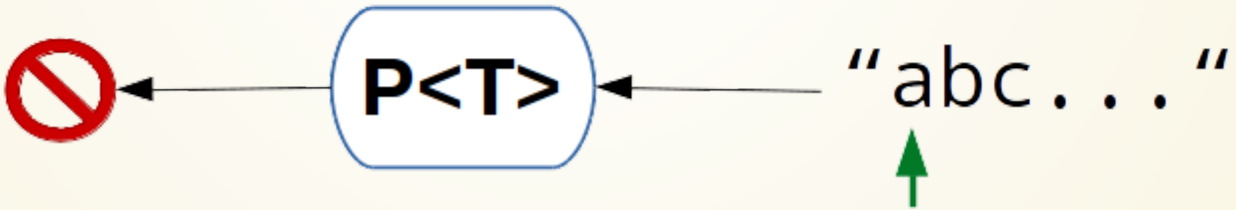
$\Longrightarrow$ *Rückgabe* des Parsers soll den Rest der *Eingabe* anzeigen

DEFINITION PARSER

- bekommt als *Eingabe* eine **Position** innerhalb des *Quelltextes*
- *Ausgabe* ist entweder
  - **neue** Position *zusammen* mit **erkannten Wert**
  - oder der Parser **schlägt fehl**

DEFINITION PARSER






```
1: delegate ParserResult<T> Parser<T>(ParserPosition position);
```

PARSERPOSITION

```
1: delegate ParserResult<T> Parser<T>(ParserPosition position);
2:
3: class ParserPosition
4: {
5:     public string Text { get; }
6:     public int Index { get; }
7:     public char? CurrentChar { get {...} }
8:     public ParserPosition Next () { ... }
9:     ...
10: }
```

FUNKTIONAL

`ParsePosition` ist **immutable**

```
1:      public string Text { get; }
2:
3:      public ParserPosition Next () { ... }
```

ABER NULL?

```
1:      public char? CurrentChar { get {...} }
```

zumindest mit *Resharper* und C# 6 (`?.`, `??`, ...) **ok**

PARSERRESULT

OOP Version eines **Co**-*Produkts*

```
 1: abstract class ParserResult<T>
 2: {
 3:     public static ParserResult<T>
 4:      Succeed(T value, ParserPosition rest) { ... }
 5:
 6:     public static ParserResult<T>
 7:      Failed(string error, ParserPosition position) { ... }
 8:
 9:     private class Success : ParserResult<T> { ... }
10:     private class Failure : ParserResult<T> { ... }
11: }
```

WIE AN DIE WERTE KOMMEN?

```
 1: abstract class ParserResult<T>
 2: {
 3:     abstract TRes Match<TRes> (
 4:         Func<T, ParserPosition, TRes> onSuccess,
 5:         Func<string, ParserPosition, TRes> onFailure );
 6:   ...
 7:
 8:     class Success {
 9:          private T Value { get; }
10:
11:          public override TRes Match<TRes> (
12:              Func<T, ParserPosition, TRes> onSuccess,
13:              Func<string, ParserPosition, TRes> onFailure )
14:          {
15:              return onSuccess(Value, RestText);
16:          }
```

PARSERRESULT IST EIN FUNKTOR

BEISPIEL

```
1:    ParserResult<string> textResult = ...
2:    ParserResult<int> zahlResult =
3:        textResult.Map(int.Parse);
```

- falls `Failure<string>` gibt `Failure<int>`
- falls `Success<string>` mit Wert `s` gibt `Success<int>` mit Wert `int.Parse(s)`

IMPLEMENTATION

```
1: abstract class ParserResult<T>
2: {
3:     ParserResult<TRes> Map<TRes>(Func<T, TRes> map)
4:     {
5:        return Match(
6:          onSuccess: (v,r) =>
7:            ParserResult<TRes>.Succeed( map(v), r ),
8:          onFailure: ParserResult<TRes>.Failed);
9:     }
```

FUNKTOR GESETZE

muss eine Reihe *Gesetze* erfüllen:

- `result.Map(x => x)` ist das Gleiche wie `result`
- `result.Map(x => g(f(x)))` ist das Gleiche wie `result.Map(f).Map(g)`

WEITERE BEISPIELE

- `IEnumerable<T>` (mit `Enumerable.Select`)
- `Task<T>` mit

```
1: async Task<TRes> Map<T,TRes>(Task<T> task, Func<T,TRes> map)
2: {
3:     var result = await task;
4:     return map(result);
5: }
```

PARSER BENUTZEN

Wie bekomme ich das `ParserResult` zu einem *Eingabestring*?

PARSEN

```
 1: public static class Parsers
 2: {
 3:         public static TRes TryParse<T, TRes>(
 4:             this Parser<T> parser, string text,
 5:             Func<T, TRes> onSuccess,
 6:             Func<string, ParserPosition, TRes> onFailure)
 7:         {
 8:             var start = ParserPosition.StarteMit(text);
 9:             var result = parser(start);
10:             return
11:                 result.Match((v, _) => onSuccess(v), onFailure);
12:         }
```

EXCEPTION FALLS GESCHEITERT

```
1: public static T Parse<T>(this Parser<T> parser, string text)
2: {
3:     return
4:         parser.TryParse(text,
5:             onSuccess: x => x,
6:             onFailure: (err, _) => throw new Exception(err));
7: }
```

PARSER KOMBINIEREN

ERINNERUNG: KOMBINATOR

Funktionen,

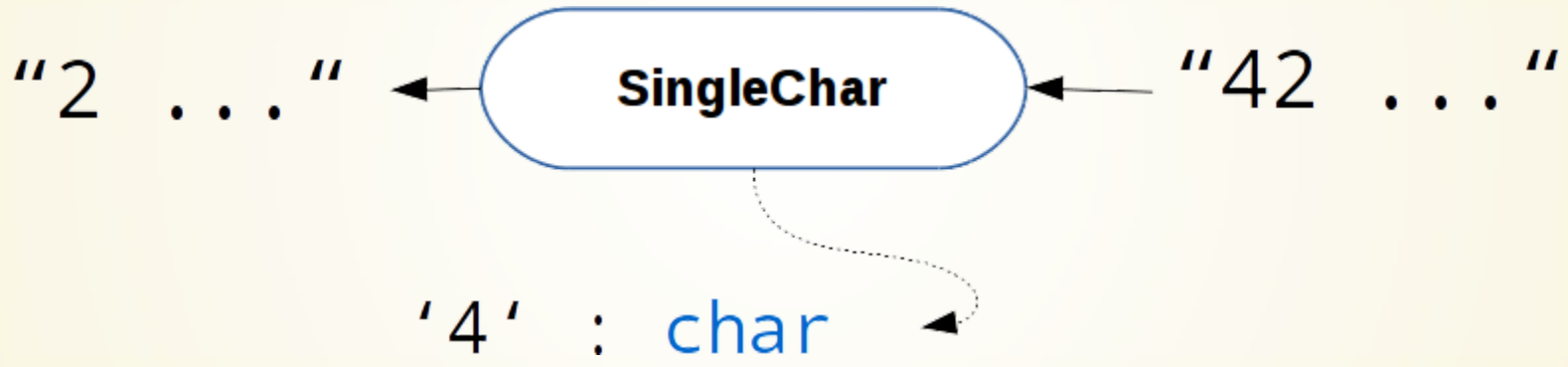
*die Parser als **Eingabe** erhalten können*

*und Parser **zurückgeben***

PRIMITIVE KOMBINATOREN

```
 1: delegate ParserResult<T> Parser<T>(ParserPosition position);
 2:
 3:
 4: Parser<T> Return<T>(T value)
 5: {
 6:     return pos => ParserResult<T>.Succeed(value, pos);
 7: }
 8:
 9: Parser<T> Fail<T>(string error)
10: {
11:     return pos => ParserResult<T>.Failed(error, pos);
12: }
```

EINZELNES ZEICHEN



EINZELNES ZEICHEN

```
 1: Parser<char> SingleChar()
 2: {
 3:     return pos =>
 4:     {
 5:         if (!pos.CurrentChar.HasValue)
 6:             return ParserResult<char>
 7:                 .Failed("Unerwartetes Ende der Eingabe", pos);
 8:
 9:         var zeichen = pos.CurrentChar.Value;
10:         return ParserResult.Succeed(zeichen, pos.Next());
11:     };
12: }
```

bitte beachten: hier wir die Position vorgerrückt

EIN ZEICHEN PARSEN, DASS EINE EIGENSCHAFT ERFÜLLT

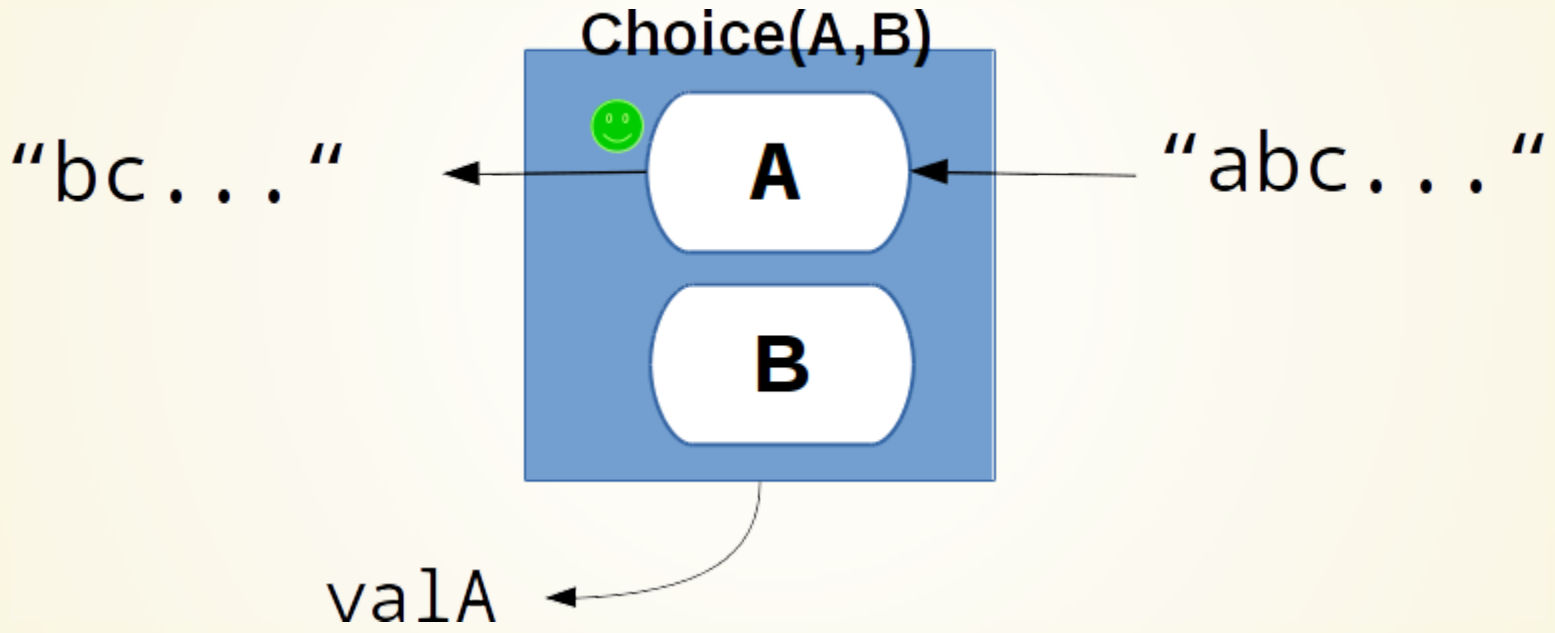
```
 1:
 2: Parser<char> Satisfy(Func<char, bool> property)
 3: {
 4:     return pos =>
 5:     {
 6:         if (!pos.CurrentChar.HasValue)
 7:             return ParserResult<char>
 8:                 .Failed("Unerwartetes Ende der Eingabe", pos);
 9:
10:         var zeichen = pos.CurrentChar.Value;
11:
12:         return !property(zeichen)
13:             ? ParserResult<char>.Failed(
14:                     $"Zeichen [{zeichen}] erfüllt Bedingung nicht", pos)
15:             : ParserResult<char>.Succeed(zeichen, pos.Next());
16:     };
17: }
18:
19:
20:
```

ENTWEDER-ODER KOMBINATOR

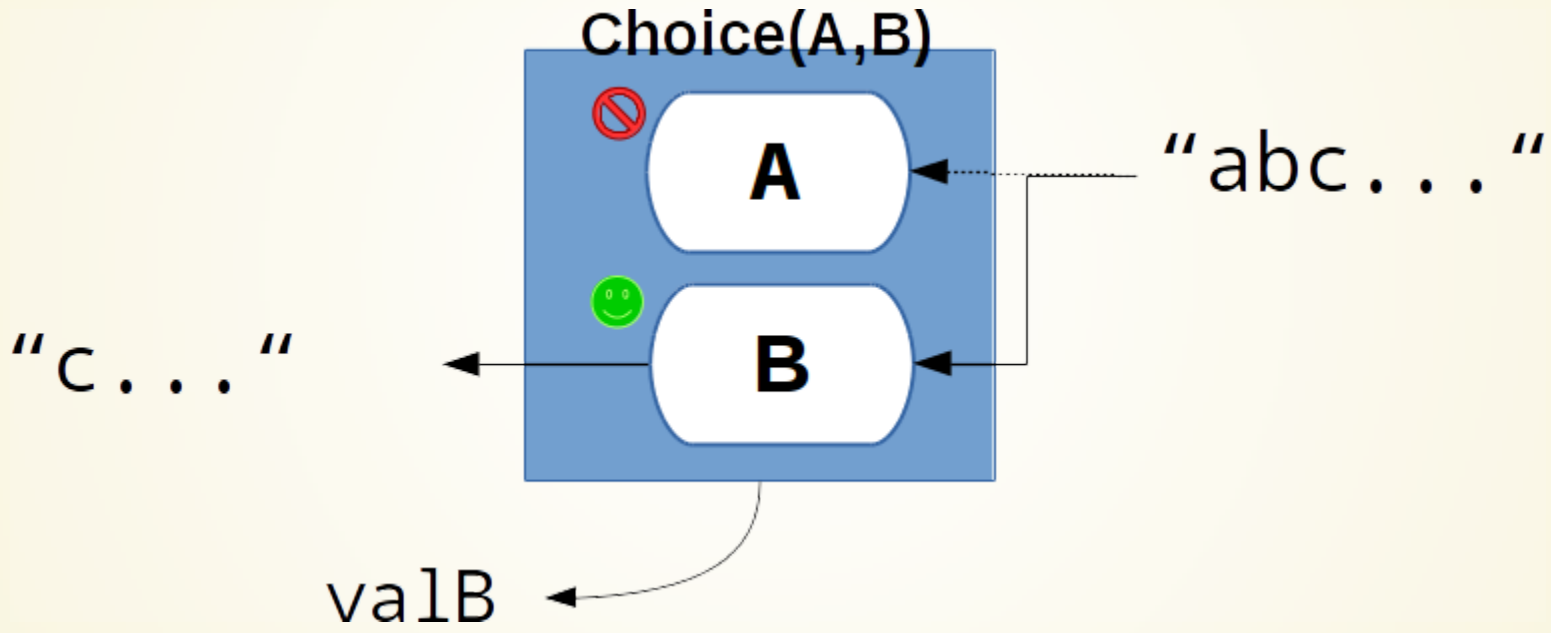
Idee: versuche einen Parser, falls dieser erfolgreich ist verwende dessen Ergebnis

sonst benutze einen *alternativen* Parser (hier *backtracking*)

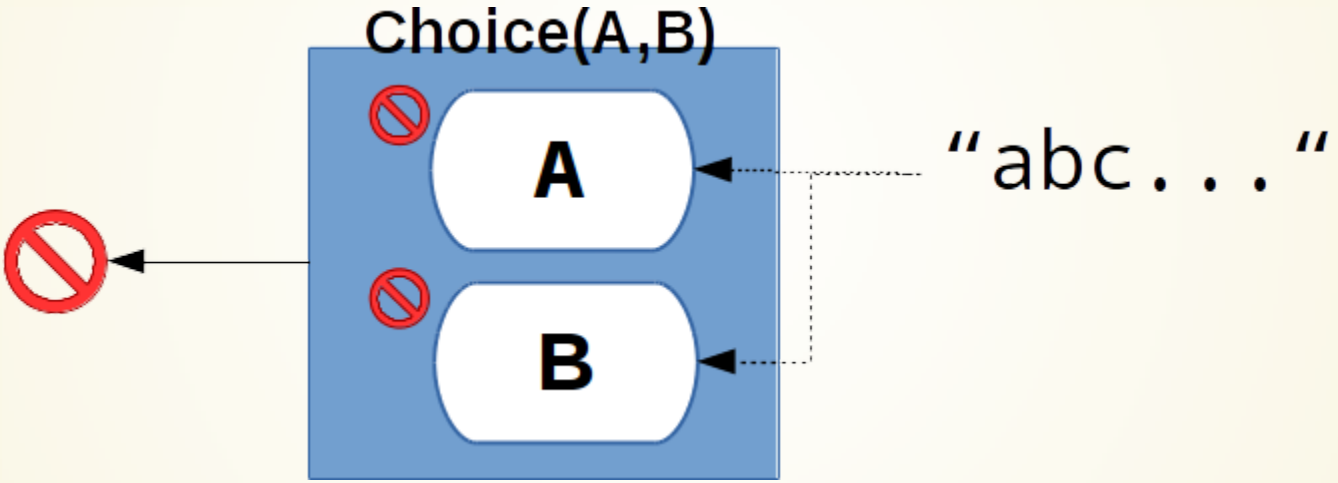
A IST ERFOLGREICH



B IST ERFOLGREICH



BEIDE SCHEITERN



IMPLEMENTATION

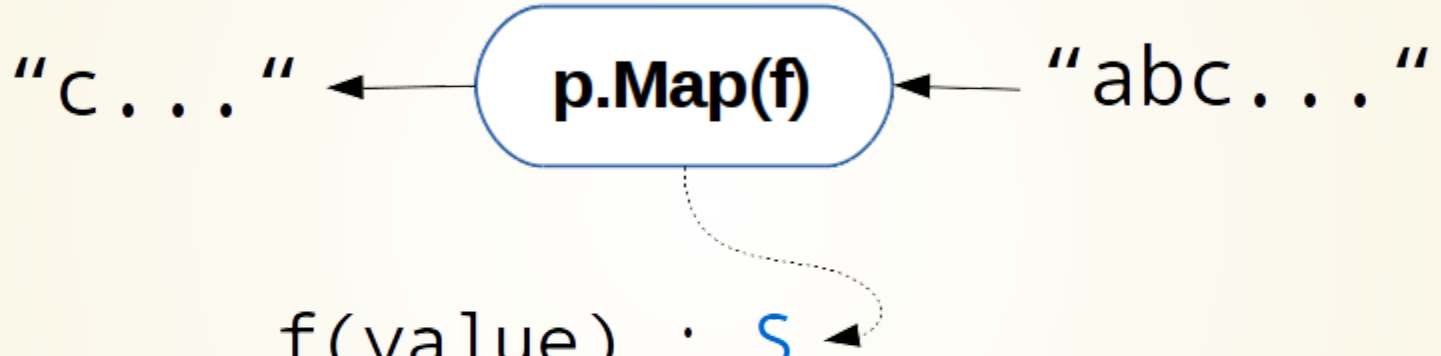
```
1: Parser<T> Choice<T>(Parser<T> parserA, Parser<T> parserB)
2: {
3:     return pos =>
4:       parserA(pos).Match(
5:             onSuccess: ParserResult<T>.Succeed,
6:             onFailure: (err, _) => parserB(pos));
7: }
```

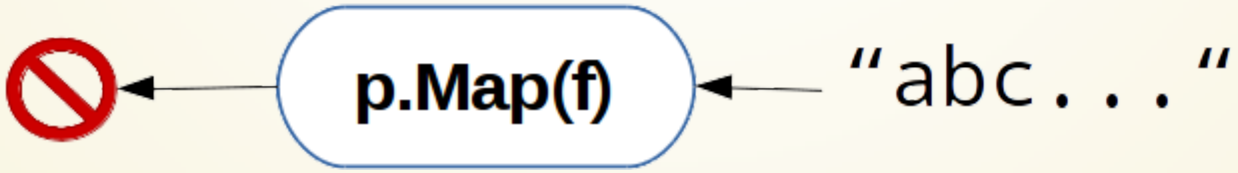
KOMPLEXERE KOMBINATOREN

FUNKTOR

erbt die Funktor-Eigenschaft von `ParserResult`

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





IMPLEMENTATION

```
1: Parser<TRes> Map<T, TRes>(this Parser<T> parser, Func<T, TRes> map)
2: {
3:     return pos => parser(pos).Map(map);
4: }
```

ERBT?

- `ParserResult<_>` ist ein Funktor
- `Func<T, _>` ist ein **Funktor**
- und die Komposition von *Funktoren* ist wieder ein Funktor

- `Func<T, _>` ist ein **Funktor**

```
1: Func<T,Tb> Map<T,Ta,Tb>(Func<T,Ta> f, Func<Ta,Tb> map)
2: {
3:     return t => map( f(t) );
4: }
```

KOMPLEXERE KOMBINATOREN

NACHEINANDER PARSEN

Idee: die *Position* nach dem Parsen des **ersten**, als Eingabe für den **zweiten** Parser nutzen.

```
1: // pseudo
2: ParserPosition pos =>
3: {
4:     var result1 = parser1(pos);
5:     var result2 = parser2(result1.Position);
6:     ...
7: }
```

Aber: *was* mit dem *Wert* von `result1` machen?

PARSER - *FACTORY*

nicht *einfach* zweiter Parser **sondern**

```
1: Func<Tausgabe1, Parser<Tausgabe>> factory
```

```
1: // pseudo
2: ParserPosition pos =>
3: {
4:     var result1 = parser1(pos);
5:     var parser2 = factory(result1.Wert);
6:     return parser2(result1.Position);
7: }
```

IMPLEMENTATION

```
1: Parser<TRes> Bind<T, TRes>(
2:         this Parser<T> parser, Func<T,
3:         Parser<TRes>> bind)
4: {
5:     return pos => parser(pos)
6:         .Match(
7:             onSuccess: (v, p) => bind(v)(p),
8:             onFailure: ParserResult<TRes>.Failed
9:         );
10: }
```

A *WARM FUZZY THING*

```
1: Parser<T> Return<T>(T value) {...}
2:
3: Parser<TRes> Bind<T, TRes>(
4:       this Parser<T> parser, Func<T,
5:       Parser<TRes>> bind) { ... }
```

macht das den Parser zu einer **Monade**

KOMPLEXERE KOMBINATOREN

APPLY KOMBINATOR

einen Parser, der eine **Funktion** zurückliefert mit einem Parser, der ein dazu passendes **Argument** liefert verknüpfen

```
1: Parser<TRes> Apply<T, TRes>(
2:     this Parser<Func<T, TRes>> fParser,
3:     Parser<T> valueParser)
4: {
5:     return
6:         fParser.Bind(f =>
7:             valueParser.Map(x => f(x)));
8: }
```

η-conversion

```
x => f(x) == f
```

```
            valueParser.Map(x => f(x))
=
            valueParser.Map(f)
```

```
1: Parser<TRes> Apply<T, TRes>(
2:     this Parser<Func<T, TRes>> fParser,
3:     Parser<T> valueParser)
4: {
5:     return
6:         fParser.Bind(f => valueParser.Map(f));
7: }
```

nochmal Eta

```
1: Parser<TRes> Apply<T, TRes>(
2:       this Parser<Func<T, TRes>> fParser,
3:       Parser<T> valueParser)
4: {
5:     return fParser.Bind(valueParser.Map);
6: }
```

KOMPLEXERE KOMBINATOREN

MANY KOMBINATOR

versucht einen *Parser* so oft wie möglich zu benutzen und liefert eine Sequenz (`IEnumerable<T>`) der Ergebnisse

```
1:  Parser<IEnumerable<T>> Many<T>(this Parser<T> parser)
2:  {
3:      var leererParser = Return(Enumerable.Empty<T>());
4:      return Choice(Many1(parser), leererParser);
5:  }
6:
7:  Parser<IEnumerable<T>> Many1<T>(this Parser<T> parser)
8:  {
9:      return parser.Bind(item =>
10:         Many(parser).Map(items =>
11:             // item vor items hängen
12:             new[] {item}.Concat(items)));
13: }
```

WÄRE SCHÖNER

```
1: Parser<IEnumerable<T>> Many1<T>(this Parser<T> parser)
2: {
3:     return
4:         from item in parser
5:         from items in Many(parser)
6:         select new[] {item}.Concat(items);
7: }
```

LINQ

```
1: Parser<TRes> SelectMany<TSrc, TRes>(
2:     this Parser<TSrc> source,
3:     Func<TSrc, Parser<TRes>> selector)
4: {
5:     return source.Bind(selector);
6: }
```

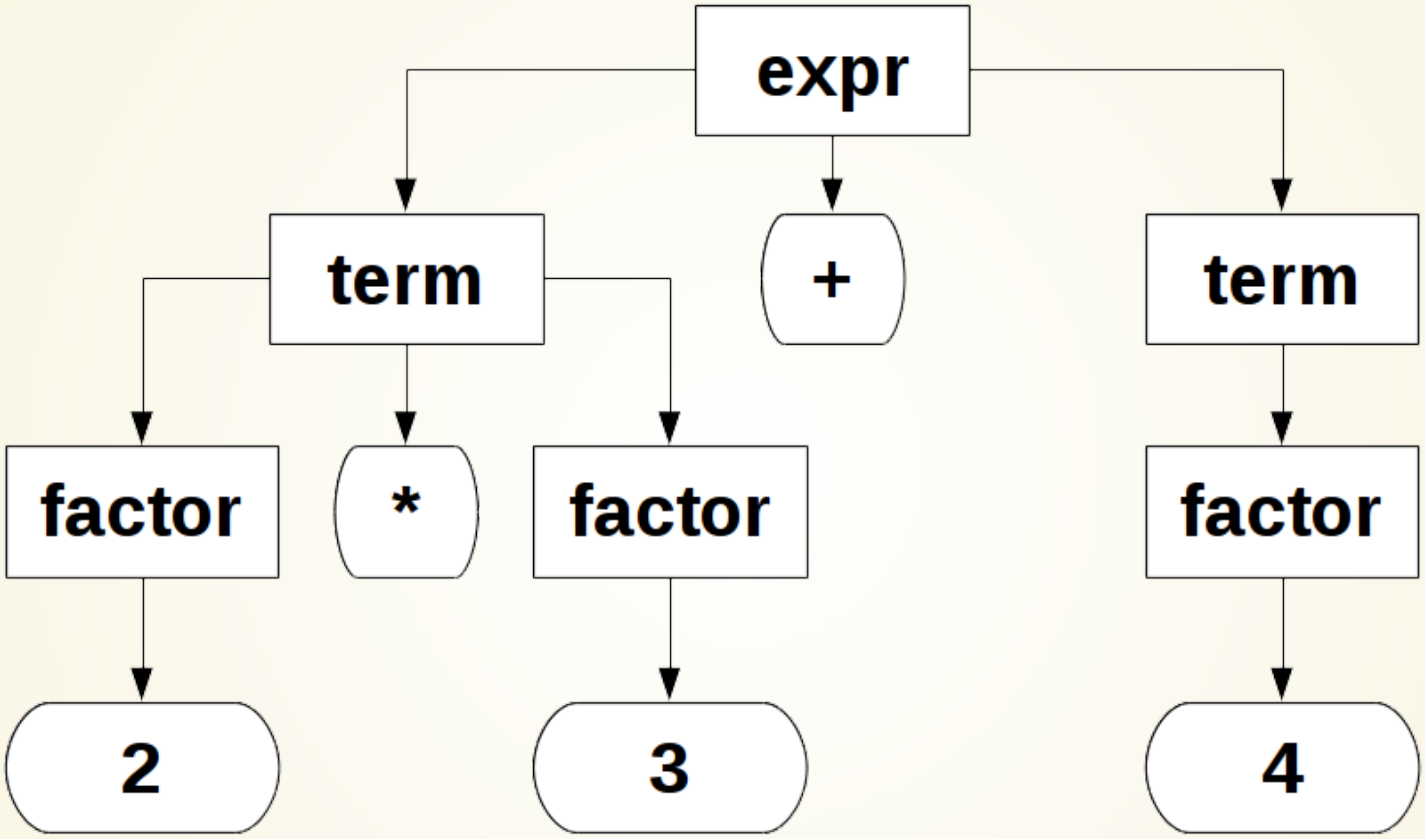
```
Parser<TRes> SelectMany<TSrc, TCol, TRes>(
    this Parser<TSrc> source,
    Func<TSrc, Parser<TCol>> collectionSelector,
    Func<TSrc, TCol, TRes> resultSelector)
{
    return pos => source(pos)
        .Match(
            onSuccess: (src, p) =>
                collectionSelector(src)(p)
                .Map(col => resultSelector(src, col)),
            onError: ParserResult<TRes>.Failed);

}
```

TASCHENRECHNER

ZIEL

```
1: > 2 * 3 + 4
2: 10
```

ERINNERUNG



GRAMMATIK

wollen folgende Grammatik parsen:

```
1: expr   ::= expr addop term | term
2: term   ::= term mulop factor | factor
3: factor ::= int | ( expr )
4: int    ::= - digits | digits
5: digit  ::= (0 | 1 | . . . | 9)*
6: addop  ::= + | -
7: mulop  ::= * | /
```

TOKENS

```
addop  ::= + | -
mulop  ::= * | /
int    ::= - digits | digits
digit  ::= (0 | 1 | . . . | 9)*
```

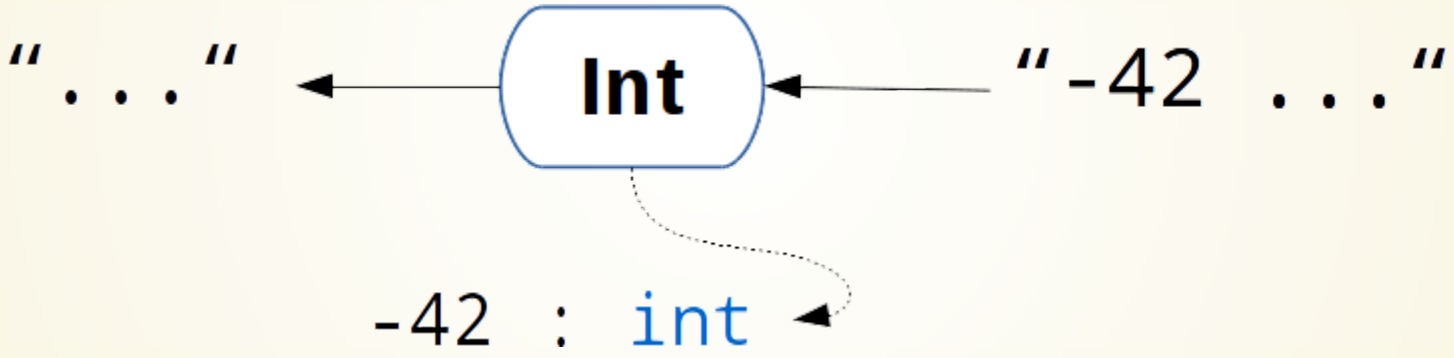
SYMBOL

parst genau ein Zeichen und ignoriert *whitespace* dahinter

```
1: Parser<char> Symbol(char symbol)
2: {
3:     return Parsers
4:         .Satisfy(c => symbol == c)
5:         .IgnoreWhitespaceRight();
6: }
```

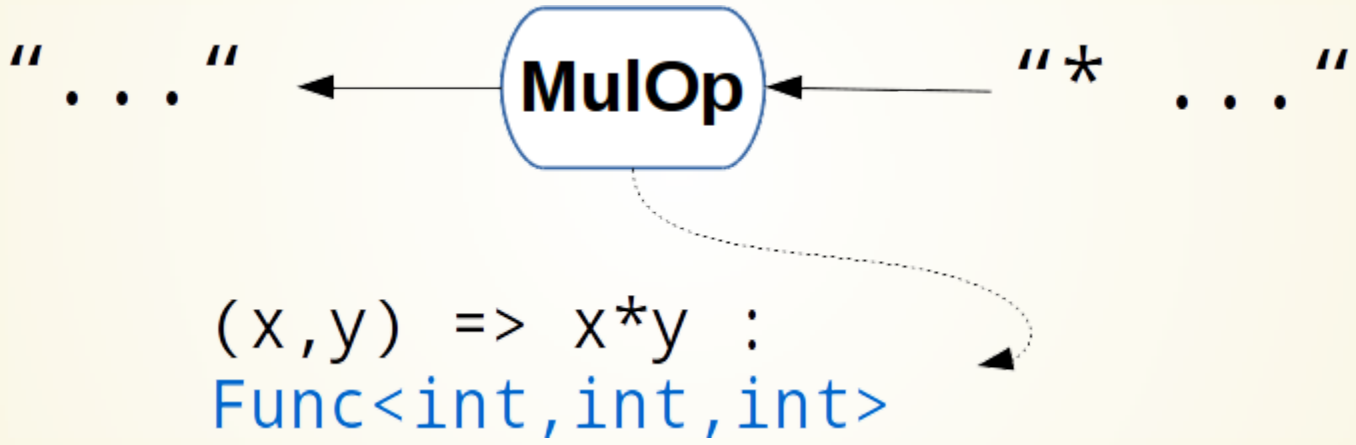
```
1:  Parser<T> IgnoreWhitespaceRight<T>(this Parser<T> parser)
2:  {
3:      return
4:          from p in parser
5:          from s in _whitespace
6:          select p;
7:  }
8:
9:  ...
10:
11: Parser<string> _whitespace = Parsers
12:     .Satisfy(char.IsWhiteSpace)
13:     .Many()
14:     .Map(ToString);
```

GANZZAHL PARSER



```
Parser<int> Int { get {

    var sign = Parsers
        .Choice(
            Symbol('-').Const<char, Func<int, int>>(x => -x),
            Parsers.Return<Func<int, int>>(x => x));

    var digits = Parsers
        .Satisfy(char.IsDigit)
        .Many1()
        .Map(ToString)
        .Map(int.Parse);

    return sign
        .Apply(digits)
        .IgnoreWhitespaceRight();

    }
}
```

MULOP / ADDOP

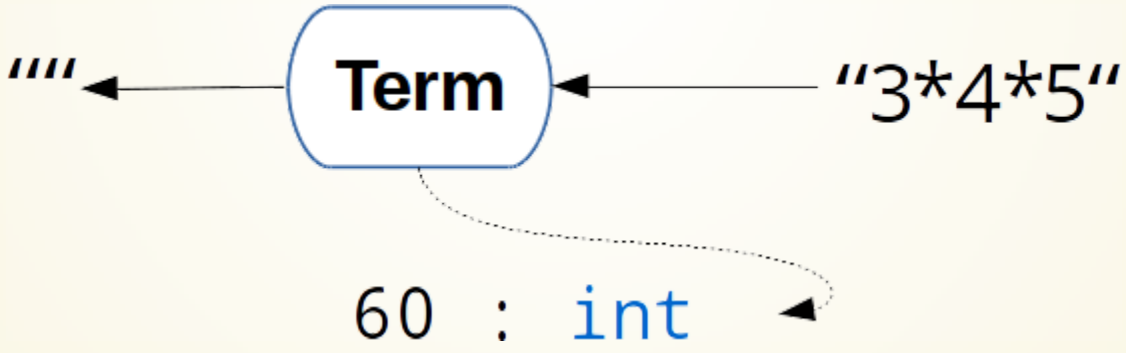


```
1:  Parser<Func<int, int, int>> MulOp
2:  {
3:      get
4:      {
5:          Func<int,int,int> mul = (x,y) => x*y;
6:          Func<int,int,int> div = (x,y) => x/y;
7:
8:          return Choice(
9:                  Tokens.Symbol('*')
10:                     .Const<char, Func<int, int, int>>(mul),
11:                 Tokens.Symbol('/')
12:                     .Const<char, Func<int, int, int>>(div));
13:     }
14: }
```

SYNTAX

```
1: expr    ::= expr addop term | term
2: term    ::= term mulop factor | factor
```

```
1: Parser<int> Expression =>
2:     Term.Chainl1(AddOp);
3:
4: Parser<int> Term =>
5:     Factor.Chainl1(MulOp);
6:
```



WEITEREN KOMBINATOR

Ziel: Eingaben der Form

```
element ° element ° ... ° element
```

parsen, wobei

- `element` ein `Parser<T>`
- der Operator `°` ein `Parser<Func<T,T,T>>`
- Ergebnis den Operator-Wert *links-assoziativ* verknüpft wird `(element ° element) ° element`

IMPLEMENTATION

```
Parser<T> Chainl1<T>(
    this Parser<T> elemParser,
    Parser<Func<T, T, T>> opParser)
{
    Parser<T> Rest(T a)
    {
        var more =
            from f in opParser
            from b in elemParser
            from r in Rest(f(a, b))
            select r;

        return Choice(more, Return(a));
    }

    return
        from x in elemParser
        from y in Rest(x)
        select y;
}
```

LETZTER BAUSTEIN

```
1: factor ::= int | ( expr )
```

```
 1: Parser<int> Factor
 2: {
 3:     get
 4:     {
 5:         var inParents =
 6:                 from l in Tokens.Symbol('(')
 7:                 from i in Expression
 8:                 from r in Tokens.Symbol(')')
 9:                 select i;
10:
11:         return Choice(Tokens.Int, inParents);
12:     }
13: }
```

BLICK ÜBER DEN TELLERRAND

- F#: FParsec
- Elm (`bind` = `andThen`):
  - Parser elm-combine
  - Json Decoder/Encoder
  - Random

REFERENZEN

- G. Hutton, E. Meijer Monadic Parsing in Haskell
- G. Hutton, E. Meijer Monadic Parser Combinators
- fertige Implementation - Github: Sprache

THANK YOU!