Compilerbau

„Eine Programmiersprache entwickeln“

Wieso?

ASM

```
000213C7  lea           eax,[products]
000213CA  mov           dword ptr [p],eax
000213CD  jmp           main+58h (213D8h)
000213CF  mov           eax,dword ptr [p]
000213D2  add           eax,8
000213D5  mov           dword ptr [p],eax
000213D8  lea           eax,[ebp-4]
000213DB  cmp           dword ptr [p],eax
000213DE  je            main+83h (21403h)
000213E0  mov           eax,dword ptr [p]
000213E3  mov           ecx,dword ptr [eax]
000213E5  movsx         edx,byte ptr [ecx]
000213E8  cmp           edx,41h
000213EB  jne           main+81h (21401h)
000213ED  mov           eax,dword ptr [id]
000213F0  mov           ecx,dword ptr [p]
…
```

C

```
for (product* p = products; p != products + numProducts; ++p) {
   if (p->name[0] == 'A') {
         *(ids++) = p->productID;
   }
}
```

C#/LINQ

```
var ids = from prod in products
          where prod.name.StartsWith("A")
          select prod.productID;
```

Expressiveness

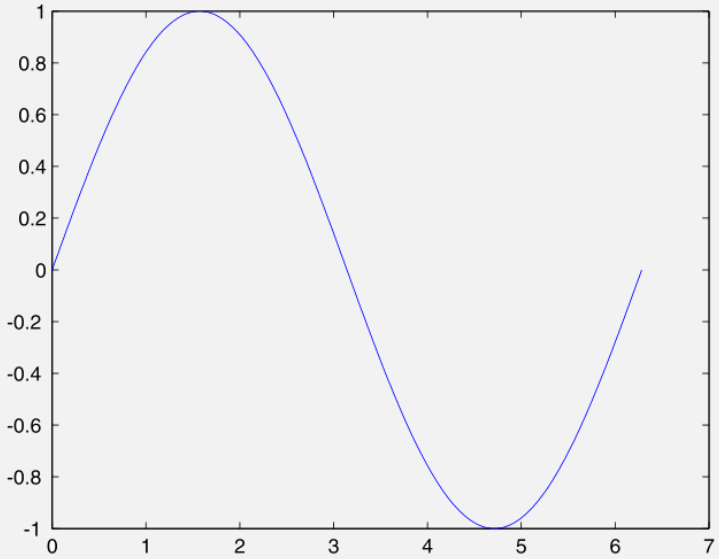
Eigenes Problemfeld – Eigene Sprache?

Prolog

```
parent(adam,peter).
parent(eve,peter).
parent(adam,paul).
parent(marry,paul).

descendant(D,A) :- parent(A,D).
descendant(D,A) :- parent(P,D),descendant(P,A).

? descendant(X, adam).
  X = peter
  X = paul
```

Matlab

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
```



Spezialgebiete

Domain specific languages (DSL)

SQL

```sql
SELECT *
  FROM BOOKS
  WHERE price > 100.00
  ORDER BY title;
```

POV-Ray

```
#declare angle = 0;
#while (angle < 360)
          box { <-0.5, -0.5, -0.5>
                <0.5, 0.5, 0.5>
                texture { pigment { color Red }
                          finish  { specular 0.6 }
                          normal  { agate 0.25 scale 1/2 } }
                rotate angle}
          #declare angle = angle + 45;
#end
```

POV-Ray

```
#declare angle = 0;
#while (angle < 360)
          box { <-0.5, -0.5, -0.5>
                <0.5, 0.5, 0.5>
                texture { pigment { color Red }
                          finish  { specular 0.6 }
                          normal  { agate 0.25 scale 1/2 } }
                rotate angle}
          #declare angle = angle + 45;
#end
```

DSL

- Datenabfrage
- Automatisierung (Makros)
- Scripting
- Meta-Sprachen (Compiler-Compiler)

DSL

Wieso?

- Präzise
- Leichter verständlich
- Fehlerunanfällig

Wieso nicht?

- Kompliziert
- Sorgfältige Planung
- Wirklich nötig?

Learning by doing

1. Parsen

Mit F#

Parser in F#

```
parse
```

Parser in F#

```
parse number "12xyz"
```

Parser in F#

```
parse number "12xyz"
> Success (12, ['x'; 'y'; 'z'])
```

Parser in F#

```
parse number "abc"
> Error
```

Parser in F#

```
number
> val number : Parser<int>
```

Einfache Parser

```
> val anyChar : Parser<char>

parse anyChar "xyz"
> Success ('x', ['y'; 'z'])
```

Einfache Parser

```
parse digit "4"
> Success ('4', [])

> val digit  : Parser<char>
> val letter : Parser<char>
> val char   : char -> Parser<char>
```

Einfache Parser

```
parse digit "4"
> Success ('4', [])

> val digit  : Parser<char>
> val letter : Parser<char>
> val char   : char -> Parser<char>

parse (char 'x') "abc"
> Error

parse (char 'a') "abc"
> Success ('a', ['b'; 'c'])
```

Mehr Parser

```
> val string : string -> Parser<string>

parse (string "Hello") "Hello, World"
> Success ("Hello", [','; ' '; 'W'; 'o'; 'r'; 'l'; 'd'])
```

Parser kombinieren

```
parse (many digit) "123xyz"
> Success (['1'; '2'; '3'], ['x'; 'y'; 'z'])
```

Parser kombinieren

```
parse (many digit) "123xyz"
> Success (['1'; '2'; '3'], ['x'; 'y'; 'z'])

> val many : Parser<T> -> Parser<T list>
```

Alternativen

```
let greeting = (string "Hello") <|> (string "G'day")
```

Parser kombinieren

Age: *42*

Parser kombinieren

```
let parseAge =
    let   _ = string "Age: "
    let age = number
    age
```

Leider nicht so einfach …

Eher …

```
match parse (string "Age: ") input with
| Error -> Error
| Success(_, rest) ->
   match parse number rest with
   | Error -> Error
   | Success(age, rest2) -> Success(age, rest2)
```

Oder doch?

Parser kombinieren

```
let parseAge =
    let   _ = string "Age: "
    let age = number
    age
```

Computation Expressions

```
let parseAge = parser {
    let!   _ = string "Age: "
    let! age = number
    return age
}
```

Computation Expressions

```
let parseAge = parser {
    let!   _ = string "Age: "
    let! age = number
    return age
}

parseAge
> val parseAge : Parser<int>

parse parseAge "Age: 42"
> Success (42, [])
```

Komplexe Parser

[1/3, 4/5, 6/8, 7/18]

[1/3, 4/5, 6/8, 7/18]

```
let fraction = parser {
    let! num = number
    let!   _ = char '/'
    let! den = number
    return (float num) / (float den)
}

let fractionList = parser {
    let! _ = char '['
    let! data = sepBy fraction ", "
    let! _ = char ']'
    return data
}
```

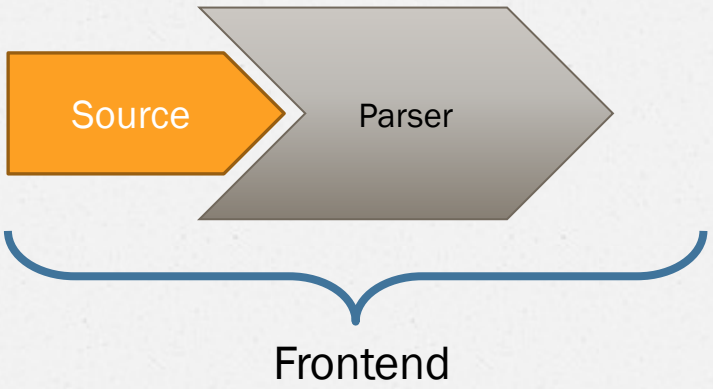
[1/3, 4/5, 6/8, 7/18]

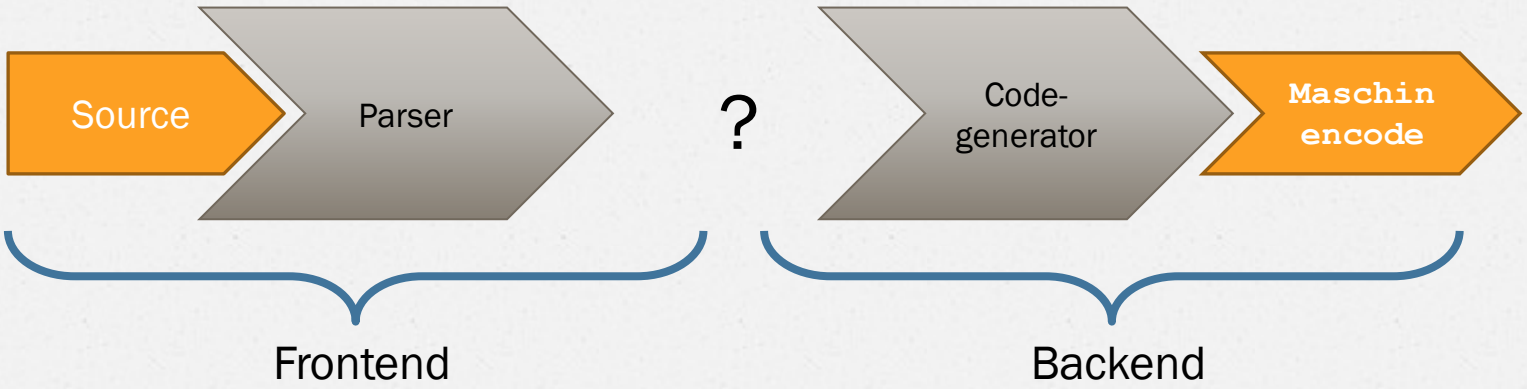
```
let fraction = parser {
    let! num = number
    let!   _ = char '/'
    let! den = number
    return (float num) / (float den)
}

let fractionList = parser {
    let! _ = char '['
    let! data = sepBy fraction ", "
    let! _ = char ']'
    return data
}

parse fractionList "[1/3, 4/6, 9/9]"
> Success ([0.3333333333; 0.6666666667; 1.0], [])
```

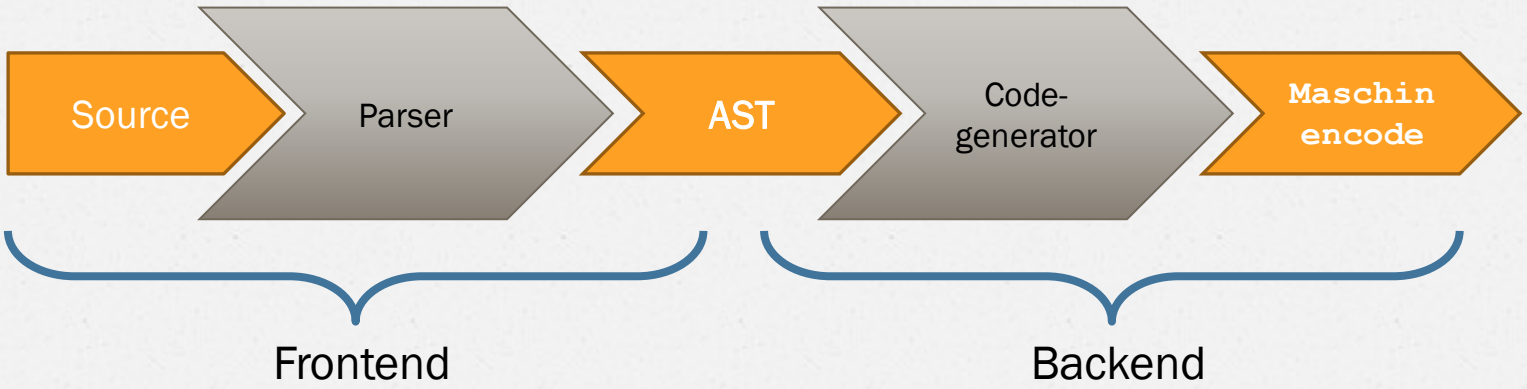
Nützliche Kombinatoren

`many, many1, sepBy, optional, between`

2. Und nun?


Source
Parser
Frontend

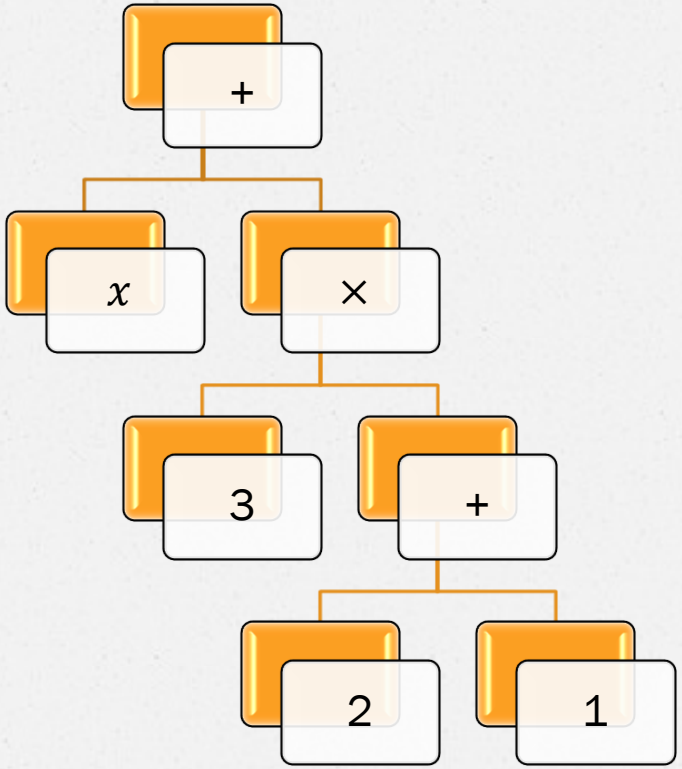

Source
Parser
?
Code-
generator
Maschin
encode
Frontend
Backend


Source
Parser
AST
Code-
generator
Maschin
encode
Frontend
Backend

Am Anfang war der Term

$$x + 3 \cdot (2 + 1)$$

Abstrakter Syntaxbaum = AST



… in code

```
(+ (var 'x')
   (* (number 3)
      (+ (number 2)
         (number 1))))
```

… in F#

```
Plus(Var("x"),
        Mult(Number(3),
             Plus(Number(2),
                  Number(1))))
```

F#-Unions

```
type Expression =
   | Number of int
   | Var    of string
   | Plus   of Expression * Expression
   | Mult   of Expression * Expression
```

Gesucht:

```
var plusTerm : Parser<Expression>
```

$$x + 3 \cdot (2 + 1) + \cdots + \cdots$$

$$x + 3 \cdot (2 + 1) + \cdots + \cdots$$

$$\underbrace{x}_{} + \underbrace{3 \cdot (2 + 1)}_{\texttt{multTerm}} + \underbrace{\cdots}_{} + \underbrace{\cdots}_{}$$

```
let plusTerm = sepBy1 multTerm "+"
```

$$x + \underbrace{3}_{\texttt{atom}} \cdot \underbrace{(2 + 1)}_{\texttt{atom}} + \cdots + \cdots$$

```
let plusTerm = sepBy1 multTerm "+"
let multTerm = sepBy1 atom "*"
```

$$x + 3 \cdot (2 + 1) + \cdots + \cdots$$

```
let plusTerm = sepBy1 multTerm "+"
let multTerm = sepBy1 atom "*"
let atom     =
      number
  <|> variable
  <|> between '(' ')' plusTerm
```

$$x + 3 \cdot (2 + 1) + \cdots + \cdots$$

```
let rec plusTerm = sepBy1 multTerm "+"
and multTerm = sepBy1 atom "*"
and atom     =
      number
  <|> variable
  <|> between '(' ')' plusTerm
```

```
let integer  = parser { let! i = number in return Number(i) }
let variable = parser { let! letters = many1 letter
                        in return Var (toStr letters) }

let rec plusTerm = parser {
    let! operands = sepBy1 multTerm "+"
    return reduceBack (fun x sum -> Plus(x, sum)) operands
 }

and multTerm = parser {
    let! operands = sepBy1 atom "*"
    return reduceBack (fun x sum -> Mult(x, sum)) operands
 }

and atom =
      integer
  <|> variable
  <|> betweenChars '(' ')' plusTerm
```

```
parse plusTerm "1+2*3+3*(1+2)"
> Success
    (Plus
       (Number 1,
        Plus
          (Mult (Number 2,Number 3),
           Mult (Number 3,Plus (Number 1,Number 2)))),
     [])
```

Interpretation

Rekursion

```
let rec eval vars = function
    | Number(x)  -> x
    | Var(name)  -> Map.find name vars
    | Plus(a, b) -> (eval vars a) + (eval vars b)
    | Mult(a, b) -> (eval vars a) * (eval vars b)
```

Fertig

```
let rec eval vars = function
    | Number(x)  -> x
    | Var(name)  -> Map.find name vars
    | Plus(a, b) -> (eval vars a) + (eval vars b)
    | Mult(a, b) -> (eval vars a) * (eval vars b)

let test = "1+3*(x+10)"

let (Success(ast, _)) = parse plusTerm test

let vars = Map [ "x", 1 ]
let res = eval vars ast

> val res : int = 34
```