

Parsen and Backtracking

WS 2007-2008

Christian Wawrzinek

1. Grammatiken
2. Parser
3. Backtracking
4. Kombinatoren

1. Grammatiken

2. Parser

3. Backtracking

4. Kombinatoren

ein Beispiel

Start \rightarrow Ausdruck

Ausdruck \rightarrow Term

Ausdruck \rightarrow Term "+" Ausdruck

Term \rightarrow Faktor

Term \rightarrow Faktor "*" Term

Faktor \rightarrow "X"

Faktor \rightarrow "(" Ausdruck ")"

Start

=> Ausdruck

=> Term '+' Ausdruck

=> Faktor '+' Ausdruck

=> 'X' '+' Ausdruck

=> 'X' '+' Term

=> 'X' '+' Faktor '*' Term

=> 'X' '+' 'X' '*' Term

=> 'X' '+' 'X' '*' '(' Ausdruck ')'

=> ...

=> 'X' '+' 'X' '*' '(' 'X' '+' 'X' ')'

1. Grammatiken

2. Parser

3. Backtracking

4. Kombinatoren

Was ist ein Parser?

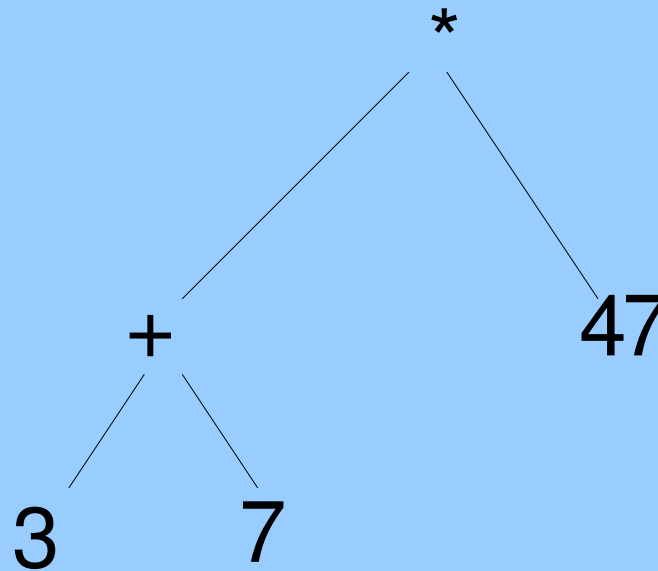
- Ist meistens Teil eines Compilers
- Besteht in der Regel aus 2 Phasen
 1. Lexikalische Analyse
 2. Syntaxanalyse

- entfernt Leerzeichen und Kommentare
- zerlegt Ausdrücke in logische Einheiten
- entfernt überflüssige Einrückungen

- fasst Ausdrücke zu grammatikalischen Sätzen zusammen
- erstellt Syntax-Baum

Ein möglicher Syntax-Baum für folgende
Anweisung:

$(3+7)*47$



1. Grammatiken

2. Parser

3. Backtracking

4. Kombinatoren

Try and Error prinzip:

d.h. es wird versucht, eine erreichte Teillösung schrittweise zu einer Gesamtlösung auszubauen. Wenn absehbar ist, dass eine Teillösung nicht zu einer endgültigen Lösung führen kann, wird der letzte Schritt bzw. die letzten Schritte zurückgenommen, und es werden stattdessen alternative Wege probiert.

1. Grammatiken

2. Parser

3. Backtracking

4. Kombinatoren

sind Funktionen, die andere Funktionen Transformieren

z.B.: map

Funktion auf werte -> Funktion auf Listen

```
map :: (a -> b) -> ([a] -> [b])
```

z.B.: (+1) addiert 1 zu seinem Argument.

map (+1) addiert 1 zu jedem Element in der Argumentliste

```
Hugs> map (+1) [0,1,2]  
[1,2,3]
```

```
type Parser_alg a b = [a] -> [(a, b)]
```

Zum besseren
Verständnis nehmen
diesen Spezialfall an:

a = Char

```
type Parser b = String -> [(String, b)]
```

überprüft ob der 1. Buchstabe eines Strings ein 'x' ist

```
x :: Parser Char
```

```
x ('x':rest) = [(rest, 'x')]
```

```
x _ = []
```


überprüft ob der 1. Buchstabe eines Strings ein 'x' ist:

```
x :: Parser Char  
x ('x':rest) = [(rest, 'x')]  
x _ = []
```

Wenn ja, gibt er dieses und den Rest aus:

```
Main> x "xhhhh"  
[("hhhh",'x')]
```

Primitiver Parser „success“

18

Liefert immer einen Erfolg mit dem angegebenen Wert

```
success :: a -> Parser a
```

```
success wert string = [(string, wert)]
```

Primitiver Parser „success“

Liefert immer einen Erfolg mit dem angegebenen Wert

```
success :: a -> Parser a  
success wert string = [(string, wert)]
```

```
Main> success ';' "bhuhu"  
[("bhuhu",';')]
```

Primitiver Parser „fail“

20

„Fail“ ist ein Parser, der niemals einen Erfolg liefert

```
fail :: Parser a  
fail _ = []
```

Primitiver Parser „fail“

21

„Fail“ ist ein Parser, der niemals einen Erfolg liefert

```
fail :: Parser a  
fail _ = []
```

```
Main> fail "bhuhu"  
[]
```

BNF: $a ::= b \mid c$

parser1 `//` parser2

soll parser1 **oder** parser2 **oder** beide „parsen“
lassen

`(//)` :: Parser a -> Parser a -> Parser a

`(parser1 // parser2) string = parser1 string`

`++ parser2 string`

```
Main> (/|/) x y "xHallo"  
[("Hallo",'x')]
```

Parser 'X'

```
Main> (/|/) x y "yHallo"  
[("Hallo",'y')]
```

Parser 'Y'

```
Main> (/|/) x y "zHallo"  
[]
```

keiner von beiden

optional Kombinator

24

BNF: $a ::= b \mid \langle \text{nichts} \rangle$

optional :: Parser a -> a -> Parser a
optional p wert = p // success wert

```
Main> optional x 'z' "xhuhu"  
[("huhu",'x'),("xhuhu",'z')]
```


$p1 \text{ /-/ } p2$ ist ein Parser, der zunächst $p1$ anwendet, dann $p2$ anwendet und beide Ergebnisse als Tupel zurückliefert.

```
(/-/) :: Parser a -> Parser b -> Parser (a, b)
(parser1 /-/ parser2) string =
  [ (rest2, (ret1, ret2)) |
    (rest1, ret1) <- parser1 string,
    (rest2, ret2) <- parser2 rest1 ]
```

`x /-/ y`

verlangt „xy“ in der Eingabe, und liefert das Tupel ('x', 'y') zurück. Dieses kann mit dem modify-Kombinator weiterverarbeitet werden.

modify Kombinator

27

Der modify Kombinator wendet eine Funktion fun auf die Ausgabe eines Parsers an:


```
modify :: (a -> b) -> Parser a -> Parser b
modify fun parser string =
  [ (rest, fun a) | (rest, a) <- parser string ]
```

modify Kombinator

28

so wird aus:

```
Main> p1 "1Hallo"  
[("Hallo",1)]
```



```
Main> (modify (+1) p1) "1Hallo"  
[("Hallo",2)]
```

Kann auch noch mit z.B.: many kombiniert werden:

```
Main> many (modify (+1) p1) "11Hallo"  
[("Hallo",[2,2]),("1Hallo",[2]),("11Hallo",[1])]
```

many Kombinator

29

BNF: vieleAs ::= a vieleAs

vieleAs ::= <nichts>

ein a könnte man ableiten durch:

vieleAs => a vieleAs (Regel 1) => a <nichts> = a

```
many :: Parser a -> Parser [a]
```

```
many parser = viele // keine
```

```
where viele = modify tcons (parser /-/ many parser)
```

```
tcons (x, xs) = x:xs
```

```
keine = success []
```

many Kombinator

Der 'many' Kombinator gibt eine Liste mit allen möglichen Ableitungen zurück:

```
Main> many x "xxxxHallo"  
[("Hallo", "xxxx"), ("xHallo", "xxx"),  
 ("xxHallo", "xx"), ("xxxHallo", "x"),  
 ("xxxxHallo", "")]
```

$x />-/ y$ parst ein x , das von einem y gefolgt sein muss.
Das y trägt aber zum Rückgabewert nicht bei.

$(/>-/) :: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } a$

$\text{parser1 } />-/ \text{ parser2} = \text{modify fst (parser1 } /-/ \text{ parser2)}$

„rechts-ignorieren“

32

```
Main> (x />-/ y) "xyhuhu"  
[("huhu",'x')]
```


„links ignorieren“

$p1 /->/ p2$ wendet $p1$ und dann $p2$ an, und liefert die Rückgabe von $p2$. Die Rückgabe von $p1$ wird verworfen.

$(/->/) :: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } b$

$\text{parser1 } /->/ \text{ parser2} = \text{modify snd } (\text{parser1 } /-/ \text{ parser2})$

```
data SyntaxBaum = SyntaxBaum Ausdruck
                deriving (Show)
```

```
data Ausdruck = Ausdruck :* Ausdruck
              | Ausdruck :+: Ausdruck
              | X
              deriving (Show)
```

```
start string = modify SyntaxBaum ausdruck
```

ausdruck :: Parser Ausdruck

ausdruck = term

//

modify plusAusdruck (term /-/ (char '+' /->/
ausdruck))

where

plusAusdruck (a1, a2) = a1 :+: a2

term :: Parser Ausdruck

term = faktor

//

modify malAusdruck (faktor /-/ (char '*' /->/ term))

where

malAusdruck (a1, a2) = a1 :* a2

faktor :: Parser Ausdruck

faktor = modify (_ -> X) (char 'X')

//

(char '(' /->/ (ausdruck />-/ char ''))