

# LL-Parser selbst implementiert

---

Carsten Gips (HSBI)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

## Erinnerung Lexer: Zeichenstrom => Tokenstrom

```
def nextToken():
    while (peek != EOF): # globale Variable, über consume()
        switch (peek):
            case ' ': case '\t': case '\n': WS(); continue
            case '[': consume(); return Token(LBRACK, '[')
            ...
            default: raise Error("invalid character: "+peek)
    return Token(EOF_Type, "<EOF>")

def match(c): # Lookahead: Ein Zeichen
    consume()
    if (peek == c): return True
    else: rollBack(); return False

def consume():
    peek = buffer[start]
    start = (start+1) mod 2n
    if (start mod n == 0):
        fill(buffer[start:start+n-1])
    end = (start+n) mod 2n
```

```
r : X s ;
```

```
def r():  
    match(X)  
    s()
```

# LL(1)-Parser

```
list      : '[' elements ']' ;  
elements : INT (',' INT)* ;  
  
INT       : ('0'..'9')+ ;
```

# LL(1)-Parser

```
list      : '[' elements ']' ;  
elements : INT (',' INT)* ;  
  
INT       : ('0'..'9')+ ;
```

```
def list():  
    match(LBRACK); elements(); match(RBRACK);  
  
def elements():  
    match(INT)  
    while lookahead == COMMA: # globale Variable, über consume()  
        match(COMMA); match(INT)
```

## Detail: *match()* und *consume()*

```
def match(x):  
    if lookahead == x: consume()  
    else: raise Exception()  
  
def consume():  
    lookahead = lexer.nextToken()
```

Quelle: Eigener Code basierend auf einer Idee nach (Parr 2010, p. 43)

# Vorrangregeln

$$1+2*3 == 1+(2*3) \neq (1+2)*3$$

Tafel: Unterschiede im AST

# Vorrangregeln

$1+2*3 == 1+(2*3) \neq (1+2)*3$

Tafel: Unterschiede im AST

```
expr : expr '+' term
      | term
      ;
term  : term '*' INT
      | INT
      ;
```



# Vorrangregeln

$1+2*3 == 1+(2*3) \neq (1+2)*3$

Tafel: Unterschiede im AST

```
expr : expr '+' term
      | term
      ;

term : term '*' INT
      | INT
      ;
```

```
expr : expr '*' expr
      | expr '+' expr
      | INT
      ;
```

# Linksrekursion

```
expr : expr '*' expr | expr '+' expr | INT ;
```

```
expr      : addExpr ;  
addExpr   : multExpr ('+' multExpr)* ;  
multExpr  : INT ('*' INT)* ;
```

# Linksrekursion

```
expr : expr '*' expr | expr '+' expr | INT ;
```

```
expr      : addExpr ;  
addExpr   : multExpr ('+' multExpr)* ;  
multExpr  : INT ('*' INT)* ;
```

**Achtung:** Mit *indirekter* Linksrekursion kann ANTLR (v4) *nicht* umgehen:

```
expr : expM | ... ;  
expM : expr '*' expr ;
```

```
expr : ID '++'    // x++  
      | ID '--'    // x--  
      ;
```

```
expr : ID '++'    // x++  
      | ID '--'    // x--  
      ;
```

```
expr : ID ('++' | '--') ;    // x++ oder x--
```

# LL(k)-Parser: Implementierung mit Ringpuffer

```
def match(x):  
    if lookahead(1) == x: consume()  
    else: raise Exception()  
  
def consume():  
    lookahead[start] = lexer.nextToken()  
    start = (start+1) % k  
  
def lookahead(i):  
    return lookahead[(start+i-1) % k]  # i==1: start
```

Quelle: Eigener Code basierend auf einer Idee nach (Parr 2010, p. 47)

Tafel: Beispiel mit Ringpuffer:  $k=3$  und "[1,2,3,4,5]"

- LL(1) und LL(k) mit festem Lookahead
- Implementierung von Vorrang- und Assoziativitätsregeln
- Beachtung und Auflösung von Linksrekursion

# LICENSE



Unless otherwise noted, this work is licensed under CC BY-SA 4.0.