Parsing durch rekursiven Abstieg

- `seq ::= "0" | "1" seq | "2" seq seq`

- **Prüfer:**

```
fun test (0::tr) = tr
  | test (1::tr) = test tr
  | test (2::tr) = test (test tr)
  | test _ = raise Error "seq"
```

- **Parser:**

```
datatype tree = A | B of tree | C of tree * tree

fun parse (0::tr) = (A, tr)
  | parse (1::tr) = let val (s,ts) = parse tr
          in (B s,ts) end
  | parse (2::tr) = let
        val (s,ts) = parse tr
        val (s', ts') = parse ts
        in (C(s,s'),ts') end
  | parse _ = raise Error "parse"
```

RA-taugliche Grammatik

- Eine konkrete Grammatik heißt **RA-tauglich** wenn gilt:
  - Rekursion verringert Argumentliste um mindestens ein Wort
  - Wenn mehrere Alternativen: Wahl aufgrund des ersten Wortes möglich

- `seq ::= "0" | "1" seq | "2" seq seq`

  seq ist RA-tauglich

Ein Parser für Typen

```
ty := pty ["->" ty]
pty ::= "bool" | "int" | "(" ty ")"
```

```
fun ty ts = case pty ts
     of (t, ARROW::tr) => extend (t,tr) ty Arrow
      | s => s
and pty (BOOL::tr) = (Bool, tr)
  | pty (INT::tr) = (Int, tr)
  | pty (LPAR::tr) = match (ty tr) RPAR
```

Hilfsprozeduren:

```
fun extend (a,ts) p f = let val (a',tr) = p ts
         in (f(a,a'),tr) end
```

```
fun match (a,ts) t = if null ts orelse hd ts <> t
     then raise Error "match"
     else (a, tl ts)
```

Rechtsklammernd vs. Linksklammernd

<table>
<tr><th>Rechts-klammernd:</th><th>Links-Klammernd:</th></tr>
<tr><td><pre>ty ::= pty ["->" ty]
pty ::= "bool" | "int" | "(" ty ")"</pre></td><td><pre>ty ::= [ty "->"] pty
pty ::= "bool" | "int" | "(" ty ")"</pre></td></tr>
<tr><td><pre>ty := pty ty'
ty' ::= ["->" ty]
pty ::= "bool" | "int" | "(" ty ")"</pre></td><td><pre>ty := pty ty'
ty' ::= ["->" pty ty']
pty ::= "bool" | "int" | "(" ty ")"</pre></td></tr>
<tr><td><pre>fun ty ts = ty' (pty ts)
and ty' (t, ARROW::tr) =
   extend (t,tr) ty Arrow
  | ty' s => s
and pty ...</pre></td><td><pre>fun ty ts = ty' (pty ts)
and ty' (t, ARROW::tr) =
   ty' (extend (t,tr) pty Arrow)
  | ty' s => s
and pty ...</pre></td></tr>
</table>

Arithmetische Ausdrücke

- **Abstrakte Syntax:**

  $z \in \mathbb{Z}$
  $x \in \mathit{Id}$
  $e \in \mathit{Exp} = z \mid x \mid e + e \mid e * e$

- **Phrasale Syntax:**

```
exp ::= [exp "+"] mexp
mexp ::= [mexp "*"] pexp
pexp ::= num | id | "(" exp ")"
```

- **Lexikalische Syntax:**

```
word ::= "+" | "*" | "(" | ")" | num | id
num ::= [" "] pnum
pnum ::= digit [pnum]
digit ::= "0" | ... | "9"
id ::= letter [id]
letter ::= "a" | ... | "y" | "A" | ... | "Z"
```

Lexer

```
datatype token = ADD | MUL | LPAR | RPAR
               | ICON of int | ID of string

fun lex nil = nil
  | lex (#" "::cr) = lex cr
  | lex (#"\t":: cr) = lex cr
  | lex (#"\n":: cr) = lex cr
  | lex (#"+"::cr) = ADD::lex cr
  | lex (#"*"::cr) = MUL::lex cr
  | lex (#"("::cr) = LPAR::lex cr
  | lex (#")"::cr) = RPAR::lex cr
  | lex (#"~"::c::cr) = if Char.isDigit c
                    then lexInt  1 0 (c::cr)
                    else raise Error "~"
  | lex (c::cr) = if Char.isDigit c
                    then lexInt 1 0 (c::cr)
                    else if Char.isAlpha c
                        then lexId [c] cr
                        else raise Error "lex"
```

Lexer

```
and lexInt s v cs = if null cs orelse not(Char.isDigit(hd cs))
                    then ICON(s*v) :: lex cs
                    else lexInt s (10*v + (ord(hd cs) - ord #"0"))

and lexId cs cs' = if null cs' orelse not (Char.isAlpha(hd cs'))
                   then ID(implode(rev cs)) :: lex cs'
                   else lexId (hd cs' :: cs) (tl cs')
```

Parser

```
datatype exp = Con of int | Id of string
             | Sum of exp * exp | Pro of exp * exp

fun exp ts = exp' (mexp ts)
and exp' (e, ADD::tr) = exp' (extend (e,tr) mexp Sum)
  | exp' s = s
and mexp ts = mexp' (pexp ts)
and mexp' (e, MUL::tr) = mexp' (extend (e,tr) pexp Pro)
  | mexp' s = s
and pexp (ICON z::tr) = (Con z, tr)
  | pexp (ID x :: tr) = (Id x, tr)
  | pexp (LPAR :: tr) = match (exp tr) RPAR
  | pexp _ = raise Error "pexp"
```

Konkrete Syntax für F

- **Phrasale Syntax:**

```
ty ::= pty ["->" ty]
pty ::= "bool" | "int" | "(" ty ")"
exp ::= "if" exp "then" exp "else" exp
    | "fn" id ":" ty "=>" exp
    | aexp [ "<=" aexp]
aexp ::= [ aexp ("+" | "-" ) ] mexp
mexp ::= [ mexp "*"] sexp
sexp ::= [sexp] pexp
pexp ::= "false" | "true" | num | id | "(" exp ")"
```

- **Lexikalische Syntax:**

```
word :: = "+" | "-" | "*" | "(" | ")" |
    "->" | ":" | "=>" | "<=" | num | id

datatype token = ARROW | LPAR | RPAR | COLON (* : *)
    | DARROW (* => *) | LEQ | LEQ | ADD | SUB | MUL
    | BOOL | INT | IF | THEN | ELSE | FN | FALSE | TRUE
    | ICON of int | ID of string
```