Programmierung WS 2002 / 03:
Musterlösung zum 12. Übungsblatt

Prof. Dr. Gert Smolka, Dipl.-Inform. Thorsten Brunklaus

Aufgabe 12.1: Syntaxbäume ($10 = 2 + 2 + 3 * 2$)

(a)

```
ty
├── atty
│   ├── "("
│   ├── ty
│   │   ├── atty
│   │   │   └── "bool"
│   │   ├── "->"
│   │   └── ty
│   │       ├── atty
│   │       │   └── "int"
│   │       ├── "->"
│   │       └── ty
│   │           └── atty
│   │               └── "bool"
│   └── ")"
├── "->"
└── ty
    └── atty
        └── "int"
```

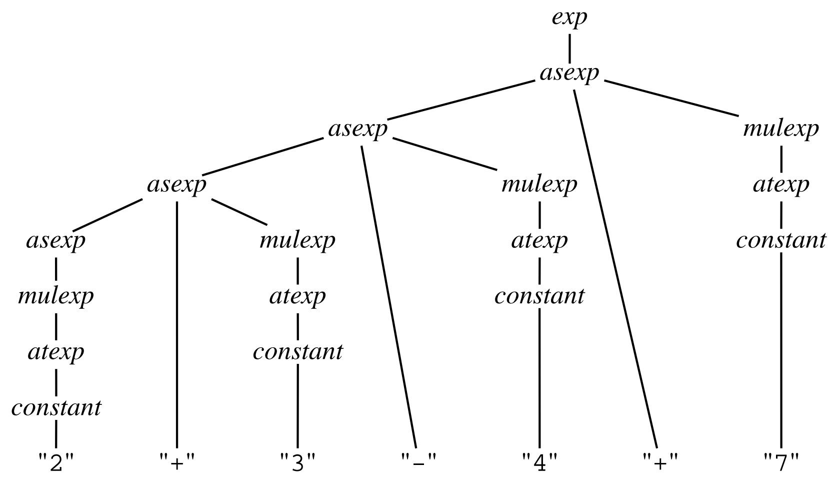
Es gibt genau einen Syntaxbaum, denn die Grammatik ist eindeutig.

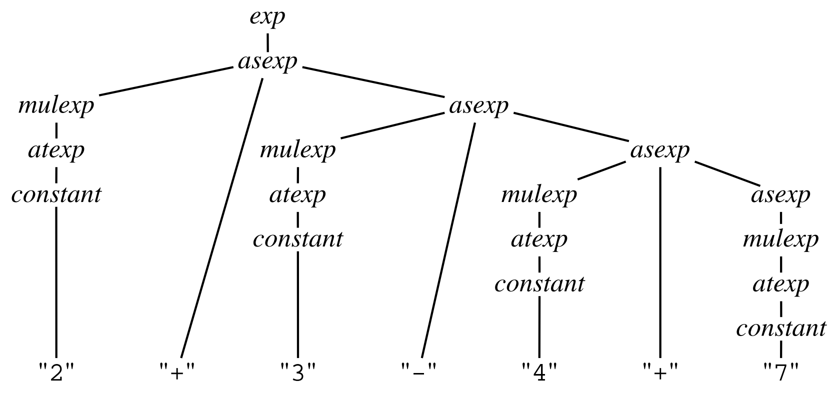
(b)

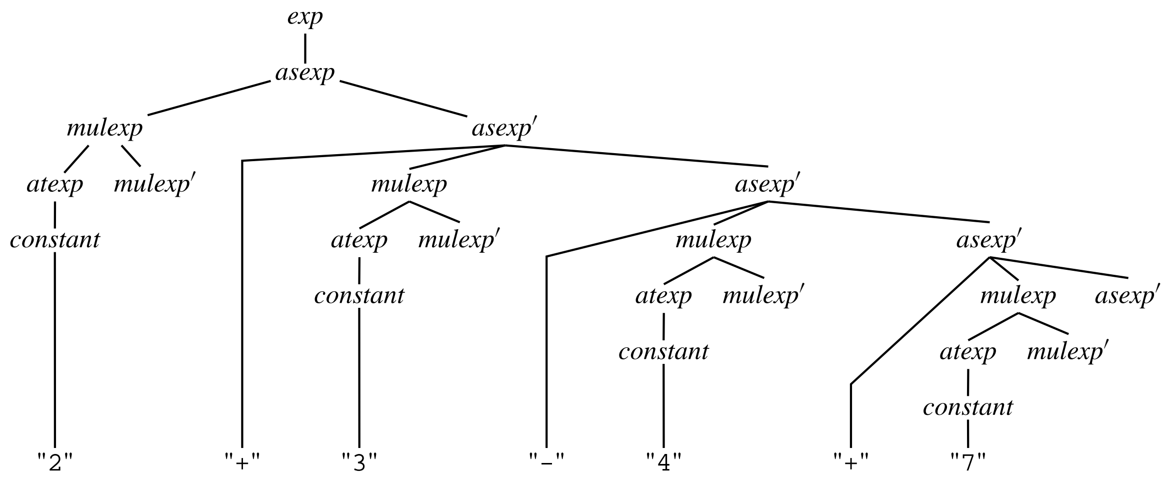
```
exp
└── asexp
    └── mulexp
        ├── mulexp
        │   ├── mulexp
        │   │   └── atexp
        │   │       └── constant
        │   │           └── "3"
        │   ├── "*"
        │   └── atexp
        │       ├── "("
        │       ├── exp
        │       │   └── asexp
        │       │       ├── asexp
        │       │       │   └── mulexp
        │       │       │       ├── mulexp
        │       │       │       │   └── atexp
        │       │       │       │       └── constant
        │       │       │       │           └── "2"
        │       │       │       ├── "*"
        │       │       │       └── atexp
        │       │       │           └── identifier
        │       │       │               └── "y"
        │       │       ├── "+"
        │       │       └── mulexp
        │       │           └── atexp
        │       │               └── identifier
        │       │                   └── "x"
        │       └── ")"
        ├── "*"
        └── atexp
            └── identifier
                └── "z"
```

Es gibt genau drei Syntaxbäume, entweder beginnend mit dem obersten *exp*, *asexp* oder *mulexp*.

(c)

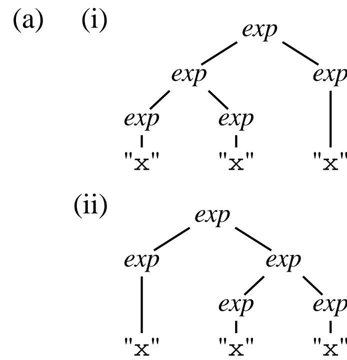






Aufgabe 12.2: Mehrdeutige Grammatik ($6 = 2 + 4$)

(a) (i)



(b)

$$exp \;=\; \texttt{"x"} \mid exp \;\; \texttt{"x"}$$

Aufgabe 12.3: Zeichendarstellung von Typen (8)

```
fun ty (Arrow(t,t')) = atty t ^ "->" ^ ty t'
  | ty      t        = atty t

and atty Bool = "bool"
  | atty Int  = "int"
  | atty  t   = "(" ^ ty t ^ ")"
```

Aufgabe 12.4: Zeichendarstellung von Ausdrücken (10)

```
fun exp (Op(e,Leq,e')) = asexp e ^ "<=" ^ asexp e'
  | exp      e         = asexp e

and asexp (Op(e,Add,e')) = asexp e ^ "+" ^ mulexp e'
  | asexp (Op(e,Sub,e')) = asexp e ^ "-" ^ mulexp e'
  | asexp      e         = mulexp e

and mulexp (Op(e,Mul,e')) = mulexp e ^ "*" ^ atexp e'
  | mulexp      e         = atexp e

and atexp (Con n) = Int.toString n
  | atexp    e    = "(" ^ exp e ^ ")"
```

Aufgabe 12.5: Lexer und Parser für Typen mit Pfeil und Stern $(20 = 6 + 6 + 8)$

(a)

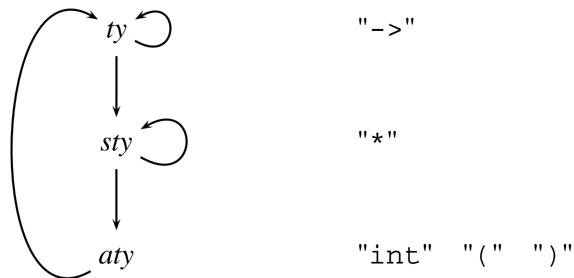
```
fun lex s = lex' nil (String.explode s)

and lex' ts nil                              = rev ts
  | lex' ts (#"i" :: #"n" :: #"t" ::rs) = lex' (INT::ts) rs
  | lex' ts (#"-" :: #">" ::rs)         = lex' (ARROW::ts) rs
  | lex' ts (#"*"::rs)                  = lex' (STAR::ts) rs
  | lex' ts (#"("::rs)                  = lex' (LPAR::ts) rs
  | lex' ts (#")"::rs)                  = lex' (RPAR::ts) rs
  | lex' ts  cs                         = raise Error
```

(b)

ty = *sty* ["->" *ty*]

sty = *aty* ["*" *sty*]

aty = "int" | "(" *ty* ")"



(c)

```
(*
     ty   = sty [ "->" ty ]
     sty  = aty [ "*" sty' ]
     sty' = aty [ "*" sty' ]
     aty  = "int" | "(" ty ")"
*)

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

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

```
fun ty ts = (case sty ts of
               (t, ARROW::tr) =>
                 combine t tr ty (fn (t, t') => Arrow(t, t'))
             | sts            => sts)

and sty ts = (case aty ts of
                (t, STAR::tr) =>
                combine t tr sty' (fn (t, ts) => Star (t :: ts))
              | sts           => sts)

and sty' ts = (case aty ts of
                 (t, STAR::tr) =>
                 combine t tr sty' (fn (t, ts) => (t :: ts))
               | (t, tr)       => ([t], tr))

and aty (INT::tr)  = (Int, tr)
  | aty (LPAR::tr) = match (ty tr) RPAR
  | aty _          = raise Error

fun parse ts = (case ty ts of
                  (t, nil) => t
                | _        => raise Error)
```

Aufgabe 12.6: Kontextfreie Syntax von applikativen Ausdrücken $(16 = 4 * 4)$

(a)

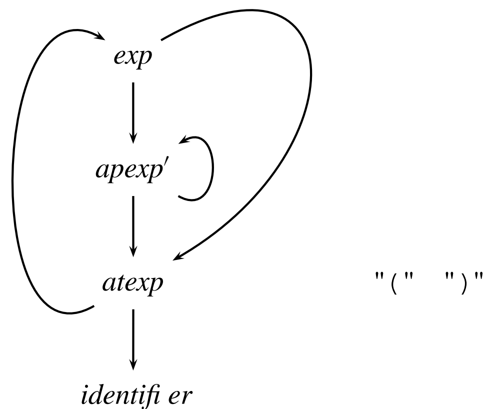
$$\begin{aligned} exp &= [\ exp\ ]\ atexp \\ atexp &= identifier\ \mid\ \texttt{"("}\ exp\ \texttt{")"} \end{aligned}$$

(b)

```
fun exp (App(e,e')) = exp e ^ " " ^ atexp e'
 |  exp e           = atexp e

and atexp (Id s)      = s
 |  atexp e           = "(" ^ exp e ^ ")"
```

(c)

$$\begin{aligned} exp &= atexp\ \ apexp' \\ apexp' &= [\ atexp\ \ apexp'\ ] \\ atexp &= identifier\ \mid\ \texttt{"("}\ exp\ \texttt{")"} \end{aligned}$$



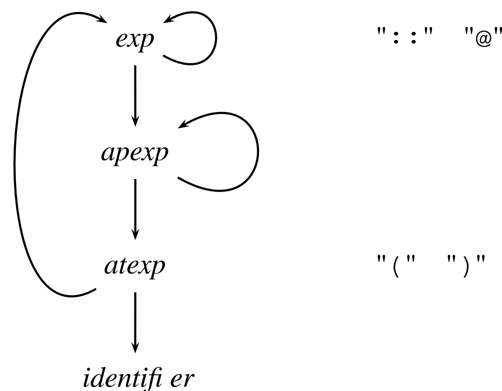
(d)
```
fun exp ts = apexp' (atexp ts)

and atexp (ID _::ts)  = ts
  | atexp (LPAR::ts)  = (case exp ts of RPAR::tr => tr | _ => raise Error)
  | atexp _           = raise Error

and apexp' (ID _::tr) = apexp' tr
  | apexp' (LPAR::tr) = apexp'(atexp (LPAR::tr))
  | apexp' tr         = tr

and test ts = (exp ts = nil) handle Error => false
```

Aufgabe 12.7: Lexer und Parser für Ausdrücke mit Cons und Append ($30 = 10 + 6 + 4 + 10$)

(a)
```
fun lex' ts nil                  = rev ts
  | lex' ts (#" " ::cs)          = lex' ts cs
  | lex' ts (#"\n"::cs)          = lex' ts cs
  | lex' ts (#"\t"::cs)          = lex' ts cs
  | lex' ts (#":" :: #":" ::cs)  = lex' (CONS::ts) cs
  | lex' ts (#"@" ::cs)          = lex' (APPEND::ts) cs
  | lex' ts (#"(" ::cs)          = lex' (LPAR::ts) cs
  | lex' ts (#")" ::cs)          = lex' (RPAR::ts) cs
  | lex' ts (c    ::cs)          = if Char.isAlpha c
                                     then lexA ts [c] cs
                                   else raise Error

and lexA ts xs cs =
  if not(null cs) andalso Char.isAlphaNum(hd cs)
  then lexA ts (hd cs::xs) (tl cs)
  else lex'(ID(implode(rev xs)) :: ts) cs

fun lex s =lex' nil (explode s)
```

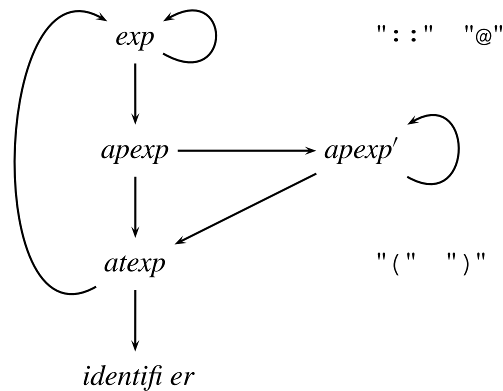
(b)

$$exp = apexp\ [\ (\texttt{"::"}\ |\ \texttt{"@"})\ exp\ ]$$
$$apexp = [\ apexp\ ]\ atexp$$
$$atexp = identifier\ |\ \texttt{"("}\ exp\ \texttt{")"}$$



(c)

$$exp = apexp \; [\; (\texttt{"::"} \mid \texttt{"@"}) \; exp \;]$$
$$apexp = atexp \;\; apexp'$$
$$apexp' = [\; atexp \;\; apexp' \;]$$
$$atexp = identifier \mid \texttt{"("} \; exp \; \texttt{")"}$$



(d)

```
(*
exp     = apexp [ ("::" | "@") exp ]
apexp   = atexp apexp'
apexp'  = [ atexp apexp' ]
atexp   = identifier | "(" exp ")"
*)

datatype opr = Cons | Append

datatype exp = Id of string
             | Op of exp * opr * exp
             | App of exp * exp

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

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

fun opa opr (a,a') = Op(a,opr,a')
```

```
fun firstAtexp (ID _  ::_) = true
  | firstAtexp (LPAR  ::_) = true
  | firstAtexp      _      = false

fun exp ts = case apexp ts of
               (a, CONS  ::tr) => combine a tr exp (opa Cons)
             | (a, APPEND::tr) => combine a tr exp (opa Append)
             |     ats         => ats

and apexp ts = apexp'(atexp ts)

and apexp'(a,ts) = if firstAtexp ts
                   then apexp'(combine a ts atexp App)
                   else (a,ts)

and atexp (ID s  ::ts) = (Id s, ts)
  | atexp (LPAR  ::ts) = match (exp ts) RPAR
  | atexp      _       = raise Error

fun parse ts = case exp ts of
                  (a, nil) => a
                |    _     => raise Error
```