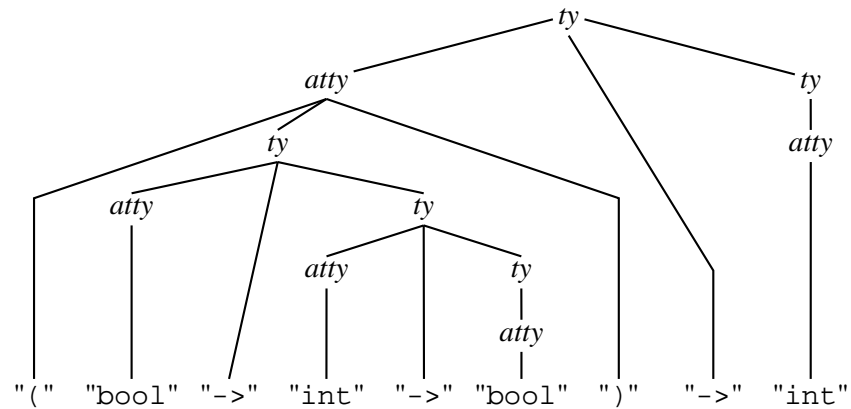
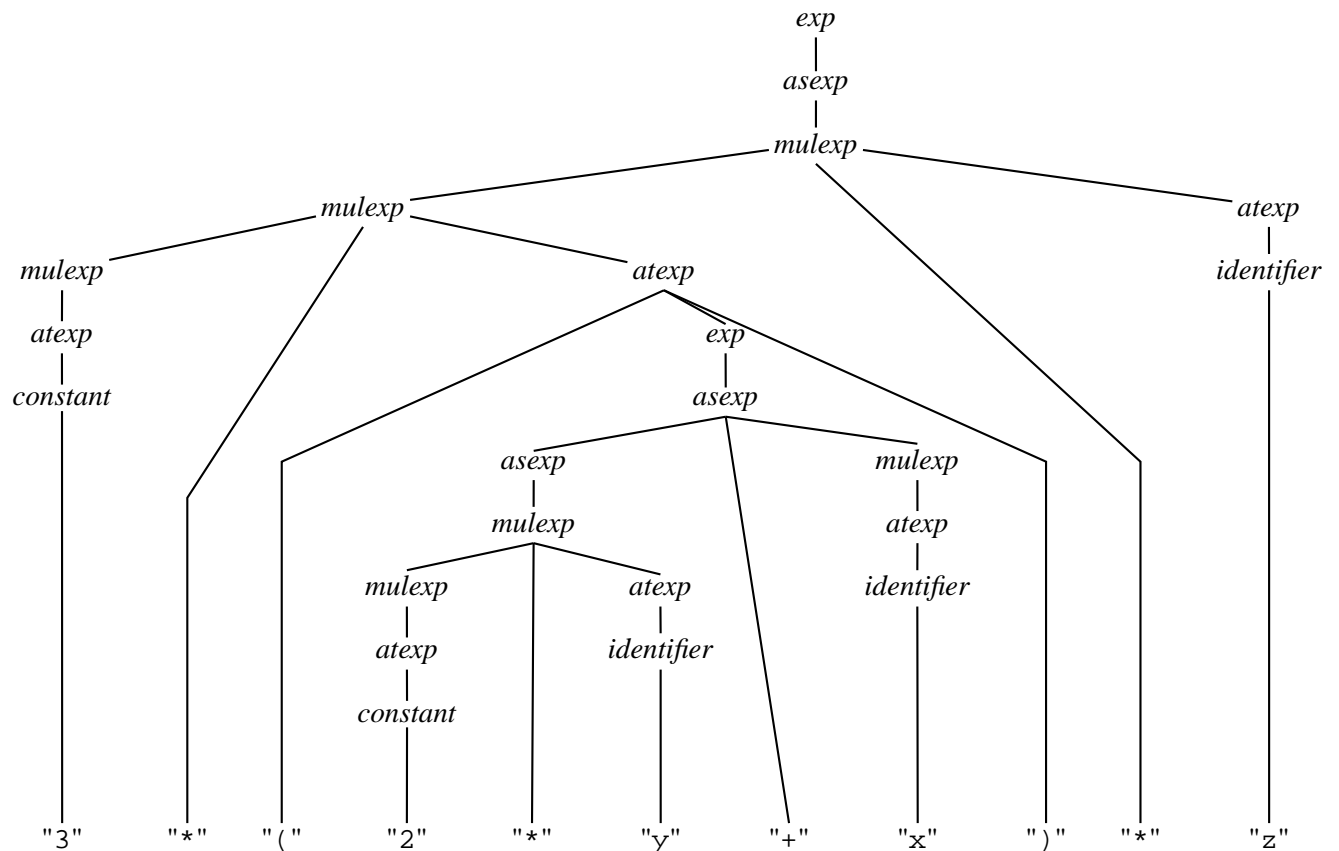


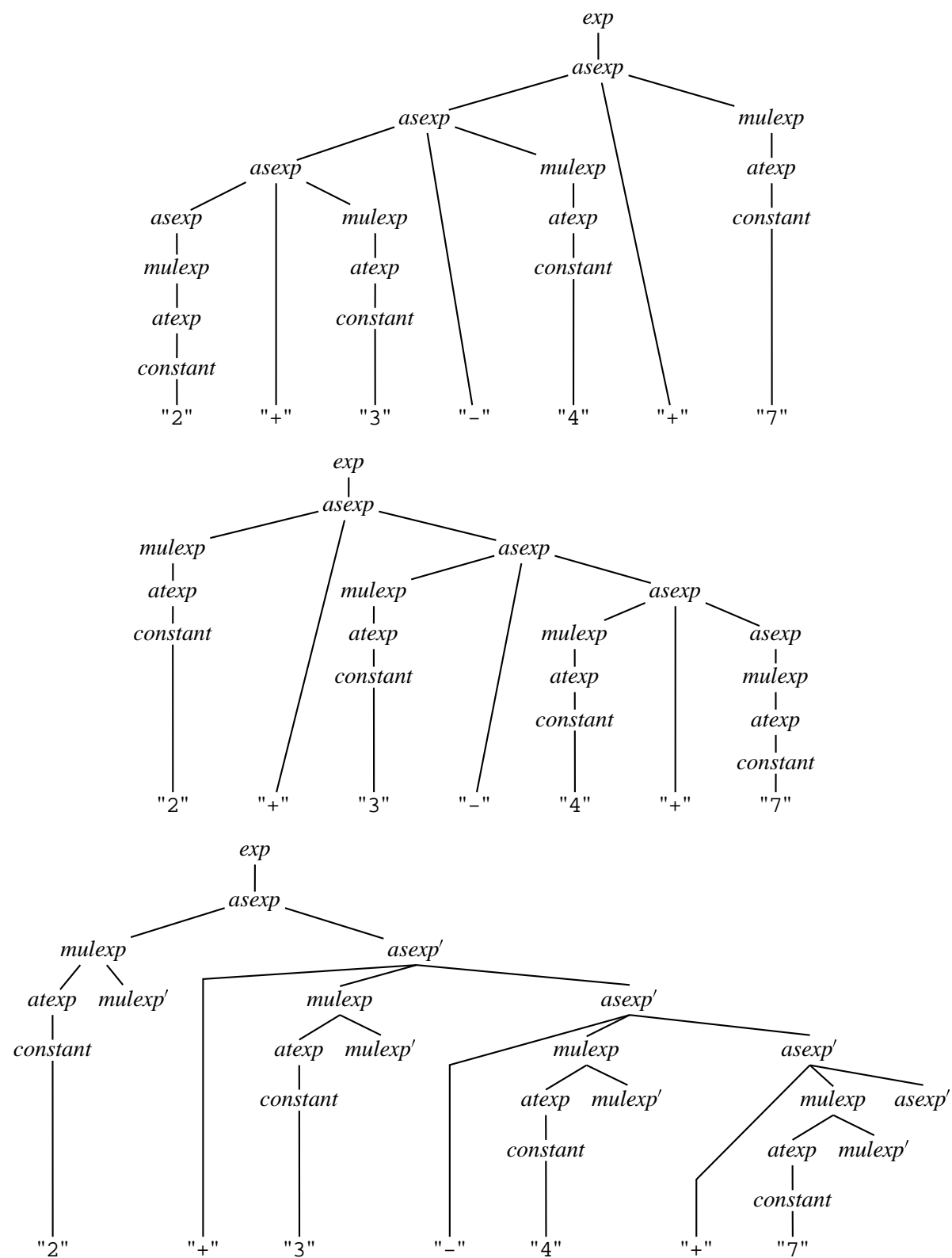
### Aufgabe 12.1: Syntaxbäume (2+2+2)



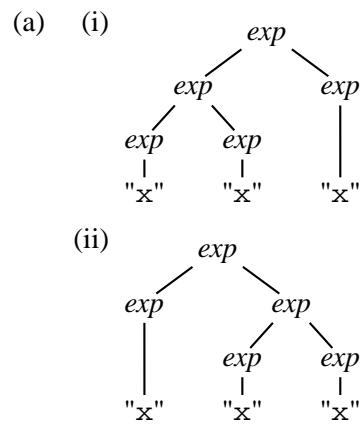
Es gibt genau einen Syntaxbaum, denn die Grammatik ist eindeutig.



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



### Aufgabe 12.2: Mehrdeutige Grammatik (8)



(b)

$exp = "x" \mid exp \ "x"$

### Aufgabe 12.3: Darstellung 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: Darstellung von Ausdrücken (8)

```

datatype ops = Add | Sub | Mul | Leq

datatype exp = Con of int | Op of exp * ops * exp

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: Ausdrücke mit Cons und Applikation (8+4+8)

(a)

```

datatype ops = Cons | Append

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

datatype token = CONS | APPEND | LPAR | RPAR
  | ID of string

fun lex' ts nil = rev ts
  | lex' ts (#" " ::cs) = lex' ts cs
  | lex' ts (#"\n"::cs) = lex' ts cs
  | lex' ts (#"␣"::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 = apex [ ("::" | "@") exp ]
apexp = atexp apex'
apexp' = [ atexp apex' ]
atexp = identifier | "(" exp ")"

```

```

(c)  (*

exp   = apexp [ ( "::" | "@" ) exp ]

apexp = atexp apexp'

apexp' = [ atexp apexp' ]

atexp = identifier | "(" 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 ops (a,a') = Op(a,ops,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

```