Fachrichtung 6.2 — Informatik
Universität des Saarlandes
Tutorenteam der Vorlesung Programmierung 1

Programmierung 1 (Wintersemester 2015/16)
Lösungsblatt: Aufgaben für die Übungsgruppen: 12

(Kapitel 13)

Hinweis: Dieses Übungsblatt enthält von den Tutoren für die Übungsgruppe erstellte Aufgaben.
Die Aufgaben und die damit abgedeckten Themenbereiche sind für die Klausur weder relevant noch irrelevant.

Lexer

Aufgabe 12.1
Wir betrachten die Sprache *abab*, die aus den beiden Schlüsselwörtern *A* und *B* und Bezeichnern besteht. Die Bezeichner dürfen aus beliebig vielen *a*s und *b*s bestehen, jedoch mit der Restriktion, dass nie zwei gleiche Buchstaben aufeinander folgen. Schreiben Sie einen Lexer, der folgende Tokens aus einer `char list` ausliest:

```
datatype token = BIGA | BIGB | ID of string
```

Lösung 12.1:

```
exception Error

fun lex nil = nil
  | lex (#"␣" :: cr) = lex cr
  | lex (#"\t" :: cr) = lex cr
  | lex (#"\n" :: cr) = lex cr
  | lex (#"A" :: cs) = BIGA::lex cs
  | lex (#"B" :: cs) = BIGB::lex cs
  | lex (c::cs) = if c=(#"a") orelse c=(#"b") then lexID [c] cs else raise Error

and lexID cs cs' = if null cs' orelse ((hd cs')<>(#"a") andalso (hd cs')<>(#"b"))
                    then ID(implode(rev cs))::lex cs'
                    else if (hd cs) = (hd cs') then raise Error
                         else lexID (hd cs'::cs) (tl cs')
```

Aufgabe 12.2 (*Was wollt ihr denn? Alle Bezeichner!*)
Bei unserem klassischen Ansatz, Schlüsselwörter direkt in `lex` zu lexen, ist es nicht einfach so möglich, Bezeichner zu erlauben, die so beginnen wie Schlüsselwörter. In SML sind aber `iff` und `lettie` gültige Bezeichner. Wie würden Sie einen Lexer aufbauen, der `if` als Schlüsselwort `IF`, `iff` jedoch als Bezeichner `ID "iff"` erkennt? Schreiben Sie einen Lexer, der folgende Tokens aus einer `char list` ausliest:

```
datatype token = TRUE | FALSE | ID of string | CON of int | AND | OR
```

Lösung 12.2:

```
fun testNext [] = true
  | testNext (x::xr) = if Char.isAlpha x then false else true

exception Error of string
datatype token = TRUE | FALSE | ID of string | CON of int | AND | OR

fun lex nil = nil
  | lex (#"␣" :: cr) = lex cr
```

```
| lex (#"\t" :: cr) = lex cr
| lex (#"\n" :: cr) = lex cr
| lex (#"t" :: #"r" :: #"u" :: #"e" :: cr) =
                          if testNext cr then TRUE :: lex cr
                          else lexId [#"e",#"u",#"r",#"t"] cr
| lex (#"f" :: #"a" :: #"l" :: #"s" :: #"e" :: cr) =
                          if testNext cr then FALSE :: lex cr
                          else lexId [#"e",#"s",#"l",#"a",#"f"] cr
| lex (#"a" :: #"n" :: #"d" :: cr) =
                          if testNext cr then AND :: lex cr
                          else lexId [#"d",#"n",#"a"] cr
| lex (#"o" :: #"r" :: cr) = if testNext cr then OR :: lex cr
                           else lexId [#"r",#"o"] cr
| lex (#"~" ::c::cr) = if Char.isDigit c then lexInt ~1 0 (c::cr)
                     else raise Error "~"
| lex (c::cr) = if Char.isAlpha c then lexId [c] cr
              else if Char.isDigit c then lexInt 1 0 (c::cr)
              else raise Error "lex"

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

Aufgabe 12.3 (*Zahlennamen*)
Im Folgenden sollen Sie einen Lexer `lex: string → int` schreiben, der aus einem String eine Zahl ausliest. Die Zahlen in dem String sind durch Ziffern in Wortdarstellung dargestellt. Die einzelnen Ziffern sind dabei durch genau ein Leerzeichen getrennt. Beispielsweise soll `lex("eins␣sieben␣sechs␣null")` nach 1760 auswerten. Ist der String keine gültige Darstellung einer Zahl, so soll eine geeignete Ausnahme geworfen werden.

(a) Schreiben Sie eine Prozedur `lex': char list → int`, die Ziffern in Wortdarstellung durch Zahlen ersetzt.

(b) Schreiben Sie eine Prozedur `diToInt: int list → int`, die eine Ziffernliste zu einer Zahl zusammensetzt.

(c) Schreiben Sie mit Hilfe der ersten beiden Teilaufgaben die Prozedur `lex`.

Lösung 12.3:

(a)
```
exception Error of string

fun lex' (#"␣" :: #"n" :: #"u" :: #"l" :: #"l" :: cr) = 0 :: lex' cr
  | lex' (#"␣" :: #"e" :: #"i" :: #"n" :: #"s" :: cr) = 1 :: lex' cr
  | lex' (#"␣" :: #"z" :: #"w" :: #"e" :: #"i" :: cr) = 2 :: lex' cr
  | lex' (#"␣" :: #"d" :: #"r" :: #"e" :: #"i" :: cr) = 3 :: lex' cr
  | lex' (#"␣" :: #"v" :: #"i" :: #"e" :: #"r" :: cr) = 4 :: lex' cr
  | lex' (#"␣" :: #"f" :: #"u" :: #"e" :: #"n" :: #"f" :: cr) = 5 :: lex' cr
  | lex' (#"␣" :: #"s" :: #"e" :: #"c" :: #"h" :: #"s" :: cr) = 6 :: lex' cr
  | lex' (#"␣" :: #"s" :: #"i" :: #"e" :: #"b" :: #"e" :: #"n" :: cr) =
                                                      7 :: lex' cr
  | lex' (#"␣" :: #"a" :: #"c" :: #"h" :: #"t" :: cr) = 8 :: lex' cr
  | lex' (#"␣" :: #"n" :: #"e" :: #"u" :: #"n" :: cr) = 9 :: lex' cr
  | lex' (#"␣" :: cr) = lex' cr
  | lex' nil = nil
  | lex' _ = raise Error "lex"
```

Das Leerzeichen zu Beginn jeder Ziffer garantiert uns, dass zwischen zwei Ziffern mindestens ein Leerzeichen ist.

(b)
```
fun diToInt nil = raise Error "NoDigits"
  | diToInt xs  = foldl (fn (x,s) => 10 * s + x) 0 xs
```

(c)
```
fun lex xs = diToInt (lex' (#"␣" :: explode xs))
```

Da wir in `lex'` vor jeder Ziffer ein Leerzeichen erwarten, fügen wir ein Leerzeichen zum Eingabestring hinzu, da wir kein Leerzeichen zu Beginn der Eingabe vorausgesetzt haben.

Rechtsklammernde Parser

Aufgabe 12.4
Wir wollen mit folgender Grammatik arbeiten: $exp ::= num \mid id \mid ''+''\ \ exp\ exp$

Dabei ist $num \in \mathbb{Z}$ und id steht für Strings, die Bezeichner darstellen sollen.

(a) Zeichnen Sie den Ableitungsbaum für $++1y+x7$. Welchem arithmetischen Ausdruck entspricht das?

(b) Konstruieren Sie einen Konstruktortyp `token`, der alle Zeichen unserer Grammatik darstellen kann.

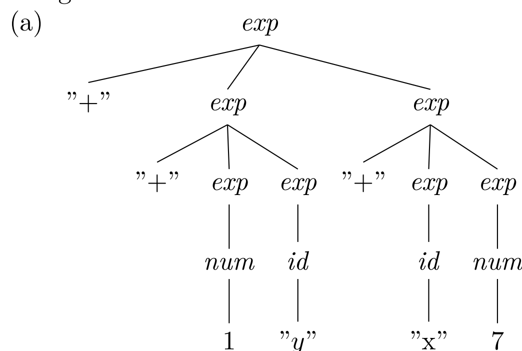
(c) Stellen Sie den arithmetischen Ausdruck $(y + 4) + 2$ als Tokenliste dar.

(d) Konstruieren Sie einen Konstruktortyp `formel`, der Formeln unserer Grammatik darstellen kann.

(e) Schreiben Sie einen Parser `parse: token list → formel`, der eine Tokenliste in seine Darstellung als `formel` überführt. Für ungültige Eingaben soll `Parse` geworfen werden.

Lösung 12.4:

(a)



Das entspricht $(1 + y) + (x + 7)$.

(b)
```
datatype token = ADD | Z of int | B of string
```

(c)
```
[ADD, ADD, B "y", Z 4, Z 2]
```

(d)
```
datatype formel = N of int | Id of string | A of formel * formel
```

(e)
```
exception Parse
fun exp (ADD::xr) = let
                        val (vorn, xr') = exp xr
                        val (hinten, xr'') = exp xr'
                    in
                        (A(vorn, hinten), xr'')
                    end
  | exp ((Z x)::xr) = (N x, xr)
  | exp ((B x)::xr) = (Id x, xr)
  | exp _ = raise Parse

fun parse xs = (case exp xs of
                    (t, nil) => t
                  |     _    => raise Parse)
```

Aufgabe 12.5 (*Binärbäume I*)
Betrachten Sie folgenden Konstruktortypen: `datatype bintree = Leaf | B of bintree * bintree`.
Schreiben Sie eine Prozedur `reprel: int list → bintree`, die aus der Prälinearisierung eines Baumes diesen rekonstruiert. Orientieren Sie sich dabei am Prinzip des Parsens und entwerfen Sie zunächst eine Prozedur `reprel': int list → bintree * int list`. Werfen Sie die Ausnahme `Domain` für ungültige Eingaben.

Lösung 12.5:

```
fun reprel' (0::nr) = (Leaf, nr)
  | reprel' (2::nr) = let val (b, br) = reprel' nr
                          val (b', br') = reprel' br
                      in (B(b, b'), br') end
  | reprel' _ = raise Domain

fun reprel xs = case reprel' xs of (b, nil) ⇒ b | _ ⇒ raise Domain
```

Aufgabe 12.6
Wir betrachten Ausdrücke mit Fakultäten. Alle Ausdrücke beginnen mit dem Fakultätszeichen. Die Fakultät klammert am stärksten, weswegen !(!5) gültig ist, !!5 hingegen nicht. Gegeben ist folgender Konstruktortyp:

```
datatype exp = F of exp | Icon of int
```

(a) Schreiben Sie die phrasale Syntax für die oben beschriebene Grammatik.

(b) Schreiben Sie einen Parser für Ihre Syntax.

Lösung 12.6:

$$exp ::= \text{“!”}\ pexp$$
$$pexp ::= num \mid \text{“(”}\ exp\ \text{“)”}$$

```
fun exp (FAK::tr) = let val (e, ts) = pexp tr in (F(e), ts) end
  | exp _ = raise Error "Match"
and pexp ((ICON i)::tr) = (Icon i, tr)
  | pexp (LPAR::tr) = (case exp tr of (e, RPAR::ts) ⇒ (e, ts)
                                    |       _       ⇒ raise Error "Match")
  | pexp _ = raise Error "pexp"
```

Aufgabe 12.7 (*Große Arithmetik*)
Wir wollen mit folgender Grammatik arbeiten:

$$\begin{aligned} exp &::= mexp\ [''+''\ exp] \\ mexp &::= pexp\ [''*''\ mexp] \\ pexp &::= num \mid id \mid ''(''\ exp\ '')'' \end{aligned}$$

Dabei ist $num \in \mathbb{Z}$ und id steht für Strings, die Bezeichner darstellen sollen.

(a) Geben Sie den Ableitungsbaum von $(1 * x) * (y + z)$ in der Grammatik an.

(b) Konstruieren Sie einen Konstruktortyp `token`, der alle Zeichen unserer Grammatik darstellen kann.

(c) Stellen Sie den Ausdruck $(1 * x) * ((3 + 2) + y)$ als Tokenliste dar.

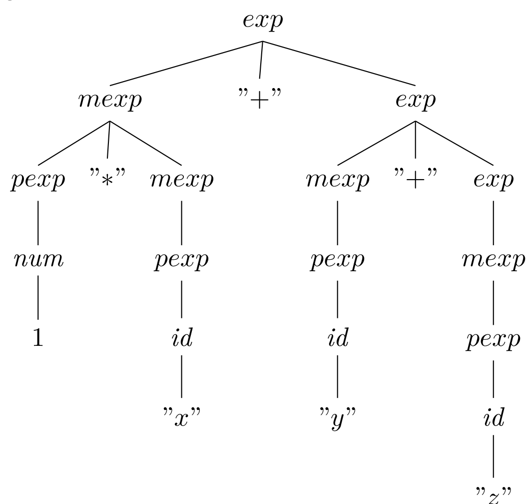
(d) Konstruieren Sie einen Konstruktortyp `formel`, der Formeln unserer Grammatik darstellen kann.

(e) Stellen Sie den Ausdruck $(1 * x) * ((3 + 2) + y)$ als `formel` dar.

(f) Schreiben Sie einen Parser `parse: token list → formel`, der eine Tokenliste in seine Formel übersetzt.

Lösung 12.7:

(a)



(b) `datatype token = ADD | MULTI | Z of int | B of string | LPAR | RPAR`

(c)
```
[LPAR, Z 1, MULTI, B "x", RPAR, MULTI, LPAR, LPAR, Z 3, ADD, Z 2,
RPAR, ADD, B "y", RPAR]
```

(d)
```
datatype formel = A of formel * formel | M of formel * formel |
                  N of int | V of string
```

(e) `M(M(N 1, V "x"), A(A(N 3, N 2), V "y"))`

(f)
```
exception Parse
fun exp xs = (case mexp xs of
                 (vorn, ADD::xr) => let val (hinten, xr') = exp xs in
                                        (A(vorn, hinten), xr') end
               | s => s)
and mexp xs = (case pexp xs of
                  (vorn, MULTI::xr) => let val (hinten, xr') = mexp xs in
                                           (M(vorn, hinten), xr') end
                | s => s)
and pexp ((Z x)::xr) = (N x, xr)
  | pexp ((B x)::xr) = (V x, xr)
  | pexp (LPAR::xr) = (case exp xr of
                          (t, RPAR::xr') => (t, xr')
                        | _ => raise Parse)
| pexp _ = raise Parse

fun parse xs = (case exp xs of
                   (t, nil) => t
                 |    _     => raise Parse)
```

Aufgabe 12.8 (*Na logisch!*)
Wir wollen einen Parser für logische Ausdrücke schreiben. Logische Ausdrücke bestehen aus Bezeichnern, *true*, *false*, Verundungen ($\wedge$), Veroderungen ($\vee$), Negationen ($\neg$) und Implikationen ($\rightarrow$). Negationen sind unär, sollen gestapelt werden können ($\neg\neg(\neg true)$) und klammern am stärksten, Verundungen klammern stärker als Veroderungen, und Implikationen klammern so stark wie Veroderungen. Alle Operatoren sind rechtsassoziativ. Es soll Klammern geben.

(a) Stellen Sie eine Grammatik für logische Ausdrücke auf.

(b) Konstruieren Sie einen Konstruktortyp `token`, der alle Zeichen Ihrer Grammatik darstellen kann.

(c) Konstruieren Sie einen Konstruktortyp `log`, der logische Ausdrücke darstellt.

(d) Schreiben Sie einen Parser `parse: token list` $\to$ `log` für logische Ausdrücke.

Lösung 12.8:

(a)

exp	$::=$ *mexp* [$''\rightarrow''$ *exp*] | *mexp* [$''\vee''$ *exp*]
mexp	$::=$ *vexp* [$''\wedge''$ *mexp*]
vexp	$::=''\neg''$ *vexp* | *pexp*
pexp	$::=$ *id*|*true*|*false*|$''(''$ *exp* $'')''$

(b)
```
datatype token = IMP | OR | AND | NEG | TRUE | FALSE | ID of string | LPAR | RPAR
```

(c)
```
datatype log = Imp of log * log | Or of log * log | And of log * log |
               Neg of log | True | False | Id of string
```

(d)
```
exception Parse
fun exp xs = (case mexp xs of
                  (vorn, IMP::xr) => let val (hinten, xr') = exp xr in
                                         (Imp(vorn, hinten), xr') end
                | (vorn, OR::xr) => let val (hinten, xr') = exp xr in
                                        (Or(vorn, hinten), xr') end
                | s => s)
and mexp xs = (case vexp xs of
                  (vorn, AND::xr) => let val (hinten, xr') = mexp xr in
                                         (And (vorn, hinten), xr') end
                | s => s)
and vexp (NEG::xr) = let val (t, xr') = vexp xr in (Neg t, xr') end
  | vexp xs = pexp xs
and pexp ((ID x)::xr) = (Id x, xr)
  | pexp (TRUE::xr) = (True, xr)
  | pexp (FALSE::xr) = (False, xr)
  | pexp (LPAR::xr) = (case exp xr of
                          (t, RPAR::xr') => (t, xr')
                        | _ => raise Parse)
  | pexp _ = raise Parse

fun parse xs = (case exp xs of
                    (t, nil) => t
                  | _ => raise Parse)
```

Aufgabe 12.9 (*Tupel. Oder Listen?*)

Wir wollen mit folgender Grammatik für Bezeichner und Tupel arbeiten:

exp	$::=$ *id* | $''(''$ [*row*] $'')''$
row	$::=$ *exp* [$'',''$ *row*]

(a) Kann die Grammatik geschachtelte Tupel wie $((x, y), z)$ darstellen?

(b) Warum ist *row* in *exp* optional?

(c) Konstruieren Sie einen Konstruktortyp `token`, der alle Zeichen der Grammatik darstellen kann.

(d) Konstruieren Sie einen Konstruktortyp `value`, der Bezeichner und Tupel darstellt.

(e) Schreiben Sie einen Parser `parse: token list` $\to$ `value`, der eine Tokenliste in ihren Wert überführt.

Lösung 12.9:

(a) Ja, da *row* Zugriff auf *exp* hat.

(b) Dadurch kann das leere Tupel, (), zurückgegeben werden.

(c)
```
datatype token = LPAR | RPAR | B of string | COMMA
```

(d) `datatype value = V of string | T of value list`

(e)
```
fun exp ((B x)::xr) = (V x, xr)
  | exp (LPAR::RPAR::xr) = (T nil, xr)
  | exp (LPAR::xr) = (case row xr of
                           (t, RPAR::xr') => (t, xr')
                         | _ => raise Parse)
  | exp _ = raise Parse
and row xs = (case exp xs of
                  (x, COMMA::xr) => let val (T liste, xr') = row xr in
                                         (T (x::liste), xr') end
                | (x, xr) => (T [x], xr))

fun parse xs = (case exp xs of
                    (t, nil) => t
                  |     _    => raise Parse)
```

Aufgabe 12.10 (*Recht abstrakt*)
Stellen Sie eine Grammatik für die wie folgt definierten Ausdrücke auf:

- Es gibt die Operatoren A und B.
- A ist ein unärer Operator. Ist φ ein gültiger Ausdruck, so ist es auch $A\ \varphi$.
- B ist ein binärer Infix-Operator. Sind φ und ψ gültige Ausdrücke, so ist es auch $\varphi\ B\ \psi$.
- A klammert stärker als B.
- B klammert implizit rechts.
- 1 ist immer ein gültiger Ausdruck.
- Ausdrücke dürfen Klammern enthalten, ist also φ ein gültiger Ausdruck, so ist es auch (φ).

Schreiben Sie anschließend einen Parser für Ihre Grammatik mit dem folgenden Konstruktortyp `exp` und den angegebenen Tokens:

```
datatype token = OpA | OpB | Eins
datatype exp = A of exp | B of exp * exp | ConEins
```

Lösung 12.10:
Wie geht man nun vor? Der schwächste Operator steht in der syntaktischen Kategorie am weitesten oben, d. h. in der ersten Kategorie, die man definiert. In unserem Fall ist dies das B.

$$bexp ::= aexp\ [\text{“B”}\ bexp]$$

Es ist angegeben, dass B implizit rechts klammert. Intuitiv bedeutet das: Rechts kann wieder ein Ausdruck, der mit *bexp* (also einer Kategorie, die gleich stark klammert) gebildet wurde, vorkommen, ohne dass wir ihn direkt klammern müssten. Deshalb stehen die Optionalklammern rechts. Links dagegen muss ein stärker geklammerter (also weiter unten vorkommender) Ausdruck stehen.

Unäre Operatoren sind sehr einfach:

$$\begin{aligned} bexp &::= aexp\ [\text{“B”}\ bexp] \\ aexp &::= [\text{“A”}]\ pexp \end{aligned}$$

Entweder sie kommen vor einem beliebigen Ausdruck oder eben nicht. Deshalb enthalten die Optionalklammern hier nur den Operator selbst. Würde der Operator nicht vor, sondern nach einem Ausdruck stehen, so würde die Optionalklammer rechts stehen. Da aber keine syntaktischen Kategorien in den Optionalklammern vorkommen, man also am ersten Wort eines Satzes entscheiden kann, ob es zur Kategorie gehört oder weiter mit *pexp* verfahren werden soll, spielt dies für die RA-Tauglichkeit keine Rolle.

Die Kategorie *pexp* enthält Konstanten und Bezeichner, so wie eine geklammerte Darstellung der „obersten“ Kategorie. Diese sorgt dafür, dass ein schwacher Ausdruck (bei uns *bexp*) in der am stärksten klammerten Kategorie stehen kann.

$$\begin{aligned} bexp &::= aexp\ [\text{“B”}\ bexp] \\ aexp &::= [\text{“A”}]\ pexp \\ pexp &::= \text{“1”} \mid \text{“(”}\ bexp\ \text{“)”} \end{aligned}$$

Einen Parser zu einer Grammatik schreiben ist sehr mechanisch. Pro Kategorie gibt es eine Prozedur und alle Prozeduren werden mit **and** verbunden. Das Grundgerüst zu der Grammatik oben sieht also so aus:

```
fun bexp ts = ...
and aexp ts = ...
and pexp ts = ...
```

Jetzt muss man die richtigen Fallunterscheidungen machen: Im Fall von `bexp` heißt das: Beginnt das, was man von `aexp` zurückerhält mit einem B, so muss man die restliche Liste nochmals mit `bexp` parsen und gibt dies zurück, beginnt die Liste dahingegen mit etwas anderen, so ist man fertig. Dies lässt sich am einfachsten durch ein **case** ausdrücken:

```
fun bexp ts = case aexp ts of
      (a1, OpB::tr) => let val (a2,tr') = bexp tr
                       in  (B(a1,a2),tr') end
    | s => s

and aexp ts = ...

and pexp ts = ...
```

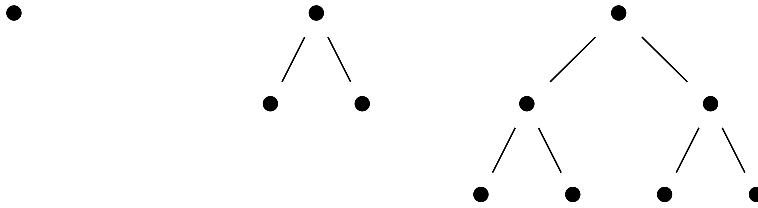
In `aexp` muss man unterscheiden, ob die Liste nun mit dem Operator A beginnt oder eben nicht. Beginnt sie damit, so wird vor den restlichen Ausdruck ein A gesetzt – ansonsten nicht.

```
fun bexp ts = case aexp ts of
      (a1, OpB::tr) => let val (a2,tr') = bexp tr
                       in  (B(a1,a2),tr') end
    | s => s

and aexp (OpA::tr) = let val (a, tr') = pexp tr
                     in  (A a, tr') end
  | aexp ts = pexp ts

and pexp ts = ...
```

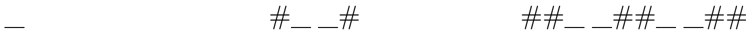
Die Kategorie *pexp* ist in fast allen Parsern sehr ähnlich: Sie enthält, falls es Klammern in den Ausdrücken gibt und die Klammern in *pexp* vorkommen, einen Aufruf mit *match* und das Parsen der Konstanten, in unserem Fall also nur die 1. Gleichzeitig wirft Sie einen Fehler, wenn etwas nicht so geparst werden kann. Intuitiv werden alle Fehler nach unten durchgegeben, da in allen Fällen, die in den anderen Prozeduren nicht behandelt werden können, einfach die nächst-tieferliegende aufgerufen wird.

```
fun bexp ts = case aexp ts of
      (a1, OpB::tr) => let val (a2,tr') = bexp tr
                       in  (B(a1,a2),tr') end
    | s => s

and aexp (OpA::tr) = let val (a, tr') = pexp tr
                     in  (A a, tr') end
  | aexp ts = pexp ts

and pexp (Eins::tr) = (ConEins, tr)
  | pexp (LPAR::tr) = match (bexp tr) RPAR
  | pexp s = raise Error "pexp"
```

Aufgabe 12.11 (*Binärbäume II*)
Binäre Bäume können wir mithilfe von # und _ darstellen. Die folgenden drei Bäume



können wir auch so darstellen:



(a) Machen Sie sich den Zusammenhang klar und schreiben Sie sich bei Bedarf ein paar Beispiele auf.

(b) Stellen Sie eine passende phrasale Syntax auf.

(c) Zeichnen Sie Syntaxbäume für Ihre phrasale Syntax und folgende Baumdarstellungen:

- _
- #_ _#
- ##_ _##_ _##
- # ##_ _##_ _## ##_ _##_ _## #

(d) Deklarieren Sie sich einen passenden Konstruktortyp `token` und schreiben Sie einen Lexer `lex: char list` → `token list`, der die alternative Baumdarstellung in Form einer `char list` erhält und die Wortdarstellung in Form einer `token list` ausgibt.

(e) Deklarieren Sie einen Konstruktortyp `bintree` (die abstrakte Syntax in SML), der es Ihnen erlaubt, die alternative Darstellung in SML umzusetzen.

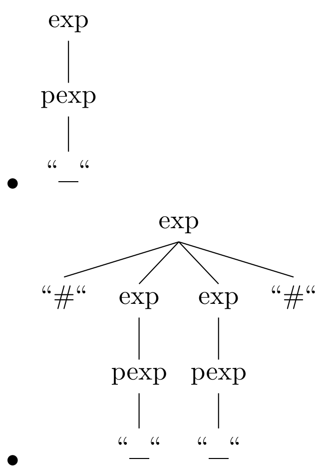
(f) Schreiben Sie nun einen Parser `parse: token list` → `bintree`. Deklarieren Sie zunächst passende Hilfsprozeduren, die die Überführung in die Baumdarstellung (`bintree`) übernehmen.

Lösung 12.11:

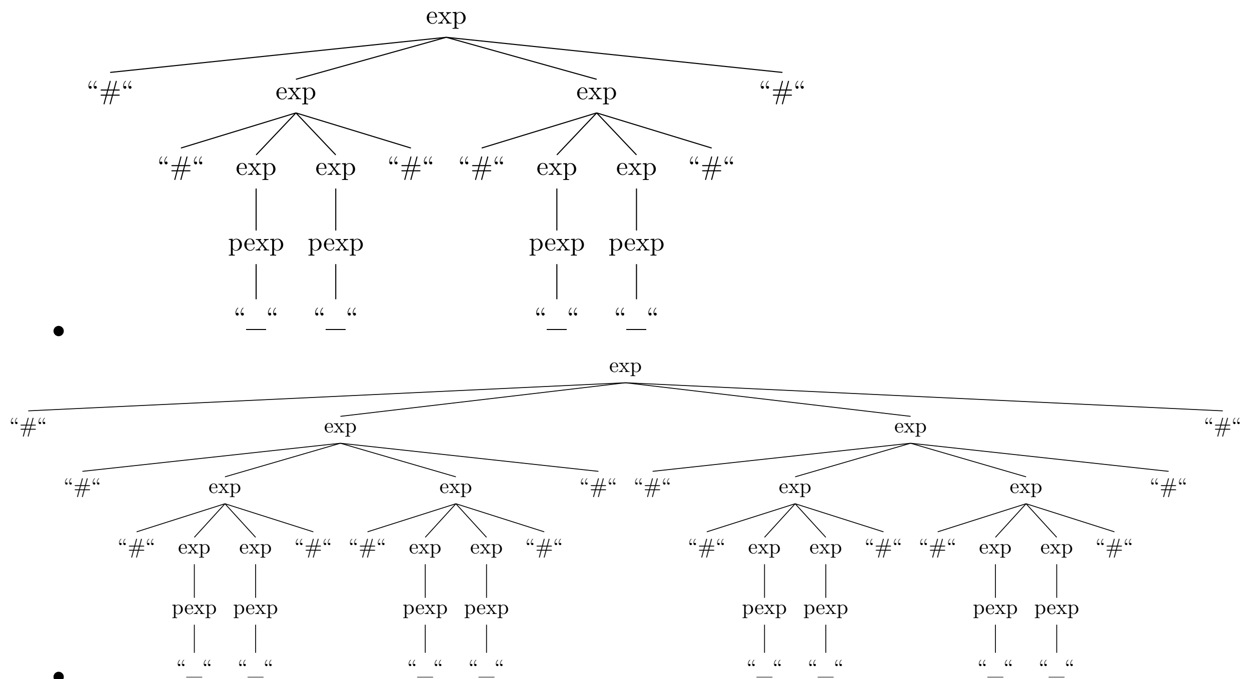
(a)

exp	::=	"#" *exp exp* "#" | *pexp*
pexp	::=	"_"

(b) Folgende Syntaxbäume:

- 

- exp
  “#“ exp exp “#“
  pexp pexp
  “_“ “_“



(c)
```
datatype token = HASHTAG | UNDERSCORE

fun lex nil = nil
  | lex (#"␣" ::cr) = lex cr
  | lex (#"\t" ::cr) = lex cr
  | lex (#"\n" ::cr) = lex cr
  | lex (#"#" :: cr) = HASHTAG :: lex cr
  | lex (#"_" :: cr) = UNDERSCORE :: lex cr
  | lex _ = raise Error "match"
```

(d)
```
datatype bintree = Atree | Btree of bintree * bintree
```

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

fun exp (HASHTAG::ts) = let
                           val (e, tr) = exp ts
                           val (e', tr') = exp tr
                        in
                           match (Btree(e, e'), tr') HASHTAG
                        end
  | exp s = pexp s

and pexp (UNDERSCORE :: ts) = (Atree, ts)
  | pexp _ = raise Error "parse"

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

Linksklammernde Parser

Aufgabe 12.12
Gegeben sei folgende Grammatik und der Konstruktortyp
datatype `token = OpPlus | OpMinus | RPAR | LPAR | EINS`

plusexp	::=	[*plusexp* "+"] *unminusexp*
unminusexp	::=	["−"] *pexp*
pexp	::=	"1" | "(" *plusexp* ")"

(a) Die gegebene Grammatik ist nicht RA-tauglich. Erklären Sie wieso.

(b) Bilden Sie nach dem bekannten Schema eine RA-taugliche Grammatik.

(c) Zeichnen Sie einen Syntaxbaum nach der originalen Grammatik für den Ausdruck $1 + (1 + -1) + 1$.

(d) Geben Sie einen Konstruktortyp `exp` an, mit dem Sie Ausdrücke der Grammatik darstellen können.

(e) Schreiben Sie den Ausdruck $-1 + 1 + -(-1 + 1)$ als Liste von Token.

(f) Schreiben Sie einen Parser `parse: token list` → `exp`, der Ausdrücke gemäß der Grammatik parst.

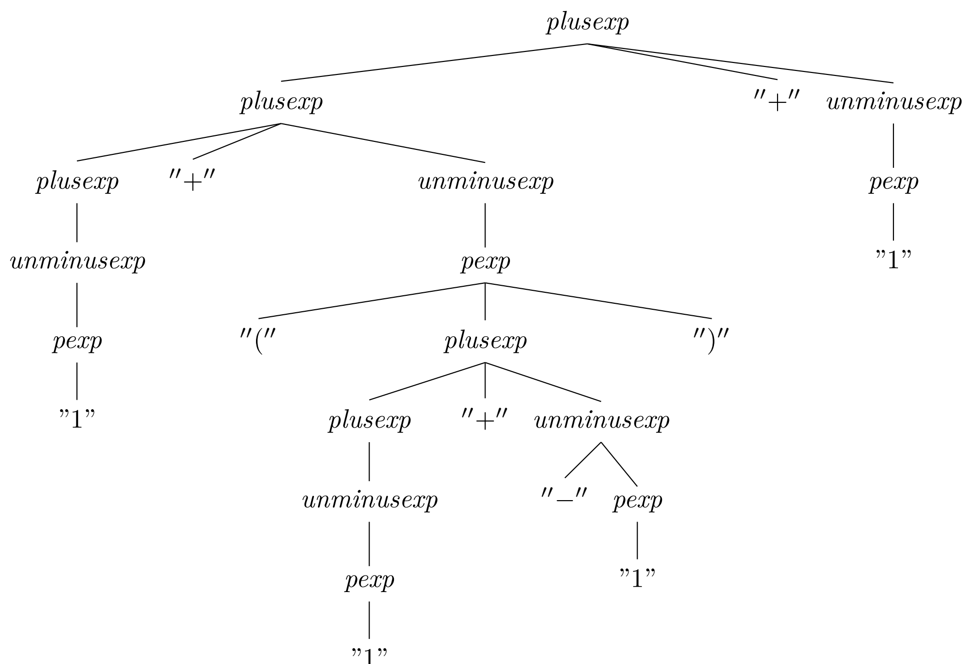
Lösung 12.12:

(a) Die Grammatik ist nach der Definition aus dem Buch nicht RA-tauglich, da der zweite Teil der Definition verletzt wird. Siehe Buch S. 265.

(b)

$$
\begin{aligned}
plusexp &::= unminusexp\ plusexp' \\
plusexp' &::= [\text{“+”}\ unminusexp\ plusexp'] \\
unminusexp &::= [\text{“−”}]\ pexp \\
pexp &::= \text{“1”} \mid \text{“(”}\ plusexp\ \text{“)”}
\end{aligned}
$$

(c)



(d)
```
datatype exp   = Plus of exp * exp | Minus of exp | ConEins
```

(e) `[OpMinus,EINS,OpPlus,EINS,OpPlus,OpMinus,LPAR,OpMinus,EINS,OpPlus,EINS,RPAR]`

(f)
```
exception Error of string

fun plusexp ts = plusexp' (unminusexp ts)
```

```
and plusexp' (a, OpPlus::tr) =
            let
               val (a', tr') = unminusexp tr
            in
               plusexp'(Plus(a, a'),tr')
            end
| plusexp' s = s

and unminusexp (OpMinus::tr) = let val (a, tr') = pexp tr
                               in (Minus a, tr') end
  | unminusexp ts = pexp ts

and pexp (EINS::tr) = (ConEins, tr)
  | pexp (LPAR::tr) = (case plusexp tr of
                          (a, RPAR::tr') => (a, tr')
                        |       _        => raise Error "parse")
  | pexp s = raise Error "pexp"

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

Aufgabe 12.13 (*Reguläre Ausdrücke*)
Es gibt drei Operatoren für reguläre Ausdrücke: Kleene'scher Stern $*$, Konkatenation $\circ$ und Vereinigung $+$.

- $*$ klammert stärker als $\circ$ und $\circ$ stärker als $+$ (und somit klammert $*$ auch stärker als $+$).
- $+$ und $\circ$ klammern implizit links (wie die arithmetischen Operatoren auch).
- $*$ ist ein unärer Operator, d. h. ist φ ein regulärer Ausdruck, so ist φ^* auch ein regulärer Ausdruck.
- $+$ und $\circ$ sind binäre Operatoren, d. h. sind φ und ψ reguläre Ausdrücke, so sind es auch $\varphi + \psi$ und $\varphi \circ \psi$.

Wortfolgen werden mit den folgenden Token dargestellt:

```
datatype token = STERN | PLUS | KRINGEL | CON of string | LPAR | RPAR
```

Gültige Ausdrücke wären z.B. $A \circ B + B \circ A$, A^*, $A + B$ und $(A + B)^*$. Konstanten sind also beliebige Strings.

(a) Stellen Sie eine links-rekursive Grammatik für reguläre Ausdrücke auf. Sie dürfen die Kategorie *con* verwenden, die für einen beliebigen String steht.

(b) Machen Sie Ihre Grammatik RA-tauglich.

(c) Schreiben Sie einen Parser zu Ihrer Grammatik. Verwenden Sie dabei den folgenden Konstruktortyp `exp`:

```
datatype exp = Stern of exp | Plus of exp * exp
             | Kringel of exp * exp | Con of string
```

Lösung 12.13:

(a)

$$\begin{aligned}
\textit{plusexp} &::= [\textit{plusexp}\ \text{“}+\text{”}]\ \textit{kringelexp} \\
\textit{kringelexp} &::= [\textit{kringelexp}\ \text{“}\circ\text{”}]\ \textit{sternexp} \\
\textit{sternexp} &::= \textit{pexp}\ [\text{“}*\text{”}] \\
\textit{pexp} &::= \textit{con} \mid \text{“(”}\textit{plusexp}\text{“)”}
\end{aligned}$$

Hierbeit bezeichnet *con* einen beliebigen String.

(b)

$$\begin{aligned}
plusexp &::= kringelexp\ plusexp' \\
plusexp' &::= [\text{“}+\text{”}\ kringelexp\ plusexp'] \\
kringelexp &::= sternexp\ kringelexp' \\
kringelexp' &::= [\text{“}\circ\text{”}\ sternexp\ kringelexp'] \\
sternexp &::= pexp\ [\text{“}*\text{”}] \\
pexp &::= con \mid \text{“(”}plusexp\text{“)”}
\end{aligned}$$

Hierbei bezeichnet *con* einen beliebigen String.

(c)
```
fun plusexp ts = plusexp' (kringelexp ts)

and plusexp' (e, PLUS::tr) = plusexp' (extend (e,tr) kringelexp Plus)
  | plusexp'       s       = s

and kringelexp ts = kringelexp' (sternexp ts)

and kringelexp' (e, KRINGEL::tr) =
                            kringelexp' (extend (e,tr) sternexp Kringel)
  | kringelexp' s = s

and sternexp ts = case pexp ts of (a, STERN::tr) => (Stern(a),tr)
                                | s => s

and pexp (CON z :: tr) = (Con z, tr)
  | pexp (LPAR :: tr) = match (plusexp tr) RPAR
  | pexp _  = raise Error "pexp"
```

Alternative Deklarationen ohne *match* und *extend*:

```
...
and plusexp' (e, PLUS::tr) = plusexp' (case (kringelexp tr) of
                                          (e', tr') => (Plus (e, e')), tr')
  | plusexp' s = s

...
and pexp (CON z :: tr) = (Con z, tr)
  | pexp (LPAR :: tr)  = (case (plusexp tr) of
                               (a, RPAR :: tr') => (a, tr')
  | s => raise Error "match")
  | pexp  _ = raise Error "pexp"
```

Aufgabe 12.14 (*Ganz abstrakt*)
Zusätzlich zu Bezeichnern und expliziten Klammern seien folgende Operatoren gegeben: ♣, ♡ und ♠.

- ♡ und ♣ sind binäre Infix-Operatoren und ♠ ein unärer Operator.
- ♣ und ♡ klammern jeweils links.
- ♡ klammert stärker als ♣ und ♠ klammert am stärksten.

(a) Erstellen Sie zunächst eine Grammatik, welche die obigen Bedingungen erfüllt.

(b) Geben Sie einen Ausdruck nach unserer Grammatik an, in der jeder Operator einmal vorkommt, und zeichnen Sie dafür einen Syntaxbaum.

(c) Erstellen Sie einen Konstruktortyp `token` und einen Konstruktortyp `sign` für die Baumdarstellung der Ausdrücke.

(d) Schreiben Sie nun einen Parser `parse: token list → sign`. Bei einer ungültigen Eingabe soll die Ausnahme `Parse` geworfen werden.

Lösung 12.14:

(a)

$$\begin{aligned} kreuz &::= [kreuz\ ''\clubsuit'']herz \\ herz &::= [herz\ ''\heartsuit'']schippe \\ schippe &::= [''\spadesuit'']prim \\ prim &::= id\ |''(''kreuz'')'' \end{aligned}$$

RA-taugliche Grammatik mit Hilfskategorien:

$$\begin{aligned} kreuz &::= herz\ kreuz' \\ kreuz' &::= [''\clubsuit''\ herz\ kreuz'] \\ herz &::= schippen\ herz' \\ herz' &::= [''\heartsuit''\ schippen\ herz'] \\ schippen &::= [''\spadesuit'']prim \\ prim &::= id\ |''(''\ kreuz\ '')'' \end{aligned}$$

(b)

(c) `datatype token = HERZ | KREUZ | PIK | ID of string | LPAR | RPAR`

(d) `datatype sign = Herz of sign*sign | Kreuz of sign*sign | Pik of sign | Id of string`

(e)
```
exception Parse

fun kreuz ts = kreuz'(herz ts)

and kreuz' (a,KREUZ::tr) = let val (a',tr') = herz tr
                           in kreuz'(Kreuz(a,a'), tr') end
  | kreuz' s = s

and herz ts = herz'(schippen ts)

and herz' (a, HERZ::tr) = let val (a',tr') = schippen tr
                          in herz'(Herz(a,a'), tr') end
  | herz' s = s

and schippen (PIK::tr) = let val (a,tr') = prim tr
                         in (Pik a, tr') end

   | schippen s = prim s

and prim ((ID x)::tr) = (Id x, tr)
  | prim (LPAR::tr) = (case exp tr of
                          (a, RPAR::tr') => (a,tr')
                        |       _        => raise Parse)
  | prim _ => raise Parse

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