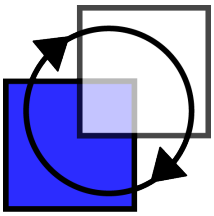
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Prof. Bernd Finkbeiner, Ph.D.
Jana Hofmann, M.Sc.
Reactive Systems Group



Programmierung 1 (WS 2020/21)
Aufgaben für die Übungsgruppe K (Lösungsvorschläge)

Hinweis: Diese Aufgaben wurden von den Tutoren für die Übungsgruppe erstellt. Sie sind für die Klausur weder relevant noch irrelevant. 🤔 markiert potentiell schwerere Aufgaben.

Konkrete Syntax

Lexer

Aufgabe TK.1 *(abab)*
Wir betrachten die Sprache *abab*, die aus den beiden Schlüsselwörtern A und B und Bezeichnern besteht. Die Bezeichner dürfen aus beliebig vielen as und bs 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
```

Lösungsvorschlag TK.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 TK.2 *(Bezeichner)*

(a) Schreiben Sie einen Lexer `lex: char list → token list`, der durch Leerzeichen getrennte Bezeichner erkennt. Ein Bezeichner muss mit einem Buchstaben beginnen und kann danach aus Ziffern und Buchstaben bestehen.

(b) Erweitern Sie `lex` so, dass zusätzlich auch durch Leerzeichen getrennte Zahlen erkannt werden. Eine Zahl besteht aus mindestens einer Ziffer und kann mit dem Vorzeichen "-" beginnen.

(c) Erweitern Sie `lex` um die durch Leerzeichen getrennten Schlüsselwörter "plus", "mal", "l. Klammer" und "r. Klammer". Achten Sie darauf, dass Wörter wie "plus1" oder "malt" weiterhin gültige Bezeichner sind.

Lösungsvorschlag TK.2

(a)

```
exception Error
datatype token = Id of string

fun lex [] = []
  | lex (#"␣" :: cr) = lex cr
  | lex (c :: cr) = if Char.isAlpha c then chars [] (c :: cr) else raise Error

and chars xs nil = [Id (implode (rev xs))]
  | chars xs (#"␣" :: cr) = Id (implode (rev xs)) :: lex cr
  | chars xs (c :: cr) =
      if Char.isDigit c orelse Char.isAlpha c then chars (c :: xs) cr else raise Error
```

(b)

```
exception Error
datatype token = Num of int | Id of string

fun lex [] = []
  | lex (#"␣" :: cr) = lex cr
  | lex (#"~" :: c  :: cr) = if Char.isDigit c then numbers ~1 0 (c :: cr)
                                               else raise Error
  | lex (c :: cr) =
      if Char.isAlpha c then chars [] (c :: cr)
                        else if Char.isDigit c then numbers 1 0 (c :: cr)
                                               else raise Error
and chars xs nil = [Id (implode (rev xs))]
  | chars xs (#"␣" :: cr) = Id (implode (rev xs)) :: lex cr
  | chars xs (c :: cr) = if Char.isDigit c orelse Char.isAlpha c then chars (c :: xs) cr
                                                                 else raise Error
and numbers d n nil = [Num (d * n)]
  | numbers d n (#"␣" :: cr) = Num (d * n) :: lex cr
  | numbers d n (c :: cr) = if Char.isDigit c then numbers d (n * 10 + ord c - 48) cr
                                              else raise Error
```

(c)

```
exception Error
datatype token = Num of int | Id of string | ADD | MUL | LPAR | RPAR

fun endOfKW cl = null cl orelse hd cl = #"␣"

fun lex [] = []
| lex (#"␣" :: cr) = lex cr
| lex (#"p" :: #"l" :: #"u" :: #"s" :: cr) = if endOfKW cr then ADD :: lex cr
                                             else chars (#"s" :: #"u" :: #"l" :: #"p" :: nil) cr
| lex (#"m" :: #"a" :: #"l" :: cr) = if endOfKW cr then MUL :: lex cr
                                                   else chars (#"l" :: #"a" :: #"m" :: nil) cr
| lex (#"l" :: #"." :: #"␣" :: #"K" :: #"l" :: #"a" :: #"m" :: #"m" :: #"e" :: #"r" ::cr)
                 = if endOfKW cr then LPAR :: lex cr else raise Error
| lex (#"r" :: #"." :: #"␣" :: #"K" :: #"l" :: #"a" :: #"m" :: #"m" :: #"e" :: #"r" ::cr)
                 = if endOfKW cr then RPAR :: lex cr else raise Error
| lex (#"~" :: c  :: cr) = if Char.isDigit c then numbers ~1 0 (c :: cr)
                                             else raise Error
| lex (c :: cr) = if Char.isAlpha c then chars [] (c :: cr)
                                   else if Char.isDigit c then numbers 1 0 (c :: cr)
                                                          else raise Error
and chars xs nil = [Id (implode (rev xs))]
  | chars xs (#"␣" :: cr) = Id (implode (rev xs)) :: lex cr
  | chars xs (c :: cr) = if Char.isDigit c orelse Char.isAlpha c then chars (c :: xs) cr
                                                                 else raise Error
and numbers d n nil = [Num (d * n)]
  | numbers d n (#"␣" :: cr) = Num (d * n) :: lex cr
  | numbers d n (c :: cr) = if Char.isDigit c then numbers d (n * 10 + ord c - 48) cr
                                              else raise Error
```

Aufgabe TK.3 *(Call me Lexer)*

Schreiben Sie einen Lexer `numbers : string → int list`, sodass `numbers "123␣456␣␣␣789"` zu `[123, 456, 789]` auswertet. Geben Sie zuerst eine lexikalische Grammatik an. Falls der Eingabestring einen anderen Buchstaben außer Ziffern und Leerzeichen enthält, soll eine Ausnahme geworfen werden.

Sie dürfen die Prozedur `Char.isDigit` verwenden.

Lösungsvorschlag TK.3

$$\begin{aligned} \mathit{word} &::= \mathit{num}\ [\mathit{word}] \\ \mathit{num} &::= \mathit{digit}\ [\mathit{num}] \\ \mathit{digit} &::= 0 \mid \cdots \mid 9 \end{aligned}$$

```
1  exception Error
2
3  fun numbers s = numbers'(explode s)
4  and numbers' [] = []
5    | numbers' (#"␣" :: cr) = numbers' cr
6    | numbers' (c :: cr) = if Char.isDigit c then numbers'' 0 (c :: cr) else raise Error
7  and numbers'' n nil = [n]
8    | numbers'' n (#"␣" :: cr) = n :: numbers' cr
9    | numbers'' n (c :: cr) = if Char.isDigit c then numbers'' (n * 10 + ord c - ord #"0") cr
10                                               else raise Error
```

Grammatiken und Parser

Aufgabe TK.4 *(Parser verstehen)*
Diese Aufgabe soll Ihnen helfen, die Funktionsweise von Parsern zu verstehen. Betrachten wir hierfür den Parser aus der Zusatzerklärung:

```
exception Error of string

datatype token = LPAR | RPAR | NEG | AND | IMP | TRUE | FALSE

datatype exp = Neg of exp
             | And of exp * exp | Imp of exp * exp
             | True | False

fun exp ts = (case (uexp ts) of (e, IMP :: tr) ⇒ let val (e', tr') = exp tr
                                                 in (Imp (e, e'), tr')
                                                 end
                              | (e, ts') ⇒ (e, ts'))
and uexp ts = uexp' (mexp ts)
and uexp' (e, AND :: tr) = let val (e', tr') = mexp tr
                           in uexp' (And (e, e'), tr')
                           end
  | uexp' s = s
and mexp (NEG :: tr) = let val (e, ts) = aexp tr
                       in (Neg e, ts)
                       end
  | mexp ts = aexp ts
and aexp (TRUE  :: tr) = (True, tr)
  | aexp (FALSE :: tr) = (False, tr)
  | aexp (LPAR  :: tr) = (case (exp tr) of (a, RPAR :: tr') ⇒ (a, tr')
                                         | _ ⇒ raise Error "parse")
  | aexp _ = raise Error "parse"
```

Überlegen Sie sich, welche Ausgabe die folgenden Prozeduraufrufe erzeugen (bzw. an welcher Stelle ein Fehler fliegt). Könnten die jeweiligen Prozeduraufrufe beim Parsen eines gültigen Ausdrucks vorkommen? Wenn ja, wann?

Hinweis: Falls Sie sich an einer Stelle nicht sicher sind, befragen Sie **NICHT den Interpreter**, sondern fertigen Sie stattdessen ein (informelles) Ausführungsprotokoll an, um das Ergebnis bestimmen zu können.

(a) `exp [NEG, LPAR, TRUE, AND, FALSE, RPAR]`

(b) `uexp [TRUE, AND, FALSE, IMP, TRUE]`

(c) `aexp [TRUE, AND, FALSE, IMP, TRUE]`

(d) `exp [NEG, TRUE, RPAR, AND, TRUE]`

(e) `exp [NEG, LPAR]`

(f) `uexp' (Neg(True), [AND, FALSE, AND, TRUE])`

(g) `exp [TRUE, FALSE]`

(h) `exp [TRUE, IMP]`

Lösungsvorschlag TK.4

(a) Der Prozeduraufruf wertet zu `(Neg(And(True, False)), [])` aus. Da die eingegebene Liste bereits einen gültigen Ausdruck repräsentiert, könnte ein solcher Prozeduraufruf auch beim Parsen eines gültigen Ausdrucks vorkommen.

(b) Der Prozeduraufruf wertet zu `(And(True, False), [IMP, TRUE])` aus. Ein solcher Prozeduraufruf würde beispielsweise beim Parsen des Ausdrucks $true \wedge false \rightarrow true$ vorkommen. `exp` ruft zunächst `uexp` auf, um den Ausdruck links von der Implikation zu parsen. `uexp` gibt die noch nicht geparste Restliste an `exp` zurück.

(c) Der Prozeduraufruf wertet zu `(True, [AND, FALSE, IMP, TRUE])` aus. Ein solcher Prozeduraufruf würde beispielsweise beim Parsen des Ausdrucks $true \wedge false \rightarrow true$ vorkommen. Ein Ausdruck, der mit einem elementaren Ausdruck beginnt, wird zunächst bis zu `aexp` „durchgereicht“, wo dieser erste elementare Ausdruck geparst wird. Die Restliste wird zu `mexp` zurückgegeben.

(d) Der Prozeduraufruf wertet zu `(Neg(True), [RPAR, AND, TRUE])` aus. Ein solcher Prozeduraufruf würde beispielsweise beim Parsen des Ausdrucks $(\neg true) \wedge true$ vorkommen. Der Ausdruck wird zunächst bis zu `aexp` durchgereicht, anschließend wird wieder `exp` mit dem Argument `[NEG, TRUE, RPAR, AND ↳ , TRUE]` aufgerufen, um den Teil innerhalb der Klammern zu parsen. Sobald man beim Parsen auf `RPAR` trifft, wird diese Restliste an `aexp` zurückgeliefert.

(e) Hier fliegt ein Fehler: Nachdem der Parser auf `LPAR` trifft, wird `exp` mit der leeren Liste aufgerufen. Diese wird bis ganz nach unten zu `aexp` durchgereicht, und hier fliegt dann ein Fehler. Entsprechend kann dieser Prozeduraufruf auch nicht beim Parsen eines gültigen Aufrufs vorkommen.

(f) Der Prozeduraufruf wertet zu `(And(And(Neg(True), False), True), [])` aus. Ein solcher Prozeduraufruf würde beispielsweise beim Parsen des Ausdrucks $\neg true \wedge false \wedge true$ vorkommen. Die Hilfsprozedur `uexp'` nimmt als erstes Argument den Ausdruck, der auf der linken Seite der Verundung stehen soll: Das ermöglicht die Linksklammerung.

(g) Der Prozeduraufruf wertet aus zu `(True, [FALSE])`. Hier tritt derselbe Fall ein, wie wenn anstelle von `FALSE` ein `RPAR` stehen würde: Nachdem das `TRUE` geparst wurde, geben `uexp` und `exp` die Restliste unverändert zurück. Trotzdem könnte ein solcher Prozeduraufruf nicht beim Parsen eines gültigen Ausdrucks entstehen, da $true$ und $false$ niemals direkt hintereinander stehen können.

(h) Hier fliegt ein Fehler: Nachdem das `IMP` geparst wurde, wird `exp` wie oben mit der leeren Liste aufgerufen, um den Teil rechts vom `AND` zu parsen. Entsprechend kann dieser Prozeduraufruf auch nicht beim Parsen eines gültigen Ausdrucks vorkommen.

Aufgabe TK.5 *(!!!)*

Wir betrachten die Ausrufezeichen-Sprache. Eine Phrase dieser Sprache beginnt immer mit einem Ausrufezeichen, darauf folgt eine Zahl oder eine geklammerte Phrase. Beispielsweise ist !!5 kein gültiger Ausdruck, wohingegen !(!5) gültig ist. Gegeben sind folgende Konstruktortypen:

```
datatype exp = A of exp | Icon of int
datatype token = AUS | ICON of int | LPAR | RPAR
```

(a) Geben Sie eine formale Grammatik für die oben beschriebene phrasale Syntax an.

(b) Implementieren Sie einen Parser für Ihre Grammatik.

Lösungsvorschlag TK.5

(a)

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

(b)

```
fun exp (AUS::tr) = let val (e, ts) = pexp tr in (A(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 TK.6 *(Probleme mit Linksklammerung)*
Dieter Schlau hat ein Problem mit einem Parser. Er möchte folgende Grammatik für bool'sche und-Ausdrücke realisieren:

$$\begin{aligned} exp &::= [exp\ \texttt{"and"}]\ pexp \\ pexp &::= \texttt{"true"} \mid \texttt{"false"} \mid \texttt{"("}\ exp\ \texttt{")"} \end{aligned}$$

Dabei ist *and* linksklammernd.

Aus der Vorlesung hat Dieter mitgenommen, dass er die Grammatik erst ein bisschen umformen muss, damit sie für den rekursiven Abstieg tauglich ist. Dementsprechend hat er eine Hilfskategorie eingeführt:

$$\begin{aligned} exp &::= pexp\ exp' \\ exp' &::= [\texttt{"and"}\ pexp\ exp'] \\ pexp &::= \texttt{"true"} \mid \texttt{"false"} \mid \texttt{"("}\ exp\ \texttt{")"} \end{aligned}$$

Einen funktionierenden Lexer hat er bereits, der ihm Wortfolgen mit folgenden Token darstellt:

```
1 datatype token = T | F | A | LPAR | RPAR
```

Den Parser hat er nun versucht, mit folgendem Code zu realisieren:

```
1  exception Error of string
2  datatype exp = True | False | And of exp * exp
3
4  fun exp xs = exp' (pexp xs)
5  and exp' (e1, A::xr) = let val (e2, ys) = exp' (pexp xr) in (And (e1, e2), ys) end
6    | exp' s = s
7  and pexp (T::xr) = (True, xr)
8    | pexp (F::xr) = (False, xr)
9    | pexp (LPAR::xr) = (case exp xr of (e, RPAR::yr) ⇒ (e, yr) | _ ⇒ raise Error "match")
10   | pexp _ = raise Error "pexp"
```

Er hat jedoch den Verdacht, dass sich ein Fehler bei der Prozedur `exp'` eingeschlichen hat.

(a) Machen Sie sich anhand des Ausdrucks `[T, A, F, A, T]` klar, dass Dieter tatsächlich einen Fehler in seinem Programm hat. Geben Sie dazu die Rekursionsfolge von `exp'` an. Schreiben Sie dann vom letzten bis zum ersten Aufruf das jeweilige Ergebnis hinzu. Leiten Sie daraus den explizit geklammerten Ausdruck her. Falls Dieter die Linksklammerung korrekt implementiert hätte, müsste dieser (*true and false*) *and true* lauten.

(b) Als Dieter die Rekursionsfolge und die dazugehörigen Ergebnisse sieht, hat er die Idee, diese etwas abzuändern:

Rekursionsfolge	Ergebnis
$(True,\ [A,\ F,\ A,\ T])$	$(And\,(And\,(True,\ False),\ True),\ [])$
$\downarrow$	
$(And\,(True,\ False),\ [A,\ T])$	$(And\,(And\,(True,\ False),\ True),\ [])$
$\downarrow$	
$(And\,(And\,(True,\ False),\ True),\ [])$	$(And\,(And\,(True,\ False),\ True),\ [])$

Tatsächlich lässt sich Dieters Idee auch durch eine kleine Änderung im Code umsetzen. Korrigieren Sie Zeile 5 des Parsers, sodass Linksklammerung korrekt implementiert wird.

Lösungsvorschlag TK.6

(a) Es ergibt sich der rechtsgeklammerte Ausdruck *true and* (*false and true*):

Rekursionsfolge	Ergebnis
$(True,\ [A,\ F,\ A,\ T])$	$(And\,(True,\ And\,(False,\ True)),\ [])$
$\downarrow$	
$(False,\ [A,\ T])$	$(And\,(False,\ True),\ [])$
$\downarrow$	
$(True,\ [])$	$(True,\ [])$

(b) Der korrekte Parser lautet:

```
exception Error of string
datatype exp = True | False | And of exp * exp

fun exp xs = exp' (pexp xs)
and exp' (e1, A::xr) = let val (e2, ys) = pexp xr in exp' (And (e1, e2), ys) end
  | exp' s = s
and pexp (T::xr) = (True, xr)
  | pexp (F::xr) = (False, xr)
  | pexp (LPAR::xr) = (case exp xr of (e, RPAR::yr) ⇒ (e, yr)
                                    | _ ⇒ raise Error "match")
  | pexp _ = raise Error "pexp"
```

Aufgabe TK.7 *(Form-Parser)*
Zusätzlich zu Bezeichnern und expliziten Klammern seien die beiden Operatoren □ und ◯ gegeben.

- □ ist ein binärer und ◯ ein unärer Operator.
- □ klammert rechts und schwächer als ◯.

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

(b) Erstellen Sie einen Konstruktortyp `token` und einen Konstruktortyp `shape` für die Baumdarstellung der Ausdrücke.

(c) Fügen Sie ihrer Grammatik den binären Operator △ hinzu. △ klammert stärker als □ und ◯, sowie links.

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

Lösungsvorschlag TK.7

(a)

$$\begin{aligned} \mathit{square} &::= \mathit{circle}\ [\text{“}\square\text{”}\ \mathit{square}] \\ \mathit{circle} &::= [\text{“}\ \bigcirc\ \text{”}]\ \mathit{prim} \\ \mathit{prim} &::= \mathit{id} \mid \text{“(”}\mathit{square}\text{“)”} \end{aligned}$$

(b)

```
datatype token = SQUARE | CIRCLE | ID of string | LPAR | RPAR

datatype shape = Square of shape * shape | Circle of shape | Id of string
```

(c)

$$\begin{aligned} \mathit{circle} &::= [\text{“}\ \bigcirc\ \text{”}]\ \mathit{triangle} \\ \mathit{triangle} &::= [\mathit{triangle}\ \text{“}\triangle\text{”}]\ \mathit{prim} \\ \text{RA-Tauglich: } \mathit{triangle} &::= \mathit{prim}\ \mathit{triangle}' \\ \mathit{triangle}' &::= [\text{“}\triangle\text{”}\ \mathit{prim}\ \mathit{triangle}'] \end{aligned}$$

```
token = ... | TRIANGLE
shape = ... | Triangle of shape * shape
```

(d)

```
exception Parse


fun square ts = case circle ts of
                    (e, SQUARE :: ts') ⇒ let val (e', ts'') = square ts'
                                         in (Square (e, e'), ts'')
                                         end
                  | s ⇒ s

```

```
10 and circle (CIRCLE :: ts) = let val (e, ts') = triangle ts
11                             in (Circle e, ts')
12                             end
13   | circle s = triangle s
14
15 and triangle ts = triangle' (prim ts)
16 and triangle' (s, TRIANGLE :: ts) = (case prim ts of
17                                         (s', tr) ⇒ triangle' (Triangle(s, s'), tr))
18   | triangle' (s, tr) = (s, tr)
19
20 and prim (ID x :: ts) = (Id x, ts)
21   | prim (LPAR :: ts) = (case square ts of
22                            (a, RPAR :: tr') ⇒ (a, tr')
23                          | _ ⇒ raise Parse)
24   | prim _ = raise Parse
25
26
27 fun parse ts = case square ts of
28                   (a, nil) ⇒ a
29                 | _ ⇒ raise Parse
```

Aufgabe TK.8 *(Herzchen-Parser)*
Zusätzlich zu Bezeichnern und expliziten Klammern seien folgende Operatoren gegeben: ♣, ♡ und ♠.

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

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

(b) Verändern Sie die Grammatik, sodass sie RA-tauglich wird.

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

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

(e) Schreiben Sie nun einen Parser. Bei einer ungültigen Eingabe soll die Ausnahme `Parse` geworfen werden.

Lösungsvorschlag TK.8

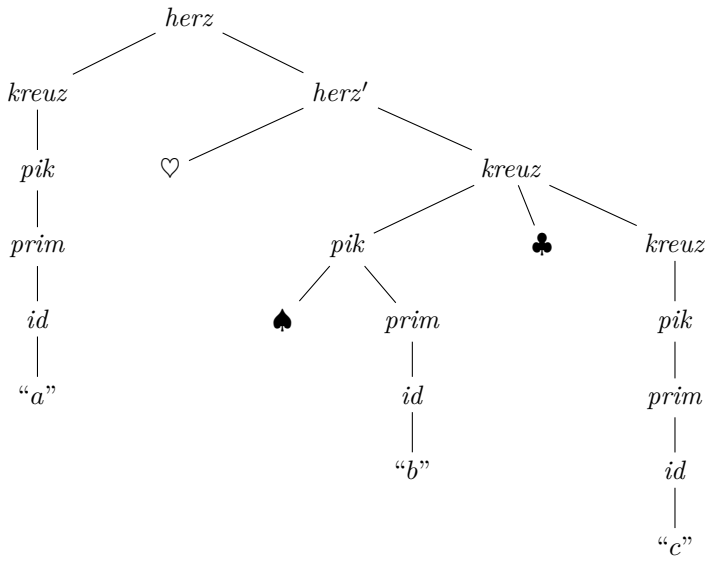
(a)

$$\begin{aligned} herz &::= kreuz \mid herz\ \text{“♡”}\ kreuz \\ kreuz &::= pik \mid pik\ \text{“♣”}\ kreuz \\ pik &::= prim \mid \text{“♠”}\ prim \\ prim &::= id \mid \text{“(”}herz\text{“)”} \end{aligned} \qquad \begin{aligned} herz &::= [herz\ \text{“♡”}]\ kreuz \\ kreuz &::= pik\ [\text{“♣”}\ kreuz] \\ pik &::= [\text{“♠”}]\ prim \\ prim &::= id \mid \text{“(”}herz\text{“)”} \end{aligned}$$

(b) RA-taugliche Grammatik mit Hilfskategorien:

$$\begin{aligned} herz &::= kreuz\ herz' \\ herz' &::= [\text{“♡”}\ kreuz\ herz'] \\ kreuz &::= pik\ [\text{“♣”}\ kreuz] \\ pik &::= [\text{“♠”}]\ prim \\ prim &::= id \mid \text{“(”}herz\text{“)”} \end{aligned}$$

(c) $a \heartsuit \spadesuit b \clubsuit c$



(d)

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

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

(e)

```
exception Parse

fun herz ts = herz' (kreuz ts)

and herz' (e, HERZ :: ts) = let val (e', ts') = kreuz ts
                            in herz' (Herz (e, e'), ts')
                            end
  | herz' s = s

and kreuz ts = (case pik ts of (e, KREUZ :: ts') ⇒ let val (e', ts'') = kreuz ts'
                                                   in (Kreuz (e, e'), ts'')
                                                   end
                             | s ⇒ s)

and pik (PIK :: ts) = let val (e, ts') = prim ts
                      in (Pik e, ts')
                      end
  | pik s = prim s

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

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

Aufgabe TK.9 *(Reine freie Bäume)*
Wir können reine Bäume durch Klammern darstellen. Beispielsweise lässt sich der atomare Baum `T[]` durch `()` darstellen oder `T [T[], T[]]` durch `(() ())`. Wir wollen einen Parser für reine Bäume schreiben, die sich nach folgender Grammatik bilden lassen:

$$A ::= \text{“(”}\ B$$
$$B ::= \text{“)”}\ \mid\ A\ B$$

(a) Deklarieren Sie Konstruktortypen für Wörter (Tokens) und reine Bäume.

(b) Schreiben Sie einen Parser `treeparse : token list` → `tree` für reine Bäume nach obiger Grammatik.

(c) Schreiben Sie einen Deparser `treedeparse : tree` → `token list` für reine Bäume.

Lösungsvorschlag TK.9

(a)

```
datatype token = LPAR | RPAR
datatype tree = T of tree list
```

(b)

```
fun a (LPAR :: tr) = let val (ts, tr') = b tr
                     in (T ts, tr')
                     end
| a _ = raise Error

and b (RPAR :: tr) = (nil, tr)
  | b ts = let val (t, tr) = a ts
               val (ts, tr') = b tr
           in (t::ts, tr')
           end

fun treeparse l = case a l of
                      (t, nil) ⇒ t
                    | _ ⇒ raise Error
```

(c)

```
fun treedeparse (T l) = LPAR :: List.concat (map treedeparse l) @ [RPAR]
```

Aufgabe TK.10 *(RegEx)*

Es gibt drei Operatoren für sogenannte *reguläre Ausdrücke*: Kleene-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
```

Lösungsvorschlag TK.10

(a)

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

Hierbeit bezeichnet *con* einen beliebigen String.

(b)

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

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"

fun parse ts = case plusexp ts of (e, []) ⇒ e
                                |    _    ⇒ raise Error "Not a RegEx"
```

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 kringelexp' (e, KRINGEL :: tr) = kringelexp' (case sternexp tr
                                                   of (e', tr') ⇒ (Kringel (e, e'), tr'))
  | kringelexp'          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"

fun parse ts = case plusexp ts of (e, []) ⇒ e
  |    _    ⇒ raise Error "Not a RegEx"
```

Datenstrukturen

Aufgabe TK.11 *(3d-Vektoren)*
Wir wollen nun die 3-dimensionalen mathematischen Vektoren (nicht zu verwechseln mit denen aus SML!) implementieren. Dazu sei folgende Signatur gegeben:

```
signature VECTOR3D = sig
  type vec3d
  val vec3d : real → real → real → vec3d
  val add: vec3d → vec3d → vec3d
  val mul: vec3d → real → vec3d
  val prod: vec3d → vec3d → real
end
```

Implementieren Sie nun die Datenstruktur `Vector3d`, wobei folgendes gelten soll:

- `vec3d` soll einen 3d-Vektor anlegen, wobei die übergebenen Werte die x, y und z-Komponenten sind.
- `add` soll zwei 3d-Vektoren komponentenweise addieren.
- `mul` soll die herkömmliche Skalarmultiplikation darstellen, also die Komponenten des 3d-Vektors mit einer Zahl multiplizieren.
- `prod` soll das herkömmliche Skalarprodukt bereitstellen, also die x, y und z-Komponenten der zwei 3d-Vektoren jeweils miteinander multiplizieren und diese Produkte dann aufsummieren.

Hinweis: Überlegen Sie sich zuerst, wie Sie intern einen solchen 3d-Vektor darstellen.

Lösungsvorschlag TK.11

```
structure Vector3d :> VECTOR3D = struct
  type vec3d = real * real * real
  fun vec3d a b c = (a, b, c)
  fun add (a, b, c) (d, e, f) : vec3d = (a + d, b + e, c + f)
  fun mul (a, b, c) (k : real) = (k * a, k * b, k * c)
  fun prod (a, b, c) (d, e, f) : real = a * d  + b * e + c * f
end
```

Aufgabe TK.12 *(Schnellere Mengen)*
Sei die folgende Signatur bzw. Struktur gegeben[1]:

```
signature ISET = sig
  type set
  val set : int list → set
  val member : set → int → bool
  val union : set → set → set
  val subset : set → set → bool
end
```

```
structure ISet :> ISET = struct
  type set = int list
  fun set xs = xs
  fun member ys x = List.exists (fn y ⇒ y = x) ys
  fun union xs ys = xs @ ys
  fun subset xs ys = List.all (member ys) xs
  end
```

Schreiben Sie nun eine Struktur `ISSet`, die die selben Operationen anbietet wie `ISet`, allerdings effizienter! Folgende Komplexitäten sollen erfüllt sein:

- `set`: $\mathcal{O}(n \cdot \log n)$
- `member`: $\mathcal{O}(n)$
- `union`: $\mathcal{O}(n)$
- `subset`: $\mathcal{O}(n)$

Hinweis: Falls Sie keinen Ansatz haben, erinnern Sie sich an Abschnitt 5.5 im Buch zurück.

Lösungsvorschlag TK.12

```
structure ISSet = struct
  type iset = int list

  fun split xs = foldl (fn (x, (ys, zs)) ⇒ (zs, x :: ys)) ([], []) xs

  fun smerge    []        ys     = ys
    | smerge    xs        []     = xs
    | smerge (x :: xr) (y :: yr) = case Int.compare(x, y)
                                     of LESS    ⇒ x :: smerge xr (y :: yr)
                                      | EQUAL   ⇒ x :: smerge xr yr
                                      | GREATER ⇒ y :: smerge (x :: xr) yr

  fun ssort [ ] = [ ]
    | ssort [x] = [x]
    | ssort xs  = let val (ys, zs) = split xs
          in smerge (ssort ys) (ssort zs) end
```

[1] https://sosml.org/share/ea6109eeae7d92c6c0e4457cf267c387ac03bf0a440564ca5a8c45bd2e319a93

```
17
18    fun sublist     []         ys      = true
19      | sublist     xs         []      = false
20      | sublist (x :: xr) (y :: yr) = case Int.compare(x, y)
21                                        of LESS ⇒ false
22                                         | EQUAL ⇒ sublist xr yr
23                                         | GREATER ⇒ sublist (x :: xr) yr
24
25    fun set xs = ssort xs
26    fun union xs ys = smerge xs ys
27    fun member ys x = List.exists (fn y ⇒ y = x) ys
28    fun subset xs ys = sublist xs ys
29  end
```

Aufgabe TK.13 *(Schnellste Mengen)*
Betrachten Sie die Signatur aus Aufgabe TK.12.

(a) Schreiben Sie eine Struktur, die die Signatur `ISET` implementiert, allerdings *noch* effizienter! Folgende Komplexitäten sollen erfüllt sein:

- `set`: $\mathcal{O}(1)$
- `union`: $\mathcal{O}(1)$
- `member`: Beliebig

Können Sie unter diesen Bedingungen auch `subset` implementieren? Wieso bzw. wieso nicht?

(b) Ergänzen Sie die Signatur und Ihre Implementierung um `N : set` und `negate : set → set`. Diese sollen die Menge $\mathbb{N}$ und das Mengenkomplement darstellen.

Welche anderen Ihnen bekannten Mengenoperationen können sie implementieren?

Lösungsvorschlag TK.13
Es ist nicht möglich, `subset` zu implementieren.

```
1  signature SET = sig
2      type set
3      val set    : int list → set
4      val member : int → set → bool
5      val union  : set → set → set
6      val N      : set
7      val negate : set → set
8  end
9
10 structure Set :> SET = struct
11     type set = int → bool
12     fun set xs = fn i ⇒ List.exists (fn x ⇒ x = i) xs
13     fun member x s = s x
14     fun union s1 s2 = fn x ⇒ s1 x orelse s2 x
15     val N = fn _ ⇒ true
16     fun negate s = fn x ⇒ not (s x)
17 end
```

Aufgabe TK.14 *(Spaß mit dem LSF)*
Sie wurden von der UdS als Programmierer für eine neue, verbesserte Version des LSF eingestellt. Da Sie sich noch daran erinnern, in Programmierung 1 gelernt zu haben, dass SML die beste Programmiersprache der Welt ist, wollen Sie das LSF 2.0 in Ihrer altbekannten Lieblingssprache programmieren.

Schreiben Sie eine Datenstruktur `Leistungen`, die Leistungsinformationen eines Studenten nach Fächern speichert sowie insgesamt geleistete Credit Points sowie die Durchschnittsnote liefert. Wir wollen zu jeder Veranstaltung Namen, Note und Credit Points speichern.
Überlegen Sie sich zunächst, wie Sie Leistungen und Vorlesungen in SML darstellen können. Implementieren Sie dann eine Datenstruktur, die folgende Signatur hat:

```
type leistungen
val empty : leistungen
val add : leistungen → (string * int * real) → leistungen
val creditpoints: leistungen → int
val avggrade: leistungen → real
```

Lösungsvorschlag TK.14
Eine Vorlesung wird als Tupel `(name, cp, grade)` implementiert.
Ein Leistungsverzeichnis wird durch ein Quadrupel `(cp, grade, n, vorlesungen)` beschrieben. `vorlesungen` ist dabei eine Liste von Vorlesungen und `n` deren Länge.

```
signature LEISTUNGEN = sig
  type leistungen

  val empty : leistungen
  val add : leistungen → (string * int * real) → leistungen
  val creditpoints : leistungen → int
  val avggrade : leistungen → real
end

structure Leistungen :> LEISTUNGEN = struct
  type leistungen = (int * real * int * (string * int * real) list)
  val empty = (0, 0.0, 0, nil)
  fun add (cpt, gt, n, l) (name, cp, grade) = (cpt + cp, ((gt * Real.fromInt(n)) + grade)/Real
    ↳ .fromInt(n + 1), n + 1, (name, cp, grade)::l)
  fun creditpoints (cp, _, _, _) = cp
  fun avggrade (_, g, _, _) = g
end
```

```
val prog1 = ("Prog1", 9, 1.0)
val mfi = ("MFI1", 9, 2.0)

val leistungen = Leistungen.add (Leistungen.add Leistungen.empty prog1) mfi
val cp = Leistungen.creditpoints leistungen
val note = Leistungen.avggrade leistungen
```