Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation

Prof. Dr.-Ing. Gregor Snelting — Jun.-Prof. Dr. Oliver Hummel

gregor.snelting@kit.edu — hummel@kit.edu

Klausur Programmierparadigmen (Wiederholer) — Beispiellösung

SS2013, 1. Oktober 2013, 14:00 – 16:00 Uhr

Zugelassene Hilfsmittel: Papierbasierte Quellen (Vorlesungsfolien, Übungsblätter, eigene Aufzeichnungen, Bücher, ...)
Die Verwendung von elektronischen Geräten ist verboten.

Bearbeitungszeit: 120 min

Aufgabe 1 (Haskell, Laziness, Streams) [6 Punkte]

Ein *pythagoreisches Tripel* ist ein Tripel (x, y, z) aus natürlichen Zahlen, für die gilt: $x^2 + y^2 = z^2$ und $x, y, z > 0$. Geben Sie eine Haskell-Funktion `triples :: [(Integer, Integer, Integer)]` an, die die unendliche Liste aller pythagoreischen Tripel zurückliefert.

Hinweis: Sie können List Comprehensions verwenden.

Beispiellösung:

```haskell
triples :: [(Integer, Integer, Integer)]
triples = [(x, y, z) | z <- [1..], y <- [1..z-1], x <- [1..z-1],
                       x^2 + y^2 == z^2]

-- Effizienter ist es, die ganzzahlige Wurzel aus x^2+y^2 zu berechnen.
-- Die folgende Loesung funktioniert aber nur, solange die Zahlen
-- hinreichend klein sind, da bei grossen Zahlen Rundungsfehler
-- in der Berechnung auftreten koennen.
isqrt = floor . sqrt . fromIntegral
triples' :: [(Integer, Integer, Integer)]
triples' = [(x,y,z) | x <- [1..], y <- [1..x], z <- [isqrt (x^2 + y^2)],
                             x^2 + y^2 == z^2]
```

Aufgabe 2 (Programmieren im Lambda-Kalkül) [15 Punkte]

In dieser Aufgabe werden Sie die Definition einer Vorgängerfunktion `pred` für Church-Zahlen herleiten.

Verwenden Sie im Folgenden die Funktionen

$$\begin{aligned} \texttt{pair} &= \lambda a.\ \lambda b.\ \lambda f.\ f\ a\ b \\ \texttt{fst} &= \lambda p.\ p\ (\lambda a.\ \lambda b.\ a) \\ \texttt{snd} &= \lambda p.\ p\ (\lambda a.\ \lambda b.\ b) \end{aligned}$$

sowie die aus der Vorlesung bekannten Funktionen `succ` und c_n (Church-Zahlen).

Bei allen Beta-Reduktionen dürfen Sie verwenden, dass jeweils gilt:

$$\begin{aligned} \texttt{fst (pair } a\ b\texttt{)} &\Rightarrow^* a \\ \texttt{snd (pair } a\ b\texttt{)} &\Rightarrow^* b \\ \texttt{succ } c_n &\Rightarrow^* c_{n+1} \end{aligned}$$

(a) Geben Sie einen Lambdaterm `next` an, der zu einem gegebenem Paar $p = (x, y)$ das Paar $(y, \texttt{succ } y)$ berechnet (x und y seien Church-Zahlen). [3 Punkte]
Beispiellösung:
$\texttt{next} = \lambda p.\ \texttt{pair}\ (\texttt{snd}\ p)\ (\texttt{succ}\ (\texttt{snd}\ p))$
Oder äquivalent z.B.:
$\texttt{next} = \lambda p.\ (\lambda y.\ \texttt{pair}\ y\ (\texttt{succ}\ y))\ (p\ (\lambda a.\ \lambda b.\ b))$

(b) Zeigen Sie, dass $\texttt{next}\ (\texttt{pair}\ n\ m)$ zu $\texttt{pair}\ m\ (\texttt{succ}\ m)$ reduziert. [2 Punkte]
Beispiellösung:

$$\begin{aligned} \texttt{next}\ (\texttt{pair}\ n\ m) &\Rightarrow \texttt{pair}\ (\texttt{snd}\ (\texttt{pair}\ n\ m))\ (\texttt{succ}\ (\texttt{snd}\ (\texttt{pair}\ n\ m))) \\ &\Rightarrow^* \texttt{pair}\ m\ (\texttt{succ}\ (\texttt{snd}\ (\texttt{pair}\ n\ m))) \\ &\Rightarrow^* \texttt{pair}\ m\ (\texttt{succ}\ m) \end{aligned}$$

(c) Verwenden Sie `next`, um eine Vorgängerfunktion `pred` für Church-Zahlen anzugeben. [5 Punkte]
Hinweis: n-malige Anwendung von `next` auf das Paar $(0, 0)$ ergibt $(n-1, n)$

Beispiellösung:
$\texttt{pred} = \lambda n.\ \texttt{fst}\ (n\ \texttt{next}\ (\texttt{pair}\ c_0\ c_0))$

(d) Berechnen Sie den Vorgänger von c_2, indem Sie die zugehörige Beta-Reduktion angeben. [5 Punkte]
Beispiellösung:

$$\begin{aligned} \texttt{pred}\ c_2 &\Rightarrow \texttt{fst}\ (c_2\ \texttt{next}\ (\texttt{pair}\ c_0\ c_0)) \\ &= \texttt{fst}\ ((\lambda s.\ \lambda z.\ s\ (s\ z))\ \texttt{next}\ (\texttt{pair}\ c_0\ c_0)) \\ &\Rightarrow^* \texttt{fst}\ (\texttt{next}\ (\texttt{next}\ (\texttt{pair}\ c_0\ c_0))) \\ &\Rightarrow^* \texttt{fst}\ (\texttt{next}\ (\texttt{pair}\ c_0\ (\texttt{succ}\ c_0))) \\ &\Rightarrow^* \texttt{fst}\ (\texttt{pair}\ (\texttt{succ}\ c_0)\ (\texttt{succ}\ (\texttt{succ}\ c_0))) \\ &\Rightarrow^* \texttt{succ}\ c_0 \\ &\Rightarrow^* c_1 \end{aligned}$$

Aufgabe 3 (Typinferenz) [15 Punkte]

Gegeben seien folgende Lambdaterme:

$\texttt{A} = \lambda f.\ \lambda x.\ f\ x$
$\texttt{B} = \lambda f.\ \lambda x.\ f\ (f\ x)$
$\texttt{C} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$
$\texttt{D} = \lambda x.\ \lambda y.\ y\ (x\ y)$
$\texttt{E} = \lambda z.\ z$
$\texttt{F} = \texttt{D E} = (\lambda x.\ \lambda y.\ y\ (x\ y))\ (\lambda z.\ z)$
$\texttt{G} = \lambda y.\ \textbf{let}\ \texttt{f} = \lambda x.\ x\ \texttt{0}\ \textbf{in}\ \texttt{f}\ (\lambda z.\ y)$
$\texttt{H} = \lambda y.\ \textbf{let}\ \texttt{f} = \lambda x.\ x\ \texttt{0}\ \textbf{in}\ \texttt{f}\ (\lambda z.\ z\ y)$

Hinweis: Die 0 in den Termen G und H hat den Typ int.

(a) Geben Sie für jeden Typ in der unten stehenden Tabelle *alle* Terme A – H an, die diesen Typ haben können. [13 Punkte]
Unterstreichen Sie die eingetragenen Terme, wenn der angegebene Typ auch der *allgemeinste* Typ des entsprechenden Terms ist.

	ist Typ von Term
$\alpha \to \alpha$	$\underline{\texttt{E}}$, $\underline{\texttt{G}}$
$(\alpha \to \alpha) \to \alpha \to \alpha$	A, $\underline{\texttt{B}}$, E, G
$(\alpha \to \beta) \to \alpha \to \beta$	$\underline{\texttt{A}}$, E, G
$((\alpha \to \beta) \to \alpha) \to (\alpha \to \beta) \to \beta$	$\underline{\texttt{D}}$
nicht typisierbar	, C, F, H

(b) Geben Sie das **Typschema** für die **let**-gebundene Variable f im Term G an. [2 Punkte]
Beispiellösung: Typschema für f in G: $\texttt{f} : \forall\alpha.(\mathsf{int} \to \alpha) \to \alpha$

Aufgabe 4 (Haskell, Prolog, Listenverarbeitung) [15 Punkte]

(a) Implementieren Sie eine Haskell-Funktion [8 Punkte]

```
splits :: [t] -> [([t], [t])]
```

die alle möglichen Zerlegungen der übergebenen Liste in einen Anfangsteil und einen Endteil berechnet, so dass Anfangsteil und Endteil aneinandergehängt wieder die ursprüngliche Liste ergeben. Anfangs- und Endteil sollen jeweils in einem Tupel gespeichert und alle möglichen Tupel als Liste zurückgeliefert werden.
Hinweis: Sie können List Comprehensions verwenden.
Beispiel:

```
> splits [1,2,3]
[ ([],[1,2,3]), ([1],[2,3]), ([1,2],[3]), ([1,2,3],[]) ]
```

Beispiellösung:

```
splits :: [t] -> [([t], [t])]
splits l  = [ (take n l, drop n l) | n <- [0..length l] ]

-- Alternativ
splits' :: [t] -> [([t], [t])]
splits' []     = [([], [])]
splits' (x:l) = ([], x:l) : [ (x:s, e) | (s, e) <- splits' l ]
```

(b) Implementieren Sie ein Prolog-Prädikat [7 Punkte]

`splits(L, Res)`

das **bei Reerfüllung** alle möglichen Zerlegungen der übergebenen Liste `L` in einen Anfangsteil und einen Endteil generiert, so dass Anfangsteil und Endteil aneinandergehängt wieder die ursprüngliche Liste ergeben. Anfangs- und Endteil sollen jeweils als Tupel in `Res` zurückgeliefert werden.

Beispiel:

```
? splits([1,2,3], Res).
Res = ([], [1,2,3]) ;
Res = ([1], [2,3]) ;
Res = ([1,2], [3]) ;
Res = ([1,2,3], []) ;
No
```

Beispiellösung:

```
splits(L, ([], L)).
splits([X|L], ([X|S], E)) :- splits(L, (S, E)).

% Alternativ (nutzt Rueckwaertsausfuehrung von append)
splits(L, (Xs, Ys)) :- append(Xs, Ys, L).
```

Aufgabe 5 (Prolog, Differenzlisten) [14 Punkte]

Differenzlisten in der logischen Programmierung gehören zu den unvollständigen Datenstrukturen und stellen eine Alternative zu herkömmlichen Listen dar. Dabei wird eine Liste als ein „Tupel" zweier herkömmlicher Listen `Xs` und `Ys` dargestellt (Schreibweise `Xs-Ys`), wobei `Ys` ein Suffix von `Xs` ist. Die Differenzliste `Xs-Ys` wird dann als die Liste `Xs` interpretiert, von der das Suffix `Ys` abgeschnitten wurde.

Die Konkatenation auf Differenzlisten ist definiert als

```
append_dl(As-Bs, Bs-Cs, As-Cs).
```

D.h. wenn man an die Differenzliste `As-Bs` die Differenzliste `Bs-Cs` anhängt, erhält man `As-Cs`.

Beispiel:
Die Liste `[1,2,3]` lässt sich z.B. durch folgende Differenzlisten darstellen:

```
[1,2,3,4,5]-[4,5]
[1,2,3|Xs]-Xs
```

Für `append_dl` gilt z.B.:

```
append_dl([1,2,3,4]-[3,4], [3,4]-[], [1,2,3,4]-[]).
```

(a) Gegeben ist die Implementierung von `rev/2` (Umdrehen einer Liste) für herkömmliche Listen: [9 Punkte]

```
rev([], []).
rev([X|Xs], Ys) :- rev(Xs, Zs), append(Zs, [X], Ys).
```

Implementieren Sie das Prädikat `rev_dl(Xs, Ys-Zs)` so, dass nach einem Aufruf von `rev_dl(Xs, Ys-[])` in `Ys` die Umkehrung von `Xs` steht. Verwenden Sie `append_dl`.

Beispiellösung:

```
rev_dl([], Ys-Ys) :- !.
rev_dl([X|Xs], D) :-
        rev_dl(Xs, D1),
        append_dl(D1, [X|Rs]-Rs, D).

# Mit ausgeschriebenen Differenzlisten
rev_dl([], Ys-Ys) :- !.
rev_dl([X|Xs], Ys-Zs) :-
        rev_dl(Xs, Ys1-Zs1),
        append_dl(Ys1-Zs1, [X|Rs]-Rs, Ys-Zs).
```

Hinweis: Da die Laufzeit von `append` in $\mathcal{O}(n)$ liegt, die Laufzeit von `append_dl` jedoch in $\mathcal{O}(1)$ ist, ist das Umdrehen einer Liste mit Differenzlisten wesentlich effizienter.

(b) Unifizieren Sie den Aufruf von `append_dl` aus ihrer Lösung der vorherigen Teilaufgabe mit der definierenden Regel von `append_dl`: [5 Punkte]

```
append_dl(As-Bs, Bs-Cs, As-Cs).
```

Geben Sie den Unifikator an.

Beispiellösung: Unifikator: $\{\ \mathtt{D} \mapsto \mathtt{As} - \mathtt{Rs}, \mathtt{D1} \mapsto \mathtt{As} - [\mathtt{X}|\mathtt{Rs}], \mathtt{Bs} \mapsto [\mathtt{X}|\mathtt{Rs}], \mathtt{Cs} \mapsto \mathtt{Rs}\}$

Für ausgeschriebene Differenzlisten: $\{\ \mathtt{As} \mapsto \mathtt{Ys}, \mathtt{Ys1} \mapsto \mathtt{Ys}, \mathtt{Bs} \mapsto [\mathtt{X}|\mathtt{Zs}], \mathtt{Zs1} \mapsto [\mathtt{X}|\mathtt{Zs}], \mathtt{Rs} \mapsto \mathtt{Zs}, \mathtt{Cs} \mapsto \mathtt{Zs}\}$

Aufgabe 6 (C-Deklarationen) [4 Punkte]

Benutzen Sie die in der Vorlesung (VL 4_4, Folie 26) vorgestellte Precedence-Regel nach Linden zum Lesen der folgenden C-Deklaration. Tragen Sie in untenstehender Tabelle jeweils ein, welches Token Sie mit Hilfe welcher Regel erkannt haben und wie es zur Benennung der Deklaration beiträgt.

```
long *volatile *(*foo)(long bar);
```

Regel	**Bearbeitetes Token**	**Gelesener Text**
A	`foo`	`foo` is a
B.3	`*`	pointer to
B.1	`(...)`	*(keiner)*
B2.1	`(long)`	a function taking a long (named bar) and returning
B.3	`*`	a pointer to
B.4	`* volatile`	a volatile pointer to
B.6	`long`	a long

Aufgabe 7 (MPI) [14 Punkte]

(a) Analysieren Sie folgenden Ausschnitt aus einem MPI-Programm unter der Annahme, dass es mit 4 Prozessen ausgeführt wird. Geben Sie in untenstehender Tabelle an, welche Werte die Puffer `sendbuffer` und `recvbuffer` nach Ausführung von `MPI_Reduce` innerhalb der jeweiligen Prozesse enthalten. [4 Punkte]

```
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int sendbuffer[4];
int recvbuffer[4];

for (int i = 0; i < 4; i++) {
    sendbuffer[i] = rank + i;
}

MPI_Reduce(sendbuffer, recvbuffer, 4, MPI_INT,
           MPI_SUM, 0, MPI_COMM_WORLD);
```

`sendbuffer` bei Rank 0	0	1	2	3
`sendbuffer` bei Rank 1	1	2	3	4
`sendbuffer` bei Rank 2	2	3	4	5
`sendbuffer` bei Rank 3	3	4	5	6
`recvbuffer` bei Rank 0	6	10	14	18

(b) Implementieren Sie die kollektive Operation `MPI_Reduce` für das Aufsummieren von `int`-Arrays mit Hilfe der MPI-Funktionen `MPI_Send`, `MPI_Recv`, `MPI_Comm_size` und `MPI_Comm_rank`. Ergänzen Sie dazu den unten angegebenen Funktionsheader `my_int_sum_reduce`, so dass ein Aufruf der Funktion die Daten in derselben Weise verteilt, wie ein Aufruf von `MPI_Reduce` mit dem Parameterwert `MPI_SUM`. [10 Punkte]

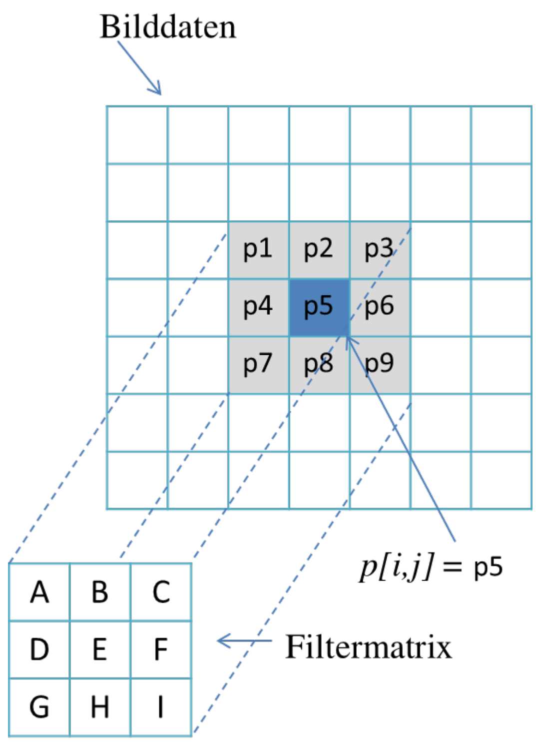
Hinweis: Sie dürfen davon ausgehen, dass `my_int_sum_reduce` nur mit gültigen Argumenten aufgerufen wird. Sie brauchen sich also nicht um Fehlerbehandlung aufgrund falscher Argumente zu kümmern.
Vermeiden Sie Aufrufe von `MPI_Send`, bei denen Sender und Empfänger identisch sind, da dies zu einem Deadlock führen kann.

```c
void my_int_sum_reduce(int *sendbuf, int *recvbuf, int count,
                       int root, MPI_Comm comm)
{

    int size, rank;
    MPI_Comm_size(comm, &size);
    MPI_Comm_rank(comm, &rank);

    if (rank == root) {
        /* Initialize recvbuf with our own values. */
        for (int i = 0; i < count; ++i)
            recvbuf[i] = sendbuf[i];

        /* Receive values from every other node and accumulate. */
        for (int i = 0; i < size; ++i) {
            if (i == root)
                continue;

            int other[count];
            MPI_Recv(other, count, MPI_INT, i, 0, comm, MPI_STATUS_IGNORE);
            for (int j = 0; j < count; ++j)
                recvbuf[j] += other[j];
        }
    } else {
        /* Send our values to root. */
        MPI_Send(sendbuf, count, MPI_INT, root, 0, comm);
    }

}
```

Aufgabe 8 (Scala – Digitale Filter) [12 Punkte]

Das Grundprinzip von digitalen Filtern ist relativ einfach: Eine sogenannte Filtermatrix wird über die zu filternden Daten (z.B. Bild- oder Audiodaten) geschoben und für jeden Datenpunkt wird die mit den Matrix-Werten gewichtete Summe seiner Umgebung berechnet.

Im zweidimensionalen Fall eines Bildes könnte das beispielhaft wie folgt aussehen:



Der gefilterte Wert für $p[i,j]$ berechnet sich im Beispiel also wie folgt:

$$p[i,j] = \frac{Ap_1 + Bp_2 + Cp_3 + Dp_4 + Ep_5 + Fp_6 + Gp_7 + Hp_8 + Ip_9}{A + B + C + D + E + F + G + H + I}$$

Um einen digitalen Filter nebenläufig zu implementieren, müssen Bild- und Filterdaten natürlich für alle Threads zugreifbar sein. Eine Möglichkeit dies zu erreichen, könnte eine Übergabe der benötigten Daten im Konstruktor sein, wie im folgenden Scalacode-Auszug angedeutet:

```
class Filter(image  : Array[Array[Float]],
             filter : Array[Array[Float]],
             column : Int) extends Actor {
  // ... Implementierung ...
}
```

(a) Erklären Sie kurz, welches grundlegende Prinzip zum Einsatz von Nebenläufigkeit in Scala hier verletzt wird, welche Probleme das verursachen kann und wie der Datenaustausch in nebenläufigen Scala-Programmen üblicherweise vonstattengehen soll. [4 Punkte]

Beispiellösung: Bei Scala-Aktoren gilt das „shared nothing“-Prinzip, d.h. es sollte kein Zugriff auf Objektzustände bzw. -daten aus verschiedenen Threads erfolgen, da ansonsten die üblichen Nebenläufigkeitsprobleme wie Deadlocks oder Race Conditions auftreten können. Scala verwendet daher üblicherweise den Austausch von Nachrichten.

(b) Erklären Sie (in Worten), wie Sie den oben gezeigten Filter-Algorithmus in Scala parallelisieren würden (natürlich unter Verwendung des korrekten Datenaustausch-Prinzips) und sprechen Sie dabei auch mögliche Komplikationen kurz (d.h. in Stichworten oder maximal einem Satz) an. [5 Punkte]

Beispiellösung: Zunächst wäre eine Anzahl von Actors anzulegen (typischerweise so viele, wie Cores vorhanden sind), die in ihrer act-Methode auf Nachrichten warten. Jeder dieser Actors erhält (s)einen Teil des Bildes zusammen mit der Filtermatrix und einer ID als Nachricht gesendet. Es bietet sich an, die Nachrichten in ein entsprechendes Data Transfer Object zu verpacken, das auch eine Referenz auf den gefilterten Bildteil enthält (es darf nicht wie oben einfach die Bildreferenz verschickt werden!).

Die Aufteilung der Bilddaten kann beispielsweise in Blöcke von Bildzeilen/Actors Zeilen erfolgen (ggf. auf Restzeilen achten). Dabei ist auch darauf zu achten, dass an den Blockrändern jeweils (Filterhöhe – 1) /2 Zeilen Überlapp für den Filter benötigt werden und an den Bildrändern eine Lösung gefunden wird, die dort fehlenden Bildpunkte zu „ersetzen“ (z.B. spiegeln des Rands o.ä.).

(c) Vervollständigen Sie abschließend die folgende Methode zur Berechnung der Summe der Filtergewichte im zweidimensionalen Raum, `filterSum` soll also den Nenner von $p[i, j]$ (s.o.) berechnen. [3 Punkte]

```
def filterSum(filter : Array[Array[Float]]) : Float = {
   var sum = 0.0f
   for (x <- 0 until filter.length) {
      for (y <- 0 until filter(x).length) {
         sum += filter(x)(y)
      }
   }
   return sum
}
```

Aufgabe 9 (Haskell, Compiler, Rekursiver Abstieg) [25 Punkte]

Gegeben sei die folgende Grammatik mit Startsymbol S (Terminale sind unterstrichen):

$$\begin{array}{rcl} S & \rightarrow & T \mid T\ \underline{\texttt{+}}\ S \\ T & \rightarrow & \underline{\texttt{value}} \mid \underline{\texttt{(}}\ S\ \underline{\texttt{)}} \end{array}$$

Die Grammatik beschreibt einfache arithmetische Ausdrücke, bei denen es neben Ganzzahlen nur Summen und geklammerte Ausdrücke gibt.

Nach Linksfaktorisierung erhält man:

$$\begin{array}{rcl} S & \rightarrow & T\ S' \\ S' & \rightarrow & \underline{\texttt{+}}\ S \mid \varepsilon \\ T & \rightarrow & \underline{\texttt{value}} \mid \underline{\texttt{(}}\ S\ \underline{\texttt{)}} \end{array}$$

(a) Geben Sie einen Haskell-Datentyp `Token` für die Tokens an. Gehen Sie davon aus, dass die lexikalische Analyse das Token für Zahlen mit einem **`Int`**-Wert versieht. [2 Punkte]

Beispiellösung:

```
data Token = LeftPar | RightPar | Value Int | Plus deriving Eq
```

(b) Geben Sie einen Haskell-Datentyp `Expr` für den abstrakten Syntaxbaum an. [3 Punkte]

Beispiellösung:

```
data Expr = Const Int | Add Expr Expr
```

(c) Implementieren Sie einen Parser mit rekursivem Abstieg für die linksfaktorisierte Grammatik in Haskell, der aus einer gegebenen Tokensequenz den abstrakten Syntaxbaum aufbaut. Geben Sie dazu Implementierungen der Parser-Funktionen `parseS'` und `parseT` an, deren Signaturen rechts vorgegeben sind. Brechen Sie bei Parsefehlern durch Aufruf der Funktion `stop :: t` ab. [20 Punkte]

Hinweis: Zur Orientierung ist die komplette Implementierung von `parseS` vorgegeben. Die Parser-Funktionen bekommen jeweils die aktuelle Tokenliste übergeben und liefern ein Tupel bestehend aus einem abstrakten Syntaxbaum und der Token-Restliste zurück. Die Funktion `parseS'` benötigt ein zusätzliches Argument vom Typ `Expr`.

Beispiellösung:

```
stop :: t
stop = error "parse␣error"

parseS :: [Token] -> (Expr, [Token])
parseS ts = let (exp, toks) = parseT ts
            in parseS' toks exp

parseS' :: [Token] -> Expr -> (Expr, [Token])
parseS' (Plus : toks)     lhs = let (rhs, toks') = parseS toks
                                in (Add lhs rhs, toks')
parseS' l                 e   = (e, l)

parseT :: [Token] -> (Expr, [Token])
parseT ((Value val) : ts) = (Const val, ts)
parseT (LeftPar : ts)     = let (exp, toks) = parseS ts
                            in if toks == [] || (head toks) /= RightPar
                               then stop
                               else (exp, tail toks)
parseT _                  = stop
```