Prof. Dr. Thomas Breuel
Dipl.-Inf. Patrick Michel (AG ST)
Dipl.-Ing. Ludwig Schmidt-Hackenberg

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

Lösungshinweise/-vorschläge zum Übungsblatt 6: Software-Entwicklung 1 (WS 2010/11)

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

Aufgabe 1 Rekursive Datentypen (freiwillige Zusatzaufgabe)

a) Schreiben Sie eine Funktion `partialEval :: Exp -> Exp`, welche einen Ausdruck soweit wie möglich auswertet.

```
partialEval :: Exp -> Exp
partialEval e@(Leaf _) = e
partialEval e@(Id _)   = e
partialEval e@(Err _)  = e
partialEval (Node op a b) = partialEval' op (partialEval a) (partialEval b)

partialEval' :: Op -> Exp -> Exp -> Exp
partialEval' _      (Err s  ) _         = Err s
partialEval' _      _         (Err s  ) = Err s
partialEval' Plus   (Leaf 0 ) x         = x
partialEval' Plus   x         (Leaf 0 ) = x
partialEval' Minus  x         (Leaf 0 ) = x
partialEval' Mult   (Leaf 1 ) x         = x
partialEval' Mult   x         (Leaf 1 ) = x
partialEval' Div    x         (Leaf 1 ) = x
partialEval' Div    x         (Leaf 0 ) = Err "division by zero"
partialEval' Plus   (Leaf a') (Leaf b') = Leaf (a' + b')
partialEval' Minus  (Leaf a') (Leaf b') = Leaf (a' - b')
partialEval' Mult   (Leaf a') (Leaf b') = Leaf (a' * b')
partialEval' Div    (Leaf a') (Leaf b') = Leaf (a' `div` b')
partialEval' op     a'        b'        = Node op a' b'
```

b) Definieren Sie einen geeigneten Datentyp `Env`, der eine Bezeichnerumgebung darstellt und damit Bezeichner (`String`) auf Werte (`Integer`) abbildet.

Schreiben Sie nun eine Funktion `eval :: Exp -> Env -> Exp`, welche einen Ausdruck mit Hilfe der übergebenen Bezeichnerumgebung komplett auswertet.

```
type Env = [(String, Integer)]

eval :: Exp -> Env -> Exp
eval e@(Leaf _) _      = e
eval e@(Err _) _       = e
eval (Id i) env        = evalLook i (lookup i env)
eval (Node op a b) env = eval' op (eval a env) (eval b env)

eval' :: Op -> Exp -> Exp -> Exp
eval' _      (Err s  ) _         = Err s
eval' _      (_      ) (Err s  ) = Err s
```

```
eval' Plus   (Leaf a') (Leaf b') = Leaf (a' + b')
eval' Minus  (Leaf a') (Leaf b') = Leaf (a' - b')
eval' Mult   (Leaf a') (Leaf b') = Leaf (a' * b')
eval' Div    (Leaf a') (Leaf b')
  | b' /= 0                      = Leaf (a' `div` b')
  | otherwise                    = Err "division by zero"
eval' _ _ _                      = error "internal error: unreachable case"

evalLook :: String -> Maybe Integer -> Exp
evalLook i Nothing  = Err ("not in scope: " ++ show i)
evalLook _ (Just v) = Leaf v
```

Aufgabe 2 Parametrische Typen (Präsenzaufgabe)

Geben Sie ohne die Hilfe des GHC(i) die Typen der Ausdrücke bzw. Funktionen an:

```
[True]                              -- [Bool]
[]                                  -- [a]
head                                -- [a] -> a
\xs -> tail xs                      -- [a] -> [a]
\xs -> tail (tail xs)               -- [a] -> [a]
\x -> x                             -- a -> a
\x -> (if x == [] then x else x)    -- Eq a => [a] -> [a]
\x y -> (y, x)                      -- a -> b -> (b, a)
f x y z = if x then y else z        -- Bool -> a -> a -> a
g x y z = if x == y then y else z   -- Eq a => a -> a -> a -> a
```

Aufgabe 3 Parametrische Typen (Einreichaufgabe)

a) Geben Sie zu jedem Typ eine Funktion an, die diesen Typ hat:

```
-- (a -> b) -> a -> b
f1 f x = f x                                -- ($)
-- a -> a
f2 x = x                                    -- id
-- Eq a => [a] -> Bool
f3 l = l == []                              -- (== [])
-- Eq a => a -> a
f4 x = if x == x then x else x              -- \x -> if x == x then x else x
-- (a -> b) -> [a] -> [b]
f5 _ []       = []                          -- map
f5 f (h : t) = f h : f5 f t
-- (a -> b -> c) -> [a] -> [b] -> [c]
f6 _ [] _ = []                              -- zipWith
f6 _ _ [] = []
f6 f (a : l) (b : m) = f a b : f6 f l m
-- (a -> b -> c) -> b -> a -> c
f7 f a b = f b a                            -- flip
-- (a -> b) -> (b -> c) -> (a -> c)
f8 f g x = g (f x)                          -- flip (.)
```

b) Bestimmen Sie ohne Hilfe des GHC(i) die Typen der Funktionen `h` bis `m`:

```
h :: Baum Char -> [Char]
i :: Baum a -> b -> b
j :: Baum a -> Baum a
k :: Maybe a -> Maybe b
l :: Baum a -> [a]
m :: Baum a -> Baum Int
```

Aufgabe 4 Unendliche Listen (Einreichaufgabe)

Lösen Sie diese Aufgabe ohne Bibliotheksfunktionen oder Kurzschreibweisen für Listen zu verwenden.

a) Implementieren Sie die Funktionalität der beiden Kurzschreibweisen `[n..]` und `[n, m..]` in Form von:

`fromN :: Integer -> [Integer]` und `fromNM :: Integer -> Integer -> [Integer]`

Beide Funktionen liefern also eine unendliche Liste von ganzen Zahlen, wobei durch die Angabe der zweiten Zahl `m` auch implizit die Schrittweite angegeben werden kann.

```
fromN :: Integer -> [Integer]
fromN n = n : fromN (n + 1)

fromNM :: Integer -> Integer -> [Integer]
fromNM start next = start : fromNM next (next + next - start)
```

b) Schreiben Sie eine Haskell-Funktion `summ :: [Integer] -> [Integer]`, welche für eine unendliche Liste die Liste der Summen benachbarter Zahlen als unendliche Liste zurückgibt.

```
summ :: [Integer] -> [Integer]
summ (a : b : r) = a + b : summ (b : r)
```

c) Definieren Sie den Wert `oneZero :: [Integer]`, welcher die Liste repräsentiert, die mit eins anfängt und dann unendlich viele Nullen enthält. Überlegen Sie sich, wie Sie mit diesem Startwert und der Funktion `summ` sehr simpel die Funktion `pascal` von Blatt 4 implementieren können.

```
oneZero :: [Integer]
oneZero = 1 : 0 : tail oneZero

pascal :: Int -> [Integer]
pascal 0 = oneZero
pascal n = summ (0 : pascal (n - 1))

-- for those who insist on cleaning up the zeros:
pascal' :: Int -> [Integer]
pascal' = takeWhile (> 0) . pascal
```

d) Schreiben Sie eine Haskell-Funktion `dropDivN :: Integer -> [Integer] -> [Integer]`, welche aus einer unendlichen Liste von Zahlen alle entfernt, die durch die übergebene Zahl `n` teilbar sind.

```
dropDivN :: Integer -> [Integer] -> [Integer]
dropDivN n (m : r) = if m `mod` n == 0
  then dropDivN n r else m : dropDivN n r
```

e) Verwenden Sie die Funktion `dropDivN`, um nach dem Prinzip des "Sieb des Eratosthenes" den Wert

`primes :: [Integer]`

zu definieren, der die unendliche Liste aller Primzahlen darstellt.

```
primes :: [Integer]
primes = primesHelp (fromN 2)  where

  primesHelp :: [Integer] -> [Integer]
  primesHelp (p : r) = p : primesHelp (dropDivN p r)
```

Aufgabe 5 Funktionen höherer Ordnung (Präsenzaufgabe)

a) Schreiben Sie eine Implementierung der Haskell-Funktion `map`, die also eine gegebene Funktion `f :: a -> b` auf jedes Element einer Liste vom Typ `[a]` anwendet und als Ergebnis entsprechend eine Liste vom Typ `[b]` erzeugt.

```
myMap :: (a -> b) -> [a] -> [b]
myMap _ []    = []
myMap f (h:t) = f h : myMap f t
```

Hier die beiden anderen Funktionen, auch wenn diese nicht gefragt sind:

```
myFoldl :: (a -> b -> a) -> a -> [b] -> a
myFoldl _ s []    = s
myFoldl o s (h:t) = myFoldl o (s `o` h) t

myFoldr :: (b -> a -> a) -> a -> [b] -> a
myFoldr _ s []    = s
myFoldr o s (h:t) = h `o` myFoldr o s t
```

b) Testen Sie Ihre Implementierung im GHCi gegen die aus `Prelude`, indem Sie mit beiden die Funktion `succ` über die Liste `[1..10]` mappen.

c) Geben Sie an, was die Aufrufe `foldl (+) 0 x` und `foldr (+) 0 x` für eine Liste `x` berechnen und wo der Unterschied im Verlauf der Berechnung liegt.

Sie berechnen beide die Summe der Liste. Dabei summiert `foldl` von links und `foldr` von rechts auf. Insbesondere arbeitet `foldl` beim rekursiven Abstieg und ist damit tailrekursiv, während `foldr` erst beim Aufstieg wirklich addiert. Da die Addition assoziativ ist ändert sich das Ergbenis nicht.

Was muss gelten, damit man eine Funktion in beiden Varianten von `fold` verwenden kann?

Die Funktion muss sowohl links- als auch rechtsrekursiv anwendbar sein (also ihr eigenes Ergebnis links bzw. rechts als Parameter nehmen können). Damit muss die Funktion zwei Parameter vom selben Typ nehmen und auch diesen Typ zurückliefern.

(Damit auch noch das gleiche Ergebnis herauskommt muss die Operation assoziativ sein und das Startelement muss sowohl links- als auch rechtsneutral bzgl. dieser Operation sein.)

Aufgabe 6 Funktionen höherer Ordnung (Einreichaufgabe)

a) Schreiben Sie eine Haskell-Funktion `quadrieren :: [Integer] -> [Integer]` mit Hilfe von `map`, die eine Liste von Zahlen als Parameter nimmt und eine Liste mit deren Quadraten zurückgibt.

```
quadrieren :: [Integer] -> [Integer]
quadrieren l = map (\x -> x * x) l
```

b) Schreiben Sie eine Haskell-Funktion `filtern :: (a -> Bool) -> [a] -> [a]`, die eine Prädikatsfunktion als Parameter nimmt und, angewandt auf eine Liste, alle Elemente dieser Liste entfernt, die dieses Prädikat erfüllen – also alle Elemente auswählt, die es nicht erfüllen.

```
filtern :: (a -> Bool) -> [a] -> [a]
filtern _ []    = []
filtern p (h:t) = if p h then filtern p t else h : filtern p t
```

c) Schreiben Sie eine Haskell-Funktion `partitionieren :: (a -> Bool) -> [a] -> ([a], [a])`, die ein Prädikat als Parameter nimmt und, angewandt auf eine Liste, zwei Listen zurückgibt, wobei erstere alle Elemente enthält, die das Prädikat erfüllen, und die andere die restlichen Elemente enthält.

```
partitionieren :: (a -> Bool) -> [a] -> ([a], [a])
partitionieren _ [] = ([], [])
partitionieren p (h:t) = let (a, b) = partitionieren p t in
  if p h then (h : a, b) else (a, h : b)
```

d) Schreiben Sie eine Haskell-Funktion `maximumGem :: (a -> Integer) -> [a] -> a`, die angewandt auf eine Liste, das Maximum gemäß einer gegebenen Bewertungsfunktion ermittelt.

```
maximumGem :: (a -> Integer) -> [a] -> a
maximumGem _ [x] = x
maximumGem f (a : r) = let b = maximumGem f r in
  if f a < f b then b else a
```

e) Schreiben Sie mit Hilfe von `foldl` eine Haskell-Funktion `laenge :: [a] -> Int`, die die Länge einer Liste bestimmt.

```
laenge :: [a] -> Int
laenge = foldl (\n _ -> n + 1) 0
```

Die Implementierung vieler Higher-Order Funktionen lässt sich dadurch abkürzen, dass die "hintersten" formalen Parameter nicht benötigt werden. In diesem Fall wird das Symbol `laenge` durch den Ausdruck auf der rechten Seite definiert, was eine partielle Anwendung der `foldl` Funktion ist. Die Notation nennt sich *Point-Free*.

Die Definition ist aber natürlich (im wesentlichen) äquivalent zu dieser Schreibweise, mit explizitem weiteren Parameter:

```
laenge :: [a] -> Int
laenge l = foldl (\n _ -> n + 1) 0 l
```

f) Schreiben Sie mit Hilfe von `foldl` die Funktion `rev :: [a] -> [a]`, die eine Liste umkehrt.

```
rev :: [a] -> [a]
rev = foldl (flip (:)) []
```

g) Schreiben Sie die Funktionen `und` und `oder` von Blatt 4 mit Hilfe von `foldl`.

```
und :: [Bool] -> Bool
und = foldl (&&) True

oder :: [Bool] -> Bool
oder = foldl (||) False
```

h) Schreiben Sie die Funktion `sequentiell :: [Programm] -> Programm` mit Hilfe von `foldl1`, welche eine nicht-leere Liste von Roboterbefehlen in ein Programm umwandelt, das diese sequentiell ausführt.

```
sequentiell :: [Programm] -> Programm
sequentiell = foldl1 (:>)
```

i) (*freiwillige Zusatzaufgabe*) Schreiben Sie die Funktion `map` mit Hilfe von `foldl`.

```
map :: (a -> b) -> [a] -> [b]
map f = foldl (\l e -> l ++ [f e]) []
```

j) (*freiwillige Zusatzaufgabe*) Schreiben Sie die Funktion `maximumGem` mit Hilfe von `foldl1`.

```
maximumGem' ::  (a -> Integer) -> [a] -> a
maximumGem' f = foldl1 (\a b -> if f a < f b then b else a)
```

k) (*freiwillige Zusatzaufgabe*) Optimieren Sie die Darstellung der Funktion `permutations` – und ihrer Hilfsfunktionen – von Blatt 5, indem Sie Funktionen höherer Ordnung der Standardbibliothek verwenden, wie unter anderem `map` und `fold`.

```
augment e l        = map (e:) l
everywhere n l     = zipWith3 (\a b c -> a ++ b : c) (inits l) (repeat n) (tails l)
everywhereList n l = concatMap (everywhere n) l
permutations []    = []
permutations l     = foldr everywhereList [[]] l
```

l) (*freiwillige Zusatzaufgabe*) Optimieren Sie auf dieselbe Art die Darstellung der Funktion `primfaktoren` von Blatt 4, wo beispielsweise `filter` oder `find`, zusammen mit der Liste aller potentiellen Teiler, interessant sein können.

```
primfaktoren :: Integer -> [Integer]
primfaktoren x = let c = head $ filter (\y -> x `mod` y == 0) [2 .. x] in
  if c == x then [x] else c : primfaktoren (x `div` c)

-- oder

primfaktoren' :: Integer -> [Integer]
primfaktoren' x = maybe [x] (\c -> c : primfaktoren' (x `div` c)) $
  find (\y -> x `mod` y == 0) [2 .. x `div` 2]

-- oder

primfaktoren'' :: Integer -> [Integer]
primfaktoren'' = unfoldr faktor  where

  faktor :: Integer -> Maybe (Integer, Integer)
  faktor 1 = Nothing
  faktor x = let c = head $ filter (\y -> x `mod` y == 0) [2 .. x]
    in Just (c, x `div` c)
```

Aufgabe 7 Lexing und Parsing in Haskell (Einreichaufgabe)

In dieser Aufgabe geht es darum einfache Grammatiken und deren Sprachen in Haskell umzusetzen. Verwenden Sie zur Lösung dieser Aufgabe möglichst keine Bibliotheksfunktionen.

a) Schreiben Sie eine Funktion `paren :: String -> Bool`, welche alle Strings über dem Alphabet {'(', ')'} akzeptiert, die korrekt geklammert sind. Geben Sie ebenfalls an, wie sich Ihre Lösung zu der folgenden Grammatik in Beziehung setzt, die diese Sprache erzeugt: $\Gamma = \left(\{S\}, \{'(', ')'\}, \{S \to (S)S, S \to \epsilon\}, S\right)$

```
paren :: String -> Bool
paren w = null (s w)  where

  s, s' :: String -> String
  s ('(' : r) = s (s' (s r))
  s w = w

  s' (')' : r) = r
  s' _ = "x"
```

```
paren' :: String -> Bool
paren' w = s 0 w  where

  s :: Int -> String -> Bool
  s n ('(' : r) = s (n + 1) r
  s 0 (')' : r) = False
  s n (')' : r) = s (n - 1) r
  s 0 ""        = True
  s _ _         = False
```

Die linke Variante funktioniert nach dem Prinzip, dass man für jedes Nichtterminal Funktionen schreibt, welche von der Eingabe den Präfix entfernen, der von diesem Nichtterminal legal erzeugt werden kann und den Rest zurück gibt. Ist es unmöglich einen solchen Präfix zu finden, gibt man einfach eine nicht legale Resteingabe zurück. Die Hauptfunktion muss dann nur noch testen, dass die Eingabe komplett konsumiert wurde.

Die beiden Grammatikregeln spiegeln sich in den beiden Fällen der Funktion s wider, also bei einer Klammer wird die entsprechende Produktion aktiviert, ansonsten wird davon ausgegangen, dass die Epsilon Produktion genutzt wurde. Entsprechend wird danach wieder rekursiv ein s verarbeitet, worauf dann aber sicher eine Klammer folgen muss. Danach wird noch einmal ein s gelesen, was ja aber auch leer sein kann.

Die zweite Variante verwaltet die Verschachtelungstiefe (sprich die Rekursionen auf dem *ersten* S) und löst die Umsetzung mit einem look-ahead. Das heißt, wenn eine öffnende Klammer kommt aktivieren wir die erste Produktion und gehen in die Rekursion. Kommt eine schließende, kann nur noch der Epsilon Fall zutreffen und wir müssen die aktuelle Inkarnation der Produktion beenden. Um das folgende S zu konsumieren, gehen wir wieder in die Rekursion, allerdings *ohne* die Tiefe zu ändern. Befinden wir uns in keiner Inkarnation, wenn eine schließende Klammer kommt, ist dies kein Wort der Sprache. Ist die Eingabe zu Ende muss ebenfalls die Epsilon Variante gewählt sein und wir dürfen uns nicht mehr in einer linken Rekursion befinden.

Also kurz gesagt, ohne die Parallelen zur Grammatik: Wir zählen die Differenz aus öffnenden und schließenden Klammern, am Ende muss die Zahl Null sein und zwischenzeitlich darf sie nie negativ sein.

b) Schreiben Sie eine Funktion `csv :: String -> Maybe [String]`, die eine Liste von sogenannten *comma-separated values* in eine Liste von Strings überführt.

Geben Sie zusätzlich zu dieser Funktion eine Grammatik an, welche die beschriebene Sprache erzeugt, und schildern Sie auch hier den Zusammenhang zu Ihrer Lösung.

```
csv :: String -> Maybe [String]
csv = s  where

  s word = s' (t word)
  s' (Just ("", value)) = Just [value]
  s' (Just (',' : rest, value)) = s'' value (s rest)
  s' _ = Nothing
  s'' value (Just values) = Just (value : values)
  s'' _ _ = Nothing

  t ('"' : rest) = t' (u (rest, ""))
  t _ = Nothing
  t' ('"' : rest, value) = Just (rest, reverse value)
  t' _ = Nothing

  u (c : rest, value) | c /= '"' = u (rest, c : value)
  u x = x
```

Die Grammatik dazu ist $\Gamma = \left(\{S, T, U\}, \texttt{Char}, \Pi, S\right)$

$$\Pi = \left\{ \begin{array}{lcll} S & \rightarrow & T & \\ S & \rightarrow & T, S & \\ T & \rightarrow & "U" & \\ U & \rightarrow & \epsilon & \\ U & \rightarrow & cU & \forall c \in \texttt{Char} \setminus \{"\} \end{array} \right\}$$

Mit der Grammatik werden nur nicht leere Ketten beschrieben, was aber im Sinne der Sache sein sollte. Besonders zu achten ist auf den Umgang mit Kommas, dass diese wirklich nur zwischen Elementen auftauchen. Die Lösung verwendet `Maybe`, um die Analyse mit Fehler abzubrechen und die Funktionen geben jeweils sowohl ihr Ergebnis, als auch den Rest der Eingabe, zurück. Die Funktion `s` konsumiert also ein `T` und ruft dann mit dem Ergebnis und Rest `s'` auf. Ist die Eingabe dann zu Ende wurde die erste Produktion gewählt und wir sind fertig mit einer Singleton Liste. Kommt hingegen ein Komma wird wieder ein `S` eingelesen und wir rufen `s''` auf, um die Ergebnisse zu kombinieren und eventuelle Fehler zu behandeln. Die Funktion `t` erwartet ein Anführungszeichen und liest dann ein `U`. `t'` wird benutzt um schließlich auch das abschließende Anführungszeichen einzufordern. In `u` schließlich kann durch die Epsilon Produktion nichts schief gehen und es liest alle Elemente außer ". Ist die Eingabe beendet liefert die Funktion allerdings auch keinen Fehler, das wird dann aber von `t` direkt korrigiert. Zu beachten in der Lösung ist sonst nur, dass das Ergebnis von `u` invertiert aufgebaut wird und nur am Ende in `t` einmal umgedreht wird, anstatt in `u` andauernd hinten an einen String anzuhängen.

c) Schreiben Sie die inverse Funktion `toCSV :: [String] -> String`, welche eine Liste von Strings ins CSV Format umwandelt. Überlegen Sie sich für welche Eingabe die folgende Gleichung nicht erfüllt ist und lösen Sie das Problem, indem Sie sowohl `csv`, also auch `toCSV` entsprechend sinnvoll anpassen:

Das Problem hier ist, dass Anführungszeichen in einem String nicht anders behandelt werden und beim Einlesen alles zerschießen oder zumindest das Ergebnis ändern. Lösen kann man dies durch einen Escape-Character, wie zum Beispiel \, wobei dieser selbst dann escaped werden muss (mit sich selbst), sonst hat man das gleiche Problem gleich wieder! Die Lösung hier verwendet außerdem nun cases anstatt Funktionsgleichungen, was ein einfach ein anderer Programmierstil ist.

```
toCSV :: [String] -> String
toCSV [] = ""
toCSV [x] = '"' : escape x ++ ['"']
toCSV (h : t) = toCSV [h] ++ "," ++ toCSV t

escape :: String -> String
escape "" = ""
escape ('"' : rest) = "\\\"" ++ escape rest
escape ('\\' : rest) = "\\\\" ++ escape rest
escape (c : rest) = c : escape rest

csv' :: String -> Maybe [String]
csv' = s  where

  s word = case t word of
    Just ("", value) -> Just [value]
    Just (',' : rest, value) -> case s rest of
      Just values -> Just (value : values)
      _ -> Nothing
    _ -> Nothing

  t ('"' : rest) = case v (rest, "") of
    ('"' : rest', value) -> Just (rest', reverse value)
    _ -> Nothing
  t _ = Nothing

  v ('\\' : '"' : rest, value) = v (rest, '"' : value)
  v ('\\' : '\\' : rest, value) = v (rest, '\\' : value)
  v (c : rest, value) | c /= '"' = v (rest, c : value)
  v x = x
```