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

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

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

Ausgabe: in der Woche vom 29.11. bis zum 03.12.10
Abgabe: in der Woche vom 06.12. bis zum 10.12.10
Abnahme: max. zwei Tage nach der Übung

Bitte beachten Sie folgenden Hinweis für **dieses und die weiteren** Übungsblätter:

*Geben Sie Quelldateien, die Sie zur Lösung einer Aufgabe erstellen, von nun an immer in **ausgedruckter** Form ab und schicken Sie sie zusätzlich **per Email** an Ihren Betreuer. Alle anderen Lösungen zu Aufgaben sind weiterhin in **handschriftlicher** Form abzugeben!*

Aufgabe 1 Rekursive Datentypen (freiwillige Zusatzaufgabe)

Wir betrachten eine Erweiterung von Aufgabe 5 auf Blatt 5:

```
data Exp =
    Leaf Integer
  | Node Op Exp Exp
  | Id String
  | Err String
  deriving (Eq, Ord, Show)

data Op = Mult | Div | Plus | Minus
  deriving (Eq, Ord, Show)
```

Es ist nun möglich Bezeichner (Konstruktor `Id`) in Ausdrücken zu verwenden, die durch die Umgebung an einen Wert gebunden sein müssen. Zusätzlich kann nun die Fehlerbehandlung bei der Auswertung direkt in den Ausdrücken geschehen. Der Konstruktor `Err` symbolisiert eine abrupte Terminierung mit einem im `String` angegebenem Grund.

Laden Sie sich für diese Aufgabe die neue Version der Datei "`Exp.hs`" von der Vorlesungsseite. Sie definiert die Datentypen, stellt Ihnen eine neue `parse` Variante zur Verfügung und sorgt dafür, dass Ausdrücke automatisch in Infix-Notation ausgegeben werden. Sie können außerdem die Funktion `someExp` verwenden, die aus einer ganzen Zahl einen zufälligen Ausdruck generiert, indem die Variablen `a, b, c, x, y` und `z` vorkommen können.

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

*Hinweis: Denken Sie daran, dass Sie nicht nur Operationen auf Konstanten auswerten können, sondern `Plus` und `Mult` links- sowie rechtsneutral bzgl. `0` bzw. `1` sind. Zusätzlich sind `Minus` und `Div` rechtsneutral bzgl. `0` bzw. `1`. Multiplikation und Division mit `0` lassen sich hingegen **nicht** immer vereinfachen, da der entsprechend andere Operand einen Fehler liefern könnte!*

Hinweis: Denken Sie auch daran, dass eine Operation einen Fehler erzeugt, sobald einer der beiden Operanden einen Fehler erzeugt. Falls beide einen Fehler erzeugen können Sie diese entweder zusammenfassen oder einfach einen von beiden weitergeben.

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.

Hinweis: Die Funktion bildet nicht direkt auf `Integer` ab, da eine Auswertung auch zu `Error` führen kann. Trotzdem erhalten Sie die Garantie, dass eine Auswertung immer zu einem der beiden Konstrukte führt. Leiten Sie Ihre `eval` Implementierung her, indem Sie Ihre `partialEval` Implementierung kopieren und von dieser Garantie Gebrauch machen. Denken Sie auch daran, dass ein ungebundener Bezeichner nun eine neue Fehlerquelle darstellt.

Aufgabe 2 Parametrische Typen (Präsenzaufgabe)

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

```
[True]
[]
head
\xs -> tail xs
\xs -> tail (tail xs)
```

```
\x -> x
\x -> (if x == [] then x else x)
\x y -> (y, x)
f x y z = if x then y else z
g x y z = if x == y then y else z
```

Aufgabe 3 Parametrische Typen (Einreichaufgabe)

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

```
(a -> b) -> a -> b
a -> a
Eq a => [a] -> Bool
Eq a => a -> a
```

```
(a -> b) -> [a] -> [b]
(a -> b -> c) -> [a] -> [b] -> [c]
(a -> b -> c) -> b -> a -> c
(a -> b) -> (b -> c) -> (a -> c)
```

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

```
data Baum a =
    Blatt
  | Zweig [a] (Baum a) (Baum a)

h Blatt           = "c"
h (Zweig "x" l r) = "x"
h (Zweig  x  l r) =  x
```

```
i (Zweig  x  l r) = id
j (Zweig [ ] l r) =  l
k Nothing         = Nothing
l (Zweig a b c)   = a
m (Zweig x l r)   =
  Zweig [length x] (m l) (m r)
```

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.

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

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.

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.

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

```
primes :: [Integer]
```

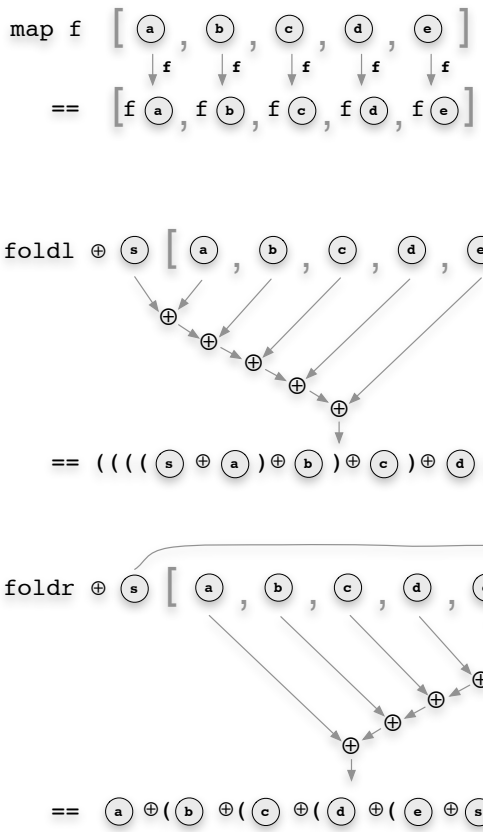
zu definieren, der die unendliche Liste aller Primzahlen darstellt.

Aufgabe 5 Funktionen höherer Ordnung (Präsenzaufgabe)

In dieser Aufgabe wollen wir uns mit den Typsignaturen und vor allem der Semantik der Funktionen

```
map :: (a -> b) -> [a] -> [b]
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (b -> a -> a) -> a -> [b] -> a
```

beschäftigen. Die nebenstehende Grafik verdeutlicht die Effekte der drei Funktionen.



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.

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.

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

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 [1..6]  ==  [1, 4, 9, 16, 25, 36]
```

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 even [1..10]  ==  [1, 3, 5, 7, 9]
filtern odd  [1..10]  ==  [2, 4, 6, 8, 10]
```

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 odd [1..10]  ==  ([1, 3, 5, 7, 9], [2, 4, 6, 8, 10])
```

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

```
maximumGem id          [1..10]  ==  10
maximumGem (`mod` 7)   [1..10]  ==   6
```

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

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

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

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.

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

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

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`.

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.

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 "(()(()))"  ==  True
paren "(()))())"  ==  False
```

b) Schreiben Sie eine Funktion `csv :: String -> Maybe [String]`, die eine Liste von sogenannten *comma-separated values* in eine Liste von Strings überführt. Elemente der Sprache sollen für diese Aufgabe alle Sequenzen sein, die aus durch *Komma* getrennten *Werten* bestehen, wobei ein *Wert* eine *Zeichenreihe* ist, die von *Anführungszeichen* umschlossen ist. Die Funktion `csv` soll `Nothing` zurückliefern, falls die Eingabe nicht zur Sprache gehört. Die Anführungszeichen gehören nicht zum Wert, der schließlich in der Ergebnisliste steht.

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

Beispiele für Elemente der Sprache:

```
"a","b","cde"          "Max","Mustermann"          ""          "","  ,  ","  ","  "
```

Hinweis: Nehmen Sie als Menge der Terminalsymbole für Ihre Grammatik einfach `Char`. Sie dürfen mit entsprechend sinnvoller Notation auch Produktionen angeben, die "für alle `Char`, außer `x`" gelten.

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:

```
test :: [String] -> Bool
test x =   csv (toCSV x)  ==  Just x
```