7. Übungsaufgabe zu
Fortgeschrittene funktionale Programmierung
Thema: Parsing (mit Kombinatoren und Monaden)
ausgegeben: Di, 06.06.2017, fällig: Di, 13.06.2017

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der im folgenden angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `AufgabeFFP7.hs` auf **oberstem Niveau in Ihrem Gruppenverzeichnis** ablegen. Kommentieren Sie Ihre Programme aussagekräftig und benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

- In Kapitel 13.1 ist eine univervelle Parser-Basis in Form einer Parser-Bibliothek bestehend aus 4 Grundparsern und 3 Kombinatoren über dem Parser-Typ

  ```
  type Parse0 a b = [a] -> [(b,[a])] -- Parse0 statt Parse, um
                                     -- Namenskonflikte mit anderen
                                     -- Teilaufgaben zu vermeiden
  ```

  angegeben.

  Weiters ist folgender Haskell-Typ für arithmetische Ausdrücke eingeführt:

  ```
  data Expr = Lit Int | Var Char | Op Ops Expr Expr deriving (Eq,Ord,Show)
  data Ops  = Add | Sub | Mul | Div | Mod deriving (Eq,Ord,Show)
  ```

  sowie zwei Funktionen `parser` und `topLevel` mit den Signaturen

  ```
  parser   :: Parse0 Char Expr
  topLevel :: Parse0 a b -> [a] -> b
  ```

  die es erlauben, wohlgeformte Ausdrücke der Form `"((234+~42)*b)"` in entsprechende Ausdrücke des Typs `Expr` zu überführen: `Op Mul (Op Add (Lit 234) (Lit -42)) (Var 'b')`

  - Implementieren Sie die Parser-Basis und die Funktionen `parser` und `topLevel` aus Kapitel 13.1. Ergänzen Sie dabei die Implementierungen der in Kapitel 13.1 nicht ausprogrammierten Hilfsfunktionen.
  - Testen Sie Ihre Implementierung anhand ausgewählter wohlgeformter Ausdrücke durch geeignete Aufrufe der Funktion `topLevel`, ob wohlgeformte Ausdrücke wie gewünscht erkannt und in Werte des Typs `Expr` überführt werden.

    Dabei gelten für Argumentausdrücke der Form `"(234+~42)*b"` die Wohlgeformtheitsannahmen von Folie 985 (Stand Folien vom 06.06.2017) und zusätzlich, dass das Symbol ~ als negatives Vorzeichen entweder gar nicht oder genau einmal vor einer Zifffernfolge (Literalwert) steht: `~~~42` ist also kein wohlgeformter Ausdruck.

– Entwickeln Sie aus den Funktionen `parser` und `topLevel` zwei Funktionen

```
parserWSp   :: Parse0 Char Expr
topLevelWSp :: Parse0 a b -> [a] -> b
```

mit gleicher Funktionalität, allerdings sollen wohlgeformte Ausdrücke nun in üblicher Weise Leerzeichen enthalten dürfen (Literale und Variablennamen dürfen weiterhin und wie üblich keine Leerzeichen enthalten, d.h. Literale und Variablennamen enden spätestens mit dem Zeichen vor dem ersten auf sie folgenden Leerzeichen).

– Testen Sie Ihre Implementierung anhand ausgewählter wohlgeformter Ausdrücke durch geeignete Aufrufe der Funktion `topLevelWsp`, ob wohlgeformte Ausdrücke, die in üblicher Weise Leerzeichen enthalten dürfen, wie gewünscht erkannt und in Werte des Typs `Expr` überführt werden.

- In Kapitel 13.2 wird ein monadischer Parser über dem Parser-Typ

```
newtype Parser a = Parse (String -> [(a,String)])
```

eingeführt und zur Erkennung und Auswertung wohlgeformter arithmetischer Ausdrücke, die gemäß der Regeln der Grammatik

```
expr   ::=  expr addop term | term
term   ::=  term mulop factor | factor
factor ::=  digit | (expr)
digit  ::=  0 | 1 | ... | 9

addop  ::=  + | -
mulop  ::=  * | /
```

gebildet sind, eingesetzt.

– Implementieren Sie den Parser aus Kapitel 13.2, insbesondere die Funktion `expr :: Parser Int` und ergänzen Sie dabei die Implementierungen möglicherweise nicht ausprogrammierter Hilfsfunktionen.

– Testen Sie Ihre Implementierung anhand ausgewählter wohlgeformter Ausdrücke durch geeignete Aufrufe der Form `apply expr "1-2*3+4"`, wobei die Argumentausdrücke an den üblichen Stellen auch Leerzeichen enthalten dürfen wie etwa im Ausdruck `"  1- 2* 3 +4 "`.

- Implementieren Sie eine Variante des monadischen Parsers aus Kapitel 13.2 über dem Parser-Typ

```
newtype Parser2 a b = Parse2 ([a] -> [(b,[a])])
```

d.h., machen Sie den Typkonstruktor `(Parser2 a)` zur Instanz von Monad. Listen von `a`-Werten stellen dabei den zu *parsende* Liste von Objekten dar, `b`-Werte sind von der Analyse erkannte Objekte.

Benutzen Sie folgende Namenskonvention: Unterscheiden Sie die Namen, die in Teilaufgabe 2) verwendet wurden, von denen in Teilaufgabe 3), in dem Sie eine "2äls Postfix an den entsprechenden Namen aus Teilaufgabe 2) anhängen entsprechend dem Beispiel von `newtype Parser2 a b = Parse2 ([a] -> [(b,[a])])`.

- Testen Sie Ihren modifizierten Parser anhand wohlgeformter Ausdrücke
  - gemäß Teilaufgabe 2) (`apply2a`)
  - vom Typ `Expr`, die keine Teilausdrücke der Form `Var 'x'` enthalten und keine negativen `Lit`-Werte (`apply2b`).

    Ausdrücke vom Typ `Expr` sollen dabei als Werte vom Typ `[Int]` vorliegen, wobei folgende Codierung zugrunde gelegt werden soll:

    Wenn `exp` ein gültiger Wert vom Typ `Expr` ist, und wenn s diejenige Zeichenreihe ist, die der Aufruf `show exp` liefert, so entsteht die Codierung von s in Form einer Liste von `Int`-Werten, indem die passenden Teilzeichenreihen von s gemäß folgender Tabelle umgesetzt werden:

Teilzeichenreihe von s	Codierung als `Int`-Wert
"Lit"	-1
Ziffernfolge	dargestellter `Int`-Wert >= 0
"Var"	Hier ausgeschlossen
"Op"	-3
"Add"	-4
"Sub"	-5
"Mul"	-6
"Div	-7
"Mod"	-8
"("	-9
")"	-10

Sehen Sie entsprechende Funktionen `apply2a` und `apply2b` vor, die den unterschiedlichen und insbesondere gegenüber Teilaufgabe 2) geänderten Typen Rechnung tragen.