Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Lars Noschinski, Dr. Jasmin Blanchette, Dmitriy Traytel

WS 2012/13
08.01.2013
Abgabe: 15.01.2013, 15:30

Einführung in die Informatik 2
11. Übung

Aufgabe G11.1 Unscramble schlägt zurück

Der Datentyp

```
data Either a b = Left a | Right b
```

ist neben `Maybe` eine der häufig verwendeten Möglichkeiten, Fehlerzustände zu signalisieren. Es ist Konvention, dass der `Left`-Konstruktor für den Fehlerwert und der `Right`-Konstruktor für einen korrekten Rückgabewert verwendet wird (Eselsbrücke: *right* ↔ *richtig*).

In Aufgabe H7.3 sollten Sie eine Funktion

```
unscrambleWords :: [String] -> [String] -> [String]
```

schreiben, die ein Wörterbuch `vocabs` und eine Liste von potentiell falsch geschriebenen Wörten `text` nimmt und einen korrigierten Text zurück gibt. Ein Wort ist hier *falsch geschrieben*, wenn es nicht in der Vokabelliste *vocabs* steht. Falls (mindestens) eine Permutation eines falsch geschriebene Wort im Wörterbuch vorkommt, wird das Wort durch eine seiner Permutationen ersetzt.

Dies Funktion hat das Problem, dass man dem Ergebnis nicht ansehen kann, welche Wörter korrekt waren, welche korrigiert und welche nicht korrigiert wurden, weil kein passendes Wort im Wörterbuch vorkommt.

Schreiben Sie daher eine Funktion

```
safeUnscrambleWords :: [String] -> [String]
    -> [Either (String, [String]) String]
```

die einen korrigierten Text wie folgt zurückliefert: Ist ein Wort w falsch geschrieben, so wird es durch `Left (`w`,` vs`)` ersetzt, wobei vs die Liste der in *vocabs* vorkommenden Permutationen von w ist. Andernfalls wird w durch `Right` w ersetzt.

Bauen Sie gegebenfalls auf die Musterlösung für H7.2 auf.

Beispiele:

```
vocabs = ["i", "kyoto", "more", "now", "tokyo", "want", "won"]

safeUnscrambleWords vocabs ["i", "want", "mroe"] ==
  [Right "i", Right "want", Left ("mroe", ["more"])]
safeUnscrambleWords vocabs ["and", "i", "tnaw", "it", "now"] ==
  [Left ("and", []) , Right "i", Left ("tnaw", ["want"]),
   Left ("it", []), Right "now"]
safeUnscrambleWords vocabs ["tokyo", "toyko", "kyoto"] ==
```

```
[Right "tokyo", Left ("toyko", ["kyoto","tokyo"]),
 Right "kyoto"]
```

Aufgabe G11.2 To parse, **or** not to parse, that is the question

Erweitern Sie den Parser für boolesche Formeln (Modul `FormParser2`) um das Parsen von Disjunktionen. Hierfür erweitern Sie zunächst den Token-Datentyp um einen weiteren Konstruktor `OrT`. Der Scanner soll das Symbol "`|`" in das neue Token übersetzen.

Natürlich wollen wir eine sinnvolle Klammerung im Ergebnis erhalten. Sinnvoll bedeutet hier, dass "`&`" stärker bindet als "`|`". D.h. `x1 & x2 | x3` soll als `(x1 & x2) | x3` geparst werden und nicht als `x1 & (x2 | x3)`. Dies erreichen Sie indem Sie drei Parse-Funktionen benutzen, die sich gegenseitig aufrufen:

- eine zum Parsen der Formeln der Form φ `|` ψ, wobei φ (auf der obersten Ebene) keine Disjunktion ist;
- eine zum Parsen der Formeln der Form φ `&` ψ, wobei φ (auf der obersten Ebene) keine Konjunktion und keine Disjunktion ist;
- eine zum Parsen der Formel φ, wobei φ eine Konstante (`T` oder `F`), ein Literal (`x` oder `~x`), oder eine geklammerte beliebige boolesche Formel ist.

Beachten Sie, dass das Modul `FormParser2` die letzten zwei Funktionen bereits implementiert.

Aufgabe G11.3 Identifiers, die diesen Namen verdienen

Die vom Modul `FormParser2` unterstützten Formeln dürfen nur Identifiers der Form

$$a_1 \dots a_m d_1 \dots d_n \qquad \text{für } m \geq 1 \text{ und } n \geq 0$$

enthalten. Erweitern Sie den Tokenizer, sodass alle Kombinationen von Buchstaben (`a`–`z`, `A`–`Z`), Zahlen (`0`–`9`), Unterstrichen (`_`) und Dollarzeichen (`$`), die mit einem Nicht-Zahl-Zeichen anfangen, erlaubt sind.

Aufgabe H11.1 Zahlenlesen (10 Punkte)

In dieser Aufgabe wollen wir einige Parser für unterschiedliche Zahlenformate definieren. Für alle diese Parser soll wie auch in der Vorlesung gelten: Kann der Parser keine Zahl parsen, gibt er `Nothing` zurück; ist nur ein Prefix der Eingabe eine Zahl, so wird nur dieses Prefix verarbeitet und der Rest der Eingabe wird weitergegeben. Zum Beispiel für den Parser aus der ersten Teilaufgabe:

```
nat "text" == Nothing
nat "10p" == Just (10, "p")
```

1. Definieren Sie einen Parser `nat :: Parser Char Int`, der Strings von Ziffern als natürliche Zahlen inklusive der 0 einliest. Zum Beispiel:

```
nat "123" == Just (123, [])
nat "012" == Just (12, [])
nat "0" == Just (0, [])
nat "0000" == Just (0, [])
```

Sie dürfen die Funktion `digitToInt :: Char -> Int` zum Umwandeln einzelner Ziffern nach `Int` benutzen. Dagegen sollen die Funktion `read :: Read a => [Char] -> a` *nicht* verwenden (auch nicht in den nachfolgenden Teilaufgaben).

2. Definieren Sie einen Parser `int :: Parser Char Int`, der Strings von Ziffern mit einem eventuell davorgestellten `'-'`-Zeichen als ganze Zahlen einliest. Zum Beispiel:

```
int "123" == Just (123, [])
int "-123" == Just (-123, [])
int "-012" == Just (-12, [])
int "0" == Just (0, [])
int "-0" == Just (0, [])
```

3. Definieren Sie einen Parser `double :: Parser Char Double` zum Einlesen von Gleitkommazahlen der Form $x.y$. Hierbei sollte x als ganze Zahl (also als natürliche Zahl mit optionalem Minuszeichen) und y als eine natürliche Zahl einlesbar sein. Die eingelesene Zahl soll insgesamt den Wert $(\hat{x} + \hat{y} \cdot 10^{-|y|})$ haben, wobei $\hat{x}$ (bzw. $\hat{y}$) der Wert der von x (bzw. y) dargestellten Zahl und $|y|$ die Anzahl der Zeichen von y ist.

```
double "-123.2" == Just (-123.2, [])
double "0.01" == Just (0.01, [])
double "-0.0001" == Just (-0.0001, [])
double "-0005.0000000000000" == Just (-5.0, [])
```

Sie dürfen die Funktion `fromIntegral :: (Integral a, Num b) => a -> b` zum Umwandeln von `Int` nach `Double` benutzen.

Sie müssen sich *nicht* um zu große (außerhalb von `Int`) bzw. Zahlen mit zu großer Genauigkeit kümmern. Gehen Sie davon aus, dass die Eingabe keinen Überlauf/Abschneiden der Nachkommastellen produziert.

Aufgabe H11.2 HTML (10 Punkte)

In den Aufgaben G7.3 und H7.1 sollten Sie Funktionen schreiben, die zu einem Wert vom Typ

```
data Html = Text String | Block String [Html]
```

eine HTML-Darstellung generieren. In dieser Aufgabe wollen wir den umgekehrten Weg gehen und dazu die Parser-Kombinatoren aus der Vorlesung verwenden.

1. Schreiben Sie zunächst einen Tokenizer `htmlS :: Parser Char [HtmlT]`, der einen String in eine Liste der folgenden Tokens aufteilt:

```
data HtmlT = OpenT String    -- oeffnende Tags: <id>
           | CloseT String   -- schliessende Tags: </id>
           | WordT String    -- Ein Wort: string
```

`id` ist eine Zeichenkette, die nur aus Buchstaben, eventuell gefolgt von Ziffern, besteht (siehe `Parser.identifier`). `string` ist eine möglichst lange, nicht leere Zeichenkette, die weder Leerzeichen (`Data.Char.isSpace`) noch `'<'` oder `'>'` enthält. Tokens dürfen durch beliebig viele Leerzeichen getrennt sein.

Wenn der Tokenizer nicht mindestens ein Token erkennt, soll er `Nothing` zurückgeben.

Beispiele:

```
htmlS "<a> hello\n\nworld!</a>   " ==
  Just ([OpenT "a", WordT "hello", WordT "world!",
    CloseT "a"], "")
htmlS "<a>   <b>hi</c>" ==
  Just ([OpenT "a", OpenT "b", WordT "hi", CloseT "c"], "")
htmlS "</a32>>" == Just ([CloseT "a32"], ">")
htmlS "<a32/b>" == Nothing
```

2. Die Funktion `Parser.enclose` aus der Vorlesung benutzt die Konkatenation von Parsern, um einen mit den Zeichen `l` und `r` geklammerten Ausdruck mit dem Parser `p` zu parsen: `item l *** p *** item r`.

   Für das Parsen von HTML reicht `(***)` nicht aus, da das schließende Tag von dem öffnenden Tag abhängt: Beide müssen den gleichen Identifier haben.

   Schreiben Sie daher eine Funktion

   ```
   (***=) :: Parser a b -> (b -> Parser a c) -> Parser a (b,c)
   ```

   die `(***)` so verallgemeinert, dass der zweite Parser vom Ergebnis des ersten Parsers abhängen kann. Als Spezialfall soll dann `p1 *** p2 == p1 ***= \_ -> p2` für alle Parser `p1`, `p2` gelten.

3. Schreiben Sie einen Parser `htmlP :: Parser HtmlT Html`, der einer Liste von Tokens zu einem `Html`-Wert parst. Dabei sollen folgende Regeln gelten:

   - Eine nicht-leere Folge von `WortT`s wird zu `Text` s, wobei s aus allen Wörtern, getrennt durch Leerzeichen besteht.
   - `OpenT id : xs @ [CloseT id]` wird zu `Block id hs`, wenn `xs` als `hs :: [Html]` geparst wird.

Beispiele:

```
htmlP [WordT "hello", WordT "world"] ==
    Just (Text "hello world", [])
htmlP [OpenT "sv", WordT "hej", WordT "vaerld",
       CloseT "sv"] ==
    Just (Block "sv" [Text "hej vaerld"], [])
htmlP [OpenT "a", CloseT "a", Text "b"] ==
    Just (Block "a" [], [Text "b"])
htmlP [OpenT "a", Text "foo", Text "bar"] ==
    Nothing
htmlP [OpenT "a", OpenT "b", Text "c", CloseT "b",
       Text "d", CloseT "a"] ==
    Just (Block "a" [Block "b" [Text "c"], Text "d"], [])
```

Aufgabe H11.3 `http://xkcd.com/208/` (0 Punkte, Wettbewerbsaufgabe)

Reguläre Ausdrücke (kurz: *Regexes*) sind aus der Automatentheorie bekannt. Ähnlich wie die Wildcard-Muster aus Aufgabe G7.1 beschreiben sie Mengen von Strings. Unix-Programme wie `grep` und `sed` basieren auf Regexes, jedoch jeweils mit einer leicht abweichenden Syntax. Nach Donald Knuth: „I define UNIX as 30 definitions of regular expressions living under one roof.“

Diese Aufgabe führt die 31. Definition ein. Für beliebige Regexes r und s sei:

1. Das Zeichen `.` (Punkt) ist eine Regexp, die ein beliebiges Zeichen erkennt.
2. Ein Zeichen c außer `.\()[]{}*?+|` ist eine Regex, die c erkennt.
3. Die Regex `(`r`)` erkennt genau die Strings, die von r erkannt werden.
4. Die Regex rs (Verkettung) erkennt Strings, deren Beginn von r und deren Ende von s erkannt wird.
5. Die Regex r`|`s (Auswahl) erkennt Strings, die von r oder von s erkannt werden.
6. Ein Zeichenbereich `[`$s_1 \ldots s_n$`]` ($n \geq 0$) ist eine Regex, die ein beliebiges Zeichen aus einem beliebigen Teilbereich s_j erkennt. Ein *Teilbereich* ist entweder ein Zeichen $c \notin \{\backslash,]\}$ oder die drei Zeichen c`-`d (d.h. c, Bindestrich, d), wobei $c \notin \{\backslash,]\}$ und $d \neq \backslash$.
7. Die Regex r`*` erkennt null oder mehr Vorkommen von Strings, die jeweils von r erkannt werden.
8. Die Regex r`?` erkennt null oder ein Vorkommen eines Strings, der von r erkannt wird.
9. Die Regex r`+` erkennt ein oder mehr Vorkommen von Strings, die jeweils von r erkannt werden.
10. Die Regex $r\{m\}$ erkennt m Vorkommen von Strings, die jeweils von r erkannt werden.
11. Die Regex $r\{m,\}$ erkennt m oder mehr Vorkommen von Strings, die jeweils von r erkannt werden.

12. Die Regex $r\{m,n\}$ erkennt m bis n Vorkommen von Strings, die jeweils von r erkannt werden.
13. Überall, wo ein beliebiges Zeichen c stehen kann, muss auch die Maskierungssyntax $\backslash c$ (d.h. Backslash, c, wobei c auch ein Sonderzeichen wie \ sein kann) unterstützt werden. Das Backslash nimmt dem folgenden Zeichen seine spezielle Bedeutung, siehe Beispiele.

Beim Parsen haben die Wiederholungsoperatoren (`*`, `?` usw.) Vorrang vor dem Verkettungsoperator, der wiederum Vorrang vor dem Auswahloperator hat. Zum Beispiel ist die Regex `ab|cd*` als `(ab)|(c(d*))` zu verstehen. Die Verkettungs- und Auswahloperatoren sind linksassoziativ; d.h. `abc` = `(ab)c`.

Ihre Aufgabe besteht darin, eine Funktion `regEx :: Parser Char RegEx` zu implementieren, die Regexes parst. Der Typ `RegEx` ist in `RegEx.hs` definiert:

```
data RegEx =
  Any |
  One [(Char, Char)] |
  Repeat RegEx (Int, Maybe Int) |
  Concat RegEx RegEx |
  Alt RegEx RegEx
```

Beispiele:

```
regEx "" == Nothing
regEx "." == Just (Any, "")
regEx "a" == Just (One [('a', 'a')], "")
regEx "[a]" == Just (One [('a', 'a')], "")
regEx "[a-z_9-0]" ==
  Just (One [('a', 'z'), ('_', '_'), ('9', '0')], "")
regEx "abc" ==
  Just (Concat (Concat (One [('a', 'a')]) (One [('b', 'b')]))
               (One [('c', 'c')]), "")
regEx "a|b|c" ==
  Just (Alt (Alt (One [('a', 'a')]) (One [('b', 'b')]))
            (One [('c', 'c')]), "")
regEx "(ab)*" ==
  Just (Repeat (Concat (One [('a', 'a')]) (One [('b', 'b')]))
               (0, Nothing), "")
regEx "a)" == Just (One [('a', 'a')], ")")
regEx "a*?{3,5}" ==
  Just (Repeat (Repeat (Repeat (One [('a', 'a')]) (0, Nothing))
                       (0, Just 1))
               (3, Just 5), "")
regEx "\\(" == Just (One [('(', '(')], "")
regEx "[\\]-\\\\]" == Just (One [(']', '\\')], "")
```

Für den Wettbewerb zählt die Tokenanzahl (je kleiner, desto besser[1]), mit zwei Ausnahmen: `)` wird ignoriert und `=` zählt als -1. Um teilzunehmen müssen Sie die vollständige Lösung (außer `import`s) innerhalb der beiden Kommentare {`-WETT-`} und {`-TTEW-`} eingeben.

Wichtig: Wenn Sie diese Aufgabe als Wettbewerbsaufgabe abgeben, stimmen Sie zu, dass Ihr Name ggf. auf der Ergebnisliste auf unserer Internetseite veröffentlicht wird. Sie können diese Einwilligung jederzeit widerrufen, indem Sie eine Email an `fp@fp.in.tum.de` schicken. Wenn Sie nicht am Wettbewerb teilnehmen, sondern die Aufgabe allein im Rahmen der Hausaufgabe abgeben möchten, lassen Sie bitte die {`-WETT-`} ... {`-TTEW-`} Kommentare weg. Bei der Bewertung Ihrer Hausaufgabe entsteht Ihnen hierdurch kein Nachteil.

[1] `http://www21.in.tum.de/teaching/info2/WS1213/wettbewerb.html#token`