Praktische Informatik 3 WS 08/09

Christoph Lüth
Dominik Dietrich
Marcus Ermler
Klaus Hartke
Christian Maeder
Dominik Luecke
Ewaryst Schulz

3. Übungsblatt

Ausgabe: 01.12.08
Abgabe: 15.12.08

7 *Das sieht Ihnen ähnlich!* *5 Punkte*

In dieser Aufgabe soll eine Funktion

```
similar :: String-> String-> Bool
```

implementiert werden, die prüft, ob zwei Zeichenketten einander ähnlich sind. Dabei definieren wir die *Ähnlichkeit* zunächst auf der Ebene einzelner *Zeichen* wie folgt:

- Jedes Zeichen ist sich selbst ähnlich;
- Wenn zwei Zeichen einem dritten ähnlich sind, sind sie auch einander ähnlich;
- Die Zeichen `O` und `0` sind einander ähnlich;
- Die Zeichen `u` und `v` sind einander ähnlich;
- Die Zeichen `l`, `I` und `1` sind einander ähnlich.

Darauf aufbauend definieren wir die Ähnlichkeit von *Zeichenketten*:

- Aus einem einzelnen Zeichen bestehende Zeichenketten sind einander ähnlich, wenn die Zeichen einander ähnlich sind.
- Wenn die Zeichen c, d und e einander ähnlich sind, sind sowohl die Zeichenketten cd und e als auch die Zeichenketten c und de einander ähnlich.
- Zwei Zeichenketten sind einander ähnlich, wenn sie sich durch die Verkettung ähnlicher Zeichenketten ergeben, d.h. die Verkettung von $a_1, \ldots, a_n$ ist der Verkettung von $b_1, \ldots, b_n$ ähnlich, wenn a_i und b_i einander ähnlich sind, für alle $i = 1, \ldots, n$.

Formulieren Sie eine formale Spezifikation der Ähnlichkeit von Zeichenketten, aus der sie dann ausführbare Eigenschaften der Implementation herleiten können. Die Verwendung von `quickCheck` ist optional.

8 *Ausdrücklich* *15 Punkte*

Dieses ist die erste in einer losen Serie von Aufgaben, die sich mit der Entwicklung einer Tabellenkalkulation beschäftigen. Tabellenkalkulationen, auf neudeutsch auch *spreadsheets* genannt, sind die Grundlage der modernen Finanzmathematik und Buchhaltung. Ohne sie wäre die Finanzkrise der vergangenen Monate undenkbar gewesen, deshalb kann nur eine in Haskell entwickelte Tabellenkalkulation der Ausweg aus der Krise sein!

In dieser Aufgabe entwickeln wir einen Parser, der Formeln für unsere zu entwickelnde Tabellenkalkulation parsieren kann, d.h. aus einem String in eine interne Repräsentation (die sogenannte *abstrakte Syntax*) überführt.

Unsere Ausdrücke sind durch folgende Grammatik beschrieben:

```
Expr ::= Term '+' Expr
       | Term '-' Expr
       | 'if' BoolExpr 'then' Expr 'else' Expr
       | Term

Term ::= Factor '*' Term
       | Factor '/' Term
       | Factor

Factor ::= Number
         | Variable
         | '-' Factor
         | 'sum' Range
         | 'mult' Range
         | 'count' Range
         | '(' Expr ')'

Digits  ::= ('0'..'9')+
Variable ::= ('A'..'z')+ Digits
Number   ::= Digits '.' Digits
           | Digits

Range ::= '(' Variable ':' Variable ')'

BoolExpr ::= Conjuction '|' BoolExpr
           | Conjuction

Conjunction ::= Comparison '&' Conjunction
              | Comparison

Comparison  ::= '!' Comparison
              | '(' BoolExpr ')'
              | Expr '=' Expr
              | Expr '<' Expr
```

Hier sind einige Beispielausdrücke in der Sprache:

```
20+a2*3.234
sum(a1:a20)/count(a1:a20)
if b15 = b16 & zz94 < 23 then sum(b15:b16) else 0 + 3
```

Für die Entwicklung des Parsers benutzen wir eine Parserkombinatorbibliothek, deren Funktionsweise im Tutorium und Vorlesung näher erläutert wird.

Der Parser besteht aus mehreren Modulen:

- Dem *Lexer*, der den Eingabestring in eine Sequenz von *Token* übersetzt;
- Der *abstrakten Syntax*, einem Datentyp, der Ausdrücke in der Sprache repräsentiert;
- Dem eigentlichen *Parser*, welcher unter Benutzung des Lexers die Eingabe in die abstrakte Syntax übersetzt.

1. Für den Lexer definieren wir zuerst den Datentyp `Token`. Ein Token soll durch folgende Konstrukturen gegeben werden:

   - `TokNum`: eine Zahl, gegeben durch eine nicht-leere Sequenz von Ziffern, gefolgt ggf. von einem Dezimalpunkt und weiteren Ziffern, z.B. `37` oder `3.14159`;
   - `TokVar`: eine Variable, gegeben durch eine nicht-leere Sequenz von Buchstaben, gefolgt von einer Sequenz von Ziffern, z.B. `A13` oder `ab4`;
   - `TokSum, TokMult, TokCount, TokIf, TokThen, TokElse`: die Schlüsselwörter `sum`, `mult`, `count`, `if`, `then` und `else`;
   - `TokChar`: ein einzelnes Symbol (kein Buchstabe oder Zahl, z.B. '`?`', '`)`' oder '`-`').

   Der Lexer soll alle leeren Zeichen[1] aus der Eingabe filtern, und diese als eine Liste von Token zurückgeben:

   ```
   lexer :: String-> [Token]
   ```

   (*4 Punkte*)

2. Die abstrakte Syntax besteht im wesentlichen aus zwei Datentypen, die Ausdrücke repräsentieren, und zwar:

   - `BoolExpr` sind Ausdrücke mit einem boolschen Wert, gegeben durch die boolschen Operatoren (Konjunktionen, Disjunktionen, Negation), und Vergleiche (Gleich und Kleiner);
   - `Expr` sind Ausdrücke mit einem numerischen Wert, gegeben durch die arithmetischen Operatoren (Addition, Subtraktion, Multiplikation, Division), die Bereichsoperatoren (summieren, multiplizieren, zählen), Fallunterscheidung sowie Zahlen und Variablen.

   (*2 Punkte*)

3. Der Parser nutzt die beiden Module `Absy` und `Lexer`, um die Hauptfunktion

   ```
   parse :: String-> [Expr]
   ```

   zu implementieren, welche alle möglichen Parsierungen der Eingabe zurückgibt.

   (*8 Punkte*)

4. Eine Schwäche der oben angegeben Grammatik ist, dass sie die arithmetischen Operatoren linksassoziativ klammert, was bei Subtraktion und Division falsch ist, d.h. `3- 4- 5` wird als `3- (4- 5)` parsiert. Geben Sie eine Änderung der Grammatik an, welche diese Schwäche repariert. (*1 Punkt*)

Dies ist Revision 496 vom 2008-12-08.

[1] Leerzeichen, Tabulatoren, Zeilenvorschübe — siehe `isSpace` aus dem Modul `Char`.