Compiler
— Blatt 3 —

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik

Hochschule München

Sommersemester 2014

10.05.14 10:22

Aufgabe 1 — Reguläre Ausdrücke

Unter http://www.regexr.com/ finden Sie ein Online-Tool zum Ausprobieren von Regulären Ausdrücken.

Es nutzt die RegExp-Engine des Browsers und basiert auf dem JavaScript-RegExp-Standard. Es gibt verschiedene leicht voneinander abweichende RegExp-Syntaxen, z.B. Perl oder Java. Je nach Programmiersprache müssen Sie sich leicht umgewöhnen. Für die Klausur nutzen wir die unter obiger URL ausprobierbare JavaScript-Syntax.

Versuchen Sie zunächst die Beispiele für Reguläre Ausdrücke aus der Vorlesung unter http://www.regexr.com/ zum Laufen zu bringen.

Entwickeln Sie anschließend reguläre Ausdrücke für folgendes (schauen Sie sich dazu die Doku für die Ihnen noch nicht aus der Vorlesung bekannten Features an):

- Eine Zeile auf der genau ein Benutzername ohne Whitespace steht, der mindestens 3 und höchsten 16 Zeichen lang ist. Gültige Benutzernamen dürfen nur Buchstaben, Ziffern, den Unterstrich und das Minuszeichen enthalten
- Lassen Sie bei dem Benutzernamen zusätzlich davor und/oder dahinter Whitespace zu.
- Alle Zeichenketten für die gilt: Sie bestehen nur aus Kleinbuchstaben und der erste und der letzte Buchstabe sind gleich. Die Zeichenketten sollen maximale Länge haben.
  Beipiel: In `available` gibt es die beiden Zeichenketten `ava` und `availa` für die die

ersten beiden Bedingungen gelten. `availa` ist aber die längere Zeichenkette und soll folglich durch den Reguläre Ausdruck erkannt werden.

- Ändern Sie den Regulären Ausdruck so, dass die kürzere Zeichenkette erkannt wird.

Aufgabe 2 — Haskell

Gegeben sei folgendes Haskell-Programm:

```haskell
module Main where

import Data.Char (isDigit)

data Token = NatNum Integer | OpeningParen | ClosingParen | Add | Mult
    deriving (Show)

scan :: String -> Maybe [Token]
scan ""          = Just []
scan (' ':xs)    = scan xs
scan ('\t':xs)   = scan xs
scan ('(':xs)    = case scan xs of
    Nothing     -> Nothing
    Just tokens -> Just (OpeningParen : tokens)
scan (')':xs) = case scan xs of
    Nothing     -> Nothing
    Just tokens -> Just (ClosingParen : tokens)
scan ('+':xs) = case scan xs of
    Nothing     -> Nothing
    Just tokens -> Just (Add : tokens)
scan ('*':xs) = case scan xs of
    Nothing     -> Nothing
    Just tokens -> Just (Mult : tokens)
scan str@(x:xs)
    | isDigit x = let (digits, rest) = span isDigit str
                  in case scan rest of
                      Nothing -> Nothing
                      Just tokens -> Just (NatNum (read digits) : tokens)
    | otherwise = Nothing

main :: IO ()
main = do
    input <- getLine
```

```
case scan input of
    Nothing     -> putStrLn "error"
    Just tokens -> print tokens
main
```

- Analysieren Sie das Programm und versuchen Sie zu verstehen:
  1. was es macht und
  2. wie es das macht.
- Erzeugen Sie ein Haskell-Projekt und probieren Sie aus, was das Programm akzeptiert.
- Erweitern Sie das Programm so, dass es zusätzlich die Operatoren "-" und "/" akzeptiert.
- Überlegen Sie sich, wie Sie das Programm auf negative Zahlen erweitern könnten und wo es dabei Probleme geben könnte.

Anmerkungen:

- Nutzen Sie die Haskell-API-Dokumentation bzw. Hoogle um z.B. heraus zu finden was `span` macht.
- Statt jedes Mal `case scan xs of ...` zu schreiben, können Sie auch die praktische `maybe`-Funktion nutzen. Dadurch können Sie die `scan`- und die `main`-Funktion etwas kompakter schreiben:

```
scan :: String -> Maybe [Token]
scan ""          = Just []
scan (' ' :xs)  = scan xs
scan ('\t':xs)  = scan xs
scan ('(' :xs)  =
    maybe Nothing (\tokens -> Just (OpeningParen:tokens)) $ scan xs
scan (')' :xs)  =
    maybe Nothing (\tokens -> Just (ClosingParen:tokens)) $ scan xs
scan ('+' :xs)  =
    maybe Nothing (\tokens -> Just (Add         :tokens)) $ scan xs
scan ('*' :xs)  =
    maybe Nothing (\tokens -> Just (Mult        :tokens)) $ scan xs
scan str@(x:xs)
    | isDigit x =
        let (digits, rest) = span isDigit str
        in maybe Nothing
                 (\tokens -> Just (NatNum (read digits):tokens))
                 $ scan rest
    | otherwise = Nothing
```

```haskell
main :: IO ()
main = do
    input <- getLine
    maybe (putStrLn "error") print $ scan input
    main
```

Diesen Stand können Sie auch unter https://www.fpcomplete.com/user/obcode/compiler/blatt-03-aufgabe-3 clonen.