Compiler

Blatt 4

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 28.04.2017 21:02

Aufgabe 1 — Reguläre Ausdrücke

Ihr Repository bekommen Sie über den GitHub-Classroom-Link.

Das bereits fertige Programm (siehe `app/Main.hs`) nutzt die Haskell-Bibliothek regex-pcre um in Haskell mit Perl Compatible Regular Expressions (PCRE) arbeiten zu können.

Starten Sie das Programm nachdem Sie es mit `stack build` compiliert haben, mit dem Kommando

```
stack exec regexplayground
```

Anschließend können Sie Zeichenketten und reguläre Ausdrücke eingeben:

```
Geben Sie am string-Prompt eine Zeichenkette und am regex-Prompt
einen regex oder ":q" zum Beenden ein.
string> Hallo 1234 Welt!
regex> \d+
--------------------------------------------------------------------------------
match erfolgreich
--------------------------------------------------------------------------------
Matching: "Hallo 1234 Welt!" =~ "\\d+" :: MatchResult String
mrBefore: Hallo
mrMatch:  1234
mrAfter:   Welt!
```

```
-------------------------------------------------------------------------------
Matching: "Hallo 1234 Welt!" =~ "\\d+" :: AllTextMatches [] String
getAllTextMatches: ["1234"]
-------------------------------------------------------------------------------
string>
```

Schauen Sie sich den Haskell-Code in der Datei `app/Main.hs` an, um zu verstehen wie Sie reguläre Ausdrücke in Haskell nutzen können.

Gematcht wird in dem Code immer nur mit dem `=~`-Operator. Dieser lässt keine Modifier zu, um beispielsweise nicht case-sensitiv zu matchen. Natürlich ist auch so etwas in Haskell möglich, aber etwas umständlicher. Sollten Sie dies in einer zukünftigen Aufgabe benötigen, sprechen Sie mich einfach an.

Probieren Sie in der Anwendung aus wie reguläre Ausdrücken matchen.

Aufgabe 2 — Scanner für arithmetische Ausdrücke

In dieser Aufgabe soll ein Scanner für arithmetische Ausdrücke entwickelt werden.

Ihr Repository bekommen Sie über den GitHub-Classroom-Link.

Die `scan`-Funktion im Modul `Scanner` kann bisher nur `+`-Zeichen in das Token `Add` umwandeln. Erweitern Sie die Funktion so, dass außerdem die Zeichen `(`, `)`, `-`, `*` und `/` erkannt und in das entsprechende Token (siehe Modul `Types`) umgewandelt wird.

Außerdem sollen `Integer` mit Hilfe von einem Regex erkannt werden. Das repräsentierende Token `NatNum` enthält den `Integer` dann als Bestandteil, also

```
scan "1234" == Just [ NatNum 1234]
```

Nachdem Sie die Zeichenkette extrahiert haben, die den Integer repräsentiert, können Sie ihn mit der Funktion `read` in einen `Integer` umwandeln.