P. Thiemann, S. Wehr Wintersemester 2006/2007

**Compilerbau**

http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2006ws/

# Übungsblatt 3

15.11.2006

**Aufgabe 1** (Zeichenklassen in regulären Ausdrücken; 4 Punkte)
In Aufgabe 4 auf dem Übungsblatt 2 haben wir reguläre Ausdrücke unter anderem um Zeichenklassen erweitert. Der erweiterte reguläre Ausdruck $\texttt{[}s\texttt{]}$ passt auf alle Zeichen der Zeichenklasse $s$, während $\texttt{[\^{}}s\texttt{]}$ auf alle Zeichen passt, die nicht in $s$ enthalten sind. Wir haben auf diesem Übungsblatt auch gesehen, dass man eine Zeichenklasse in einen "normalen" regulären Ausdruck übersetzen kann. Allerdings ist der resultierende reguläre Ausdruck unter Umständen sehr groß und das Matching wird ineffizient.

Um diese Probleme zu umgehen kann man eine Alternativdefinition für reguläre Ausdrücke angeben, die direkt Zeichenklassen benutzt. Wir betrachten zunächst Zeichenklassen abstrakt als Teilmengen von $\Sigma$. Dann definieren wir die Menge $RE_s(\Sigma)$ der regulären Ausdrücke mit Zeichenklassen als kleinste Menge, die folgende Bedingungen erfüllt:

(i) $\{\underline{\emptyset}, \underline{\varepsilon}\} \subseteq RE_s(\Sigma)$.

(ii) Falls $s \in \mathcal{P}(\Sigma)$ dann $\texttt{[}s\texttt{]} \in RE_s(\Sigma)$.

(iii) Falls $\{p_1, p_2\} \subseteq RE_s(\Sigma)$ dann $\{p_1p_2, p_1|p_2, p_1^*\} \subseteq RE_s(\Sigma)$.

Man kann dann eine Übersetzung $[\![\cdot]\!] : RE_s(\Sigma) \to RE(\Sigma)$ von den um Zeichenklassen erweiterten regulären Ausdrücken in die "normalen" regulären Ausdrücke wie folgt angeben:

$$
\begin{aligned}
[\![\underline{\emptyset}]\!] &= \underline{\emptyset} \\
[\![\underline{\varepsilon}]\!] &= \underline{\varepsilon} \\
[\![\texttt{[}s\texttt{]}]\!] &= \begin{cases} \underline{\emptyset} & \text{falls } s = \emptyset \\ \underline{a}|[\![\texttt{[}s'\texttt{]}]\!] & \text{falls } s = \{a\} \cup s' \end{cases} \\
[\![p_1p_2]\!] &= [\![p_1]\!][\![p_2]\!] \\
[\![p_1|p_2]\!] &= [\![p_1]\!]|[\![p_2]\!] \\
[\![p^*]\!] &= [\![p]\!]^*
\end{aligned}
$$

(a) Definieren sie die Funktionen $L_s : RE_s(\Sigma) \to \mathcal{P}(\Sigma)$, $E_s : RE_s(\Sigma) \to RE(\Sigma)$ und $D_s : RE_s(\Sigma) \times \Sigma \to RE(\Sigma)$ analog zu den Funktionen $L : RE(\Sigma) \to \mathcal{P}(\Sigma)$, $E : RE(\Sigma) \to RE(\Sigma)$ und $D : RE(\Sigma) \times \Sigma \to RE(\Sigma)$ aus der Vorlesung.

(b) Zeigen sie, dass für alle $p \in RE_s(\Sigma)$ und $a \in \Sigma$ gilt:

- $L_s(p) = L([\![p]\!])$
- $E_s(p) = E([\![p]\!])$

- $D_s(p, a) = D(\llbracket p \rrbracket, a)$

In den letzten beiden Fällen ist die Gleichheit modulo den Vereinfachungsregeln für reguläre Ausdrücke zu interpretieren.

**Aufgabe 2** (Implementierung von Zeichenklassen; 6 Punkte)
Auf der Homepage der Vorlesung steht das auch im Skript besprochene Module `regexp.ml` zusammen mit der Interfacedefinition `regexp.mli` zur Verfügung. Ändern sie den Datentyp für reguläre Ausdrücke gemäß Aufgabe 1, passen sie die vorhandenen Funktionen an und fügen sie einen Datentyp sowie Konstruktorfunktionen für Zeichenklassen hinzu. Die Signatur des geänderten Moduls sollte wie folgt aussehen:

```
type 'a symclass
type 'a regexp

val symclass_range : 'a -> 'a -> 'a symclass
val symclass_union : 'a symclass -> 'a symclass -> 'a symclass
val symclass_symbol : 'a -> 'a symclass
val symclass_symbols : 'a list -> 'a symclass
val union_list : 'a symclass list -> 'a symclass

val epsilon : 'a regexp
val symclass_pos : 'a symclass -> 'a regexp  (* positive symbol class [s] *)
val symclass_neg : 'a symclass -> 'a regexp  (* negative symbol class [^s] *)
val symbol : 'a -> 'a regexp
val concat : 'a regexp -> 'a regexp -> 'a regexp
val alternate : 'a regexp -> 'a regexp -> 'a regexp
val repeat : 'a regexp -> 'a regexp

val repeat_one : 'a regexp -> 'a regexp
val concat_list : 'a regexp list -> 'a regexp
val alternate_list : 'a regexp list -> 'a regexp

val is_null : 'a regexp -> bool

val accepts_empty : 'a regexp -> bool
val after_symbol : 'a -> 'a regexp -> 'a regexp

val matches : 'a regexp -> 'a list -> bool
```

Testen sie ihre Implementierung, indem sie die regulären Ausdrücke für JavaScript Bezeichner (siehe Skript) mit Hilfe des geänderten Moduls definieren und auf geeignete Strings anwenden. Überlegen sie sich auch einen Testfall für negierte Zeichenklassen.

**Aufgabe 3** (Lexemdefinitionen für arithmetische Ausdrücke; 6 Punkte)
Auf dem ersten Übungsblatt wurde ein Datentyp und eine Evaluierungsfunktion für arithmetische Ausdrücke definiert. In dieser Aufgaben sollen nun die Lexeme und Token für arithmetische Ausdrücke definiert werden. Um die Sache etwas interessanter zu machen, sollen

nicht nur Integer–Literale in Dezimalnotation sondern auch Integer–Literale in Hexadezimalnotation sowie Gleitkomma–Literale unterstützt werden. Außerdem sollen Variablen und Variablenbindungen Teil der arithmetischen Ausdrücke sein. Als Operatoren stehen weiterhin Addition, Subtraktion, Multiplikation und Division zur Verfügung.

1. Spezifizieren sie die lexikalische Syntax der arithmetischen Ausdrücke (analog zur Spezifikation im Skript auf Seite 12).

2. Setzen sie ihre Spezifikation mit Hilfe von `ocamllex` um. Dabei muss die Einstiegsregel in den Lexer den Namen `token` tragen und vom Typ `Lexing.lexbuf -> token` sein. Der Typ `token` ist wie folgt zu definieren:

```
type token =
    PLUS
  | MINUS
  | STAR
  | SLASH
  | IDENT of string
  | INT of int
  | FLOAT of float
  | LET
  | BE
  | IN
  | LPAREN
  | RPAREN
  | EOF
```

---

**Abgabe:** 22.11.2006
Die Abgabe erfolgt bis zu Beginn der Übungsstunde. Einreichungen, die nicht den Abgabemodalitäten entsprechen, werden abgelehnt. Für Plagiate werden keine Punkte vergeben.

**Abgabemodalitäten:**

- Code muss per Email an die Adresse `wehr@informatik.uni-freiburg.de` geschickt werden, die Lösungen zu den restlichen Aufgaben sind auf Papier abzugeben.
- Der Code muss in einem Archiv mit dem Namen `vorname_nachname.tar.gz` oder `vorname_nachname.zip` an die Email angehängt werden.
- Entpacken des Archivs muss ein einzelnes Verzeichnis mit dem Namen `vorname_nachname` liefern.
- Innerhalb dieses Verzeichnisses müssen sich folgende Dateien befinden: `regexp_mod.mli`, `regexp_mod.mli` (für Aufgabe 2) sowie `arith.mll` (für Aufgabe 3).
- Alle Dateien müssen compilierbar sein; Teile einer Datei, welche nicht vom Compiler akzeptiert werden, müssen auskommentiert sein. Abgaben, die der Compiler nicht akzeptiert, werden nicht bewertet.

- Wenn eine Aufgabe von der Lösung ein gewisses Format verlangt (wie etwa einen festen Namen oder Typ für eine Funktion, eine vorgegebene Signatur für ein Modul etc.), so ist dieses Format zwingend einzuhalten.