PHILIPPS-UNIVERSITÄT MARBURG
Fachbereich Mathematik und Informatik
Prof. Dr. R. Loogen
J.Berthold

D-35032 Marburg
Hans Meerwein Straße

3.November 2005

2. Übung zu „Grundlagen des Compilerbaus“, WS 2005/06

Abgabe schriftlicher Aufgaben: Do, 10.November 2005 (vor der Vorlesung)
Besprechung mündlicher Aufg.: ab 7.November 2005 in der Übung

---

Mündliche Aufgaben

2.1 **Regulärer Ausdruck → Automat**
Gegeben sei der reguläre Ausdruck $\alpha = (a \mid b \mid c)(a \mid b \mid c)^* \in \mathrm{RA}(\{a, b, c\})$.

(a) Geben Sie einen möglichst einfachen DFA $\mathcal{A}$ zur Erkennung von L_α an.

(b) Bestimmen Sie *nach dem Satz von Kleene* einen NFA $\mathcal{A}_n$ zu α.

(c) Erzeugen Sie durch Potenzmengenkonstruktion und Minimierung einen zu $\mathcal{A}_n$ äquivalenten Minimalautomaten $\mathcal{A}_{min}$. Handelt es sich um $\mathcal{A}$?

2.2 **Durchlauf durch einen Backtrack-DFA**
Gegeben seien die regulären Ausdrücke

$$\begin{aligned} \alpha_1 &= ab \\ \alpha_2 &= ac \\ \alpha_3 &= (ab)^*c \end{aligned}$$

(a) Geben Sie einen DFA zu jedem Ausdruck an und konstruieren Sie den Produktautomaten für die lexikalische Analyse per Backtrack-DFA.

(b) Geben Sie die Zustandsfolgen des Backtrack-DFA während der Analyse des Eingabestrings `abababacca` an.

Schriftliche Aufgaben

2.3 **Erweiterte reguläre Ausdrücke** 5 Punkte
Reguläre Ausdrücke wurden in der Vorlesung allein durch paarweise Verknüpfung mit $\cdot$, $\mid$ und $*$ definiert. Für die praktische Anwendung wird diese Definition in der Regel um zusätzliche Konstrukte erweitert, etwa wie folgt:

- $[a_1 a_2 \ldots a_n] := (a_1 \mid a_2 \mid \ldots \mid a_n)$
  eines der Zeichen $a_1, \ldots, a_n \in \Sigma$.
- $\alpha^+ := \alpha\alpha^*$ für $\alpha \in \mathrm{RA}(\Sigma)$.
  evtl. mehrfach, *mindestens einmal* eine auf α passende Zeichenkette.
- $\alpha? := (\alpha \mid \Lambda^*)$ für $\alpha \in \mathrm{RA}(\Sigma)$.
  optional eine auf α passende Zeichenkette.

Bitte wenden!

(a) Erweitern Sie das Konstruktionsverfahren von Kleene um NFAs für diese Ausdrücke, so dass kleinere NFAs erzeugt werden.

(b) Bestimmen Sie nach dem Satz von Kleene und Ihrer Erweiterung NFAs zu den beiden (äquivalenten) regulären Ausdrücken

$$\alpha = (c \mid \Lambda^*)(a \mid b \mid c)(a \mid b \mid c)^* \qquad \text{und} \qquad \alpha' = c?[abc]^+$$

2.4 Haskell-NFA

7 Punkte

Gegeben seien die folgenden Grunddefinitionen zur Behandlung von (nichtdeterministischen) endlichen Automaten in Haskell:

Haskell Code

```
type Sigma       = [Char]
type DeltaNFA state = state -> Maybe Char -> [state]
data NFA state = NFA [state]        -- Liste der Zustaende
                     Sigma          -- Alphabet
                     (DeltaNFA state)  -- Transitionsfunktion
                     state          -- Anfangszustand
                     [state]        -- Menge der Endzustaende
```

Der `Maybe`-Typ modelliert dabei die ε-Übergänge der Transitionsfunktion.[1] Mengen von Zuständen werden als Listen modelliert.[2]

(a) Konstruieren Sie (zunächst auf Papier) einen NFA für vorzeichenlose Dezimalzahlen, die durch folgenden regulären Ausdruck definiert sind: / 1

$$(0 \mid (1-9)(0-9)^*).(0-9)^+$$

(b) Definieren Sie einen Haskell-NFA mit `Int`-Zuständen `decimalNFA :: NFA Int`, der Ihrem Automaten aus (a) entspricht. / 2

Benutzung des Haskell-NFA

(c) Schreiben Sie eine Funktion `epsCl :: Eq st => DeltaNFA st -> [st] -> [st]`, welche die ε-Hülle einer Menge von Zuständen bestimmt. / 2

(d) Schreiben Sie nun eine Funktion / 2

```
checkWord :: Eq st => (NFA st) -> String -> Bool
```

die zu einem gegebenen NFA und einem Wort überprüft, ob das Wort von dem Automaten erkannt wird.

Hinweis: Im Wesentlichen ist die Funktion $\overline{\delta}$ für NFAs zu programmieren. Hilfreich sind die Funktionen höherer Ordnung `map` und `foldl` und "list-comprehensions".

---

1) Der Haskell-Typ `Maybe a` zu einem beliebigen Typ `a` besitzt die Konstruktoren `Nothing` und `Just`, wobei `Just` noch ein Element enthält. Mit Pattern Matching lassen sich die Fälle unterscheiden:

```
f Nothing  = "Keine Eingabe"
f (Just x) = "Eingabe " ++ show x
```

2) Das Modul `List` enthält hilfreiche Funktionen zur Mengendarstellung über Listen, etwa:

`elem::`	`Eq a => a -> [a] -> Bool`	"ist Element von"
`concat::`	`[[a]] -> [a]`	konkateniert Listen zu einer einzigen Liste
`nub::`	`[a] -> [a]`	entfernt Duplikate aus einer Liste
`union,intersect::`	`Eq a => [a] -> [a] -> [a]`	Vereinigung und Schnitt von Listen

siehe auch: *http://www.haskell.org/onlinelibrary*