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

D-35032 Marburg
Hans Meerwein Straße
Lahnberge

03.November 2003

Übungen zur „Theorie des Compilerbaus“, WS 2003/04

Nr. 2 , Abgabe und Besprechung: 10. November 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) Bestimmen Sie mit Methode aus dem Satz von Kleene zu α einen NFA $\mathcal{A}$.

(b) Erzeugen Sie mit der Potenzmengenkonstruktion einen äquivalenten DFA $\mathcal{A}'$.

(c) Geben Sie zu $\mathcal{A}'$ einen äquivalenten minimalen Automaten $\mathcal{A}''$ an.

2 .2 **Erweiterung von regulären Ausdrücken**
Für reguläre Ausdrücke sind die folgenden Abkürzungen allgemein üblich:

- $[a_1, a_2, \ldots, a_n] := (a_1 \mid a_2 \mid \ \ldots \ \mid a_n)$ für $a_1, a_2, \ldots, a_n \in \Sigma$.
- $\alpha^+ := \alpha\alpha^*$ für $\alpha \in \mathrm{RA}(\Sigma)$.
- $\alpha? := (\alpha \mid \Lambda^*)$ für $\alpha \in \mathrm{RA}(\Sigma)$.

Das Konstruktionsverfahren aus dem Satz von Kleene erlaubt dagegen nur *paarweise* Verknüpfungen von regulären Ausdrücken, was zu recht komplexen NFAs führt.

Erweitern Sie das Verfahren derart, dass für diese Abkürzungen kleinere NFAs erzeugt werden. Geben Sie jeweils auch die NFAs an, die ohne Ihre Erweiterungen erzeugt würden.
Wenden Sie Ihre Erweiterungen auf $\alpha = [a, b, c]^+ \in \mathrm{RA}(\{a, b, c\})$ an.

Schriftliche Aufgaben

2 .3 **Nichtrekursive Klammerung** 4 Punkte
In Java können Kommentare der Form `/* Kommentar. */` angegeben werden, die aber nicht geschachtelt sein dürfen. Der Kommentartext darf beliebige Zeichen des Alphabets enthalten; der Abschnitt `*/` darf aber nicht vorkommen. Wir können diese Art Kommentare mit einem Konstrukt $R_1 \cdot \texttt{until}(R_2)$ beschreiben, wobei R_i beliebige reguläre Ausdrücke seien. $\texttt{until}(R_2)$ beschreibt Zeichenketten, in denen R_2 nur einmal am Ende vorkommt.

(a) Geben Sie eine formale Definition für `until` und beschreiben Sie o.g. Kommentare über einem Alphabet $\Sigma = \{/, *, a, b\}$ als regulären Ausdruck mit und ohne `until`. / 2

(b) Konstruieren Sie einen DFA $\mathcal{A}$ für die in a) beschriebene Sprache. / 2

2 .4 Haskell-NFA

8 Punkte

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

Haskell Code

```
type Sigma       = [Char] -- = String, hier aber nicht gemeint...

type DeltaNFA state = state -> Maybe Char -> [state]

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

Der `Maybe`-Typ modelliert dabei die ε-Übergänge.[1] Die Transitionsfunktion liefert zu einem Zustand und einer Eingabe (evtl. ε) die Liste aller möglichen Nachfolgezustände. Wodurch würde sich ein analog modellierter DFA unterscheiden?

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

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

definiert sind.

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

Benutzung des Haskell-NFA

(c) Schreiben Sie zunächst eine Funktion, welche die ε-Hülle eines Zustands bestimmt. / 2

(d) Schreiben Sie nun eine Funktion / 3

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

die zu einem gegebenen NFA und einem Wort überprüft, ob das Wort von dem Automaten erkannt wird. Sie müssen dazu im Wesentlichen die Funktion $\overline{\delta}$ für NFAs programmieren.[2]

Testen Sie Ihre Implementierung an dem Beispielautomaten.

---

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) Sie sollten sog. "List Comprehensions" einsetzen, wenn Sie die Übergänge modellieren. Denken Sie daran, dass `String=[Char]` ist. Außerdem sind die folgenden Funktionen aus Modul `List` hilfreich für die Bearbeitung dieser Aufgabe:

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

siehe auch: *http://haskell.cs.yale.edu/onlinelibrary*