

Übungsblatt 04

E-Learning

Absolvieren Sie die Tests bis Di., 14.05., 14 Uhr

Die Tests sind in der Stud.IP-Veranstaltung *Grundlagen der Praktischen Informatik (Informatik II)* unter *Lernmodule* hinterlegt.

Sie können einen Test **nur einmal durchlaufen**. Sobald Sie einen Test starten steht Ihnen nur eine **begrenzte Zeit** zu Verfügung, um den Test zu bearbeiten.

Alle Punkte, die Sie beim Test erreichen, werden ihnen angerechnet.

ILIAS – 13 Punkte

Formale Sprachen

Absolvieren Sie die folgenden Tests.

- GdPI 04 - Formale Sprachen - Repräsentationen von Automaten
- GdPI 04 - Formale Sprachen - Automat und Grammatik
- GdPI 04 - Formale Sprachen - Linkslinare Grammatik \rightarrow rechtslineare Grammatik

(13 Punkte)

Achtung

Zum ordnungsgemäßen Beenden eines Ilias-Test müssen Sie die Schaltfläche **Test beenden** betätigen.

Wenn Sie einen Ilias-Test einmal vollständig durchlaufen haben bekommen Sie auf die Seite *Testergebnisse*. Starten Sie den Test erneut aus Stud.IP, ist jetzt auch eine Schaltfläche *Testergebnisse anzeigen* vorhanden, die auf diese Seite führt.

Auf der Seite *Testergebnisse* können Sie sich unter *Übersicht der Testdurchläufe* zu jedem Testdurchlauf *Details anzeigen* lassen.

Falls eine **Musterlösung** vorhanden ist, führt der Titel einer Aufgabe in der Auflistung der Aufgaben zur Musterlösung.

Hinweis

- Eine häufige Fehlerquelle ist das Schließen des Browser-Fensters vor **Test beenden**.
- Wenn Sie einen JavaScript Blocker einsetzen, sollten Sie für Ilias eine Ausnahme hinterlegen.

Übung

Abgabe bis Di., 14.05., 14 Uhr

Allgemein

Die Aufgaben müssen in **Dreiergruppen** abgegeben werden. Vierergruppen sind ebenfalls möglich.

Es ist **wichtig**, dass Sie sich an folgendes **Verfahren für die Abgabe** halten.

Die Lösungen werden in geeigneter Form in der Stud.IP-Veranstaltung Ihrer Übungsgruppe über das Vips-Modul hochgeladen. Sie müssen diese Abgaben nicht mit Markdown+AsciiMath erstellen. Sie können Ihre Bearbeitungen auch mit \LaTeX formatieren, es ist aber auch die direkte Eingabe von Text oder der Upload von Text- und Bilddateien in gängigen Formaten möglich.

Weitere Hinweise zur Abgabe der Lösungen finden Sie in den Aufgabenstellungen.

Aufgabe 1 – 17 Punkte

Reguläre Sprachen

Sei L die Sprache aller Wörter $w \in \{0, 1, \dots, 9\}^*$ für die gilt, dass die Ziffernfolge w , interpretiert als ganze Zahl, ohne Rest durch 3 teilbar ist. Besteht die Ziffernfolge w aus mehr als einer Ziffer, muss die erste Ziffer ungleich 0 sein. Das leere Wort ist nicht in der Sprache enthalten.

Ist die Sprache L regulär?

(17 Punkte)

Hinweis

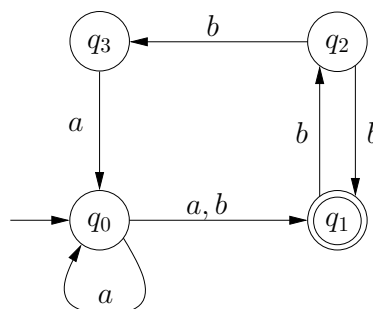
Ein Zahl ist ohne Rest durch 3 teilbar, wenn die Quersumme ohne Rest durch 3 teilbar ist.

Aufgabe 2 – 20 Punkte

Äquivalente Automaten

Geben Sie zu dem nachfolgend abgebildeten Zustandsgraphen eines nichtdeterministischen endlichen Automaten über dem Alphabet $\Sigma = \{a, b\}$ einen äquivalenten (vollständigen) deterministischen endlichen Automaten an.

(20 Punkte)



Hinweis.

Bei einem (vollständigen) deterministischen endlichen Automaten ist für jeden Zustand für alle Zeichen des Alphabets ein nachfolgender Zustand definiert. Die Übergänge in den Fehlerzustand können dabei aber ausgelassen werden. Sie können den Automaten entweder durch einen Zustandsgraphen darstellen oder über eine Tabelle mit der Übergangsfunktion. Wollen Sie einen Zustandsgraphen erstellen, können sie folgende Seite verwenden: Finite State Machine Designer. Dort können Sie sich ebenfalls direkt \LaTeX Code für den Automaten generieren lassen.

Aufgabe 3 – 25 Punkte

Operationen auf regulären Sprachen

Behauptung.

Sind L_1, L_2 reguläre Sprachen, dann ist auch der Durchschnitt der beiden Sprachen regulär.

$$L_1 \cap L_2 := \{w \mid w \in L_1 \text{ und } w \in L_2\}$$

Seien $A_1 = (\Sigma_1, Q_1, q_1, F_1, \delta_1)$ und $A_2 = (\Sigma_2, Q_2, q_2, F_2, \delta_2)$ deterministische endliche Automaten, für die gilt A_1 akzeptiert L_1 und A_2 akzeptiert L_2 .

Zeigen Sie die Behauptung, indem Sie aus A_1 und A_2 einen deterministischen endlichen Automaten konstruieren, der $L_1 \cap L_2$ akzeptiert.

(25 Punkte)

Praktische Übung 04+05

Abgabe der Prüfsumme bis Di., 21.05., 14 Uhr

Testat Di., 21.05. ab 18 Uhr.

Hilfe zum Bearbeiten der praktischen Übungen können Sie grundsätzlich jeden Tag in den Rechnerübungen bekommen. Die Testate finden ebenfalls in **Dreiergruppen** und Vierergruppen statt. Dabei sind die Gruppen identisch zu denen, die auch die theoretischen Aufgaben zusammen bearbeiten. In diesem Fall reserviert nur ein Gruppenmitglied einen Termin. Es ist ausreichend, wenn nur **eine** Person aus der Gruppe eine Prüfsumme abgibt.

Achtung: Die **Praktische Übung** ist im Gegensatz zur theoretischen Abgabe und Ilias für **zwei Wochen** angesetzt. Entsprechend ist auch die Abgabe der Prüfsumme erst in zwei Wochen.

Abgabe der Prüfsumme

- Siehe vorherige Übungen.
- Übermitteln Sie die Prüfsumme mit dem Test *GdPI 04+05 - Testat*.

Aufgabe 1 – 50 Punkte

Parser

Zur Fehlerbehandlung kann die Funktion `error` aus `Prelude` verwendet werden, die als Argument eine Zeichenkette erwartet. Der Aufruf von `error` stoppt die Abarbeitung und gibt die Zeichenkette aus.

Beispiel

```
-- calculate roots of  $x^2 + p x + q = 0$ 
parabolaRootA :: Double -> Double -> (Double, Double)
parabolaRootA p q
  | y < 0      = error "no roots"
  | otherwise = (-x + sqrt y, -x - sqrt y)
  where
    x = p/2.0
    y = x*x - q
```

Gibt es Nullstellen werde diese zurückgeliefert, ansonsten wird die Funktion abgebrochen und eine entsprechende Meldung ausgegeben.

```
> parabolaRootA 2.0 1.0
(1.0,1.0)
> parabolaRootA 1.0 2.0
no roots
```

Der Nachteil dieser Fehlerbehandlung ist das Abbrechen der Funktion.

Eine andere Möglichkeit zur Fehlerbehandlung ist die Benutzung des Typs `Maybe`.

```
data Maybe a = Just a | Nothing
    deriving (Eq, Ord)
```

Wird `Data.Maybe` importiert stehen u.a. die Funktionen

```
isNothing :: Maybe a -> Bool
fromJust  :: Maybe a -> a
```

zur Verfügung. Die Funktion `isNothing` liefert genau dann `True` zurück, wenn das Argument `Nothing` ist. Die Funktion `fromJust` extrahiert das ursprünglich Element aus dem `Just`, ist das Argument `Nothing` kommt es zu einem Fehler.

Beispiel

```
-- calculate roots of  $x^2 + p x + q = 0$ 
parabolaRootB :: Double -> Double -> Maybe (Double, Double)
parabolaRootB p q
    | y < 0      = Nothing
    | otherwise = Just (-x + sqrt y, -x - sqrt y)
    where
        x = p/2.0
        y = x*x - q
```

Gibt es Nullstellen werde diese in einem `Just` zurückgeliefert, ansonsten wird `Nothing` zurückgeliefert.

```
> import Data.Maybe
> parabolaRootB 2.0 1.0
Just(-1.0,-1.0)
> parabolaRootB 1.0 2.0
Nothing
> fromJust (parabolaRootB 2.0 1.0)
(-1.0,-1.0)
> isNothing (parabolaRootB 1.0 2.0)
True
```

Der Vorteil dieser Methode ist, dass mit dem Rückgabewert der Funktion weitergearbeitet werden kann.

1. Programmieren Sie, nach dem Beispiel der Vorlesung für den rekursiven Abstieg, einen LL(1)-Parser für die folgende Grammatik.

$G = \{N, T, P, S\}$

- $N = \{\text{id_list}, \text{id_list_tail}\}$
- $T = \{\text{"id"}, \text{","}, \text{";"}, \text{"\$\$"}\}$
- $S = \text{id_list}$
- $P = \left\{ \begin{array}{ll} \text{id_list} & \rightarrow \text{"id"} \text{id_list_tail} \\ \text{id_list_tail} & \rightarrow \text{","} \text{"id"} \text{id_list_tail} \\ \text{id_list_tail} & \rightarrow \text{";" " \$\$"} \end{array} \right\}$

Programmieren Sie mindestens folgende Funktionen, wobei das `[String]`-Argument die aktuell noch nicht abgearbeitete Eingabe repräsentiert.

```
match :: String -> [String] -> [String]
id_list_tail :: [String] -> [String]
id_list :: [String] -> [String]
```

Die Eingabe wird komplett abgearbeitet, wenn sie von der Grammatik erzeugt werden kann, d.h. die Funktion `id_list` liefert in diesem Fall eine leere Liste zurück. Behandeln Sie Fehler mit der Funktion `error`.

(20 Punkte)

Beispiel

```
> id_list ["id", ",", "id", ";", "$$"]
[]
> id_list ["id", "$$"]
error message
```

2. Programmieren Sie, nach dem Beispiel der Vorlesung für den rekursiven Abstieg, einen LL(1)-Parser für die folgende Grammatik.

$G = \{N, T, P, S\}$

- $N = \{\text{prog}, \text{expr}, \text{term}, \text{ttail}, \text{factor}, \text{ftail}\}$
- $T = \{\text{'+'}, \text{'*'}, \text{'c'}, \text{'\$'}\}$
- $S = \text{prog}$
- $P = \left\{ \begin{array}{ll} \text{prog} & \rightarrow \text{expr ' \$'} \\ \text{expr} & \rightarrow \text{term ttail} \\ \text{term} & \rightarrow \text{factor ftail} \\ \text{ttail} & \rightarrow \text{'+' term ttail} \mid \varepsilon \\ \text{factor} & \rightarrow \text{'c'} \\ \text{ftail} & \rightarrow \text{'*' factor ftail} \mid \varepsilon \end{array} \right\}$

Programmieren Sie für jedes Nichtterminal eine Funktion nach folgendem Vorbild, wobei das `Maybe String`-Argument die aktuell noch nicht abgearbeitete Eingabe repräsentiert, die auch `Nothing` (ein Fehler ist aufgetreten) sein kann.

```
match :: Char -> Maybe String -> Maybe String
...
ttailA :: Maybe String -> Maybe String
expr :: Maybe String -> Maybe String
prog :: String -> Maybe String
```

Die Eingabe wird komplett abgearbeitete, wenn sie von der Grammatik erzeugt werden kann, d.h. die Funktion `prog` liefert in diesem Fall einen leeren `String` in einem `Just` zurück, kommt es zu einem Fehler wird `Nothing` zurückgeliefert.

(30 Punkte)

Beispiel

```
> prog "c+c*c$"
Just ""
> prog "c+c-c$"
Nothing
```