Prof. Dr. Ralf Hinze
Cass Alexandru, M.Sc.
Alexander Dinges, M.Sc.
Sebastian Schloßer, M.Sc.

RPTU
Fachbereich Informatik
AG Programmiersprachen

Lösungshinweise/-vorschläge zum Übungsblatt 9: Grundlagen der Programmierung (WS 2024/25)

Ein- und Ausgabe Auf diesem Übungsblatt betrachten wir Ein- und Ausgabe (Kapitel 7, Effekte). Die folgenden Funktionen (aus dem Modul `Mini.fs`) können Sie verwenden:[1]

```
putstring: String -> Unit // Schreibt den gegebenen String auf die Konsole.
putline: String -> Unit   // Schreibt den gegebenen String gefolgt von einem
                          // Zeilenumbruch auf die Konsole.
putchar: Char -> Unit     // Schreibt das gegebene Zeichen auf die Konsole.
print: 'a -> Unit         // Schreibt einen beliebigen Wert (entsprechend formatiert)
                          // gefolgt von einem Zeilenumbruch auf die Konsole.
getchar: Unit -> Char     // Liest das nächste einzelne Zeichen von der Konsole.
getline: Unit -> String   // Liest die nächste komplette Zeile von der Konsole.
```

Die Funktionen, die etwas von der Konsole einlesen, warten so lange bis eine Eingabe verfügbar ist. Für `getchar` reicht schon ein einzelnes Zeichen in der Eingabe. Bei `getline` wird so lange gewartet, bis ein Zeilenumbruch (Enter-Taste) erzeugt wurde. Zurückgegeben wird der String ohne den Zeilenumbruch.

Weitere hilfreiche Funktionen sind:

```
readNat: String -> Nat    // Nimmt einen String und gibt den Wert als Zahl zurück.
show: 'a -> String        // Nimmt einen beliebigen Wert und gibt einen String zurück,
                          // der den Wert beschreibt, meist in F# Syntax (z.B. "5N").
string: 'a -> String      // Wandelt einen beliebigen Wert in einen String um.
```

In den Beispielen bei den Aufgaben ist die Ausgabe des Programms `blau` und die Eingabe `rot` markiert. Leerzeichen sind durch das Symbol ␣ dargestellt, Zeilenumbrüche durch ↩.

[1] In der abstrakten Syntax auf den Vorlesungsfolien und im Skript haben die Funktionen einen Bindestrich im Namen. Auch wenn diese Schreibweise möglicherweise schöner ist, ist ein Bindestrich in F# kein gültiges Zeichen für Bezeichner. Für die Übung brauchen wir Funktionsnamen, die in F# tatsächlich gültig sind, daher verzichten wir auf den Bindestrich.

Aufgabe 1 Warm Up (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie sich mit der Ein- und Ausgabe vertraut machen. Sie können sich an den Vorlesungsfolien 715 bis 751 sowie am Skript Kapitel 7.1 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Program.fs` *aus der Vorlage* `Aufgabe-9-1.zip`.

a) Machen Sie sich mit den oben vorgestellten Funktionen vertraut. Starten Sie den F# Interpreter und laden Sie das Modul `Mini`. Führen Sie dazu `dotnet fsi Mini.fs` aus.

Geben Sie nun die folgenden Ausdrücke jeweils gefolgt von `;;` ein:

- `putline("Hallo F#!")`
- `putstring("Hallo F#!")`
- `putchar('X')`
- `let tupel = (10N,20N) in print(tupel)`
- `let x = getline()`
- `let y = getchar()`
- `readNat "123N"`
- `readNat "123"`
- `show 5N`
- `string 5N`
- `show (10N, 'X')`
- `show [1N; 2N; 3N]`

Für diese Teilaufgabe ist keine Bearbeitung der Vorlage notwendig.

```
> putline("Hallo F#!");;
Hallo F#!
val it : unit = ()

> putstring("Hallo F#!");;
Hallo F#!val it : unit = ()

> putchar('X');;
Xval it : unit = ()

> let tupel = (10N,20N) in print(tupel);;
(10N, 20N)
val it : unit = ()

> let x = getline();;
Jetzt kann ich hier schreiben bis ich Enter drücke↩
val x : string = "Jetzt kann ich hier schreiben bis ich Enter drücke"

> let y = getchar();;
Zval y : char = 'Z'

> readNat "123N";;
val it : Nat = 123N

> readNat "123";;
val it : Nat = 123N

> show 5N;;
val it : string = "5N"

> string 5N;;
val it : string = "5"

> show (10N, 'X');;
val it : string = "(10N, 'X')"

> show [1N; 2N; 3N];;
val it : string = "[1N; 2N; 3N]"
```

b) Schreiben Sie eine Funktion `queryNat: String -> Nat`, welche als Argument einen String entgegennimmt, der auf die Konsole ausgegeben wird. Anschließend wird die Eingabe einer natürlichen Zahl erwartet (die Eingabe wird durch Drücken der Enter-Taste abgeschlossen).

Die eingegebene Zahl soll von der Funktion als Wert vom Typ `Nat` zurückgegeben werden. Falls die Eingabe keine gültige natürliche Zahl ist, soll das Programm die Fehlermeldung `"Eingabe ist keine natuerliche Zahl!"` ausgeben und die Eingabeaufforderung so lange wiederholen, bis eine gültige Eingabe vorliegt.

Beispielaufruf: `queryNat "Bitte geben Sie eine natuerliche Zahl ein: "`

```
Bitte␣geben␣Sie␣eine␣natuerliche␣Zahl␣ein:␣↩
Eingabe␣ist␣keine␣natuerliche␣Zahl!↩
Bitte␣geben␣Sie␣eine␣natuerliche␣Zahl␣ein:␣-1↩
Eingabe␣ist␣keine␣natuerliche␣Zahl!↩
Bitte␣geben␣Sie␣eine␣natuerliche␣Zahl␣ein:␣a↩
Eingabe␣ist␣keine␣natuerliche␣Zahl!↩
Bitte␣geben␣Sie␣eine␣natuerliche␣Zahl␣ein:␣0↩
```

```
let rec queryNat (msg: String): Nat =
    putstring msg
    let s = getline ()
    if s <> "" && String.forall Char.IsDigit s then
         readNat s
    else putline "Eingabe ist keine natuerliche Zahl!"
         queryNat msg
```

c) Schreiben Sie eine Funktion `main`, die mit Hilfe der Funktion `queryNat` drei natürliche Zahlen einliest und deren Minimum ausgibt. Sie können das Programm mit `dotnet run` ausführen.

Beispiel:

```
Bitte␣geben␣Sie␣drei␣natuerliche␣Zahlen␣ein.↩
Erste␣Zahl:␣a↩
Eingabe␣ist␣keine␣natuerliche␣Zahl!↩
Erste␣Zahl:␣815↩
Zweite␣Zahl:␣4711↩
Dritte␣Zahl:␣2021↩
Minimum:␣815↩
```

```
let main(): Unit =
    putline "Bitte geben Sie drei natuerliche Zahlen ein."
    let n1 = queryNat "Erste Zahl: "
    let n2 = queryNat "Zweite Zahl: "
    let n3 = queryNat "Dritte Zahl: "
    let min3 = min n1 (min n2 n3)
    putline ("Minimum: " + (string min3))
```

Aufgabe 2 Ein- und Ausgabe: Nim-Spiel (Einreichaufgabe, 20 Punkte)

Schreiben Sie Ihre Lösungen in die Datei `Nim.fs` *aus der Vorlage* `Aufgabe-9-2.zip`.

In dieser Aufgabe werden wir eine Variante des Nim-Spiels[2] implementieren. Von einem Haufen Streichhölzer, dessen Größe wir zu Beginn des Spiels festlegen, müssen zwei Spieler abwechselnd zwischen einem und drei Streichhölzern aufnehmen. Der Spieler, welcher das letzte Streichholz aufnimmt, verliert das Spiel.

In den folgenden Aufgabenteilen werden wir das Nim-Spiel Schritt für Schritt implementieren.

Das Ein- und Ausgabeverhalten des Spiels muss zwingend einer fest vorgegebenen Struktur folgen!
Beachten Sie die Hinweise auf der ersten Seite.

a) Schreiben Sie eine Funktion `queryNat: String -> Nat`, welche als Argument einen String entgegennimmt, der auf die Konsole ausgegeben wird. Anschließend wird die Eingabe einer natürlichen Zahl erwartet (die Eingabe wird durch Drücken der Enter-Taste abgeschlossen).

   Die eingegebene Zahl soll von der Funktion als Wert vom Typ `Nat` zurückgegeben werden. Falls die Eingabe keine gültige natürliche Zahl ist, soll das Programm die Fehlermeldung `"Eingabe ist keine natuerliche Zahl!"` ausgeben und die Eingabeaufforderung so lange wiederholen, bis eine gültige Eingabe vorliegt. Die gültige Eingabe einer Zahl wird durch eine Meldung der Form `"Die Zahl ... wurde eingegeben."` bestätigt.

   Beispielaufruf: `queryNat "Wie viele Streichhoelzer sollen es sein? "`

```
Wie␣viele␣Streichhoelzer␣sollen␣es␣sein?␣↩
Eingabe␣ist␣keine␣natuerliche␣Zahl!↩
Wie␣viele␣Streichhoelzer␣sollen␣es␣sein?␣-1↩
Eingabe␣ist␣keine␣natuerliche␣Zahl!↩
Wie␣viele␣Streichhoelzer␣sollen␣es␣sein?␣a↩
Eingabe␣ist␣keine␣natuerliche␣Zahl!↩
Wie␣viele␣Streichhoelzer␣sollen␣es␣sein?␣0↩
Die␣Zahl␣0␣wurde␣eingegeben.↩
```

```fsharp
let rec queryNat (msg: String): Nat =
    putstring msg
    let s = getline ()
    if s <> "" && String.forall Char.IsDigit s then
         putline ("Die Zahl " + s + " wurde eingegeben.")
         readNat s
    else putline "Eingabe ist keine natuerliche Zahl!"
         queryNat msg
```

[2] https://de.wikipedia.org/wiki/Nim-Spiel

b) Schreiben Sie eine Funktion `queryMove: Nat -> Nat -> Player -> Nat`, welche die Anzahl von Streichhölzern `n`, die Anzahl der maximal auswählbaren Streichhölzer `k` sowie den aktuellen Spieler `p` nimmt. Der Spieler soll aufgefordert werden seinen Zug einzugeben. Falls der Zug ungültig ist, also wenn weniger als ein bzw. mehr als `k` Streichhölzer gezogen werden, soll die Fehlermeldung `"Ungueltige Eingabe!"` ausgegeben werden. Ist die Anzahl der gezogenen Streichhölzer kleiner oder gleich `k`, aber größer der noch verfügbaren Menge an Streichhölzern `n`, wird der Zug dennoch als gültig betrachtet. Die Funktion soll schließlich die Zahl der gezogenen Streichhölzer zurückgeben.

Beispielaufruf: `queryMove 10N 3N A`

```
Es␣sind␣noch␣10␣Streichhoelzer␣uebrig.␣Spieler␣A␣ist␣am␣Zug:␣↩
Eingabe␣ist␣keine␣natuerliche␣Zahl!↩
Es␣sind␣noch␣10␣Streichhoelzer␣uebrig.␣Spieler␣A␣ist␣am␣Zug:␣a↩
Eingabe␣ist␣keine␣natuerliche␣Zahl!↩
Es␣sind␣noch␣10␣Streichhoelzer␣uebrig.␣Spieler␣A␣ist␣am␣Zug:␣0↩
Die␣Zahl␣0␣wurde␣eingegeben.↩
Ungueltige␣Eingabe!↩
Es␣sind␣noch␣10␣Streichhoelzer␣uebrig.␣Spieler␣A␣ist␣am␣Zug:␣4↩
Die␣Zahl␣4␣wurde␣eingegeben.↩
Ungueltige␣Eingabe!↩
Es␣sind␣noch␣10␣Streichhoelzer␣uebrig.␣Spieler␣A␣ist␣am␣Zug:␣3↩
Die␣Zahl␣3␣wurde␣eingegeben.↩
```

Beispielaufruf: `queryMove 2N 3N A`

```
Es␣sind␣noch␣2␣Streichhoelzer␣uebrig.␣Spieler␣A␣ist␣am␣Zug:␣3↩
Die␣Zahl␣3␣wurde␣eingegeben.↩
```

Hinweis: Konstruktoren des Typs `Player` können Sie mit Hilfe der Funktion `string: 'a -> String` in einen String umwandeln, z. B. `string A = "A"`.

```
let rec queryMove (n: Nat) (k: Nat) (p: Player): Nat =
    let msg = "Es sind noch " + (string n) + " Streichhoelzer uebrig. "
            + "Spieler " + (string p) + " ist am Zug: "
    let zug = queryNat msg
    if zug > 0N && zug <= k then
        zug
    else
        putline "Ungueltige Eingabe!"
        queryMove n k p
```

c) Schreiben Sie eine Funktion `nim: Nat -> Nat -> Player -> unit`, welche die Anzahl von Streichhölzern `n`, die Anzahl der pro Zug maximal auswählbaren Streichhölzer `k` sowie den aktuellen Spieler `p` nimmt. Die Funktion soll die Spieler abwechselnd ziehen lassen und schließlich den Gewinner ausgeben.

Beispielaufruf: `nim 10N 4N B`

```
Es␣sind␣noch␣10␣Streichhoelzer␣uebrig.␣Spieler␣B␣ist␣am␣Zug:␣3↩
Die␣Zahl␣3␣wurde␣eingegeben.↩
Es␣sind␣noch␣7␣Streichhoelzer␣uebrig.␣Spieler␣A␣ist␣am␣Zug:␣4↩
Die␣Zahl␣4␣wurde␣eingegeben.↩
Es␣sind␣noch␣3␣Streichhoelzer␣uebrig.␣Spieler␣B␣ist␣am␣Zug:␣2↩
Die␣Zahl␣2␣wurde␣eingegeben.↩
Es␣sind␣noch␣1␣Streichhoelzer␣uebrig.␣Spieler␣A␣ist␣am␣Zug:␣1↩
Die␣Zahl␣1␣wurde␣eingegeben.↩
Spieler␣B␣gewinnt␣das␣Spiel!↩
```

```
let nextPlayer (p: Player): Player =
    match p with
    | A -> B
    | B -> A
```

```
let rec nim (n: Nat) (k: Nat) (p: Player): Unit =
    let zug = queryMove n k p
    if n - zug < 1N then
        // der andere Spieler gewinnt
        putline ("Spieler " + (string (nextPlayer p)) + " gewinnt das Spiel!")
    else
        // der andere Spieler ist am Zug
        nim (n - zug) k (nextPlayer p)
```

d) Schreiben Sie eine Funktion `main`, die zunächst den String `"Willkommen zu Nim"` ausgibt, dann die Anzahl `n` der Streichhölzer abfragt und anschließend das `nim` Spiel mit der eingegebenen Menge an Streichhölzern und der oben festgelegten Regel `k=3N` startet. Spieler `A` darf zuerst ziehen.

Sie können das fertige Spiel mit dem Befehl `dotnet run` ausführen.

Beispiel:

```
Willkommen zu Nim↩
Wie viele Streichhoelzer sollen es sein? 10↩
Die Zahl 10 wurde eingegeben.↩
Es sind noch 10 Streichhoelzer uebrig. Spieler A ist am Zug: 3↩
Die Zahl 3 wurde eingegeben.↩
Es sind noch 7 Streichhoelzer uebrig. Spieler B ist am Zug: 2↩
Die Zahl 2 wurde eingegeben.↩
Es sind noch 5 Streichhoelzer uebrig. Spieler A ist am Zug: 1↩
Die Zahl 1 wurde eingegeben.↩
Es sind noch 4 Streichhoelzer uebrig. Spieler B ist am Zug: 2↩
Die Zahl 2 wurde eingegeben.↩
Es sind noch 2 Streichhoelzer uebrig. Spieler A ist am Zug: 3↩
Die Zahl 3 wurde eingegeben.↩
Spieler B gewinnt das Spiel!↩
```

```
let main(): Unit =
    putline "Willkommen zu Nim"
    let n = queryNat "Wie viele Streichhoelzer sollen es sein? "
    nim n 3N A
```

e) Achten Sie darauf, dass Ihr Programmcode möglichst lesbar ist und keine unnötig komplexen Ausdrücke enthält (vgl. Übungsblatt 3 Aufgabe 5). Dafür vergeben wir bei dieser Aufgabe 3 Punkte.

Siehe Lösungsvorschläge oben.

Aufgabe 3 Reguläre Ausdrücke automatisiert (Trainingsaufgabe)

Motivation: Anhand dieser freiwilligen Zusatzaufgabe können Sie nachvollziehen wie Akzeptoren für reguläre Ausdrücke automatisiert generiert werden können.

Schreiben Sie Ihre Lösungen in die Datei `Program.fs` *aus der Vorlage* `Aufgabe-9-3.zip`.

Harry Hacker erinnert sich, warum wir den seiner Ansicht nach komplizierten Weg über die Rechtsfaktoren gehen, anstatt uns passende Funktionen einfach so auszudenken: Das Argument für die Rechtsfaktoren ist, dass sie sich komplett automatisiert berechnen lassen. Dies möchte Harry Hacker nun einmal ausprobieren. Helfen Sie ihm, die dazu nötigen Funktionen zu implementieren. Folgenden Typ hat er schon definiert, um reguläre Ausdrücke in F# beschreiben zu können:

```
type Reg<'T> =
    | Eps                       // das leere Wort
    | Sym of 'T                 // einzelnes Zeichen / Terminalsymbol
    | Cat of Reg<'T> * Reg<'T>  // Konkatenation / Sequenz
    | Empty                     // die leere Sprache
    | Alt of Reg<'T> * Reg<'T>  // Alternative
    | Rep of Reg<'T>            // Wiederholung
```

Beispiel zur Beschreibung des regulären Ausdrucks `(ab)*` in diesem Typ:

```
type Alphabet = | A | B
let abstar: Reg<Alphabet> = Rep (Cat (Sym A, Sym B))
```

Tipp: Für die Teilaufgaben a und b müssen Sie lediglich die Definitionen aus den Vorlesungsfolien in gültigen F#-Code übertragen. Teil c ist etwas komplizierter, d und e sind wieder einfacher.

a) Schreiben Sie eine Funktion `nullable: Reg<'T> -> Bool`, die berechnet, ob der gegebene reguläre Ausdruck nullable ist, d.h. ob er das leere Wort ϵ akzeptiert.

   Beispiele:

   ```
   nullable abstar = true // abstar aus der Definition oben
   nullable Eps = true
   nullable (Sym A) = false
   ```

```
let rec nullable<'T> (r: Reg<'T>): Bool =
    match r with
    | Sym _ -> false
    | Eps -> true
    | Cat (r1, r2) -> nullable r1 && nullable r2
    | Empty -> false
    | Alt (r1, r2) -> nullable r1 || nullable r2
    | Rep _ -> true
```

b) Schreiben Sie eine Funktion `divide: 'T -> Reg<'T> -> Reg<'T>` die ein Zeichen `x` aus dem Alphabet sowie einen regulären Ausdruck `r` nimmt und den Rechtsfaktor `x\r` berechnet.

Beispiele:

```
divide A (Sym A) = Eps
divide B (Sym A) = Empty
divide A (Cat (Sym A, Sym B)) = Alt (Cat (Eps, Sym B), Cat (Empty, Empty))
```

Das Resultat im letzten Beispiel lässt sich vereinfachen zu `Sym B`. Sie brauchen keine Vereinfachungen einzubauen, in `Helpers.fs` steht eine Funktion `simplify: Reg<'T> -> Reg<'T>` bereit, die derartige Vereinfachungen durchführt. Damit ist dann `simplify (divide A abstar) = Cat (Sym B, abstar)`.

```
let rec divide<'T when 'T: comparison> (x: 'T) (r: Reg<'T>): Reg<'T> =
    match r with
    | Sym a -> if a = x then Eps else Empty
    | Eps -> Empty
    | Cat (r1, r2) when nullable r1 ->
        Alt (
            Cat (divide x r1, r2),
            divide x r2
        )
    | Cat (r1, r2) -> Cat (divide x r1, r2)
    | Empty -> Empty
    | Alt (r1, r2) -> Alt (divide x r1, divide x r2)
    | Rep r -> Cat (divide x r, Rep r)
```

c) Nun wollen wir nicht nur einen Rechtsfaktor berechnen, sondern alle. Also auch die Rechtsfaktoren der Rechtsfaktoren usw. Wir nutzen dazu folgenden Datentyp:

```
type Automaton<'T when 'T: comparison> = Map<Reg<'T>, Map<'T, Reg<'T>> * Bool>
```

Wir betrachten also eine Map (endliche Abbildung), deren Schlüssel reguläre Ausdrücke sind. Als Werte in dieser Map sind Paare gespeichert. Die zweite Komponente des Paars ist ein boolescher Wert, der angibt, ob der reguläre Ausdruck nullable ist. Die erste Komponente des Paars ist eine weitere Map, die wiederum Zeichen des Eingabealphabets auf reguläre Ausdrücke abbildet.

Wenn der reguläre Ausdruck `r` auf das Paar `(m, false)` abgebildet wird und `m` das Zeichen `x` auf den regulären Ausdruck `r'` abbildet, dann bedeutet das, dass `x\r = r'` ist und dass `r` nicht nullable ist.

Das beschriebene Konstrukt ist ein endlicher Automat: Jeder reguläre Ausdruck ist ein Zustand des Automaten. Die `Map<'T, Reg<'T>>` beschreibt die Transitionen vom Zustand des regulären Ausdrucks ausgehend. Der boolesche Wert (zweite Komponente des Paars) gibt an, ob es sich beim jeweiligen Zustand um einen akzeptierenden Zustand handelt. Daher haben wir diesen Datentyp `Automaton` genannt.

Machen Sie sich mit dem `Map` Modul aus der Standardbibliothek[3] vertraut, insbesondere mit `Map.empty`, `Map.add`, `Map.find` und `Map.containsKey`.

Schreiben Sie eine Funktion `calculateAutomaton: Reg<'T> -> Automaton<'T>`, die für einen gegebenen regulären Ausdruck einen solchen Automaten berechnet. Gehen Sie dabei wie folgt vor:

1. Definieren Sie sich eine rekursive Hilfsfunktion, die als Eingabe einen `Automaton<'T>` sowie einen regulären Ausdruck `r` vom Typ `Reg<'T>` erhält und einen aktualisierten `Automaton<'T>` zurückgibt.
2. Die Hilfsfunktion überprüft, ob `r` bereits im Automaten enthalten ist, also ob dieser Schlüssel in der Map existiert. Ist dies der Fall, dann wird der Automat unverändert zurückgegeben.
3. Andernfalls wird der gegebene Automat aktualisiert, indem zum regulären Ausdruck `r` zunächst das Paar `(Map.empty, nullable r)` hinterlegt wird. Dies ist notwendig, damit rekursive Aufrufe in die Abbruchbedingung aus dem vorherigen Schritt gelangen.

[3] https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-mapmodule.html

4. Mit `cases<'T>()` erhalten Sie eine Liste vom Typ `List<'T>`, die alle Symbole des Eingabealphabets enthält. Beispielsweise ist `cases<Alphabet>() = [A; B]` (für den im Beispiel oben definierten Typ `Alphabet`). Für jedes dieser Symbole `x` berechnen wir den Rechtsfaktor `r' = x\r`. Nutzen Sie die Funktion `simplify` um `r'` zu vereinfachen.
   Rufen Sie nun die Hilfsfunktion rekursiv auf, um `r'` und alle seine Rechtsfaktoren in den Automaten einzutragen. Anschließend tragen Sie in den Automaten ein, dass der Rechtsfaktor `x\r = r'` ist. Dazu müssen Sie zunächst die innere Map für die Transitionen von `r` aktualisieren und die aktualisierte Map anschließend in die äußere Map eintragen. Achten Sie darauf, die zweite Komponente des Paars (also ob `r` nullable ist) nicht zu verändern.
   Tipp: Da Sie den Automaten schrittweise für jedes Symbol aus dem Alphabet aktualisieren müssen, bietet sich die Verwendung von `List.fold` an.
5. Zum Schluss muss die Haupt-Funktion die Hilfsfunktion mit einem leeren Automaten (`Map.empty`) und dem gegebenen regulären Ausdruck aufrufen.

```
let calculateAutomaton<'T when 'T: comparison> (r: Reg<'T>): Automaton<'T> =
    let rec insert (automaton: Automaton<'T>) (r: Reg<'T>): Automaton<'T> =
        if Map.containsKey r automaton then automaton
        else
            let automaton = automaton |> Map.add r (Map.empty, nullable r)
            cases<'T>() |> List.fold (
                fun automaton x ->
                    let r' = divide x r |> simplify
                    let automaton = insert automaton r'
                    let (transitions, isFinite) = automaton |> Map.find r
                    let transitions = transitions |> Map.add x r'
                    automaton |> Map.add r (transitions, isFinite)
            ) automaton
    insert Map.empty r
```

d) Wir definieren nun `type Alphabet = | Zero | One | Dot`. Definieren Sie einen Wert `floatRegex` vom Typ `Reg<Alphabet>`, um den folgenden regulären Ausdruck für Fließkommazahlen zu beschreiben:
`((0|1(0|1)*).(0|1)*) | (.(0|1)(0|1)*)`

```
type Alphabet = | Zero | One | Dot

let floatRegex: Reg<Alphabet> =
    Alt (
        Cat (
            Alt (
                Sym Zero,
                Cat (
                    Sym One,
                    Rep (Alt (Sym Zero, Sym One))
                )
            ),
            Cat (
                Sym Dot,
                Rep (Alt (Sym Zero, Sym One))
            )
        ),
        Cat (
            Sym Dot,
            Cat (
                Alt (Sym Zero, Sym One),
                Rep (Alt (Sym Zero, Sym One))
            )
        )
    )
```

e) Starten Sie das Programm mit `dotnet run`. Dabei wird der reguläre Ausdruck `mainRegex` betrachtet. Sie können `let mainRegex = floatRegex` definieren, um den Ausdruck aus der vorherigen Teilaufgabe zu benutzen, oder Sie definieren einen weiteren regulären Ausdruck. In der Ausgabe finden Sie eine Beschreibung des Aufrufgraphen, die Sie mit Graphviz[4] verarbeiten können sowie F# Code für die Akzeptorfunktion.[5]

Sie können sich selbst weitere reguläre Ausdrücke ausdenken und die Rechtsfaktoren zur Übung von Hand berechnen. Anschließend lassen Sie sich mit dem Programm aus dieser Aufgabe den Graphen generieren und kontrollieren so Ihre händisch erstellte Lösung.

> Die Struktur des Programms ist gleich, jedoch kann die Benennung der einzelnen Funktionen und die Reihenfolge, in der sie definiert sind, abweichen.

[4] Den Code können Sie einfach bei `http://www.webgraphviz.com/` einfügen, wenn Graphviz bei Ihnen nicht installiert ist.

[5] Die Datei `Main.fs` enthält Funktionen, die den `Automaton` in die textuelle Beschreibung für Graphviz und in gültigen F# Programmcode (als String) umwandeln.

Aufgabe 4 Kreuzworträtsel mit regulären Ausdrücken (Trainingsaufgabe)

Motivation: In dieser Aufgabe sollen Sie üben, aus gegebenen regulären Ausdrücken ein Wort abzuleiten. Sie können sich an den Vorlesungsfolien 552 bis 622 sowie am Skript Kapitel 6.1 und 6.2 orientieren.

Lösen Sie die folgenden Kreuzworträtsel. In jedes Kästchen muss ein Element des Alphabets eingetragen werden, sodass die zeilen- und spaltenweise gelesenen Wörter durch die gegebenen regulären Ausdrücke beschrieben werden.

a) Alphabet $A := \{0, 1\}$

	$(0 \mid 1)^*$	1^*0^*
01^*	0	1
100^*	1	0

b) Alphabet $A := \{A, B, C, D\}$

	$(C \mid \epsilon)DD^*$	$C^* \mid (BA)^* \mid (A \mid D)^*$
$(C \mid A)(B \mid D)$	C	B
$DA \mid B(A^*)$	D	A

c) Alphabet $A := \{A, I, L, S, T, U\}$

	$(UA \mid LA)A^*S$	$(A \mid I \mid L \mid T)^*(LT \mid LU)$	$(S \mid \epsilon)(\epsilon \mid T)I(A \mid I)$
$(L \mid A)I(A \mid S)^*$	L	I	S
$(A \mid \epsilon)L^*I^*$	A	L	I
$(IS \mid S)T(A^*)$	S	T	A