

Lösungshinweise/-vorschläge zum Übungsblatt 9: Konzepte der Programmierung (WS 2025/26)

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 `↵`.

¹In 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:

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

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:↵2023↵
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: Würfel-Black-Jack-Spiel (Einreichaufgabe, 26 Punkte)

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

In dieser Aufgabe werden wir eine einfache Variante des Black Jack-Spiels² implementieren. Eine Anzahl an Spielern würfelt reihum. Jeder Spieler addiert die Werte, die er würfelt auf. Ziel ist es, so nah wie möglich an den Wert 21 zu kommen, ohne diesen jedoch zu überschreiten. Im Fall des Überschreitens verliert der Spieler automatisch. Jeder Spieler hat die Möglichkeit, auszusteigen, wobei er seine Punktzahl behält.

In den folgenden Aufgabenteilen werden wir das Würfel-Black-Jack-Spiel Schritt für Schritt implementieren.

Wir modellieren einen Spieler durch einen `String`. Den Zustand des Spielers, also den aktuellen Spielstand und ob er noch teilnimmt, speichern wir in

```
type PlayerState = { score : Nat ; active : Bool }
```

Den Gesamtspielstand speichern wir in einer `MapUnsortedList<String, PlayerState>`. Von Aufgabe 2 auf Übungsblatt 7 kennen Sie `MapSortedList<'k, 'v>`. Der Datentyp `MapUnsortedList<'k, 'v>` funktioniert ganz analog. Die Funktionen `insert<'k, 'v>: 'k*'v -> MapUnsortedList<'k, 'v> -> MapUnsortedList<'k, 'v>` und `tryFind<'k, 'v>: 'k -> MapUnsortedList<'k, 'v> -> Option<'v>` stehen Ihnen zur Verfügung. Wir verwenden die Abkürzung

```
type Players = MapUnsortedList<String, PlayerState>
```

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 `queryBool: String -> Bool`, welche als Argument einen `String` entgegennimmt, der auf die Konsole ausgegeben wird. Anschließend wird die Eingabe von `"ja"` oder `"nein"` erwartet (die Eingabe wird durch Drücken der Enter-Taste abgeschlossen).

Wird `"ja"` eingegeben, soll `true` und bei `"nein"` soll `false` zurückgegeben werden. Falls die Eingabe weder `"ja"` noch `"nein"` ist, soll das Programm die Fehlermeldung `"Eingabe ist weder ja noch nein!"` ausgeben und die Eingabeaufforderung so lange wiederholt werden, bis eine gültige Eingabe vorliegt.

Beispielaufruf: `queryBool "Spieler A ist an der Reihe. Moechte Spieler A wuerfeln? (ja/nein)"`

```
Spieler_A_ist_an_der_Reihe._Moechte_Spieler_A_wuerfeln?_(ja/nein)↵
Eingabe_ist_weder_ja_noch_nein!↵
Spieler_A_ist_an_der_Reihe._Moechte_Spieler_A_wuerfeln?_(ja/nein)j↵
Eingabe_ist_weder_ja_noch_nein!↵
Spieler_A_ist_an_der_Reihe._Moechte_Spieler_A_wuerfeln?_(ja/nein)yes↵
Eingabe_ist_weder_ja_noch_nein!↵
Spieler_A_ist_an_der_Reihe._Moechte_Spieler_A_wuerfeln?_(ja/nein)ja↵
```

```
let rec queryBool (msg: String): Bool =
  putstring msg
  let s = getline ()
  if s = "ja" then
    true
  elif s = "nein" then
    false
  else
    putline "Eingabe ist weder ja noch nein!"
    queryBool msg
```

²https://de.wikipedia.org/wiki/Black_Jack

- b) Schreiben Sie eine Funktion `queryMove: (Unit -> Nat) -> String -> Players -> Players`, welche einen „Würfel“, den wir durch eine Funktion mit dem Typen `Unit -> Nat` modellieren, einen Spieler sowie den Gesamtspielstand nimmt. Der Spieler soll mithilfe von `queryBool` gefragt werden, ob er würfeln möchte. Falls er dies bejaht, soll der Würfel „geworfen“ werden und die Augenzahl dem Spieler zugeschrieben werden. Außerdem soll die gewürfelte Zahl ausgegeben werden. Der Zustand des Spielers soll aktualisiert werden, d.h. die Augenzahl soll auf seinen Punktestand addiert werden. Ist der neue Punktestand zudem größer als 21, wird `active` auf `false` gesetzt. Falls der Spieler die Frage verneint, soll `active` des Spielers auf `false` gesetzt werden. In beiden Fällen wird der aktualisierte Gesamtstand zurückgegeben.

Im Fall, dass der Spieler gar nicht existiert, soll der Spielstand unverändert zurückgegeben werden. Sie müssen außerdem nicht vorher überprüfen, ob der Spieler noch aktiv ist.

Beispielaufruf:

```
let spielstand = [("A", { score = 10N; active = true }); ("B", { score = 15N; active = true })]
queryMove gdpWuerfel "A" spielstand
```

```
Spieler_A_ist_an_der_Reihe_Moechte_A_wuerfeln?_ja↵
Eingabe_ist_weder_ja_noch_nein!↵
Spieler_A_ist_an_der_Reihe_Moechte_A_wuerfeln?_ja↵
Wuerfeln... Der Wuerfel zeigt 4↵
```

Im Beispiel soll `[("A", { score = 14N; active = true }); ("B", { score = 15N; active = true })]` zurückgegeben werden.

```
let queryMove (dice: Unit -> Nat) (player: String) (players: Players): Players =
  let cont = queryBool ("Spieler " + player + " ist an der Reihe. "
    + "Moechte " + player + " wuerfeln? (ja/nein) ")
  let maybeOldState = tryFind player players
  match maybeOldState with
  | None -> players
  | Some oldState ->
    if cont
    then
      let diceValue = dice ()
      putline ("Wuerfeln... Der Wuerfel zeigt "
        + (string diceValue))
      let newScore = oldState.score + diceValue
      let newState = { score = newScore
        ; active = oldState.active && newScore <= 21N}
      insert (player, newState) players
    else
      insert (player, {oldState with active = false }) players
```

- c) Schreiben Sie eine Funktion `scoreOverview: Players -> Unit`, welche den gegebenen Gesamtspielstand ausgibt.

Beispielaufruf: `scoreOverview spielstand`

```
Aktueller Spielstand:↵
A: 10 Punkte↵
B: 15 Punkte↵
```

```
let scoreOverview (players: Players): Unit =
  putline "Aktueller Spielstand:"
  let rec h (players: Players): Unit =
    match players with
    | [] -> ()
    | (name, score)::restPlayers ->
      putline (name + ": " + (string score.score) + " Punkte")
      h restPlayers
  h players
```

- d) Schreiben Sie eine Funktion `evaluateScore: Players -> Unit`, welche den gegebenen Gesamtspielstand nimmt und ausgibt, wer gewonnen hat. Sollte niemand gewonnen haben (weil alle Spieler mehr als 21 Punkte haben), soll das ausgegeben werden.

Sie können ignorieren, ob der Spieler aktiv ist oder nicht. Bei Gleichstand können Sie willkürlich einen Sieger ermitteln.

Beispielaufruf: `evaluateScore spielstand`

`Spieler_B_hat_mit_15_Punkten_gewonnen.↵`

Beispielaufruf: `evaluateScore [("A", { score = 22N; active = false })]`

`Kein_Spieler_hat_gewonnen.↵`

```
let evaluateScore (players: Players): Unit =
  let candidates = players
    |> List.filter (fun (_, s) -> s.score <= 21N)
  match candidates with
  | [] -> putline "Kein Spieler hat gewonnen."
  | _ -> let (name, state) = List.maxBy (fun (_, s) -> s.score) candidates
         putline ("Spieler " + name + " hat mit "
                  + (string state.score) + " Punkten gewonnen.")
```

- e) Schreiben Sie eine Funktion `blackjack: (Unit -> Nat) -> String -> Players -> Unit`, welche einen „Würfel“, einen Spielernamen und den Gesamtspielstand nimmt. Die Funktion soll den Spielstand ausgeben und dann den Spieler ziehen lassen. Anschließend soll geprüft werden, ob das Spiel fertig ist. In diesem Fall geben Sie aus, wer gewonnen hat. Andernfalls soll das Spiel mit dem nächsten Spieler fortgeführt werden.

Sie dürfen davon ausgehen, dass der Spielername tatsächlich in der Liste existiert.

Hinweis: Verwenden Sie die vorgefertigte Funktion `nextPlayer: String -> Players -> Option<String>`. Diese nimmt den aktuellen Spielernamen und den aktuellen Spielstand und gibt aus, wer der nächste Spieler ist. Sollte es keinen nächsten Spieler geben, wird `None` zurückgegeben. In diesem Fall ist das Spiel beendet.

Beispielaufruf: `blackjack gdpWuerfel "A" spielstand`

```
Aktueller_Spielstand:↵
A:_10_Punkte↵
B:_15_Punkte↵
Spieler_A_ist_an_der_Reihe._Moechte_A_wuerfeln?_(ja/nein)_ja↵
Wuerfeln..._Der_Wuerfel_zeigt_3↵
Aktueller_Spielstand:↵
A:_13_Punkte↵
B:_15_Punkte↵
Spieler_B_ist_an_der_Reihe._Moechte_B_wuerfeln?_(ja/nein)_ja↵
Wuerfeln..._Der_Wuerfel_zeigt_3↵
Aktueller_Spielstand:↵
A:_13_Punkte↵
B:_18_Punkte↵
Spieler_A_ist_an_der_Reihe._Moechte_A_wuerfeln?_(ja/nein)_ja↵
Wuerfeln..._Der_Wuerfel_zeigt_2↵
Aktueller_Spielstand:↵
A:_15_Punkte↵
B:_18_Punkte↵
Spieler_B_ist_an_der_Reihe._Moechte_B_wuerfeln?_(ja/nein)_ja↵
Wuerfeln..._Der_Wuerfel_zeigt_4↵
Aktueller_Spielstand:↵
A:_15_Punkte↵
B:_22_Punkte↵
```

```

Spieler_A_ist_an_der_Reihe._Moechte_A_wuerfeln?(ja/nein)_ja↵
Wuerfeln..._Der_Wuerfel_zeigt_4↵
Aktueller_Spielstand:↵
A:_19_Punkte↵
B:_22_Punkte↵
Spieler_A_ist_an_der_Reihe._Moechte_A_wuerfeln?(ja/nein)_nein↵
Spieler_A_hat_mit_19_Punkten_gewonnen.↵

```

```

let rec blackjack (dice: Unit -> Nat) (player: String) (players: Players): Unit =
  scoreOverview players
  let newPlayers = queryMove dice player players
  let next = nextPlayer player newPlayers
  match next with
  | None -> evaluateScore newPlayers
  | Some nextName -> blackjack dice nextName newPlayers

```

- f) Schreiben Sie eine Funktion `main`, die zunächst den String `"Willkommen zu Wuerfel-Black-Jack!"` ausgibt und anschließend das `blackjack` Spiel mit der vorgefertigten Funktion `gdpWuerfel` als Würfel, `"Harry"` als beginnenden Spieler und der Liste `gdpPlayers` als initialen Spielstand startet.

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

Beispiel:

```

Willkommen_zu_Wuerfel-Black-Jack!↵
Aktueller_Spielstand:↵
Harry:_0_Punkte↵
Lisa:_0_Punkte↵
Spieler_Harry_ist_an_der_Reihe._Moechte_Harry_wuerfeln?(ja/nein)_ja↵
Wuerfeln..._Der_Wuerfel_zeigt_1↵
Aktueller_Spielstand:↵
Harry:_1_Punkte↵
Lisa:_0_Punkte↵
Spieler_Lisa_ist_an_der_Reihe._Moechte_Lisa_wuerfeln?(ja/nein)_nein↵
Aktueller_Spielstand:↵
Harry:_1_Punkte↵
Lisa:_0_Punkte↵
Spieler_Harry_ist_an_der_Reihe._Moechte_Harry_wuerfeln?(ja/nein)_\<nein↵
Spieler_Harry_hat_mit_1_Punkten_gewonnen.↵

```

```

let main(): Unit =
  putline "Willkommen zu Wuerfel-Black-Jack!"
  blackjack gdpWuerfel "Harry" gdpPlayers

```

- g) Achten Sie darauf, dass Ihr Programmcode möglichst lesbar ist und keine unnötig komplexen Ausdrücke enthält (vgl. Übungsblatt 5 Aufgabe 3). 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 Paares ist ein boolescher Wert, der angibt, ob der reguläre Ausdruck nullable ist. Die erste Komponente des Paares 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 Paares) 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 Standardbibliothek³ 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 Paares (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, isNullable) = automaton |> Map.find r
                    let transitions = transitions |> Map.add x r'
                    automaton |> Map.add r (transitions, isNullable)
            ) 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))
            )
        )
    )

type Alphabet2 = | A | B

let alphabetRegex: Reg<Alphabet2> = // (ab)(ab)*|(ba)(ba)*
    Alt (
        Cat (
            Cat (Sym A, Sym B),
            Rep (Cat (Sym A, Sym B))
        )
    )
```



```
),  
Cat (  
  Cat (Sym B, Sym A),  
  Rep (Cat (Sym B, Sym A))  
)  
)
```

- 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 Graphviz⁴ 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.

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

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