

Übungsblatt 9: Grundlagen der Programmierung (WS 2023/24)

Ausgabe: 09. Januar 2024

Abgabe: 15./16./17. Januar 2024, siehe [Homepage](#)

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 716 bis 752 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.

- 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↵
```

- 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↵
```

Aufgabe 2 Black Jack (23 Punkte)

Motivation: In dieser Aufgabe sollen Sie anhand eines etwas komplexeren Projekts die Funktionen der Ein- und Ausgabe einüben. Sie können sich an den Vorlesungsfolien 716 bis 752 sowie am Skript Kapitel 7.1 orientieren.

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

Harry Hacker möchte beim Black Jack sein Weihnachtsgeld verdoppeln. Nachdem Lisa Lista ihn vergeblich vor den Gefahren des Glücksspiels gewarnt hat, bietet sie an, für Harry eine vereinfachte Black Jack Simulation zu schreiben, mit der Harry etwas üben kann.

Sie schlüpfen in den folgenden Teilaufgaben in die Rolle von Lisa Lista, indem Sie die einzelnen Funktionen der Black Jack Simulation implementieren. Wir vereinfachen für diese Aufgabe die Regeln von Black Jack etwas. Zum einen tätigen wir keine Einsätze und zum anderen gibt es nur einen Spieler bzw. nur eine Spielerin und den Croupier, wobei der Croupier vom Computer gespielt wird.

Um im Folgenden den Lesefluss zu vereinfachen meinen wir mit Spieler sowohl männliche als auch weibliche Spielende.

1. Der Spieler und der Croupier erhalten jeweils eine Karte.
2. Der Spieler kann nun so lange weitere Karten ziehen, bis er der Meinung ist nahe genug an den 21 Punkten zu sein. Er kann dann seinen Zug beenden. Hat er über 21 Punkte, so hat sich der Spieler „überkauft“ und verliert das Spiel sofort.
3. Hat der Spieler seinen Zug beendet, ist der Croupier an der Reihe. Dieser muss so lange Karten ziehen, bis er mindestens 17 Punkte hat. Überschreitet er dabei die 21 Punkte, hat der Spieler direkt gewonnen.
4. Hat auch der Croupier seinen Zug beendet, so gewinnt, wer näher an den 21 Punkten ist. Haben der Spieler und der Croupier die gleiche Punktezahl erreicht, so endet das Spiel unentschieden.

Zur Berechnung der Punktzahl der Spielkarten gilt es einige Besonderheiten zu beachten:

- Jeder Spielkarte ist ein numerischer Wert zugeordnet (mehr dazu in Kürze).
- Die Spielkarte Ass kann den Wert 11 oder 1 annehmen.
- Der Croupier muss ein Ass immer als 11 Punkte zählen, sofern er dadurch nicht die 21 Punkte überschreitet. Würde er die 21 Punkte überschreiten, zählt er es als einen Punkt. Beispiel: Der Croupier zieht eine Sieben und ein Ass. Damit steht er auf 18 Punkten, hat also die Schwelle von 17 Punkten überschritten und muss aufhören zu ziehen. Er darf das Ass nicht als einen Punkt werten, um weiter Karten zu ziehen.
- Für den Spieler wird der Wert des Ass immer zum Vorteil des Spielers gewertet. Würde der Spieler mit dem Ass die 21 Punkte überschreiten, so wird es automatisch als 1 Punkt gezählt.

Die Spielkarten modellieren wir durch den folgenden Variantentyp:

```
type Karte =  
  | Zwei | Drei | Vier | Fuenf | Sechs | Sieben | Acht | Neun // Wert 2-9  
  | Zehn | Bube | Dame | Koenig // Wert 10  
  | Ass // Wert 11 oder 1
```

Die Farben der Karten vernachlässigen wir, da sie im Spiel nicht von Bedeutung sind.

Gewöhnlich wird Black Jack mit sechs oder acht Kartendecks gespielt. Es kann also nicht beliebig oft die gleiche Karte gezogen werden (der Spieler kann durch „Kartenzählen“ darauf schließen wie wahrscheinlich es ist als nächstes eine bestimmte Karte zu erhalten; daher wird auch nicht nur ein einziges Deck für das Spiel verwendet). Diesen Sachverhalt bilden wir in der Simulation nicht ab, wir ziehen die Karten gleichverteilt. Da wir immer nur eine Runde spielen und es nur einen einzigen Spieler gibt, fällt diese Vereinfachung kaum ins Gewicht.

Hinweis: An einigen Stellen können Sie wieder Funktionen des List Moduls sinnvoll einsetzen.

Hinweis: Achten Sie bei den Aufgabenteilen mit Ein- und Ausgabe darauf, dass Ihre Strings exakt mit den Vorgaben übereinstimmen. So führen zum Beispiel auch fehlende Leer- oder Satzzeichen dazu, dass die Testfälle fehlschlagen. Da das Abtippen der Strings eine häufige Fehlerursache darstellt, haben wir diese in der Vorlage mitgeliefert. Sie können in Ihrem Code entweder die Bezeichner verwenden oder Sie können die Strings an den entsprechenden Stellen in Ihrem Code einfügen.

- a) Wir beginnen damit eine Funktion zu schreiben, welche jeder Karte einen Wert zuordnet. Da das Ass zwei verschiedene Werte annehmen kann, geben wir für jede Karte eine Liste möglicher Werte zurück. Schreiben Sie dazu eine Funktion `kartenwert: Karte -> List<Nat>`, die eine Karte nimmt und ihr einer Liste möglicher numerischer Werte zuordnet.

Die Zahlkarten nehmen den numerischen Wert ihrer Zahl an. Die Bildkarten Bube, Dame und König sind jeweils zehn Punkte wert. Das Ass kann einen oder elf Punkte wert sein.

- b) Bisher können wir nur den Wert bzw. die möglichen Werte einer einzigen Karte berechnen. Was uns aber interessiert, ist der Gesamtwert aller Karten eines Spielers. Implementieren Sie dazu die Funktion `kartenPunkte`, die eine Liste von Karten nimmt und eine Liste aller möglichen Punktzahlen zurückgibt, die mit diesen Karten gebildet werden können. Stellen Sie sicher, dass dieselbe Punktzahl nicht mehrfach in der Liste auftaucht.
- c) Nach den eingangs aufgeführten Regeln möchten wir aus den möglichen Punktzahlen für eine Liste von Karten jetzt die gültige Punktzahl auswählen. Schreiben Sie dazu die Funktion `punkteBerechnen: List<Karte> -> Nat`, welche die genannte Zuordnung berechnet.
- d) Schreiben Sie eine Funktion `zugCroupier: (Unit -> Karte) -> List<Karte> -> Option<Karte>`, die eine Funktion `zieheKarte` nimmt, mit der eine Karte gezogen werden kann. Darüber hinaus nimmt sie noch eine Liste der bisher gezogenen Karten des Croupiers. Implementieren Sie hier die eingangs genannte Regel für den Croupier. Zieht der Croupier keine weitere Karte, so soll `None` zurückgegeben werden. Zieht er die Karte `k`, so soll `Some k` zurückgegeben werden.
- e) Implementieren Sie die Funktion `zugSpieler: (Unit -> Karte) -> List<Karte> -> Option<Karte>`, die den Spieler auffordert einen Zug zu tätigen. Die Funktion nimmt als Argumente wieder eine Funktion `zieheKarte`, mit deren Hilfe wir eine Karte ziehen können sowie die Liste der bisher gezogenen Karten des Spielers.

Prüfen Sie zunächst, ob der Spieler mit seinen bisher gezogenen Karten 21 oder mehr Punkte hat. Ist dies der Fall, braucht bzw. kann er keinen weiteren Zug mehr machen und wir können direkt `None` zurückgeben. Ansonsten geben wir zur besseren Übersicht zunächst eine leere Zeile aus und fragen den Spieler dann mit der Nachricht `"Moechten Sie eine weitere Karte ziehen? [j/n]"` ob er wünscht eine weitere Karte zu ziehen. Lehnt er mit `n↵` ab, sind wir fertig und können `None` zurückgeben. Bestätigt er mit `j↵` ziehen wir eine Karte `k` und geben `Some k` zurück. Gibt der Spieler etwas anderes als `j` oder `n` ein, soll die Fehlermeldung `"Ungueltige Eingabe."` erscheinen und die Frage wird wiederholt.

Hinweis: Verwenden Sie die show Funktion, um eine Liste von Karten auszugeben.

Beispielaufruf: `zugSpieler Main.zieheKarte []`

```
↵  
Moechten_Sie_eine_weitere_Karte_ziehen?_[j/n]_42↵  
Ungueltige_Eingabe.↵  
↵  
Moechten_Sie_eine_weitere_Karte_ziehen?_[j/n]_j↵
```

Die Rückgabe könnte z.B. `Some Sieben` sein. Bei der Eingabe von `n↵` wäre die Rückgabe dagegen `None`.

Noch ein weiteres Beispiel: `zugSpieler Main.zieheKarte [Ass; Koenig] = None`

- f) Implementieren Sie eine Funktion `zuegeCroupier: (Unit -> Karte) -> List<Karte> -> Nat`, die eine Funktion zum Ziehen weiterer Karten sowie eine Liste der bisher gezogenen Karten nimmt. Die Funktion soll den Croupier so lange mit der Funktion `zugCroupier` ziehen lassen, bis er keinen weiteren Zug mehr durchführen möchte. Die Funktion soll die Gesamtpunktzahl der gezogenen Karten zurückgeben.

Beachten Sie bei der Ausgabe, dass neu gezogene Karten vorne in die Liste der gezogenen Karten eingefügt werden.

Beispielaufruf: `zuegeCroupier Main.zieheKarte [Zwei]`

↩

`Karten_des_Croupiers: [Fuenf; Dame; Zwei]` ↩

In diesem Beispiel wäre die Rückgabe 17N.

- g) Implementieren Sie eine Funktion `zuegeSpieler: (Unit -> Karte) -> List<Karte> -> Nat`, die eine Funktion zum Ziehen weiterer Karten sowie eine Liste der bisher gezogenen Karten nimmt. Die Funktion soll den Spieler so lange mit der Funktion `zugSpieler` ziehen lassen, bis er keinen weiteren Zug mehr durchführen möchte. Nach jedem Zug soll ausgegeben werden welche Karte gezogen wurde und welche Karten der Spieler nach dem Zug besitzt. Die Funktion soll schließlich die Gesamtpunktzahl der gezogenen Karten zurückgeben.

Beispielaufruf: `zuegeSpieler Main.zieheKarte [Zwei; Drei]`

↩

`Moechten_Sie_eine_weitere_Karte_ziehen? [j/n] j` ↩

`Sie_haben_Koenig_gezogen, damit_haben_Sie_folgende_Karten: [Koenig; Zwei; Drei]` ↩

↩

`Moechten_Sie_eine_weitere_Karte_ziehen? [j/n] j` ↩

`Sie_haben_Drei_gezogen, damit_haben_Sie_folgende_Karten: [Drei; Koenig; Zwei; Drei]` ↩

↩

`Moechten_Sie_eine_weitere_Karte_ziehen? [j/n] n` ↩

In diesem Beispiel wäre die Rückgabe 18N.

- h) In der schon vorgegebenen Funktion `spiel: (Unit -> Karte) -> Unit` werden die Funktionen zum gesamten Spiel zusammengefügt. Die Funktion `spiel` begrüßt zuerst den Spieler und stellt die Startkonfiguration her und gibt sie anschließend aus. Danach werden die Züge des Spielers durchgeführt und zuletzt zieht der Croupier. Zum Schluss wird der Spielausgang ausgegeben. Schauen Sie sich die Funktion `spiel` an und probieren Sie das Spiel mit `dotnet run` aus. (Diese Aufgabe gibt keine Punkte.)
- i) Da es sich hier um eine umfangreichere Aufgabe handelt, erwarten wir, dass Sie Ihren Code lesbar schreiben und ihn verständlich kommentieren. Hierfür vergeben wir bis zu 3 Punkte.

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
```

- 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 $x \setminus r$ 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)`.

- 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 $x \setminus r = r'$ ist und dass r nicht nullable ist.

Das beschriebene Konstrukt ist ein endlicher Automat: Jeder reguläre Ausdruck ist ein Zustand des Automaten. Die $\text{Map}<'T, \text{Reg}<'T>>$ 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 Standardbibliothek² 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 $(\text{Map.empty}, \text{nullable } r)$ hinterlegt wird. Dies ist notwendig, damit rekursive Aufrufe in die Abbruchbedingung aus dem vorherigen Schritt gelangen.
 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 $r' = x \setminus r$. 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 $x \setminus r = r'$ 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.
- 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)*)
```

<sup>2</sup><https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-mapmodule.html>

- 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<sup>3</sup> verarbeiten können sowie F# Code für die Akzeptorfunktion.<sup>4</sup>

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.

---

<sup>3</sup>Den Code können Sie einfach bei <http://www.webgraphviz.com/> einfügen, wenn Graphviz bei Ihnen nicht installiert ist.

<sup>4</sup>Die Datei `Main.fs` enthält Funktionen, die den Automaten in die textuelle Beschreibung für Graphviz und in gültigen F# Programmcode (als String) umwandeln.