Lehrstuhl für Softwaretechnik und Programmiersprachen
Jens Bendisposto

Funktionale Prorammierung – WS 13/14
Projektaufgabe

1 Formales

1.1 Abgabe

Die Projekt-Aufgabe muss bis zum 14.03.2014, 23:59 Uhr abgegeben werden. Bitte legen Sie sich dazu einen Account bei `https://bitbucket.org/` an. Erstellen Sie dort ein **privates** git Repository und teilen Sie den Zugriff mit mir. Mein Username ist `bendisposto`. Legen Sie das Repository schnellstmöglich an um sicherzustellen, dass es keine Probleme gibt. Es ist mir wichtig, dass Sie regelmässig auf Bitbucket pushen. Gewertet wird der letzte vor der Deadline auf Bitbucket gepushte commit.

Eine vollständige Abgabe umfasst folgende Bestandteile.

1. Der Sourcecode der Anwendung
2. Mindestens drei Beispielnetze in dem von Ihnen gewählten Format[1]
3. Eine Readme Datei, die die Installation beschreibt.
4. Ein Dokument, dass Ihre Implementierung beschreibt.

1.2 Regeln

Die Aufgabe ist eigenständig zu lösen. Sie dürfen mit anderen Teilnehmern gerne Ihre Lösungsansätze diskutieren, müssen Ihre Abgabe aber selber implementieren. Da es sich um Teil der Prüfung handelt, wird die Gesamtprüfung bei einem Täuschungsversuch als nicht bestanden gewertet.

1.3 Bewertung

Die Aufgabe wird mit 25 Punkten bewertet. Kriterium für die Bewertung ist in erster Linie die Qualität des Codes. Entwickeln Sie Ihre Anwendung nach den in der Vorlesung diskutierten Prinzipien für Simplicity.

[1] Beispielnetze, die Sie übersetzen können: `http://www.informatik.uni-hamburg.de/TGI/PetriNets/introductions/aalst/`

2 Aufgabenstellung

Es soll ein Simulator für Petri-Netze[2] implementiert werden. Ein Petri-Netz ist einem endlichen Automaten sehr ähnlich.

2.1 Mathematische Beschreibung

Ein Petrinetz besteht aus einer Menge von Transitionsknoten T und Platzknoten P sowie zwei Mengen von Kanten $K_{out} \in T \times P$ und $K_{in} \in P \times T$. Eine Kante k ist immer gerichtet und geht entweder von genau einem Platz zu genau einer Transition (also $k \in K_{in}$) oder von genau einer Transition zu genau einem Platz (also $k \in K_{out}$). Zwischen einem Platz und einem Transitionsknoten darf es höchstens eine Kante geben. Sowohl Plätze als auch Transitionsknoten können ohne Kanten existieren.

Jede Kante k ist mit einer Zahl beschriftet. Die Beschriftung sei durch die Funktionen $\lambda_{in} : K_{in} \to \mathbb{N}_0$ und $\lambda_{out} : K_{out} \to \mathbb{N}_0$ gegeben. Falls $k \in K_{in}$ ist, beschreibt $\lambda_{in}(k)$ wieviele Token aus dem Platz entfernt werden. Falls $k \in K_{out}$ ist, beschreibt $\lambda_{out}(k)$ wieviele Token in dem Platz hinzugefügt werden.

Der Zustand des Petrinets wird durch eine totale Funktion $\tau : P \to \mathbb{N}_0$ beschrieben. Jedem Platz ordnet die Funktion die Anzahl der dort vorhandenen Token zu.

Eine Transition t kann gefeuert werden, wenn für jede Kante k, die in dem Transitionsknoten t mündet, genügend Token im Platz vorhanden sind. Sei $k_1 = (p_1, t), ..., k_n = (p_n, t) \in K_{in}$, dann muss gelten: $\forall i \in 1..n \implies \tau(p_i) \geq \lambda(k_i)$. Der Einfachheit halber werden keine Indexmengen verwendet, stattdessen werden die Plätze und Kanten gleich nummeriert. Da es nur höchstens eine Kante zwischen einem Platz und einer Transition gibt, ist das ohne Beschränkung der Allgemeingültigkeit möglich.

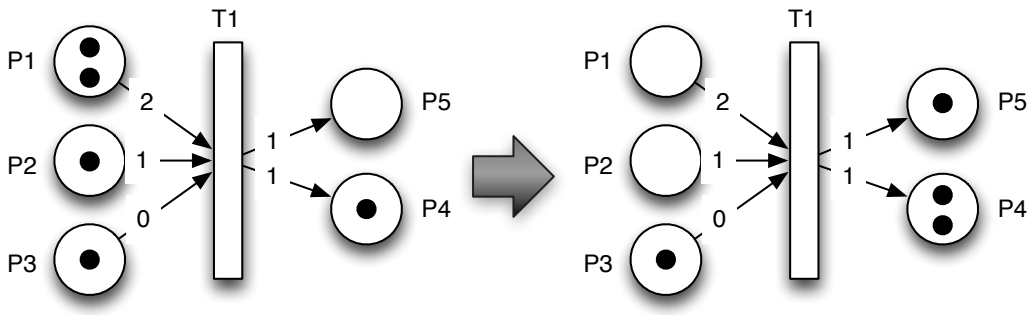
Jedes Petri-Netz hat einen Namen, ebenso jeder Platz und jede Transition. Ob Kanten einen Namen bekommen ist Ihnen überlassen. Hinweise: Sie könnten für alle Elemente eines Netzes ein Namensschema aus lokalem Namen und den[3] Netznamen verwenden. Ein Name muss nicht zwanglsüfig ein einzelner String sein.

2.2 Intuitive Beschreibung

Die folgende Graphik zeigt eine Transition T1 mit drei Eingangsplätzen (P1,P2,P3) und zwei Ausgangsplätzen (P4,P5). Die Kanten zwischen Platz und Transition sind mit der Anzahl der Token beschriftet, die konsumiert bzw. produziert werden. In diesem Beispiel kann T1 feuern, denn in allen drei Plätzen sind genügend Token vorhanden. Im Folgezustand kann T1 nicht mehr feuern, da in P1 und P2 keine Token mehr vorhanden sind, aber 2 bzw. 1 Token benötigt werden.

[2] `http://de.wikipedia.org/wiki/Petri-Netz`

[3] Plural! Wenn Sie zwei Netze kombinieren, benötigen Sie ggf. so etwas wie Pfade aus Netznamen



2.3 Eingabe für den Simulator

Als Eingabeformate soll ein selbstgewähltes DSL Format dienen. Das eigene Format soll keine eigene Syntax haben, die geparsed wird, sondern eine Sammlung von Macros oder Funktionen sein, die ein Petrinetz manipulieren. Hier ein Beispiel, wie eine DSL für Plätze aussehen könnte:

```
(add-place net :p 5) ;; Erzeugt einen Platz mit dem lokalen Namen :p (oder auch "p"),
                     ;; belegt den Platz initial mit 5 Token und fuegt
                     ;; ihn in das Petrinetz net ein.
```

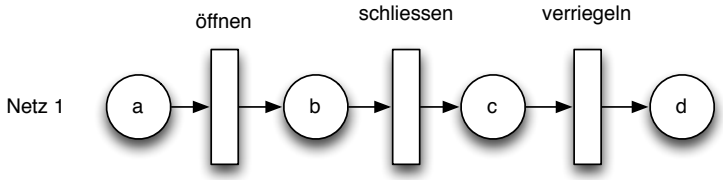
Ihre DSL soll nach Möglichkeit eine interne Struktur konstruieren und diese für die Simulation verwenden.

Folgende Operationen sollen unterstützt werden:

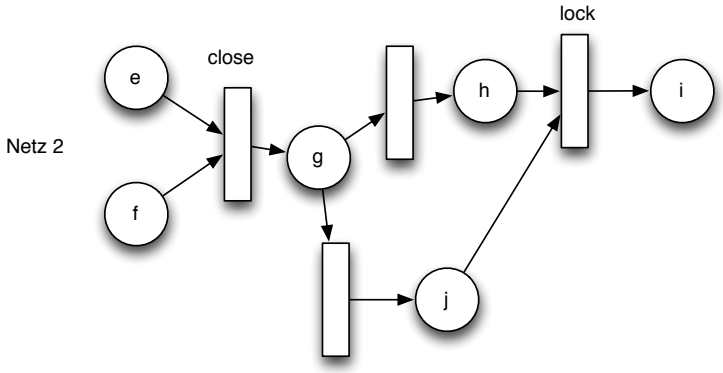
- Erweitern eines Petri-Netzes um Knoten (Plätze/Transitionen) und Kanten.
  Ob Sie Kanten immer zusammen mit Transitionen einfügen oder separat, bleibt Ihnen überlassen. Es müssen aber alle validen Petrinetze konstruiert werden können.
- Zwei Petrinetze werden zu einem einzigen Netz vereinigt. Die Operation soll als Eingabe zwei Netze bekommen und ein Gesamtnetz produzieren. Es muss angegeben werden, welche Transitionen und Plätze aus den beiden Netzen zusammengefasst werden.
- Instanziierung eines Petrinets. Ein Petrinetz wird kopiert, dabei werden alle Transitionen und Plätze systematisch umbenannt.

Ihre DSL soll aus simplen Operationen im Sinne der Definition aus der Vorlesung bestehen. Sie sollten aber auch bequeme Kompositionen der simplen Funktionen bereitstellen. Es soll möglich sein beliebig viele Netze zu definieren und gleichzeitig im Werkzeug zu haben. Es soll auch möglich sein Netze zu entfernen.

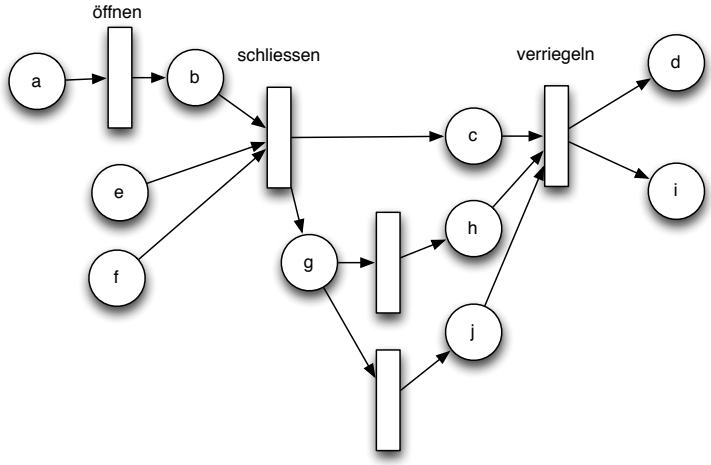
Ein Beispiel für die Vereinigung von Petrinetzen: Netz 1 hat die Transitionen öffnen, schliessen und verriegeln.



Netz 2 hat unter anderem die Transitionen close und lock.



Dann soll es bei der Vereinigung möglich sein zu spezifizieren, dass close und schliessen bzw lock und verriegeln jeweils eine einzige Transition in dem vereinigten Netz sind.



Für Plätze soll analog die Vereinigung spezifiziert werden indem man angibt, welche Plätze identisch sein sollen.

2.4 Laden und Speichern

Implementieren Sie Funktionen um Netze aus einer Datei zu laden und in eine Datei zu speichern.

2.5 Simulation

Unter Simulation verstehen wir das schrittweise Ausführen eines der definierten Petrinetze. Ausführen bedeutet, dass eine Transition des Netzes die feuern kann gewählt und gefeuert wird. Die Auswahl wird dem Nutzer überlassen (der aber auch dem Tool sagen kann, dass eine zufällige Transition gewählt werden soll). Die Simulation wird nach dem Feueren der Transition in dem Folgezustand des Netzes fortgesetzt. Wenn mehrere Netze definiert sind, werden alle Netze parallel animiert (d.h. dem Benutzer werde aus allen Netzen alle feuerbaren Transitionen zur Auswahl angeboten). Es soll eine Funktion geben, die es dem Benutzer erlaubt den Simulator automatisch für eine festgelegte Zeit bzw. für eine festgelegte Anzahl von Schritten laufen zu lassen. Der Simulator wählt aus den zur Verfügung stehenden Transitionen zufällig aus.

Es soll möglich sein automatisch in jedem Zustand Eigenschaften zu überprüfen. Eigenschaften werden per DSL dem Petri-Netz hinzugefügt. Als Eigenschaften in unserem Werkzeug kommen in Betracht:

- **NetAlive**: Es gibt mindestens eine Transition, die gefeuert werden kann
- **TransitionAlive** t_1 t_2 **...**: Mindestens eine der Transition t_1, t_2, ... kann gefeuert werden
- **NonEmpty** p_1 p_2 **...**: Es gibt mindestens ein Token im einem der spezifizierten Plätze
- **Not** und **Or** als logische Junktoren für die Eigenschaften.

2.6 User Interface

Es soll zwei Benutzerschnittstellen geben: die REPL und eine graphische Schnittstelle. In der REPL soll ihr Namespace mit use geladen werden und dann per Hand ein Netz erstellt und simuliert werden können.

Bei dem graphischen User-Interface ist Ihre Kreativität gefragt. Sie können die Applikation als Standaloneanwendung (z.B. mit Swing, SWT, JavaFX, etc.) schreiben oder auch als Webanwendung. Schön wäre eine graphische Darstellung des Netzes, es ist aber nicht obligatorisch. Die GUI soll die Ausführung der wesentlichen Funktionen (Erstellen/Laden/Speichern und Simulation) erlauben. Wenn Sie keinen Zugriff auf das Dateisystem haben (zum Beispiel, weil Sie ihre Anwendung als reine ClojureScript Anwendung schreiben) ist es auch akzeptable, wenn man den Dateiinhalt in ein Textfeld kopieren muss.

2.7 Erlaubte Bibliotheken

Jede Bibliothek, die sie wollen, ausser Bibliotheken zur Petrinetz Simulation! Es gibt zur Darstellung von Graphen eine Reihe von brauchbaren Java und JavaScript Bibliotheken. Einige Stichworte zum Füttern von Google: Java Universal Network/Graph Framework (JUNG), Domain Driven Documents D3, ClojureScipt C2, Viz.js, Grappa

2.8 Implementierung

Ihre Anwendung soll ein funktionales Programm in Clojure oder ClojureScript sein. Wenn es erforderlich ist, können kleine Teile in anderen Sprachen implementiert werden. Bibliotheken, die Sie verwenden dürfen auch in anderen Sprachen implementiert sein. Es müssen aber Bibliotheken sein, die per öffentlichem Maven Repository verfügbar sind und es sollen keine nativen Bibliotheken benutzt werden[4].

2.9 Docstrings, Tests, Schemas, Types, Contracts

Bitte schreiben sie für alle öffentlichen Funktion einen Docstring, der Funktion und Parameter **kurz und präzise** beschreibt. Ob Sie Tests, Typen, Contracts oder Schemas verwenden bleibt Ihnen überlassen. Es wird aber dazu geraten die Codequalität durch eines oder mehrere dieser Werkzeuge sicherzustellen. Im Zweifel wird das Vorhandensein von (sinnvollen!!!) Tests, etc. zu Ihren Gunsten gewertet.

3 Dokumentation

Genauso wichtig wie die Anwendung selber ist die Dokumentation des Projekts. Sie sollen hier nicht beschreiben, was im Quelltext sehr einfach nachgelesen werden kann. Das Dokument soll die Architekturentscheidungen dokumentieren. Insbesondere sollen Sie beschreiben, wie Simplicity erreicht wurde, wo es Tradeoffs gab, und wo komplexe Konstrukte nötig waren.

Es ist wichtig, dass Sie über Ihren Quellcode reflektieren und verschiedene Ansätze ausprobieren. Dokumentieren Sie auch Ansätze, die Sie verworfen oder radikal geändert haben. Ich empfehle Ihnen während der Implementierung eine Art Forschungstagebuch zu führen in dem Sie ihre Ansätze und Ideen festhalten. Das wird die Erstellung des Dokuments radikal vereinfachen.

4 Installationsanleitung

Die Installationsanleitung soll **knapp** beschreiben, welche Schritte zum Bauen und Starten der Anwendung nötig sind. Benutzen Sie nach Möglichkeit das Buildttool leiningen.

[4] Ihr Programm muss auf meinem Mac Book laufen