Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Informatik

Programming Concepts and Paradigms (PCP)

PCP-Übung zu Prolog 5 + 6

Hauptthemen: Negation, Constraint-Logik-Programmierung (CLP), HTTP-Kommunikation, Umgang mit JSON
Zeitfenster: ca. 2-6 Lektionen

Ruedi Arnold

Diese Übung repetiert und vertieft den in Prolog 5 und 6 vermittelten Stoff. Gehen Sie zur Vorbereitung nochmals durch die Unterrichtsfolien durch und schauen Sie insbesondere, dass Sie alle auf den Folien angegebenen Code-Beispiele verstanden haben.

1. Fibonacci-Zahlen Berechnen mit CLP-R

In der Vorlesung wurde erwähnt, dass die gezeigte Berechnung (Prädikat fib_clp/2) von Fibonacci-Zahlen mit CLP-R ein Problem hat, wenn das zweite Argument von fib_clp/2 gar keine Fibonacci-Zahl ist (siehe Prolog 5, Folie 31). Dasselbe Problem tritt auch auf, wenn zu einer gegeben Fibonacci-Zahl nach der ersten, korrekten Antwort weitere Lösungen (durch drücken von ; bzw. der Leerschlagtaste) gesucht werden.

a) Wie manifestiert sich dieses Problem in SWI-Prolog?

b) Wieso tritt dieses Problem auf?

c) Lässt sich das oben beschriebene Problem einfach beheben? Falls ja: Modifizieren Sie das Prädikat entsprechend und testen sie das neue Prädikat. Falls Nein: Begründen Sie Ihre Antwort.

2. Anwendung der Negation: Mengen-Differenz

Implementieren Sie die Relation `set_difference(Set1, Set2, SetDifference)`, bei welcher in `SetDifference` alle Elemente sind, welche in `Set1` sind, jedoch nicht in `Set2`. Alle drei Mengen werden dabei als Listen repräsentiert. Zwei Anwendungsbeispiele:

```
?- set_difference([a, b, c, d], [b, d, e, f], [a, c]).
true .
?- set_difference([1, 2, 3, 4, 5, 6], [2, 4, 6], L).
L = [1, 3, 5] .
```

Hinweise: Verwenden Sie für Ihr Prädikat set_difference/3 das eingebaute Negations-Prädikat \+/1 und das Listen-Zugehörigkeits-Prädikat member/2, oder unsere eigene Variante davon: mem/2.

3. CLP: Rätsel lösen

In der Vorlesung haben Sie gesehen, wie mit Hilfe von Constraint-Logik-Programmierung (CLP) Rätsel gelöst werden können.

a) Lösen Sie mit Hilfe von CLP in Prolog das folgende Rätsel: "Die Tochter ist 15 Jahre alt, die Mutter dreimal so alt. In wie vielen Jahren wird die Mutter nur noch doppelt so alt sein wie ihre Tochter?"

b) Lösen Sie analog zum in der Vorlesung gelösten Zahlenrätsel (`SEND + MORE = MONEY`) mit einem eigenen Prolog-Programm das folgende Zahlenrätsel:

```
  DONALD
 +GERALD
 -------
  ROBERT
```

4. Familiäre Beziehungsabfragen & Sudokus lösen via HTTP

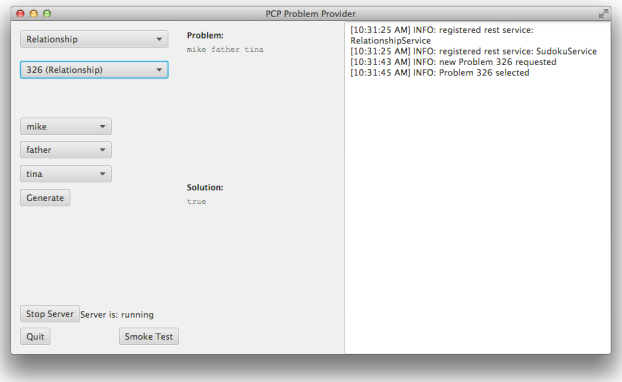
Als eigentliche Abschlussübung wollen wir nun sehen, wie wir Prolog zusammen mit Java verwenden können. Die Grundidee ist, dass wir Prolog für Dinge verwenden, für welche Prolog geeignet ist, also für Wissensrepräsentation und -abfrage (z.B. familiäre Beziehungen) oder für das Lösen von Rätseln (z.B. Sudokus).
In dieser Aufgabe verwenden Sie also ein Java-Programm, welches Problem-Instanzen erzeugt und diese visualisiert. Diese Java-Komponente nennen wir Problem-Provider, und unser Problem-Provider stellt Problem-Instanzen in den zwei Domänen "Familiäre Beziehungen" und "Sudoku" zur Verfügung.

a) Der Problem-Provider ist in Form von einer Java Archiv-Datei (jar) online verfügbar im GitLab-Repo vom PCP-Modul. Laden Sie diese Datei herunter und starten Sie das Programm mittels:

```
> java -jar PCPProblemProvider-Java-X.jar
```

Hinweis: Dieses Programm benötigt **Java 8 oder 9**, es liegen zwei unterschiedliche jar-Dateien im Repo unter https://gitlab.enterpriselab.ch/PLAB/plab_public. Verwenden Sie also zwingend Java Version 8 oder 9, mit Java Version >= 10 sind diese jar-Dateien nicht lauffähig. ☹

Dies Startet den Problem-Provider, d.h. Sie sehen die entsprechende graphische Benutzerschnittstelle, siehe Screenshot nächste Seite. Der ProblemProvider nimmt nun HTTP-GET und –POST Anfragen entgegen unter dem Port 16316. (Optional können Sie den gewünschten Port beim Start von der jar-Datei als Argument mitgeben, als z.B. mittels `java -jar PCPProblemProvider.jar 17777`.)
Testen Sie den ProblemProvider, indem Sie in diesem links unten den Knopf "Smoke Test" (siehe Screenshot unten) drücken. Dadurch sollte Ihr Standard-Browser aufgehen und testweise einen HTTP-GET-Request auf den ProblemProvider absetzten.



b) Implementieren Sie in Prolog ein Programm, welches Anfragen zu familiären Beziehungen (siehe Übung Woche 2) beantworten kann. Ihr Programm soll am Schluss wie folgt in der Prolog-Konsole aufgerufen werden können:

```
?- solve(relationship, 326).
```

Ihr zu implementierendes Prädikat solve/2 soll als erstes Argument den Problem-Typen (hier: relationship) und als zweites Argument die ID vom zu lösenden Problem entgegen nehmen. Um überhaupt Problem-Instanzen zur Verfügung zu haben, müssen Sie diese zuerst im Problem-Provider erzeugen, indem Sie im Auswahl-Menu zwei Personen und eine Beziehung anwählen und dann den Knopf "Generate" drücken. Im Auswahl-Menu oben können Sie anschliessend Ihre erzeugte Problem-Instanz anwählen und anzeigen lassen. Jede Problem-Instanz erhält bei der Erzeugung eine ID, diese Zahl verwenden wir zur Identifikation von Problem-Instanzen. Im Beispiel-Screenshots rechts oben sehen Sie, wie Im Problem-Provider eine Problem-Instanz für die Anfrage `father(mike, tina)` erzeugt wird.
Ihr Prolog-Programm soll dann also via HTTP-GET-Anfrage beim Problem-Provider eine Problem-Instanzen abholen, das Problem lösen und die Lösung in einer HTTP-POST-Anfrage dem Problem-Provider zurück liefern. Die Daten werden jeweils in einem JSON-Format übertragen. Hier ein Beispiel der entsprechenden HTTP-Kommunikationssequenz:

```
GET http://localhost:16316/problem/relationship/396
```

Diese Anfrage liefert eine Antwort in folgendem JSON-Format zurück:

```
{
  "firstPerson":"mike",
  "problemKey":326,
  "relationship":"father",
  "secondPerson":"tina"
}
```

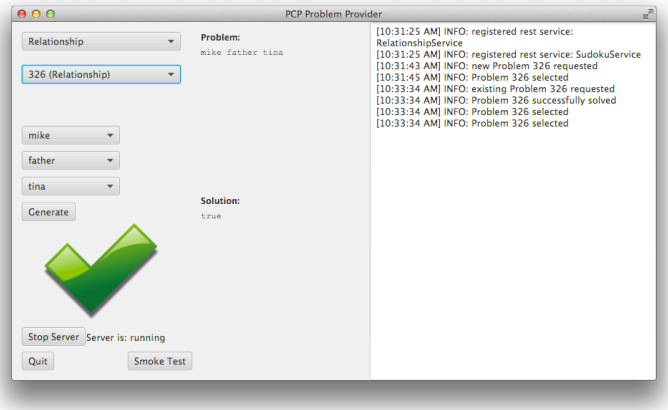
Diese Anfrage soll dann in Prolog gelöst werden und danach die Antwort mit Hilfe von einer POST-Anfrage an den Problem-Provider zurück geschickt werden:

```
POST http://localhost:16316/problem/relationship
```

Die Antwort wird dabei in folgendem JSON-Format als POST-Argument mitgegeben:

```
{
  solution=true,
  problemKey=326
}
```

Wenn der Problem-Provider die korrekte Antwort aus Prolog erhält, wird dies entsprechend mit einem grünen Häkchen dargestellt, siehe Screenshot rechts.



Hinweis: In der Vorlesung Prolog 6 wurden zu dieser Übung diverse Informationen und Hinweise gegeben, schauen Sie sich diese an und berücksichtigen Sie diese beim Lösen dieser Aufgabe.

c) Analog zum Lösen von Anfrage bezüglich familiärer Beziehungen soll nun Ihr Prolog-Programm so erweitert werden, dass es Sudokus lösen kann. Der Ablauf ist analog zur letzten Aufgabe, ihr Prolog-Programm soll also wie folgt aufgerufen werden können:

```
?- solve(sudoku, 326).
```

Ihr Prolog-Programm soll dann also wie vorhin via HTTP-GET-Anfrage beim Problem-Provider eine Problem-Instanzen abholen, das Problem lösen und die Lösung in einer HTTP-POST-Anfrage dem Problem-Provider zurück liefern. Die Daten werden jeweils in einem JSON-Format übertragen. Hier ein Beispiel der entsprechenden HTTP-Kommunikationssequenz:

```
GET http://localhost:16316/problem/sudoku/140
```

Diese Anfrage liefert eine Antwort in folgendem JSON-Format zurück:

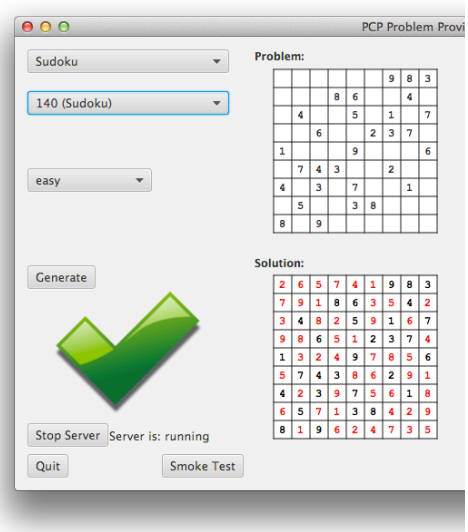
```
{
  "problemKey":140,
  "sudoku":
    [
      [0,0,0,0,0,0,9,8,3],
      [0,0,0,8,6,0,0,4,0],
      [0,4,0,0,5,0,1,0,7],
      [0,0,6,0,0,2,3,7,0],
      [1,0,0,0,9,0,0,0,6],
      [0,7,4,3,0,0,2,0,0],
      [4,0,3,0,7,0,0,1,0],
      [0,5,0,0,3,8,0,0,0],
      [8,0,9,0,0,0,0,0,0]
    ]
}
```

Dieses Sudoku soll dann in Prolog gelöst werden und danach die Lösung mit Hilfe von einer POST-Anfrage an den Problem-Provider zurückgeschickt werden:

```
POST http://localhost:16316/problem/sudoku
```

Die Antwort wird dabei in folgendem JSON-Format als POST-Argument mitgegeben:

```
{
  solution=
    [
      [2,6,5,7,4,1,9,8,3],
      [7,9,1,8,6,3,5,4,2],
      [3,4,8,2,5,9,1,6,7],
      [9,8,6,5,1,2,3,7,4],
      [1,3,2,4,9,7,8,5,6],
      [5,7,4,3,8,6,2,9,1],
      [4,2,3,9,7,5,6,1,8],
      [6,5,7,1,3,8,4,2,9],
      [8,1,9,6,2,4,7,3,5]
  ],
  problemKey=140
}
```



Auch hier wird im Problem-Provider entsprechend ein grünes Häckchen angezeigt, wenn die richtige Antwort kommt, siehe Screenshot rechts.

Hinweis: Achten Sie darauf, dass Sie nicht zu viel Code von b) duplizieren, sondern diesen nach Möglichkeit umbauen und wiederverwenden, ganz gemäss dem wohl wichtigsten Design-Prinzip "DRY" (Don't repeat yourself)! ☺

Damit haben Sie mittels unserem Problem-Provider erfolgreich Probleme aus zwei Domänen in Java erzeugt und dargestellt, und in Prolog (hoffentlich) elegant deklarativ-logisch gelöst. Und dazwischen mit Hilfe der HTTP- und JSON-Standards kommuniziert. ☺