TINF 13 AI-BI

Duale Hochschule Baden-Württemberg Mannheim

Studienarbeit Modul T3201 DBHW Mannheim

Implementierung von SetlX-Programmen in Python

Name:	Johann Boschmann	Name:	Joseph Palackal
Matrikelnummer:	7644311	Matrikelnummer:	5839999
Kurs:	TINF13AIBI		
Studiengangsleiter:	Prof. Dr. A. Müller	Betreuer:	Prof. Dr. K. Stroetmann
Datum:	25.05.2016		
Zeitraum:	14.09.2015 – 30.05.2016	Unterschrift Betreuer:	_____

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

In der Vorlesung „Grundlagen und Logik“ des Moduls Theoretische Informatik I führt der Dozent Prof. Dr. Karl Stroetmann die Programmiersprache SetlX ein. SetlX ist eine auf Java basierende Sprache, die sehr gut geeignet ist, um den Pseudocode aus Vorlesungen ausführbar zu machen. Diese Programmiersprache wirbt damit, dass die Verwendung von Mengen und Listen sehr gut unterstützt wird. Außerdem können Ausdrücke aus der Mengenlehre, so wie andere mathematischen Ausdrücke in einer Syntax, die sehr ähnlich zur mathematischen Notation ist, implementiert werden. (Stroetmann & Herrmann, 2015) Da diese Vorlesung bereits im ersten Semester stattfindet und die Studenten parallel dazu eine Vorlesung aus dem Modul Mathematik I besuchen, können die Studenten Themen wie beispielsweise die Mengenlehre schneller kennen lernen. Die komplementäre Auseinandersetzung mit ähnlichen bis gleichen Themen in beiden Vorlesungen ermöglicht das gleichzeitige Lernen für zwei Vorlesungen.

Ein weiterer Vorteil für die Studenten ist, dass die Syntax von SetlX, zusätzlich zum sehr mathematischen Stil, auch starke Ähnlichkeiten zur Programmiersprache C aufweist. Selbst für die Studenten, die zuvor keinen Kontakt mit C hatten ist das ein großer Vorteil, da im ersten Semester parallel zur theoretischen Informatik Vorlesung auch eine Vorlesung mit dem Titel „Programmieren in C“ besucht werden muss. So muss kein starkes Umdenken stattfinden, wenn von SetlX zu C und auch umgekehrt gewechselt wird.

Die Hauptintention dieser Arbeit ist es, zu prüfen, ob es möglich wäre die SetlX-Programme, die in der Vorlesung gezeigt werden, in Python-Skripte zu übersetzen. Ziel dabei ist es, die Eleganz der Programme beizubehalten, damit die Studenten die verschiedenen Algorithmen besser verstehen und erlernen können. Auch wenn es in Python teilweise möglich wäre das Problem über einen anderen Weg, eventuell mit weniger Codezeilen, zu lösen, soll dennoch der Hauptfokus auf den Lernprozess in der theoretischen Informatik liegen.

2. Warum Python?

Viele Informatik-Kurse oder Vorlesungen für Anfänger benutzen die Programmiersprache Python als erste Programmiersprache. Von den 39 besten Einführungskursen für Informatik in den USA verwendeten im Jahr 2014 27 Kurse Python als erste Programmiersprache. (Guo, 2014) Mit 69% ist Python somit, mit einer eindeutigen Mehrheit die meist verwendete Programmiersprache unter diesen Kursen. Einige Internet-Artikel, die die Beliebtheit von heutigen Programmiersprachen beleuchten, referenzieren öfter den Blogbeitrag für die Association for Computing Machinery (ACM). In dem Beitrag wird beschrieben, dass Python Java als häufigste Programmiersprache für Anfänger abgelöst hat. Auch wenn der Artikel bereits 2014 veröffentlicht wurde, lässt sich vermuten, dass die Verbreitung von Python nicht zurückgegangen ist. Grund hierfür ist die steigende Beliebtheit der Sprache nach dem TIOBE Index[1], wie auch ein fünfter Platz in der Statistik von Coding Dojo[2].

Die Online-Lernplattform Udacity verwendet für den Kurs „Intro to Computer Science“ Python als Sprache, um die Themen der theoretischen Informatik zu erläutern. Diesen Online-Kurs haben bereits über 500.000 Personen besucht.[3] Als Proargumente werden die Mächtigkeit, die leichte Erlernbarkeit und die weite Verbreitung aufgeführt.

[1] http://www.tiobe.com/tiobe_index
[2] http://www.codingdojo.com/blog/9-most-in-demand-programming-languages-of-2016/
[3] Stand 09.05.2016

3. Skripte ohne spezielle Module

Es ist bereits möglich einige SetlX-Programme ohne zusätzliche Module in Python anzufertigen. Diese Python-Skripte wurden als erstes angefertigt, um feststellen zu können, ob es möglich ist Python Syntax zu verwenden, ohne die Eleganz des Codes zu verlieren.

Beim Schreiben von Python-Skripten muss, im Gegensatz zu SetlX, bei der Einrückung auf eine Besonderheit geachtet werden. In Python ist es möglich sogenannte unsichtbare Fehler zu erhalten, indem man eine inkonsistente Einrückung verwendet. Es ist möglich sowohl mit Leerzeichen, als auch mit Tabulatoren die Einrückung zu gestalten. Was jedoch verboten ist, ist die Verwendung beider Arten gleichzeitig, da Python daraufhin auf einen Fehler stößt. Deshalb ist es wichtig bereits am Anfang festzulegen, ob mit Leerzeichen oder mit Tabulatoren eingerückt wird. Alle Python-Skripte, die im Rahmen dieser Arbeit erstellt wurden, werden Leerzeichen zur Einrückung verwendet.

Das erste Codebeispiel aus dem Logik-Skript befasst sich mit der Berechnung einer Summe der Zahlen von 1 bis zur eingegebenen Zahl. Dieses Programm lässt sich auch nahezu identisch in Python abbilden. Das originale SetlX Programm verwendet hierfür eine Menge, die die Zahlen von 1 bis zur eingegebenen Zahl n enthält. Über den „+/“-Operator wird die Summe aller Zahlen in der Menge ermittelt und ausgegeben.

```
n := read("Type a natural number and press return: ");
s := +/ { 1 .. n };
print("The sum 1 + 2 + ... + ", n, " is equal to ", s, ".");
```

In Python wurde fast dasselbe Verhalten nachgebildet. Jedoch wurde, statt einer Menge, eine Range der Zahlen von 0 bis n verwendet. Die Summe wird über die in Python bereits integrierte Funktion `sum()` berechnet und daraufhin ausgegeben.

```
n = int(input('Type a natural number and press return: '))
s = sum(range(n + 1))
print('The sum 1 + 2 + ... + ', n, ' is equal to ', s, '.')
```

Allgemein kann gesagt werden, dass ein SetlX-Programm, ohne spezielle Funktionen oder Strukturen die nicht in Python wiedergefunden werden, meist ähnlich in Python nachgebildet werden kann.

4. Python Modul `lecture`

Das Verhalten der Mengen in SetlX ist in einigen Bereichen anders als bei den Mengen in Python. Die Mengen in Python dürfen nur gewisse Werte enthalten, diese müssen unveränderbar sein Zusätzlich werden auch gewisse Funktionen von den Python-Mengen nicht unterstützt, die in SetlX häufig verwendet werden. Aus diesen Gründen wurde das Python Modul lecture im Rahmen dieser Studienarbeit angefertigt.

Ein großes Problem, das sehr früh erkannt wurde, ist, dass die Mengen in Python auf den Hashwerten der enthaltenen Elemente operieren. Somit ist es beispielsweise verboten Mengen in Mengen zu hinterlegen, da diese verändert werden könnten und somit der alte Hashwert ungültig werden würde. Diese Funktionalität wird jedoch in den SetlX Programmen, der Vorlesungen mehrfach verwendet. Während analoge Python-Skripte den in *Abbildung 1* gezeigten Fehler zurückgeben. Zwar könnten bereits vorhandene sets in frozensets umgewandelt werden um Mengen in Mengen zu ermöglichen. Allerdings würden auch hier die umgewandelten Mengen nicht mehr geändert werden können, da die Mengen nur so ihren Hashwert beibehalten. Die Unveränderbarkeit der Mengenelemente ist von Python dadurch bedingt, dass eine immer wiederkehrende Hashberechnung vermieden werden möchte. Eine Änderung wird durch die Art der Abspeicherung bereits verhindert. Den Elementen einer Menge werden Hashwerte zugewiesen und sobald diese feststehen dürfen sich die Elemente nicht mehr ändern, da das unmittelbar eine neue Hashberechnung verlangen würde.

Eine weitere Möglichkeit, das Problem der Mengen zu lösen, wäre die Verwendung von Listen, anstelle von Mengen. Prinzipiell ist das in einigen Python-Übersetzungen der SetlX Programme möglich und wurde so auch teilweise umgesetzt. Es werden andere Datentypen verwendet, um die Informationen zu hinterlegen, meist Listen statt Mengen, da für die Ausführung einiger SetlX-Programme keine besonderen Eigenschaften der Mengen benötigt werden. Ein großes Problem an dieser Lösung ist allerdings, dass Listen nun mal keine Mengen sind und sobald Mengeneigenschaften oder Mengenoperatoren, die nicht für Listen gelten, verwendet werden, Listen eher ungeeignet sind. Der Workaround, besondere Funktionen für die Listen zu schreiben, um das Verhalten von Mengen zu imitieren, wurde auch als Ansatz bedacht, allerdings nach einigen kleinen Beispielübersetzungen wieder verworfen. Da eine wichtige Anforderung, die Erhaltung der Eleganz nicht erfüllt werden konnte. Somit war dies keine Lösung, die global für alle Programme verwendet werden konnte. Beispiele für die benötigten Funktionen, die implementiert werden mussten, um Mengeneigenschaften zu imitieren, waren das Entfernen von Duplikaten aus einer Liste, so wie die Ermittlung der Differenz zweier Listen und die Ermittlung der Potenzmenge.

```
>Traceback (most recent call last):
>  File "<stdin>", line 1, in <module>
>TypeError: unhashable type: 'set'
```

Abbildung 1: Fehler bei Mengen in Mengen

Allerdings sind Mengen nicht der einzige Grund, warum das Modul lecture benötigt wird. Ein in SetlX sehr hilfreiches Konstrukt, namens `match`, wird in Python nicht wiedergefunden. Dessen Syntax ist an die sehr bekannte Switch-Case-Syntax angelehnt, welche in Python nicht enthalten ist. Bei Matches können vier verschiedene Datentypen verwendet werden: Strings, Listen, Mengen und Terme. Das Matchen von Strings, Listen und Mengen kann für das Erkennen des ersten Zeichens und dem Rest oder auch das Herauspicken von Paaren verwendet werden. In dem SetlX-Tutorial wird das

Matching beispielsweise zur Generierung des Inversen oder das Erstellen einer sortierten Liste aus einer Menge verwendet. (Stroetmann & Herrmann, 2015)

Die interessanteste Anwendung von Matches findet sich jedoch, bei Termen. Die „[…]Art von Matchen [in SetlX] ist ähnlich zum Matching das in den Programmiersprachen Prolog und ML gegeben ist.“ (Stroetmann & Herrmann, 2015) Dieses Matching wird auch in einigen Programmen der Logik-Vorlesung, die als Grundlage dient, verwendet. Deshalb ist es wichtig, dass diese Funktion auch in einer Python Version der Programme möglich ist.

4.1. Sets

In den Vorlesungs-Programmen, die im Fokus dieser Arbeit stehen, werden häufig Mengen, sowie Ausdrücke aus der Mengenlehre sehr ähnlich zur mathematischen Darstellung verwendet. Neben den Mengenoperationen werden zusätzlich diverse Eigenschaften von Mengen implementiert. Beispielsweise wird genutzt, dass Mengen keine Duplikate enthalten. In SetlX wird eine Sortierung der Elemente durchgeführt, wodurch Vorteile in der Programmierung entstehen.

Um die Mengen, wie sie in den SetlX-Programmen verwendet werden, auch in Python verwenden zu können wurde eine eigene Mengen Klasse implementiert, die alle notwendigen Funktionalitäten erfüllt.

Zur internen Sicherung von Elementen innerhalb der Mengen wird der Datentyp SortedListWithKey aus dem Modul sortedcontainers verwendet. Das Besondere an SortedListWithKey ist, dass es sich nicht nur um sortierte Listen handelt, sondern die Möglichkeit besteht festzulegen nach welcher Eigenschaft die Listen-Objekte sortiert werden sollen. In dem Schlüssel einer SortedListWithKey wird diese Eigenschaft hinterlegt und kann auch dort eingesehen werden. Sodass bei Mengen die interne SortedListWithKey zurückgegeben wird, da Mengen sich nicht sortieren lassen, aufgrund der Operator Überladungen zur Anpassung an die SetlX Syntax.

In der Implementierung der Mengen werden für viele Operationen Operatoren verwendet, wobei versucht wurde möglichst nahe der SetlX Syntax zu bleiben. Die Ähnlichkeit zu SetlX soll vorhanden sein, da diese bereits sehr mathematisch ist und somit sinnvoller zu lernen ist für die Studenten. Die unterstützten mathematischen Operatoren sind in *Tabelle 1* zu sehen.

Operator	Bedeutung
`+`	Bildet Vereinigung zweier Mengen
`+=`	Ergebnis der Vereinigung wird in die erste Menge geschrieben
`-`	Bildet Differenz zweier Mengen
`*`	Bildet Schnitt zweier Mengen
`2**`	Bildet Potenz einer Menge
`%`	Bildet symmetrische Differenz zweier Mengen
`<`	Gibt True zurück wenn die linke Menge kleiner ist
`>`	Gibt True zurück wenn die linke Menge größer ist
`>=`	Gibt True zurück wenn die rechte Menge eine Teilmenge der linken ist
`<=`	Gibt True zurück wenn die linke Menge eine Teilmenge der rechten ist
`==`	Gibt True zurück wenn beide Mengen dieselben Elemente enthalten
`!=`	Gibt True zurück wenn die Mengen unterschiedlich sind (Gegenteil zu ==)
`in`	Prüft ob das gegebene Element in der Menge ist

Tabelle 1: Mathematische Operatoren für Sets

Zusätzlich gibt es noch weitere Methoden, die nicht in der mathematischen Darstellung verwendbar sind, dennoch für typische mathematische Operationen verwendet werden. Diese werden anstatt in

der mathematischen Notation, wie Funktionen auf die Mengen ausgeführt. Außer diesen mathematischen Funktionen können noch weitere, nicht-mathematische Operationen, wie beispielsweise die visuelle Darstellung von Mengen, verwendet werden. Eine sehr wichtige Funktion ist die `put`-Funktion. Diese sorgt dafür, dass keine Duplikate in ein Set eingetragen werden können. In *Tabelle 2* sind die restlichen Mengen-Funktionen aufgelistet.

Funktion	Bedeutung
`str`	Stellt die Menge als String dar
`contains`	Dasselbe wie in (siehe *Tabelle 1*)
`Set[i]`	Gibt das Element an `i`-ter Stelle zurück
`cartesian_product`	Bildet das kartesische Produkt zweier Mengen
`arb`	Gibt ein beliebiges (in diesem Fall das erste) Element der Menge zurück
`random`	Gibt ein zufälliges Element der Menge zurück
`put`	Fügt ein Element in die Menge ein
`peek`	Gibt das letzte Element der Menge zurück
`pop`	Gibt das letzte Element der Menge zurück und entfernt es aus der Menge
`sum`	Gibt die Summe aller Elemente in der Menge zurück

Tabelle 2: Funktionen für Sets

An dieser Stelle muss noch bemerkt werden, dass die Selektion des `i`-ten Elements einer Menge auch das Array Slicing (z.B. `[1:2]`), wie es von Python bekannt ist, unterstützt. Mit diesen Operatoren und Funktionen können alle Aufgabestellungen, die in der Vorlesung zur theoretischen Informatik vorkommen, bewältigt werden. Im Nachfolgenden wird eine Anwendung der Mengenoperatoren gezeigt. Das SetlX-Programm, das die Grundlage bietet, wird den Studenten gezeigt, damit sie ein Gefühl dafür bekommen, welche Operation welche Resultate liefert.

```
a := { 1, 2, 3 };
b := { 2, 3, 4 };

c := a + b;
print(a, " + ", b, " = ", c);

c := a * b;
print(a, " * ", b, " = ", c);

c := a - b;
print(a, " - ", b, " = ", c);

c := 2 ** a;
print("2 ** ", a, " = ", c);

print("(", a, " <= ", b, ") = ", (a <= b));

print("1 in ", a, " = ", 1 in a);
```

Abbildung 2: Nutzung von Mengen in simple.stlx

```
a = Set(1,2,3)
b = Set(2,3,4)

c = a + b
print('%s + %s = %s' % (a, b, c))

c = a * b
print('%s * %s = %s' % (a, b, c))

c = a - b
print('%s - %s = %s' % (a, b, c))

c = 2 ** a
print('2 ** %s = %s' % (a, c))

print('%s <= %s = %s' % (a, b, a <= b))

print('1 in %s = %s' % (a, 1 in a))
```

Abbildung 3: Nutzung von Mengen in simple.py

In den Beispielen, die in Abbildung 2 und in *Abbildung 3* zu sehen sind, werden einige Operatoren sowohl in SetlX, als auch in Python dargestellt. Zuerst werden die Vereinigung, der Schnitt und die Differenz zweier Mengen gebildet. Daraufhin die Bildung einer Potenzmenge, die Prüfung einer Teilmengen-Relation und die Prüfung, ob ein Element sich in einer Menge befindet, gezeigt.

Die Syntax zur Erzeugung einer Menge unterscheidet sich bereits, allerdings sind alle Operatoren komplett identisch. Ein weiterer Unterschied der beiden Implementierungen ist die unterschiedliche Erstellung der Ausgabestrings. Dieser Unterschied ist jedoch irrelevant, da im Vordergrund steht, wie die Operatoren eingesetzt werden können.

4.2. Matches

Die Implementierung der Match-Strukturen ist in dem lecture-Module unter dem Verzeichnis util in der Datei *match.py* als Klasse Match zu finden.

Der Parser erkennt gewisse Operatoren, Funktionen und Klammerungen. Die unterstützten Operatoren sind:

„+“, „-“, „*“, „/“, „%“, „**“, „&&“, „||“, „<“, „>“, „<=“, „>=“, „=>“, „<==>“, „==“, „!=“ und „!“.

Die unterstützten Funktionen sind:

„sin“, „log“, „exp“, „cos“, „tan“, „asin“, „acos“, „atan“, „sqrt“ und „ln“.

Es werden runde öffnende Klammern „(“ und runde schließende Klammern „)“ erkannt.

Die wichtigste Funktion für den Benutzer ist `match(self, scheme, value)`. Da die Funktion auf einem erzeugten Match ausgeführt wird, sind nur die Variablen `scheme` und `value` für den Anwender interessant. Unter `scheme` wird der zu parsende Ausdruck gegeben und `value` enthält

den Wert nach dem gematched werden soll. Wichtig hierbei ist, dass das Matching nur auf Strings basierend ausgeführt werden kann, während Matches in SetlX auch die Verwendung Literals und direkte Operationen auf den Ausgaben ermöglichen. Dieser Unterschied ist bei einem direkten Vergleich im Code sofort erkennbar. Im Nachfolgenden wird ein Match-Konstrukt, das mathematische Funktionen ableiten soll, in SetlX, mit der neuen Struktur, wie sie in Python entwickelt wurde, verglichen.

```
diff := procedure(t, x) {
    match (t) {
        case a + b :
            return diff(a, x) + diff(b, x);
        case a - b :
            return diff(a, x) - diff(b, x);
        case a * b :
            return diff(a, x) * b + a * diff(b, x);
...
```

Abbildung 4: Ausschnitt aus `diff()` (SetlX)

```
def diff(t,x):
    match = Match()
    if match.match('a+b', t):
        return '{diff_a} + {diff_b}'.format(
                diff_a=diff(match.values['a'], x),
                diff_b=diff(match.values['b'], x))
    elif match.match('a-b', t):
        return '{diff_a} - {diff_b}'.format(
                diff_a=diff(match.values['a'], x),
                diff_b=diff(match.values['b'], x))
    elif match.match('a*b', t):
        return '{diff_a} * {b} + {a} * {diff_b}'.format(
                diff_a=diff(match.values['a'], x),
                b=match.values['b'], a=match.values['a'],
                diff_b=diff(match.values['b'], x))
...
```

Abbildung 5: Ausschnitt aus `diff()` (Python)

Was direkt auf den ersten Blick auffällt ist, dass der Code, der in SetlX sehr kompakt dargestellt wird, deutlich umfangreicher ist. Dementsprechend leidet auch die Leserlichkeit unter der Python-Version. Es ist nicht direkt klar, wie der Code zu lesen ist, da die Ausdrücke als Strings abgebildet sein müssen. Während in *Abbildung 4* im return die Ableitregeln, durch rekursive Aufrufe von `diff()`, zu den mathematischen Funktionen im jeweiligen case stehen, sind in *Abbildung 5* dieselben mathematischen Funktionen als Strings im match.match-Teil zu erkennen, allerdings ist nicht sofort ersichtlich was im return-Statement steht. Der String, der zurückgegeben wird enthält dieselben Ableitregeln wie sie im SetlX-Code zu sehen sind, allerdings werden die Variablen nicht direkt genannt, sondern durch Platzhalter dargestellt. In den Parametern der `format`-Funktion werden die Platzhalter gefüllt. Die Platzhalter mit dem Präfix „diff_“ werden rekursiv Abgeleitet, wobei dem erneuten `diff`-Aufruf die Werte, die im Match für die jeweilige Variable hinterlegt sind und das „x“

weil nach x abgeleitet wird, übergeben werden. Wenn ein Platzhalter kein Präfix besitzt, so werden nur die Werte aus dem Match herausgelesen und eingesetzt und nicht zusätzlich evaluiert.

4.3. Übersetzung komplexerer Programme

Es wurden zwar einige SetlX-Programme in Python-Skripte übersetzt, allerdings werden in dieser Arbeit hauptsächlich Programme, die die Eleganz der Programmiersprache SetlX verdeutlichen, genauer betrachtet.

Wie zuvor beschrieben, ermöglicht SetlX dem Programmierer in einem sehr mathematischen Stil zu programmieren. Somit können Personen, die ersten Berührungen mit der Mengenlehre oder von der Mathematik kommen, sowie Studenten, die mathematische Konstrukte verstehen und anwenden müssen, beim Programmieren diese Erfahrungen sammeln.

4.3.1. Schiebepuzzle

Das Schiebepuzzle ist eine Aufgabe die den Studenten mit Lücken als Aufgabe gegeben wird, um Vorlesungsinhalte direkt anwenden zu können. Mit diesem Programm sollen die Studenten eine für Menschen nicht triviale Lösung zu einem Schiebepuzzle berechnen lassen. Aufgrund der Berechnung aller möglichen Pfade, das Puzzle zu lösen, lässt sich das Programm nicht so schnell wie die meisten anderen SetlX-Programme durchführen.

Sowohl das SetlX-Programm, wie auch die Übersetzung in Python definieren zu Beginn die Funktion, mit der aus einem State (einem derzeitigen Zustand des Puzzles, abgelegt in einer Liste) ein String erzeugt werden kann, um eine bessere Visualisierung zu ermöglichen. Bei der Übersetzung ist in dieser Methode nichts großartig Interessantes zu sehen, da in den meisten Zeilen fast eins-zu-eins dasselbe steht. Allerdings ist zu beachten, dass die for-Schleifen in SetlX über Listen von 1-3 iterieren, während in Python dafür eine Range mit den Werten 0-2 verwendet wird. Es wird allerdings die selbe Ausführung erreicht, da Listen-Indizes in SetlX bei 1 anfangen, während Python die 0 als Index verwendet, um das erste Element aufzurufen.

```
findPath := procedure(start, goal, nextStates) {
    count    := 1;
    paths    := { [start] };
    states   := { start };
    explored := {};
    while (states != explored) {
        print("iteration number $count$");
        count += 1;
        explored := states;
        paths    := { l + [s]
                    : l in paths, s in nextStates(l[-1])
                    | !(s in states)
                    };
        states   += { p[-1] : p in paths };
        print("number of states: $#states$");
        if (goal in states) {
            return arb({ l : l in paths | l[-1] == goal });
        }
    }
};
```

Abbildung 6: `findPath` im Schiebepuzzle (SetlX)

```
def find_path(start, goal, next_states):
    count_iteration = 1
    count_states    = 0
    paths           = Set([start])
    states          = Set(start)
    while len(states) != count_states:
        count_states = len(states)
        print('Iteration number %s' % count_iteration)
        count_iteration += 1
        paths = Set(x + [s]
                    for x in paths for s in next_states(x[-1])
                    if not s in states)
        states += Set(p[-1] for p in paths)
        print('Number of states: %s' % len(states))
        if goal in states:
            return Set(l for l in paths if l[-1] == goal).arb()
```

Abbildung 7: `find_path` im Schiebepuzzle (Python)

Die Funktion, mit der letztendlich auch der Pfad vom Start-Zustand zum Ziel-Zustand ermittelt wird, ist in Python die `find_path` Methode. Die Parameter, die übergeben werden, sind identisch zu der SetlX-Implementierung. Die am Anfang der Funktion definierten Variablen weichen vom SetlX-Code um eine Variable ab. Die Variablen `paths` und `states` sind in beiden Versionen zu finden und der Integer `count_iteration` ist in SetlX als `count` zu finden. Allerdings werden zur Prüfung ob neue Zustände hinzukommen unterschiedliche Ansätze verwendet. In der Python-Implementierung keine Menge mit allen entdeckten Zuständen, sondern die Prüfung der Anzahl der verschiedenen Zustände verwendet. Somit vergleicht die äußere while-Schleife zwei Integer, während die Vorlage zwei Mengen vergleicht. Abgesehen davon ist der Ablauf sehr ähnlich. Es wird die derzeitige Iteration

angegeben, daraufhin die neuen Pfade anhand von `next_states` ermittelt und dann alle derzeitigen Zustände der Pfade ermittelt und die Anzahl ausgegeben. Zuletzt wird noch im Falle, dass das Ziel bereits erreicht wurde, ein beliebiger Lösungspfad zurückgegeben.

```
nextStates := procedure(state) {
    directions := { [1, 0], [-1, 0], [0, 1], [0, -1] };
    [row, col] := findBlank(state);
    return { moveDir(state, row, col, [dx, dy])
           : move in directions
           | row + dx in {1, 2, 3} && col + dy in {1, 2, 3}
           };
};
```

Abbildung 8: `nextStates` im Schiebepuzzle (SetlX)

```
def next_states(state):
    directions = [(1, 0), (-1 ,0), (0, 1), (0, -1)]
    (row, col) = find_blank(state)
    ns         = [move_dir(state, row, col, (dx, dy))
                  for (dx, dy) in directions
                  if 0 <= row + dx <= 2 and 0 <= col + dy <= 2]
    return ns
```

Abbildung 9: `next_states` im Schiebepuzzle (Python)

In `find_path` muss, für die Ermittlung der möglichen Pfade, die Funktion `next_states` aufgerufen werden. Diese gibt eine Liste zurück, die alle erreichbaren Zustände vom Zustand `state`, der als Parameter übergeben wird, enthält. In SetlX wird diese Liste als Menge zurückgegeben, allerdings werden Listen in diesem Fall bevorzugt, da sie in Python geläufiger verwendet werden und keine Mengeneigenschaften in dieser Situation benötigt werden. Die Richtungen werden allerdings nicht wie im SetlX-Programm als Menge von Listen, sondern als Liste von Tupeln definiert, da Tupel in Python, so wie Listen in SetlX, unzipped werden können. Eine Änderung, die an der Vorlage unternommen wurde ist, dass die Bewegung von einer Variable `move` auf zwei Variablen `dx` und `dy` aufgeteilt wurden.

```
moveDir := procedure(state, row, col, dir) {
    [dx, dy]  := dir;
    nextState := state;
    nextState[row     ][col     ] := state[row + dx][col + dy];
    nextState[row + dx][col + dy] := 0;
    return nextState;
};
```

Abbildung 10: `movDir` im Schiebepuzzle (SetlX)

```
def move_dir(state, row, col, direction):
    (dx, dy)   = direction
    next_state = [list(x) for x in state]
    next_state[row][col] = next_state[row + dx][col + dy]
    next_state[row + dx][col + dy] = 0
    return next_state
```

Abbildung 11: `mov_dir` im Schiebepuzzle (Python)

Die Methode `move_dir` erhält dieselben Parameter wie in der Vorlage und gibt den nächsten Zustand zurück, nachdem vom derzeitigen Zustand aus eine Bewegung in entweder die x- oder die y-Achse erfolgt. Einer der Unterschiede zur ursprünglichen Umsetzung ist hier, wie zuvor bereits erwähnt wurde, dass die Richtung als Tupel und nicht als Liste gewertet wird.

Ein erheblicher Unterschied zwischen den beiden Implementierungen ist das initiale Setzen der Variable `next_state`. Während in SetlX der Parameter `state` dafür verwendet wird und eine einfache Zuweisung erfolgt, muss in Python die `list()` Funktion auf alle Listen in `state` verwendet werden. Grund dafür ist, dass ansonsten die Referenzen übergeben werden und somit dann die Werte von `state` ebenfalls geändert werden, wenn sie in `next_state` bearbeitet werden. Durch die `list()`-Funktion wird eine Kopie erzeugt und somit sind `next_state` und `state` zwei unabhängige Listen. SetlX erkennt an dieser Stelle intern, ob ein Objekt geändert wurde und erstellt eine Kopie falls nötig. Deshalb wird dieser Aufruf nur in der Python-Version benötigt.

```
findBlank := procedure(state) {
    for (row in [1 .. 3]){
        for (col in [1 .. 3]){
            if (state[row][col] == 0){
                return [row, col];
            }
        }
    }
};
```

Abbildung 12: `findBlank` im Schiebepuzzle (SetlX)

```
def find_blank(state):
    for row in range(3):
        for col in range(3):
            if state[row][col] == 0:
                return (row, col)
```

Abbildung 13: `find_blank` im Schiebepuzzle

Damit in `next_states` eine Zahl „bewegt“ werden kann, muss über `find_blank` das freie Feld gefunden werden. In der Vorlage wird in `findBlank` eine Menge erzeugt, die alle Reihen-Zeilen-Kombinationen enthält und aus dieser einer der Werte, an denen der übergebene Zustand die 0 enthält, zurückgegeben. In dieser Arbeit wird nur ein Tupel mit einem Reihen- und einem Zeilenwert, an denen der Zustand die Null enthält zurückgegeben. Zu Vergleichszwecken ist die Funktion, so wie sie in Python geschrieben wurde, auch in SetlX implementiert. In SetlX merkt man keine Unterschiede in der Performance. Diese Methode zeigt wie ähnlich der Code in SetlX und Python sein kann.

Nachdem die Funktionen alle definiert sind ist der Ablauf komplett identisch zur Vorlage. Es wird die Zeitmessung begonnen, der Start- und End-Zustand definiert, daraufhin der Pfad ermittelt und die Zeitmessung beendet. Abschließend werden der Lösungspfad und die Zeitmessungsergebnisse ausgegeben.

Im Allgemeinen ist der Code in beiden Programmiersprachen sehr ähnlich und der Ablauf, so wie die Syntax, teilweise sogar identisch. Einige Ausdrücke sehen auf Grund der Programmiersprache unterschiedlich aus, erfüllen aber denselben Zweck. Bei der Zeitmessung beispielsweise wird in SetlX nur die Methode `now()` aufgerufen, während in Python `timeit.default_timer()` aufgerufen wird und sogar ein Import dafür notwendig ist. Andere Abweichungen sind unterschiedliche Datentypen, die in Python gewählt wurden. Diese werden verwendet, um die Laufzeit etwas zu verbessern, weil die Implementierung, die für Mengen verwendet werden muss, nicht so effizient wie die SetlX-Implementierung ist. An einigen Stellen sind aber auch keine Mengen notwendig und werden deshalb durch Listen ersetzt.

Die genaue Zeit, wie lange das Programm für die Berechnung gebraucht hat, wird in der Kommandozeile ausgegeben. Somit wird den Studenten klar, dass selbst der Rechner diese Berechnungen nicht sofort liefern kann. Um einen Vergleich der Performance von SetlX zu Python zu haben, wird auch die Berechnungszeit der Python-Implementierung aufgeführt. Die Eigenschaften des Rechners, mit dem die Berechnungen durchgeführt wurden sind:

- Prozessor: Intel i7 6700hq 2,6-3,1 GHz
- Hauptspeicher: 8 GB DDR4

Das SetlX-Programm lief in 17,4 Sekunden, während das Python-Skript 46,6 Sekunden für die Berechnung benötigte. Auffällig ist, dass in SetlX die Ausführung über doppelt so schnell wie bei der Python-Implementierung ist. Grund hierfür ist, dass die virtuelle Maschine, in der Java ausgeführt wird, etwas effizienter als die virtuelle Maschine von Python ist. Da SetlX auf Java basiert, wird die Effizienz von Java zu Python verglichen. Außer dem Unterschied bei den virtuellen Maschinen, unterstützt die Programmiersprache Java zusätzlich eine statische Typisierung. Python hingegen unterstützt, wie es für Skriptsprachen üblich ist, eine dynamische Typisierung, die etwas ineffizienter ist. Eine statische Typisierung ist effizienter da der Rechenaufwand für eine Typüberprüfung wegfällt. Der allgemeine Leistungsunterschied der Sprachen ist unter https://benchmarksgame.alioth.debian.org/u64q/python.html zu sehen. Python schneidet in fast allen Tests deutlich schlechter ab als Java. Eine unterschiedliche Implementierung der für diese Arbeit entwickelten Mengen und der Mengen, die in SetlX verwendet werden, kann nicht die Ursache für diese Abweichungen in der Performanz sein. Die Sets des Python-Moduls „lecture“ wurden basierend auf die Implementierung, wie die in SetlX verwendet wird, umgesetzt.

4.3.2. Watson

Oftmals werden Rechner für einfachere Rechenoperationen verwendet, für die ein Mensch bereits die Überlegungen zur Logik getätigt hat. Für die Entwickler wird es interessant, wenn der Rechner auch komplexe Zusammenhänge erkennen soll. Dieses Umfeld ist als künstliche Intelligenz bekannt und stellt neue Herausforderungen dar.

Die Aufgabe Watson soll anhand von gegebenen Tatsachen den oder die Täter aus drei Verdächtigen herausfinden.

```
evaluate := procedure(f, i) {
    match (f) {
        case true:          return true;
        case false:         return false;
        case ^variable(p): return i[p];
        case !g:            return !evaluate(g, i);
        case g && h:     return  evaluate(g, i) && evaluate(h, i);
        case g || h:     return  evaluate(g, i) || evaluate(h, i);
        case g => h:     return  evaluate(g, i) => evaluate(h, i);
        case g <==> h:   return  evaluate(g, i) == evaluate(h, i);
        default:      abort("syntax error in evaluate($f$, $i$)");
    }
};
```

Abbildung 14: `evaluate` in Watson (SetlX)

```
def evaluate(f, i):
    match = Match()
    if match.is_variable(f):
        return i[f]
    elif match.match('!g', f):
        return not evaluate(match.values['g'], i)
    elif match.match('g && h', f):
        return evaluate(match.values['g'], i) and
               evaluate(match.values['h'], i)
    elif match.match('g || h', f):
        return evaluate(match.values['g'], i) or
               evaluate(match.values['h'], i)
    elif match.match('g => h', f):
        return not(evaluate(match.values['g'], i)) or
                   evaluate(match.values['h'], i)
    elif match.match('g <==> h', f):
        return evaluate(match.values['g'], i) ==
               evaluate(match.values['h'], i)
    else:
        raise SyntaxError('Syntax error in evaluate(%s,%s)' %
                          (f, i))
```

Abbildung 15: `evaluate` in Watson (Python)

Die Methode `evaluate` wird bereits am Anfang des Programms definiert und gibt an, wie verschiedene Ausdrücke ausgewertet werden sollen. Es sind ein paar kleine Unterschiede in beiden Umsetzungen zu erkennen.

In der Python-Implementierung werden `True` und `False` nicht als Ausdrücke behandelt. Das liegt daran, dass die Schlüsselwörter „True“ und „False“ in der Match-Implementierung nicht eingetragen sind. Außerdem hat das Herauslassen dieser beiden Fälle keine Auswirkungen auf die Ausführung des Skripts.

Wenn der Ausdruck `g => h` gelesen wird, wird in SetlX die eingebaute Implikation aufgerufen. Da Python keinen Implikations-Operator besitzt, wird an dieser Stelle die Umschreibung für diesen

Junktor verwendet. Die Implikation von `g` und `h` kann auch als $\neg g \vee h$ geschrieben werden.

```
createValuation := procedure(m, v) {
    return { [ x, x in m ] : x in v };
};
```

Abbildung 16: `createValuation` in Watson (SetlX)

```
def create_valuation(m, v):
    return [(x, x in m) for x in v]
```

Abbildung 17: `create_valuation` in Watson (Python)

Die Methode `createValuation` gibt eine Menge mit Tupeln, die für jedes Element in `v` angeben, ob es sich auch in `m` befindet, zurück. In `create_valuation` wird das gleiche Resultat geliefert, allerdings befinden sich die Tupel in einer Liste. Die Struktur wurde so gewählt, da zu einem späteren Verlauf aus einer solchen Menge/Liste der Wahrheitswert (`x in m`) durch das Aufrufen aus der Liste, anhand der Bezeichnung des Elements abgefragt wird. In Python ist das mit einem Dictionary möglich, jedoch unterstützt Python eine Sortierung von Dictionaries nicht, weshalb diese nicht in die Sets eingefügt werden können. Das heißt dass keine Dictionaries in Sets möglich sind. Die Struktur, Key-Value-Paare (zweistellige Tupel) in einer Liste zu hinterlegen, kann allerdings über den `dict()`-Befehl in ein Dictionary konvertiert werden. Somit wird die Liste zum Abspeichern in einem Set verwendet und sobald die Valuation wieder ausgelesen wird, wird diese als Dictionary weiterverwendet.

Nun werden die Tatsachen, die zu dem Mordfall bekannt sind, als Strings von Bedingungen eingelesen. In SetlX werden diese über den `parse`-Befehl in Variablen gelesen, während in Python die Strings direkt in die Variablen `f1` bis `f6` gespeichert werden.

```
fs := { f1, f2, f3, f4, f5, f6 };
v  := { "a", "b", "c" };
p := 2 ** v;
print("p = ", p);
b  := { createValuation(m, v) : m in p };
s  := { i : i in b | forall (f in fs | evaluate(f, i)) };
print("Set of all valuations satisfying all facts: ", s);
if (#s == 1) {
    i := arb(s);
    offenders := { x : x in v | i[x] };
    print("Set of offenders: ", offenders);
}
```

Abbildung 18: Lösung von Watson (SetlX)

```
fs = [f1, f2, f3, f4, f5, f6]
v = Set('a', 'b', 'c')
p = 2 ** v
print('p = ', p)
b = Set(create_valuation(m, v) for m in p)
s = [dict(i) for i in b if all(evaluate(f, dict(i)) for f in fs)]
print('List of all valuations satisfying all facts: ', s)
if len(s) == 1:
    i = s[0]
    offenders = [x for x in v if i[x]]
    print('List of offenders: ', offenders)
```

Abbildung 19: Lösung von Watson (Python)

Die Tatsachen werden alle in einer Liste/Menge gespeichert und die Täter, als Buchstaben „a“, „b“ und „c“, in einer Menge abgelegt. Daraufhin wird die Potenzmenge der Täter gebildet, in der alle möglichen Täter-Kombinationen enthalten sind und Wahrheitswerte den Variablen zugewiesen. Bei der Ermittlung, welche Täterkombinationen möglich sind, wird in SetlX die Funktion `forall` und in Python `all` verwendet. Beide Funktionen geben nur dann `True` zurück, wenn alle Elemente der angegebenen Liste/Menge ebenfalls den Wahrheitswert `True` besitzen. In Python werden die erzeugten Tupel in Dictionaries gespeichert, um den Aufruf für `evaluate` und die Ausgabe zu erleichtern. Zuletzt wird das Ergebnis noch ausgegeben.

4.3.3. Wolf Ziege Kohl

Bei der Wolf-Ziege-Kohl-Aufgabe besteht die Herausforderung darin, dass ein Bauer einen Wolf, eine Ziege und einen Kohl von einer Seite eines Flusses auf die andere bringen möchte. Das Problem dabei ist, dass er nur ein Element auf einmal transportieren kann. Wenn er den Wolf mit der Ziege alleine lässt, so frisst der Wolf die Ziege und wenn er die Ziege mit dem Kohl alleine lässt so frisst die Ziege den Kohl. Diese Problemstellung sollen die Studenten anhand eines Programms lösen. Die Aufgabe wird auch als wolf-goat-cabbage (wgc) bezeichnet.

```
findPath := procedure(x, y, r) {
    p := { [x] };
    while (true) {
        oldP  := p;
        p     := p + pathProduct(p, r);
        found := { l : l in p | l[-1] == y };
        if (found != {}) {
            return arb(found);
        }
        if (p == oldP) {
            return;
        }
    }
};
```

Abbildung 20: `findPath` in wgc (SetlX)

```
def find_path(x, y, r):
    p = Set([x])
    while True:
        old_p = p
        p     = p + path_product(p, r)
        found = Set(l for l in p if l[-1] == y)
        if found:
            return found.arb()
        if p == old_p:
            return
```

Abbildung 21: `find_path` in wgc (Python)

Die Funktion `find_path` ermittelt, ob es einen Pfad vom Knoten `x` zum Knoten `y` innerhalb von `r` gibt und gibt diesen zurück, sofern er vorhanden ist. In der Variable `p` befinden sich alle möglichen Pfade, die gefunden werden. Pro Schleifendurchgang werden die nächst möglichen Pfadschritte berechnet und geprüft, ob die Lösung sich danach bereits in `p` befindet. Die erste Abbruchbedingung tritt ein, wenn die Lösung gefunden wurde. In dem Fall wird die Lösung zurückgegeben. Die zweite Abbruchbedingung ist, wenn bei der Ermittlung des nächsten Schritts keine neuen Pfade entdeckt wurden. In diesem Fall gibt die Methode keinen Wert zurück.

Die Abläufe und sind in beiden Programmiersprachen nahezu identisch. Die erste Abbruchbedingung wird syntaktisch anders formuliert. Während in SetlX geprüft wird, ob die Menge `found` nicht leer ist (Abbildung 20 Zeile 7), wird in Python geprüft ob `found` Elemente enthält (Abbildung 21 Zeile 7). Diese beiden Aussagen sind logischerweise dieselben.

```
pathProduct := procedure(p, q) {
    return { add(x,y) : x in p, y in q |
             x[-1] == y[1] && noCycle(x,y) };
};
```

Abbildung 22: `pathProduct` in wgc (SetlX)

```
def path_product(p, q):
    return Set(add(x, y) for x in p for y in q
            if x[-1] == y[0] and no_cycle(x, y))
```

Abbildung 23: `path_product` in wgc (Python)

In `path_product` werden die nächsten Schritte, die von `p` aus nach `q` erreichbar sind zurückgegeben.

Wie sich leicht erkennen lässt, sind diese beiden Implementierungen gleich. Das einzige, das zu beachten ist, ist, dass das erste Element einer Liste in SetlX den Index 1 besitzt, während es in Python unter dem Index 0 zu finden ist.

```
noCycle := procedure(l1, l2) {
    return #({ x : x in l1 } * { x : x in l2 }) == 1;
};
```

Abbildung 24: `noCycle` in wgc (SetlX)

```
def no_cycle(l1, l2):
    length = len(Set(x for x in l1) * Set(x for x in l2))
    return length == 1
```

Abbildung 25: `no_cycle` in wgc (Python)

Die Funktion `no_cycle` prüft, ob es zu Zyklen kommen kann, indem geprüft wird wie viele gleiche Elemente in `l1` und `l2` enthalten sind. Nur wenn es nur ein Element, das in beiden Listen enthalten ist, gibt, tritt kein Zyklus auf.

Hier fällt auf, dass es nur syntaktische Unterschiede zwischen den beiden Implementierungen gibt.

```
add := procedure(p, q) {
    return p + [ q[2] ];
};
```

Abbildung 26: `add` in wgc (SetlX)

```
def add(p, q):
    return p + [q[1]]
```

Abbildung 27: `add` in wgc (Python)

Die Rückgabe von `add` ist die Summe der Mengen `p` und `q`, wobei das erste Element von `q` das letzte von `p` sein muss.

An dieser Stelle muss wieder auf die unterschiedliche Indexierung der beiden Sprachen verwiesen werden. Beide Methoden greifen auf das zweite Element von `q` zu, benötigen aber dafür unterschiedliche Indizes.

Wgc ist in drei Bereiche aufgeteilt. Der erste Teil, der aus den eben beschriebenen Funktionen besteht, beinhaltet die Methoden, die zur Problemlösung benötigt werden. Der zweite Teil, der im Nachfolgenden beschrieben wird, beinhaltet den problemspezifischen Code. Der letzte Teil behandelt die visuelle Darstellung der Lösung. Auf diesen Teil wird nicht genauer eingegangen, da nur die Lösung von Problemen in der theoretischen Informatik im Vordergrund steht.

```
problem := procedure(s) {
    return !("farmer" in s) &&
           ("goat" in s && "cabbage" in s ||
            "wolf" in s && "goat" in s);
};
```

Abbildung 28: `problem` in wgc (SetlX)

```
def problem(s):
    return not 'farmer' in s and \
           (('goat' in s and 'cabbage' in s) or
            ('wolf' in s and 'goat' in s))
```

Abbildung 29: `problem` in wgc (Python)

Die Methode `problem` prüft, ob es bei dem Pfad `s` zu Problemen kommen kann. Mit Problemen sind die Restriktionen in der Aufgabenstellung, wie beispielsweise der Wolf und die Ziege dürfen nicht unbeaufsichtigt alleine sein, gemeint.

Hierzu ist nicht viel zu sagen, da beide Implementierungen bis auf die Syntax komplett gleich sind.

```
all := { "farmer", "wolf", "goat", "cabbage" };
p   := { s : s in 2 ** all | !problem(s) && !problem(all - s) };
r1  := { [s, s - b]: s in p, b in 2 ** s
                   | s - b in p && "farmer" in b && #b <= 2
       };
r2  := { [y, x] : [x, y] in r1 };
r   := r1 + r2;

start := all;
goal  := {};

path  := findPath(start, goal, r);
```

Abbildung 30: Lösung des wgc-Problems (SetlX)

```
wgc_all = Set('farmer', 'wolf', 'goat', 'cabbage')
p       = Set(s for s in 2 ** wgc_all if not problem(s)
                        and not problem(wgc_all - s))
r1      = Set([s, s - b] for s in p for b in 2 ** s
                    if (s - b) in p and 'farmer' in b and
                       len(b) <= 2)
r2      = Set([y, x] for [x, y] in r1)

r       = r1 + r2
start   = wgc_all
goal    = Set()
path    = find_path(start, goal, r)
```

Abbildung 31: Lösung des wgc-Problems (Python)

Da nun alle benötigten Funktionen definiert sind, kann das Problem in Angriff genommen werden. Die Menge `wgc_all` wurde anders benannt als in SetlX, da `all()` bereits eine eingebaute Funktion in Python ist. In der Menge befinden sich die Bezeichnungen der Elemente, die in der Problemstellung beschrieben wurden. In `p` werden daraufhin alle möglichen Kombinationen der Elemente in `wgc_all`, die zu keinen Problemen führen, festgehalten. In `r1` findet man alle Möglichkeiten, in denen der Bauer eine Situation in `p` mit einem weiteren Element oder alleine verlässt. Die Menge `r` wird aus einer Vereinigung von `r1` und `r2` (dem Inversen von `r1`) gebildet. Nun kann ein möglicher Lösungspfad mit `find_path()` ermittelt werden, indem als Start `wgc_all`, als Ziel eine leere Menge und als mögliche Pfade `r` übergeben werden.

Im Allgemeinen kann gesagt werden, dass dieses Programm mit Hilfe der Set-Implementierung sehr gut in Python nachkonstruiert werden konnte. Im Python-Skript sind nur wenige Abweichungen auffindbar.

4.3.4. 8 Damen Problem

Das 8 Damen Problem beschäftigt sich mit der Frage, wie acht Damen auf ein Schachbrett gestellt werden können, ohne sich gegenseitig schlagen zu können. Als Hinführung zu diesem Problem wird den Studenten zuvor das Davis-Putnam-Verfahren vorgestellt. Nachdem das Davis-Putnam-Verfahren implementiert wurde, soll anhand dessen das 8 Damen Problem gelöst werden.

```
davisPutnam := procedure(clauses, literals) {
    clauses := saturate(clauses);
    if ({} in clauses) {
        return { {} };
    }
    if (forall (c in clauses | #c == 1)) {
        return clauses;
    }
    l := selectLiteral(clauses, literals);
    notL := negateLiteral(l);
    r := davisPutnam(clauses + {{l}}, literals + {l, notL});
    if (r != { {} }) {
        return r;
    }
    return davisPutnam(clauses + {{notL}}, literals + {l, notL});
};
```

Abbildung 32: `davisPutnam` in Davis Putnam (SetlX)

```
def davis_putnam(clauses, literals):
    s = saturate(clauses)
    if Set() in s:
        return Set(Set())
    if all(len(c) == 1 for c in s):
        return s
    l = select_literal(s, literals)
    not_l = negate_literal(l)
    r = davis_putnam(s + Set(Set(l)), literals + Set(l, not_l))
    if r != Set(Set()):
        return r
    return davis_putnam(s + Set(Set(not_l)), literals +
           Set(l, not_l))
```

Abbildung 33: `davis_putnam` in Davis Putnam (Python)

Zu Beginn der `davis_putnam`-Funktion wird die `saturate`-Methode auf die Menge `clauses` ausgeführt. Das Ergebnis wird in die Variable `s` gespeichert, während in SetlX dafür die Variable `clauses` überschrieben wird. Daraufhin wird geprüft, ob die ermittelte Menge eine leere Menge enthält, da in diesem Fall das Ergebnis ein Falsum wäre. Dadurch würde eine leere Menge in einer leeren Menge zurückgegeben werden. Wenn in `s` nur Unit-Klauseln enthalten sind, so ist das Ergebnis auch bereits ermittelt und es wird `s` zurückgegeben. Auch wenn beide if-Abfragen leicht unterschiedlich aussehen führen sie jedoch die gleiche Abfrage aus. Danach wird ein beliebiges Literal aus `s`, das sich nicht in `literals` befindet gewählt und in einer weiteren Variable negiert. Nun folgt ein rekursiver Aufruf, bei dem `s` mit `l` addiert und `literals` mit sowohl `l` als auch `not_l` addiert als Parameter übergeben werden. Wenn das Ergebnis keine Menge einer leeren

Menge ist, wird es zurückgegeben. Anderen Falls wird das Ergebnis des rekursiven Aufrufs mit `not_l` anstelle von `l` zurückgegeben.

Zwischen den beiden Implementierungen ist kein bedeutender Unterschied zu bemerken. Das Einzige, das bei der Übersetzung bedacht werden muss ist, dass Funktionen wie `len` oder `all` bzw. einzeilige for-Schleifen in beiden Sprachen unterschiedlich dargestellt werden.

```
saturate := procedure(s) {
    units := { k : k in s | #k == 1 };
    used  := {};
    while (units != {}) {
        unit  := arb(units);
        used  += { unit };
        l     := arb(unit);
        s     := reduce(s, l);
        units := { k : k in s | #k == 1 } - used;
    }
    return s;
};
```

Abbildung 34: `saturate` in Davis Putnam (SetlX)

```
def saturate(s):
    units = Set(k for k in s if len(k) == 1)
    used  = Set()
    while len(units) != 0:
        unit  = units.arb()
        used += Set(unit)
        l     = unit.arb()
        s     = _reduce(s, l)
        units = Set(k for k in s if len(k) == 1) - used
    return s
```

Abbildung 35: `saturate` in Davis Putnam (Python)

In `saturate` werden alle Klauseln, die mit Unit-Schnitten aus `s` ableitbar sind zurückgegeben. Hierbei werden die Klauseln, auf denen ein Unit-Schnitt bereits durchgeführt wurde, aus `s` entfernt.

Hier ist auch wieder zu erkennen wie groß die Ähnlichkeit der beiden Implementierungen ist. Bis auf syntaktische Unterschiede, wie beispielsweise der Aufruf von `arb` in Zeile 5 beider Umsetzungen, gibt es keine Unterschiede. Die Methoden `_reduce`, `select_literal`, `is_positive` und `negate_literal` sind ebenfalls nahezu identisch. Die enthaltenen Unterschiede, wie die Match-Syntax, wurden bereits bei vorherigen Code-Schnipseln gezeigt und somit an dieser Stelle nicht wiederholt durchleuchtet. Die Methode `_reduce` hat einen Unterstrich als erstes Zeichen, da es in Python bereits eine Funktion mit der Bezeichnung `reduce` gibt.

Da damit der Davis-Putnam-Algorithmus implementiert ist, kann dieser nun in Queens (8 Damen Problem) verwendet werden.

```
atMostOne := procedure(s) {
    return { { !p, !q } : p in s, q in s | p != q };
};
```

Abbildung 36: `atMostOne` in Queens (SetlX)

```
def at_most_one(s):
    return Set(Set("!" + p, "!" + q) for p in s for q in s
                                     if p != q)
```

Abbildung 37: `at_most_one` in Queens (Python)

Bei der Betrachtung der `at_most_one` Methode fällt auf, dass in Python Strings verwendet werden, während in SetlX eine einfache Negierung stattfindet. Dies liegt daran, dass in SetlX für die Feldpositionen Variablen erzeugt werden, was jedoch in Python nicht möglich ist. Die Lösung mit den Strings erschwert zwar teilweise die Schreibweise, erfüllt jedoch denselben Zweck wie die Lösung in SetlX.

```
atMostOneInRow := procedure(row, n) {
    return atMostOne({ Var(row, column) : column in [1 .. n] });
};
```

Abbildung 38: `atMostOneInRow` in Queens (SetlX)

```
def at_most_one_in_row(row, n):
    return at_most_one(Set("varr%sc%s" % (str(row), str(column))
                       for column in range(n)))
```

Abbildung 39: `at_most_one` in Queens (Python)

In der Funktion `at_most_one_in_row` kann gesehen werden, wie die „Variablen“ in Python dargestellt werden. Das Präfix „var“ steht für Variable, nach dem „r“ kommt die Reihennummer (`row`) und nach dem „c“ die Zeilennummer (`column`). An dieser Stelle wird wieder, wie bereits zuvor, die automatische Generierung einer Liste in SetlX über den `range`-Befehl in Python gelöst.

Die Funktion `one_in_column` ist sehr ähnlich zu `at_most_one_in_row` und gleich aufgebaut. Deshalb wird die Methode an dieser Stelle nicht dargestellt. Anstatt die Rückgabe von `at_most_one` zurückzugeben, wird hier eine Menge zurückgegeben.

```
atMostOneInLowerDiagonal := procedure(k, n) {
    s := { Var(r, c) : c in [1..n], r in [1..n] | r - c == k };
    return atMostOne(s);
};
```

Abbildung 40: `atMostOneInLowerDiagonal` in Queens (SetlX)

```
def at_most_one_in_lower_diagonal(k, n):
    return at_most_one(
        Set(
            "varr%sc%s" % (str(row), str(column))
            for row in range(n)
            for column in range(n)
            if (row+1) - (column+1) == k
        )
    )
```

Abbildung 41: `at_most_one_in_lower_diagonal` in Queens (Python)

In `at_most_one_in_lower_diagonal` lässt sich wieder erkennen, dass, bis auf die Eigenart des Strings als Variable, beide Implementierungen derselben Struktur folgen. Die Implementierungen sind sehr gut gegenüberstellbar.

In `at_most_one_in_upper_diagonal` ist die Umsetzung analog zu der in `at_most_one_in_lower_diagonal`.

```
allClauses := procedure(n) {
    return +/ {atMostOneInRow(row, n)      : row in {1..n}}
     + +/ {atMostOneInLowerDiagonal(k, n) : k in {-(n-2) .. n-2}}
     + +/ {atMostOneInUpperDiagonal(k, n) : k in {3 .. 2*n - 1}}
     + +/ {oneInColumn(column, n)         : column in {1 .. n}};
};
```

Abbildung 42: `allClauses` in Queens (SetlX)

```
def all_clauses(n):
    return Set(at_most_one_in_row(row, n)
               for row in range(n)).sum() + \
           Set(at_most_one_in_lower_diagonal(k, n)
               for k in range(-(n-2), n-2)).sum() + \
           Set(at_most_one_in_upper_diagonal(k, n)
               for k in range(3, 2*n - 1)).sum() + \
           Set(one_in_column(column, n)
               for column in range(n)).sum()
```

Abbildung 43: `all_clauses` in Queens (Python)

Beide Ausführungen von `all_clauses` weisen wieder große Ähnlichkeiten auf. Allerdings wird zur Ermittlung der Summe einer Menge in SetlX der Operator „`+/`“ verwendet, während in Python die Methode `sum` eingesetzt wird.

```
solve := procedure(n) {
    clauses  := allClauses(n);
    print(clauses);
    solution := davisPutnam(clauses, {});
    if (solution != { {} }) {
        printBoard(solution, n);
    } else {
        print("The problem is not solvable for $n$ queens!");
        print("Try to increase the number of queens.");
    }
};
```

Abbildung 44: `solve` in Queens (SetlX)

```
def solve(n):
    clauses = all_clauses(n)
    print(clauses)
    solution = davis_putnam(clauses, Set())
    if solution != Set(Set()):
        print_board(solution, n)
    else:
        print("The problem is not solvable for %s queens!"
              % str(n))
        print("Try to increase the number of queens.")
```

Abbildung 45: `solve` in Queens (Python)

Da nun alle benötigten Methoden definiert wurden, kann die Lösung ermittelt werden. Zuerst werden alle Klauseln mit dem `all_clauses`-Befehl erstellt. Diese werden daraufhin ausgegeben und dann das Problem anhand von `davis_putnam` gelöst. Zuletzt wird, sofern ein Ergebnis vorhanden ist, dieses Ausgegeben, oder eine Fehlermeldung ausgegeben.

Die Zeit für *queens.py* wird, wie beim Schiebepuzzle, gemessen. Die nachfolgenden Messwerte wurden mit demselben Rechner, wie er in *4.3.1* beschrieben wurde, ermittelt. Während das SetlX-Programm in ca. 500-600 Millisekunden durchläuft, werden beim Python-Skript Zeiten von 1,7-6 Sekunden gemessen. Prüfungen mit einem Profiler ergaben, dass vergleichbar viele Aufrufe von Methoden in beiden Sprachen getätigt werden. Hier scheint erneut die Java Virtual Machine, welche in SetlX genutzt wird, der Python Virtual Machine weit überlegen.

5. Fazit

Prinzipiell kann gesagt werden, dass in den meisten Fällen die SetlX-Programme, mit Hilfe des lecture-Moduls, in Python ähnlich programmiert werden können. Die Syntax beider Sprachen unterscheidet sich stellenweise und manchmal muss ein Ausdruck nach einem anderen Schema aufgebaut werden. Jedoch bieten beide Sprachen für viele Aufgaben geeignete Methoden die eine einfache Problemlösung ermöglichen.

Eine Umstellung der Programmiersprache in der Vorlesung wäre durchaus möglich. Die Performance-Differenzen, die in zwei der umgesetzten Implementierungen aufgetreten sind, sind zwar ärgerlich, jedoch befinden sich die Laufzeiten noch in akzeptablen Bereichen. Der Wechsel der Programmiersprache würde einer der Intentionen von SetlX, den Studenten eine Sprache näherbringen, die eine sehr ähnliche Syntax wie die Sprachen Java und C besitzt, nicht mehr verfolgen. Allerdings wäre Python eine stärker verbreitete Programmiersprache und die Studenten würden somit auch die möglichen Unterschiede zwischen Programmiersprachen erkennen. Das wäre auch eine gute Vorbereitung auf die Praxis, da dort häufig eine Vielzahl an verschiedenen Programmiersprachen Einsatz findet. Ein weiteres Problem wäre, dass die sehr mathematische Notation von SetlX nicht mehr vorhanden wäre. In Python werden häufig Wörter als Operatoren anstatt mathematischen Operatoren verwendet. Im Gegenzug dazu könnten sich Anfänger, die noch nie Kontakt mit einer Programmiersprache den Code leichter lesen, da anfangs weniger mathematische Operatoren gelernt werden müssen um den Code verstehen zu können.

Literaturverzeichnis

Guo, P. (2014, Juli 7). *Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities | blog@CACM | Communications of the ACM*. Retrieved from Communications of the ACM: http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext

Stroetmann, K., & Herrmann, T. (2015, Juli 30). SetlX - A Tutorial. Mannheim, Stuttgart, Baden-Württemberg, Deutschland.