# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 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. [SH15] 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.

# Kapitel 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. [Guo14] Mit 69% ist Python somit mit einer eindeutigen Mehrheit deutlich 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` (Stand 09.05.2016)
[2] `http://www.codingdojo.com/blog/9-most-in-demand-programming-languages-of-2016/` (Stand 09.05.2016)
[3] Stand 09.05.2016

# Kapitel 3

# Skripte ohne spezielle Module

Trotz dessen, dass die Erstellung des Python Moduls als Hauptbestandteil dieser Arbeit gesehen wird, ist es durchaus möglich einige SetlX Programme ohne zusätzliche, nicht enthaltene Module anzufertigen. Diese Skripte wurden als erstes angefertigt, um feststellen zu können, ob es möglich ist Python Syntax zu verwenden, ohne die Eleganz des Codes zu verlieren. Einer der Ziele der Übersetzung ist die Eleganz der Programme beizubehalten.

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 eins-zu-eins so in Python abbilden. Das originale SetlX Programm verwendet hierfür eine Menge, die die Zahlen von 1 bis zur eingegebenen Zahl n enthält. Daraufhin wird die Summe aller in der Menge enthaltenen Zahlen mit dem „+/“- Operator ermittelt und ausgegeben. In Python wurde fast

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

Abbildung 3.1: Einfache Summenberechnung in SetlX

dasselbe Verhalten nachgebildet. Jedoch wurde anstatt eine Menge anzufertigen eine Range der Zahlen von 0 bis n angelegt. Die Summe wird über die in Python bereits integrierte Funktion `sum()` berechnet und daraufhin ausgegeben. Allgemein kann gesagt werden, dass ein SetlX-Programm,

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

Abbildung 3.2: Einfache Summenberechnung in Python

ohne spezielle Funktionen oder Strukturen die nicht in Python wiedergefunden werden, meist eine große Ähnlichkeit mit der Python- Implementierung hat.

# Kapitel 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, wobei einige wichtige Datentypen nicht unterstützt werden. 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 nur hashbare Werte in einer normalen Menge hinterlegt werden dürfen. Somit ist es beispielsweise verboten normale Mengen in Mengen zu hinterlegen. Diese Strukturen finden in den SetlX Programmen, die in diversen Vorlesungen gezeigt werden, öfter Anwendung, jedoch gibt es in Python den Fehler, der in Abbildung 4.1 zu sehen ist. Zwar können anstelle der „normalen“ sets frozensets verwendet werden, allerdings können die Elemente in der Menge nicht geändert werden, weil die Mengen, wie der Name es bereits impliziert, eingefroren sind. Die Unveränderbarkeit der Mengenelemente ist von Python bewusst gewählt. 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 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 der SetlX-Programme keine besonderen Eigenschaften der Mengen verwendet 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. Eine wichtige Anforderung, die Erhaltung der Eleganz konnte nicht immer erfüllt werden. Somit war dies keine Lösung, die so für alles verwendet werden kann. Die Funktionen, die implementiert werden mussten waren das Entfernen von Duplikaten aus einer Liste, so wie die Ermittlung der Differenz zweier Listen und die Ermittlung der Potenzmenge (wobei jedoch eigentlich die „Potenzliste“ ermittelt wurde). Allerdings sind die Mengen nicht der einzige

```
1  >Traceback (most recent call last):
2  >  File "<stdin>", line 1, in <module>
3  >TypeError: unhashable type: 'set'
4
```

Abbildung 4.1: Fehler bei Mengen in Mengen

Grund, warum das Modul 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, welches auch nicht in Python enthalten ist. 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 zur Generierung des Inversen oder das Erstellen einer sortierten Liste aus einer Menge verwendet. [SH15]

Die interessanteste Anwendung von Matches ist jedoch, wenn Terme verwendet werden. Die

> [...]Art von Matchen [in SetlX] ist ähnlich zum Matching das in den Programmiersprachen Prolog und ML gegeben ist.

[SH15] 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 und die beliebige Reihenfolge wird verwendet. 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 wurden eigene Mengen implementiert, die alle notwendigen Aufgaben erfüllen können.

Als Grundlage für die 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 das Objekt sortiert werden sollen. Unter dem Schlüssel einer SortedListWithKey wird diese Eigenschaft hinterlegt und kann auch dort eingesehen werden. In diesem Fall sind die sortierten Listen, die die Elemente der Sets enthalten, als Key eingetragen.

In der Implementierung der Mengen werden für viele Operationen Operatoren verwendet, wobei versucht wurde möglichst nahe der SetlX-Implementierung 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 4.1 zu sehen. 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-`, beziehungsweise die `_put`-Funktion. Diese sorgt dafür, dass keine Duplikate in ein Set eingetragen werden können. Die `put`-Funktion ruft nur die `_put`-Funktion auf und ist somit keine neue Methode. In Tabelle 4.2 sind die restlichen Mengen-Funktionen aufgelistet. 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. In den Beispielen, die in Abbildung 4.2 und in Abbildung 4.3 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, der 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 unterschied-

| 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 4.1: Mathematische Operatoren für Sets

| Funktion | Bedeutung |
|---|---|
| `str` | Stellt die Menge als String dar |
| `contains` | Dasselbe wie in (siehe Tabelle 4.1) |
| `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 |
| `put/_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 4.2: Funktionen für Sets

```
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 4.2: 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 4.3: Nutzung von Mengen in simple.py

liche 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 *parser.py* als Klasse mit dem Titel MatchParser zu finden. Für diese Klasse ist es wichtig, dass, die auch im Modul util befindlichen Klassen, TokenType und Scanner, so wie

die Hilfsfunktion `is_number` benötigt werden. TokenType enthält die IDs für die verschiedenen Token-Arten die auftreten können, Scanner erstellt aus einem String eine Liste von Tokens und `is_number` überprüft ob eine Zahl an die Funktion übergeben wurde.

Der Parser erkennt gewisse Operatoren, Funktionen und Klammerungen. Die unterstützen Operatoren sind „`+`“, „`-`“, „`*`“, „`/`“, „`%`“, „`**`“, „`&&`“, „`||`“, „`<`“, „`>`“, „`<=`“, „`>=`“, „`=>`“, „`<==>`“, „`==`“, „`!=`“ und „`!`“. Die unterstützten Funktionen sind „`sin`“, „`log`“, „`exp`“, „`cos`“, „`tan`“, „`asin`“, „`acos`“, „`atan`“, „`sqrt`“ und „`ln`“. Die erkannte Klammerung besteht nur aus der runden öffnenden Klammer „`(`“ und der runden schließenden Klammer „`)`“.

Die wichtigste Funktion für den Benutzer ist `match(self, scheme, value)`. Da die Funktion auf einem erzeugten MatchParser 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 nur nach Strings gematched wird, während Matches in SetlX auch Operatoren und Variablen erkennen. 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. Was direkt auf den ersten Blick auffällt ist, dass der

```
1  diff := procedure(t, x) {
2      match (t) {
3          case a + b :
4              return diff(a, x) + diff(b, x);
5          case a - b :
6              return diff(a, x) - diff(b, x);
7          case a * b :
8              return diff(a, x) * b + a * diff(b, x);
9  ...
```

Abbildung 4.4: Ausschnitt aus `diff()` (SetlX)

```
1  def diff(t,x):
2      match = Match()
3      if match.match('a+b', t):
4          return '{diff_a} + {diff_b}'.format(
5                  diff_a=diff(match.values['a'], x),
6                  diff_b=diff(match.values['b'], x))
7      elif match.match('a-b', t):
8          return '{diff_a} - {diff_b}'.format(
9                  diff_a=diff(match.values['a'], x),
10                 diff_b=diff(match.values['b'], x))
11     elif match.match('a*b', t):
12         return '{diff_a} * {b} + {a} * {diff_b}'.format(
13                 diff_a=diff(match.values['a'], x),
14                 b=match.values['b'], a=match.values['a'],
15                 diff_b=diff(match.values['b'], x))
16 ...
```

Abbildung 4.5: Ausschnitt aus `diff()` (Python)

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 4.4 im return die Ableitregeln, durch rekursive Aufrufe von `diff()`, zu den mathematischen Funktionen im jeweiligen case stehen, sind in Abbildung 4.5 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.

## 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. 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. 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 Menge-

```
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 4.6: `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 4.7: `find_path` im Schiebepuzzle (Python)

neigenschaften 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 (Abbildung 4.8 Zeile 4) ist, dass die Bewegung von einer Variable `move` auf zwei Variablen `dx` und `dy` aufgeteilt wurden. 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

```
1  nextStates := procedure(state) {
2      directions := { [1, 0], [-1, 0], [0, 1], [0, -1] };
3      [row, col] := findBlank(state);
4      return { moveDir(state, row, col, [dx, dy])
5             : move in directions
6             | row + dx in {1, 2, 3} && col + dy in {1, 2, 3}
7             };
8  };
```

Abbildung 4.8: `nextStates` im Schiebepuzzle (SetlX)

```
1  nextStates := procedure(state) {
2      directions := { [1, 0], [-1, 0], [0, 1], [0, -1] };
3      [row, col] := findBlank(state);
4      return { moveDir(state, row, col, [dx, dy])
5             : move in directions
6             | row + dx in {1, 2, 3} && col + dy in {1, 2, 3}
7             };
8  };
```

Abbildung 4.9: `next_states` im Schiebepuzzle (Python)

```
1  moveDir := procedure(state, row, col, dir) {
2      [dx, dy]  := dir;
3      nextState := state;
4      nextState[row     ][col     ] := state[row + dx][col + dy];
5      nextState[row + dx][col + dy] := 0;
6      return nextState;
7  };
```

Abbildung 4.10: `moveDir` im Schiebepuzzle (SetlX)

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

Abbildung 4.11: `move_dir` im Schiebepuzzle (Python)

als Liste gewertet wird. (Siehe Abbildung 4.11 Zeile 2)

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 (Abbildung 4.11 Zeile 3). 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 gegebenenfalls eine Kopie. Deshalb wird dieser Aufruf nur in der Python-Version benötigt. Damit in `next_states` eine Zahl „bewegt“ werden kann, muss über `find_blank` das

```
1  findBlank := procedure(state) {
2      for (row in [1 .. 3]){
3          for (col in [1 .. 3]){
4              if (state[row][col] == 0){
5                  return [row, col];
6              }
7          }
8      }
9  };
```

Abbildung 4.12: `findBlank` im Schiebepuzzle (SetlX)

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

Abbildung 4.13: `find_blank` im Schiebepuzzle (Python)

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 können.

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 RAM

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 einen Mordfall lösen. Die Methode

```
1  createValuation := procedure(m, v) {
2      return { [ x, x in m ] : x in v };
3  };
```

Abbildung 4.14: `createValuation` in Watson (SetlX)

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

Abbildung 4.15: `create_valuation` in Watson (Python)

`createValuation` gibt eine Menge mit Tupel, die für jedes Element in `v` angeben, ob es sich auch in `m` befindet. 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 unterstützen die im lecture-Modul implementierten Sets die Sortierung von Dictionaries nicht. Das heißt dass keine Dictionaries in Sets möglich sind. Die Struktur, Paare (zweistellige Tupel) in Listen zu hinterlegen, können 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 weiter verwendet.

### 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. Die Funktion `find_path` ermittelt,

```
1  findPath := procedure(x, y, r) {
2      p := { [x] };
3      while (true) {
4          oldP  := p;
5          p     := p + pathProduct(p, r);
6          found := { l : l in p | l[-1] == y };
7          if (found != {}) {
8              return arb(found);
9          }
10         if (p == oldP) {
11             return;
12         }
13     }
14 };
```

Abbildung 4.16: `findPath` in wgc (SetlX)

```
1  def find_path(x, y, r):
2      p = Set([x])
3      while True:
4          old_p = p
5          p     = p + path_product(p, r)
6          found = Set(l for l in p if l[-1] == y)
7          if found:
8              return found[0]
9          if p == old_p:
10             return
```

Abbildung 4.17: `find_path` in wgc (Python)

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 mögliche 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 4.16 Zeile 7), wird in Python geprüft ob `found` Elemente enthält (Abbildung 4.17 Zeile 7). Diese beiden Aussagen sind logischerweise dieselben. In der nächsten Zeile wird daraufhin in SetlX ein beliebiges Element, während die Python-Implementierung immer

das erste Element der Menge zurückgibt. Es wäre auch möglich die, in den Python Sets implementierte, Funktion `arb()` aufzurufen, allerdings gibt diese auch nur das erste Element zurück. Um das Skript einfach und übersichtlich zu halten, wurde das direkt Aufrufen des ersten Elements bevorzugt. In `path_product` werden die nächsten Schritte, die von `p` aus nach `q` erreichbar sind

```
1  pathProduct := procedure(p, q) {
2      return { add(x,y) : x in p, y in q |
3               x[-1] == y[1] && noCycle(x,y) };
4  };
```

Abbildung 4.18: `pathProduct` in wgc (SetlX)

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

Abbildung 4.19: `path_product` in wgc (Python)

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. Die Funktion `no_cycle` prüft, ob es zu Zyklen kommen

```
1  noCycle := procedure(l1, l2) {
2      return #({ x : x in l1 } * { x : x in l2 }) == 1;
3  };
```

Abbildung 4.20: `noCycle` in wgc (SetlX)

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

Abbildung 4.21: `no_cycle` in wgc (Python)

kann, indem geprüft 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. 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

```
1  add := procedure(p, q) {
2      return p + [ q[2] ];
3  };
```

Abbildung 4.22: `add` in wgc (SetlX)

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

Abbildung 4.23: `add` in wgc (Python)

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. Die

```
1  problem := procedure(s) {
2      return !("farmer" in s) &&
3             ("goat" in s && "cabbage" in s ||
4              "wolf" in s && "goat" in s);
5  };
```

Abbildung 4.24: `problem` in wgc (SetlX)

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

Abbildung 4.25: `problem` in wgc (Python)

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. 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.

```
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 4.26: 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 4.27: Lösung des wgc-Problems (Python)

### 4.3.4 8 Damen Problem

# Literaturverzeichnis

[Guo14] Philip Guo. Python is now the most popular introductory teaching language at top u.s. universities, 07.07.2014.

[SH15] Karl Stroetmann and Tom Herrmann. Setlx - a tutorial: Version 2.4.0. Tutorial for programming language, 30.07.2015.