
Fachbereich Informatik und Mathematik
Institut für Informatik

# Masterarbeit

## Automatisches Lösen von Logikrätseln unter Verwendung von SAT-Solvern mit einer intelligenten Benutzungsschnittstelle

**Aybike Demirsan**

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz / Softwaretechnologie


Fachbereich Informatik und Mathematik
Institut für Informatik

**Masterarbeit**

# Automatisches Lösen von Logikrätseln unter Verwendung von SAT-Solvern mit einer intelligenten Benutzungsschnittstelle

**Aybike Demirsan**

29. Mai 2015

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz

Selbstständigkeitserklärung gemäß §24 Abschnitt 12 der Masterordnung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt, den 29. Mai 2015

Aybike Demirsan

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Kapitel 1

# Einleitung

In den letzten zwei Jahrzehnten, seit der digitalen Revolution, hat die Rolle der Informatik stetig zugenommen[1]. Anforderungen an Software, Benutzerfreundlichkeit und Massentauglichkeit sind gestiegen, da viele Softwareprodukte nicht nur für Entwickler, sondern für Privatanwender konzipiert werden.
Gerade im Bereich der Produktkonfiguration ist dies durch prozedurale Programmierung, die auch als „klassische" oder „traditionelle" Art der Programmierung bezeichnet wird, nur umständlich umsetzbar.
Dadurch gewann das SAT-Problem der Komplexitätstheorie, einem Zweig der theoretischen Informatik, immer mehr an Bedeutung. Durch Darstellung der Produktkonfiguration als eine Instanz des SAT-Problems gewinnt die Implementierung an Schnelligkeit bezüglich der Laufzeit und an Flexibilität. Das bedeutet, dass beispielsweise bei Pkw-Produktkonfigurationen auch nach der Auswahl bestimmter Kriterien wie Leistung des Motors, Größe der Felgen oder Farbe der Innenausstattung noch problemlos Änderungen vorgenommen werden können, was bei prozeduraler Programmierung oft zu stundenlangen Berechnungen bis hin zum Abbruch des Programms führen kann.

Aber nicht nur für Privatanwender haben SAT-Solver erhebliche Vorzüge: Auch in der Entwicklung logischer Schaltungen finden SAT-Solver regelmäßig Einsatz. Hier erleichtern Eingaben der Form „wenn Option A, dann nicht Option B" erheblich die Verifikation und den Entwurf der Schaltungen. Außerdem gewannen SAT-Solver erheblich an Popularität, da diese in den letzten Jahren immer schneller geworden sind und für viele praktisch relevante Instanzen trotz der NP-Vollständigkeit des SAT-Problems schnell Lösungen finden.[2]

Das Ziel dieser Arbeit ist die Entwicklung eines Programms, das verwendet werden kann, um Logikrätsel automatisch zu lösen. Das eigentliche Finden der Lösung soll durch moderne SAT-Solver erfolgen. Zusätzlich soll das Programm auch für Privatanwender relativ einfach benutzbar sein.

[1] Siehe [Wika].
[2] Absatz vgl. [Lan12].

Daher ist für den Programmentwurf nicht nur die Fähigkeit des SAT-Solvers, sondern auch die Benutzungsschnittstelle für die Kommunikation mit dem SAT-Solver von Relevanz.

## 1.1 Motivation

Die Motivation dieser Arbeit liegt daher in der Kombination der Benutzung von SAT-Solvern und der Erstellung einer Benutzungsschnittstelle, um den Zugang für Privatanwender zu erleichtern.

Allerdings lassen sich Logikrätsel oft nicht einfach in Aussagenlogik oder gar direkt als aussagenlogische Klauselmenge ausdrücken.

Daher wird im Rahmen dieser Arbeit eine neue Logik entworfen, die für viele Logikrätsel passend erscheint. Diese Logik wird als Zwischenschicht der eigentlichen Benutzereingabe und dem Backend (dem SAT-Solver) verwendet. Der Vorteil einer solchen Zwischenschicht besteht darin, dass wir die Logik formal mit den Methoden und Techniken der Informatik behandeln können.

Tatsächlich werden wir die Syntax und Semantik der Logik präzise formulieren und analysieren. Formeln dieser Logik werden schließlich in die Aussagenlogik übersetzt und vom SAT-Solver gelöst.

Da Syntax und Semantik beider Logiken (Zwischenschicht und Aussagenlogik) mathematisch einwandfrei definiert sind, können wir die Korrektheit der entsprechenden Übersetzungen nachweisen, und daher die Korrektheit unseres Ansatzes garantieren.

Auch die Zwischenschicht eignet sich eher nicht als direkte Eingabeschnittstelle für Benutzer, da jene hierfür die formale Sprache der Mathematik beherrschen müssen. Daher werden wir versuchen, eine Eingabeschnittstelle zu entwerfen, die Eingaben verarbeiten kann, die näher an der natürlichen Sprache liegen. Die entsprechende Verarbeitung einer solchen Eingabe, ihre Erkennung und ihre Übersetzung in die Zwischenschicht ist ein wohlbekanntes Problem der Informatik, welches mit den Methoden und Techniken der lexikalischen und syntaktischen Analyse der Informatik gelöst werden kann.

Die Zielgruppe von Logikrätseln sind im Allgemeinen keine Informatiker, die Lösungsprogramme schreiben, sondern Leute, die Spaß am Lösen solcher Rätsel ohne maschinelle Hilfsmittel haben.
Die Motivation, Logikrätsel mit einem Programm zu lösen, liegt aber nicht darin, den Spaß am Knobeln zu nehmen, sondern verfolgt andere Ziele:

- Zum manuellen Lösen der Rätsel muss der Mensch Intelligenz aufwenden. Das große Forschungsgebiet der Künstlichen Intelligenz versucht die menschliche Intelligenz maschinell nachzuahmen bzw. maschinelle Intelligenz (u.U. mit nichtmenschlichen Methoden) zu entwickeln.
  Unser Ansatz leistet hierbei einen kleinen Beitrag für dieses Gebiet, und untersucht, inwieweit mit der Methode des SAT-Lösens ein intelligentes Softwaresystem entwickelt werden kann, welches die „Intelligenz des Lösens von Logikrätseln" beherrscht.
- Das Programm kann auch für Ersteller von Logikrätseln von Vorteil sein: Es kann unter anderem dazu verwendet werden, die Lösbarkeit von Logikrätseln zu überprüfen. Der Ersteller eines Logikrätsels kann daher mithilfe des Softwarewerkzeugs überprüfen, ob das entworfene Rätsel tatsächlich lösbar ist.
- Da durch einen SAT-Solver sämtliche Lösungen - d.h. alle Modelle einer Formel - berechnet werden können, wird unser Softwaresystem ebenfalls alle Lösungen eines Logikrätsels berechnen. Dies ist ebenfalls sehr hilfreich beim Entwurf von Logikrätseln, wie sie z.B. in Zeitschriften abgedruckt werden, da unser Programm verwendet werden kann, um die Eindeutigkeit der Lösung zu überprüfen.

Somit lassen sich zwei Ziele dieser Arbeit zusammenfassen: Zum einen ein Programm zu implementieren, das Logikrätsel korrekt löst (siehe Kapitel 4), und zum anderen eine Benutzungsschnittstelle zu erstellen, die den Umgang mit dem Programm erleichtert (siehe Kapitel 5).

## 1.2 Aufbau

Ziel dieser Arbeit ist es, durch Implementierung einer Benutzungsschnittstelle die Eingabe eines Problems, in unserem Fall Logikrätsel, zu erleichtern und das Problem mithilfe eines SAT-Solvers, der auf dem DPLL-Algorithmus beruht, zu lösen.

In Kapitel 2 werden alle Themen, die im Rahmen dieser Arbeit relevant sind, einführend vorgestellt.
Dafür wird zunächst in Kapitel 2.1 erklärt, was man unter Logikrätseln versteht und in 2.1.1 ein Rätsel vorgestellt, das das von mir implementierte Programm später lösen wird.
Anschließend werden im Abschnitt 2.2 in die Aussagenlogik eingeführt, beginnend mit Syntax und Grammatik der Aussagenlogik, gefolgt von der Semantik, sowie Deduktionstheoreme wie logische Folgerung, syntaktische Ableitung und logische Äquivalenz erläutert. Zuletzt betrachten wir einige Beispielsätze in Unterkapitel 2.2.5, um eine bessere Vorstellung der Aussagenlogik zu ermöglichen.
In Unterkapitel 2.3 wird erweiternd zur Aussagenlogik die Prädikatenlogik vorgestellt, wobei zunächst wieder die Syntax und Grammatik dieser Logik in 2.3.1

beschrieben und dann die einzelnen Bestandteile erklärt werden. Wieder schließt das Unterkapitel dann mit Beispielen der Logik - diesmal Prädikatenlogik - in 2.3.3 ab.
Als letztes Themengebiet des Kapitels Grundlagen wird das SAT-Problem in Abschnitt 2.4 vorgestellt und darin die Funktionsweise von SAT-Solvern anhand des DPLL-Algorithmus erläutert und Einsatzbereiche von SAT-Solvern aufgezeigt.

In Kapitel 3 werden die verschiedenen Transformationen, die in meinem Programm implementiert wurden, erläutert. Außen vor ist hier die intelligente Benutzungsschnittstelle, auf die wir in Kapitel 5 eingehen, da sie separat mithilfe eines Parsergenerators erstellt wurde.
Dafür stellen wir in 3.1 die Syntax und Grammatik der eingeschränkten Prädikatenlogik vor, in die die Eingabe des Programms umgewandelt wird.
Anschließend wird in Abschnitt 3.2 und 3.3 erläutert, wie die rätsellogische Formel in eine aussagenlogische Formel und diese in eine Klauselmenge transformiert wird.
Abschließend wird in Abschnitt 3.4 erläutert, wie das Ergebnis des SAT-Solvers, dem wir die Klauselmenge übergaben, wieder in Rätsellogik zurück transformiert wird und betrachten eine Beispiel-Rückrechnung.

Die wichtigsten Teile meiner Implementierung werden dann in Kapitel 4 hervorgehoben. Dafür werden die importierten Module in Abschnitt 4.1 und die Datentypen in Abschnitt 4.2 vorgestellt.
Darauf folgt der Code der tatsächlichen Transformationen von Rätsellogik in Aussagenlogik in Abschnitt 4.4, die Implementierung des Lösens des aussagenlogischen Erfüllbarkeitsproblems in Abschnitt 4.5 und der Code der Rückrechnung des Ergebnisses in Rätsellogik in Abschnitt 4.6.
Wir schließen das Kapitel mit einigen beispielhaften Testfällen ab, um die Kodierung der Testfälle zu verstehen.

In Kapitel 5 wird dann die von mir erstellte intelligente Benutzungsschnittstelle vorgestellt.
In Abschnitt 5.1 werden zunächst Parser und Parsergeneratoren erklärt.
Anschließend betrachten wir die Implementierung der Parsers in Abschnitt 5.2 und betrachten abschließend noch ein paar Tests des Parsers.

In Kapitel 6 werden dem Programm Rätsel, unter anderem das Rätsel aus 2.1.1, als Eingabe übergeben und die Ergebnisse der Rätsel untersucht, ebenso wie die Laufzeit und der Speicherplatzverbrauch der einzelnen Testfälle.
Dann wird in Abschnitt 6.2 ein Zwischenfazit gezogen, was wir über die Tests aussagen können bezüglich erwarteter und tatsächlicher Performanz.
In Abschnitt 6.3 testen wir dann die intelligente Benutzungsschnittstelle und betrachten auch hier die Performanz.

Abschließend fassen wir in Kapitel 7 die Arbeit zusammen, ziehen ein Fazit in

Abschnitt 7.2 und eröffnen einen Ausblick für zukünftige Arbeiten in Abschnitt 7.3.

Die Implementierung die im Rahmen dieser Arbeit erstellt wurde ist unter

www.ki.informatik.uni-frankfurt.de/master/programme/logicals

verfügbar.

# Kapitel 2

# Grundlagen

Diese Arbeit beschäftigt sich mit dem automatischen Lösen von Logikrätseln unter Verwendung von SAT-Solvern mit einer intelligenten Benutzungsschnittstelle. Dafür bedarf es einer Einführung in die Grundlagen dieser Thematik, wofür wir in diesem Kapitel zuerst Logikrätsel in Abschnitt 2.1 einführen und ihren Aufbau erläutern.
Dann werden die beiden Logiken vorgestellt, in die wir die Logikrätsel umschreiben, nämlich die Aussagenlogik in Abschnitt 2.2 und die Prädikatenlogik in Abschnitt 2.3.
Zum Schluss werden wir das SAT-Problem in Abschnitt 2.4 sowie SAT-Solver in Abschnitt 2.4.1 im Allgemeinen betrachten, da die Logikrätsel in eine Instanz des SAT-Problems transformiert und vom SAT-Solver gelöst werden. Letztendlich gehen wir auf den speziell in meiner Arbeit verwendeten DPLL-SAT-Solver in Abschnitt 2.4.1 ein.

## 2.1 Logikrätsel

Logikrätsel sind eine besondere Form von Rätseln, die man mithilfe logischen Schlussfolgerns, der Deduktion, löst.
Das Rätsel besteht aus einer Beschreibung, in der die Elemente oder Objekte vorgegeben werden, sowie einigen Hinweisen, die Aussagen darüber enthalten, welche Objekte miteinander in Verbindung stehen. [1]

Größere Bekanntheit erlangten diese Rätsel aus der Unterhaltungsmathematik in der Wochenzeitung *Die Zeit* unter dem Begriff „Logeleien".

[1] Vgl. [Wikd].

### Mathematische Denkaufgaben

Es gibt verschiedene Sorten von Logikrätseln, manche sind nur kurze mathematische Denkaufgaben:

*Wenn drei Hühner in drei Tagen drei Eier legen - wie viele Hühner legen dann in neun Tagen neun Eier?* [2]

Dabei kann man durch Aufstellen einer mathematischen Gleichung oder durch Nachdenken auf das Ergebnis kommen, dass drei Hühner in neun Tagen neun Eier legen.

### Kreuzzahlenrätsel

Andere bestehen aus *Kreuzzahlenrätseln*, auch Kakuro genannt. Sie ähneln Kreuzworträtseln in ihrem Aufbau, nur, dass Ziffern anstelle von Buchstaben eingetragen werden müssen. Es dürfen nur die Ziffern von 1 bis 9 vorkommen, jede Zeile und jede Spalte muss der danebenstehenden Summe entsprechen und in jeder Summe darf jede Ziffer nur einmal vorkommen.

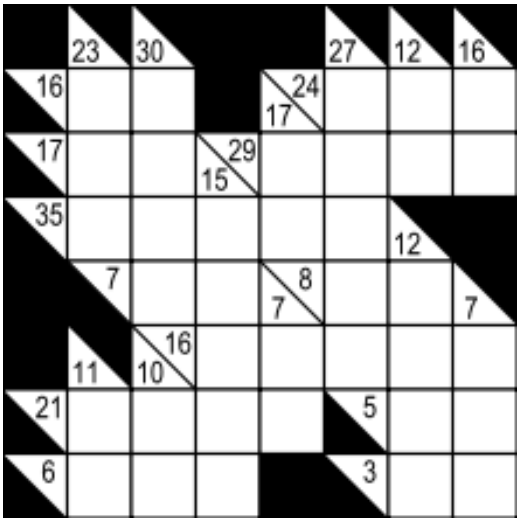

Abbildung 2.1: Kreuzzahlenrätsel [Wikc]

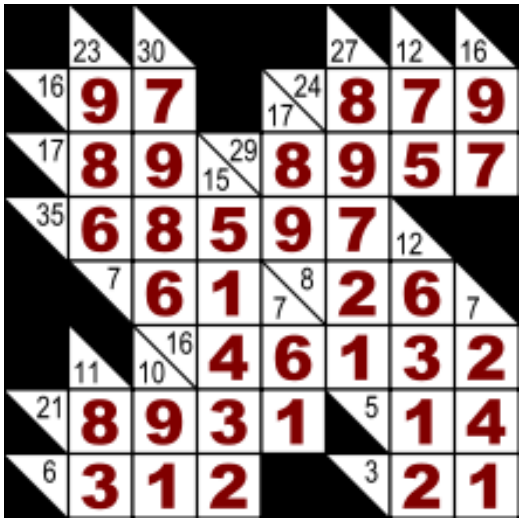

Abbildung 2.2: Die Lösung zu 2.1[Wikc]

Eine sehr bekannte Version von Logikrätseln sind allerdings Textaufgaben, die mittels Deduktion gelöst werden können, wie im folgenden Unterkapitel 2.1.1 vorgestellt wird.

[2] Quelle: [Seca]

### 2.1.1 Eine Logelei: Vier Ehepaare aus der Sonnenstraße

Betrachten wir ein Rätsel aus der *Zeit* [3] aus dem Jahr 2004.

| |
|---|
| In den vier Wohnungen der Sonnenstraße 17 wohnen vier Ehepaare. Die Herren heißen Dominik, Michael, Robert und Stefan, die Damen Eveline, Katja, Marta und Sandra. Jeder von den acht geht einem besonderen Hobby nach (Go spielen, Fußball spielen, Jonglieren, Kochen, Kakteen züchten, Reiten, Schwimmen und Tanzen). |
| Gegenüber wohnt Frau Glotke. Sie ist sehr neugierig und möchte wissen, wer mit wem verheiratet ist und wer welches Hobby hat. Sie hat deswegen die Sonnenstraße 17 beobachtet und sich dabei die folgenden Notizen gemacht: |
| Der Ehemann von Sandra kann nicht kochen. |
| Entweder Katja tanzt, oder Stefan spielt Go. |
| Entweder Robert kocht, oder Michael und Marta sind nicht miteinander verheiratet. |
| Die Frau von Robert schwimmt. |
| Der Ehepartner der Person, die Kakteen züchtet, spielt Fußball. |
| Entweder Sandra tanzt, oder Katjas Ehemann kann nicht kochen. |
| Entweder der Ehepartner der Person, die reitet, ist der Koch, oder Michael spielt kein Go. |
| Der Ehepartner der Person, die schwimmt, spielt Go. |
| Entweder Katjas Ehemann spielt Go, oder Dominik kocht. |
| Der Ehemann von Marta züchtet Kakteen. |
| Und wer ist jetzt mit wem verheiratet und hat welches Hobby? |

Tabelle 2.1: Logelei der vier Ehepaare

Dieses Rätsel werden wir später als Eingabe für das von mir erstellte Programm transformieren und uns die Lösung im Kapitel 6 anschauen.

[3] Quelle: [Secb]

## 2.2 Aussagenlogik

Aussagenlogik ist ein Teilgebiet der Logik, das sich mit Aussagen befasst, die entweder *wahr* oder *falsch* sein können. Sie können mit verschiedenen *Konnektoren* zu einer weiteren Aussage verknüpft werden. Der Wahrheitswert einer Aussage ist somit nur abhängig von den Wahrheitswerten ihrer Teilaussagen. Diese und folgende Erklärungen zur Aussagenlogik mit Übungen zum besseren Verständnis lassen sich genauer nachschlagen in [Waga], sowie in den Büchern [KK06] und [Sch95].

### 2.2.1 Syntax der Aussagenlogik

Die Syntax der Aussagenlogik besteht aus drei Komponenten:

- den Aussagenvariablen *p, q, r, ...*
- den Konnektoren $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$
- und den Hilfszeichen ( , )

#### Aussagenvariablen

Aussagenvariablen stehen stellvertretend für Aussagen und können einen Wert der Menge `True, False` annehmen. Wir schreiben sie symbolisch $p, q, r, ....$
Wir verwenden sie, um offen zu lassen, ob eine Aussage wahr oder falsch ist und es gegebenenfalls selbst zu bestimmen, wenn wir verschiedene Aussagen gegeben haben und Schlussfolgerungen ziehen können.

#### Konnektoren

Es existieren verschiedene Konnektoren, um Aussagen in der Aussagenlogik miteinander zu verknüpfen.

| **Bezeichnung** | **Bedeutung** | **Symbol** |
|---|---|---|
| Negation | *nicht* | $\neg$ |
| Konjunktion | *und* | $\wedge$ |
| Disjunktion | *oder* | $\vee$ |
| Kontravalenz | *entweder-oder* | $\oplus$ |
| Implikation | *wenn ..., dann* | $\Rightarrow$ |
| Äquivalenz | *genau dann, wenn* | $\Leftrightarrow$ |

Tabelle 2.2: Konnektoren der Aussagenlogik

Im Folgenden sind die Wahrheitswerte der einzelnen Konnektoren in Wahrheitstafeln zu sehen.

| $p$ | $\neg p$ |
|---|---|
| 1 | 0 |
| 0 | 1 |

Tabelle 2.3: Negation

| $p$ | $q$ | $p \wedge q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Tabelle 2.4: Konjunktion

| $p$ | $q$ | $p \vee q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Tabelle 2.5: Disjunktion

| $p$ | $q$ | $p \oplus q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Tabelle 2.6: Kontravalenz

| $p$ | $q$ | $p \Rightarrow q$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Tabelle 2.7: Implikation

| $p$ | $q$ | $p \Leftrightarrow q$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Tabelle 2.8: Äquivalenz

Dabei gilt zu beachten, dass die Negation der einzige Konnektor ist, der einstellig ist, während alle anderen Konnektoren zweistellig sind.
Die Bindungsstärke der Konnektoren nimmt in folgender Reihenfolge ab:
$\neg$ (nicht), $\wedge$ (und), $\vee$ (oder), $\Rightarrow$ (Implikation), $\Leftrightarrow$ (Äquivalenz).

### Grammatik der Aussagenlogik

**Definition 2.1** (Grammatik der Aussagenlogik)**.** *Das Aussagenkalkül ist durch folgende Grammatik bestimmt*[4]*:*

$F ::= A \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Rightarrow F_2 \mid F_1 \Leftrightarrow F_2$

*Hierbei sind $F, F_1$ und $F_2$ Formeln und $A$ ein Atom, also in diesem Fall eine Aussagenvariable.*

In Worten lauten diese Formationsregeln:

1. Eine Aussagenvariable ist eine Formel.

[4]Vgl. [SS12] für den folgenden Abschnitt.

2. Ist $F$ eine Formel, dann ist auch $\neg F$ eine Formel.
3. Sind $F$ und $G$ Formeln, dann sind
   - $(F \wedge G)$
   - $(F \vee G)$
   - $(F \Rightarrow G)$
   - $(F_1 \Leftrightarrow F_2)$

   ebenfalls Formeln.
4. Ein Ausdruck ist nur dann eine Formel, wenn er durch Anwendung der obenstehenden Regeln konstruiert werden kann.

### 2.2.2 Semantik der Aussagenlogik

Die Semantik legt in der Aussagenlogik fest, wie die Formeln zu ihren Wahrheitswerten ausgewertet werden. Dafür betrachten wir nun die Auswertungen der möglichen Interpretationen.

**Definition 2.2** (Interpretation)**.** *Eine* Interpretation $I$ *bildet jede aussagenlogische Formel auf ein Element der Menge* $\{\texttt{True}, \texttt{False}\}$ *ab.*

*Die Erweiterung einer Interpretation auf eine Formel $F$ ist definiert durch:*

$$I(\neg F) = \begin{cases} \texttt{True}, & \textit{gdw. } I(F) = \texttt{False} \\ \texttt{False}, & \textit{gdw. } I(F) = \texttt{True} \end{cases}$$

$$I(F_1 \wedge F_2) = \begin{cases} \texttt{True}, & \textit{gdw. } I(F_1) = \texttt{True} \textit{ und } I(F_2) = \texttt{True} \\ \texttt{False}, & \textit{sonst} \end{cases}$$

$$I(F_1 \vee F_2) = \begin{cases} \texttt{True}, & \textit{gdw. } I(F_1) = \texttt{True} \textit{ oder } I(F_2) = \texttt{True} \\ \texttt{False}, & \textit{sonst} \end{cases}$$

$$I(F_1 \Rightarrow F_2) = \begin{cases} \texttt{True}, & \textit{gdw. } I(F_1) = \texttt{False} \textit{ oder } I(F_1) = \texttt{True} \textit{ und } I(F_2) = \texttt{True} \\ \texttt{False}, & \textit{sonst} \end{cases}$$

$$I(F_1 \Leftrightarrow F_2) = \begin{cases} \texttt{True}, & \textit{gdw. } I(F_1) = I(F_2) \\ \texttt{False}, & \textit{sonst} \end{cases}$$

**Definition 2.3** (Modell)**.** *Wenn $I(F) = \texttt{True}$ gilt, so nennt man $I$ ein* Modell *für die Formel $F$.*

**Definition 2.4.** *Eine aussagenlogische Formel $F$ ist*

- allgemeingültig *genau dann, wenn für alle Interpretationen $I$ gilt: $I(F) = \texttt{True}$*

- erfüllbar *genau dann, wenn es eine Interpretation I gibt mit:* $I(F) =$ `True`.
- falsifizierbar *genau dann, wenn es eine Interpretation I gibt mit* $I(F) =$ `False`.
- widersprüchlich *genau dann, wenn für alle Interpretationen I gilt:* $I(F) =$ `False`

*Allgemeingültige Formeln nennt man* **Tautologien**, *widersprüchliche Formeln werden* **Kontradiktionen** *genannt.*

### 2.2.3 Deduktionstheoreme der Aussagenlogik

Betrachten wir nun im Folgenden die Begriffe der logischen Folgerung und der syntaktischen Ableitung und der logischen Äquivalenz.

**Definition 2.5** (Logische Folgerung)**.** *Wenn sich eine aussagenlogische Formel G aus einer anderen Formel F folgern lässt, d.h. für jede Interpretation I mit* $I(F) = 1$ *gilt stets auch* $I(G) = 1$*, dann schreiben wir als logische Folgerung* $F \models G$.

Wenn beispielsweise gilt $I(F)$=1, dann schreiben wir auch $I \models F$. Man sagt dann auch: I ist ein Modell für F. Oder: I erfüllt F.

**Definition 2.6** (Syntaktische Ableitung)**.** *Wenn sich eine aussagenlogische Formel G aus einer anderen Formel F syntaktisch ableiten lässt, dann schreiben wir als syntaktische Ableitung* $F \vdash G$.

**Definition 2.7** (Logische Äquivalenz)**.** *Zwei aussagenlogische Formeln F und G heißen logisch äquivalent, symbolisch* $F \equiv G$ *oder* $F \sim G$*, genau dann, wenn sie unter den gleichen Belegungen ihrer Variablen wahr oder falsch sind, d.h., wenn sie bei gleichen Belegungen stets den gleichen Wahrheitswert haben.*
*Zwei Formeln F und G sind* logisch äquivalent *genau dann, wenn* $F \Leftrightarrow G$ *eine Tautologie ist. Sind F und G* logisch äquivalent*, dann gilt für alle Interpretationen I:* $I \models F$ *genau dann, wenn* $I \models G$.

Beispielsweise sind die Formeln $\neg(p \vee q)$ und $\neg p \wedge \neg q$ logisch äquivalent (siehe Tabelle 2.9) und lassen sich somit ineinander überführen, was für die Transformationen, die in Kapitel 3 erläutert werden, relevant ist.

| $p$ | $q$ | $\neg(p \vee q)$ | $\neg p \wedge \neg q$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |

Tabelle 2.9: Wahrheitstafel zweier logisch äquivalenter Formeln

### 2.2.4 Bivalenzprinzip

Für die klassische Aussagenlogik gilt das Bivalenzprinzip, auch Polaritätsprinzip genannt, das Folgendes besagt:

Jede Aussage ist entweder wahr oder falsch, d.h.:

- keine Aussage kann zugleich wahr und falsch sein (Prinzip vom ausgeschlossenen Widerspruch)
- keine Aussage kann etwas anderes als wahr oder falsch sein (Prinzip vom ausgeschlossenen Dritten)

### 2.2.5 Beispielsätze

Betrachten wir ein paar natürlichsprachige Beispielsätze, die in Aussagenlogik umgeformt werden.

*Sybille ist nicht faul.*
$p$ = *Sybille ist faul.*
In Aussagenlogik: $\neg p$

*David ist fleißig und klug.*
$p$ = *David ist fleißig.*
$q$ = *David ist klug.*
In Aussagenlogik: $p \wedge q$

*Maria ist klug oder nett.*
$p$ = *Maria ist nett.*
$q$ = *Maria ist klug.*
In Aussagenlogik: $p \vee q$

*Martha ist entweder im Kino oder im Theater.*
$p$ = *Martha ist im Kino.*
$q$ = *Martha ist im Theater.*
In Aussagenlogik: $p : q$

*Wenn Sabrina krank ist, liegt sie im Bett.*
$p$ = *Sabrina ist krank.*
$q$ = *Sabrina liegt im Bett.*
In Aussagenlogik: $p \Rightarrow q$

*Josephine putzt genau dann das Haus, wenn es schmutzig ist.*
$p$ = *Josephine putzt das Haus.*
$q$ = *Das Haus ist schmutzig.*
In Aussagenlogik: $p \Leftrightarrow q$

*Josephine putzt genau dann das Haus, wenn es schmutzig ist und Nacht ist oder wenn Julia sie darum bittet und es tagsüber ist.*
$p$ = *Josephine putzt das Haus.*
$q$ = *Das Haus ist schmutzig.*
$r$ = *Es ist Nacht.*
$s$ = *Julia bittet Josephine, das Haus zu putzen.*
$t$ = *Es ist Tag.*
In Aussagenlogik: $p \Leftrightarrow ((q \wedge r) \vee (s \wedge t))$

Eine einfache **Tautologie** wäre:

*Es ist nicht der Fall, dass Sabrina zu Hause und unterwegs ist.*

Ein Beispiel für eine **Kontradiktion** wäre:

*Martha putzt und putzt nicht.*

Wir sehen, dass sich viele Aussagen mithilfe der Aussagenlogik formulieren und beliebig schachteln lassen.
Auch Aussagen der Form „Für alle Menschen gilt, dass sie gutes Wetter mögen." lassen sich zwar als atomare Aussagen in Aussagenlogik formulieren, jedoch lassen sich keine logischen Schlüsse über verschiedene atomare Aussagen ziehen.
Dabei gibt es Sätze, deren inneren Aufbau wir mithilfe von Existenzquantoren und Allquantoren sehr wohl miteinander vergleichen und aus ihnen Schlüsse ziehen könnten. Betrachten wir folgendes Beispiel:

*Menschen sind sterblich.*
*Einstein ist ein Mensch.*

Daraus folgt logisch:

*Einstein ist sterblich.*

Um solche Aussagen entsprechend zu formulieren, bedarf es nun einer höheren Logik. Daher betrachten wir im Folgenden die Syntax der Prädikatenlogik als Erweiterung der Aussagenlogik.

## 2.3 Prädikatenlogik

Im letzten Unterkapitel 2.2 haben wir den Kalkül der Aussagenlogik betrachtet. Nun betrachten wir eine Erweiterung dieser Logik, die über mehr Operatoren verfügt: die Prädikatenlogik.

### 2.3.1 Die Syntax der Prädikatenlogik

Nachfolgend stehen die Definitionen der Syntax und der Regeln der Prädikatenlogik erster Stufe[5].

[5]Vgl. [Wagb] für den gesamten Abschnitt 2.3.1.

### Syntax der Prädikatenlogik

Die Syntax der Prädikatenlogik besteht aus folgenden Komponenten:

- den Individuenkonstanten *x, y, z, ...*
- den Individuenvariablen *a, b, c, ...*
- den Funktionen (mit verschiedenen Stelligkeiten $> 0$) *f, g, h,...*
- den Prädikaten (mit verschiedenen Stelligkeiten $> 0$) *p, q, r,...*
- der Negation $\neg$
- der Konjunktion und Disjunktion $\wedge$, $\vee$
- der Implikation und Äquivalenz $\Rightarrow$, $\Leftrightarrow$
- dem Allquantor $\forall$
- dem Existenzquantor $\exists$
- und den Hilfszeichen ( , ) , { , } , ','

### Individuenkonstanten

Individuenkonstanten werde Begriffe genannt, die ein einziges Individuum repräsentieren und werden konventionell mit den ersten Buchstaben des Alphabets $(a, b, c, ...)$ dargestellt. Dazu gehören Eigennamen wie *Fritz, Hans, Maria*, wohingegen Wörter wie *Planet, Mensch, Katze* Allgemeinbegriffe sind, die eine Klasse repräsentieren und deren Prädikat somit auf alle Elemente dieser Klasse zutreffen.

### Individuenvariablen

Nun gibt es aber auch Sätze, die keinen Individualbegriff enthalten. Oft braucht man dann *Individuenvariablen*, die üblicherweise durch $(x, y, z, ...)$ dargestellt werden.

Ein Beispiel dafür wäre die Aussage:

„Katzen sind Säugetiere."

Dies lässt sich umformulieren in:

„Wenn etwas eine Katze ist, dann ist es ein Säugetier."

Wollte man also einen Satz der Form „etwas ist eine Katze" sagen, so würde man eine Individuenvariable verwenden: Katze($x$). Der gesamte Satz würde dann in Prädikatenlogik lauten:

$$\forall x \text{ Katze}(x) \Rightarrow \text{Säugetier}(x)$$

### Funktionen

Man nennt $f(t_1, ..., t_n)$ einen Funktionsterm. Funktionen sind abgewandelte Repräsentationen von Individuen.
Im Rahmen dieser Arbeit vernachlässigen wir Funktionen, da wir sie auch als Prädikate schreiben können und für das Lösen der Rätsel nicht benötigen.

### Prädikate

Prädikate kann man als eine Art Funktion auffassen, die aber im Gegensatz zu jenen Wahrheitswerte repräsentieren. Prädikate können für Allgemeinbegriffe stehen wie *Planet, Mensch, Reptil*, aber auch für Eigenschaften wie *groß, dünn, klug*.
So lassen sich einige natürliche Sätze folgendermaßen mit Prädikaten umschreiben:

| Satz | Prädikatenlogik |
|---|---|
| Sandra ist ein Mensch. | Mensch(Sandra) |
| Florian ist nicht dumm. | ¬dumm(Florian) |
| Wenn Lea zu Hause ist, ist Hans wach. | zu Hause(Lea) ⇒ wach(Hans) |

Tabelle 2.10: Prädikate

### Junktoren

Die Junktoren $(\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow)$ sind äquivalent zu den Konnektoren der Aussagenlogik, siehe Abschnitt 2.2.1.

### Quantoren

Die Quantoren sind Grundzeichen der Prädikatenlogik und binden Variablen. Es gibt den Allquantor, der natürlichsprachig als „für alle" oder „für jeden" zu verstehen ist, und den Existenzquantor, der mit „es existiert mindestens ein" übersetzt werden kann.

#### Der Allquantor

$\forall\ x.p(x)$ bedeutet, dass für alle $x$ gilt, dass $p(x)$ wahr ist. Wenn wir, wie in meiner Arbeit bei der getypten Prädikatenlogik der Fall, nur mit endlichen Mengen arbeiten, dann lässt sich dies auch mithilfe der Konjunktion ausdrücken:
$\bigwedge\ x.p(x)$. Dies steht für $p(x_1) \wedge p(x_2) \wedge ... \wedge p(x_n)$ und veranschaulicht damit auch logisch, dass $p(x)$ für alle, also jedes einzelne $x$ einer endlichen Menge gilt, bzw.

gelten muss, wenn die gesamte Formel wahr sein soll.

#### Der Existenzquantor

$\exists\, x.p(x)$ bedeutet, dass (mindestens) ein $x$ existiert, für das gilt, dass $p(x)$ wahr ist. Wieder gilt: Wenn wir nur mit endlichen Mengen für $x$ arbeiten, dann lässt sich der Existenzquantor auch mithilfe der Disjunktion ausdrücken:
$\bigvee\, x.p(x)$, was für $p(x_1) \vee p(x_2) \vee ... \vee p(x_n)$ steht und damit logisch veranschaulicht, dass $p(x)$ für nur ein $x$ der endlichen Menge gelten muss, um als gesamte Formel wahr zu sein.

### Gebundene und freie Variablen

Der Bindungsbereich eines Quantors, auch Skopus genannt, bezeichnet die Reichweite eines Quantors.
Der Skopus von $\forall x$ in $\forall x.(F \vee G)$ ist $F \vee G$, der Skopus von $\exists x$ in $\exists x.F \wedge G$ hingegen ist nur $F$.

**Definition 2.8** (Skopus eines Quantors)**.** *Ein Quantor bindet stärker als die Konnektoren. Somit reicht der Skopus eines Quantors bis zum nächsten Konnektor, sofern nicht durch Hilfszeichen anders gesetzt.*

Von einer *gebundenen* Variablen spricht man, wenn sie in einer Formel direkt nach einem Quantor steht oder wenn sie im Skopus eines Quantors mit der gleichen Variablen vorkommt.
Eine Variable wird *frei* genannt, wenn sie nicht gebunden ist.

Wenn in einer Formel F alle Variablen durch Quantoren gebunden sind und demnach keine freien Variablen existieren, wird F als *geschlossene Formel* bezeichnet.

*Beispiele:*

In der Formel $\exists x.p(x, y) \vee q(x)$ kommt $x$ das erste und zweite Mal gebunden vor, beim dritten Mal jedoch, nach der Disjunktion, ist es frei. $y$ ist ebenfalls frei.
In der Formel $\forall x.(p(x, x) \wedge q(x))$ hingegen sind alle Variablen gebunden, da der Skopus des Allquantors durch die Klammern die ganze Formel umfasst und nur $x$ in ihr vorkommt.

## 2.3.2 Semantik der Prädikatenlogik

Betrachten wir nun die Semantik der Prädikatenlogik.

### Terme

Ein Term wird in der Prädikatenlogik folgendermaßen definiert:

1. Jede Individuenvariable ist ein Term.
2. Jede Individuenkonstante ist ein Term.
3. Ist $f$ eine $n$-stellige Funktion und sind $t_1, ..., t_n$ Terme, dann ist $f(t_1, ..., t_n)$ ein Term.
4. Nur so gebildete Zeichenketten sind Terme.

#### Formeln

Ist $p$ ein $n$-stelliges Prädikat und sind $t_1, ..., t_n$ Terme, dann ist $p(t_1, ..., t_n)$ eine *Primformel*.
Und ist $p(t_1, ..., t_n)$ eine Primformel, so heißen die Terme $t_1, ..., t_n$ die *Argumente* der Primformel.

Eine Formel wird über die Grammatik definiert:

$F ::= A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid F_1 \Rightarrow F_2 \mid F_1 \Leftrightarrow F_2 \mid \forall x.F \mid \exists x.F$

Dies lässt sich auch in Worten mit folgenden Regeln umschreiben:

1. Primformeln sind Formeln.
2. Ist $F$ eine Formel, so ist auch $\neg F$ eine Formel.
3. Sind $F$ und $G$ Formeln, so sind auch $(F \wedge G), (F \vee G), (F \Rightarrow G)$ und $(F \Leftrightarrow G)$ Formeln.
4. Ist $F$ eine Formel und $x$ eine Variable, dann sind $(\forall x.F)$ und $(\exists x.F)$ Formeln.

#### Interpretation der Prädikatenlogik

Im Rahmen dieser Arbeit vernachlässigen wir die Interpretationen der allgemeinen Prädikatenlogik erster Stufe, da diese recht aufwendig zu beschreiben sind und in dieser Arbeit eine getypte Prädikatenlogik über endlichen Mengen in Abschnitt 3.1 eingeführt und betrachtet wird.
Einen genauen Einblick in die Interpretationen und die Semantik der Prädikatenlogik erster Stufe nach Tarski gibt es jedoch nachzuschlagen in Quelle [SS12], in Abschnitt 4.2: „Semantik von PL1 (nach Tarski)" auf den Seiten 73 bis 82.

### 2.3.3 Beispiele

Betrachten wir abschließend noch einige Beispielsätze in Prädikatenlogik in Tabelle 2.11.

| Satz | Prädikatenlogik |
|---|---|
| Es gibt eine Stadt, die nördlich von Frankfurt liegt. | $\exists x.(s(x) \wedge n(x, f))$ |
| Jede Stadt liegt nördlich von jeder Stadt, die südlich von ihr liegt. | $\forall x \forall y.(s(x) \wedge s(y) \wedge n(x, y)) \Rightarrow s(y, x)$ |
| Keine Stadt liegt nördlich von sich selbst. | $\neg \exists x.(s(x) \wedge n(x, x))$ |
| Wenn Frankfurt eine Stadt ist und nördlich von München liegt, dann gibt es eine Stadt, die nördlich von München liegt. | $(s(f) \wedge n(f, m)) \Rightarrow \exists x.(s(x) \wedge n(x, m))$ |

Tabelle 2.11: Prädikatenlogische Sätze (Quelle: [Esc])

Wir sehen nun, dass sich mithilfe der Prädikatenlogik Sätze formulieren lassen, die näher an unserer natürlichen Sprache sind und demnach ausdrucksstärker ist als die Aussagenlogik.
Daraus ergibt sich, dass es sich mit der Absicht, die natürliche Sprache (oder eine eingeschränkte Version von ihr) als Eingabe für einen SAT-Solver zu formulieren, anbietet, die eingeschränkte natürliche Sprache in Prädikatenlogik zu transformieren und diese dann wiederum in Aussagenlogik zu übersetzen, um sie leicht abgewandelt in KNF dem SAT-Solver als Eingabeinstanz zu übergeben.
Genau diese Transformationen nehme ich in dieser Arbeit vor und erläutere sie in Kapitel 3.

## 2.4 SAT - Das Erfüllbarkeitsproblem der Aussagenlogik

Das SAT-Problem war in der Komplexitätstheorie das erste nachgewiesene NP-vollständige Problem, also ein Problem, das in der Komplexitätsklasse NP liegt[6] und NP-schwer ist[7]. Somit gehört SAT zur Klasse der Probleme, die in polynomieller Zeit von einer nichtdeterministischen Turingmaschine gelöst werden kann.

[6] Das bedeutet, dass ein deterministisch arbeitender Rechner nur polynomiell viel Zeit benötigt, um eine vorgeschlagene Lösung als tatsächliche Lösung des dazugehörigen Suchproblems zu verifizieren oder zu falsifizieren.

[7] „Mindestens so schwer" wie alle Probleme in NP: Ein Algorithmus, der ein NP-schweres Problem löst, könnte also dazu benutzt werden, alle Probleme in NP zu lösen, vgl. [Sch].

Das SAT-Problem taucht in einer großen Vielfalt von praktischen Anwendungen auf, wie zum Beispiel in der Erkennung von künstlicher Intelligenz, in der automatischen Planung („automated planning and scheduling", Zweig der künstlichen Intelligenz) und im Model Checking - ein Verfahren zur vollautomatischen Verifikation einer Systembeschreibung gegen eine Spezifikation.[8] Weitere Erklärungen zum SAT-Problem lassen sich im Buch [ST12] zum Erfüllbarkeitsproblem nachschlagen.

Das (Boolesche) SAT-Problem besteht aus einer Formel, die aus Klauseln besteht, die wiederum Literale in negierter und nicht-negierter Form enthalten und aus der Frage, ob die Formel erfüllbar ist.[9]
Liegt das SAT-Problem in KNF (Konjunktiver Normalform) vor, so sind alle Klauseln in der Formel miteinander verundet (*konjungiert*) und alle Literale in den Klauseln miteinander verodert (*disjungiert*).
Eine typische KNF kann also folgendermaßen aussehen:

$$((a \vee b) \wedge (a \vee c) \wedge d \wedge (b \vee e))$$

Diese kann man auch leicht in die *Klauselmenge* überführen:

**Konjunktive Normalform in Klauselmenge:**

Die konjunktive Normalform ist eine Konjunktion von Disjunktionen oder Literalen. Die Klauselform ist lediglich eine andere Schreibweise der konjunktiven Normalform, bei der eine Menge von Mengen angegeben wird.
Die Überführung der beiden Schreibweisen verhält sich wie folgt:

$$((a \vee b) \wedge (a \vee c) \wedge d \wedge (b \vee e)) \Leftrightarrow \{\{a, b\}, \{a, c\}, \{d\}, \{b, e\}\}$$

Es existieren auch einige Variationen des SAT-Problems. Beispielsweise ist 3-SAT eine Variante, in der die Klauseln der KNF höchstens drei Literale enthalten. Trotz dieser Einschränkung bleibt auch 3-SAT NP-vollständig. MAX-SAT beschreibt das Problem, die maximale Anzahl erfüllbarer Klauseln einer Formel zu bestimmen und liegt in der Komplexitätsklasse PSPACE[10].

### 2.4.1 SAT-Solver

Es sind keine Algorithmen bekannt, die das SAT-Problem effizient, korrekt und für alle Eingabeinstanzen lösen, und auch wenn die Vermutung besteht, dass keine existieren, konnte das bisher nicht mathematisch bewiesen werden. Dennoch können

[8] Vgl. [Lan12].
[9] „Boolesche" besagt, dass jede Variable genau den Wert True oder False annehmen kann und nichts anderes.
[10] Die Klasse NP liegt in PSPACE, NP $\subseteq$ PSPACE.

einige in der Praxis auftretende Versionen des SAT-Problems recht effizient durch heuristische SAT-Solver gelöst werden. SAT-Solver erwarten in der Regel eine Eingabe in Form einer Konjunktiven Normalform (KNF). Dafür liegen die Klauseln nur in verundeter Form (Konjunktionen) vor und enthalten ihrerseits nur Veroderungen (Disjunktionen).

Sat4j ist eine Open Source Java Library, die das Boolesche SAT-Problem löst. [11]
„Chaff" ist ein Algorithmus, der das Boolesche SAT-Problem als Eingabe erwartet und auf DPLL basiert.
Das Verfahren, das 1960 von Martin Davis und Hilary Putnam entwickelt wurde, wurde als Davis-Putnam-Verfahren bekannt und 1962 mit Hilfe von George Logemann und Donald Loveland optimiert und wird im folgenden Kapitel 2.4.1 ausführlich erläutert.

Durch die breite Anwendung finden regelmäßig Wettbewerbe zwischen den neuesten SAT-Solvern statt, welche von modernen Implementierungen des DPLL-Verfahrens angeführt werden. Für meine Implementierung wurde ebenfalls der DPLL-Algorithmus als SAT-Solver gewählt.

### Der DPLL-Algorithmus

Der Davis-Putnam-Logemann-Loveland-Algorithmus wurde im Jahr 1962 von Martin Davis, Hilary Putnam, George Logemann und Donald Loveland vorgeschlagen und ist eine Weiterentwicklung des Davis-Putnam-Algorithmus aus dem Jahr 1960. Es löst das Erfüllbarkeitsproblem für Formeln in konjunktiver Normalform und ist nach über 50 Jahren immer noch Basis der effizientesten SAT-Solver. In dem Verfahren wird jede erfüllende Variablenbelegung sukzessive konstruiert und die gegebene Formel dabei vereinfacht. Der Algorithmus nutzt dabei das Backtracking-Prinzip und arbeitet demnach nach dem Prinzip der Tiefensuche. Dabei wird nach dem Versuch-und-Irrtum-Prinzip (*trial and error*) vorgegangen, was bedeutet, dass versucht wird, aus einer erreichten Teillösung eine Gesamtlösung auszubauen. Wenn eine Teillösung nicht zu einer endgültigen Lösung führen kann, wird der letzte Schritt, bzw. werden die letzten Schritte zurückgenommen, und stattdessen alternative Wege ausprobiert. So wird sichergestellt, dass alle möglichen Lösungswege auch getestet werden.
Die Tiefensuche und somit auch Backtracking haben eine Worst-Case-Laufzeit von $O(z^N)$, wobei $z$ die Anzahl der maximalen Verzweigungen und $N$ die maximale Tiefe des Lösungsbaums ist.

Dieser rekursive Algorithmus eliminiert in jedem Schritt mindestens eine Variable, wenn möglich durch die Elimination einer Einheitsklausel, also einer Klausel, die genau ein Literal enthält. Wenn dieses Literal positiv (nicht-negiert) vorliegt, kann die Klausel nur erfüllt werden, wenn diese einzige Variable auf 'True' gesetzt wird, somit ist keine weitere Verzweigung (elseif) nötig.

[11] Siehe [NE].

```
DPLL(M)
Vorbedingung: M ist eine Klauselmenge
if M = {}
    return 1
elseif {} ∈ M
    return 0
elseif there exists unit clause in M
    let L be such that {L} ∈ M
    return DPLL(M|L)
else
    let X ∈ vars(M)
    if DPLL(M|X) = 1
      return 1
    else
      return DPLL(M|¬X)
Nachbedingung: return ∈ {0,1} und return = 1 genau dann, wenn ⋀⋁M erfüllbar ist.
```

Abbildung 2.3: Der DPLL Algorithmus[Kie]

Betrachten wir den Umgang des DPLL mit einer Beispielinstanz2.4.

Sei $M = X_0, X_1, -X_0, -X_1, X_1, X_2, -X_1, -X_2, X_2, X_0, -X_2, -X_0$:

**Beispiel** Es sei $M = \{\{X_0, X_1\}, \{\neg X_0, \neg X_1\}\{X_1, X_2\}, \{\neg X_1, \neg X_2\}, \{X_2, X_0\}, \{\neg X_2, \neg X_0\}\}$

Dann arbeitet der Algorithmus wie folgt:

```
DPLL({{X0, X1}, {¬X0, ¬X1}{X1, X2}, {¬X1, ¬X2}, {X2, X0}, {¬X2, ¬X0}} )
    Wähle Variable X0.
    DPLL({{¬X1}, {X1, X2}, {¬X1, ¬X2}, {¬X2}} )
        Wähle Literal ¬X1.
        DPLL({{X2}, {¬X2}})
            Wähle Literal X2.
            DPLL({{}})
                return 0
    DPLL({{X1}, {X1, X2}, {¬X1, ¬X2}, {X2}} )
        Wähle Literal X1.
        DPLL({{¬X2}, {X2}})
            Wähle Literal X2.
            DPLL({{}})
                return 0
    return 0
```

Abbildung 2.4: Der DPLL Algorithmus[Kie]

An diesem Beispiel lässt sich beobachten, dass der DPLL-Algorithmus nur zwei Fallunterscheidungen vornimmt, während der naive Algorithmus $2^3$ (wegen zwei Belegungsoptionen und drei Variablen), also acht Fallunterscheidungen vornehmen müsste.

Der DPLL Algorithmus terminiert immer, ist korrekt und vollständig. Seine Worst-Case-Laufzeit beträgt $O(2^n)$ und sein Worst-Case-Speicherplatz beträgt $O(n)$.[12]

### Einsatz von SAT-Solvern

Man bemerkt den großen Nutzen von SAT-Solvern beispielsweise bei der Kombinierbarkeit von Optionen: Eine Option A kann nicht mit Option B zusammen gewählt werden; oder wenn Option A gewählt wird, darf B nicht gewählt werden, etc. - diese Bedingungen eignen sich gut für SAT-Solver, wodurch SAT-Solver sich auch bei Hardware-Verifikationen großer Beliebtheit erfreuen.[13] Auch bei Hierarchien von einzelnen Komponenten lassen sich SAT-Solver sinnvoll einsetzen, beispielsweise: Ein Pkw enthält einen Motor, der Motor enthält einen Zylinder, und so weiter.
Und auch bei Vervollständigungen von Teilspezifikationen sind SAT-Solver von Vorteil: So lassen sich einfache Bedingungen der Form „egal welche Marke, Hauptsache ein 4-Kern-Prozessor" umsetzen.
Somit können Produktkonfigurationen verschiedener Produkte wie Pkws, Telekommunikationsanlagen, Flugzeugen und Computern bis hin zu medizinischen Geräten von SAT-Solvern profitieren.

Dank intelligenter Heuristiken können moderne SAT-Solver typischerweise Instanzen mit etwa $10^5$ Variablen und $10^6$ Klauseln lösen. Dabei bleibt zu beachten, dass es auch weit kleinere Instanzen geben kann, die trotzdem nicht effizient gelöst werden können. [14]

[12] Siehe Quellen: [Wikb], [DP60], [DLL62] und [NOT06].
[13] Vgl. [Lan12].
[14] Vgl. [Kas].

# Kapitel 3

# Eine getypte Prädikatenlogik auf endlichen Mengen - die Rätsellogik

In diesem Kapitel erläutere ich die Transformationen, die notwendig sind, um die Ausgabe der Benutzungsschnittstelle, auf die in Kapitel 5 eingegangen wird, in eine Eingabeinstanz für den SAT-Solver umzuwandeln.
Dafür wird im folgenden Abschnitt 3.1 die getypte Prädikatenlogik auf endlichen Mengen mit Syntax und Semantik vorgestellt, die im Rahmen dieser Arbeit verwendet wird. Anschließend betrachten wir die Transformation der Rätsellogik in Aussagenlogik in Abschnitt 3.2, wobei wir auch die Komplexitäten des SAT-Problems in Prädikaten- sowie Aussagenlogik betrachten und die Korrektheit und Vollständigkeit der Transformation nachweisen.
In Abschnitt 3.3 betrachten wir dann die Transformation der Aussagenlogik in eine Klauselmenge unter Zuhilfenahme der schnellen KNF. Und in Abschnitt 3.4 betrachten wir abschließend die Rückrechnung der Ausgabe des SAT-Solvers in die Rätsellogik.

## 3.1 Syntax und Semantik der Rätsellogik

Wie die Überschrift dieses Unterkapitels bereits verrät, betrachten wir im Kontext dieser Arbeit eine besondere Form der Prädikatenlogik.

In der allgemeinen Prädikatenlogik erster Stufe können Individuenvariablen wie $x$ in $\forall x(P(x))$ jede beliebige Grundmenge für $x$ annehmen und jede beliebige Interpretation des Prädikats $P$ ist möglich.
Im Rahmen dieser Arbeit jedoch betrachten wir eine getypte Prädikatenlogik auf endlichen Mengen, die feste vordefinierte Mengen der Individuenvariablen und bestimmte Interpretationen besitzen. Bei den Rätseln, die wir zu lösen versuchen, ist der Kontext von vornherein gegeben und klar, welche Elemente - z.B. Personen - existieren und damit eine bestimmte, endliche Menge an Konstanten für $x$ gegeben. Der Einfachheit halber nennen wir die getypte Prädikatenlogik auf endlichen Mengen im Folgenden **„Rätsellogik"**.

### 3.1.1 Syntax der Rätsellogik

**Definition 3.1** (Signatur)**.** *Eine Signatur ist ein Tupel* $(K, S, P)$ *wobei*

- $K$ *ist eine nichtleere endliche Menge von* Konstanten*.*
- $S$ *ist eine nichtleere endliche Menge von* Sorten $S$*. Für jede Sorte* $s \in S$ *gibt es eine endliche nichtleere Menge von Konstanten konstanten*$(s) \subseteq K$*.*
- $P$ *ist eine Menge von* Prädikaten*, wobei jedes Prädikat* $p \in P$ *eine Stelligkeit* $\mathrm{ar}(p) \geq 0$ *hat. Jedes* $p \in P$ *hat einen* Typ $p :: s_1 \to \ldots s_n \to \{\texttt{True}, \texttt{False}\}$*.*

*Jede Konstante* $k \in K$ *gehört zu mindestens einer einer Sorte* $k \in K$*, mit sorte*$(k) \subseteq S$ *bezeichnen wir die Sorten von* $k$ *und fordern sorte*$(k) \neq \emptyset$ *für jedes* $k \in K$*. Wenn* $s_i \in$ *sorte*$(k)$*, dann schreiben wir auch* $k :: s_i$*.*

**Definition 3.2** (Syntax der Rätsellogik)**.** *Sei* $V$ *eine abzählbar-unendliche Menge von Variablen. Ein* Atom $A$ *über einer Signatur* $(K, S, P)$ *ist ein Ausdruck der Form* $p(x_1, ..., x_n)$*, wenn* $p \in P$ *ein* $n$*-stelliges Prädikat ist und für alle* $i$ *gilt:* $x_i \in V \vee x_i \in K$

Formeln $F$ *über einer Signatur* $(K, S, P)$ *können wie folgt gebildet werden:*

$$F ::= A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid F_1 \Rightarrow F_2 \mid F_1 \Leftrightarrow F_2 \mid \forall x \in s.F \mid \exists x \in s.F$$

### 3.1.2 Wohlgetyptheit von Formeln

Sei $\Gamma$ eine Umgebung, die Variablen eine Sorte zuordnet (geschrieben als $x :: s$, wenn $x \in V$ und $s \in S$). Wir schreiben $\Gamma(x_i) = s_i$, wenn $x :: s_i \in \Gamma$.

**Definition 3.3** (Typisierung)**.** *Eine Formel* $F$ *ist* wohl-getypt *gdw. sich* $\emptyset \vdash F$ *mit den Typisierungsregeln in Abbildung 3.1 herleiten lässt:*

$$\frac{p :: s_1 \to \ldots \to s_n \to \{\texttt{True}, \texttt{False}\} \quad \forall x_i \in V : \Gamma(x_i) = s_i \quad \forall x_i \in K : x_i :: s_i}{\Gamma \vdash p(x_1, \ldots, x_n)}$$

$$\frac{\Gamma \cup \{x :: s\} \vdash F}{\Gamma \vdash \exists x \in s.F} \qquad \frac{\Gamma \cup \{x :: s\} \vdash F}{\Gamma \vdash \forall x \in s.F} \qquad \frac{\Gamma \vdash F}{\Gamma \vdash \neg F}$$

$$\frac{\Gamma \vdash F_1 \quad \Gamma \vdash F_2}{\Gamma \vdash F_1 \otimes F_2} \text{ für } \otimes \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$$

Abbildung 3.1: Typisierungsregeln für die Rätsellogik

### 3.1.3 Semantik der Rätsellogik

Für eine Formel $F$ schreiben wir $F[k/x]$ für die Ersetzung aller freien Vorkommen der Variablen $x$ durch die Konstante $k$.

Sei $F$ eine wohl-getypte Formel der Rätsellogik über einer Signatur $(K, S, P)$ und einer Menge von Variablen $x$.

**Definition 3.4** (Interpretation)**.** *Eine* Interpretation $I$ *bildet jedes Prädikat* $p \in P$ *mit Stelligkeit* $n$ *auf eine Menge von* $n$*-Tupeln typgerecht ab, d.h.* $I(p) \subseteq (\mathit{konstanten}(s_1) \times \ldots \times \mathit{konstanten}(s_n))$*, wenn* $p :: s_1 \to \ldots \to s_n \to \{\texttt{True}, \texttt{False}\}$.

*Die Erweiterung einer Interpretation auf wohl-getypte Formeln* $F$ *ist definiert durch:*

$$I(p(k_1, \ldots, k_n)) = \begin{cases} \texttt{True}, & \textit{gdw. } (k_1, \ldots, k_n) \in I(p) \\ \texttt{False}, & \textit{sonst} \end{cases}$$

$$I(\neg F) = \begin{cases} \texttt{True}, & \textit{gdw. } I(F) = \texttt{False} \\ \texttt{False}, & \textit{gdw. } I(F) = \texttt{True} \end{cases}$$

$$I(F_1 \wedge F_2) = \begin{cases} \texttt{True}, & \textit{gdw. } I(F_1) = \texttt{True} \textit{ und } I(F_2) = \texttt{True} \\ \texttt{False}, & \textit{sonst} \end{cases}$$

$$I(F_1 \vee F_2) = \begin{cases} \texttt{True}, & \textit{gdw. } I(F_1) = \texttt{True} \textit{ oder } I(F_2) = \texttt{True} \\ \texttt{False}, & \textit{sonst} \end{cases}$$

$$I(F_1 \Rightarrow F_2) = \begin{cases} \texttt{True}, & \textit{gdw. } I(F_1) = \texttt{False} \textit{ oder } I(F_2) = \texttt{True} \\ \texttt{False}, & \textit{sonst} \end{cases}$$

$$I(F_1 \Leftrightarrow F_2) = \begin{cases} \texttt{True}, & \textit{gdw. } I(F_1) = I(F_2) \\ \texttt{False}, & \textit{sonst} \end{cases}$$

$$I(\exists x \in s.F) = \begin{cases} \texttt{True}, & \textit{gdw. es } k \in \mathit{konstanten}(s) \textit{ gibt mit } I(F[k/x]) = \texttt{True} \\ \texttt{False}, & \textit{sonst} \end{cases}$$

$$I(\forall x \in s.F) = \begin{cases} \texttt{True}, & \textit{gdw. für alle } k \in \mathit{konstanten}(s) \textit{ gilt: } I(F[k/x]) = \texttt{True} \\ \texttt{False}, & \textit{sonst} \end{cases}$$

**Definition 3.5.** *Eine wohl-getypte Rätselformel* $F$ *ist*

- allgemeingültig *genau dann, wenn für alle Interpretationen* $I$ *gilt:* $I(F) =$ `True`.
- erfüllbar *genau dann, wenn es eine Interpretation* $I$ *gibt mit:* $I(F) =$ `True`.
- falsifizierbar *genau dann, wenn es eine Interpretation* $I$ *gibt mit* $I(F) =$ `False`.
- widersprüchlich *genau dann, wenn für alle Interpretationen* $I$ *gilt:* $I(F) =$ `False`.

*Wenn* $I(F) =$ `True` *gilt, so nennt man* $I$ *ein* Modell *für* $F$*.*

Betrachten wir nun ein Beispiel, um eine solche Formel $F$ in Rätsellogik mit einer Signatur $(K, S, P)$ zu sehen.

**Beispiel 3.6.** *Sei ein Freundesrätsel gegeben der folgenden Form:*

| |
|---|
| *Elly und Vanessa sind Blondinen.* |
| *Klaudia und Louisa sind Brünetten.* |
| *Alle jungen Blondinen sind mit einer Brünetten befreundet.* |
| *Vanessa und Elly sind jung.* |
| *Vanessa ist nicht mit Klaudia befreundet.* |
| *Elly ist nicht mit Louisa befreundet.* |
| *Wer ist mit wem befreundet?* |

Tabelle 3.1: Freundesrätsel

*Sei F die dazugehörige Formel in Rätsellogik. Sei (K, S, P) die Signatur über der Formel F mit:*

$K = \{Elly,\ Vanessa,\ Klaudia,\ Louisa\}$
$S = \{Blondinen,\ Brünetten\}$
$Blondinen = \{Elly,\ Vanessa\}$
$Brünetten = \{Klaudia,\ Louisa\}$
$P = \{jung(x),\ befreundet(x,y)\}$

$F = \forall\ x \in Blondinen\ \exists\ y \in Brünetten\ (jung(x) \Rightarrow befreundet(x,y) \wedge jung(Vanessa) \wedge jung(Elly) \wedge \neg\ (befreundet(Vanessa,Klaudia)) \wedge \neg\ (befreundet(Elly,Louisa)))$

### 3.1.4 Anzahlbeschränkte Existenzquantoren

Abgesehen von den in 3.1.1 vorgestellten Quantoren gibt es auch noch die anzahlbeschränkten Existenzquantoren. Da diese anzahlbeschränkten Existenzquantoren für unsere Rätsellogik aber nicht zwingend notwendig sind, betrachten wir sie hier in diesem Abschnitt gesondert.
Der gewöhnliche Existenzquantor stellt die Bedingung, dass *mindestens ein Element* wahr sein muss, um die Formel zu erfüllen. Im Folgenden betrachten wir drei weitere Existenzquantoren mit anderen Bedingungen.

- $\exists \geq n$ = **ExAtLeast n**: Alle Teilmengen der Mächtigkeit $n$ als Klauseln, und die Klauseln werden verodert. D.h. mindestens einmal sind $n$ wahr, es können aber auch mehr wahr sein.

- $\exists = n$ = **ExExactly n**: Wie ExAtLeast aber die Klauseln sind ver-XOR-t, d.h. genau eine der n-elementigen Teilmengen ist wahr und alle anderen falsch, da eine Teilmenge der Mächtigkeit genau $n$ wahr sein muss.

- $\exists \leq n$ **ExAtMost n**: Betrachte alle Teilmengen der Größe n+1: Jede dieser Teilmengen muss eine falsche Formel beinhalten. Dies lässt sich mit einer negierten Konjunktion über alle Teilmengen der Größe n+1 darstellen.

Die Interpretation der anzahlbeschränkten Existenzquantoren ist formal definiert als:

**Definition 3.7** (Interpretation der anzahlbeschränkten Existenzquantoren)**.**

$$I(\exists \geq n\ x \in s.F) \quad = \quad \begin{cases} \texttt{True}, & \textit{gdw. es } \{k_1, \ldots, k_n\} \subseteq K(s) \textit{ gibt mit} \\ & |\{k_1, \ldots, k_n\}| \geq n \textit{ und } I(F[k_i/x]) = \texttt{True} \\ \texttt{False}, & \textit{sonst} \end{cases}$$

$$I(\exists = n\ x \in s.F) \quad = \quad \begin{cases} \texttt{True}, & \textit{gdw. es } \{k_1, \ldots, k_n\} \subseteq K(s) \textit{ gibt mit} \\ & |\{k_1, \ldots, k_n\}| = n \textit{ und } I(F[k_i/x]) = \texttt{True} \\ \texttt{False}, & \textit{sonst} \end{cases}$$

$$I(\exists \leq n\ x \in s.F) \quad = \quad \begin{cases} \texttt{True}, & \textit{gdw. für alle } \{k_1, \ldots, k_{n+1}\} \subseteq K(s) \textit{ mit} \\ & |\{k_1, \ldots, k_{n+1}\}| = n+1 \textit{ gilt: } I(F[k_i/x]) = \texttt{False} \\ \texttt{False}, & \textit{sonst} \end{cases}$$

**Definition 3.8** (Erweiterte Typisierungsregeln)**.** *Für die anzahlbeschränkten Existenzquantoren gelten die in Abbildung 3.2 gezeigten Typisierungsregeln:*

$$\frac{\Gamma \vdash \exists x \in s.F}{\Gamma \vdash \exists \geq n\ x \in s.F} \qquad \frac{\Gamma \vdash \exists x \in s.F}{\Gamma \vdash \exists = n\ x \in s.F} \qquad \frac{\Gamma \vdash \exists x \in s.F}{\Gamma \vdash \exists \leq n\ x \in s.F}$$

Abbildung 3.2: Typisierungsregeln für die anzahlbeschränkten Existenzquantoren

Und die Übersetzung $[\![F]\!]$ einer Rätselformel $F$ mit einem der anzahlbeschränkten Existenzquantoren in eine aussagenlogische Formel ist wie folgt definiert:

**Definition 3.9** (Übersetzung der Existenzquantoren in Aussagenlogik)**.**

$$[\![\exists \geq n\ x \in s.F]\!] \quad := \quad \bigvee_{\substack{M \subseteq K(s),\\ |M|=n}} \left( \bigwedge_{k_i \in M} [\![F[k_i/x]]\!] \right)$$

$$[\![\exists = n\ x \in s.F]\!] \quad := \quad \bigvee_{\substack{M \subseteq K(s),\\ |M|=n}} \left( \bigwedge_{k_i \in M} [\![F[k_i/x]]\!] \right)$$

$$[\![\exists \leq n\ x \in s.F]\!] \quad := \quad \bigwedge_{\substack{M \subseteq K(s),\\ |M|=n+1}} \left( \neg \bigwedge_{k_i \in M} [\![F[k_i/x]]\!] \right)$$

## 3.2 Transformation der Rätsellogik in Aussagenlogik

In diesem Abschnitt wollen wir die Transformation der Rätsellogik in Aussagenlogik betrachten. Dabei stellt sich aber zunächst die Frage: Warum wollen wir das überhaupt?
Warum benutzen wir nicht nur die Prädikatenlogik und versuchen eine prädikatenlogische Formel einem Programm oder einem angepassten SAT-Solver zu übergeben und so zu lösen, um uns den Zwischenschritt der Transformation in Aussagenlogik zu ersparen?
Die Antwort liegt in der Komplexität der beiden Fragestellungen:
„Ist eine aussagenlogische Formel $F$ erfüllbar?" und
„Ist eine prädikatenlogische Formel $F$ erfüllbar?"

Daher betrachten wir im folgenden Abschnitt die Komplexität dieser beiden Fragestellungen.

### 3.2.1 Komplexität

**Zur Aussagenlogik:**[1]

- Es ist entscheidbar, ob eine gegebene Aussage eine Tautologie / ein Widerspruch / erfüllbar ist. Entscheidbar bedeutet, dass es einen Algorithmus gibt, der für jede Eingabe terminiert und für jedes Element der Menge beantworten kann, ob es eine bestimmte Eigenschaft besitzt oder nicht.
- Die Frage, ob eine Aussage erfüllbar ist, ist NP-vollständig.

[1] Siehe [SS12], Satz 2.7.

- Die Frage, ob eine Aussage eine Tautologie / ein Widerspruch ist, ist co-NP-vollständig.

Um eine Lösung zu diesen Fragen zu erhalten, kann man als naiven Ansatz Wahrheitstafeln verwenden und schlichtweg jede Interpretation ausprobieren. Dieser Lösungsweg bedarf exponentielle Zeit (exponentiell in der Anzahl der Variablen).

**Zur Prädikatenlogik:**[2]

- Es ist unentscheidbar, ob eine geschlossene Formel ein Satz der Prädikatenlogik ist.

Wir sehen also, dass es einen Unterschied in der Komplexität macht, ob wir unser Problem in Aussagenlogik oder in Prädikatenlogik vorliegen haben.
Während die Erfüllbarkeit in der Prädikatenlogik unentscheidbar ist, so gilt dies nicht für die getypte Prädikatenlogik auf endlichen Mengen, denn das Erfüllbarkeitsproblem für Rätselformeln kann in die Aussagenlogik transformiert werden. Diese Transformation werden wir im nächsten Abschnitt definieren und anschließend die Korrektheit und Vollständigkeit der Transformation zeigen.

[2]Siehe: [SS12], Satz 4.24.

### 3.2.2 Transformation der Rätsellogik in die Aussagenlogik

Wir definieren die Übersetzung $[\![\cdot]\!]$ von wohl-getypten Formeln $F$ in aussagenlogische Formeln formal.

**Definition 3.10** (Übersetzung $[\![\cdot]\!]$)**.** *Sei $F$ ein wohl-getypte Formel über einer Signatur $(K, P, S)$. Erzeuge für jedes Atom $p(k_1, \ldots, k_n)$ (mit $k_i \in K$ und $p \in P$) eine aussagenlogische Variable $X_{p(k_1,\ldots,k_n)}$.*

*Die Übersetzung $[\![F]\!]$ ist induktiv definiert durch die folgenden Fälle:*

$$
\begin{aligned}
[\![p(k_1, \ldots, k_n)]\!] &:= X_{p(k_1,\ldots,k_n)} \\
[\![\neg F]\!] &:= \neg [\![F]\!] \\
[\![F_1 \otimes F_2]\!] &:= [\![F_1]\!] \otimes [\![F_2]\!] \text{ für } \otimes \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\} \\
[\![\exists x \in s.F]\!] &:= \bigvee_{k \in konstanten(s)} [\![F[k/x]]\!] \\
[\![\forall x \in s.F]\!] &:= \bigwedge_{k \in konstanten(s)} [\![F[k/x]]\!]
\end{aligned}
$$

Veranschaulichen wir die Übersetzung $[\![\cdot]\!]$ anhand eines Beispiels. Betrachten wir dafür die Formel aus 3.6.

**Beispiel 3.11.** *Sei die Formel F in Rätsellogik wieder unser Beispiel aus 3.6 mit der Signatur (K, S, P):*

*K = {Elly, Vanessa, Klaudia, Louisa}*
*S = {Blondinen, Brünetten}*
*Blondinen = {Elly, Vanessa}*
*Brünetten = {Klaudia, Louisa}*
*P = {jung(x), befreundet(x,y)}*

*F = ∀ x ∈Blondinen ∃ y ∈Brünetten (jung(x) ⇒ befreundet(x,y) ∧ jung(Vanessa) ∧ jung(Elly) ∧ ¬ (befreundet(Vanessa,Klaudia)) ∧ ¬ (befreundet(Elly,Louisa)))*

*Betrachten wir dafür die Übersetzung $[\![F]\!]$ in den Tabellen 3.2 und 3.3.*

| *Formel* | *Übersetzung* |
|---|---|
| $[\![\forall x \in Blondinen\ \exists y \in Brünetten\ ((jung(x) \Rightarrow befreundet(x,y)) \wedge jung(Vanessa) \wedge jung(Elly) \wedge \neg befreundet(Vanessa,Klaudia) \wedge \neg befreundet(Elly,Louisa))]\!]$ | $[\![\forall x \in s.F]\!] := \bigwedge_{k \in K(s)} [\![F[k/x]]\!]$ |
| $[\![\exists y \in Brünetten\ ((jung(Elly) \Rightarrow befreundet(Elly,y)) \wedge jung(Vanessa) \wedge jung(Elly) \wedge \neg befreundet(Vanessa,Klaudia) \wedge \neg befreundet(Elly,Louisa))]\!] \wedge [\![\exists y \in Brünetten\ ((jung(Vanessa) \Rightarrow befreundet(Vanessa,y)) \wedge jung(Vanessa) \wedge jung(Elly) \wedge \neg befreundet(Vanessa,Klaudia) \wedge \neg befreundet(Elly,Louisa))]\!]$ | $[\![\exists x \in s.F]\!] := \bigvee_{k \in K(s)} [\![F[k/x]]\!]$ |
| $([\![((jung(Elly) \Rightarrow befreundet(Elly,Klaudia)) \wedge jung(Vanessa) \wedge jung(Elly) \wedge \neg befreundet(Vanessa,Klaudia) \wedge \neg befreundet(Elly,Louisa))]\!] \vee ([\![((jung(Elly) \Rightarrow befreundet(Elly,Louisa)) \wedge jung(Vanessa) \wedge jung(Elly) \wedge \neg befreundet(Vanessa,Klaudia) \wedge \neg befreundet(Elly,Louisa))]\!]))$ $\wedge ([\![((jung(Vanessa) \Rightarrow befreundet(Vanessa,Klaudia)) \wedge jung(Vanessa) \wedge jung(Elly) \wedge \neg befreundet(Vanessa,Klaudia) \wedge \neg befreundet(Elly,Louisa))]\!] \vee [\![((jung(Vanessa) \Rightarrow befreundet(Vanessa,Louisa)) \wedge jung(Vanessa) \wedge jung(Elly) \wedge \neg befreundet(Vanessa,Klaudia) \wedge \neg befreundet(Elly,Louisa))]\!])$ | $[\![F_1 \wedge F_2]\!] := [\![F_1]\!] \wedge [\![F_2]\!]$ |
| $([\![((jung(Elly) \Rightarrow befreundet(Elly,Klaudia))]\!] \wedge [\![jung(Vanessa) \wedge jung(Elly) \wedge \neg befreundet(Vanessa,Klaudia) \wedge \neg befreundet(Elly,Louisa))]\!] \vee ([\![((jung(Elly) \Rightarrow befreundet(Elly,Louisa))]\!] \wedge [\![jung(Vanessa) \wedge jung(Elly) \wedge \neg befreundet(Vanessa,Klaudia) \wedge \neg befreundet(Elly,Louisa))]\!]))$ $\wedge ([\![((jung(Vanessa) \Rightarrow befreundet(Vanessa,Klaudia))]\!] \wedge [\![jung(Vanessa) \wedge jung(Elly) \wedge \neg befreundet(Vanessa,Klaudia) \wedge \neg befreundet(Elly,Louisa))]\!] \vee [\![((jung(Vanessa) \Rightarrow befreundet(Vanessa,Louisa))]\!] \wedge [\![jung(Vanessa) \wedge jung(Elly) \wedge \neg befreundet(Vanessa,Klaudia) \wedge \neg befreundet(Elly,Louisa))]\!])$ | $[\![F_1 \Rightarrow F_2]\!] := [\![F_1]\!] \Rightarrow [\![F_2]\!]$ |

Tabelle 3.2: Beispiel Übersetzung Teil 1

| *Formel* | *Übersetzung* |
|---|---|
| $(((\llbracket jung(Elly) \rrbracket \Rightarrow \llbracket befreundet(Elly,Klaudia) \rrbracket \wedge \llbracket jung(Vanessa) \rrbracket \wedge \llbracket jung(Elly) \rrbracket \wedge \llbracket \neg\ befreundet(Vanessa,Klaudia) \rrbracket \wedge \llbracket \neg befreundet(Elly,Louisa) \rrbracket) \vee (((\llbracket jung(Elly) \rrbracket \Rightarrow \llbracket befreundet(Elly,Louisa) \rrbracket \wedge \llbracket jung(Vanessa) \rrbracket \wedge \llbracket jung(Elly) \rrbracket \wedge \llbracket \neg\ befreundet(Vanessa,Klaudia) \rrbracket \wedge \llbracket \neg befreundet(Elly,Louisa) \rrbracket))) \wedge (((\llbracket jung(Vanessa) \rrbracket \Rightarrow \llbracket befreundet(Vanessa,Klaudia) \rrbracket \wedge \llbracket jung(Vanessa) \rrbracket \wedge \llbracket jung(Elly) \rrbracket \wedge \llbracket \neg\ befreundet(Vanessa,Klaudia) \rrbracket \wedge \llbracket \neg befreundet(Elly,Louisa) \rrbracket) \vee (((\llbracket jung(Vanessa) \rrbracket \Rightarrow \llbracket befreundet(Vanessa,Louisa) \rrbracket \wedge \llbracket jung(Vanessa) \rrbracket \wedge \llbracket jung(Elly) \rrbracket \wedge \llbracket \neg\ befreundet(Vanessa,Klaudia) \rrbracket \wedge \llbracket \neg befreundet(Elly,Louisa) \rrbracket)))$ | $\llbracket \neg F \rrbracket := \neg \llbracket F \rrbracket$ |
| $((\llbracket jung(Elly) \rrbracket \Rightarrow \llbracket befreundet(Elly,Klaudia) \rrbracket \wedge \llbracket jung(Vanessa) \rrbracket \wedge \llbracket jung(Elly) \rrbracket \wedge \neg \llbracket\ befreundet(Vanessa,Klaudia) \rrbracket \wedge \neg \llbracket befreundet(Elly,Louisa) \rrbracket) \vee ((\llbracket jung(Elly) \rrbracket \Rightarrow \llbracket befreundet(Elly,Louisa) \rrbracket \wedge \llbracket jung(Vanessa) \rrbracket \wedge \llbracket jung(Elly) \rrbracket \wedge \neg \llbracket\ befreundet(Vanessa,Klaudia) \rrbracket \wedge \neg \llbracket befreundet(Elly,Louisa) \rrbracket))) \wedge (((\llbracket jung(Vanessa) \rrbracket \Rightarrow \llbracket befreundet(Vanessa,Klaudia) \rrbracket \wedge \llbracket jung(Vanessa) \rrbracket \wedge \llbracket jung(Elly) \rrbracket \wedge \neg \llbracket\ befreundet(Vanessa,Klaudia) \rrbracket \wedge \neg \llbracket befreundet(Elly,Louisa) \rrbracket) \vee (((\llbracket jung(Vanessa) \rrbracket \Rightarrow \llbracket befreundet(Vanessa,Louisa) \rrbracket \wedge \llbracket jung(Vanessa) \rrbracket \wedge \llbracket jung(Elly) \rrbracket \wedge \neg \llbracket\ befreundet(Vanessa,Klaudia) \rrbracket \wedge \neg \llbracket befreundet(Elly,Louisa) \rrbracket)))$ | $\llbracket p(k_1, \ldots, k_n) \rrbracket := X_{p(k_1,\ldots,k_n)}$ |
| $(((X_{jung(Elly)} \Rightarrow X_{befreundet(Elly,Klaudia)}) \wedge X_{jung(Vanessa)} \wedge X_{jung(Elly)} \wedge \neg\ X_{befreundet(Vanessa,Klaudia)} \wedge \neg\ X_{befreundet(Elly,Louisa)}) \vee ((X_{jung(Elly)} \Rightarrow X_{befreundet(Elly,Louisa)}) \wedge X_{jung(Vanessa)} \wedge X_{jung(Elly)} \wedge \neg\ X_{befreundet(Vanessa,Klaudia)} \wedge \neg\ X_{befreundet(Elly,Louisa)})) \wedge (((X_{jung(Vanessa)} \Rightarrow X_{befreundet(Vanessa,Klaudia)}) \wedge X_{jung(Vanessa)} \wedge X_{jung(Elly)} \wedge \neg\ X_{befreundet(Vanessa,Klaudia)} \wedge \neg\ X_{befreundet(Elly,Louisa)}) \vee ((X_{jung(Vanessa)} \Rightarrow X_{befreundet(Vanessa,Louisa)}) \wedge X_{jung(Vanessa)} \wedge X_{jung(Elly)} \wedge \neg\ X_{befreundet(Vanessa,Klaudia)} \wedge \neg\ X_{befreundet(Elly,Louisa)}))$ | |

Tabelle 3.3: Beispiel Übersetzung Teil 2

### 3.2.3 Korrektheit und Vollständigkeit der Transformation

Wir definieren die Größe size($F$) für Rätselformeln $F$:

**Definition 3.12.** *Sei $F$ eine wohl-getypte Rätselformel. Dann ist* size$(F) \geq 1$ *definiert als:*

$$\begin{array}{lcl} \text{size}(p(x_1,\ldots,x_n)) & := & 1 \\ \text{size}(F_1 \otimes F_2) & := & 1 + \text{size}(F_1) + \text{size}(F_2) \quad \textit{für } \otimes \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\} \\ \text{size}(\exists x \in s.F) & := & 1 + \text{size}(F) \\ \text{size}(\forall x \in s.F) & := & 1 + \text{size}(F) \end{array}$$

**Lemma 3.13.** *Sei $F$ eine wohl-getypte Formel der Rätsellogik und $I$ eine Interpretation von $F$. Sei die aussagenlogische Interpretation $I'$ definiert als $I'(X_{p,k_1,\ldots,k_n}) := I(p(k_1,\ldots,k_n))$. Dann gilt $I(F) = I'([\![F]\!])$.*

*Beweis.* Wir verwenden Induktion über size($F$).

**Induktionsbasis:** size$(F) = 1$. Dann ist $F = p(x_1,\ldots,x_n)$. Da die Übersetzung $[\![\cdot]\!]$ nur solche Atome übersetzt, für die alle $x_i \in$ *konstanten* sind, genügt es, diesen Fall zu betrachten. Offensichtlich gilt $I'([\![p(k_1,\ldots,k_n)]\!]) = I'(X_{p,k_1,\ldots,k_n}) = I(p(k_1,\ldots,k_n))$.

**Induktionsannahme:** Die Aussage aus Lemma 3.13 gilt für alle Formeln $F$ mit size$(F) = n$.

**Induktionsschritt:** Sei $F$ eine Formel mit size$(F) = n+1$. Wir führen eine Fallunterscheidung nach der Struktur von $F$ durch:

- Fall $F = \neg F'$: Da size$(F') = n$, darf die Induktionsannahme für $F'$ verwendet werden. Sie liefert $I(F') = I'([\![F']\!])$. Wir unterscheiden zwei Fälle:
  - $I(F) = 0$. Dann gilt $I(F') = 1$ und daher auch $I'([\![F']\!]) = 1$. Nun können wir schließen $I'([\![F]\!]) = I'([\![\neg F']\!]) = I'(\neg[\![F']\!]) = 0$, da $I'([\![F']\!]) = 1$.
  - $I(F) = 1$. Dann gilt $I(F') = 0$ und daher auch $I'([\![F']\!]) = 0$. Nun können wir schließen $I'([\![F]\!]) = I'([\![\neg F']\!]) = I'(\neg[\![F']\!]) = 1$, da $I'([\![F']\!]) = 0$.

  In beiden Fällen haben wir daher $I(F) = I'([\![F]\!])$ gezeigt.
- Fall $F = F_1 \otimes F_2$: Die Induktionsannahme kann für $F_1$ und für $F_2$ verwendet werden, da size$(\mathrm{F_i}) \leq n$ für $i = 1, 2$. Daher gilt $I(F_i) = I'([\![F_i]\!])$ für $i = 1, 2$.

  Wir betrachten alle Fälle:
  - $\otimes = \wedge$ und $I(F) = 0$. Da $I(F_1 \wedge F_2) = 0$ gibt es ein $F_i$ (mit $i = 1, 2$) mit $I(F_i) = 0$. Die Induktionsannahme liefert $I'([\![F_i]\!]) = 0$. Nun können wir schließen: $I'([\![F]\!]) = I'([\![F_1 \wedge F_2]\!]) = I'([\![F_1]\!] \wedge [\![F_2]\!]) = 0$, da $I'([\![F_i]\!]) = 0$.
  - $\otimes = \wedge$ und $I(F) = 1$. Da $I(F_1 \wedge F_2) = 1$ gilt $I(F_i) = 1$ für $i = 1, 2$. Die Induktionsannahme liefert $I'([\![F_1]\!]) = I'([\![F_2]\!]) = 1$. Nun können wir schließen: $I'([\![F]\!]) = I'([\![F_1 \wedge F_2]\!]) = I'([\![F_1]\!] \wedge [\![F_2]\!]) = 1$, da $I'([\![F_1]\!]) = 1$ und $I'([\![F_2]\!]) = 1$.

- $\otimes = \vee$ und $I(F) = 0$. Da $I(F_1 \vee F_2) = 0$ gilt $I(F_i) = 0$ für $i = 1, 2$. Die Induktionsannahme liefert $I'([\![F_1]\!]) = I'([\![F_2]\!]) = 0$. Nun können wir schließen: $I'([\![F]\!]) = I'([\![F_1 \vee F_2]\!]) = I'([\![F_1]\!] \vee [\![F_2]\!]) = 0$, da $I'([\![F_1]\!]) = 0$ und $I'([\![F_2]\!]) = 0$.
- $\otimes = \vee$ und $I(F) = 1$. Da $I(F_1 \vee F_2) = 1$ gibt es ein $F_i$ (mit $i = 1, 2$) mit $I(F_i) = 1$. Die Induktionsannahme liefert $I'([\![F_i]\!]) = 1$. Nun können wir schließen: $I'([\![F]\!]) = I'([\![F_1 \vee F_2]\!]) = I'([\![F_1]\!] \vee [\![F_2]\!]) = 1$, da $I'([\![F_i]\!]) = 1$.
- $\otimes = \Rightarrow$ und $I(F) = 0$. Da $I(F_1 \Rightarrow F_2) = 0$ gilt $I(F_1) = 1$ und $I(F_2) = 0$. Die Induktionsannahme liefert $I'([\![F_1]\!]) = 1$ und $I'([\![F_2]\!]) = 0$. Nun können wir schließen: $I'([\![F]\!]) = I'([\![F_1 \Rightarrow F_2]\!]) = I'([\![F_1]\!] \Rightarrow [\![F_2]\!]) = 0$, da $I'([\![F_1]\!]) = 1$ und $I'([\![F_2]\!]) = 0$.
- $\otimes = \Rightarrow$ und $I(F) = 1$. Da $I(F_1 \Rightarrow F_2) = 1$ gilt $I(F_1) = 0$ oder $I(F_2) = 1$. Die Induktionsannahme liefert $I'([\![F_1]\!]) = 0$ oder $I'([\![F_2]\!]) = 1$. Nun können wir schließen: $I'([\![F]\!]) = I'([\![F_1 \Rightarrow F_2]\!]) = I'([\![F_1]\!] \Rightarrow [\![F_2]\!]) = 1$, da $I'([\![F_1]\!]) = 0$ oder $I'([\![F_2]\!]) = 1$.
- $\otimes = \Leftrightarrow$ und $I(F) = 0$. Da $I(F_1 \Leftrightarrow F_2) = 0$ gilt $I(F_1) \neq I(F_2)$. Die Induktionsannahme liefert $I'([\![F_1]\!]) \neq I'([\![F_2]\!])$. Nun können wir schließen: $I'([\![F]\!]) = I'([\![F_1 \Leftrightarrow F_2]\!]) = I'([\![F_1]\!] \Leftrightarrow [\![F_2]\!]) = 1$, da $I'([\![F_1]\!]) \neq I'([\![F_2]\!])$.
- $\otimes = \Leftrightarrow$ und $I(F) = 1$. Da $I(F_1 \Leftrightarrow F_2) = 1$ gilt $I(F_1) = I(F_2)$. Die Induktionsannahme liefert $I'([\![F_1]\!]) = I'([\![F_2]\!])$. Nun können wir schließen: $I'([\![F]\!]) = I'([\![F_1 \Leftrightarrow F_2]\!]) = I'([\![F_1]\!] \Leftrightarrow [\![F_2]\!]) = 1$, da $I'([\![F_1]\!]) = I'([\![F_2]\!])$.

- Fall $F = \exists x \in s.F'$. Dann gilt $\text{size}(F'[k/x]) = n$ für jede Konstante $k$, und daher kann die Induktionsannahme für jede Formel $F[k/x]$ verwendet werden. Die Induktionsannahme liefert dabei $I(F'[k/x]) = I'([\![F'[k/x]]\!])$. Wir betrachten zwei Fälle:
  - $I(F) = 0$. Dann gilt für alle Konstanten $k \in \mathit{konstanten}(s)$ die Gleichung $I(F'[k/x]) = 0$ und die Induktionsannahme liefert für alle $k \in \mathit{konstanten}(s)$: $I'([\![F'[k/x]]\!]) = 0$. Nun können wir schließen: $I'([\![\exists x \in s.F']\!]) = I'(\bigvee_{k \in \mathit{konstanten}(s)} [\![F'[k/x]]\!]) = 0$, da $I'([\![F'[k/x]]\!]) = 0$ für alle $k \in \mathit{konstanten}(s)$.
  - $I(F) = 1$. Dann existiert eine Konstante $k_0 \in \mathit{konstanten}(s)$ mit $I(F'[k_0/x]) = 1$ und die Induktionsannahme liefert $I'([\![F'[k_0/x]]\!]) = 1$. Nun können wir schließen: $I'([\![\exists x \in s.F']\!]) = I'(\bigvee_{k \in \mathit{konstanten}(s)} [\![F'[k/x]]\!]) = 1$, da $I'([\![F'[k_0/x]]\!]) = 1$ und $k_0 \in \mathit{konstanten}(s)$.
- Fall $F = \forall x \in s.F'$. Dann gilt $\text{size}(F'[k/x]) = n$ für jede Konstante $k$, und daher kann die Induktionsannahme für jede Formel $F[k/x]$ verwendet werden. Die Induktionsannahme liefert dabei $I(F'[k/x]) = I'([\![F'[k/x]]\!])$. Wir betrachten zwei Fälle:
  - $I(F) = 0$. Dann gibt es eine Konstante $k_0 \in \mathit{konstanten}(s)$ mit $I(F'[k_0/x]) = 0$ und die Induktionsannahme liefert

$I'([\![F'[k_0/x]]\!]) = 0$. Nun können wir schließen: $I'([\![\forall x \in s.F']\!]) = I'(\bigwedge_{k \in konstanten(s)} [\![F'[k/x]]\!]) = 0$, da $I'([\![F'[k_0/x]]\!]) = 0$ und $k_0 \in konstanten(s)$.

- $I(F) = 1$. Dann gilt für alle Konstanten $k \in konstanten(s)$ die Gleichung $I(F'[k/x]) = 1$ und die Induktionsannahme liefert $I'([\![F'[k/x]]\!]) = 1$ für alle $k \in konstanten(s)$. Nun können wir schließen: $I'([\![\forall x \in s.F']\!]) = I'(\bigwedge_{k \in konstanten(s)} [\![F'[k/x]]\!]) = 1$, da $I'([\![F'[k/x]]\!]) = 1$ für alle $k \in konstanten(s)$.

□

**Korollar 3.14.** *Wenn eine wohl-getypte Rätselformel $F$ erfüllbar ist, dann ist auch $[\![F]\!]$ erfüllbar.*

*Beweis.* Das vorherige Lemma 3.13 zeigt, dass für jedes Modell $I$ von $F$, die Interpretation $I'$ ein Modell für $[\![F]\!]$ ist. □

**Lemma 3.15.** *Sei $F$ eine wohlgetypte Rätselformel und $I$ eine (aussagenlogische) Interpretation für $[\![F]\!]$. Sei $I'$ die Interpretation der Rätsellogik definiert durch $(k_1, \ldots, k_n) \in I'(p)$ genau dann, wenn $I(X_{p,k_1,\ldots,k_n}) = 1$. Dann gilt $I'(F) = I([\![F]\!])$.*

*Beweis.* Wir verwenden Induktion über size($F$).

**Induktionsbasis:** size($F$) = 1. Dann ist $F$ ein Atom $p(x_1, \ldots, x_n)$ Wir betrachten nur wohl-getypte Formeln und daher geschlossene Formeln. Für die Berechnung der Interpretation einer Rätselformel genügt es daher, Atome der Form $p(k_1, \ldots, k_n)$, wobei alle $k_i$ Konstanten sind, zu betrachten. Für diese Atome gilt $I'(p(k_1, \ldots, k_n)) = 1$ gdw. $(k_1, \ldots, k_n) \in I(p)$. Da $I([\![p(k_1, \ldots, k_n)]\!]) = I(X_{p,k_1,\ldots,k_n})$ gilt die Aussage.

**Induktionsannahme:** Die Aussage aus Lemma 3.15 gilt für alle Formeln $F$ mit size($F$) = $n$.

**Induktionsschritt:** Sei $F$ eine Formel mit size($F$) = $n+1$. Wir führen eine Fallunterscheidung nach der Struktur von $F$ durch:

- Fall $F = \neg F'$: Da size($F'$) = $n$, darf die Induktionsannahme für $F'$ verwendet werden. Sie liefert $I'(F') = I([\![F']\!])$. Wir unterscheiden zwei Fälle:
  - $I([\![F]\!]) = 0$. Dann gilt $I([\![F']\!]) = 1$ und daher auch $I'(F') = 0$. Nun können wir schließen $I'(F) = I'(\neg F') = 0$, da $I'(F') = 1$.
  - $I([\![F]\!]) = 1$. Dann gilt $I([\![F']\!]) = 0$ und daher auch $I'(F') = 0$. Nun können wir schließen $I'(F) = I'(\neg F') = 1$, da $I'([\![F']\!]) = 0$.

  In beiden Fällen haben wir daher $I'(F) = I([\![F]\!])$ gezeigt.
- Fall $F = F_1 \otimes F_2$: Die Induktionsannahme kann für $F_1$ und für $F_2$ verwendet werden, da size($\mathrm{F_i}$) $\leq n$ für $i = 1, 2$. Daher gilt $I'(F_i) = I([\![F_i]\!])$ für $i = 1, 2$.

  Wir betrachten alle Fälle:

- $\otimes = \wedge$ und $I([\![F]\!]) = 0$. Da $I([\![F_1 \wedge F_2]\!]) = I([\![F_1]\!] \wedge [\![F_2]\!]) = 0$ gibt es ein $F_i$ (mit $i = 1, 2$) mit $I([\![F_i]\!]) = 0$. Die Induktionsannahme liefert $I'(F_i) = 0$. Nun können wir schließen: $I'(F) = I'(F_1 \wedge F_2) = 0$, da $I'(F_i) = 0$.
- $\otimes = \wedge$ und $I([\![F]\!]) = 1$. Da $I([\![F_1 \wedge F_2]\!]) = I([\![F_1]\!] \wedge [\![F_2]\!]) = 1$ gilt $I([\![F_i]\!]) = 1$ für $i = 1$ und $i = 2$. Die Induktionsannahme liefert $I'(F_i) = 1$ für $i = 1$ und $i = 2$. Nun können wir schließen: $I'(F) = I'(F_1 \wedge F_2) = 1$, da $I'(F_1) = 1$ und $I'(F_2) = 1$.
- $\otimes = \vee$ und $I([\![F]\!]) = 0$. Da $I([\![F_1 \vee F_2]\!]) = I([\![F_1]\!] \vee [\![F_2]\!]) = 0$ gilt $I([\![F_i]\!]) = 0$ für $i = 1$ und $i = 2$. Die Induktionsannahme liefert $I'(F_i) = 0$ für $i = 1$ und $i = 2$. Nun können wir schließen: $I'(F) = I'(F_1 \vee F_2) = 0$, da $I'(F_1) = 0$ und $I'(F_2) = 0$.
- $\otimes = \vee$ und $I([\![F]\!]) = 1$. Da $I([\![F_1 \vee F_2]\!]) = I([\![F_1]\!] \vee [\![F_2]\!]) = 1$ gibt es ein $F_i$ (mit $i = 1, 2$) mit $I([\![F_i]\!]) = 1$. Die Induktionsannahme liefert $I'(F_i) = 1$. Nun können wir schließen: $I'(F) = I'(F_1 \vee F_2) = 1$, da $I'(F_i) = 1$.
- $\otimes = \Rightarrow$ und $I([\![F]\!]) = 0$. Da $I([\![F_1 \Rightarrow F_2]\!]) = I([\![F_1]\!] \Rightarrow [\![F_2]\!]) = 0$ gilt $I([\![F_1]\!]) = 1$ und $I([\![F_2]\!]) = 0$. Die Induktionsannahme liefert $I'(F_1) = 1$ und $I'(F_2) = 0$. Nun können wir schließen: $I'(F) = I'(F_1 \Rightarrow F_2) = 0$, da $I'(F_1) = 1$ und $I'(F_2) = 0$.
- $\otimes = \Rightarrow$ und $I([\![F]\!]) = 1$. Da $I([\![F_1 \Rightarrow F_2]\!]) = I([\![F_1]\!] \Rightarrow [\![F_2]\!]) = 1$ gilt $I([\![F_1]\!]) = 0$ oder $I([\![F_2]\!]) = 1$. Die Induktionsannahme liefert $I'(F_1) = 0$ oder $I'(F_2) = 1$. Nun können wir schließen: $I'(F) = I'(F_1 \Rightarrow F_2) = 1$, da $I'(F_1) = 0$ oder $I'(F_2) = 1$.
- $\otimes = \Leftrightarrow$ und $I([\![F]\!]) = 0$. Da $I([\![F_1 \Leftrightarrow F_2]\!]) = I([\![F_1]\!] \Leftrightarrow [\![F_2]\!]) = 0$, gilt $I([\![F_1]\!]) \neq I([\![F_2]\!])$. Die Induktionsannahme liefert $I'(F_1) \neq I'(F_2)$. Nun können wir schließen: $I'(F) = I'(F_1 \Leftrightarrow F_2) = 0$, da $I'(F_1) \neq I'(F_2)$.
- $\otimes = \Leftrightarrow$ und $I([\![F]\!]) = 1$. Da $I([\![F_1 \Leftrightarrow F_2]\!]) = I([\![F_1]\!] \Leftrightarrow [\![F_2]\!]) = 1$ gilt $I([\![F_1]\!]) = I([\![F_2]\!])$. Die Induktionsannahme liefert $I'(F_1) = I'(F_2)$. Nun können wir schließen: $I'(F) = I'(F_1 \Leftrightarrow F_2) = 1$, da $I'(F_1) = I'(F_2)$.

- Fall $F = \exists x \in s.F'$. Dann gilt $\text{size}(F'[k/x]) = n$ für jede Konstante $k$, und daher kann die Induktionsannahme für jede Formel $F[k/x]$ verwendet werden. Die Induktionsannahme liefert dabei $I'(F'[k/x]) = I([\![F[k/x]]\!])$. Wir betrachten zwei Fälle:
  - $I([\![F]\!]) = 0$. Da $[\![F]\!] = \bigvee_{k \in konstanten(s)} [\![F'[k/x]]\!]$), gilt für alle $k \in konstanten(s)$: mit $I([\![F'[k/x]]\!]) = 0$. Die Induktionsannahme liefert $I'(F'[k/x]) = 0$ für alle und $k \in konstanten(s)$ daher können wir schließen: $I'(\exists x \in s.F') = 0$.
  - $I([\![F]\!]) = 1$. Da $[\![F]\!] = \bigvee_{k \in konstanten(s)} [\![F'[k/x]]\!]$), gibt es eine eine Konstante $k_0 \in konstanten(s)$ mit $I([\![F'[k_0/x]]\!]) = 1$. Die Induktionsannahme liefert $I'(F'[k_0/x]) = 1$ und daher können wir schließen: $I'(\exists x \in s.F') = 1$.

- Fall $F = \forall x \in s.F'$. Dann gilt size$(F'[k/x]) = n$ für jede Konstante $k$, und daher kann die Induktionsannahme für jede Formel $F[k/x]$ verwendet werden. Die Induktionsannahme liefert dabei $I(F'[k/x]) = I'([\![F'[k/x]]\!])$. Wir betrachten zwei Fälle:
  - $I([\![F]\!]) = 0$. Da $[\![F]\!] = \bigwedge_{k \in konstanten(s)} [\![F'[k/x]]\!]$), gibt es ein $k_0 \in konstanten(s)$: mit $I([\![F'[k_0/x]]\!]) = 0$. Die Induktionsannahme liefert $I'(F'[k_0/x]) = 0$ und daher können wir schließen: $I'(\forall x \in s.F') = 0$.
  - $I([\![F]\!]) = 1$. Da $[\![F]\!] = \bigwedge_{k \in konstanten(s)} [\![F'[k/x]]\!]$), gilt für alle $k \in konstanten(s)$: $I([\![F'[k/x]]\!]) = 1$. Die Induktionsannahme liefert $I'(F'[k/x]) = 1$ für alle $k \in konstanten(s)$ und daher können wir schließen: $I'(\forall x \in s.F') = 1$.

□

**Korollar 3.16.** *Sei $F$ eine wohl-getypte Rätselformel. Wenn $[\![F]\!]$ erfüllbar ist, dann ist auch $F$ erfüllbar.*

*Beweis.* Das vorherige Lemma 3.15 zeigt, dass für jedes Modell $I$ von $[\![F]\!]$, die Interpretation $I'$ ein Modell für $F$ ist. □

Aus den Korollaren 3.14 und 3.16 folgt:

**Satz 3.17.** *Sei $F$ eine wohl-getypte Formel der Rätsellogik. Dann gilt $[\![F]\!]$ ist (aussagenlogisch) erfüllbar genau dann, wenn $F$ erfüllbar ist.*

Da das SAT-Problem der Aussagenlogik entscheidbar ist, folgt mit Satz 3.17:

**Korollar 3.18.** *Das Erfüllbarkeitsproblem in der Rätsellogik ist entscheidbar.*

## 3.3 Transformation der Aussagenlogik in eine Klauselmenge

Im Rahmen meiner Arbeit wird die schnelle KNF, auch Tseitin-Kodierung genannt, für die Transformation der Aussagenlogik in die Klauselmenge verwendet. Im folgenden Abschnitt betrachten wir den Algorithmus genauer und hinterfragen die Komplexität dieser Anwendung.

### 3.3.1 Die schnelle KNF: die Tseitin-Kodierung

Wir betrachten nun einen Algorithmus, der eine aussagenlogische Formel in polynomieller Zeit in eine KNF umwandelt: die Tseitin-Kodierung. [3]

[3] Siehe Kapitel 2.4 in [SS12].

Die Tseitin-Kodierung transformiert eine aussagenlogische Formel in eine KNF, aber nicht in eine ihr äquivalente Formel, sondern erhält dabei lediglich die Erfüllbarkeit der Formel. Ihre Vorgehensweise besteht darin, komplexe Subformeln iterativ durch neue Variablen zu ersetzen. Die Anzahl der Variablen wird durch dieses Verfahren erhöht.
Sei $F[G]$ eine aussagenlogische Formel mit der Subformel $G$. Die Tseitin-Kodierung erzeugt daraus $(A \Leftrightarrow G) \wedge F[A]$ mit der neuen Variable $A$.

**Lemma 3.19.** *Sei $F[G]$ eine aussagenlogische Formel mit der Subformel $G$ und $A$ eine Variable, die nicht in $G$ vorkommt. Dann gilt: $F[G]$ ist erfüllbar genau dann, wenn $(G \Leftrightarrow A) \wedge F[A]$ erfüllbar ist. Es gilt also:*
*$F[G] \Rightarrow ((G \Leftrightarrow A) \wedge F[A])$, sowie $((G \Leftrightarrow A) \wedge F[A]) \Rightarrow F[G]$.*

*Beweis.* $F[G] \Rightarrow ((G \Leftrightarrow A) \wedge F[A])$

Sei $F[G]$ erfüllbar und sei $I$ eine beliebige Interpretation mit $I(F[G]) = 1$.
Sei $I'(A) := I(G)$. Werte die Formel $(G \Leftrightarrow A) \wedge F[A]$ unter $I'$ aus:
Es gilt $I'(G \Leftrightarrow A) = 1$ und $I\prime(F[G]) = I'(F[A]) = 1$. □

*Beweis.* $((G \Leftrightarrow A) \wedge F[A]) \Rightarrow F[G]$

Sei I(G $\Leftrightarrow$ A) $\wedge F[A] = 1$ für eine Interpretation I. Dann ist I(G) = I(A) und I(F[A]) = 1. Damit muss auch I(F[G]) = 1 sein. □

**Definition 3.20** (Tiefe einer Formel)**.** *Die Tiefe einer Formel ist definiert als die maximale Länge eines Pfads im Syntaxbaum.*
*Die Tiefe einer Subformel definiert man entsprechend folgendermaßen: Sei die Tiefe einer Formel F 0 in sich selbst. Sei G eine Subformel von F der Tiefe n. Es gelte weiterhin:*

- *Wenn $G = \neg H$, dann hat H die Tiefe n+1 in F.*
- *Wenn $G = (H_1 \otimes H_2)$ mit $\otimes \in \{\wedge, \vee, \Leftarrow, \Leftrightarrow\}$, dann sind $H_1, H_2$ Subformeln mit Tiefe $n + 1$ in F.*

**Definition 3.21** (Schneller KNF-Algorithmus)**.** *Wenn eine Formel F bereits in KNF vorliegt mit $F_1 \wedge \ldots \wedge F_n$, und eine Klausel $F_i$ eine Tiefe $\geq 4$ hat, dann ersetze $F_i$ wie folgt:*
*Wenn $G_j$ kein Atom ist, dann ersetze alle Subformeln $G_1, \ldots, G_m$ von $F_i$ mit Tiefe 3 in $F_j$ durch neue Variablen $A_j$: Ersetze $F_i = F_i'[G_1, \ldots, G_m]$ durch $(G_1 \Leftrightarrow A_1) \wedge \ldots \wedge (G_m \Leftrightarrow A_m) \wedge F_i'[A_1 \ldots A_m]$ in der Formel $F_1 \wedge \ldots \wedge F_n$.*
*Iteriere diesen Schritt so oft wie möglich.*
*Danach wandle die verbliebene Formel in KNF um.*

Zur Veranschaulichung der schnellen KNF betrachten wir ein Beispiel:

**Beispiel 3.22** (Schnelle KNF)**.** *Sei $F = (A \vee (B \vee (C \vee (D \vee E))))$ eine aussagenlogische Formel, wobei $A, B, C, D, E$ für Variablen, nicht für Subformeln stehen. Wenden wir die Tseitin-Kodierung aus 3.21 an, so erhalten wir unter Behalt der Erfüllbarkeit die Formel $F_{knf} = (A \vee (B \vee (C \vee X))) \wedge ((D \vee E) \Leftrightarrow X)$.*

### 3.3.2 Komplexität der Tseitin-Kodierung

Doch warum kommen wir bei einem Verfahren, das mehr Variablen erzeugt, auf eine verbesserte Komplexität? Dies ist der Fall, da bei der schnellen KNF aus einer Formel $F$ der Tiefe $n$ im ersten Schritt eine Formel der maximalen Tiefe 3 wird und es Klauseln der Form $G \Leftrightarrow A$ hinzugefügt werden, die ein geringere Tiefe haben als F. Da $A$ eine Variable ist, kann nur in $G$ wieder ersetzt werden und die Anzahl neu hinzugefügter Formeln ist kleiner als die Anzahl aller Subformeln der ursprünglichen Formel $F$. Der Aufwand der Umformung der kleinen Formeln ist konstant. Die Laufzeit ist linear, da die Größe der Formel $F$ gerade die Anzahl der Subformeln ist.
Das SAT-Problem lässt sich allerdings nicht ohne Preis in der Komplexität verbessern, daher sind folgende Einschränkungen zu beachten:

**Aussage 3.23.** *Wenn eine Formel F mittels Tseitin-Kodierung aus 3.21 in eine Formel F' transformiert wird, dann gilt:* [4]

1. *F ist erfüllbar gdw. F' ist erfüllbar.*
2. *F ist Widerspruch gdw. F' ein Widerspruch ist.*
3. *Wenn F eine Tautologie ist, dann ist F' erfüllbar.*
4. *F' $\Rightarrow$ F ist eine Tautologie.*
5. *F' ist i. Allg. keine Tautologie, ebenso wie F $\Rightarrow$ F'.*

### 3.3.3 Berechnung eines Modells mithilfe des SAT-Solvers

Nach Transformation in KNF kann ein beliebiger aussagenlogischer SAT-Solver verwendet werden, um das entstandene aussagenlogische Erfüllbarkeitsproblem zu lösen. Wir verwenden, wie in Abschnitt 2.4.1 erklärt, eine Implementierung des DPLL-Algorithmus, die neben der reinen Erfüllbarkeit auch ein (oder auch alle) Modell(e), im Falle der Erfüllbarkeit berechnet.

## 3.4 Rückrechnung: Transformation der Ausgabe in Rätsellogik

Nachdem der SAT-Solver ein Modell für unsere aussagenlogische Formel berechnet hat, transformieren wir dieses Modell wieder in Rätsellogik wie folgt.

Hierbei bleibt zu beachten, dass wir dem SAT-Solver nicht die ursprüngliche Formel $F$ übergeben haben, sondern die Ausgabe der schnellen KNF, deren Vorgehensweise in Abschnitt 3.21 erläutert wurde.

[4] Für Beweis siehe Kapitel 2.4 in [SS12].

Das bedeutet, dass der SAT-Solver im Falle der Erfüllbarkeit ein Modell für die Formel $F_{knf}$ liefert, bei dem neue Variablennamen enthalten sein können, die nicht in der Formel $F$ vorkommen.

Um nun aus diesem Modell $I$ für $F_{knf}$ ein Modell $I'$ für $F$ zu gewinnen, in dem nur die Variablen aus $F$ vorkommen dürfen, ist die Rückrechnung wie folgt definiert:

**Definition 3.24.** *Wenn I das aussagenlogische Modell ist, dann konstruiere I' als Modell der Rätsellogik durch:*

- *für alle $p \in P$ mit Stelligkeit > 0 setze I'(p) = $\{(k_1, \ldots, k_n) \mid I(X_{p(k_1,\ldots,k_n)}) =$ `True`$\}$*
- *für alle $p \in P$ mit Stelligkeit 0 setze I'(p) = $I(X_{p(k_1,\ldots,k_n)})$*

### 3.4.1 Beispiel einer Rückrechnung

Betrachten wir wieder unser Beispiel aus 3.6. Zur Erinnerung:

K = {Elly, Vanessa, Klaudia, Louisa}
S = {Blondinen, Brünetten}
Blondinen = {Elly, Vanessa}
Brünetten = {Klaudia, Louisa}
P = {jung (x), befreundet (x,y)}

F = ∀ x ∈Blondinen ∃ y ∈Brünetten (jung(x) ⇒ befreundet(x,y) ∧ jung(Vanessa) ∧ jung(Elly) ∧ ¬ (befreundet(Vanessa,Klaudia)) ∧ ¬ (befreundet(Elly,Louisa)))

Wir haben in 3.2 die Formel F sukzessive in eine aussagenlogische Formel $[\![F]\!]$ übersetzt. Hier zur Erinnerung noch einmal $[\![F]\!]$:

| Formel | Übersetzung |
|---|---|
| $(((X_{jung(Elly)} \Rightarrow X_{befreundet(Elly,Klaudia)}) \wedge X_{jung(Vanessa)} \wedge X_{jung(Elly)} \wedge \neg X_{befreundet(Vanessa,Klaudia)} \wedge \neg X_{befreundet(Elly,Louisa)}) \vee ((X_{jung(Elly)} \Rightarrow X_{befreundet(Elly,Louisa)}) \wedge X_{jung(Vanessa)} \wedge X_{jung(Elly)} \wedge \neg X_{befreundet(Vanessa,Klaudia)} \wedge \neg X_{befreundet(Elly,Louisa)})) \wedge (((X_{jung(Vanessa)} \Rightarrow X_{befreundet(Vanessa,Klaudia)}) \wedge X_{jung(Vanessa)} \wedge X_{jung(Elly)} \wedge \neg X_{befreundet(Vanessa,Klaudia)} \wedge \neg X_{befreundet(Elly,Louisa)}) \vee ((X_{jung(Vanessa)} \Rightarrow X_{befreundet(Vanessa,Louisa)}) \wedge X_{jung(Vanessa)} \wedge X_{jung(Elly)} \wedge \neg X_{befreundet(Vanessa,Klaudia)} \wedge \neg X_{befreundet(Elly,Louisa)}))$ | |

Tabelle 3.4: Formel $[\![F]\!]$, letzter Schritt der Übersetzung aus 3.2

### Vereinfachung der Formel $[\![F]\!]$

Diese Formel $[\![F]\!]$ können wir noch weiter vereinfachen:

$$[\![F]\!] = X_{jung(Vanessa)} \wedge \neg X_{befreundet(Vanessa,Klaudia)} \wedge X_{jung(Elly)} \wedge \neg X_{befreundet(Elly,Louisa)} \wedge ((X_{jung(Elly)} \Rightarrow X_{befreundet(Elly,Klaudia)}) \vee (X_{jung(Elly)} \Rightarrow X_{befreundet(Elly,Louisa)})) \wedge ((X_{jung(Vanessa)} \Rightarrow X_{befreundet(Vanessa,Klaudia)}) \vee (X_{jung(Vanessa)} \Rightarrow X_{befreundet(Vanessa,Louisa)}))$$

### Nach Anwendung der schnellen KNF: $F_{knf}$

Nachdem wir auf der Formel $[\![F]\!]$ die schnelle KNF angewandt haben, ergibt sich folgende Formel $F_{knf}$ mit zwei neuen Variablen: $Y_0$ und $Y_1$.

$$F_{knf} = Y_0 \wedge Y_1 \wedge X_{jung(Elly)} \wedge \neg X_{befreundet(Vanessa,Klaudia)} \wedge X_{jung(Vanessa)} \wedge \neg(X_{befreundet(Elly,Louisa)} \Leftrightarrow Y_1) \wedge ((X_{jung(Elly)} \Leftrightarrow X_{befreundet(Elly,Klaudia)}) \vee ((X_{jung(Elly)} \Leftrightarrow X_{befreundet(Elly,Louisa)}))) \wedge (((X_{jung(Vanessa)} \Leftrightarrow X_{befreundet(Vanessa,Klaudia)}) \vee (X_{jung(Vanessa)} \Leftrightarrow X_{befreundet(Vanessa,Louisa)})) \Leftrightarrow Y_0)$$

### Die Interpretation I

Kodieren wir dies in Haskell als Testfall[5] und übergeben es dem Programm dieser Arbeit, dann erhalten wir als Ergebnis die Interpretation I:

$\mathrm{I}(Y_0) = \texttt{True}$
$\mathrm{I}(Y_1) = \texttt{True}$
$\mathrm{I}(X_{jung(Elly)}) = \texttt{True}$
$\mathrm{I}(X_{jung(Vanessa)}) = \texttt{True}$
$\mathrm{I}(X_{befreundet(Elly,Klaudia)}) = \texttt{True}$
$\mathrm{I}(X_{befreundet(Vanessa,Louisa)}) = \texttt{True}$
$\mathrm{I}(X_{befreundet(Elly,Louisa)}) = \texttt{False}$
$\mathrm{I}(X_{befreundet(Vanessa,Klaudia)}) = \texttt{False}$

### Rückrechnung in Rätsellogik: I $\rightarrow$ I'

Da die durch die Tseitin-Kodierung hinzugefügten Variablen keine Rolle für das Modell $I'$ spielen, betrachten wir also nur folgende Modelle $I$:

[5] Dieses und andere Beispiele betrachten wir in ihrer Kodierung in Kapitel 4.

$\mathrm{I}(X_{jung(Elly)}) = \texttt{True}$
$\mathrm{I}(X_{jung(Vanessa)}) = \texttt{True}$
$\mathrm{I}(X_{befreundet(Elly,Klaudia)}) = \texttt{True}$
$\mathrm{I}(X_{befreundet(Vanessa,Louisa)}) = \texttt{True}$
$\mathrm{I}(X_{befreundet(Elly,Louisa)}) = \texttt{False}$
$\mathrm{I}(X_{befreundet(Vanessa,Klaudia)}) = \texttt{False}$

Und daraus erhalten wir das folgende Modell $I'$ für $F$:

I'(befreundet) = {(Elly,Klaudia), (Vanessa,Louisa)}
I'(jung) = {Elly, Vanessa}

Das Antwort auf die ursprüngliche Frage des Rätsels „Wer ist mit wem befreundet?" lautet also: Elly ist mit Klaudia befreundet und Vanessa ist mit Louisa befreundet.

# Kapitel 4

# Implementierung

In diesem Kapitel betrachten wir die Implementierung des Programms dieser Arbeit. Sie stellt damit die Umsetzung der in Kapitel 3 beschriebenen Umwandlungen und Vorgehensweisen dar.

Zunächst werden die importierten Module in Abschnitt 4.1 vorgestellt und abgegrenzt, welche Module bereitgestellt wurden, um auf ihnen aufzubauen.
Danach betrachten wir die einzelnen Datentypen in Abschnitt 4.2.
Dann wenden wir uns in Abschnitt 4.3 dem implementierten Typcheck zu, führen hierzu auch einige Tests zur Korrektheit durch und überprüfen die Fehlermeldungen.

Anschließend betrachten wir den Kern des Programms: die Implementierung der Transformationen.

Dazu sehen wir uns in Abschnitt 4.4 zunächst an, wie die Transformation von Rätsellogik in Aussagenlogik, wie wir sie in Kapitel 3.2 gezeigt haben, implementiert wurde. Danach erläutern wir die aus Kapitel 3.3 bekannte Berechnung der Modelle der aussagenlogischen Formel in Abschnitt 4.5 mithilfe des SAT-Solvers.
Abschließend betrachten wir, wie das Ergebnis des SAT-Solvers in Abschnitt 4.6 in Rätsellogik zurück transformiert wird. Dies ist die implementierte Umsetzung der in Kapitel 3.4 dargestellten Rückrechnung.

Um erst einmal eine Übersicht zu erlangen, wie das Programm schrittweise funktioniert, betrachten wir die einzelnen Etappen in einer Grafik, die den eben beschriebenen Ablauf verbildlicht.
In Abbildung 4.1 ist die einführende Grafik zu sehen, die die einzelnen Schritte der Implementierung verdeutlicht, wobei die jeweiligen Repräsentationen im Code kursiv notiert sind: entweder der Datentyp des Zustands oder die Funktion der Umwandlung.

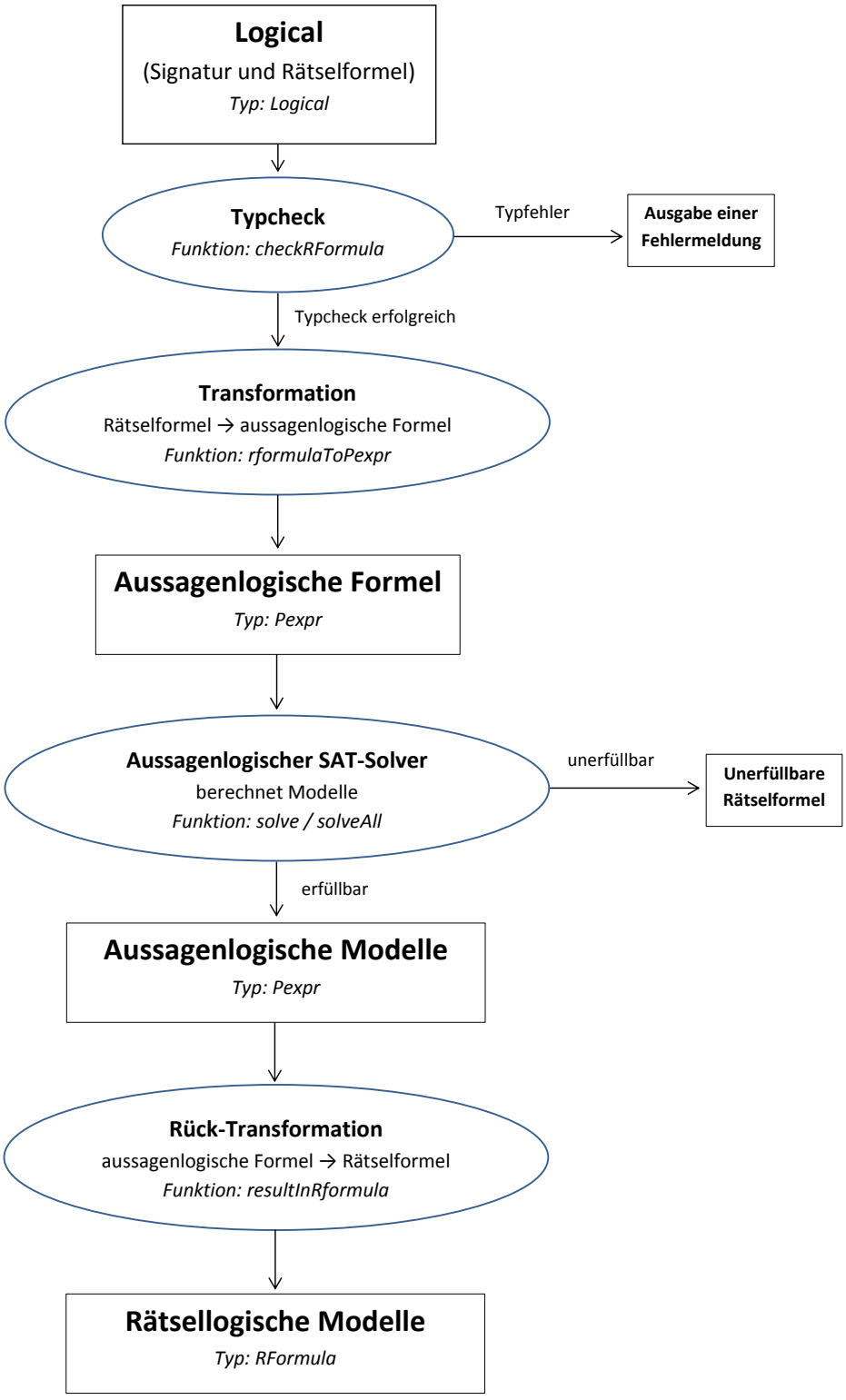


Abbildung 4.1: Die einzelnen Schritte der Implementierung

## 4.1 Importierte Module

Für das Programm bedarf es einiger Module, die am Anfang geladen werden.
Vier dieser Module wurden von der Professur für Künstliche Intelligenz bereitgestellt. Dazu zählen das Modul `Prop.hs`, in dem Ausdrücke der Aussagenlogik definiert werden, sowie das Modul `DavisPutnamschnell.hs`, welches eine Haskell-Implementierung eines einfachen SAT-Solvers ist, der auf dem DPLL-Algorithmus basiert.
Zu den zwei weiteren bereitgestellten Modulen zählen das Modul `Simp.hs`, das im Modul `DavisPutnamschnell.hs` für die KNF-Berechnung importiert wird, wobei `Simp.hs` wiederum `SimpCnf.hs` für die schnelle KNF-Berechnung importiert.

Des Weiteren importiert das Programm die Datei `Testfaelle.hs`, in der die Rätsel kodiert sind.
Ansonsten werden nur Standardbibliotheken importiert:

```
import Prop
import DavisPutnamschnell hiding(backSubs,doBackSubs)
import Testfaelle
import Data.List
import Data.Maybe
import System.Environment
import qualified Data.Set as Set
```

## 4.2 Datentypen

Betrachten wir nun die Datentypen des Programms.
Unser Datentyp für Ergebnisse des SAT-Solvers lautet `Result` und ist folgendermaßen definiert:

```
data Result name = Taut | Unsat | Sat [name] | SatAll [[name]]
 deriving(Show,Eq)
```

Das bedeutet, dass `Result` entweder eine Tautologie, eine Kontradiktion, eine erfüllbare Formel mit einem Modell oder eine mit mehreren Modellen sein kann. Dabei ist der Typ polymorph über dem Typ der Variablennamen definiert.

Aussagenlogische Ausdrücke sind wie folgt definiert:

```
data Pexpr name = Ptrue
                | Pfalse
                | Pvar name
                | Pnot  (Pexpr name)
                | Pand [(Pexpr name)]
                | Por  [(Pexpr name)]
                | Pimpl  (Pexpr name) (Pexpr name)
                | Pequiv (Pexpr name) (Pexpr name)
                deriving (Ord, Eq, Show)
```

Ein aussagenlogischer Ausdruck kann somit wahr, falsch, eine Variable, eine Negation, eine Konjunktion, eine Disjunktion, eine Implikation oder eine Äquivalenz sein. Für Konjunktionen und Disjunktionen werden hierbei Listen von Argumenten verwendet, sodass diese Junktoren mit beliebiger (statt binärer) Stelligkeit verwendet werden können.

Eine Formel der Rätsellogik besitzt ähnliche Ausdrücke, erweitert um die Existenzquantoren, den Allquantor und die Prädikate.

```
data RFormula name = Rand [RFormula name]
                   | Ror [RFormula name]
                   | Rnot (RFormula name)
                   | Rimpl (RFormula name) (RFormula name)
                   | Requiv (RFormula name) (RFormula name)
                   | Rex name Mengenname (RFormula name)
                   | RexAtLeast Int name Mengenname (RFormula name)
                   | RexAtMost Int name Mengenname (RFormula name)
                   | RexExactly Int name Mengenname (RFormula name)
                   | Rall name Mengenname (RFormula name)
                   | Rpredicate Predicate [name]
   deriving(Show,Eq)
```

Des Weiteren besitzt eine Formel der Rätsellogik eine Signatur (K,S,P), die im Programm durch den Datentyp `Preambel` dargestellt wird:

```
data Preambel =
     Preambel [(Sortname,[Element])] [(Predicate,[Sortname])]
```

Die Preambel enthält dabei die Sorten, die durch Listen `[(Sortname,[Element])]` dargestellt werden, wobei jede Sorte durch ein Paar bestehend aus dem Sortennamen und der Liste der zugehörigen Konstanten dargestellt wird. Prädikate werden durch Paare `[(Predicate,[Sortname])]` dargestellt, wobei `Sortname` den Typ des

Prädikats darstellt.
Die Konstanten sind in den Sorten in `[Element]` enthalten, und werden somit direkt ihren Sorten zugewiesen. Die Typen `Predicate`, `Element` und `Sortname` sind Typsynonyme für Strings:

```
type Sortname = String
type Element = String
type Predicate = String
```

Und ein in Rätsellogik kodiertes Rätsel lässt sich durch den Datentyp `Logical` darstellen, der die Preambel - also die Signatur - und die Formel selbst enthält:

```
type Logical = (Preambel,RFormula String)
```

Betrachten wir dafür den Code eines exemplarischen Testfalls:

```
testfall :: Logical
testfall = (preamble, rformel)
 where
   preamble = Preambel sorts praedikate
   sorts = [("S1",["K1","K2"]),
            ("S2", ["K3"])]
   praedikate = [("p1",["S1"]),
                 ("p2",["S1","S2"])]
   rformel = Rex "x" "S1" (Rpredicate "p2" ["x","K3"])
```

## 4.3 Typcheck

Bevor der Code Transformationen vornimmt, macht er einen Typcheck entsprechend der Typregeln in Abschnitt 3.1.2. Die Funktion `checkRformula` bekommt eine Formel mit ihrer Signatur übergeben und prüft anschließend mit der Hilfsfunktion `typecheckRformula` für die einzelnen Fälle, ob die verwendeten Sorten existieren, ob die verwendeten Prädikate existieren, ob eine nicht definierte Sorte in der Formel vorkommt, ob ein Prädikat zu viele oder zu wenige Argumente übergeben bekommt und ob eine Variable eine falsche Sorte hat.

Betrachten wir `checkRformula` stückweise. Zuerst wird in zwei Tests unterteilt, den Test der Signatur in `typecheckPreamble` und den Test der Formel in `typecheckRformula`:

```
checkRformula (preamble,rFormel) =
  typecheckPreamble preamble &&
  typecheckRformula [] preamble rFormel
```

In der Funktion `typecheckPreamble` wird dann getestet, ob ein Prädikat zu einer Sorte zugewiesen wurde, die nicht existiert:

```
typecheckPreamble (Preambel sorts predicates) =
 go predicates
  where
   go [] = True
   go ((p,typ):ps) =  (check typ) && (go ps)
    where
     check [] = True
     check (t:ts) =
      case (lookup t sorts) of
       Just _ -> check ts
       Nothing ->
         error -- Typ verwendet, der nicht existiert
          (unlines
            [""
            , "*** Fehler waehrend des Typchecks:"
            , "*** Das Praedikat " ++ show p ++ " wurde mit dem Typ "
            , "***    " ++ concat (intersperse " -> " (typ ++ ["Bool"]))
            , "*** definiert, aber die Sorte " ++ t ++ " existiert nicht."
            ])
```

Die Funktion `typecheckRformula` überprüft dann die Formel, zunächst wird dabei unterteilt, wie die Funktion in Abhängigkeit vom Aufbau der Formel aufgerufen wird:

```
typecheckRformula gamma preamble (Rand xs) =
  all (== True) (map (typecheckRformula gamma preamble) xs)
typecheckRformula gamma preamble (Ror xs) =
  all (== True) (map (typecheckRformula gamma preamble) xs)
typecheckRformula gamma preamble (Rimpl f1 f2) =
 (typecheckRformula gamma preamble f1) &&
 (typecheckRformula gamma preamble f2)
typecheckRformula gamma preamble (Requiv f1 f2) =
 (typecheckRformula gamma preamble f1) &&
 (typecheckRformula gamma preamble f2)
typecheckRformula gamma preamble (Rnot f) =
 (typecheckRformula gamma preamble f)
```

Im Folgenden wird für die verschiedenen Quantoren überprüft, ob die in der Formel verwendete Sorte in der Signatur existiert, wobei wir hier aus Platzgründen nur einen Ausschnitt der Funktion `typecheckRformula` betrachten:

```
typecheckRformula gamma (Preambel sorts predicates) (Rex x s f) =
 if s `elem` (map fst sorts) -- die verwendete Sorte existiert
  then typecheckRformula ((x,s):gamma) (Preambel sorts predicates) f
  else
   error -- Sorte existiert nicht
       (unlines
            [""
            , "*** Fehler waehrend des Typchecks:"
            , "*** Die Sorte " ++ show s ++ ", die in"
            , "***    EXISTS " ++ s ++ x ++ "..."
            , "*** verwendet wurde, existiert nicht."
            ])
              . . .
```

Die Fehlermeldungen sind auch für die anderen Quantoren implementiert, unter dem Link www.ki.informatik.uni-frankfurt.de/master/programme/logicals befindet sich der gesamte Code zum Nachschlagen.

Anschließend wird überprüft, ob die in der Formel verwendeten Prädikate in der Signatur existieren. Wenn nicht, wird die entsprechende Fehlermeldung geworfen:

```
typecheckRformula gamma (Preambel sorts predicates) (Rpredicate pred args) =
 case lookup pred predicates of
  Nothing ->
   error -- Praedikat existiert nicht
   (unlines
        [""
        , "*** Fehler waehrend des Typchecks:"
        , "*** Das in "
        , "***    " ++ pred ++ "(" ++ concat (intersperse "," args) ++ ")"
        , "*** verwendete Praedikat " ++ show pred ++ " existiert nicht."
        ])
  Just typ  -> check gamma sorts args typ
   where
    check gamma sorts args typ
     | length args /= length typ =
        error -- Stelligkeit nicht beachtet
         (unlines
        [""
```

```
        ,"*** Fehler waehrend des Typchecks:"
        , "*** Praedikat " ++ show pred ++
         " wird mit falscher Stelligkeit verwendet."
        , "*** Die Stelligkeit von " ++ pred ++ " ist
         " ++ show (length typ) ++ ", aber in"
        , "***  " ++ pred ++ "(" ++ concat (intersperse "," args) ++ ")"
        , "*** wird es mit Stelligkeit " ++ show (length args) ++
         " verwendet."
        , "*** Bitte" ++ if length args < length typ then " mehr "
                         else " weniger " ++ "Argumente verwenden."
        ])
```

Wobei die Hilfsfunktion `check` überprüft, ob die Prädikate die korrekte Stelligkeit haben, ob in der Formel eine nicht definierte Sorte vorkommt und ob die Variablen die richtigen Sorten haben:

```
check _ _ [] [] = True
check gamma sorts (a:args') (t:typ') =
 case lookup t sorts of
  Nothing ->
    error  -- Im Typ kommt eine nicht definierte Sorte vor
     "Fehler sollte nicht auftreten (unbekannte Sorte im Typ)"
  Just elements ->
    if a `elem` elements -- a ist also ein Objekt von der richtigen Sorte
     then check gamma sorts args' typ' -- alles ok, weitermachen
     else if a `elem` (concatMap snd sorts)
      then error -- a ist von der falschen Sorte
       (unlines
        [""
        ,"*** Fehler waehrend des Typchecks:"
        ,"*** Praedikat " ++ show pred ++
         " wird mit falschem Typ verwendet."
        ,"*** Der Typ von " ++ pred ++ " ist " ++ (concat (intersperse
         " -> " (typ ++ ["Bool"])))
        ,"*** Das Argument " ++ show a ++ "in der Verwendung"
        ,"***    " ++  pred ++ "(" ++ concat (intersperse "," args)
         ++ ")"
        ,"*** ist jedoch nicht von der Sorte " ++ t ++
          ", sondern hat die Sorte(n)"
        ,"*** " ++ concat (intersperse ", " (map fst $ filter
        (\(x,y) -> a `elem` y) sorts))
        ])
```

```
else case lookup a gamma of -- ist a eine Variable?
 Nothing -> error
  (unlines
   [""
    ,"*** Fehler waehrend des Typchecks:"
    ,"*** Die Formel ist nicht geschlossen, da die Variable "
     ++ show a
    ,"*** in " ++ pred ++ "(" ++ concat (intersperse "," args)
     ++ ")"
    ,"*** frei vorkommt."
    ])

 Just s -> if s == t -- richtige Sorte
  then check gamma sorts args' typ' -- alles ok, weitermachen
  else error -- "Variable hat falsche Sorte"
   (unlines
    [""
    ,"*** Fehler waehrend des Typchecks:"
    ,"*** Praedikat " ++ show pred ++
     " wird mit falschem Typ verwendet."
    ,"*** Der Typ von " ++ pred ++ " ist " ++
    (concat (intersperse " -> " (typ ++ ["Bool"])))
    ,"*** Die Argumentvariable " ++ show a ++ "in der Verwendung"
    ,"***    " ++  pred ++ "(" ++ concat (intersperse "," args)
     ++ ")"
    ,"*** ist jedoch nicht von der Sorte " ++ t ++
     ", sondern hat die Sorte"
    ,"***    " ++  s
    ])
```

### 4.3.1 Testen des Typchecks

Betrachten wir nun drei Testfälle, mit denen wir den Typcheck testen:

- Als Erstes ein Testfall mit mehreren möglichen Belegungen, der keinen Typfehler erzeugen sollte.

```
-- SatAll Fall, mehrere Belegungen
testSatAll :: Logical
testSatAll = (preamble,rformel)
 where
   preamble = Preambel sorts predicates
   sorts  = [("Maenner",["M1","M2","M3"])]
```

```
  predicates = [("gluecklich",["Maenner"])]
  rformel   = Rex "x" "Maenner" (Rpredicate "gluecklich" ["x"])

*Main> checkRformula testSatAll
True
```

- Als zweites Beispiel ein Testfall, in dem die Sorte "Kinder" verwendet wird, obwohl sie nicht in der Signatur definiert wurde.

```
-- Sorte verwendet, die nicht existiert!
testError1 :: Logical
testError1 = (preamble,rformel)
 where
  preamble = Preambel sorts predicates
  sorts  = [("Maenner",["Bert"])
           , ("Frauen", ["Anastasia"])]
  predicates = [("gluecklich",["Maenner"])]
  rformel   = Rex "x" "Kinder" (Rpredicate "gluecklich" ["x"])

*Main> checkRformula testError1
*** Exception:
*** Fehler waehrend des Typchecks:
*** Die Sorte "Kinder", die in
***    EXISTS x IN Kinder...
*** verwendet wurde, existiert nicht.
```

- Und als drittes und letztes Beispiel ein Testfall, indem die Sorte "Kinder" in der Signatur bei den Typen der Prädikate verwendet wird, obwohl sie nicht bei den Sorten definiert wurde.

```
-- Im Typ kommt eine nicht definierte Sorte vor
testError2 :: Logical
testError2 = (preamble,rformel)
 where
  preamble = Preambel sorts predicates
  sorts  = [("Maenner",["Bert"])
           , ("Frauen", ["Anastasia", "Lena"])]
  predicates = [("MutterVon",["Frauen","Kinder"])]
  rformel   = Rpredicate "MutterVon" ["Anastasia","Lena"]

*Main> checkRformula testError2
*** Exception:
*** Fehler waehrend des Typchecks:
*** Das Praedikat "MutterVon" wurde mit dem Typ
***    Frauen -> Kinder -> Bool
*** definiert, aber die Sorte Kinder existiert nicht.
```

Wenn alle potentiellen Fehler ausgeschlossen wurden, beginnt die Transformation in die Aussagenlogik.

## 4.4 Rätsellogik in Aussagenlogik

Die Transformation der in Rätsellogik gegebenen Formel in Aussagenlogik hat den Hauptaufruf `rformulaToPexpr` und erwartet als Eingabe - wie erwartet - den Typ `Logical`, also eine Rätselformel mit ihrer Signatur. Sie gibt ein Tupel bestehend aus einem aussagenlogischen Ausdruck und einer Liste, die aus einem Tupel der Abbildung des Variablennamens in Aussagenlogik (als String) und dem Prädikat in Rätsellogik besteht. Dieses Tupel der Abbildung ist wichtig, um später wieder von der aussagenlogischen Variable $X_{p(k_1,\ldots,k_n)}$ zum Prädikat der Rätsellogik $p(k_1,\ldots,k_n)$ zurückzugelangen:

```
rformulaToPexpr :: (Preambel, RFormula Element) ->
(Pexpr String, [(String, RFormula String)])
rformulaToPexpr (preamble,rFormel) =
 if (checkRformula (preamble,rFormel))
  then subst (transformRinP (preamble,rFormel)) []
  else error "Typcheck fehlgeschlagen"
```

Die Funktion `transformRinP`, aufgerufen in `rformulaToPexpr`, stellt den wesentlichen Kern der Transformation dar und nimmt nun den ersten Schritt eben dieser in Angriff, in dem sie die Quantoren eliminiert:

```
transformRinP :: Logical -> RFormula Element
transformRinP (preamble,rFormel) = eliminateQuantifiers (preamble,rFormel)
```

Die Funktion `eliminateQuantifiers` tilgt nun, wie der Name schon verrät, die Quantoren:

```
eliminateQuantifiers :: (Preambel, RFormula Element) -> RFormula Element
eliminateQuantifiers (preamble, Rall var sort prumpf) =
    eliminateOneAll (preamble, Rall var sort
    (eliminateQuantifiers (preamble,prumpf)))
    -- von innen nach außen Quantoren tilgen
eliminateQuantifiers (preamble, Rex var sort prumpf) =
    eliminateOneEx (preamble, Rex var sort
    (eliminateQuantifiers (preamble,prumpf)))
eliminateQuantifiers (preamble, RexExactly n var sort prumpf) =
    eliminateOneExExactly (preamble, RexExactly n var sort
```

```
    (eliminateQuantifiers (preamble,prumpf)))
eliminateQuantifiers (preamble, RexAtLeast n var sort prumpf) =
    eliminateOneExAtLeast (preamble, RexAtLeast n var sort
    (eliminateQuantifiers (preamble,prumpf)))
eliminateQuantifiers (preamble, RexAtMost n var sort prumpf) =
    eliminateOneExAtMost (preamble, RexAtMost n var sort
    (eliminateQuantifiers (preamble,prumpf)))
```

Auch für Formeln mit anderen Operatoren wie Konjunktionen muss `eliminateQuantifiers` rekursiv überprüfen, ob innerhalb der Formel noch weitere Quantoren vorkommen und diese ersetzen:

```
eliminateQuantifiers (preamble, Rand xs) =
    Rand [ eliminateQuantifiers (preamble,formel) | formel <- xs]
eliminateQuantifiers (preamble, Ror xs) =
    Ror [ eliminateQuantifiers (preamble,formel) | formel <- xs]
eliminateQuantifiers (preamble, Rnot formula) =
    Rnot (eliminateQuantifiers (preamble,formula))
eliminateQuantifiers (preamble, Rimpl n1 n2) =
    Rimpl (eliminateQuantifiers (preamble,n1))
    (eliminateQuantifiers (preamble,n2))
eliminateQuantifiers (preamble, Requiv n1 n2) =
    Requiv (eliminateQuantifiers (preamble,n1))
    (eliminateQuantifiers (preamble,n2))
eliminateQuantifiers (preamble, Rpredicate pred xs) =
    Rpredicate pred xs
```

Die tatsächliche Ersetzung des All- und Existenzquantors übernimmt dann `eliminateOneAll`, bzw. `eliminateOneEx`:

```
-- \A x element M (glücklich (x)) ->
-- ((glücklich (M1)) and (glücklich (M2)) and (glücklich (M3)))
eliminateOneAll :: (Preambel, RFormula Element) -> RFormula Element
eliminateOneAll (preamble, Rall var sort (rformel)) =
    let elemente = (getElemente (preamble,Rall var sort (rformel)))
    in Rand [substitute element var (rformel)| element <- elemente]

-- \E x element M (glücklich (x)) ->
-- ((glücklich (M1)) or (glücklich (M2)) or (glücklich (M3)))
eliminateOneEx :: (Preambel, RFormula Element) -> RFormula Element
eliminateOneEx (preamble, Rex var sort (rformel)) =
   let elemente = (getElemente (preamble,Rex var sort (rformel)))
   in Ror [ (substitute element var (rformel)) | element <- elemente]
```

Der Kommentar im Code erläutert was geschieht: Die Elemente der Sorte werden durch die Funktion `getElemente` aus der Sorte geholt, in die Formel eingesetzt und dann im Fall des Allquantors verundet, im Fall des Existenzquantors verodert.
Daher hier ein Blick auf die Hilfsfunktion `getElemente`:

```
-- hiermit kann man in den preamble Listen die "M1","M2","M3" rausholen
getElemente :: (Preambel, RFormula t) -> [Element]
getElemente ((Preambel sorts praedikate),(Rall var sort rumpf)) =
   case lookup sort sorts of
     Just elemente -> elemente
     -- jetzt hat man hier die Elemente drin in elemente
     Nothing -> error "Sorte nicht gefunden"

getElemente ((Preambel sorts praedikate),(Rex var sort rumpf)) =
   case lookup sort sorts of
     Just elemente -> elemente
     Nothing -> error "Sorte nicht gefunden"
```

Die Hilfsfunktion `substitute`, die in den Funktionen `eliminateOneAll` und `eliminateOneAll` aufgerufen wird, setzt für alle Vorkommen einer Variablen $x \in$ *Sorte* die entsprechenden Konstanten dieser Sorte ein:

```
substitute :: Eq name => name -> name -> RFormula name -> RFormula name
substitute e v (Rpredicate p xs) = Rpredicate p
  (map (\x -> if x == v then e else x) xs)
substitute e v (Rand xs)         = Rand (map (substitute e v) xs)
substitute e v (Ror xs)          = Ror (map (substitute e v) xs)
substitute e v (Rnot formula)    = (Rnot (substitute e v formula))
substitute e v (Rimpl n1 n2)     = Rimpl (substitute e v n1)
  (substitute e v n2)
substitute e v (Requiv n1 n2)    = Requiv (substitute e v n1)
  (substitute e v n2)
```

Zu diesem Zeitpunkt ist die Formel noch immer vom Typ `RFormula` - erst jetzt erfolgt in der Funktion `rformulaToPexpr` der Aufruf der Hilfsfunktion `subst` und verwandelt Prädikate und die anderen Ausdrücke der Rätsellogik in Ausdrücke der Aussagenlogik und speichert in einer Merkliste die ursprünglichen Ausdrücke der Rätsellogik.
Dafür wird das Prädikat im neuen Namen in `Pvar` vermerkt, dafür wird es in einen zusammenhängenden Variablennamen umgewandelt. "verheiratet" "Mann" "Frau" würde dadurch zu "verheiratet_Mann_Frau". Damit dabei das ursprüngliche Prädikat nicht verloren geht, wird es im Paar `(Pexpr, merkliste)` in der Merkliste gespeichert, da sie für die spätere Rückrechnung in Prädikatenlogik notwendig ist:

```
-- Hilfsfunktion subst für rformulaToPexpr

subst :: RFormula [Char] -> [([Char], RFormula [Char])]
  -> (Pexpr [Char], [([Char], RFormula [Char])])
subst (Rpredicate p xs) memorylist =
    let new_name = concat (intersperse "_" (p:xs))
    in (Pvar new_name, ((new_name,Rpredicate p xs):memorylist))

subst (Rand xs) memorylist =
  let (xs1,memorylist1) = (go xs memorylist)
  in (Pand xs1, memorylist1)
   where
    go [] memorylist = ([],memorylist)
    go (rformel:rest) memorylist =
      let (pformel,memorylist1) = (subst rformel memorylist)
         (restformeln,memorylistN) =  go rest memorylist1
      in ((pformel:restformeln),memorylistN)

              . . .

subst (Rex _ _ _)  memorylist = error "sollte nicht passieren"
subst (Rall _ _ _) memorylist = error "sollte nicht passieren"
```

Und damit wäre die Transformation in eine aussagenlogische Formel abgeschlossen. Folgt als Nächstes also die Berechnung etwaiger Modelle der Formel.

## 4.5 Aussagenlogisches Erfüllbarkeitsproblem lösen

In diesem Teil des Codes wird die schnelle KNF durch `solve` aufgerufen und angewandt. Die Funktion `solve` berechnet ein Modell, falls es eins für die Formel gibt.

Als Eingabe erhält `solve` einen aussagenlogischen Ausdruck vom Typ `Pexpr` und die Ausgabe der Funktion ist das Ergebnis mit dem Typ `Result (Pexpr a)`.
Das bedeutet, dass das Ergebnis den Ergebnistyp als `Result` angibt - also beispielsweise `Sat` für ein Modell - und in `(Pexpr a)` die tatsächlichen Belegungen der Literale, also das Modell selbst.

```
-- berechnet ein Modell und streicht die eingeführten Namen durch
-- schnelle KNF in der Ausgabe
-- solve :: Eq a => Pexpr a -> Result (Pexpr a)
```

Die Funktionen `solve` und `solveAll` führe ich hier nur mit dem Datentyp auf, da sie von der Professor der Künstlichen Intelligenz bereitgestellt wurden. Der Kern

der Funktion liegt aber in dem Aufruf des SAT-Solvers, den wir am Anfang unseres Programms mit `solver = davisPutnam` definiert haben.

Analog zur Funktion `solve` gibt es noch die Funktion `solveAll`, die nicht nur ein, sondern alle möglichen Modelle der Formel berechnet.
Auch hier gilt der gleiche Datentyp wie für `solve`:

```
-- berechnet alle Modelle
-- solveAll :: Eq name => Pexpr name -> Result (Pexpr name)
```

Daraus resultiert dann unser Ergebnis, welches dann im nächsten Schritt wieder zurück in Rätsellogik transformiert werden soll.

## 4.6 Ergebnis in Rätsellogik zurückrechnen

Da wir zuvor die `Rformulas` in der Merkliste mit an unsere Transformation in aussagenlogische Ausdrücke gehängt haben, können wir nun auch wieder auf sie zugreifen. Die Funktion `resultInRformula` berechnet für einen übergebenen Testfall, ob und wie viele Modelle für die Formel existieren. Dies geschieht durch die Funktionen `solve` und `solveAll`.
Direkt zu Beginn führt das Pattern Matching `let (a,b)` dazu, dass das Ergebnis der Formel `rformulaToPexpr` für einen leichten Zugriff in a = Pexpr String und b = [(String, RFormula String)] aufgeteilt wird.
Im Fall des SAT Modells - d.h., wenn nur ein Modell existiert - werden der Funktion getR zwei Argumente übergeben:

- das „rohe Modell", also eine Liste der erfüllenden Literale (ohne die negierten Literale)
- die Liste der Tupel `[(String, RFormula String)]`, welche im Tupel eine Abbildung des ersten Elements auf das zweite Element enthalten:
  - das erste Element im Tupel ist der Variablenname in Aussagenlogik ($X_{p(k_1,\ldots,k_n)}$),
  - das zweite Element ist das Prädikat in Rätsellogik, aus dem der Variablenname abgeleitet wurde ($p(k_1, \ldots, k_n)$.

```
resultInRformula testfall =
    let (a,b) = rformulaToPexpr testfall
        result = solveAll a
       in
         case result of
           Sat model ->
            let m = rawModel model -- nur Pvars, ohne Pnots
```

```
 in [(getR m b)]
 -- getR rohesModell RFormulasDesTestfalls
SatAll models ->
 let m = rawModels models
 in [getR model b | model  <- m]
Taut -> [getB (fst (unzip b)) b]
Unsat -> []
```

Die Hilfsfunktion `rawModels` filter lediglich die positiven Belegungen der Literale raus, ignoriert also die negierten Belegungen:

```
rawModel :: Eq t0 => [Pexpr t0] -> [Pexpr t0]
rawModel xs = filter isVar xs

rawModels [] = []
rawModels models = map rawModel models
```

Und der Zugriff auf die in der Merkliste angehängten Formeln in Rätsellogik erfolgt durch die Funktion `getR`. Sie bekommt, wie bereits erwähnt, zwei Argumente übergeben, und gibt dann eine Liste der erfüllenden Literale in Prädikatenlogik zurück. Dafür vergleicht `getR` die `Pvar Strings` aus der ersten Liste mit den `Strings` aus den Tupeln der zweiten Liste und gibt bei erfolgreichem Vergleich das zweite Element des Tupels der zweiten Liste zurück: Den `RFormula String`.

```
getR :: Eq a0 => [Pexpr a0] -> [(a0, RFormula String)] ->
        [Maybe (RFormula String)]
getR [] liste = []
getR ((Pvar var):xs) liste =
   (lookup var liste):(getR xs liste)
```

Die Funktion `resultInRformula` ist die letzte Funktion in der Transformation des Ergebnisses in rätsellogische Syntax. Nun bringen wir das Ergebnis lediglich in eine leichter lesbare Form mit der Funktion `formatAll`:

```
formatAll _ []  = []
formatAll i (x:xs) =
 unlines (
 ["============================"
 ,"Modell " ++ show i
 ,"============================"
 ]) ++ (format x) ++ (formatAll (i+1) xs )
```

Die Hilfsfunktion `format` setzt dabei die richtigen Umbrüche und formatiert die Ausgabe so, dass die Prädikate einzeln aufgezählt und ihre zugehörigen Konstanten in Tupeln aufgelistet werden. Wie die Ergebnismodelle dann aussehen, betrachten wir in Kapitel 6.1.

Diese letzte Funktion hat das Ergebnis nun übersichtlicher gestaltet und schließt unseren Code damit ab.

Als nächsten Schritt betrachten wir die intelligente Benutzungsschnittstelle, die mithilfe eines Parsers realisiert wurde und die Eingabe der Rätsel vereinfachen wird.

# Kapitel 5

# Intelligente Benutzungsschnittstelle

Dieses Kapitel widmet sich der intelligenten Benutzungsschnittstelle des Programms. Wir haben im vorherigen Kapitel 4 ausgiebig das Kernprogramm besprochen und seine wichtigsten Funktionen getestet.
Ziel dieser Arbeit ist es, dass der Benutzer ein Logikrätsel in einer Textdatei formuliert und dieses anschließend vom Programm eingelesen und verarbeitet (und im besten Fall gelöst) wird. Hierbei orientiert sich das Eingabeformat an der Syntax der Rätsellogik, versucht diese jedoch „natürlicher" zu gestalten, indem Worte statt Symbole verwendet werden, so dass die Eingabe „lesbarer" und näher an echten sprachlichen Sätzen ist. Da die deutsche Grammatik eher kompliziert ist und viele Sonderfälle erlaubt, verwenden wir hierfür englische Begriffe.

Technisch muss für diese Eingabeschnittstelle ein Parser implementiert werden, der gültige Eingaben erkennt und falsche Eingaben zurückweist und schließlich gültige Eingaben in den Haskell-Datentyp `Logical` überführt.

Die Vorverarbeitung der Eingabe besteht dabei aus drei grundlegenden Schritten:

1. lexikalische Analyse
2. syntaktische Analyse
3. semantische Analyse

Die lexikalische Analyse wird vom lexikalischen Scanner und die syntaktische Analyse vom Parser übernommen, danach erst folgt die semantische Analyse. Die semantische Analyse in Form des Typchecks haben wir bereits in Abschnitt 3.1.2 formal behandelt und seine Implementierung in Abschnitt 4.3 erläutert. Daher gehen wir in diesem Kapitel nicht mehr darauf ein.

In den folgenden Abschnitten besprechen wir erst Parser im Allgemeinen in Abschnitt 5.1, dann den Parsergenerator Happy, mit dem wir unseren Shift-Reduce-Parser erstellen und erläutern die Eigenschaften eines Shift-Reduce-Parsers.
Anschließend gehen wir in 5.2 die Implementierung des Parsers in den einzelnen Schritten durch und testen im letzten Abschnitt 5.3 den Parser.

## 5.1 Parser

Ein Parser ist ein Programm, welches (anhand einer kontextfreien Grammatik) die Gültigkeit einer Eingabe überprüft, und anschließend entweder eine Fehlermeldung ausgibt und die Eingabe zurückweist oder im Erfolgsfall die Eingabe in einen Syntaxbaum überführt. In unserem Fall soll der Parser die Eingabe des Benutzers an der Benutzungsschnittstelle in die Eingabe für das Kernprogramm vom Typ Logical umwandeln.
Ein Parser verwendet meist einen lexikalischen Scanner, der die Vorverarbeitung vornimmt. Dieser lexikalische Scanner, auch Lexer genannt, zerlegt die Benutzereingabe in sogenannte Token, die kleinste zusammenhängende Einheit, die der Parser verarbeitet. Außerdem entfernt der Lexer auch Leerzeichen und Kommentare, erkennt in der Eingabe Schlüsselwörter (`TokenKeyword`), Bezeichner (`TokenIdentifier`) und Symbole (`TokenSymbol`) und übergibt anschließend seine Liste von Token dem tatsächlichen Parser.
Dieser eigentliche Parser prüft dann anhand einer Grammatik die Eingabe, also die Aneinanderreihung der einzelnen Token.

Der Parser für die Benutzungsschnittstelle wurde mit einem Parsergenerator erstellt. Ein Parsergenerator ist ein Programm, das einen Parser mithilfe ihm gegebener Spezifikationen erstellt. Wir verwenden den Parsergenerator Happy.[1]

### 5.1.1 Parsergenerator Happy

Happy ist ein Parsergenerator für Haskell, das ein Haskell-Modul mit einem Parser für die entsprechende Grammatik erstellt. [2]
Happy kann entweder eine Folge von Zeichen direkt parsen, was in den meisten Fällen unpraktikabel ist, oder man kann in Happy einen Lexer, bzw. Tokenizer einbinden, den man selbst per Hand schreibt oder von einem Programm generieren lässt. In unserem Fall wurde per Hand ein Lexer geschrieben, der in den Parser eingebunden wurde, und den wir in Abschnitt 5.2 betrachten werden.

Parser, die von Happy generiert wurden, sind in der Regel schneller als gleichwertige Parser, die durch Parserkombinatoren oder Ähnliches realisiert wurden. Der Glasgow Haskell Compiler selbst nutzt einen Happy Parser.

Die Kernidee des Parsergenerators lautet wie folgt:

- Definiere die Parserspezifikation. Dies ist im Wesentlichen die Grammatik der Sprache, die erkannt werden soll. Dabei kann die Grammatik selbst noch Mehrdeutigkeiten enthalten, welche durch die zusätzliche Angabe von Prioritäten und Assoziativitäten der Operatoren aufgelöst werden.

[1] Parsergenerator Happy: Siehe [Mar].
[2] Dies und folgende Informationen des Abschnitts 5.1.1 findet man unter [Mar].

- Rufe die Spezifikationsdatei mit Happy auf, um ein kompilierbares Haskell Modul zu generieren: `happy parser.y` erzeugt `parser.hs`
- Importiere das Modul als Teil des Haskell Programms.

Auf diese Weise wird durch Happy ein Shift-Reduce-Parser erzeugt.

### 5.1.2 Shift-Reduce-Parser

Shift-Reduce-Parser zählen zu den LR-Parsern und verarbeiten somit eine Eingabe von links nach rechts, während eine Rechtsherleitung erzeugt wird.[3]

Shift-Reduce steht für „Schiebe-Reduziere" und beschreibt die zwei Hauptaktionen des Shift-Reduce-Parsers. Er kann insgesamt vier Aktionen durchführen:

- Schieben: Ein Zeichen aus der Eingabe auf den Stack schieben.
- Reduzieren: Das oberste Stück des Stacks durch ein Nichtterminal ersetzen.
- Akzeptieren: Wenn auf dem Stack das Startsymbol steht und die Eingabe leer ist.
- Fehler: Wenn weder Schiebe- noch Reduzieraktion möglich ist und nicht akzeptiert werden kann.

Shift-Reduce-Parser erzeugen eine Rechtsherleitung anhand einer Grammatik, die in der Parserspezifikation steht.

**Definition 5.1.** *Eine kontextfreie Grammatik ist ein 4-Tupel* $G = (N, T, P, \sigma)$ *mit*

1. *N: endliche Menge von Nichtterminalen*
2. *T: endliche Menge von Terminalen, wobei* $N \cap T = \emptyset$
3. $P \subseteq N \times (N \cup T)*$ *eine endliche Menge (Produktionssystem)*
4. $\sigma \in N$ *ist ein ausgezeichnetes Hilfszeichen (Startzeichen)*

Kontextfrei nennt man eine Grammatik, wenn auf der linken Seite der Regeln nur ein einziges Nichtterminal steht. Stehen auf der linken Seite mehrere Nichtterminale, muss nämlich der Kontext betrachtet werden, in dem Teile der Nichtterminale auftauchen.

Betrachten wir nun folgende kontextfreie Grammatik:

$$\begin{array}{lcl} \mathrm{S} & := & S + S \mid S \\ \mathrm{S} & := & S * S \\ \mathrm{S} & := & a \mid b \mid c \end{array}$$

[3] Quelle für den Abschnitt 5.1.2: [SS13].

Schauen wir uns dazu folgende Rechtsherleitung von $a + b * c$ an:

$$\begin{aligned} S &\rightarrow S + S \\ &\rightarrow S + S * S \\ &\rightarrow S + S * c \\ &\rightarrow S + b * c \\ &\rightarrow a + b * c \end{aligned}$$

Dies ist aber nicht die einzige Rechtsherleitung zu dem Ausdruck. Die andere Rechtsherleitung sieht wie folgt aus:

$$\begin{aligned} S &\rightarrow S * S \\ &\rightarrow S * c \\ &\rightarrow S + S * c \\ &\rightarrow S + b * c \\ &\rightarrow a + b * c \end{aligned}$$

Da es mehrere Rechtsherleitungen geben kann, ist es wichtig, dass die Operatorprioritäten zusätzlich zur Grammatik in die Parserspezifikation geschrieben werden, damit keine falsche Rechtsherleitung erzeugt wird.

Im folgenden Abschnitt sehen wir, wie der Shift-Reduce-Parser automatisch anhand einer Parserspezifikation generiert wurde, in der die Grammatik definiert wurde.

## 5.2 Implementierung des Parsers

Bevor wir die Implementierung des Parsers betrachten, halten wir erst einmal fest, welche Eingaben dem Benutzer nun durch die Benutzungsschnittstellen ermöglicht werden.

Die Eingabe von Rätseln passt sich nun an:
Die Übergabe der Signatur und der Formel ist nun durch die Benutzungsschnittstelle benutzerfreundlicher und besteht aus vier Teilen, wobei Wörter in Großbuchstaben Terminale darstellen:

1. Sorten:
   `SET: EineSorte := {konstante1,konstante2}`
   – Hiermit werden die Sorten und die ihnen zugehörigen Konstanten definiert.
   oder
   `SET : Sorte := Sorte1 UNION ... UNION SorteN`
   – In diesem Fall wird eine neue Sorte definiert, die die Vereinigung aller Elemente der auf der rechten Seite genannten Sorten darstellt.

2. Prädikate:
   `RELATION: prädikat SUBSET EineSorte`
   – Hier wird durch `SUBSET` festgelegt, welchen Typ das Prädikat hat. Hier wird die Mengenschreibweise verwendet: `prädikat SUBSET EineSorte` bedeutet, dass das Prädikat eine Teilmenge der Sorte ist, und `prädikat SUBSET (Sorte1 X Sorte2 X ... X Sorten)` legt fest, dass das Prädikat eine n-stellige Relation über den entsprechenden Sorten ist.
3. Formeln:
   ```
   PROPOSITION 1: Formel1 (5.2.1)
   PROPOSITION 2: Formel2 (5.2.1)
   PROPOSITION 3: Formel3 (5.2.1)
   PROPOSITION 4: Formel4 (5.2.1)
   ```
   – Die Bedingungen bzw. die einzelnen Formeln werden in Form von durchnummerierten Propositionen angegeben. Es ist auch möglich, nur eine Proposition zu definieren, das Zerlegen in mehrere Aussagen ist oft jedoch natürlicher.
4. Lösen der Formeln:
   `SOLVE 1 AND 3 AND 4`
   – Als letzten Schritt sorgt `SOLVE` dafür, dass die entsprechenden Formeln mithilfe des SAT-Solvers gelöst werden. Sollen alle definierten Propositionen verundet gelöst werden, so kann man auch `SOLVE ALL` aufrufen.

In Kapitel 6 betrachten wir unsere Rätsel kodiert als Eingabe für die Benutzungsschnittstelle.

### 5.2.1 Eingabe von Formeln

Formeln werden natürlichsprachig englisch formuliert. Sie werden durch die folgende kontextfreie Grammatik erzeugt, wobei *Formel*, *Predicate*, *Number*, *Sort* und *Variable* Nichtterminale sind und alle anderen Symbole Terminale sind:

*Formel* ::= *Predicate*
| *Formel* `AND` *Formel*
| *Formel* `OR` *Formel*
| *Formel* `XOR` *Formel*
| `NOT` *Formel*
| *Formel* `IMPLIES` *Formel*
| *Formel* `IFF` *Formel*
| `EXISTS A` *Sort Variable* `SUCH THAT` *Formel*
| `EXISTS AT LEAST` *Number Sort Variable* `SUCH THAT` *Formel*
| `EXISTS AT MOST` *Number Sort Variable* `SUCH THAT` *Formel*
| `EXISTS EXACTLY` *Number Sort Variable* `SUCH THAT` *Formel*
| `FOR ALL` *Sort Variable* `HOLDS` *Formel*
| (*Formel*)

Die Grammatiken für die weiteren Nichtterminale beschreiben wir informell:

- *Number* repräsentiert eine positive Ganzzahl.
- *Variable* ist ein String aus Buchstaben beginnend mit einem Kleinbuchstaben.
- *Sort* ist eine String aus Buchstaben beginnend mit einem Großbuchstaben.
- Prädikate (Nichtterminal *Predicate*) werden in der Form `praedikatName(argument1,...,argumentN)` geschrieben, wobei nullstellige Prädikate als `praedikatName()` geschrieben werden. Sowohl der Name des Prädikats als auch die Argumente sind dabei Strings aus Buchstaben, beginnend mit einem Kleinbuchstaben.

Die Grammatik für Formeln ist in dieser Form mehrdeutig. Daher müssen noch die Prioritäten und die Assoziativitäten der Operatoren festgelegt werden.

Betrachten wir nun den Parser.

### 5.2.2 Die Direktiven

Es kann oft günstig sein, mehrere Parser in einer Grammatik zu nutzen, beispielsweise für einen Interpreter, der im Stande sein soll, ganze Dateien sowie einzelne Ausdrücke zu parsen. In dem Fall ist es wahrscheinlich, dass die Grammatik für beide Parser die gleiche ist und daher bietet es sich an, mehrere Parser zu Beginn der Grammatik zu definieren.
In unserem Fall sind die verschiedenen Parser nur für Testzwecke da, entscheidend ist der Hauptparser '`parser`'.

```
%name parser START
%name parseSets  Sets
%name parseRels  Rels
%name parseProps  Props
%name parseSolve  Solve
```

Als Nächstes definieren wir den Typ von Token, die der Parser erkennen soll und definieren diese Token:

```
%tokentype { Token }
%token
        '{'             { TokenSymbol "{" }
        '}'             { TokenSymbol "}" }
        ':='            { TokenSymbol ":=" }
        ':'             { TokenSymbol ":" }
        '('             { TokenSymbol "(" }
```

```
')'             { TokenSymbol ")" }
'X'             { TokenSymbol "X" }
','             { TokenSymbol "," }
'UNION'                 { TokenKeyword "UNION" }
'RELATION'              { TokenKeyword "RELATION" }
'SET'                   { TokenKeyword "SET" }
'SUBSET'                { TokenKeyword "SUBSET" }
'PROPOSITION'           { TokenKeyword "PROPOSITION" }
'FOR'                   { TokenKeyword "FOR" }
'ALL'                   { TokenKeyword "ALL" }
'EXISTS'                { TokenKeyword "EXISTS" }
'EXACTLY'               { TokenKeyword "EXACTLY" }
'A'                     { TokenKeyword "A" }
'AT'                    { TokenKeyword "AT" }
'MOST'                  { TokenKeyword "MOST" }
'LEAST'                 { TokenKeyword "LEAST" }
'SUCH'                  { TokenKeyword "SUCH" }
'HOLDS'                 { TokenKeyword "HOLDS" }
'THAT'                  { TokenKeyword "THAT" }
'AND'                   { TokenKeyword "AND" }
'OR'                    { TokenKeyword "OR" }
'XOR'                   { TokenKeyword "XOR" }
'NOT'                   { TokenKeyword "NOT" }
'IMPLIES'               { TokenKeyword "IMPLIES" }
'SOLVE'                 { TokenKeyword "SOLVE" }
'IFF'                   { TokenKeyword "IFF" }
 IdG                    { TokenIdentifier1 $$ }
 IdK                    { TokenIdentifier2 $$ }
 Number                 { TokenInt $$ }
```

Anschließend legen wir die Bindungsstärken fest, damit wir später korrekt klammern können. Dabei bindet `NOT` stärker als `AND`, `AND` stärker als `OR` und `XOR`, und so weiter. Die Begriffe `%left`, `%right` und `%nonassoc` geben dabei an, ob ein Operator linksassoziativ, rechtsassoziativ oder gleichwertig ausgewertet werden soll.

```
%nonassoc 'THAT' 'HOLDS'
%left ',' 'X'
%left     'UNION'
%right    'IFF'
%right    'IMPLIES'
%left     'OR' 'XOR'
%left     'AND'
%nonassoc 'NOT'
```

### 5.2.3 Die Grammatik

Betrachten wir nun die kontextfreie Grammatik des Parsers. Sie besitzt die Direktive `START` als Startsymbol und liefert ein Vierer-Tupel zurück bestehend aus:

- eine Liste von Mengenbeschreibungen vom Typ Set
- eine Liste von Relationen
- eine Liste von nummerierten Aussagen
- eine Liste von Zahlen, die besagen, welche Aussagen gelöst werden sollen

```
START :: { ([Set], [(String,[String])], [(Int,RFormula String)], [Int]) }
START : Sets Rels Props Solve    { ($1,$2,$3,$4)   }
```

Als Nächstes sehen wir die Grammatik für Mengenausdrücke. Erlaubt sind Eingaben der Form:

```
SET: Mengenname := {konstante1,...,konstanteN}
  und
SET: Mengenname := Mengenname1 UNION ... UNION MengennameN
```

Dabei können Mengen auch als Vereinigung anderer Mengen dargestellt werden, daher können damit allein noch nicht alle Fehler abgefangen werden, falls die verwendeten Mengen beispielsweise nicht definiert sind.
Daher wird als Ausgabe der Datentyp Set verwendet, der im Grunde nur den Syntaxbaum darstellt. Später werden dann die erforderlichen Überprüfungen auf dem Syntaxbaum durchgeführt. Daher sieht die Grammatik wie folgt aus:

```
Sets :: { [Set] }
Sets :  Set         { [$1]     }
     |  Set Sets    { $1:$2    }

Set  : 'SET' ':' IdG ':=' Objects    { Def ($3,$5) }
Set  : 'SET' ':' IdG ':=' Unions     { Union $3 $5 }

Unions : IdG 'UNION' Unions   { $1:$3 }
       | IdG                  { [$1]  }

Objects :  '{' List '}'   { $2 }
List : IdK                { [$1] }
     | IdK ',' List       { $1:$3 }
```

Kommen wir nun zu der Grammatik der Aussagen. Eine Aussage wird von folgender Form erwartet: `PROPOSITION Zahl: Formel`. Dabei steht `Formel` für beliebige Formeln der Rätsellogik und die Rückgabe ist eine Liste von Aussagen, wobei jedes Listenelement ein Paar (Zahl,Formel) ist, da die Formeln nummeriert sind. Die Formel ist vom Typ `RFormula String`, d.h. bereits passend für die Repräsentation von Logicals.

```
Props : Prop         { [$1]   }
      | Prop Props   { $1:$2  }

Prop : 'PROPOSITION' Number ':' Formel    { ($2,$4)   }
```

Nun betrachten wir die Grammatik für Formeln: Die Syntax orientiert sich dabei - stark vereinfacht - an der englischen Sprache. Aussagenlogische Verknüpfungen werden mit `AND, OR, NOT, IMPLIES` und `IFF` dargestellt.
Quantoren werden folgendermaßen dargestellt:

- Eine allquantifizierte Aussage der Form 'forall x in s.F' wird als 'FOR ALL S x HOLDS F' repräsentiert, wobei S für die Sorte und F für die Formel steht.
- Eine existenzquantifizierte Aussage ohne Anzahlbeschränkung 'exists x in s.F' wird als 'EXISTS A S x SUCH THAT F' repräsentiert.
- Der Existenzquantor mit Anzahlbeschränkung „mindestens" wird als 'EXISTS AT LEAST n S x SUCH THAT F' dargestellt.
- Der Existenzquantor mit Anzahlbeschränkung „höchstens" wird als 'EXISTS AT MOST n S x SUCH THAT F' dargestellt.
- Der Existenzquantor mit Anzahlbeschränkung „genau ein" wird als 'EXISTS EXACTLY n S x SUCH THAT F' dargestellt.

```
Formel :: { RFormula String }
Formel : Predicate                { $1                      }
       | Formel 'AND' Formel      { Rand [$1,$3]            }
       | Formel 'OR' Formel       { Ror  [$1,$3]            }
       | Formel 'XOR' Formel      { rxor  [$1,$3]            }
       | 'NOT' Formel             { Rnot $2                 }
       | Formel 'IMPLIES' Formel  { Rimpl $1 $3             }
       | Formel 'IFF' Formel      { Requiv $1 $3            }
       | 'EXISTS' 'A' IdG IdK 'SUCH' 'THAT' Formel
       { Rex $4 $3 $7   }
       | 'EXISTS' 'AT' 'LEAST' Number IdG IdK 'SUCH' 'THAT' Formel
       { RexAtLeast $4 $6 $5 $9 }
       | 'EXISTS' 'AT' 'MOST' Number IdG IdK 'SUCH' 'THAT' Formel
```

```
      { RexAtMost $4 $6 $5 $9  }
      | 'EXISTS' 'EXACTLY' Number IdG IdK 'SUCH' 'THAT' Formel
      { RexExactly $3 $5 $4 $8 }
      | 'FOR' 'ALL' IdG IdK 'HOLDS' Formel
      { Rall $4 $3 $6          }
      | '(' Formel ')' {$2}
```

Kommen wir nun zu der Grammatik der Prädikate. Sie werden in der Form `name(arg1,...,argn)` repräsentiert und nullstellige Prädikate müssen als `name()` geschrieben werden:

```
Predicate : IdK '(' ')'          { Rpredicate $1 []      }
          | IdK '(' ArgList ')'  { Rpredicate $1 $3      }

ArgList   : IdK              { [$1]                }
          | IdK ',' ArgList  { $1:$3               }
```

Abschließend für die Grammatik betrachten wir die Solve-Anweisung. Sie ist von der Form `SOLVE i AND j AND k ... AND m`.
Im Anschluss sollen die Aussagen mit den entsprechenden Nummern $i, j, k, ..., m$ verundet gelöst werden.
Der Parser liefert an dieser Stelle nur die Liste der Zahlen:

```
Solve     : 'SOLVE' AndList        { $2                  }
          | 'SOLVE' 'ALL'          { [] }
AndList   : Number                 { [$1]                }
          | Number 'AND' AndList   { $1:$3                }
```

Damit hätten wir die Grammatik bestimmt und betrachten nun die Datentypen.

### 5.2.4 Die Datentypen

An dieser Stelle führen wir die verschiedenen Datentypen ein, zunächst für die Mengenbeschreibungen und die happy-Error-Funktion:

```
-- Der Datentyp fuer die Mengenbeschreibungen:

data Set = Def (Sortname,[Element])
         | Union Sortname [Sortname]
 deriving(Eq,Show)
```

```
-- Die happy-Error Funktion

happyError :: [Token] -> a
happyError [] = error "parse error: unerwartetes Ende"
happyError xs = error $ "parse error:" ++ concat
                ( intersperse " " (map showToken xs))
```

Dann noch der Datentyp für die Tokens:

```
data Token = TokenKeyword String
           | TokenSymbol  String
           | TokenIdentifier1 String
           | TokenIdentifier2 String
           | TokenInt Int
 deriving(Show)

showToken (TokenKeyword k)     = k
showToken (TokenSymbol k)      = k
showToken (TokenIdentifier1 k) =  k
showToken (TokenIdentifier2 k) =  k
showToken (TokenInt k)         =  show k
```

Wir unterscheiden daher im Wesentlichen fünf Tokens: Schlüsselworte, Symbole, Zahlen und Namen, wobei TokenIdentifier1 für Namen verwendet wird, die mit einem Großbuchstaben beginnen und TokenIdentifier2 für Namen, die mit einem Kleinbuchstaben beginnen.

Und abschließend die Schlüsselworte, die `keywords`:

```
keywords = ["UNION", "SET", "SUBSET", "RELATION", "PROPOSITION",
"EXISTS", "EXACTLY", "FOR", "ALL", "SUCH", "THAT", "AND", "OR",
"NOT", "SOLVE", "IMPLIES", "IFF","AT",  "MOST", "LEAST", "A",
"HOLDS", "XOR"]
```

### 5.2.5 Der Lexer

Abschließend für die Implementierung des Parsers betrachten wir den Lexer, den vorverarbeitenden Teil des Parsers.
Er entfernt Leerzeichen zwischen den erlaubten Begriffen und verwandelt die Symbole in `TokenSymbol`s. Durch `--` können Zeilenkommentare in die Eingabe geschrieben werden, der Lexer entfernt alle Zeichen bis zum Zeilenende:

```
lexer :: String -> [Token]
lexer [] = []
lexer ('-':'-':xs) = lexer $  unlines $ tail (lines xs)
lexer cs
 | any (\k -> k `isPrefixOf` cs && k == (takeWhile (isLetter) cs)) keywords =
    let k = head (filter (\k -> k `isPrefixOf` cs) keywords)
        l = length k
    in (TokenKeyword k):(lexer (drop l cs))
lexer ('X':xs) = (TokenSymbol "X"):(lexer xs)
lexer ('{':xs) = (TokenSymbol "{"):(lexer xs)
lexer ('}':xs) = (TokenSymbol "}"):(lexer xs)
lexer ('(':xs) = (TokenSymbol "("):(lexer xs)
lexer (')':xs) = (TokenSymbol ")"):(lexer xs)
lexer (',':xs) = (TokenSymbol ","):(lexer xs)
lexer (':':'=':xs) = (TokenSymbol ":="):(lexer xs)
lexer (':':xs) = (TokenSymbol ":"):(lexer xs)

lexer (x:xs)
 | isSpace x = lexer xs
 | isUpper x = let (v,rest) = span isLetter xs
               in (TokenIdentifier1 (x:v)) : (lexer rest)
 | isLower x = let (v,rest) = span isLetter xs
               in (TokenIdentifier2 (x:v)) : (lexer rest)
 | isDigit x = let (v,rest) = span isDigit xs
               in (TokenInt (read $ x:v)) : (lexer rest)
 | otherwise = error ("parse error, can't lex symbol " ++ show x)
```

Direkt im Anschluss an die lexikalische Analyse beginnt die syntaktische Analyse durch den Parser.

## 5.3 Testen des Parsers

Nun sollten wir testen, ob der Parser wie gewünscht funktioniert, welche Fehlermeldungen er wirft und welche Fehler er nicht erkennt.

Für einfachere Tests ist unser Parser wie folgt in Einzelparser zerlegt:

```
parse1  = parseSets  . lexer
parse2  = parseRels  . lexer
parse3  = parseProps . lexer
```

Auf diese Weise können wir bequem nur bestimmte Teile der Eingabe testen, wobei `parse1` die Mengeneingaben überprüft, `parse2` die Eingabe der Relationen und `parse3` die Propositionen.

Betrachten wir zunächst ein paar verschiedene Mengeneingaben - den Kommentaren lassen sich die Erklärungen für die Fehlermeldungen entnehmen:

```
*Parser> parse1 "SET: Mann := {bert}"
[Def ("Mann",["bert"])] -- hier wird die Eingabe korrekt verarbeitet

*Parser> parse1 "SET: MANN := {bert}"
[Def ("MANN",["bert"])] -- Sorten müssen groß beginnen, dürfen aber
auch komplett groß geschrieben werden

*Parser> parse1 "SET: Mann := {Bert}"
*** Exception: parse error:Bert } -- Fehler, da Bert groß geschrieben

*Parser> parse1 "SET: Mann := {bert manuel}"
*** Exception: parse error:manuel } -- Fehler, da kein Komma zwischen
bert und manuel
```

Als Nächstes betrachten wir die Tests zu den Prädikaten, ebenfalls mit Kommentaren:

```
*Parser> parse2 "RELATION: jung SUBSET (Frau)"
[("jung",["Frau"])] -- korrekte Eingabe

*Parser> parse2 "RELATION: jung Subset (Frau)"
*** Exception: parse error:Subset ( Frau ) -- Fehler, da das
Schlüsselwort 'Subset' komplett groß geschrieben werden muss

*Parser> parse2 "RELATION: Jung SUBSET (Frau)"
*** Exception: parse error:Jung SUBSET ( Frau ) -- Fehler, da
Prädikat 'jung' klein geschrieben werden muss

*Parser> parse2 "RELATION: verliebt SUBSET Mann X Frau"
*** Exception: parse error:X Mann -- bei mehrstelligen Prädikaten
müssen die Sorten in Klammern stehen

*Parser> parse2 "RELATION: verliebt SUBSET (Mann X Frau)"
[("verliebt",["Mann","Frau"])] -- korrekte Eingabe eines
mehrstelligen Prädikats
```

```
*Parser> parse2 "RELATION: verliebt SUBSET (Maenner ! Frauen)"
*** Exception: parse error, can't lex symbol '!' -- Fehlermeldung
des Lexers wegen ungültigem Symbol
```

Betrachten wir nun noch analoge Parsertests zu den Propositionen:

```
*Parser> parse3 "PROPOSITION 1: NOT faul (elly)"
[(1,Rnot (Rpredicate "faul" ["elly"]))] -- funktioniert, da
korrekte Eingabe

*Parser> parse3 "PROPOSITION 1: NOT faul (Elly)"
*** Exception: parse error:Elly ) -- Fehler, da Konstante groß geschrieben

*Parser> parse3 "PROPOSITION 1: NOT faul((elly))"
*** Exception: parse error:( elly ) ) -- Fehler, da zu viele Klammern

*Parser> parse3 "PROPOSITION 1: EXISTS A Mann x faul(x)"
*** Exception: parse error:faul ( x ) -- Fehler, da Schlüsselworte
'SUCH THAT' fehlen

*Parser> parse3 "PROPOSITION 1: EXISTS A Mann x SUCH THAT faul(x)"
[(1,Rex "x" "Mann" (Rpredicate "faul" ["x"]))] -- korrekte Eingabe
```

Nun können aber auch noch Fehler derart auftreten, dass Sorten, Prädikate oder Konstanten in Propositionen verwendet werden, die in der Signatur nicht definiert wurden. Daher müssen wir auch den gesamten Parser testen:

```
*Parser> parse "SET: Mann := {bert} RELATION: jung SUBSET Mann
PROPOSITION 1:jung (bert) SOLVE 1"
([Def ("Mann",["bert"])],[("jung",["Mann"])],[(1,Rpredicate "jung"
["bert"])],[1])

*Parser> parse "SET: Mann := {bert} RELATION: jung SUBSET Mann
PROPOSITION 1:jung (bert)"
*** Exception: parse error: unerwartetes Ende -- Fehler, wenn
ein Teil fehlt, wie in diesem Fall 'SOLVE'

*Parser> parse "SET: Mann := {bert} PROPOSITION 1:jung (bert) SOLVE 1"
*** Exception: parse error:PROPOSITION 1 : jung ( bert ) SOLVE 1
-- ebenfalls ein Fehler, da der Teil der Prädikate, also 'RELATION',
fehlt
```

Als Nächstes übergeben wir in Kapitel 6 zunächst Rätsel-Testfälle ohne Benutzungsschnittstelle dem Programm und überprüfen den Testvorgang bezüglich Laufzeit und Speicherplatz sowie die Ergebnisse der Tests.
Anschließend testen wir die Testfälle der Benutzungsschnittstelle und vergleichen die Performanz zu den vorherigen Tests.

# Kapitel 6

# Tests und Ergebnisse

In diesem Kapitel betrachten wir unterschiedliche Rätsel, die wir als Testfälle für das Programm kodieren und untersuchen die Ergebnisse.

Dafür stelle ich zunächst einige Rätsel vor, von denen manche bereits in vorigen Kapiteln erwähnt wurden und wieder aufgegriffen werden, und andere neu eingeführt werden.
Zu jedem Rätsel betrachten wir auch die Laufzeit und die Größe der Klauselmenge, die Anzahl der Variablen und der Klauseln.

Dieses Kapitel ist mit seinen Tests in zwei große, bzw. vier kleinere Teile aufgeteilt:

- In Abschnitt 6.1 werden Tests des Programms, das das Erfüllbarkeitsproblem der Rätsellogik löst, vorgestellt und ihre Ergebnisse betrachtet.
- In Abschnitt 6.2 fassen wir die Erkenntnisse über die Tests des Programms zusammen.
- In Abschnitt 6.3 betrachten wir die vollständigen Tests, die auch die intelligente Benutzungsschnittstelle miteinbeziehen und auch hier die Ergebnisse überprüfen.
- Und in Abschnitt 6.4 ziehen wir abschließend Erkenntnisse aus den Tests der Benutzungsschnittstelle.

Die folgenden Tests wurden mit dem Vierkernprozessor Intel(R) Core(TM) i5-2400 CPU mit 3,10GHz und einem Arbeitsspeicher von 4 GB auf einem 64 Bit-Betriebssystem (Windows) ausgeführt.
Kompiliert wurde mit dem Aufruf `ghc -O`, wobei `-O` ein Optimierungs-Flag ist.
Als SAT-Solver wurde in diesem Fall SAT4J[1] verwendet.

[1] Siehe Kapitel 2.4.1.

## 6.1 Testen des Programms

Wir betrachten in diesem Abschnitt vier Rätsel und übergeben sie dem Kernprogramm, das die Modelle berechnet. Die intelligente Benutzungsschnittstelle lassen wir hier außen vor und benutzen und testen wir dann separat in Abschnitt 6.3.

### 6.1.1 Schwimmerrätsel

Betrachten wir als einführendes Beispiel ein kleines, eindeutig lösbares Rätsel:

| |
|---|
| Es gibt eine Dame namens Martha und zwei Männer namens Daniel und Manuel. |
| Einer der beiden Männer schwimmt gerne. |
| Der Mann, der mit Martha verheiratet ist, schwimmt gerne. |
| Wer ist mit Martha verheiratet und schwimmt? |

Tabelle 6.1: Schwimmerrätsel

Als Testfall kodiert sieht das Rätsel folgendermaßen aus:

```
testSchwimmer :: Logical
testSchwimmer = (preamble, rformel)
 where
   preamble = Preambel sorts predicates
   sorts = [("Maenner",["Daniel","Manuel"]),
            ("Frauen", ["Martha"])]
   predicates = [("schwimmt",["Maenner"]),
                 ("verheiratet",["Maenner","Frauen"])]
   rformel = Rand [Rex "x" "Maenner"
             (Rpredicate "verheiratet" ["x","Martha"]),
             Rall "x" "Maenner"
             (Rimpl (Rpredicate "verheiratet" ["x","Martha"])
             (Rpredicate "schwimmt" ["x"])),
             (Rnot (Rpredicate "schwimmt" ["Daniel"]))]
```

#### Testaufruf Schwimmerrätsel

Rufen wir nun die Funktion

```
*Main> putStrLn $ formatAll 1 $ getResult testSchwimmer
```

im Programm auf, so erhalten wir als Ergebnis:

```
==========================
Modell 1
==========================
   schwimmt = {(Manuel)}

   verheiratet = {(Manuel,Martha)}
```

Und die Antwort lautet, wie wir uns denken konnten: Manuel schwimmt und ist folglich mit Martha verheiratet.

Die Laufzeit beträgt 0,47 Sekunden, der verwendete Speicherplatz liegt bei 1 MB. Größe der Klauselmenge: Es gibt 4 Variablen und 5 Klauseln.

### 6.1.2 Freundesrätsel

Betrachten wir nun das in Abschnitt 3.6 eingeführte Freundesrätsel:

| |
|---|
| Elly und Vanessa sind Blondinen. |
| Klaudia und Louisa sind Brünetten. |
| Alle jungen Blondinen sind mit einer Brünetten befreundet. |
| Vanessa und Elly sind jung. |
| Vanessa ist nicht mit Klaudia befreundet. |
| Elly ist nicht mit Louisa befreundet. |
| Wer ist mit wem befreundet? |

Tabelle 6.2: Freundesrätsel

Dieses Rätsel sieht als Testfall kodiert folgendermaßen aus:

```
testElly :: Logical
testElly = (preamble,rformel)
 where
   preamble = Preambel sorts predicates
   sorts  = [("Blondinen",["Elly","Vanessa"]),
             ("Bruenetten", ["Klaudia","Louisa"])]
   predicates = [("jung",["Blondinen"]),
   ("befreundet",["Blondinen", "Bruenetten"])]
```

```
rformel   =
 Rand [
  (Rpredicate "jung" ["Vanessa"]),
  Rnot (Rpredicate "befreundet" ["Vanessa","Klaudia"]),
  (Rpredicate "jung" ["Elly"]),
  Rnot (Rpredicate "befreundet" ["Elly","Louisa"]),
  Rall "x" "Blondinen"
    (Rex "y" "Bruenetten"
     (Rimpl (Rpredicate "jung" ["x"])
         ((Rpredicate "befreundet" ["x","y"])))))]
```

#### Testaufruf Freundesrätsel

Um das Ergebnis zu erhalten, rufen wir folgende Funktion auf:

```
*Main> putStrLn $ formatAll 1 $ getResult testElly
```

Und erhalten damit folgendes Modell:

```
============================
Modell 1
============================
   befreundet = {(Elly,Klaudia),
                 (Vanessa,Louisa)}

   jung = {(Elly),
           (Vanessa)}
```

Somit ist Elly mit Klaudia und Vanessa mit Louisa befreundet.

Die Laufzeit beträgt hier 0,45 Sekunden und der verwendete Speicherplatz 1 MB. Bei der Berechnung gibt es 8 Variablen und 19 Klauseln.

### 6.1.3 Zebrarätsel

Die wohl bekannteste Logelei ist das Zebrarätsel, das auf Albert Einstein zurückgeführt wird und daher auch Einsteins Rätsel genannt wird, auch, wenn es dafür keine Belege gibt.
Es gibt verschiedene Versionen dieses Rätsels, wir betrachten hier die Version, die im Jahr 1962 im *Life International Magazine* veröffentlicht wurde[2]:

[2]Quelle: [Mül].

| |
|---|
| Es gibt fünf Häuser. |
| Der Engländer wohnt im roten Haus. |
| Der Spanier hat einen Hund. |
| Kaffee wird im grünen Haus getrunken. |
| Der Ukrainer trinkt Tee. |
| Das grüne Haus ist direkt rechts vom weißen Haus. |
| Der Raucher von Old-Gold-Zigaretten hält Schnecken als Haustiere. |
| Die Zigaretten der Marke Kools werden im gelben Haus geraucht. |
| Milch wird im mittleren Haus getrunken. |
| Der Norweger wohnt im ersten Haus. |
| Der Mann, der Chesterfields raucht, wohnt neben dem Mann mit dem Fuchs. |
| Die Marke Kools wird geraucht im Haus neben dem Haus mit dem Pferd. |
| Der Lucky-Strike-Raucher trinkt am liebsten Orangensaft. |
| Der Japaner raucht Zigaretten der Marke Parliaments. |
| Der Norweger wohnt neben dem blauen Haus. |
| Wer trinkt Wasser? Wem gehört das Zebra? |

Tabelle 6.3: Zebrarätsel

Die dazugehörige Kodierung als Testfall ist folgende:

```
testZebra :: Logical
testZebra = (preamble,rformel)
 where
   preamble = Preambel sorts predicates
   sorts  = [("Mann", ["Englaender", "Spanier", "Ukrainer","Norweger",
               "Japaner"])
             ,("Haus", ["gelbes_Haus","gruenes_Haus","weisses_Haus",
               "blaues_Haus","rotes_Haus"])
             ,("Position", ["1","2","3","4","5"])
             ,("Tier", ["Fuchs","Pferd","Zebra","Schnecken","Hund"])
             ,("Zigaretten", ["Chesterfields","Kools","LuckyStrike",
               "OldGold","Parliaments"])
             ,("Getraenke", ["Milch","Wasser","Tee","Kaffee","OSaft"])
             ]
   predicates = [("wohnt_in",["Mann","Haus"]),
```

```
                ("hat_Tier",["Mann","Tier"]),
                ("raucht",["Mann","Zigaretten"]),
                ("trinkt",["Mann","Getraenke"]),
                ("ist_an_Position",["Haus","Position"])
                ]
  rformel   =
   Rand [
--   Es gibt fünf Häuser.
--   Der Engländer wohnt im roten Haus.
     Rpredicate "wohnt_in" ["Englaender","rotes_Haus"]
--   Der Spanier hat einen Hund.
    ,Rpredicate "hat_Tier" ["Spanier","Hund"]
--   Kaffee wird im grünen Haus getrunken.
    ,Rex "x" "Mann" (Rand [Rpredicate "wohnt_in" ["x","gruenes_Haus"],
     Rpredicate "trinkt" ["x","Kaffee"]])
-- Der Ukrainer trinkt Tee.
    ,Rpredicate "trinkt" ["Ukrainer","Tee"]
-- Das grüne Haus ist direkt rechts vom weißen Haus.
    ,Rand [
        Rnot (Rpredicate "ist_an_Position" ["weisses_Haus","5"])
       ,Rimpl (Rpredicate "ist_an_Position" ["weisses_Haus","1"])
         (Rpredicate "ist_an_Position" ["gruenes_Haus","2"])
       ,Rimpl (Rpredicate "ist_an_Position" ["weisses_Haus","2"])
         (Rpredicate "ist_an_Position" ["gruenes_Haus","3"])
       ,Rimpl (Rpredicate "ist_an_Position" ["weisses_Haus","3"])
         (Rpredicate "ist_an_Position" ["gruenes_Haus","4"])
       ,Rimpl (Rpredicate "ist_an_Position" ["weisses_Haus","4"])
         (Rpredicate "ist_an_Position" ["gruenes_Haus","5"])
      ]
-- Der Raucher von Old-Gold-Zigaretten hält Schnecken als Haustiere.
    ,Rex "x" "Mann" (Rand [Rpredicate "raucht" ["x","OldGold"],
    Rpredicate "hat_Tier" ["x","Schnecken"]])
-- Die Zigaretten der Marke Kools werden im gelben Haus geraucht.
    ,Rex "x" "Mann" (Rand [Rpredicate "raucht" ["x","Kools"],
    Rpredicate "wohnt_in" ["x","gelbes_Haus"]])
-- Milch wird im mittleren Haus getrunken.
    ,Rall "x" "Haus"
      (Rimpl
        (Rpredicate "ist_an_Position" ["x","3"])
        (Rall "y" "Mann"
          (Rimpl
            (Rpredicate "wohnt_in" ["y","x"])
            (Rpredicate "trinkt" ["y","Milch"]))))
    ,Rall "x" "Haus"
```

```
       (Rimpl
         (Rpredicate "ist_an_Position" ["x","1"])
         (Rpredicate "wohnt_in" ["Norweger","x"])
        )
-- Der Mann, der Chesterfields raucht, wohnt neben dem Mann
-- mit dem Fuchs.
     ,Rex "h1" "Haus" (Rex "h2" "Haus" (Rex "x" "Mann" (Rex "y" "Mann"
     (Rand [Rimpl (Rpredicate "ist_an_Position" ["h1","1"])
           (Rpredicate "ist_an_Position" ["h2","2"])
          ,Rimpl (Rpredicate "ist_an_Position" ["h1","2"])
           (Rpredicate "ist_an_Position" ["h2","3"])
          ,Rimpl (Rpredicate "ist_an_Position" ["h1","3"])
           (Rpredicate "ist_an_Position" ["h2","4"])
          ,Rimpl (Rpredicate "ist_an_Position" ["h1","4"])
           (Rpredicate "ist_an_Position" ["h2","5"])
          ,Rnot  (Rpredicate "ist_an_Position" ["h1","5"])
          ,Rpredicate "raucht" ["x","Chesterfields"]
          ,Rpredicate "hat_Tier" ["y","Fuchs"],
           Ror [Rand [Rpredicate "wohnt_in" ["x","h1"],
           Rpredicate "wohnt_in" ["y","h2"]],
                Rand [Rpredicate "wohnt_in" ["y","h1"],
                Rpredicate "wohnt_in" ["x","h2"]]]
          ])))
-- Die Marke Kools wird geraucht im Haus neben dem Haus mit dem Pferd.
     ,Rex "h1" "Haus" (Rex "h2" "Haus" (Rex "x" "Mann" (Rex "y" "Mann"
     (Rand [Rimpl (Rpredicate "ist_an_Position" ["h1","1"])
           (Rpredicate "ist_an_Position" ["h2","2"])
          ,Rimpl (Rpredicate "ist_an_Position" ["h1","2"])
           (Rpredicate "ist_an_Position" ["h2","3"])
          ,Rimpl (Rpredicate "ist_an_Position" ["h1","3"])
           (Rpredicate "ist_an_Position" ["h2","4"])
          ,Rimpl (Rpredicate "ist_an_Position" ["h1","4"])
           (Rpredicate "ist_an_Position" ["h2","5"])
          ,Rnot  (Rpredicate "ist_an_Position" ["h1","5"])
          ,Rpredicate "raucht" ["x","Kools"]
          ,Rpredicate "hat_Tier" ["y","Pferd"]
          ,Ror [Rand [Rpredicate "wohnt_in" ["x","h1"],
                Rpredicate "wohnt_in" ["y","h2"]],
                Rand [Rpredicate "wohnt_in" ["y","h1"],
                Rpredicate "wohnt_in" ["x","h2"]]]
          ])))
-- Der Lucky-Strike-Raucher trinkt am liebsten Orangensaft.
    ,Rall "x" "Mann" (Rimpl (Rpredicate "raucht" ["x", "LuckyStrike"])
     (Rpredicate "trinkt" ["x","OSaft"]))
```

```
-- Der Japaner raucht Zigaretten der Marke Parliaments.
    ,(Rpredicate "raucht" ["Japaner","Parliaments"])
-- Der Norweger wohnt neben dem blauen Haus.
    ,Rex "h1" "Haus" (
     Rand [
           Rimpl (Rpredicate "ist_an_Position" ["blaues_Haus","1"])
            (Rpredicate "ist_an_Position" ["h1","2"])
           ,Rimpl (Rpredicate "ist_an_Position" ["blaues_Haus","2"])
            (Ror [Rpredicate "ist_an_Position" ["h1","1"],
            Rpredicate "ist_an_Position" ["h1","3"]])
           ,Rimpl (Rpredicate "ist_an_Position" ["blaues_Haus","3"])
            (Ror [Rpredicate "ist_an_Position" ["h1","2"],
            Rpredicate "ist_an_Position" ["h1","4"]])
           ,Rimpl (Rpredicate "ist_an_Position" ["blaues_Haus","4"])
            (Ror [Rpredicate "ist_an_Position" ["h1","3"],
            Rpredicate "ist_an_Position" ["h1","5"]])
           ,Rimpl (Rpredicate "ist_an_Position" ["blaues_Haus","5"])
            (Rpredicate "ist_an_Position" ["h1","4"])
           ,Rpredicate "wohnt_in" ["Norweger","h1"]
          ])
-- Side conditions
    ,RexExactly 1 "x" "Haus" (Rpredicate "ist_an_Position" ["x","1"])
    ,RexExactly 1 "x" "Haus" (Rpredicate "ist_an_Position" ["x","2"])
    ,RexExactly 1 "x" "Haus" (Rpredicate "ist_an_Position" ["x","3"])
    ,RexExactly 1 "x" "Haus" (Rpredicate "ist_an_Position" ["x","4"])
    ,RexExactly 1 "x" "Haus" (Rpredicate "ist_an_Position" ["x","5"])
    ,RexExactly 1 "x" "Position" (Rpredicate "ist_an_Position"
     ["weisses_Haus","x"])
    ,RexExactly 1 "x" "Position" (Rpredicate "ist_an_Position"
      ["blaues_Haus","x"])
    ,RexExactly 1 "x" "Position" (Rpredicate "ist_an_Position"
      ["gruenes_Haus","x"])
    ,RexExactly 1 "x" "Position" (Rpredicate "ist_an_Position"
      ["gelbes_Haus","x"])
    ,RexExactly 1 "x" "Position" (Rpredicate "ist_an_Position"
      ["rotes_Haus","x"])
        -- genau 1 Mann pro Zigaretten
    ,RexExactly 1 "x" "Mann" (Rpredicate "raucht" ["x","Parliaments"])
    ,RexExactly 1 "x" "Mann" (Rpredicate "raucht" ["x","LuckyStrike"])
    ,RexExactly 1 "x" "Mann" (Rpredicate "raucht" ["x","Kools"])
    ,RexExactly 1 "x" "Mann" (Rpredicate "raucht" ["x","OldGold"])
    ,RexExactly 1 "x" "Mann" (Rpredicate "raucht" ["x","Chesterfields"])
    ,RexExactly 1 "x" "Zigaretten" (Rpredicate "raucht" ["Englaender","x"])
    ,RexExactly 1 "x" "Zigaretten" (Rpredicate "raucht" ["Spanier","x"])
```

```
,RexExactly 1 "x" "Zigaretten" (Rpredicate "raucht" ["Japaner","x"])
,RexExactly 1 "x" "Zigaretten" (Rpredicate "raucht" ["Ukrainer","x"])
,RexExactly 1 "x" "Zigaretten" (Rpredicate "raucht" ["Norweger","x"])
-- genau 1 Mann pro Getraenke
,RexExactly 1 "x" "Mann" (Rpredicate "trinkt" ["x","Wasser"])
,RexExactly 1 "x" "Mann" (Rpredicate "trinkt"["x","OSaft"])
,RexExactly 1 "x" "Mann" (Rpredicate "trinkt" ["x","Kaffee"])
,RexExactly 1 "x" "Mann" (Rpredicate "trinkt" ["x","Milch"])
,RexExactly 1 "x" "Mann" (Rpredicate "trinkt" ["x","Tee"])
,RexExactly 1 "x" "Getraenke" (Rpredicate "trinkt" ["Englaender","x"])
,RexExactly 1 "x" "Getraenke" (Rpredicate "trinkt" ["Spanier","x"])
,RexExactly 1 "x" "Getraenke" (Rpredicate "trinkt" ["Japaner","x"])
,RexExactly 1 "x" "Getraenke" (Rpredicate "trinkt" ["Ukrainer","x"])
,RexExactly 1 "x" "Getraenke" (Rpredicate "trinkt" ["Norweger","x"])
-- genau 1 Mann pro Tier
,RexExactly 1 "x" "Mann" (Rpredicate "hat_Tier" ["x","Pferd"])
,RexExactly 1 "x" "Mann" (Rpredicate "hat_Tier" ["x","Zebra"])
,RexExactly 1 "x" "Mann" (Rpredicate "hat_Tier" ["x","Hund"])
,RexExactly 1 "x" "Mann" (Rpredicate "hat_Tier" ["x","Schnecken"])
,RexExactly 1 "x" "Mann" (Rpredicate "hat_Tier" ["x","Fuchs"])
,RexExactly 1 "x" "Tier" (Rpredicate "hat_Tier" ["Englaender","x"])
,RexExactly 1 "x" "Tier" (Rpredicate "hat_Tier" ["Spanier","x"])
,RexExactly 1 "x" "Tier" (Rpredicate "hat_Tier" ["Japaner","x"])
,RexExactly 1 "x" "Tier" (Rpredicate "hat_Tier" ["Ukrainer","x"])
,RexExactly 1 "x" "Tier" (Rpredicate "hat_Tier" ["Norweger","x"])
-- genau 1 Mann pro Haus
,RexExactly 1 "x" "Mann" (Rpredicate "wohnt_in" ["x","blaues_Haus"])
,RexExactly 1 "x" "Mann" (Rpredicate "wohnt_in" ["x","weisses_Haus"])
,RexExactly 1 "x" "Mann" (Rpredicate "wohnt_in" ["x","rotes_Haus"])
,RexExactly 1 "x" "Mann" (Rpredicate "wohnt_in" ["x","gelbes_Haus"])
,RexExactly 1 "x" "Mann" (Rpredicate "wohnt_in" ["x","gruenes_Haus"])
,RexExactly 1 "x" "Haus" (Rpredicate "wohnt_in" ["Englaender","x"])
,RexExactly 1 "x" "Haus" (Rpredicate "wohnt_in" ["Spanier","x"])
,RexExactly 1 "x" "Haus" (Rpredicate "wohnt_in" ["Japaner","x"])
,RexExactly 1 "x" "Haus" (Rpredicate "wohnt_in" ["Ukrainer","x"])
,RexExactly 1 "x" "Haus" (Rpredicate "wohnt_in" ["Norweger","x"])
]
```

Ja, dieser Testfall ist größer.

### Testaufruf Zebrarätsel

Mithilfe des Aufrufs

```
*Main> putStrLn $ formatAll 1 $ getResult testZebra
```

erhalten wir die Lösung des Zebrarätsels:

```
============================
Modell 1
============================
   hat_Tier = {(Spanier,Hund),
               (Norweger,Fuchs),
               (Ukrainer,Pferd),
               (Japaner,Zebra),
               (Englaender,Schnecken)}

   ist_an_Position = {(rotes_Haus,3),
                      (gelbes_Haus,1),
                      (gruenes_Haus,5),
                      (blaues_Haus,2),
                      (weisses_Haus,4)}

   raucht = {(Norweger,Kools),
             (Ukrainer,Chesterfields),
             (Japaner,Parliaments),
             (Spanier,LuckyStrike),
             (Englaender,OldGold)}

   trinkt = {(Ukrainer,Tee),
             (Norweger,Wasser),
             (Japaner,Kaffee),
             (Spanier,OSaft),
             (Englaender,Milch)}

   wohnt_in = {(Englaender,rotes_Haus),
               (Ukrainer,blaues_Haus),
               (Norweger,gelbes_Haus),
               (Japaner,gruenes_Haus),
               (Spanier,weisses_Haus)}
```

Der Japaner hat also das Zebra und der Norweger trinkt Wasser.

Die Laufzeit beträgt für diesen Aufruf 66,68 Sekunden, und es wurde ein Speicherplatz von 46 MB benötigt.
Die Größe der Klauselmenge beträgt 5606 Variablen und 40385 Klauseln.

### 6.1.4 Ehepaarrätsel

Auch unser Rätsel mit den vier Ehepaaren aus der Sonnenstraße aus Kapitel 2.1.1 können wir von unserem Programm lösen lassen. Noch einmal zur Erinnerung:

| |
|---|
| Der Ehemann von Sandra kann nicht kochen. |
| Entweder Katja tanzt, oder Stefan spielt Go. |
| Entweder Robert kocht, oder Michael und Marta sind nicht miteinander verheiratet. |
| Die Frau von Robert schwimmt. |
| Der Ehepartner der Person, die Kakteen züchtet, spielt Fußball. |
| Entweder Sandra tanzt, oder Katjas Ehemann kann nicht kochen. |
| Entweder der Ehepartner der Person, die reitet, ist der Koch, oder Michael spielt kein Go. |
| Der Ehepartner der Person, die schwimmt, spielt Go. |
| Entweder Katjas Ehemann spielt Go, oder Dominik kocht. |
| Der Ehemann von Marta züchtet Kakteen. |
| Und wer ist jetzt mit wem verheiratet und hat welches Hobby? |

Tabelle 6.4: Ehepaarrätsel

Dieser Testfall bedarf einiger Randbedingungen und ist daher etwas größer, aber es sind Kommentare eingefügt zur Orientierung:

In Haskell kodiert sieht unser Rätsel so aus:

```
testEhepaare :: Logical
testEhepaare = (preamble,rformel)
 where
   preamble = Preambel sorts predicates
   sorts  = [("Mensch", ["Dominik", "Michael","Robert", "Stefan",
     "Eveline", "Katja", "Martha", "Sandra"])]
   predicates = [("jonglieren",["Mensch"]),
                 ("Go spielen",["Mensch"]),
```

```
              ("Fussball spielen",["Mensch"]),
              ("kochen",["Mensch"]),
              ("Kakteen zuechten",["Mensch"]),
              ("reiten",["Mensch"]),
              ("schwimmen",["Mensch"]),
              ("tanzen",["Mensch"]),
              ("maennlich", ["Mensch"]),
              ("weiblich", ["Mensch"]),
              ("verheiratet",["Mensch","Mensch"])]
rformel   =
 Rand [
  -- Der Ehemann von Sandra kann nicht kochen.
   (Rex "x" "Mensch"
     (Rand [Rnot (Rpredicate "kochen" ["x"])
           ,Rpredicate "verheiratet" ["x","Sandra"]
           ,Rpredicate "maennlich" ["x"]]))
   ,
   Rall "x" "Mensch"
     (Rimpl
         (Rand [Rpredicate "verheiratet" ["x","Sandra"],
          Rpredicate "maennlich" ["x"]])
         (Rnot (Rpredicate "kochen" ["x"])))

 -- Entweder Katja tanzt, oder Stefan spielt Go.
  ,rxor [Rpredicate "tanzen" ["Katja"]
        ,Rpredicate "Go spielen" ["Stefan"]]

   -- Entweder Robert kocht, oder Michael und Marta
   -- sind nicht miteinander verheiratet.

  ,rxor [Rpredicate "kochen" ["Robert"]
        ,Rnot (Rpredicate "verheiratet" ["Martha", "Michael"])]

 -- Die Frau von Robert schwimmt.
 ,(Rex "x" "Mensch" (Rand [Rpredicate "weiblich" ["x"],
 Rpredicate "verheiratet" ["x", "Robert"]
 ,Rpredicate "schwimmen" ["x"]]))
 ,(Rall "x" "Mensch" (Rimpl (Rand [Rpredicate "weiblich" ["x"]
 ,Rpredicate "verheiratet" ["x", "Robert"]])
 (Rpredicate "schwimmen" ["x"])))


 -- Der Ehepartner der Person, die Kakteen züchtet, spielt Fußball.
 , Rex "x" "Mensch" (Rex "y" "Mensch" (Rand
```

```
[Rpredicate "Kakteen zuechten" ["x"]
,Rpredicate "Fussball spielen" ["y"]
,Rpredicate "verheiratet" ["x","y"]]))
,Rall "x" "Mensch" (Rall "y" "Mensch"
    (Rimpl (Rand [Rpredicate "Kakteen zuechten" ["x"],
      Rpredicate "verheiratet" ["x","y"]])
      (Rpredicate "Fussball spielen" ["y"])))



-- hinzugefuegt: Wenn x mit y verheiratet und x Kakteen zuechtet
-- dann spielt y fussball
 ,Rall "x" "Mensch" (Rall "y" "Mensch"
 (Rimpl (Rand [Rpredicate "Kakteen zuechten" ["x"]
 ,Rpredicate "verheiratet" ["x","y"]])
 (Rpredicate "Fussball spielen" ["y"])))


-- Entweder Sandra tanzt, oder Katjas Ehemann kann nicht kochen.
,rxor [Rpredicate "tanzen" ["Sandra"]
,Rand [Rex "x" "Mensch" (Rand [Rpredicate "verheiratet" ["x", "Katja"]
,Rpredicate "maennlich" ["x"]
,Rnot (Rpredicate "kochen" ["x"])])
,Rall "x" "Mensch"  (Rimpl (Rand [Rpredicate "verheiratet" ["x", "Katja"]
,Rpredicate "maennlich" ["x"]]) (Rnot (Rpredicate "kochen" ["x"])))]]


-- Entweder der Ehepartner der Person, die reitet, ist der Koch,
-- oder Michael spielt kein Go.
,rxor [Rnot (Rpredicate "Go spielen" ["Michael"])
,Rand [(Rex "x" "Mensch" (Rex "y" "Mensch" (Rand [
Rpredicate "verheiratet" ["x","y"]
,Rpredicate "reiten" ["x"]
,Rpredicate "kochen" ["y"]])))
,(Rall "x" "Mensch"
(Rall "y" "Mensch" (Rimpl (Rand [
Rpredicate "verheiratet" ["x","y"]
,Rpredicate "reiten" ["x"]]) (Rpredicate "kochen" ["y"])))))]]

-- Der Ehepartner der Person, die schwimmt, spielt Go.
 ,Rex "x" "Mensch" (Rex "y" "Mensch" (Rand [
 Rpredicate "verheiratet" ["x","y"]
 ,Rpredicate "schwimmen" ["x"]
 ,Rpredicate "Go spielen" ["y"]]))
```

```
    ,Rall "x" "Mensch" (Rall "y" "Mensch" (Rimpl (Rand [
    Rpredicate "verheiratet" ["x","y"]
    ,Rpredicate "schwimmen" ["x"]])
    (Rpredicate "Go spielen" ["y"])))

-- Entweder Katjas Ehemann spielt Go, oder Dominik kocht.
  ,rxor [(Rpredicate "kochen" ["Dominik"])
  ,Rand [Rex "x" "Mensch" (Rand [Rpredicate "verheiratet" ["x", "Katja"]
  ,Rpredicate "Go spielen" ["x"], Rpredicate "maennlich" ["x"]])
  ,Rall "x" "Mensch" (Rimpl (Rand [Rpredicate "verheiratet" ["x", "Katja"]
  ,Rpredicate "maennlich" ["x"]]) (Rpredicate "Go spielen" ["x"]))]]

-- Der Ehemann von Marta zuechtet Kakteen.
 ,Rex "x" "Mensch" (Rand [Rpredicate "verheiratet" ["x", "Martha"]
 ,Rpredicate "Kakteen zuechten" ["x"], Rpredicate "maennlich" ["x"]])
 ,Rall "x" "Mensch" (Rimpl (Rand [Rpredicate "verheiratet" ["x", "Martha"]
 ,Rpredicate "maennlich" ["x"]]) (Rpredicate "Kakteen zuechten" ["x"]))
 ,Rpredicate "maennlich" ["Dominik"]
 ,Rpredicate "maennlich" ["Robert"]
 ,Rpredicate "maennlich" ["Stefan"]
 ,Rpredicate "maennlich" ["Michael"]
 ,Rpredicate "weiblich" ["Eveline"]
 ,Rpredicate "weiblich" ["Katja"]
 ,Rpredicate "weiblich" ["Martha"]
 ,Rpredicate "weiblich" ["Sandra"]

   -- Hinzugefuegt
   -- Jeder Mensch ist genau dann maennlich wenn er nicht weiblich ist
   ,Rall "x" "Mensch" (Requiv (Rpredicate "maennlich" ["x"])
   (Rnot (Rpredicate "weiblich" ["x"])))

   -- Nur Frauen und Maenner sind miteinander verheiratet
   ,Rall "x" "Mensch" (Rall "y" "Mensch"
    (Rimpl (Rpredicate "verheiratet" ["x","y"])
    (Rand [(Requiv (Rpredicate "maennlich" ["x"])
    (Rpredicate "weiblich" ["y"]))
   ,(Requiv (Rpredicate "weiblich" ["x"])
   (Rpredicate "maennlich" ["y"]))])))

   -- Niemand ist mit sich selbst verheiratet
   ,Rall "x" "Mensch" (Rnot (Rpredicate "verheiratet" ["x","x"]))
   -- verheiratet ist symmetrisch
   ,Rall "x" "Mensch" (Rall "y" "Mensch" (Rimpl
   (Rpredicate "verheiratet" ["x","y"])
```

```
    (Rpredicate "verheiratet" ["y","x"])))
    -- jede Taetigkeit nur von einem
    ,RexExactly 1 "x" "Mensch" (Rpredicate "jonglieren" ["x"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "Go spielen" ["x"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "Fussball spielen" ["x"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate  "kochen" ["x"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "Kakteen zuechten" ["x"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate  "reiten" ["x"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "schwimmen" ["x"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "tanzen" ["x"])
    -- jeder verheiratet mit genau einem/r
    ,RexExactly 1 "x" "Mensch" (Rpredicate "verheiratet" ["x","Dominik"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "verheiratet" ["x","Robert"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "verheiratet" ["x","Stefan"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "verheiratet" ["x","Michael"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "verheiratet" ["x","Eveline"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "verheiratet" ["x","Katja"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "verheiratet" ["x","Martha"])
    ,RexExactly 1 "x" "Mensch" (Rpredicate "verheiratet" ["x","Sandra"])
    -- jeder genau ein Hobby
    ,Rall "x" "Mensch" (rxor [Rpredicate "jonglieren" ["x"]
                             ,Rpredicate "Go spielen" ["x"]
                             ,Rpredicate "Fussball spielen" ["x"]
                             ,Rpredicate  "kochen" ["x"]
                             ,Rpredicate "Kakteen zuechten" ["x"]
                             ,Rpredicate  "reiten" ["x"]
                             ,Rpredicate "schwimmen" ["x"]
                             ,Rpredicate "tanzen" ["x"]])]
```

### Testaufruf Ehepaarrätsel

Und auch hier erhalten wir die richtige Lösung beim Testaufruf:

```
*Main> putStrLn $ formatAll 1 $ getResult testEhepaar
==========================
Modell 1
==========================
   Fussball spielen = {(Martha)}

   Go spielen = {(Robert)}

   Kakteen zuechten = {(Stefan)}
```

```
jonglieren = {(Eveline)}

kochen = {(Dominik)}

maennlich = {(Dominik),
             (Stefan),
             (Michael),
             (Robert)}

reiten = {(Michael)}

schwimmen = {(Sandra)}

tanzen = {(Katja)}

verheiratet = {(Robert,Sandra),
               (Stefan,Martha),
               (Michael,Katja),
               (Dominik,Eveline),
               (Katja,Michael),
               (Martha,Stefan),
               (Sandra,Robert),
               (Eveline,Dominik)}

weiblich = {(Martha),
            (Eveline),
            (Katja),
            (Sandra)}
```

Und damit haben wir alle Ehepaare und Zugehörigkeiten der Hobbys bestimmt.

Das Modell wurde in einer Laufzeit von 2,43 Sekunden berechnet mit der Verwendung von einem Speicherplatz von 9 MB.
Die Anzahl der Variablen beträgt 1361 und die der Klauseln 9417.

## 6.2 Ergebnisse der Programmtests

Betrachten wir nun die Ergebnisse unserer bisherigen Tests, bevor wir in Abschnitt 6.3 separat die intelligente Benutzungsschnittstelle testen.

Was wir aus den ausgiebigen Tests des Programms entnehmen können:

- Das Programm läuft korrekt und wie erwartet - die Tests gaben die richtigen Lösungen aus und auch die Fehlermeldungen waren für die Falschtests korrekt.
- Der Speicherplatzverbrauch ist mit etwa 1 MB für kleine Rätsel und etwa 50 MB für größere Rätsel recht gut.
- Die Laufzeit ist recht groß: Die größeren der getesteten Rätsel wie das Ehepaarrätsel und das Zebrarätsel befinden sich bereits im äußeren Bereich des Erreichbaren.

Die hohe Laufzeit entsteht vor allem dadurch, dass die erzeugten Aussagenlogischen Formeln und deren KNFs sehr viel größer sind als die Formeln der Rätsellogik.

Dies entsteht durch die „kombinatorische Explosion", wenn alle Möglichkeiten, Konstanten verschiedener Sorten zu kombinieren, beim Auflösen von Quantoren als aussagenlogische Variablen erzeugt werden.

Obwohl bereits die schnelle KNF-Berechnung verwendet wird, wird ein großer Teil der Laufzeit durch die KNF-Berechnung selbst verschlungen.

Ein weiteres Resultat der Experimente ist, dass die Formelgröße selbst als Maß für die zu erwartende Laufzeit nicht ausreicht. Obwohl beispielsweise die Formeln für Ehepaare und Zebra ungefähr gleich groß sind, ist die erzeugte KNF im Falle des Zebra-Rätsels wesentlich höher.

## 6.3 Testen der intelligenten Benutzungsschnittstelle

Nun kommen wir zu den Testfällen der intelligenten Benutzungsschnittstelle.
In Abschnitt 5.2.1 haben wir erfahren, wie die benutzerfreundlichere Eingabe der Rätsel aussieht.
Betrachten wir nun unsere Rätsel kodiert für die Benutzungsschnittstelle, beginnend mit den kleinsten Rätseln: dem Schwimmerrätsel und dem Freundesrätsel.

### 6.3.1 Schwimmerrätsel

Die Eingabe des Schwimmerrätsels aus Tabelle 6.1 für die Benutzungsschnittstelle sieht folgendermaßen aus:

```
SET: Maenner := {daniel,manuel}
SET: Frauen  := {martha}

RELATION: schwimmt SUBSET Maenner
RELATION: verheiratet SUBSET (Maenner X Frauen)
```

```
PROPOSITION 1: EXISTS A Maenner x SUCH THAT verheiratet(x,martha)
PROPOSITION 2: FOR ALL Maenner x HOLDS verheiratet(x,martha)
               IMPLIES schwimmt(x)
PROPOSITION 3: NOT schwimmt(daniel)

SOLVE 1 AND 2 AND 3
```

Wir sehen, dass mit `SET` die Sorten der `Maenner` und der `Frauen` definiert werden und zugleich auch die Konstanten `daniel, manuel` und `martha`. Somit ist auch die Zugehörigkeit der Konstanten zu den Sorten definiert.

`RELATION` definiert die Prädikate `schwimmt` und `verheiratet` und legt fest, dass das Prädikat `schwimmt` nur für Konstanten der Sorte `Maenner` gelten kann und dass `verheiratet` ein zweistelliges Prädikat ist, das aus einem Tupel besteht, welches eine Konstante der Sorte `Maenner` und eine Konstante der Sorte `Frauen` enthält.

Die Propositionen enthalten die verschiedenen Formeln, die am Ende durch `SOLVE` verundet und gelöst werden.

So besagt die erste `PROPOSITION`, dass es einen Mann gibt, der mit Martha verheiratet ist.
Die zweite besagt, dass alle Männer, die mit Martha verheiratet sind, schwimmen.
Und die dritte schließt Daniel als Schwimmer aus.

### Testaufruf Schwimmerrätsel

Den kodierten Testfall speichern wir nun in der Text-Datei `schwimmt.txt`, die wir der `Interface.exe` übergeben.

```
Z:\Masterarbeit\Programm>Interface.exe schwimmt.txt

============================
Modell 1
============================
   schwimmt = {(manuel)}

   verheiratet = {(manuel,martha)}
```

Auch mit der Benutzungsschnittstelle braucht dieses Rätsel 1 MB Speicherplatz und 0,54 Sekunden Laufzeit, ähnlich wie in Abschnitt 6.1.

### 6.3.2 Freundesrätsel

Betrachten wir nun unsere Eingabe des Freundesrätsels aus Tabelle 6.2:

```
SET: Blondinen  := {elly, vanessa}
SET: Bruenetten := {klaudia, louisa}

RELATION: jung SUBSET Blondinen
RELATION: befreundet SUBSET (Blondinen X Bruenetten)

PROPOSITION 1: jung(vanessa) AND jung(elly)
PROPOSITION 2: NOT befreundet(vanessa,klaudia) AND
               NOT befreundet(elly,louisa)
PROPOSITION 3: FOR ALL Blondinen x HOLDS EXISTS A Bruenetten y
               SUCH THAT jung(x) IMPLIES befreundet(x,y)

SOLVE 1 AND 2 AND 3
```

Auch hier werden mit `SET` die Sorten definiert, diesmal Blondinen und Brünetten.

`RELATION` definiert das einstellige Prädikat `jung` auf der Menge der Blondinen und das zweistellige Prädikat `befreundet` als Tupel einer Blondine und einer Brünette.

Und die `PROPOSITION`s beschreiben wieder die Bedingungen, die wir dem Rätsel entnehmen können, und werden mit `SOLVE` wieder verundet und gelöst.

#### Testaufruf Freundesrätsel

Auch hier ist der Testfall in einer Text-Datei gespeichert, daher übergeben wir der Benutzungsschnittstelle die `Freunde.txt`:

```
Z:\Masterarbeit\Programm>Interface.exe Freunde.txt

==========================
Modell 1
==========================
   befreundet = {(vanessa,louisa),
                 (elly,klaudia)}

   jung = {(vanessa),
           (elly)}
```

Der Speicherplatz beträgt hier immer noch 1 MB, so wie auch ohne die Benutzungsschnittstelle. Und die Laufzeit beträgt auch weiterhin etwa 0,5 Sekunden.

### 6.3.3 Zebrarätsel

Gehen wir nun über zu den umfangreicheren Rätseln: Das Zebrarätsel aus Tabelle 6.3.

Auch hier werden die Sorten mit `SET` definiert, die Prädikate durch `RELATION` eingeführt und die Bedingungen in den `PROPOSITION`s aufgelistet. Im Folgenden sind der Übersicht halber die umzusetzenden Sätze im Kommentar erwähnt, bevor die ihnen entsprechende `PROPOSITION` beschrieben wird.

```
SET: Mann         := {englaender,
                      spanier,
                      ukrainer,
                      norweger,
                      japaner}
SET: Haus         := {gelbeshaus,
                      grueneshaus,
                      weisseshaus,
                      blaueshaus,
                      roteshaus}
SET: Position     := {a,
                      b,
                      c,
                      d,
                      e}
SET: Tier         := {fuchs,
                      pferd,
                      zebra,
                      schnecken,
                      hund}
SET: Zigaretten   := {chesterfields,
                      kools,
                      luckystrike,
                      oldgold,
                      parliaments}
SET: Getraenke    := {milch,
                      wasser,
                      tee,
                      kaffee,
                      osaft}
```

```
RELATION: wohntin SUBSET (Mann X Haus)
RELATION: hattier SUBSET (Mann X Tier)
RELATION: raucht SUBSET (Mann X Zigaretten)
RELATION: trinkt SUBSET (Mann X Getraenke)
RELATION: istanposition SUBSET (Haus X Position)


-- Es gibt fünf Häuser. (Ist in SET umgesetzt.)

-- Der Engländer wohnt im roten Haus.
PROPOSITION 1:
  wohntin(englaender,roteshaus)


-- Der Spanier hat einen Hund.
PROPOSITION 2:
  hattier(spanier,hund)


-- Kaffee wird im grünen Haus getrunken.
PROPOSITION 3:
  EXISTS A Mann x SUCH THAT wohntin(x,grueneshaus)
  AND trinkt(x,kaffee)


-- Der Ukrainer trinkt Tee.
PROPOSITION 4:
  trinkt(ukrainer,tee)


-- Das grüne Haus ist direkt rechts vom weißen Haus.
PROPOSITION 5:
  NOT istanposition(weisseshaus,e)
  AND (istanposition(weisseshaus,a) IMPLIES istanposition(grueneshaus,b))
  AND (istanposition(weisseshaus,b) IMPLIES istanposition(grueneshaus,c))
  AND (istanposition(weisseshaus,c) IMPLIES istanposition(grueneshaus,d))
  AND (istanposition(weisseshaus,d) IMPLIES istanposition(grueneshaus,e))


-- Der Raucher von Old-Gold-Zigaretten hält Schnecken als Haustiere.
PROPOSITION 6:
  EXISTS A Mann x SUCH THAT raucht(x,oldgold)
```

```
  AND hattier(x,schnecken)


-- Die Zigaretten der Marke Kools werden im gelben Haus geraucht.
PROPOSITION 7:
  EXISTS A Mann x SUCH THAT raucht(x,kools)
  AND wohntin(x,gelbeshaus)


-- Milch wird im mittleren Haus getrunken.
PROPOSITION 8:
  FOR ALL Haus x HOLDS (istanposition(x,c) IMPLIES
  (FOR ALL Mann y HOLDS (wohntin(y,x) IMPLIES trinkt(y,milch))))


-- Der Norweger wohnt im ersten Haus.
PROPOSITION 9:
  FOR ALL Haus x HOLDS (istanposition(x,a) IMPLIES wohntin(norweger,x))


-- Der Mann, der Chesterfields raucht, wohnt neben dem Mann mit dem Fuchs.
PROPOSITION 10:
  EXISTS A Haus h SUCH THAT (EXISTS A Haus g SUCH THAT
  (EXISTS A Mann x SUCH THAT (EXISTS A Mann y SUCH THAT (
   (istanposition(h,a) IMPLIES istanposition(g,b))
 AND (istanposition(h,b) IMPLIES istanposition(g,c))
 AND (istanposition(h,c) IMPLIES istanposition(g,d))
 AND (istanposition(h,d) IMPLIES istanposition(g,e))
 AND NOT istanposition(h,e)
 AND raucht(x,chesterfields)
 AND hattier(y,fuchs)
 AND ((wohntin(x,h) AND wohntin(y,g))
  OR (wohntin(y,h) AND wohntin(x,g)))
 ))))


-- Die Marke Kools wird geraucht im Haus neben dem Haus mit dem Pferd.
PROPOSITION 11:
  EXISTS A Haus h SUCH THAT (EXISTS A Haus g SUCH THAT
  (EXISTS A Mann x SUCH THAT (EXISTS A Mann y SUCH THAT (
   (istanposition(h,a) IMPLIES istanposition(g,b))
   AND (istanposition(h,b) IMPLIES
     (istanposition(g,c) OR istanposition(g,a)))
   AND (istanposition(h,c) IMPLIES
```

```
    (istanposition(g,d) OR istanposition(g,b)))
  AND (istanposition(h,d) IMPLIES
    (istanposition(g,e) OR istanposition(g,c)))
  AND (istanposition(h,e) IMPLIES istanposition(g,d))
  AND raucht(x,kools)
  AND hattier(y,pferd)
  AND ((wohntin(x,h) AND wohntin(y,g))
   OR (wohntin(y,h) AND wohntin(x,g)))
  ))))


-- Der Lucky-Strike-Raucher trinkt am liebsten Orangensaft.
PROPOSITION 12:
  FOR ALL Mann x HOLDS (raucht(x,luckystrike) IMPLIES trinkt(x,osaft))


-- Der Japaner raucht Zigaretten der Marke Parliaments.
PROPOSITION 13:
  raucht(japaner,parliaments)


-- Der Norweger wohnt neben dem blauen Haus.
PROPOSITION 14:
  EXISTS A Haus h SUCH THAT (wohntin(norweger,h)
   AND (istanposition(blaueshaus,a) IMPLIES istanposition(h,b))
   AND (istanposition(blaueshaus,b) IMPLIES (istanposition(h,c)
     OR istanposition(h,a)))
   AND (istanposition(blaueshaus,c) IMPLIES (istanposition(h,d)
     OR istanposition(h,b)))
   AND (istanposition(blaueshaus,d) IMPLIES (istanposition(h,e)
     OR istanposition(h,c)))
   AND (istanposition(blaueshaus,e) IMPLIES istanposition(h,d)))



-- Side conditions
PROPOSITION 15:
(EXISTS EXACTLY 1 Haus x SUCH THAT (istanposition(x,a)))
AND (EXISTS EXACTLY 1 Haus x SUCH THAT istanposition(x,b))
AND (EXISTS EXACTLY 1 Haus x SUCH THAT istanposition(x,c))
AND (EXISTS EXACTLY 1 Haus x SUCH THAT istanposition(x,d))
AND (EXISTS EXACTLY 1 Haus x SUCH THAT istanposition(x,e))

AND (EXISTS EXACTLY 1 Position x SUCH THAT istanposition(weisseshaus,x))
```

```
AND (EXISTS EXACTLY 1 Position x SUCH THAT istanposition(blaueshaus,x))
AND (EXISTS EXACTLY 1 Position x SUCH THAT istanposition(grueneshaus,x))
AND (EXISTS EXACTLY 1 Position x SUCH THAT istanposition(gelbeshaus,x))
AND (EXISTS EXACTLY 1 Position x SUCH THAT istanposition(roteshaus,x))

-- genau 1 Mann pro Zigaretten
AND (EXISTS EXACTLY 1 Mann x SUCH THAT raucht(x,parliaments))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT raucht(x,luckystrike))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT raucht(x,kools))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT raucht(x,oldgold))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT raucht(x,chesterfields))
AND (EXISTS EXACTLY 1 Zigaretten x SUCH THAT raucht(englaender,x))
AND (EXISTS EXACTLY 1 Zigaretten x SUCH THAT raucht(spanier,x))
AND (EXISTS EXACTLY 1 Zigaretten x SUCH THAT raucht(japaner,x))
AND (EXISTS EXACTLY 1 Zigaretten x SUCH THAT raucht(ukrainer,x))
AND (EXISTS EXACTLY 1 Zigaretten x SUCH THAT raucht(norweger,x))

-- genau 1 Mann pro Getraenke
AND (EXISTS EXACTLY 1 Mann x SUCH THAT trinkt(x,wasser))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT trinkt(x,osaft))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT trinkt(x,kaffee))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT trinkt(x,tee))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT trinkt(x,milch))
AND (EXISTS EXACTLY 1 Getraenke x SUCH THAT trinkt(englaender,x))
AND (EXISTS EXACTLY 1 Getraenke x SUCH THAT trinkt(spanier,x))
AND (EXISTS EXACTLY 1 Getraenke x SUCH THAT trinkt(japaner,x))
AND (EXISTS EXACTLY 1 Getraenke x SUCH THAT trinkt(ukrainer,x))
AND (EXISTS EXACTLY 1 Getraenke x SUCH THAT trinkt(norweger,x))

-- genau 1 Mann pro Tier
AND (EXISTS EXACTLY 1 Mann x SUCH THAT hattier(x,pferd))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT hattier(x,zebra))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT hattier(x,hund))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT hattier(x,schnecken))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT hattier(x,fuchs))
AND (EXISTS EXACTLY 1 Tier x SUCH THAT hattier(englaender,x))
AND (EXISTS EXACTLY 1 Tier x SUCH THAT hattier(spanier,x))
AND (EXISTS EXACTLY 1 Tier x SUCH THAT hattier(japaner,x))
AND (EXISTS EXACTLY 1 Tier x SUCH THAT hattier(ukrainer,x))
AND (EXISTS EXACTLY 1 Tier x SUCH THAT hattier(norweger,x))


-- genau 1 Mann pro Haus
AND (EXISTS EXACTLY 1 Mann x SUCH THAT wohntin(x,blaueshaus))
```

```
AND (EXISTS EXACTLY 1 Mann x SUCH THAT wohntin(x,weisseshaus))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT wohntin(x,roteshaus))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT wohntin(x,gelbeshaus))
AND (EXISTS EXACTLY 1 Mann x SUCH THAT wohntin(x,grueneshaus))
AND (EXISTS EXACTLY 1 Haus x SUCH THAT wohntin(englaender,x))
AND (EXISTS EXACTLY 1 Haus x SUCH THAT wohntin(spanier,x))
AND (EXISTS EXACTLY 1 Haus x SUCH THAT wohntin(japaner,x))
AND (EXISTS EXACTLY 1 Haus x SUCH THAT wohntin(ukrainer,x))
AND (EXISTS EXACTLY 1 Haus x SUCH THAT wohntin(norweger,x))


SOLVE 1 AND 2 AND 3 AND 4 AND 5 AND 6 AND 7 AND 8 AND 9 AND 10
AND 11 AND 12 AND 13 AND 14 AND 15
```

### Testaufruf Zebrarätsel

Betrachten wir nun den Aufruf des Zebrarätsels:

```
Z:\Masterarbeit\Programm>Interface.exe einstein.txt
============================
Modell 1
============================
   hattier = {(spanier,hund),
              (ukrainer,pferd),
              (japaner,zebra),
              (englaender,schnecken),
              (norweger,fuchs)}

   istanposition = {(gelbeshaus,a),
                    (blaueshaus,b),
                    (roteshaus,c),
                    (weisseshaus,d),
                    (grueneshaus,e)}

   raucht = {(japaner,parliaments),
             (spanier,luckystrike),
             (norweger,kools),
             (englaender,oldgold),
             (ukrainer,chesterfields)}

   trinkt = {(ukrainer,tee),
             (norweger,wasser),
             (spanier,osaft),
```

```
              (japaner,kaffee),
              (englaender,milch)}

   wohntin = {(englaender,roteshaus),
              (ukrainer,blaueshaus),
              (spanier,weisseshaus),
              (norweger,gelbeshaus),
              (japaner,grueneshaus)}
```

Statt der in Abschnitt 6.1 benötigten 46 MB für die Berechnung der Lösung ohne Benutzungsschnittstelle werden hier 58 MB Speicherplatz benötigt. Auch die Laufzeit hat sich bei diesem größeren Rätsel erhöht: Statt der etwa einen Minute benötigt das Programm nun etwa 142 Sekunden, also etwas unter 2,5 Minuten.

### 6.3.4 Ehepaarrätsel

Das Ehepaarrätsel, nachzuschlagen in Tabelle 6.4, ist äquivalent zum Zebrarätsel aufgebaut und enthält ebenso die Rätselsätze im Kommentar:

```
SET: Mann   := {dominik,
                michael,
                robert,
                stefan}
SET: Frau   := {eveline,
                katja,
                martha,
                sandra}


SET: Mensch := Mann UNION Frau

RELATION: jonglieren SUBSET Mensch
RELATION: goSpielen SUBSET Mensch
RELATION: fussballSpielen SUBSET Mensch
RELATION: kochen SUBSET Mensch
RELATION: kakteenZuechten SUBSET Mensch
RELATION: reiten SUBSET Mensch
RELATION: schwimmen SUBSET Mensch
RELATION: tanzen SUBSET Mensch
RELATION: maennlich SUBSET Mensch
RELATION: weiblich SUBSET Mensch
RELATION: verheiratet SUBSET (Mensch X Mensch)
```

```
-- Der Ehemann von Sandra kann nicht kochen.
PROPOSITION 1:
  EXISTS A Mensch x SUCH THAT
    kochen(x) AND verheiratet(x,sandra) AND maennlich (x)

-- Entweder Katja tanzt, oder Stefan spielt Go.
PROPOSITION 2:
    tanzen(katja) XOR goSpielen(stefan)

-- Entweder Robert kocht, oder Michael und Martha sind
-- nicht miteinander verheiratet.
PROPOSITION 3:
    kochen(robert) XOR NOT verheiratet(martha,michael)

-- Die Frau von Robert schwimmt.
PROPOSITION 4:
   EXISTS A Mensch x SUCH THAT
     weiblich(x) AND verheiratet(x,robert) AND schwimmen(x)

-- Der Ehepartner der Person, die Kakteen züchtet, spielt Fussball.
PROPOSITION 5:
   EXISTS A Mensch x SUCH THAT
     EXISTS A Mensch y SUCH THAT
      kakteenZuechten(x) AND fussballSpielen(y) AND verheiratet(x,y)

-- Entweder Sandra tanzt, oder Katjas Ehemann kann nicht kochen.
PROPOSITION 6:
    tanzen(sandra)
    XOR
    EXISTS A Mensch x SUCH THAT
     verheiratet(x,katja) AND maennlich(x) AND NOT kochen(x)

-- Entweder der Ehepartner der Person, die reitet, ist der Koch,
-- oder Michael spielt kein Go.
PROPOSITION 7:
   NOT goSpielen(michael)
   XOR
   EXISTS A Mensch x SUCH THAT
     EXISTS A Mensch y SUCH THAT
       verheiratet(x,y) AND reiten(x) AND kochen(y)

-- Der Ehepartner der Person, die schwimmt, spielt Go.
```

```
PROPOSITION 8:
   EXISTS A Mensch x SUCH THAT
      EXISTS A Mensch y SUCH THAT
        verheiratet(x,y) AND schwimmen(x) AND goSpielen(y)

-- Entweder Katjas Ehemann spielt Go, oder Dominik kocht.
PROPOSITION 9:
   kochen(dominik)
   XOR
   EXISTS A Mensch x SUCH THAT
     verheiratet(x,katja) AND goSpielen(x) AND maennlich(x)

-- Der Ehemann von Marta züchtet Kakteen.
PROPOSITION 10:
   EXISTS A Mensch x SUCH THAT
     verheiratet(x,martha) AND kakteenZuechten(x) AND maennlich(x)


-- Sice conditions

-- Wer ist Mann, wer ist Frau
PROPOSITION 11:
   maennlich(dominik) AND maennlich(robert) AND maennlich(stefan) AND
   weiblich(eveline) AND weiblich(katja) AND weiblich(martha)
   AND weiblich(sandra)

-- Jeder Mensch ist genau dann maennlich, wenn er nicht weiblich ist.
PROPOSITION 12:
   FOR ALL Mensch x HOLDS
    maennlich(x) IFF NOT weiblich(x)

 -- Nur Frauen und Maenner sind miteinander verheiratet.
PROPOSITION 13:
   FOR ALL Mensch x HOLDS
     FOR ALL Mensch y HOLDS
        verheiratet(x,y) IMPLIES ((maennlich(x) IFF weiblich(y))
        AND (weiblich(x) IFF maennlich(y)))

-- Niemand ist mit sich selbst verheiratet.
PROPOSITION 14:
   FOR ALL Mensch x HOLDS
     NOT verheiratet(x,x)

-- verheiratet ist symmetrisch
```

```
PROPOSITION 15:
   FOR ALL Mensch x HOLDS
     FOR ALL Mensch y HOLDS
       verheiratet(x,y) IMPLIES verheiratet(y,x)

-- Jede Tätigkeit wird nur von einem ausgeführt.
PROPOSITION 16:
   (EXISTS EXACTLY 1 Mensch x SUCH THAT jonglieren(x)) AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT goSpielen(x)) AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT fussballSpielen(x)) AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT kochen(x)) AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT kakteenZuechten(x)) AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT reiten(x)) AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT schwimmen(x)) AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT tanzen(x))

-- Jeder ist mit genau einem/r verheiratet.
PROPOSITION 17:
   (EXISTS EXACTLY 1 Mensch x SUCH THAT verheiratet(x,dominik)) AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT verheiratet(x,robert))  AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT verheiratet(x,stefan))  AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT verheiratet(x,michael)) AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT verheiratet(x,eveline)) AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT verheiratet(x,katja))   AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT verheiratet(x,martha))  AND
   (EXISTS EXACTLY 1 Mensch x SUCH THAT verheiratet(x,sandra))

-- Jeder hat genau ein Hobby.
PROPOSITION 18:
   FOR ALL Mensch x HOLDS
    jonglieren(x) XOR goSpielen(x) XOR fussballSpielen(x) XOR kochen(x)
    XOR kakteenZuechten(x) XOR reiten(x) XOR schwimmen(x) XOR tanzen(x)

SOLVE 1 AND 2 AND 3 AND 4 AND 5 AND 6 AND 7 AND 8 AND 9 AND 10
AND 11 AND 12 AND 13 AND 14 AND 15 AND 16 AND 17 AND 18
```

### Testaufruf Ehepaarrätsel

Nun betrachten wir den Testaufruf des Rätsels:

```
Z:\Masterarbeit\Programm>Interface.exe Ehepaare.txt
============================
Modell 1
============================
   fussballSpielen = {(martha)}

   goSpielen = {(robert)}

   jonglieren = {(sandra)}

   kakteenZuechten = {(stefan)}

   kochen = {(dominik)}

   maennlich = {(dominik),
                (michael),
                (robert),
                (stefan)}

   reiten = {(michael)}

   schwimmen = {(eveline)}

   tanzen = {(katja)}

   verheiratet = {(katja,michael),
                  (sandra,dominik),
                  (martha,stefan),
                  (eveline,robert),
                  (stefan,martha),
                  (robert,eveline),
                  (michael,katja),
                  (dominik,sandra)}

   weiblich = {(eveline),
               (martha),
               (katja),
               (sandra)}
```

Auch bei diesem Rätsel macht sich die wachsende Laufzeit und der wachsende Speicherplatz bemerkbar:
11 MB anstatt zuvor 9 MB Speicherplatz wurden benötigt sowie 3,72 Sekunden anstatt der vorherigen 2,49 Sekunden für die Laufzeit.

## 6.4 Ergebnisse der getesteten Benutzungsschnittstelle

Wenn wir nun das Programm einschließlich der intelligenten Benutzungsschnittstelle betrachten, fallen uns neue Punkte auf:

- Die Benutzungsschnittstelle läuft korrekt und wie erwartet.
- Die Laufzeit der gleichen Rätsel hat sich bei den größeren Rätseln etwas verschlechtert, doch noch im Rahmen.
- Auch der Speicherplatzverbrauch erhöht sich bei den größeren Rätseln etwas.

Dass sich die Laufzeit und der Speicherplatzverbrauch unerheblich durch die Benutzungsschnittstelle verschlechtert, ist zu erwarten. Allerdings sind sie bei optimaler Kodierung der Testfälle zu vernachlässigen.

Bei suboptimalen Kodierungen der Testfälle können sich jedoch erhebliche Verschlechterungen der Laufzeit und des Speicherplatzverbrauchs bemerkbar machen: Gibt man beispielsweise ein paar Bedingungen redundant ein, verschlechtert sich die Laufzeit um Größenordnungen!

Beispiel: Betrachten wir die erste Bedingung des Ehepaarrätsels:

```
Der Ehemann von Sandra kann nicht kochen.
```

Wir haben nun die Möglichkeit, den Satz mit dem anzahlbeschränkten Existenzquantor `EXISTS AXACTLY 1` wie folgt zu bilden:

```
PROPOSITION 1:
  EXISTS EXACTLY 1 Mensch x SUCH THAT
    kochen(x) AND verheiratet(x,sandra) AND maennlich (x)
```

Da wir aber ohnehin in den Randbedingungen festhalten müssen, dass jeder Mensch mit genau einer Person verheiratet ist, ist der anzahlbeschränkte Existenzquantor unnötig und damit redundant. Dennoch nicht falsch: Diese Kodierung liefert ebenso das richtige Modell wie die optimierte Weise, bei der wir nur den Existenzquantor `EXISTS A` an dieser Stelle verwenden.

Aber: Die Laufzeit der auf diese Weise redundanten Kodierung als Testfall steigt damit auf mehr als das 200-fache von unter vier Sekunden auf über 13 Minuten! Der Speicherplatz explodiert dann ebenso: Statt 9 MB verzehnfacht sich der Verbrauch auf 98 MB.

Damit wissen wir, dass es sich lohnt, bei der Eingabe der Testfälle aufmerksam zu überprüfen, ob man unnötige Bedingungen eingegeben hat, die redundant sind, da sich dies erheblich in der Laufzeit und im Speicherplatzverbrauch bemerkbar macht und unter Umständen sogar Fehlermeldungen wie „`Out of memory`" werfen kann.

Weitere Tests des Programms inklusive der Benutzungsschnittstelle sind unter www.ki.informatik.uni-frankfurt.de/master/programme/logicals zu finden.

# Kapitel 7

# Zusammenfassung und Fazit

Mit diesem Kapitel schließen wir diese Arbeit ab.
Wir werden zusammenfassend die Schritte und den Schwerpunkt der Arbeit erläutern, anschließend ein Fazit aus der Implementierung und den Tests ziehen und zuletzt einen Ausblick auf weitere Forschungen im Gebiet der SAT-Solver geben.

## 7.1 Zusammenfassung

Der Kernpunkt dieser Arbeit ist das Lösen von Logikrätseln unter Verwendung des SAT-Solvers und der Implementierung einer Benutzungsschnittstelle.
Daher wurde zunächst in Kapitel 1 die Motivation für diese Arbeit erläutert und anschließend die Grundlagen für das Verständnis dieser Arbeit in Kapitel 2 aufgeführt - dabei betrachteten wir Logikrätsel und führten das Ehepaarrätsel ein, das wir später kodieren und betrachten würden.
Dann stellten wir die Aussagenlogik vor, besprachen ihre Syntax und Semantik, betrachteten die Deduktionstheoreme und Beispielsätze. Anschließend stellten wir die nächste wichtige Logik unserer Arbeit vor: die Prädikatenlogik. Auch hier betrachteten wir Syntax, Semantik und anschauliche Beispielsätze.
Als abschließenden Abschnitt des Kapitels der Grundlagen betrachteten wir das Erfüllbarkeitsproblem und die dafür zugeschnittenen SAT-Solver.

Mit diesen Grundlagen war es uns möglich, in Kapitel 3 die Rätsellogik einzuführen, die eine auf endliche Mengen eingeschränkte und um Sorten erweiterte Prädikatenlogik darstellt, und auch hier betrachteten wir Syntax und Semantik, sowie anzahlbeschränkte Existenzquantoren, und definierten zusätzlich die Wohlgetyptheit der Formeln.
Anschließend widmeten wir uns dem theoretischen Kernpunkt dieser Arbeit: Den Transformationen von Rätsellogik in Aussagenlogik und den Rück-Transformationen des rätsellogischen Modells in das aussagenlogische Modell und zeigten eine solche Rückrechnung anhand eines Beispiels.

Damit hatten wir die Voraussetzung für den praktischen Teil der Arbeit geschaffen: die Implementierung der zuvor gezeigten Transformationen. Kapitel 4 zeigte im Detail, welche Module importiert wurden, welcher SAT-Solver verwendet wurde und wie das für die Arbeit in der funktionalen Programmiersprache Haskell erstellte Programm implementiert wurde. Dabei stand vor allem der Typcheck im Vordergrund, der dafür separat getestet wurde. Dann wurden die schrittweisen Transformationen aus Kapitel 3 umgesetzt.

In Kapitel 5 stellten wir dann die eingangs erwähnte intelligente Benutzungsschnittstelle vor, die dem Benutzer die Eingabe der Rätsel erleichtern soll. Sie wurde mit einem Parser realisiert, dessen detaillierte Implementierung und automatische Generierung anhand einer Parserspezifikation in diesem Kapitel dargelegt und anschließend getestet wurde.

Schließlich wurde die Implementierung ausgiebig getestet, um die Funktionsweise experimentell zu überprüfen. In Kapitel 6 wurde über diese berichtet. Zum einem wurde das Kernprogramm, welches die Rätsel löst, getestet, zum anderen die Benutzungsschnittstelle, mit der die Eingaben vereinfacht wurden.
Außerdem wurden die Laufzeit und der Speicherplatzverbrauch betrachtet und daraus Schlüsse gezogen. Die Tests verliefen erfolgreich und die Umsetzung des Rätsellösers mit Benutzungsschnittstelle ist somit geglückt.

## 7.2 Fazit

Durch den formalen Teil der Transformationen sowie die Implementierung und die Tests können wir einige Schlüsse ziehen. In Bezug auf den formalen Teil der Arbeit in Kapitel 3 fallen folgende Punkte auf:

- Das Erfüllbarkeitsproblem der Rätsellogik kann auf das Erfüllbarkeitsproblem der Aussagenlogik zurückgeführt werden. Da schnelle SAT-Solver für die Aussagenlogik existieren, scheint die zugehörige Transformation von Rätsel- in Aussagenlogik sinnvoll.
- Wir haben für die Transformationen von Rätsellogik in Klauselmenge Wohlgetyptheit definiert.

In Bezug auf die Implementierung und die Tests aus den Kapiteln 4 und 6 des Programms lassen sich folgende Schlüsse ziehen:

- Die Implementierung dieser Transformationen funktioniert korrekt.
- Der Speicherplatzverbrauch der Implementierung ist mit etwa 50 MB für größere Rätsel relativ gut.

- Die Laufzeit ist bei den größeren Rätseln relativ hoch und stellen wohl schon fast die Grenze des Erreichbaren dar.
- Die hohe Laufzeit lässt sich auf die „kombinatorische Explosion" zurückführen, die bei der Transformation von Rätsellogik zu Aussagenlogik entsteht, wobei alle Konstanten kombiniert werden, wenn die Quantoren aufgelöst werden.
  Ein großer Teil der Laufzeit wird auch durch die KNF-Berechnung selbst verschlungen, obwohl die schnelle KNF-Berechnung verwendet wird.
- Ein weiteres Resultat der Tests ist, dass die Formelgröße selbst als Maß für die zu erwartende Laufzeit nicht ausreicht, was anhand der zwei größeren Rätseln (Zebrarätsel und Ehepaarrätsel) zu sehen ist: Beide Rätsel sind etwa gleich groß, die erzeugte KNF ist jedoch im Falle des Zebrarätsels wesentlich größer.

Für die intelligente Benutzungsschnittstelle lässt sich festhalten:

- Die intelligente Benutzungsschnittstelle funktioniert korrekt und fängt die erwarteten Fehler mit Fehlermeldungen ab.
- Die Laufzeit sowie der Speicherplatzverbrauch steigt dabei etwas an.
- Die Schnittstelle vereinfacht dem Benutzer erheblich die Eingabe der Rätsel.
- Die Umsetzung der Schnittstelle hat sich bei der geringen Erhöhung der Laufzeit und des Speicherplatzverbrauchs verglichen mit der Praktikabilität gelohnt.

## 7.3 Ausblick

Nach dieser Forschung beim Einsatz von SAT-Solvern fallen auch einige Punkte auf, die noch ausbaufähig wären und im Rahmen einer weiteren Arbeit umgesetzt werden könnten.
Dazu zählen folgende Punkte:

- Die Benutzungsschnittstelle könnte um weiteren Komfort erweitert werden. Z.B. könnten Prädikate als links- und/oder rechts-eindeutig markiert werden. Die Eingabeverarbeitung könnte in diesem Fall automatisch Formel erzeugen, die diese Eindeutigkeiten garantieren.
- Die Fehlererkennung und -ausgabe kann noch weiter verbessert werden. Einerseits kann der Lexer und Parser derart erweitert werden, dass die Position des Fehlers in der Eingabe genauer ausgegeben wird, z.B. durch die Zeilennummer. Andererseits könnte man versuchen, Tippfehler zu erkennen und Hilfestellungen zu bieten. Man könnte auch überprüfen, ob statt einer Sorte ein Prädikatname verwendet wurde oder einen heuristischen „Abstand" von definierten Sorten und Prädikaten berechnen, beispielsweise:
  - Präfix der Länge M stimmt überein,

  - höchstens N falsche Buchstaben usw.

- Die Ergebnisse lassen sich eventuell dadurch verbessern, dass optimierende Transformationen auf der Rätselformel selbst durchgeführt werden, bevor die Aussagenlogische Formel berechnet wird.
  Zum Beispiel sollte eine Minimierung des Skopus der Quantoren dazu führen, dass die erzeugte aussagenlogische Formel kleiner wird.

Mit diesen und anderen Verbesserungen könnte man weiterführende Arbeiten, die SAT-Solver verwenden, optimieren und an dieser Arbeit anschließen.

# Literaturverzeichnis


[DLL62] Davis, Martin ; Logemann, George ; Loveland, Donald W.: A machine program for theorem-proving. In: *Commun. ACM* 5 (1962), Nr. 7, 394–397. http://dx.doi.org/10.1145/368273.368557. – DOI 10.1145/368273.368557

[DP60] Davis, Martin ; Putnam, Hilary: A Computing Procedure for Quantification Theory. In: *J. ACM* 7 (1960), Nr. 3, 201–215. http://dx.doi.org/10.1145/321033.321034. – DOI 10.1145/321033.321034

[Esc] Eschenbach, Dr. C.: *Prädikatenlogik, Universität Bremen*. http://www2.informatik.uni-hamburg.de/wsv/teaching/vorlesungen/FGI1SoSe07/PL.pdf. – Zugriff: 25.02.2015

[Kas] Kassel, Universität: *Aussagenlogik, FB 16 Universität Kassel*. http://www.uni-kassel.de/eecs/fileadmin/datas/fb16/Fachgebiete/FMV/TIL-WS1213/aussagenlogik-satsolver.pdf. – Zugriff: 11.02.2015

[Kie] Kiel, Universität: *Davis-Putnam-Logemann-Loveland-Algorithmus*. http://stueckwerk-logik.informatik.uni-kiel.de/dpll.html. – Zugriff: 26.01.2015

[KK06] Kreuzer, Martin ; Kühling, Stefan: *Logik in der Informatik*. Pearson Studium, 2006

[Lan12] Lange, M.: *SAT-Solver*. http://www.uni-kassel.de/eecs/fileadmin/datas/fb16/Fachgebiete/FMV/TIL-WS1213/aussagenlogik-satsolver.pdf. Version: 2012. – Zugriff: 25.01.2015

[Mar] Marlow, Simon: *Parsergenerator Happy*. https://www.haskell.org/happy/. – Zugriff: 15.05.2015

[Mül] Müller, Rene: *Zebrarätsel*. http://www.rc-wettingen.ch/download/Content_attachments/FileBaseDoc/8-DasZebraratsel.pdf. – Zugriff: 28.04.2015

[NE] Niklas Een, Niklas S.: *Sat4j*. http://www.sat4j.org/. – Zugriff: 28.03.2015

[NOT06] Nieuwenhuis, Robert ; Oliveras, Albert ; Tinelli, Cesare: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(). In: *J. ACM* 53 (2006), Nr. 6, S. 937–977

[Sch] Schnitger, Professor G.: *Theoretische Informatik 1.* http://www.thi.informatik.uni-frankfurt.de/lehre/gl1/ws1213/gl1_ws1213_skript.pdf. – Zugriff: 28.03.2015

[Sch95] Schöning, Uwe: *Logik für Informatiker (4. Aufl.).* Spektrum Akademischer Verlag, 1995 (Reihe Informatik). – ISBN 978–3–86025–684–8

[Seca] Seckinger, Bernhard: *Logelei aus der Zeit.* http://www.zeit.de/1968/22/logelei. – Zugriff: 25.02.2015

[Secb] Seckinger, Bernhard: *Logelei aus der Zeit.* http://www.zeit.de/2004/31/Spielen_2fLogelei__31. – Zugriff: 25.02.2015

[SS12] Schmidt-Schauss, Prof. Dr. M.: *Automatische Deduktion.* http://www.ki.informatik.uni-frankfurt.de/lehre/SS2012/AD/skript/AD.pdf, 2012

[SS13] Schmidt-Schauss, Prof. Dr. M.: *Skript zur Vorlesung "Grundlagen der Programmierung 2" (Sommersemester 2014).* htt p://www.informatik.uni-frankfurt.de/∼prg2/SS2014/index.html, 2013

[ST12] Schöning, Uwe ; Torán, Jacobo: *Das Erfüllbarkeitsproblem SAT.* lehmanns media, 2012

[Waga] Wagner, Karl H.: *Aussagenlogik FB 10 Universität Bremen.* http://www.fb10.uni-bremen.de/khwagner/grundkurs2/kapitel3.aspx. – Zugriff: 22.02.2015

[Wagb] Wagner, Karl H.: *Prädikatenlogik FB 10 Universität Bremen.* http://www.fb10.uni-bremen.de/khwagner/grundkurs2/kapitel4.aspx. – Zugriff: 22.02.2015

[Wika] Wikipedia: *Digitale Revolution.* http://de.wikipedia.org/wiki/Digitale_Revolution. – Zugriff: 06.03.2015

[Wikb] Wikipedia: *DPLL-Algorithmus.* http://en.wikipedia.org/wiki/DPLL_algorithm. – Zugriff: 26.01.2015

[Wikc] Wikipedia: *Kreuzzahlenrätsel.* http://de.wikipedia.org/wiki/Kakuro. – Zugriff: 25.02.2015

[Wikd] Wikipedia: *Logicals.* http://de.wikipedia.org/wiki/Logical. – Zugriff: 25.02.2015