Logische Programmierung

Anwendungsbeispiel Prolog

Informatik Seminararbeit

Studiengang:	Informatik
Autoren:	Mira Günzburger
Betreuer:	Pierre Fierz; Peter Schwab
Datum:	18. September 2014

Versionen

Version	Datum	Status	Bemerkungen
0.1	05.03.2014	Entwurf	Konzepterstellung
0.2	10.05.2014	Entwurf	Entwurf
0.3	16.05.2014	Entwurf	Erweiterung
1.0	21.05.2014	Definitiv	Korrekturen angebracht; Reinschrift
1.1	23.05.2014	Definitiv	Fazit
1.2	25.05.2014	Definitiv	Layout angepasst; Fertigstellung

Inhaltsverzeichnis

1. Informatik Seminar Arbeit FS 2014

Bei diesem Dokument handelt es sich um das Abschlussdokument der Informatik Seminararbeit. Das entsprechende Modul wird im folgenden Unterkapitel genauer beschrieben.

1.1. Modulbeschreibung

1.1.1. Lernziele

Ziel dieses Moduls ist es einerseits Fachkenntnisse in einem oder mehreren Bereichen der Informatik zu erwerben, andererseits aber auch die Kompetenz zu erwerben, ein Thema für andere Zuhörer oder Leser verständlich aufzubereiten. Das schliesst die Kompetenz ein, dieses Thema sowohl mündlich in einer kurzen Präsentation als auch schriftlich in einem Bericht kompakt und verständlich aufzubereiten. Die Dozierenden schlagen dazu eine Reihe von möglichst attraktiven bzw. innovativen aktuellen Themen vor. Die Zuteilung der Themen wird durch die Dozierenden vorgenommen, wobei so weit wie möglich auf Interesse und Vorliebe der Studierenden eingegangen wird. Die zur Auswahl stehenden Themen sind unabhängig vom Vertiefungsgebiet des/der Studierenden und können auch ausserhalb der Kernthematik des Vertiefungsgebietes liegen. Je nach Organisation des Seminars wird entweder alleine ein Thema bearbeitet oder es wird in einer Zweiergruppe zwei Themen präsentiert. Für die Beurteilung wird sowohl die mündliche Präsentation als auch der Bericht herangezogen.[2]

1.1.2. Erworbene Kompetenzen

Am Ende dieses Moduls sollten die Studierenden in der Lage sein: (Wissen und Verstehen) Aufzählen von z.B. den Charakteristika, Funktionalitäten, innovativen Aspekten, Bewertung der Vor- und Nachteile, etc. der verschiedenen Teile des behandelten Themas. (Urteilen und Probleme bearbeiten) Eine Bibliographie-Recherche durchführen. Zusammenfassen von Informationen. Abfassen eines klaren, kurzen, prägnanten Berichtes, welcher die grundlegenden Aspekte des behandelten Themas enthält. Mündliche Präsentation der Ideen und der wesentlichen Resultate. Interesse an der Arbeiten ihrer Kollegen aufbringen und konstruktive Kritik an deren Arbeit äussern.[2]

2. Einleitung

Bei der logischen Programmierung handelt es sich um eine spezielle Form des deklarativen Programmierparadigma. Bei herkömmlichen, prozeduralen Programmierparadigmen steht die Lösung der Aufgabe im Vordergrund. Im Gegensatz dazu, steht in der deklarativen oder konkret der Logischen Programmierung die Beschreibung des Problems im Vordergrund. So besteht ein Programm nicht aus einer Folgen von Anweisungen sondern aus einer Menge von Axiomen welche Fakten und Regeln beschreiben. Bei einer Anfrage werden grundsätzlich die Regeln durch den Interpreter gegen die Fakten geprüft und somit die Anfrage ausgewertet.
Die logische Programmierung ähnelt häufig einer Spezifikation. Sie ist in vielen Fällen kürzer, kompakter und verständlicher als imperative Programmierung. Das Wissen wird deklarativ das heisst explizit und maschinenunabhängig dargestellt. Dadurch wird Wissen vielseitig verwendbar. Eine weitere Besonderheit ist, dass Beweise dank einfacher mathematischer Basis häufig leichter durchführbar sind. Logische Programme sind aber dennoch nicht für alle Probleme geeignet. In diesen Fällen kann mit logischer Programmierung Metawissen für andere Programme darstellen werden. [3]
In diesem Dokument werden die Konzepte und Anwendungsfälle der Logischen Programmierung am Beispiel der Programmiersprache Prolog beschrieben und illustriert. Prolog kann als die wichtigste logische Programmiersprache bezeichnet werden.[4]

2.1. Geschichte/ Herkunft

Der Ursprung der logischen Programmierung liegt im automatisierten Beweisen mathematischer Sätze. Wie in 4.2 (Prolog Anwendung) erläutert wird, basiert die Logische Programmierung auf der Prädikatenlogik erster Stufe. Der Mathematiker und Philosph Gottlob Frege legte mit seiner Begriffschrift 1878 die Grundlage dafür. So arbeitet er mit den Operationen Implikation, Negation und Allquantoren. Später wurde die Prädikatenlogik durch Guiseppe Peano und Bertrand Russel weiterentwickelt.
In den 30er Jahren beschäftigten sich Kurt Gödel und Jacques Herbrand mit der Idee von auf Ableitungen basierenden Berechnungen (Ursprünge von computation as deduction). J.A. Robinson veröffentlicht 1965 ein Werk, welches die Grundlage für automatisierte Ableitung darstellt. In diesem hat er auch die Resolution eingeführt.

Im seiner Arbeit „Predicate Logic as a programming Language“(1974) beschreibt Robert Kowalski wie man in dem oben genannten System Berechnungen vornehmen kann. Ebenfalls in den 1970 Jahren arbeitet der Franzose Alain Colmerauer an einer Programmiersprache zum Beweisen von Theoremen. In einer Zusammenarbeit der beiden Wissenschaftlern entsteht 1973 die Programmiersprache Prolog. Prolog (vom Französischen: Programmation en Logique) kann als wichtigste logische Programmiersprache bezeichnet werden.

Ihren Ursprung hat die Programmiersprache in der natürlichen Sprachverarbeitung. Erst später erkannte man, dass sie auch als eigenständige Programmiersprache benutzt werden kann. Die logische Programmierung und ihr Konzept beeinflusst viele Teile der Informatik und ihre Entwicklung. [5],[6]

2.1.1. Einsatzgebiete

Die logische Programmierung wird in vielen unterschiedlichen Gebieten eingesetzt.
In der Informatik wird Sie zum Beispiel in folgenden Bereichen verwendet[4]:

- Künstliche Intelligenz

- Datenbank
- Expertensysteme

Aber auch außerhalb der Informatik hat sie einen wichtigen Stellenwert:

- Medizin (Diagnosesysteme zb MYCIN)
- Mathematik (Theorembeweise und Theoremgeneratoren)
- Naturwissenschaften

3. Theoretische Grundlagen

In der Logischen Programmierung müssen Fakten und Regeln abgebildet werden. Dies wird in Form der Syntax der Prädikatenlogik erster Stufe umgesetzt.
Zusätzlich macht sie sich verschiedene Algorithmen zu nutzen. Unter anderem wird Unifaktion, Resolution und Backtracking für eine ideale Verarbeitung verwendet. Bevor also konkret auf die Programmiersprache Prolog eingegangen wird, sollen zum besseren Verständnis diese Grundlagen in diesem Kapitel kurz erklärt werden.

3.1. Aussagenlogik

Die Aussagenlogik hat die Möglichkeit mittels Variablen elementare Aussagen darzustellen. Aus elementaren oder atomaren Aussagen A und B, kann eine zusammengesetzte Aussage gebildet werden. $A \wedge B$. Es gibt verschiedene Arten zwei Aussagen zu kombinieren. Die sogenanten Junktoren werden zu einem späteren Zeitpunkt verwendet, deshalb findet sich in der Tabelle 3.1 eine Übersicht.[6]

Aussagenlogische Junktoren

Tabelle 3.1.: Aussagenlogische Junktoren

Bezeichnung	Erklärung	Zeichen	Bedeutung
Disjunktion	oder Verknüpfung	$A \vee B$	A oder B
Konjuktion	und Verknüpfung	$A \wedge B$	A und B
Negation	verneinende Aussage	$\neg A$	nicht A
Subjunktion	hinreichende Bedingung	$A \leftarrow B$	aus A folgt B
Implikation	notwendige Bedingung	$A \rightarrow B$	für A muss mindestens B erfüllt sein
Äquivalenz	hinreichende und notwendige Bedingung	$A \leftrightarrow B$	A ist genau dann wahr, wenn B wahr ist

3.2. Prädikatenlogik

Bei der Prädikatenlogik erster Stufe handelt es sich um ein Teilgebiet der mathematischen Logik. Sie bildet eine Familie logischer Systeme. Genauer ist die Prädikatenlogik eine Erweiterung der Aussagenlogik.
Sie spielt eine grosse Rolle in vielen Bereichen der Logik, Mathematik, Informatik, Linguistik und Philosophie.

Quantoren

In Prolog und in der Prädikatenlogik müssen die folgenden Quantoren bekannt sein[7]:

- Allquantor: ∀
  Für alle; jedes x gilt.

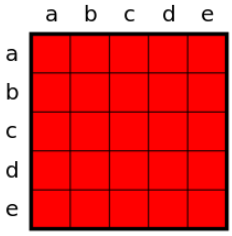

Abbildung 3.1.: Allquantor: jeder kennt jeden [7]

- Existenzquantor: ∃
  es existiert mindestens ein x.

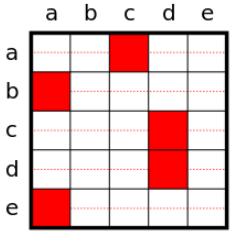

Abbildung 3.2.: Existenzquantor: jeder kennt jemanden [7]

3.3. Klauseln

Ein Literal ist ein Atom, oder eine Negation eines Atoms.[8]

Eine Klausel wird auch Disjunktionsterm genannt. Dabei werden je zwei positive oder negative Literalen durch eine Disjunktion verbunden:
(A ∨ B)

Die Konjunktive Normalform hat die Form:
(A ∨ B) ∧ ...∧ (¬ C ∨ D)
Es werden also jeweils zwei Klauseln durch eine Konjunktion verbunden.[6]

3.3.1. Hornklauseln

Die Hornklausel ist eine spezielle Form der Disjunktionsterme. Im Gegensatz zur Klausel darf aber in der Hornklausel nur ein einziges positives Literal vorkommen. [8]

Ein Beispiel: A ∨ ¬ B ∨¬ C ∨¬ **D**

Die Tabelle 3.2 zeigt mit Hilfe einer Wahrheitstafel, dass die Hornklausel in eine Implikation umgewandelt werden kann.

Tabelle 3.2.: Hornklausel zu Implikation

B	**A**	**¬B**	**¬B ∨A**	**B → A**
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	0	1	1

Somit kann die obige Hornklausel auch folgendermaßen dargestellt werden:
A ← B ∨ C ∨ D
Diese Form wird zu einem späteren Zeitpunkt noch verwendet.

Es gibt drei Arten von Hornklauseln[6] welche in Tabelle 3.3 aufgeführt sind.

Tabelle 3.3.: Die drei Typen von Hornklauseln

Name	**Beschreibung**	**Klauselschreibweise**	**logische Form**
Faktenklausel (Faktum)	kein negatives, ein postivives Literal	(A)	{A}
Definite Hornklausel (Regel)	n negative Literale, ein positives Literal	{¬ A, ¬B,¬C , K }	(A ∧ B ∧ C → K)
Zielklausel (Anfrage)	Kein positives Literal	{¬ A, ¬B,¬C}	(A ∧ B ∧ C)

3.4. Substitiution

Kurz gesagt handelt es sich bei der Substitution um das Ersetzten eines Ausdrucks durch einen anderen. Es wird zwischen der universellen und der einfachen Substitution unterschieden. Bei Ersteren müssen alle vorkommenden Ausdrücke ersetzt werden. Bei Zweiteren kann eine beliebige Anzahl Ersetzungen vorgenommen werden.[9]

3.5. Unification

In der Unifikation werden Ausdrücke in zwei Terme so substituiert, dass sie gleich werden. Im Algorithmus wird die Substitution gesucht, welche am allgemeinsten ist. Dabei wird vom allgemeinsten Unifier gesprochen. [1]

3.6. Interpretation und Model

Bei der Interpretation handelt es sich um die Zuordnung von der realen Welt zu aussagenlogischen Variablen. So gilt die Definition:

Sei M die Menge aller aussagenlogischen Formeln. Dann heisst eine Funktion I Interpretation:
I:M → {T,F}

Es handelt sich also um den Wahrheitswert des Ausdruck in Abhängigkeit der Atome.
An einem Beispiel[6] in der Tabelle 3.4 wird eine Interpretation konkret aufgezeigt.

Tabelle 3.4.: Interpretation

	I der Aussage A	**I der Aussage B**	**A**$\vee$ B
Welt 1	$5 > 4$	$1 = 2$	false
Welt 2	$2 = 2$	$7 > 4$	true

In diesem Zusammenhang ist wichtig zu wissen, dass eine Formel als erfüllbar bezeichnet wird, falls eine Interpretation existiert, so dass die Formel in der Wahrheitstafel den Wert true ergibt.

Eine Interpretation einer Menge wird als Model bezeichnet, wenn sämtliche Formeln unter der Interpretation wahr sind.[6]

3.7. Semantische Folgerung und Ableitbarkeit

Man sagt X folgt semantisch aus Y, wenn jedes Model aus X auch Model von Y ist. Somit ist X eine semantische Folgerung von Y. Dies wird folgendermassen geschrieben: $Y \models X$.

Existiert eine Folge von Inferenzschritten, so dass man von X zu Y gelangt, sagt man Y ist ableitbar aus X.
$X \vdash Y$.

Ein Beweis-Verfahren wird als korrekt bezeichnet, wenn für beliebige Formeln X und Y gilt: falls Y von X ableitbar ist, dann ist Y auch eine semantische Folgerung von X. Trifft der umgekehrte Fall zu, wird ein Verfahren als vollständig bezeichnet.[6]

3.8. Verarbeiten in regelbasierten Systemen

Um eine Anfrage zu prüfen gibt es in regelbasierten Systemen unterschiedliche Arten der Verarbeitung. Die Vorwärtsverkettung (Forward Chain) oder die Rückwärtsverkettung (Backward Chain).[8]

3.8.1. Vorwärtsverkettung

Bei der Vorwärtsverkettung wird von den vorhandenen Fakten ausgegangen. Wenn die Bedingung einer geeigneten Regel erfüllt ist wird durch Anwendung der Regel eine Aktion ausgeführt, so dass sich der Zustand modifiziert. Dieser Vorgang wird solang wiederholt, bis der gewünschte Zielzustand erreicht wird und die Anfrage beantwortet werden kann. [6] Dies ist in der Abbildung 3.3 dargestellt.

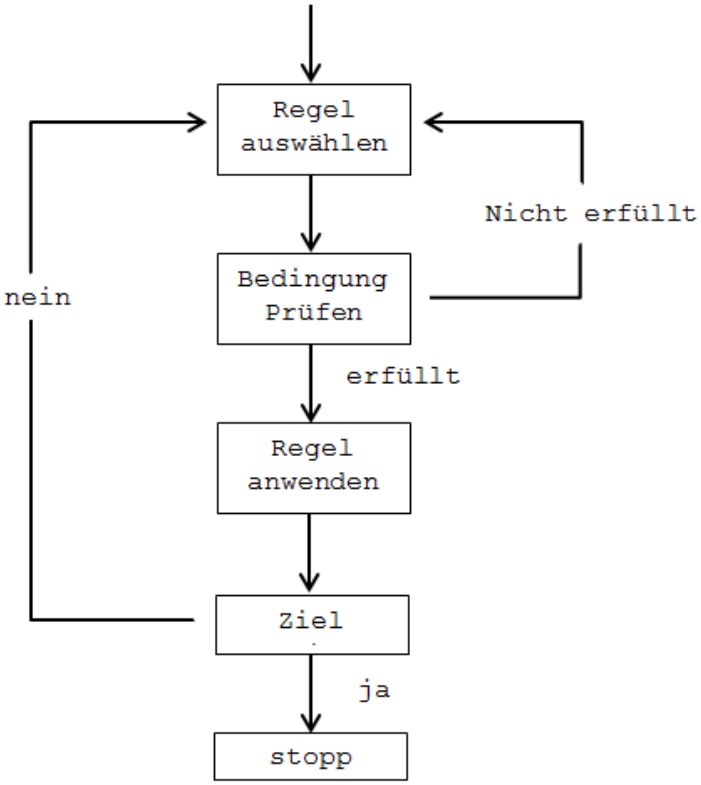

Abbildung 3.3.: Ablauf Forward Chain

3.8.2. Rückwärtsverkettung

In Prolog wird die zielorientiertere Rückwärtsverkettung angewendet. Dabei wird von der Zielanfrage ausgegangen und eine entsprechende Regel gesucht. Danach werden die Bedingungen sukzessive überprüft. Dieser Vorgang ist in der Abbildung 3.4 grob dargestellt und wird im Kapitel 4.3.2 genauer beschrieben.[6]

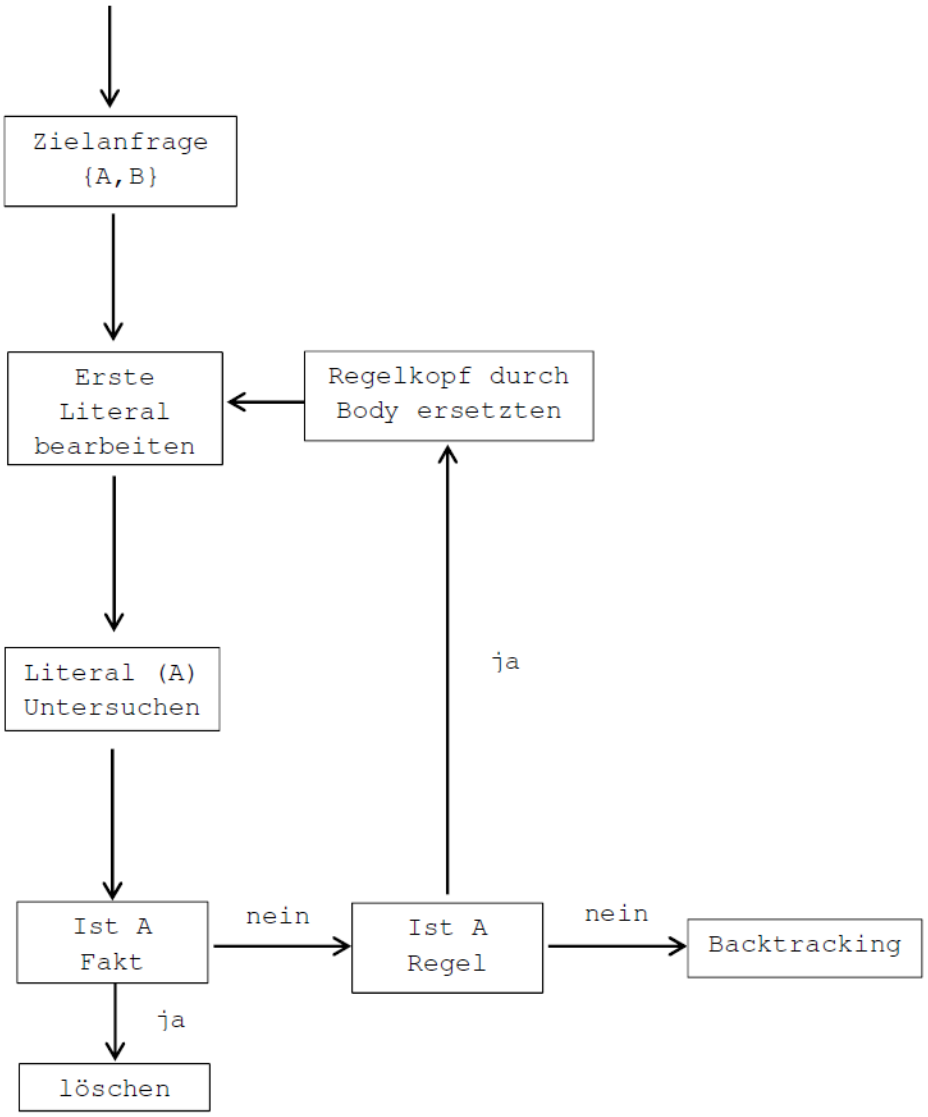

Abbildung 3.4.: Ablauf Backward Chain

3.9. Resolution

Bei der Resolution handelt es sich um ein Beweisverfahren in der Logik, mit welcher getestet werden kann, ob ein Ausdruck gültig ist. In dem Verfahren wird mit Hilfe von mehren Resolutionsschritten versucht, eine Zielanfrage zu belegen. In der Resolution werden die Schritte von links nach rechts abgearbeitet. Um zurück zur Wurzel zu gelangen wird Backtracking verwendet. Es handelt sich um eine Tiefensuche. Wie jede andere Suche kann die Resolution in einem Suchbaum (Resolution Tree) dargestellt werden. Beim Suchbaum handelt es sich um ein And Or Tree. Er besteht aus Und-Knoten und Oder-Knoten. Grundsätzlich hat die Resolution zwei wichtige Eigenschaften:

- Als Grundlage dient die Normalform von aussagenlogischen Formeln.
- Resolution basiert auf dem Widerspruchsprinzip, und führt einen Widerspruchsbeweis durch: Das Resolutionsverfahren leitet also eine logischen Wiederspruch aus der Verneinung ab.

Die Resolution ist nicht Vollständig aber Wiederlegungsvollständig. Wenn die Klauselmenge also widersprüchlich ist, erreicht man mit einer endlichen Folge von Resolutionsschritten den Wiederspruch.

In der Resolution wird häufig die Schreibweise des Modus ponens verwendet:

$A \rightarrow B$
A
———
B

Beim Modus ponens handelt es sich um eine Inferenzregel. Der oben gezeigt Ausdruck bedeutet: Aus $A \rightarrow B$ und A folgt B.

Inferenzregeln stellen eine logische oder relationale Beziehung zwischen S1 und S2 dar. Mit Hilfe der Regeln werden aus gegebenen Sätzen S1 neue Sätze S2 abgeleitet.[6]

3.9.1. Resoultionsregel

Die Resolution verwendet genau eine Inferenzregel - die Resolutionsregel. Aus diesem Grund eignet sie sich besonders gut für die automatische Beweisführung.[6]

Resolutionsregel:

Es existieren die Klauseln:

$L \in C1$,	%L ist Teil von C1
$\neg L \in C2$	% ¬ L ist Teil von C2
wie bekannt ist gilt:	
L	
¬ L	
———	
$\perp$	%Aus L und nicht L folgt die leere Menge

Dann nennt sich die Verbindung von C1 und C2 Resolvente und entspricht:

$(C1 \setminus L) \cup (C2 \setminus \{\neg L\})$

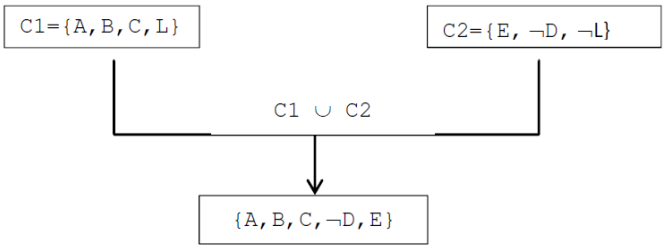

Abbildung 3.5.: Resolutionskalkül

3.9.2. SLD Resolution

In Prolog wird eine spezielle Form von Resolution verwendet, die SLD Resolution. SLD steht für Linear Resolution with Selection Function for Definite Clause. Wie der Name vermuten lässt geht die SLD Resolution linear vor. Es wird also jeweils die neu erzeugte Resolvente als Resolution benutzt. Im nächsten Resolutionsschritt wird sie zu einer „Elternklausel“. In jedem Resolutionsschritt wird das erste noch zu beweisende Literal bearbeitet. Es besteht also im Gegensatz zur klassischen Resolutions die Möglichkeit auf die Abarbeitung der Beweisschritte Einfluss

zu nehmen, indem die Literalen und Klauseln in einer adäquaten Reihenfolge aufgelistet werden. Die Anzahl der möglichen Resolutionsschritten ist in der SLD Resolution deutlich eingeschränkt. Es wird auch nur eine Teilmenge der Prädikatenlogik betrachtet. [6]

3.10. Backtracking

Beim Backtracking wird das Rücksetzverfahren genutzt. Es wird nach dem Versuch-und-Irrtum Prinzip gearbeitet. So wird also versucht, eine erreichte Teillösung zu einer Gesamtlösung auszubauen. Sobald erkannt wird, dass eine Teillösung zu keiner nützlichen Lösung führt, werden die letzten Schritte zurück genommen um einen alternativen Weg auszuprobieren. Wenn für eine Anfrage eine Lösung existiert wird diese immer gefunden. Anderenfalls kann definitiv erkannt werden, dass keine Lösung existiert [10]

4. Prolog

Nachdem wir im Kapitel theoretische Grundlagen einen kurzen Überblick über die wichtigsten Theorien erhalten haben, soll nun die konkrete Anwendung am Beispiel von Prolog veranschaulicht werden. Zum besseren Verständnis wird das allgemein bekannte Beispiel eines Stammbaum als Grundlage genommen. Dies ist eine immer wiederkehrende Vorlage, welche sich aber nichtsdestotrotz ausgezeichnet eignet, um das Prinzip von Prolog zu veranschaulichen. Der Beispielstammbaum für die hier verwendeten Erläuterungen findet sich im Anhang A.1.

4.1. Semantik und Syntax

Bei dieser Seminararbeit geht es in erster Linie um das Verständnis der Logischen Programmierung am Beispiel von Prolog. Es ist also notwendig die Semantik und Syntax von Prolog zu kennen. Sie werden hier aber nur oberflächlich beschrieben, da sie nicht als zentraler Punkt sonder nur als Mittel zum Zweck angesehen werden.

Die grundsätzliche Datenstruktur in Prolog wird als Term oder Token bezeichnet. Es wird zwischen einfachen und zusammengesetzten Token unterschieden.[5]

Einfache Token:

- Konstanten oder Atome
  beginnen mit einem Kleinbuchstaben oder sind mit Apostrophen eingeschossen
  Beispiel: 'Martin', eric, parent
- Variablen
  Beginnen mit einem Großbuchstaben, oder als Sonderfall die anonymen Variablen welche durch einen Unterstrich gekennzeichnet sind. Anonyme Variablen werden nie konkret besetzt.
  Beispiel: X,Y,_
- Zahlen
  Ganze oder reelle Zahlen

Zusammengesetzt Token:

- Strukturen (Prädikatenlogische Aussage)
  - Fakten
    Beschreiben eine Eigenschaft oder eine Beziehung. Sie können eins bis n stellig sein. Fakten werden mit einem Punkt abgeschlossen.
    Beispiel: parent(max, luca).
  - Regeln
    Regeln haben eine Regelkopf und einen Regelkörper, diese werden durch :- getrennt und mit einem Punkt abgeschlossen.
    Beispiel son(X,Y) :- father(X,Y), male(Y).
- Listen
  in eckigen Klammern aufgelistete Konstanten oder Variablen
  Beispiel: [max, luca, elias]

4.1.1. Arithmetik und Vergleichsoperatoren

In Prolog werden die gängigen arithmetischen Operatoren wie zum Beispiel +, - , *, /, mod unterstützt.

Die Operatoren = und /= werden in Prolog für die Unfikation 4.3.1 verwendet. Mit == wird in Prolog die Identität bezeichnet.
Die Auswertung eines arithmetischen Ausdrucks wird mit dem is-Prädikat explizit veranlasst.[6]

- Unifikation:
  ?- X = 4. (ist X mit 4 unifizierbar)
- Identität:
  ?- X == 4. (ist die Variable X und die Konstante 4 identisch)
- Is-Operator:
  X is 4 + 1. (X = 4 + 1; die Antwort ist also 5)

4.2. Prolog Anwendung

Die erklärten Theorien werden so kombiniert, dass Aufgrund vorangegangener Deklarationen, Anfragen ausgewertet und beantwortet werden können. Grundsätzlich werden Anfragen mit true oder false beantwortet, oder eine oder mehrere Variablen werden besetzt. Wird eine Antwort mit false beantwortet, heisst das nur, dass die Anfrage mit den vorhandenen Deklarationen nicht beantwortet werden kann. Dieses Verfahren nennt sich „closed world assumption“. Die Hauptaufgabe des Programmierers besteht darin, die reale Umwelt in Fakten und Regeln abzubilden. Damit können die Anfragen vom System beantwortet werden.

4.2.1. Fakten

Die für die Logische Programmierung notwendigen Fakten werden in Prolog durch prädikatenlogische Aussagen formuliert. Aussagen, welche ohne Vorbedingung wahr sind, werden als Fakten bezeichnet. Sie entsprechen Hornklauseln ohne Regelkörper. Diese Form entspricht der oben vorgestellten Faktumklausel. Der Regelkörper ist somit in jedem Fall wahr. Eigenschaften werden durch einstellige, Beziehungen durch mehrstellige Aussagen deklariert. Alle Fakten werden für konkrete Objekte definiert.[1]

Beispiel für Eigenschaften:

- male(max)
  Max ist männlich
- male(luca)
  Luca ist männlich
- female(lisa)
  Lisa ist weiblich
- female(tea)
  Tea ist weiblich

Beispiel für Beziehungen:

- parent(max,luca)
  Max ist ein Elternteil von Luca
- parent(max,tea)
  Max ist ein Elternteil von Tea

4.2.2. Regeln

Regeln werden in Hornklauseln formuliert. Sie bestehen aus einem Regelkopf und einem Regelkörper. Sie werden im Gegensatz zu Fakten für Variablen definiert welche bei der Verarbeitung einer Anfrage durch konkrete Objekte ersetzt werden.

```
mother(M,P):-                    % Regelkopf; Folgerung
    female(M), parent(M,P).      % Regelkörper; Voraussetzung
```

Die aufgeführte Regel von Prolog entspricht folgender prädikatenlogischen Aussage:
$\forall$ M,P(parent(M,P)$\vee$ female(M) $\rightarrow$ mother(M,P))

Wie im obigen Beispiel ersichtlich ist, kann eine Regel aus mehrere Bedingungen bestehen. Diese werden hintereinander durch Koma getrennt aufgelistet. Das Koma , entspricht einer „Und Verbindung“. Ein Strichpunkt ; signalisiert eine „Oder Verbindung“.
Eine Variable kann in einer Regel (oder auch in einer Anfrage) mehrmals verwendet werden. In diesem Fall spricht man von einer „shared Variable“. Als weiteres ist zu berücksichtigen, dass es auch notwendig sein kann, Regeln rekursiv aufzurufen.
Dies kann mit dem Beispiel der Vorfahren gut veranschaulicht werden. Um einen Vorfahren zu definieren haben wir zwei Bedingungen. Bei dem Vorfahren kann sich um einen Elternteil A von E handeln. Andererseits kann der Vorfahre A Elternteil von X sein, wobei X wiederum ein Vorfahre von E ist.

```
ancestor(A,E):-
    parent(A,E).
ancestor(A,E):-                  % Zweite Regeln für ancestor
    parent(A,X),ancestor(X,E).   % X wird hier zweimal verwendet -> shared Variable.
                                 % ancestor wird rekursiv aufgerufen
```

Gerade bei rekursiv aufgerufenen Regeln ist die Reihenfolge sehr wichtig. In Prolog werden Regeln und Fakten immer von oben nach unten und von links nach rechts abgearbeitet. Es liegt in der Verantwortung des Programmierers darauf zu achten, dass kein endlos Loop entsteht oder die Regeln in der richtigen Ordnung aufgerufen werden. [1]

4.2.3. Zielanfrage

In Prolog gibt es zwei unterschiedliche Arten von Zielanfragen. Die einfachen Anfragen, ohne Variablen, diese werden nur mit true oder false beantwortet. Sowie Anfragen mit Variablen, wenn möglich wird die Variable gesetzt. Andernfalls wird false zurück gegeben.

```
Einfache Anfrage
    ?-parent(max,luca).          % Ist Max Elternteil von Luca?
    yes                          % Max ist Elternteil von Luca
    ?-parent(tea,luca).          % Ist Tea Elternteil von Luca?
    no                           % Die Anfrage kann aufgrund der vorhandenen Fakten und Regeln
                                 % nicht beantwortet werden.

Anfrage mit Variable
    ?-parent(max,X).             % Von wem ist Max Elternteil?
    X = Luca                     % erste gefundene Übereinstimmung
    ;                            % suche nach weiteren Einträgen
    X = Tea                      % zweiter Eintrag gefunden
    ;                            % suche nach weiteren Einträgen
    no                           % keine weiteren Einträge gefunden
```

[1]

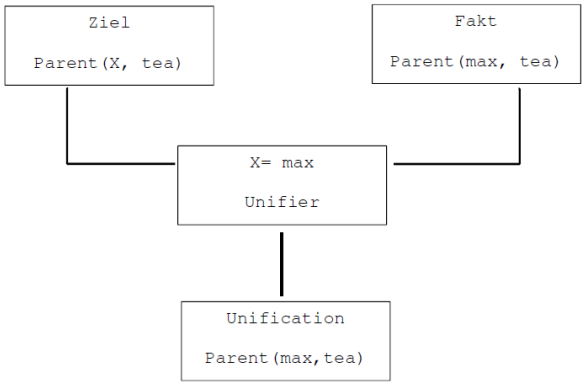

Abbildung 4.1.: Unifikation

4.3. Anfragen Auswerten

4.3.1. Unifikation

In Prolog wird versucht mit Hilfe von Unifikation Anfragen mit Regeln oder Fakten identisch zu machen. Im Beispiel der Zielanfragen sehen wir einen direkten Einsatz von Unifikation. Handelt es sich um eine Anfrage mit Variable, wird die Variable in der Antwort durch Unifikation belegt.

Im Beispiel in Abbildung 4.1 soll die Unifikation durch eine Skizze veranschaulicht werden. Es wird die Variable X durch die Konstante max belegt. Dadurch wird die Zielanfrage gleich wie der Fakt. Es handelt sich also bei max = X um Unifier. Prolog kann die Anfrage also beantworten und wird X=max zurück geben.
Der Prolog Interpreter strebt immer den allgemeinsten Unifier an. [1]

4.3.2. Resolution

In Prolog wird von einer Zielliste Z ausgegangen, welche die konkreten Zielanfragen enthält. Die Anfragen in der Liste werden von Links nach Rechts durchgearbeitet. Es wird die Regel oder der Fakt (R) gesucht, bei dem ein allgemeiner Unifier existiert, so das der Regelkopf der Zielanfrage entspricht. Der erste Literal der Zielanfrage wird durch den Regelbody ersetzt. Wird eine passender Fakt gefunden, kann die Zielanfrage aufgelöst werden und wird gelöscht. Dies geschieht anhand der im vorigen Kapitel beschriebenen Resolutionsregel. Hier die konkrete Umsetzung in Prolog.

Es existieren folgende Regeln:

```
A                    % Fakt
B:- A                % Regel
?- B                 % Zielanfrage
```

Als erstes wird die Prologschreibweise in eine Hornklausel umgewandelt:

(1) A % Fakt
(2) $B \vee \neg A$

und die Negation der Anfrage wird mit dazu genommen:

(3) $\neg B$

Nun wird die Resolution angewendet:

(4):(2) + (3):($B \vee \neg A$) + $\neg B = \neg A$
(5):(1) + (4): $\neg A + A = \{\}$

[11]

Dieses vorgehen wird rekursiv weiter geführt bis die Zielliste leer ist. Wird in der Teillösung kein passender Unifier gefunden, gelingt der Beweis also nicht, wird mittels Backtracking nach einer weiteren anwendbaren Regel gesucht. Sämtliche, durch die Anwendung der Regeln entstandenen Variablenbindungen (Unifiers) werden wieder gelöscht. Prolog verwendet die erste passende Klausel, weitere in Frage kommende Fakten oder Regeln werden erst geprüft, wenn die erste mittels Backtracking verworfen wurde.

4.3.3. Veranschaulichung am Stammbaumbeispiel

Um die Verarbeitung einer Anfrage zu veranschaulichen wird das Stammbaum Beispiel um einige Fakten und Regeln erweitert.

parent(Madita,Lisa).
parent(Hans,Lisa).

Zur Übersicht findet sich der Beispiel Familienstammbaum im Anhang A.1.
Anhand einer einfach Zielanfrage soll nun der Resolution Algorithmus veranschaulicht werden.
?-ancestor(hans,tea)

Mit dem Suchbaum in Abbildung 4.2 soll das Vorgehen veranschaulicht werden. Der Suchbaum zeigt die Auswertung der Anfragen an einem einfachen Beispiel. Dabei wird Unifikation, die Resolution und auch das Backtracking verwendet. Es handelt sich um einen And-Or-Tiefenbaum. Die grünen Kästchen symbolisieren die And-Knoten mit ihren Nachfolgerknoten. [1]

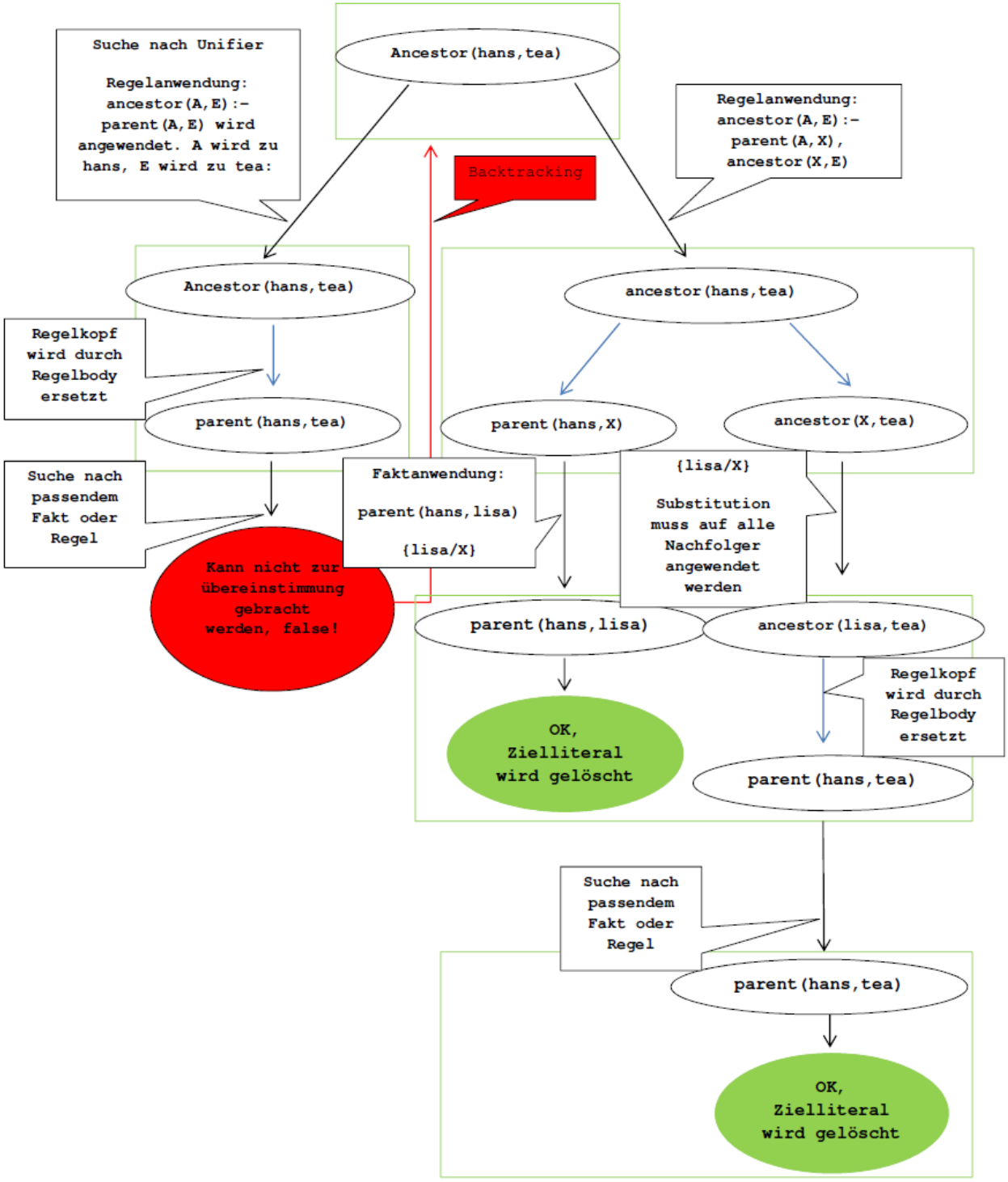

Abbildung 4.2.: Resolution Tree

4.3.4. Auswertung kontrollieren

Es wurde im obigen Kapitel 4.3.2 erklärt, nach welcher Reihenfolge Anfragen abgearbeitet und Fakten und Regeln geprüft werden. Beim Aufstellen der Wissensbasis ist es also wichtig, dies zu berücksichtigen. Damit müssen Fehler, wie zum Beispiel endlos Loops vermieden werden, oder die Performens kann verbessert werden.

Regeln und Literale

Ein Problem das häufig auftritt ist eine Fehlermeldung wegen Speichermangels. Dies geschieht, wenn zwei gleichnamige Regeln in der falschen Reihenfolge niedergeschrieben sind. Dies kann zum Beispiel in unserer ancestor Regel der Fall sein.

```
ancestor(A,E):-
    parent(A,E).                    %
ancestor(A,E):-                     %
    parent(A,X),ancestor(X,E).      % ancestor(X,E) wird hier rekursiv aufgerufen.
                                    % Wäre diese Regel an erster Stelle, würde, dies zu einem endlosen
                                    % Rekursionszyklus führen, da das Programm niemals
                                    % zur abschliessenden Regel parent(A,E). gelangen wird.
```

Die gleiche Situation gilt auch bei der Anordnung der Klausel in einer Regel. Würden wir also in der zweiten ancestor Regel zuerst ancestor(X,E) und erst dann parent(A,X) aufrufen, würde das Programm durch den rekursiven Aufruf von ancestor auch auf einen Fehler treffen.[6]

Backtracking

In gewissen Situationen kann eine Regel durch eine andere ausgeschlossen werden. In diesem Fall macht es keinen Sinn die nächsten Regeln noch zu prüfen. Das Backtracking wird also unterdrückt, beziehungsweise kontrolliert. Dies wird mit dem Cut Operator „!“ erreicht. Der Cut wird also als Hilfsmittel für die Wissenspräsentation genutzt.[6]

Anfrage mit Variable

(a) parent(C,P):- mother(C,P),!.
(b) parent(C,P):- father(C,P).

Trifft im obigen Beispiel Regel a zu, ist P also Mutter von C, muss Regel b nicht mehr geprüft werden, da es nicht möglich ist, dass P gleichzeitig Vater und Mutter von C ist.[1]

Für die Kontrolle übers Backtracking gibt es zwei Arten von Cut. Beim oben vorgestellten handelt es sich um eine grünen Cut. Die bedeutung des Programms wird in diesem Fall nicht verändert. Es wird nur eine effizientere Abarbeitung erreicht. Die andere Möglichkeit ist ein roter Cut, dies ist der Fall, wenn ein Cut die Bedeutung eines Programms verändert. Dieser Cut wird verwendet wenn beide Regel wahr sein können, die zweite Regel aber nur berücksichtigt werden soll wenn die erste falsch ist.[6]

4.3.5. Negation

Aus den bisherigen Erklärungen geht hervor, dass Regeln bis jetzt nur auf positive Informationen getestet werden können. Es gibt aber auch die Situation, in der auf negativ Tatsachen abgefragt werden soll. zum Beispiel ist eine Person ohne Kinder, Elternteil von keinem. Dies wird mit dem Cut Operator plus dem vordefinierten Prädikat fail umgesetzt.

```
childless(P):-                  % kinderlos
    parent(_,P),                % Elternteil von irgendjemandem
    !,fail.                     % expliziter Fehler
childless(P).                   % Zweite Regel
```

[1]

Trifft die Erste Regel zu, und P ist Elternteil von irgendjemandem, wird mit Cut abgebrochen. Es wird fail zurück gegeben und die Zweite Regel wird nicht mehr geprüft. Trifft die erste Regel nicht zu, wird die zweite geprüft, diese gibt in jedem fall true zurück.[6]

5. Fazit

Bei logischen Programmiersprachen wird ein komplett anderer Ansatz gewählt, als bei den herkömmlichen Programmiersprachen. Der Programmierer hat die Aufgabe mit Hilfe von Fakten und Regeln eine Wissensdatenbank aufzubauen. Dies wird mit Hilfe von Hornklauseln dargestellt. Bei Hornklauseln handelt es sich um eine Teilmenge der Prädikatenlogik. Diese Wissensdatenbank wird dann vom System verwendet um Anfragen zu beantworten. Dazu wird der Unifikationsalgorithmus, die Resolution sowie das Backtracking verwendet.
In der logischen Programmierung steht die Beschreibung des Problems und nicht der Ablauf der Problemlösung im Vordergrund.

Nach erster Einarbeitungsphase war es für mich sehr spannend einen groben Einblick in diesen Bereich der Informatik zu erhalten. Die logische Programmierung und somit Prolog ist auf ihre Weise ein mächtiges Instrument um Probleme in verschiedenen Bereichen der Informatik und der künstlichen Intelligenz zu lösen. So hat sie zum Beispiel ihren festen Platz im Bereich der Expertensysteme und Planer.
Die logische Programmierung ist aber trotzdem nicht für alle Problemlösungen geeignet.
In Prolog werden die Regeln mit Variablen deklariert und sind somit für verschieden Probleme anwendbar.
Durch den deklarativen Charakter kann Prolog auch verwendet werden um andere Programme zu spezifizieren und den Programmierer so zu unterstützen. Dies öffnet das Anwendungsfeld in weitere Ebenen.

Literaturverzeichnis

[1] Eckerle, Jürgen: *prolog FS2014*. – Script zum Wahlpflichtfach Künstliche Intelligenz

[2] Berner Fachhochschule Schenk: *Modulbeschreibung*. http://www.ti.bfh.ch/fileadmin/modules/BTI7311-de.xml/. Version: April 2014

[3] [Unbekannt]: *Was ist Logische Programmierung*. https://files.ifi.uzh.ch/rerg/arvo/courses/logische_programmierung/ws03/documents/Was_ist_LP.pdf. Version: März 2014

[4] Wikipedia: *Logische Programmierung*. http://de.wikipedia.org/wiki/Logische_Programmierung. Version: März 2014

[5] Frei, Jochen: *Logische Programmierung: Prolog*. https://www.ps.uni-saarland.de/courses/seminar-ws03/LogischeProgrammierung.pdf. Version: März 2014. – Proseminar Programmiersprachen WS 03/04

[6] Uwe Lämmel; Jürgen Cleve: *Künstliche Intelligenz*. Bd. 1. Hanser, 2012

[7] Wikipedia: *Praedikatenlogik*. de.wikipedia.org/wiki/Pr%C3%A4dikatenlogik. Version: März 2014

[8] Eckerle, Jürgen: *Logic Resolution FS2014*. – Script zum Wahlpflichtfach Künstliche Intelligenz

[9] Wikipedia: *Substitution*. http://de.wikipedia.org/wiki/Substitution_(Logik). Version: März 2014

[10] Wikipedia: *Backtracking*. http://de.wikipedia.org/wiki/Backtracking. Version: März 2014

[11] Grabowski, Prof. D.: *SWI-PROLOG - Einführung*. https://www.htwsaar.de/Members/grabowski/uebersicht/vl/material-prolog/ss12/my_skript_prolog_ss_12/download. Version: April 2014

Abbildungsverzeichnis

Tabellenverzeichnis

A. Anhang

A.1. Beispiele

A.1.1. Stammbaum

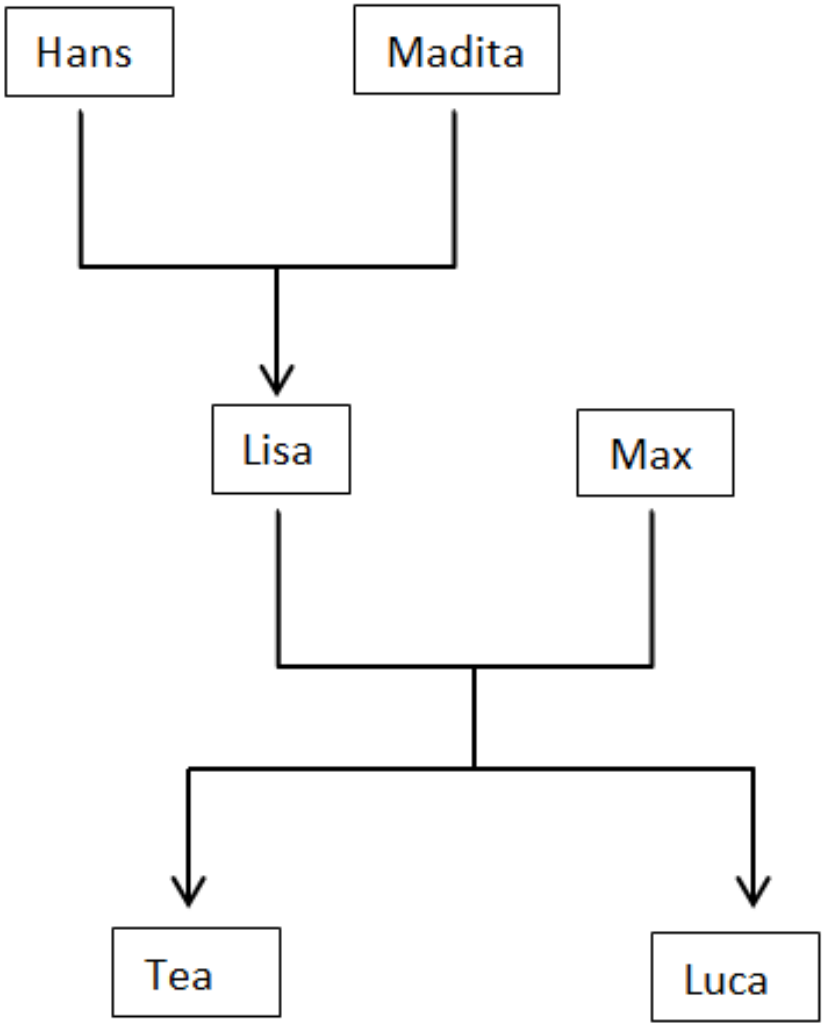

Abbildung A.1.: Beispielstammbaum

```
:- module(family,[
     male/1,
     female/1,
     parent/2,
     mother/2,
     father/2,
     sibling/2,
     grandparent/2,
     grandmother/2,
     grandfather/2,
     greatGrandParent/2,
     ancestor/2
     ]).
          male(max).
          male(luca).
          male(hans).
          female(lisa).
          female(tea).
          female(madita).
          female(doris).
          parent(max,luca).
          parent(max,tea).
          parent(lisa,luca).
          parent(lisa,tea).
          parent(madita,lisa).
          parent(hans,lisa).
          parent(doris,hans).
          mother(M,P):-
               female(M), parent(M,P).
          father(M,P):-
               male(M), parent(M,P).
          child(M,P):-
               parent(M,P).
          ancestor(A,E):-
               parent(A,E).
          ancestor(A,E):-
               parent(A,X),ancestor(X,E).
```

A.1.2. Landkarte

Abbildung A.2.: Landkarte einfärben

```
:- module(map,[
     neighbor/2,
     countries/6
     ]).
          neighbor(red,blue).
          neighbor(blue,red).
          neighbor(green,red).
          neighbor(yellow,red).
          neighbor(red,yellow).
          neighbor(blue,yellow).
          neighbor(green,yellow).
          neighbor(yellow,blue).
          neighbor(red,green).
          neighbor(blue,green).
          neighbor(green,blue).
          neighbor(yellow,green).
          % A=france
          % B=switzerland
          % C=germany
          % D=belgium
          % E=netherlands
          % F=austria
          countries(A,B,C,D,E,F):-
               neighbor(A,B),                    % france - switzerland
               neighbor(A,C),                    % france - germany
               neighbor(A,D),                    % france - belgium
               neighbor(B,C),                    % switzerland - germany
               neighbor(B,F),                    % switzerland - austria
               neighbor(C,D),                    % germany - belgium
               neighbor(C,E),                    % germany - netherlands
               neighbor(C,F),                    % germany - austria
               neighbor(D,E).                    % Begien - netherlands
```