Facharbeit

Logik und Logikprogrammierung mit Prolog

Benedikt Heffels

Kurs: Informatik-GK1

Betreuender Lehrer: Herr Matias

Abgabe: 15.02.2023

Schuljahr: 2022/2023

```
?-  name(logik_und_logikprogrammierung_in_prolog).
Facharbeit von
Benedikt Heffels
true.

?- about.
Herr Matias, IF GK 1, 2022/2023
true.

?- abgabe(heute).
Abgabe: 15.02.2023
true.
```

SWI Prolog

Inhaltsverzeichnis

Titelseite:
Bildquelle SWI-Prolog Logo:
Brooks, K., 2020. Easy Prolog installation on Windows, Linux, macOS. [Online]
Available at: https://kaibrooks.medium.com/easy-prolog-installation-on-windows-linux-macos-a5c5e4b3bc45 [Zugriff am 12 Februar 2023].

1. Einleitung

„Hey Siri, ist ein Porsche ein Kraftfahrzeug?“ Es gibt zwei Wege, wie Siri diese Frage beantworten könnte: A) Apple hat das gesamte menschliche Wissen in einer massiven Datenbank gespeichert. Da stellt sich für mich jedoch die Frage: Wo steht eine Serverfarm mit einer so riesigen Datenbank? Und wie kann Siri diese schnell genug durchsuchen, um mir schon Millisekunden später meine Antwort geben zu können? Datenbanken sind aber nicht das Thema dieser Facharbeit. Vielmehr habe ich mich mit dem anderen Lösungsweg beschäftigt: B) Siri besitzt gar keine massive Datenbank. Das scheint doch logisch, oder? Man muss nur betrachten, wie groß eine solche Datenbank sein müsste, um all diese gedoppelten Fakten zu speichern. Ich könnte auch fragen, ob ein Mercedes-Benz ein Kraftfahrzeug ist. Aber selbst, wenn die Datenbank wesentlich kleiner ist als die des vollumfänglichen Wissens, irgendwie beantwortet Siri mir ja am Ende doch meine Frage. Der Schlüssel: Logik und Logikprogrammierung! Man bringt Siri nicht bei, welche Autos alle Kraftfahrzeuge sind, sondern erklärt ihr, wann ein Auto ein Kraftfahrzeug ist. Daraus resultiert eine wesentlich kleinere Datenbank.

Da ich mich für künstliche Intelligenz interessiere, frage ich mich natürlich nun: Was ist Logik? Wie bringt man dem Computer Logik bei? Und in welcher Sprache geht das überhaupt? All dies sind spannende Fragen, die ich in folgender Arbeit eingehender untersuchen will.

Diese Facharbeit soll dem Konzept der Logik und der Logikprogrammierung auf den Grund gehen. Es soll erläutert werden, woher die Logik überhaupt stammt, und wie sie eigentlich funktioniert. Des Weiteren soll die Programmiersprache „Prolog“, die wohl bekannteste logische Programmiersprache, vorgestellt werden.

1.1. Logikprogrammierung

Logikprogrammierung bezeichnet die Idee, Logik direkt als Programmiersprache zu verwenden. Umgesetzt in Sprachen wie Prolog, ist sie damit Teil der deklarativen Programmierung. Das beschreibt das Ziel, dem Computer lediglich das Wissen über das Anwendungsszenario und das Ziel zu übergeben. Die Lösung des Problems wird dabei dem Computer überlassen. Dies steht im Kontrast zur imperativen Programmierung, wie sie beispielsweise in Sprachen wie Java, Python oder C++ genutzt wird, wo man dem Computer auch den Lösungsweg übergibt (Berkholz, 2019).

Ein weiteres Beispiel wäre das zu Beginn der Entwicklung in Prolog häufig verwendete Beispiel der Vererbung: Vater von Arnd ist Lorenz. Arnd ist wiederum der Vater von Casper. Ist Lorenz also nun der Großvater von Casper?

Solche allgemein gültigen Abläufe lassen sich in Logikprogrammiersprachen sehr viel einfacher als in imperativen Programmiersprachen erstellen. Daher haben logische Programmiersprachen wie Prolog auch in der Entwicklung künstlicher Intelligenzen eine hohe Bedeutung (Westhouse-Group, 2021).

1.2. Prolog

Prolog ist die wichtigste logische Programmiersprache, auch wenn einige Bestandteile nicht logisch sind. Die Erfinder waren Kowalski und Colmerauer Anfang der 1970er Jahre in Merseilles. Prolog steht dabei für den französischen Begriff „Programmation en logique“ (Berkholz, 2019). Manchmal wird der Name aber auch von „Programming in Logic“ (Bramer, 2013, p. v) oder „Programmieren in Logik“ (Fuchs, 1990, p. 2) hergeleitet.

1.2.1. SWI-Prolog

Eine der bekanntesten Versionen von Prolog dürfte dabei „SWI-Prolog“ sein. Diese Version der Programmiersprache ist Open Source, kann also theoretisch von jedem bearbeitet werden. Außerdem wird in aktueller Fachliteratur über Logikprogrammierung und Prolog aus verschiedenen Gründen häufig auf SWI-Prolog zurückgegriffen (etwa Bramer, 2013).

SWI-Prolog selbst ist hauptsächlich in C programmiert, was zum Vorteil hat, dass man die Software auf fast jedem Betriebssystem verwenden kann (SWI-Prolog, kein Datum). Der Hauptentwickler von SWI-Prolog ist Jan Wielemaker, der parallel auch als Wissenschaftler an der „Vrije Universiteit Amsterdam“ tätig ist (Wielemaker, kein Datum). Neben ihm wird die Open Source Version auf GitHub aber noch von vielen anderen bearbeitet. Darunter sind auch einige Unternehmen (SWI-Prolog, kein Datum).

1.2.2. Verwendete Software

Aus diesen vorherig genannten Gründen wird in dieser Facharbeit SWI-Prolog (im folgenden Synonym zu „Prolog“ eingesetzt) verwendet.

Die Java-Programme wurden in der – auch in der Schule verwendeten – Software von JetBrains implementiert. Dabei fand das Open JDK 17 (Bellsoft Liberica JDK) in der Komplettversion Verwendung. Auch dieses ist – abgesehen von einem möglichen Versionsunterschied – simultan zum JDK, welches in der Schule Verwendung findet.

Die im Rahmen dieser Facharbeit entstandenen Programme können auch über GitHub abgerufen werden (siehe dazu „Webverweis“ im „Literatur- und Quellenverzeichnis“).

2. Hauptteil

2.1. Logik

Hinter der Logik an sich steckt eine sehr lange Geschichte, die von Hölldobler in seinem Werk „Logik und Logikprogrammierung“ aus dem Jahr 2003 beschrieben wird (Hölldobler, 2003). Ihre Grundzüge gehen zurück auf Aristoteles und die Ägypter. Aus deren Regeln entstand später dann die Logik, welche wiederum zur Logikprogrammierung wurde.
Am wichtigsten sind jedoch folgende drei ursprüngliche Bestandteile:

2.1.1. Formalisierung

Überlieferungen zufolge geht die erste Idee der Formalisierung auf Aristoteles zurück. Dieser war davon überzeugt, dass sich Folgerungen einzig und allein aus der Form der beteiligten Sätze ziehen lassen und das unabhängig von der Bedeutung dieser. Dafür führte er die sogenannten Syllogismen ein, welche man wie in Formel 1 dargestellt betrachten kann:

$$\frac{\begin{matrix} SeP \\ PeQ \end{matrix}}{SeQ}\ unter\ dem\ Grundsatz\ XeY$$

Formel 1: Syllogismen

Wichtig ist bei dieser Formel vor allem das „XeY“. Das steht für „alle X sind Y“. Übertragen auf die Formel bedeutet das wiederum auch, dass alle „S“ „P“ sind und alle „P“ „Q“ sind. Zusammen führen diese beiden Teile zu dem Schluss, dass alle „S“ auch „Q“ sein müssen.
Ein Beispiel: Lassen wir „S“ einen Porsche sein. Zudem ist „P“ ein Auto und „Q“ ein Kraftfahrzeug. Dann gelten folgende Zusammenhänge: Alle Porsche sind Autos und alle Autos sind Kraftfahrzeuge. Man könnte hier unter Anwendung der Syllogismen auch von „ein Porsche ist ein Auto“ und „alle Autos sind Kraftfahrzeuge“ sprechen. Im Zusammenhang bedeutet das wiederum, dass alle Porsche Kraftfahrzeuge sind. Diesen Zusammenhang kann man auch in Formel 2 ausdrücken:

$$\frac{\begin{matrix} auto(porsche) \\ (f\ddot{u}ralle\ X)\ (wenn\ auto(X)\ dann\ kraftfahrzeug(X)) \end{matrix}}{kraftfahrzeug(porsche)}$$

Formel 2: Syllogismen am Porsche-Beispiel

Oder etwas mathematischer:

$$\frac{\begin{matrix} a(p) \\ (\forall X)\ (a(X) \rightarrow k(X)) \end{matrix}}{k(p)}$$

Formel 3: Verallgemeinerter Syllogismus

In dieser Formel steht nun das „p“ für Porsche, dass „a“ für Auto und das „k“ für Kraftfahrzeug. Der Pfeil symbolisiert die wenn-dann-Verbindung, wie er es auch in manch einer Programmiersprache tut. Und das „∀“ bedeutet so viel wie das „füralle“. Dementsprechend ist die obere Zeile eine Aussage – ein Fakt (vergleiche „2.2.2.1. Fakten“) – und die untere Aussage ein Konditional – eine Regel (vergleiche „2.2.2.3. Konjunktive Fragen: Regeln“). Anzumerken sei auch noch, dass im Schluss kein Wissen über den Porsche selbst verwendet wurde. Die Regel wäre auch dann noch gültig, wenn man für

„a“, „p“ und „k“ komplett andere Inhalte einsetzen würde, die „k“ trotzdem erfüllen. Beispielsweise könnte „a“ für Mensch stehen, „p“ für Aristoteles selbst und „k“ für „ist sterblich“ stehen. Die logische Schlussfolgerung aus dieser Formel wäre dann, dass Aristoteles sterblich ist. An dieser Stelle ist es sicherlich eine Erwähnung wert, dass Aristoteles im Jahre 322 vor Christus verstorben ist. Die Formel trifft also erneut zu.

2.1.2. Kalkülbildung

Aber Aristoteles und seine Syllogismen waren nicht als Einzige an der Entwicklung der modernen Logik beteiligt. Auch die antiken Ägypter spielten ihre Rolle in der Entwicklung der Grundidee. So sollen die Ägypter dem griechischen Geschichtsschreiber Herodot († 430 v. Chr.) zufolge bereits mit Steinen gerechnet haben, die Römer und Griechen nutzten später den Abakus. All dies sind Systeme von Regeln, die aus Anwendung bestimmter Figuren beziehungsweise Stellungen neue Figuren und Stellungen erzeugen. Dieses Konzept bezeichnet man als Kalkül und ist noch heute vergleichbar mit dem Rechenschieber.

2.1.3. Mechanisierung

Auch die Idee der Mechanisierung soll nach Herodot auf die antiken Ägypter zurückgehen. Sie entwickelten so schon für ihr Militär verschiedene Apparaturen, die sie als „mechanai“ bezeichneten. Seit Beginn des 19. Jahrhunderts nimmt auch die Mechanisierung in unseren Kreisen immer mehr Fahrt auf und – besonders in der heutigen Zeit – werden immer mehr Aufgaben an Computer übertragen.

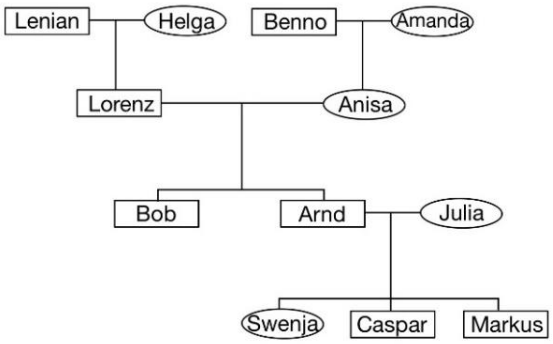
2.1.4. Verbindung der drei Ideen

Die Logik, wie wir sie heute kennen, hat sich aus einer Kombination dieser drei Ideen entwickelt. Geforscht und gearbeitet wurde daran schon seit Jahrhunderten. Aus diesen drei Grundideen entwickelten sich über die Zeit verschiedene Arten der Logik. Prädikatenlogik und Aussagenlogik fußen so unter anderem auf diesen drei Prinzipien. Ziel der Forschenden war es lange Zeit für alle Sätze F und G zu beweisen, dass F genau dann logisch aus G folgt, wenn F aus G formal ableitbar ist. Dieser Zusammenhang wird auch als „Vollständigkeitsresultat“ bezeichnet, und konnte bereits im Jahre 1930 bewiesen werden. Das erfolgte jedoch erst nach der Entwicklung der Prädikatenlogik, die auch Syntax und Semantik betrachten konnte. Die zuvor entwickelte Aussagenlogik konnte nur verschiedene Aussagen zueinander in Beziehung setzen. Man „entdeckte“ damals auch, dass es verschiedene Stufen der Logik gibt. Die verschiedenen Stufen beschreiben dabei eine unterschiedlich fortschrittliche Logik.
Aber nicht nur ein Großteil der Programmiersprachen fußt in der Logik. Auch technische Gerätschaften, wie etwa die Turing-Maschine oder die Zuse-Computer wurden im Rahmen der Logik entwickelt. Ihre Geschichte und die der Logik sind bis heute eng verstrickt (Hölldobler, 2003).

2.2. Erstes Programm: Familienstammbaum

Die unter „Formalisierung“ beschriebenen Eigenschaften der Logik lassen sich bis heute in der logischen Programmierung erkennen. Die obere Aussage des Syllogismus würde dabei einen Fakt widerspiegeln, also eine Definition oder Zuordnung. Die „wenn-dann-Verbindung“ des Syllogismus wäre in dieser Perspektive eine Regel. So beweist diese von Aristoteles entwickelte Formel noch heute automatisiert Informationen.

Wie das in Prolog genau aussieht, soll in diesem Kapitel dargestellt werden. Dabei wird in Prolog häufig statt mit dem üblichen „Hello World“ Programm bereits mit einem Stammbaum, also einer logischen Zuordnung, begonnen. Dies macht auch Sinn, denn Prolog wurde nicht wie viele andere Programmiersprachen zur reinen (grafischen) Ausgabe entwickelt, sondern zur Arbeit mit logischen Verbindungen.



Grafik 1: Stammbaum der Familie

2.2.1. Familienstammbaum der implementierten Familie

Bevor auf Basis eines Stammbaumes eine Programmstruktur erschaffen wird, ist es wichtig, sich diesen in den Kopf zu rufen. Auf Basis der „puren“ Fakten ist dies nur noch schwer möglich.

2.2.2. Umsetzung in Prolog

Prolog Programmcode besteht aus verschiedenen Klauseln. Diese lassen sich in zwei verschiedene Teile unterscheiden: Fakten und Regeln (Fuchs, 2023).

2.2.2.1. Fakten

Fakten werden in Prolog deklarativ verwendet. Es wird also eine bestimmte Sache einfach definiert. Häufig ist es bei Prolog so, dass man dem Prolog-Programm einfach einen Sachverhalt beschreibt und sich nicht darum kümmert, wie das dahintersteckende System ein Problem löst. In diesem Kontext ist auch häufig die Rede von Datenlogik: Man arbeitet mit einer Datenbank an übergebenen Informationen (Hölldobler, 2003).

Auf der linken Seite sind die Fakten für mein erstes Programm, den Familienstammbaum, zu sehen. Dabei werden verschiedene Personen auf verschiedene Weise definiert. Ich erschaffe also eine Datenbank.

„lenian“ ordne ich also in der ersten Zeile beispielsweise den Fakt „mann“ zu. Lenian ist also ein Mann. In diesem Fall stellt „lenian“ ein Konstantensymbol (Hölldobler, 2003, p. 18) dar: „lenian“ ist ein konstanter Wert, dem ich verschiedene Eigenschaften zuordnen kann. So ordne ich ihm mit „vater(lorenz, lenian)“

familienstammbaum.pl [modified]

```
mann(lenian).
frau(helga).
mann(benno).
frau(amanda).

mann(lorenz).
frau(anisa).

mann(bob).
mann(arnd).
frau(julia).

frau(swenja).
mann(caspar).
mann(markus).


vater(lorenz, lenian).
mutter(lorenz, helga).
vater(anisa, benno).
mutter(anisa, amanda).

vater(bob, lorenz).
mutter(bob, anisa).
vater(arnd, lorenz).
mutter(arnd, anisa).

vater(swenja, arnd).
mutter(swenja, julia).
vater(caspar, arnd).
mutter(caspar, julia).
vater(markus, arnd).
mutter(markus, julia).
```

Programmcode 1: Fakten in Prolog

etwa zu, dass sein Sohn die Konstante „lorenz“ sein soll. Dementsprechend steht nun in meiner Datenbank, dass Lenian der Vater von Lorenz ist.[1]

```
?- mann(lenian).
true.

?- frau(lenian).
false.

?- vater(lorenz, lenian).
true.

?- vater(lorenz, benno).
false.

?- mann(X).
X = lenian ;
X = benno ;
X = lorenz ;
X = bob ;
X = arnd ;
X = caspar ;
X = markus.

?- vater(X, arnd).
X = swenja ;
X = caspar ;
X = markus.
```

Programmcode 2: Abfragen in Prolog

2.2.2.2. Verwendung des Programms: Queries

Schon jetzt kann ich mir in der Konsole verschiedene Informationen, basierend auf diesen Fakten, ausgeben lassen. Diese Anfragen an das Programm werden als „Queries“ bezeichnet. Auf dem Bild links kann man erkennen, wie solche „Queries“ aufgebaut sind: Zu Beginn steht immer das Eingabeaufforderungszeichen „?-“, auch bezeichnet als „System Prompt“ („Prompt“ wird auch im deutschen verwendet). Dahinter gebe ich ein, was ich wissen will – also meine Frage beziehungsweise „Querie“. Die Antwort auf eine einfache Überprüfungsfrage, also ob ein Fakt existiert, kann „true“ (existiert) oder „false“ (existiert nicht) sein (Bramer, 2013).

Ein Beispiel dafür sind die Queries „mann(lenian)“ und „frau(lenian)“. Da „lenian“ als Mann definiert wurde, ist die Antwort auf die erste Anfrage „true“ und auf die zweite „false“.

Aber auch komplexere Anfragen sind möglich. Mit „komplexer“ meine ich in diesem Fall Anfragen, bei denen Prolog selbst noch eine Antwort berechnen muss. Meist sind dies Anfragen, bei denen zwar in einer Variable ein mögliches Konstantensymbol übergeben wird, die andere aber als „X“ undefiniert bleibt. Ein Beispiel ist die Anfrage „vater(X, arnd)“. Hier soll Prolog mir nun alle Kinder von „arnd“ liefern. Man könnte aber auch alle übergebenen Werte durch Variablen ersetzen: Die Anfrage „vater(X,Y)“ würde zum Beispiel alle Vater-Kind-Paare liefern.

2.2.2.3. Konjunktive Fragen: Regeln

Fakten sind nicht das einzige elementare Konzept von Prolog. Abgesehen von diesen gibt es auch noch Regeln.

Fakten werden dabei als explizit abgesicherte Relationen angesehen. Diese stehen also, so wie die Konsole sie ausgibt, im Code. Doch das mag nicht immer reichen: Manchmal braucht man auch Zugriff auf die implizit vorhandenen Informationen, die nicht direkt im Code beziehungsweise der Datenbank stehen. Die Informationen, die durch Formalisierung und Syllogismen definiert sind (vergleiche dazu „2.1.1. Formalisierung“).

[1] Anmerkung: Dieses Familienbeispiel wird auch in verschiedener Literatur aufgegriffen, zum Beispiel bei Hölldobler (Hölldobler, 2003). Hier wäre die Definition andersherum: Zuerst der Vater, dann der Sohn. Wenn man das im Programm beachtet, macht dies jedoch keinen Unterschied.

familienstammbaum.pl [modified]

```
schwester(X, Y) :-
        vater(X, Z),
        vater(Y, Z),
        frau(Y),
        not(frau(X) == frau(Y)).

bruder(X, Y) :-
        vater(X, Z),
        vater(Y, Z),
        mann(Y),
        not(mann(X) == mann(Y)).

grossvater(X, Y) :- % vaeterlicher Seits
        vater(X, Z),
        vater(Z, Y).
grossvater(X, Y) :- % muetterlicher Seits
        mutter(X, Z),
        vater(Z, Y).

grossmutter(X, Y) :- % muetterlicher Seits
        mutter(X, Z),
        mutter(Z, Y).
grossmutter(X, Y) :- % vaeterlicher Seits
        vater(X, Z),
        mutter(Z, Y).

tante(X, Y) :-
        vater(X, Z),
        schwester(Z, Y).

onkel(X, Y) :-
        vater(X, Z),
        bruder(Z, Y).
```

Programmcode 3: Regeln in Prolog

Im Beispielcode (Programmcode 1) habe ich etwa das Geschlecht und die Eltern einer Person (gemeint sind die Konstantensymbole) definiert. Trotzdem kann es mich doch auch interessieren, wer etwa die Großmutter oder der Großvater einer Person ist.

Dafür müssen verschiedene Bedingungen erfüllt sein: Mein Großvater ist auch der Vater meines Vaters. So muss ich dann nur noch überprüfen, ob mein Großvater einen Sohn hat, der mein Vater ist. Und dasselbe gilt natürlich auch für meine Mutter.

In Prolog werden solche Regeln, wie links dargestellt (Programmcode 3), umgesetzt. Hier übergibt man Prolog zwei Variablen (auch der Einsatz von mehr oder weniger Variablen ist denkbar), die in der Konsole im Anschluss mit (möglichen) Konstantensymbolen gefüllt werden. Mit diesen zwei Variablen führt der Computer dann eine Abfrage in meiner Datenbank durch. Wichtig zu beachten dabei ist, dass als Ergebnisse nur diejenigen Daten ausgegeben werden, die alle Aspekte der Regel erfüllen (mit „true“ beantworten). Dabei ist es auch möglich, zwei Daten miteinander zu vergleichen oder ein falsches Ergebnis in ein wahres zu konvertieren. Dafür setzt man einfach ein „not(…)“ davor. Wenn das Ergebnis in dieser Klammer also falsch ist, ist es als Ganzes wieder wahr und andersherum.

2.3. Prolog: Syntax

Nachdem die Prinzipien der Programmierung in Prolog nun behandelt wurden, wird es Zeit für einen Blick auf die Prolog-Syntax. Denn diese ist zwar in Grundzügen ähnlich zu den bekannten Programmiersprachen, aber doch unterschiedlich. Auffällig ist so etwa, dass jede Klausel mit einem Punkt endet. Das ist in etwa vergleichbar zum Semikolon in Java. Doch trotzdem ist das auch ein Unterschied: Regeln, die man mit Funktionen vergleichen könnte, enden mit Punkt statt mit einer geschweiften Klammer. Wenn in einer Regel innerhalb des Rumpfes mehrere Abfragen vorkommen, werden diese durch ein Komma voneinander getrennt. Auch das stellt einen Unterschied zu dem aus Java bekannten dar, denn dort endet jede Aussage mit einem Semikolon. Die Definition einer Regel – also ihr Kopf – unterscheidet sich auch ein wenig von der Definition einer Funktion dort: Statt hier geschweifte Klammern zu verwenden, wird ein Doppelpunkt-Bindestrich („:-“) genutzt. Hölldobler schreibt sogar davon, dass Nutzer, welche an Programmiersprachen wie C oder Pascal gewohnt seien, sich über die Syntax wundern würden (Hölldobler, 2003, p. 15). Abgesehen davon fällt auf, dass der gesamte Programmcode

in Kleinbuchstaben verfasst wurde. Das Einzige, was großgeschrieben wird, sind undefinierte Variablen wie „X“ und „Y“. Auch das ist ein Unterschied zu anderen Programmiersprachen aus dem Unterricht.

2.4. Logikprogrammierung

> *„Die Logikprogrammierung ist trotz ihres geringen Alters ein etabliertes Gebiet mit intensiver Forschungstätigkeit und einer stetig wachsenden Zahl von Anwendungen. Logische Programmierung verbindet in überzeugender Weise Theorie und Praxis.“*
> *Norbert E. Fuchs: Kurs in logischer Programmierung (Fuchs, 1990, p. V)*

Aus der Logik entstand über die Zeit eine logische Programmierrichtung, auch genannt Logikprogrammierung. Wie bereits mehrfach erwähnt, ist die bekannteste Programmiersprache für Logikprogrammierung Prolog. In diesem Kapitel sollen einige der (doch speziellen) Techniken der Logikprogrammierung und SWI-Prolog erläutert werden.

2.4.1. Darstellung von Wissen: Deklaration

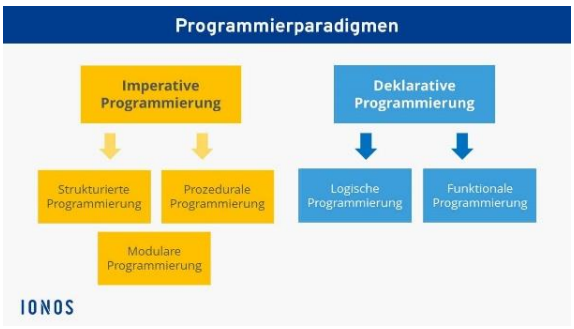
$$K :- B_1, B_2, \dots, B_n$$

Formel 4: Beispielhafte Klausel mit Bedingungen

Die Klausel aus Formel 4 zeigt eine Konsequenz K. Diese gilt (ist wahr), wenn die Voraussetzungen (Bedingungen) B_1, B_2, … und B_n gelten.
Diese Regel ist deklarativ aufgebaut: Hier liegt das Wissen explizit vor und ist (mit dem Inhalt der verschiedenen Bedingungen) eigenständig verständlich. Ein Beispiel wäre dafür die Methode „maximum(X,Y) :- X>=Y.“. Diese Methode liefert true, wenn X größer als Y ist (und dementsprechend false, wenn dies nicht der Fall ist).
In diesem Kontext wird auch der Begriff „referentielle Transparenz“ verwendet. Dieser Begriff bezeichnet erneut, ob eine Aussage für sich alleinstehen kann. Wenn ja, ist sie referentiell transparent.
Abgesehen von der Deklaration kann man Wissen aber noch auf andere Weise darstellen, etwa prozedural[2]. Dann ist das Wissen fest mit einer Prozedur verbunden. Dies bedeutet aber wiederum, dass die Anweisung nicht eigenständig verstanden werden kann. Die eben verwendete Methode „maximum“ wäre so zum Beispiel prozedural, wenn man stattdessen eine if-else-Anweisung einbauen würde: Um zu kontrollieren, ob X größer ist als Y, muss diese Schleife durchlaufen werden. Die Anweisungen in ihr sind nicht mehr eigenständig verständlich.
Die Deklaration hat auch folgende praktische Vorteile: So kann deklaratives Wissen wegen der referentiellen Transparenz für verschiedene Zwecke verwendet werden, auch für welche, an die man ursprünglich noch gar nicht gedacht hat. Zudem kann aus deklarativem Wissen anderes, nicht explizit dargestelltes Wissen, abgeleitet werden.
Mit der Deklaration zur Prozedur verhält es sich so wie mit der Iteration zur Rekursion: Ein deklaratives Programm kann meist auch als prozedurales Programm verfasst werden.

[2] Teilweise wird das prozedurale Programmierparadigma als Teil des imperativen Programmierparadigma oder als Synonym für dieses angesehen. Die Definition hier ist aber nicht ganz eindeutig (Akademie der Wissenschaften und der Literatur Mainz, kein Datum).

Fuchs schreibt davon, dass die Interpretation eines Prolog-Programmes durch den Prolog-Interpreter prozedural sei. Dabei wird also erst der Prolog-Code unvollständig gemacht, wodurch man ein prozedurales Programm bekommt. Dieses kann abweichend sein vom deklarativen Programm (Fuchs, 1990, pp. 3-5).



Grafik 2: Programmierparadigmen (IONOS, 2021)

2.4.1.1. Unterschied zur imperativen Programmierung

Sprachen wie Java, C++ oder Python sind Teil des imperativen Programmierparadigma. „Imperativ" stammt dabei vom lateinischen Wort „imperare", „befehlen", ab. In imperativen Sprachen reihen sich dabei meist verschiedene Befehle aneinander. Unterstützt werden diese Befehle von Kontrollstrukturen wie Schleifen oder Verzweigungen. Diese gibt es in deklarativen Programmiersprachen so nicht.[3] Der Quellcode imperativer Sprachen soll so sehr konkret sein und arbeitet dabei nahe am System. Dies führt jedoch auch zu längeren Quelltexten. Der Unterschied zwischen diesen Programmierparadigmen hängt sicherlich auch damit zusammen, dass sie unterschiedlich eingesetzt werden sollen (IONOS, 2021).

2.4.2. Rekursion

Im Unterricht haben wir Rekursion als das erneute Aufrufen einer Methode, um ein Ergebnis zu erhalten, definiert.[4] Statt also wie bei der Iteration Schleifen zu verwendet, wenn Informationen mehrfach ergänzt werden müssen, wird bei der Rekursion also die Methode selbst erneut aufgerufen. Dieses Prinzip der Rekursion ist auch zentrale Programmiertechnik in Prolog. Das liegt daran, dass die zentrale Datenstruktur in Prolog sogenannte Terme sind. Diese sind ebenfalls rekursiv entwickelt.

2.4.2.1. Terme

Die zentrale Datenstruktur in Prolog bezeichnet man als Terme (beziehungsweise vollständig „Terme erster Ordnung"). Sie spezifizieren die Daten in Prolog. Beim Term ist aber weniger vom mathematischen Term die Rede, vielmehr von der Aussage eines Elementes (Weisweber spricht bei diesem von einem Atom). Dabei ist die Prädikat-Argument-Notation wichtig, welche den Aufbau eines Termes darstellt. Das heißt, wenn wir zum Beispiel von „mann(lenian)" (vgl. „2.2.2.1. Fakten") sprechen, sprechen wir von „mann" als Prädikat und „lenian" als Argument. Dabei kann das Prädikat beliebig viele Argumente (n-Argumente) beinhalten, sodass man von einem n-stelligen Prädikat spricht (Weisweber, 1997, pp. 16-18).

2.4.3. Abrufen von Wissen: Backtracking

Das Backtracking ist eines der zentralen Elemente von Prolog. Man spricht vom Backtracking, wenn das Prolog-System weitere Ergebnisse für eine Fragestellung ausgibt.

[3] In der reinen deklarativen Programmierung gibt es diese nicht. Allgemein sind sie aber auch in Prolog integrierbar (siehe dazu „3.1. Ausblick").

[4] Genauer genommen wird bei der Rekursion ein Problem auf ein leichter lösbares Problem, derselben Art, reduziert. Um dies jedoch hinzukriegen, braucht es praktisch immer eine Methode, die sich erneut aufruft und währenddessen das Problem löst. Meist übergibt sie sich dafür zusätzlich selbst noch Werte.

Das ist möglich, wenn man nach der ersten Lösungsausgabe, vor der Ausgabe des System-Prompt („?-“) ein Semikolon[5] eingibt. Backtracking kann so auch an der Abbildung „Programmcode 2“ (2.2.2.2. – Verwendung des Programms: Queries) erkennen: Dort wird das System durch die Eingabe des Semikolons gezwungen, über Backtracking weitere Lösungen, beispielsweise für die Kinder von Arnd, zu finden. Im Rahmen von Backtracking wird in Fachliteratur teilweise die präzisere Aussage „forcing the Prolog system to backtrack“ verwendet (Bramer, 2013, p. 7). Auf Deutsch bedeutet das so viel wie „Rückverfolgung“ (Weisweber, 1997, p. 16). Backtracking kann aber nicht nur manuell aufgerufen werden: Das Wissen in Prolog lässt sich in der Regel als Baum darstellen. Erreicht der Interpreter nun einen fail-Knoten (Knoten, der false ist und/oder einen falschen Wert zurückgibt), wendet er Backtracking an (vergleiche hierzu Anhang 1 (A1) und Grafik 3).
Für den Abruf von Wissen ist das sogenannte „satisfying“ wichtig. Man will also Prolog (beziehungsweise den Compiler) glücklich machen, in dem man alle Bedingungen erfüllt. Beim Backtracking spricht man nun von „resatisfying“, also dem Finden eines anderen Weges, der Prolog glücklich macht (Bramer, 2013, p. 39). Beim Backtracking geht Prolog vom aktuellen Knoten zurück zum Wurzelknoten. Die Entscheidungswege, die zum aktuellen Knoten geführt haben, werden also beibehalten. Nur an der letzten Gabelung wird die andere Verzweigung verfolgt. Dabei wird häufig auch die Belegung der (ausgegebenen) Variable wieder rückgängig gemacht, beziehungsweise die aller Variablen, die ab dem Knoten verändert wurden (Weisweber, 1997, pp. 28, 63-64).

2.4.4. Verwendung

Viele Nutzer von Prolog betonen, dass Prolog mit einfachen Programmierbausteinen eine große Basis für das Lösen von sehr komplexen Problemen bietet (Bramer, 2013, p. viii) (vergleiche etwa „2.6. Logikrätsel“). Dies trifft auch auf die gesamte Logikprogrammierung zu. Prolog im Speziellen wird aber schon für viele breit gefächerte Anwendungen verwendet. So gibt es zum Beispiel Prolog-Anwendungen, die einen Text in einer natürlichen Sprache (etwa Deutsch) verstehen können, Fragen über den Inhalt beantworten und diesen sogar im Anschluss übersetzen können. Auch wird Prolog verwendet, um etwa soziale Netzwerke zu analysieren oder große Datenbanken zu verwalten. Ein Beispiel dafür wäre die Datenbank des „Human Genome Project“. Dies sind nur einige wenige Beispiele, die beweisen, wie vielfältig und themenübergreifend Prolog und Logikprogrammierung eingesetzt werden können. Von Medizin über Technik bis Recht ist alles dabei (Bramer, 2013, p. viii). So wird Prolog zurecht als wichtige Programmiersprache für künstliche Intelligenz bezeichnet. Dabei wird vor allem die Codierung von Programmierschnittstellen hervorgehoben (Westhouse-Group, 2021). So gibt es viele Forschungsprojekte, die die Logik über Deep Learning in die künstliche Intelligenz bringen wollen (Aldinger, 2021). Das erfordert auch die Verwendung von Logikprogrammierung.

2.5. Warum Prolog? Vergleich zu Java

Nachdem nun mit dem Familienstammbaum bereits ein erstes Beispielprogramm verfasst wurde, stellt sich natürlich die Frage, warum für Logikprogrammierung Prolog ver-

[5] In der aktuellen, verwendeten Prolog-Version reicht auch ein Druck auf die Leertaste (Space). Eine Eingabe des Punktes hingegen stoppt die Ausgabe von weiteren Lösungen.

wendet wird. Dafür soll der Familienstammbaum aus Prolog in Java implementiert werden, um im Anschluss den Funktionsumfang zu vergleichen. Für die Implementation in Java soll auf die behandelten Befehle des Unterrichts zurückgegriffen werden.

2.5.1. Implementation in Java

Als erstes fällt auf, dass Java natürlich ganz anders aufgebaut ist, besonders im Vergleich zu Prolog. Das fängt schon mit der Form der Programmiersprache an: Während Prolog eine deklarative Programmiersprache ist, ist Java eine imperative. Dementsprechend lassen sich die Personen (Konstantensymbole) auch nicht so schön deklarieren wie in Prolog.

Zur Vereinfachung wurde für diesen Versuch die Anzahl der Fakten und Regeln reduziert: In Java wurden überwiegend die Männer (und dementsprechend männlichen Familienmitglieder) übersetzt.

2.5.1.1. Übersetzte Fakten

Wichtigster Teil der Logikprogrammierung sind die Fakten. Diese direkt aus Prolog in Java zu übersetzen, wird sich jedoch als schwierig gestalten, da Java nicht deklarativ aufgebaut und damit komplett unterschiedlich zu Prolog ist (vgl. Hölldobler, 2003, p. 15). Daher musste in der Java-Implementation auf Variablen zurückgegriffen werden. Da ebenfalls die Prädikatsklauseln nicht eins zu eins übersetzt werden können, musste der Umweg über verschiedene Klassen genommen werden. Dies ist jedoch im Beispielprogramm noch halbwegs akzeptabel zu meistern, indem man die Konstantensymbole und Java-Variablen einfach in einen Array überführt, denn dieser ermöglicht es, weiterhin auf alle Daten zu zugreifen.

Prolog-Fakt	Java-Fakt
mann(caspar). vater(caspar, arnd). mutter(caspar, julia).	`Mann mannARR[] = new Mann[7];` `mannARR[5] = new Mann("caspar", "arnd", "julia");`

Tabelle 1: Vergleich Implementierung von Fakten Java-Prolog

Auf den ersten Blick sieht die Java-Definition natürlich einfacher aus, man spart sich insgesamt eine Codezeile. Doch sie bringt auch einen Nachteil mit sich: Zusätzlich wird eine weiter Klasse „Mann“ benötigt, um die Prädikatsklausel zu ersetzen. Diese braucht zudem einen aufwändigeren Konstruktor, um alle Konstanten zu spreichern.[6] Ein Speichern über Setter-Methoden wäre noch aufwändiger. Prolog braucht all dies nicht. In der Klasse „Mann“ müssen zudem Getter-Methoden vorhanden sein, um im Anschluss die zugeordneten Konstanten (Variablen) zurückzubekommen.

2.5.1.2. Übersetzte Regeln

Regeln sind tatsächlich ein Prinzip, das in Java und Prolog ähnlich funktioniert – auch wenn einige Autoren der Ansicht sind, man müsste alles Wissen aus den verschiedenen

[6] Aufmerksamen Lesern wird aufgefallen sein, dass es im Programm auch eine Generation ohne gespeicherte Eltern gibt (hier: Lenian und Benno). Diese wurde natürlich mit übernommen. Hier habe ich auf das sogenannte „überladen von Methoden“ zurückgegriffen, und die beiden ohne Eltern definiert. Theoretisch könnte man für die Eltern auch einfach den Wert „null“ übergeben, bei größeren Datenmengen wäre dies jedoch aufwendiger als die Entwicklung eines weiteren Konstruktors, der diese Werte einfach nicht verlangt (eine andere Paramtereliste besitzt). Java erkennt nun auf Basis der übergebenen Parameter, welcher Konstruktor (beziehungsweise allgemein welche Methode) benötigt wird (Brass, 2013/2014).

Programmiersprachen vergessen, um Prolog zu lernen (Bramer, 2013, p. vii): Beide Programmiersprachen bieten Abfragen, ob etwas stimmt, an. Bei Java übernimmt diese elementare Rolle in der Regel die if-Abfrage.

Prolog-Regel	Java-Regel[7]

bruder(X, Y) :-
vater(X, Z),
vater(Y, Z),
not(mann(X) == mann(Y)).
	public static String bruder (String x, String y){
	for (Mann mann : *mannARR*) {
	if (mann.getVater() != null) {
	String vater = mann.getVater();
	for (Mann value : *mannARR*) {
	if (value.getVater() != null) {
	if (value.getVater()
	.equals(vater) && !mann.
	equals(value)) {
	System.*out*.println("X: " +
	mann.getName() + ", Y: " +
	value.getName() + "; ");
	}
	}
	}
	}
	}
	}

Tabelle 2: Vergleich Implementierung von Regeln Prolog-Java

Im direkten Vergleich der Regeln in den beiden Programmiersprachen sind einige Unterschiede zu erkennen: So ist zum Beispiel der Java-Code mehr geschachtelt als der Prolog-Code. Das hängt damit zusammen, dass Java nicht direkt auf die Definitionen zurückgreifen kann, wie es etwa Prolog tut. Java muss also dauerhaft den gesamten definierten Array durchlaufen.[8] Und das kostet Speicherplatz (und Laufzeit). Abgesehen davon ist der Aufbau ähnlich: In beiden Programmiersprachen wird über den Vater nach dem Bruder gesucht. Und in beiden Programmiersprachen wird am Ende kontrolliert, dass die beiden Brüder nicht derselbe sind. Nur das Ganze ist sehr unterschiedlich aufgebaut.

2.5.1.3. Übersetzte Abfrage: Queries

Am wichtigsten ist bei beiden Programmen die Abfrage im Anschluss. Ohne diese bringt das ganze Programm nämlich nichts – ich will die Daten ja nicht nur entwickeln, sondern auch einsehen können.

Der direkte Vergleich zeigt: Die Ausgabe sieht mit etwas Programmierarbeit in Java und Prolog sehr ähnlich aus. Das liegt wahrscheinlich größtenteils daran, dass die Ausgabe in Java der in Prolog nachentwickelt wurde. Doch diese umständliche Nachentwicklung in Java kostet Laufzeit und Speicherplatz: Im Gegensatz zu Prolog reicht in Java nicht

[7] Theoretisch wäre auch ein Rückgriff auf die Erkennungsmethoden „mann“ beziehungsweise „vater“ möglich. Abgesehen davon, dass diese Methoden erst nach der Fertigstellung der Regeln in eigene Methoden überführt wurden, wäre das auch sehr schwer und nahezu unpraktikabel. Das Problem lässt sich zum Beispiel beim Vater erkennen: Wenn ich ihm nur Y, also den Vater, übergeben würde, würde er die Methode nach dem Finden des 1. Sohnes (X) abbrechen. Ein erneutes Ausführen der Methode würde erneut diesen zurückgeben. Und ein Löschen aus dem Array ist aufgrund des weiterhin benötigten Zugriffes nicht sinnvoll. Dementsprechend ist das zwar nicht die schönste Lösung, aber die Einfachste.

[8] Dies müsste das Programm auch tun, wenn ich auf die Methoden „mann“ und „vater“ zurückgreifen würde. Nur dann würde die Codezeile weniger wiederholt werden.

die einmalige Definition einer Regel oder einer Zuordnung, wie sie in „Übersetzte Regeln“ dargestellt wurde, um dasselbe Ergebnis zu bekommen. Für jeden unterschiedlichen Fall muss die Regel einzeln implementiert werden, damit für jeden spezifischen Fall die richtige Ausgabe gegeben werden kann. Teilweise ändert sich dabei auch der Ablauf innerhalb der Regel, weil Java diese nicht so ausführen kann wie Prolog.

Prolog Queries	Java Queries
?- mann(lenian). **true.** ?- mann(anisa). **false.** ?- onkel(X,bob). X = caspar ; X = markus. ?- grossvater(X,Y). X = bob, Y = lenian ; X = arnd, Y = lenian ; X = caspar, Y = lorenz ; X = markus, Y = lorenz ; **false.** ?- bruder(X,markus) X = caspar ; **false.**	?- *mann(lenian).* true ?- *mann(anisa).* false ?- *onkel(X,bob).* X: caspar X: markus false ?- *grossvater(X,Y).* X: bob, Y: lenian X: arnd, Y: lenian X: caspar, Y: lorenz X: markus, Y: lorenz false ?- *bruder(X,markus).* X: caspar
Programmcode 4: Prolog Abfrage	*Programmcode 5: Java Abfrage*

Tabelle 3: Vergleich Abfrage Prolog-Java

2.5.2. Quantitativer Vergleich zwischen Prolog und Java

Trotz der eben beschriebenen Probleme bei der Implementation soll versucht werden, die Prolog- und Java-Version des Familienstammbaums zu vergleichen.

2.5.2.1. Dateigröße

Dafür kann man auf verschiedene Daten blicken. Zuallererst könnte man die Anzahl der Zeilen vergleichen. Die Prolog-Version ist 45 Zeilen lang und die Java-Version insgesamt 398 Zeilen (370 in der Klasse Zugriff und 28 in der Klasse Mann). Das gibt schon einmal einen (starken) Hinweis darauf, dass die Prolog-Implementierung kürzer ist. Jedoch stellt sich bei diesen Daten das Problem dar, dass mit Zeilen unterschiedlich umgegangen wird: Programmelemente werden unterschiedlich auf verschiedene Zeilen aufgeteilt und es werden unterschiedlich Leerzeilen angewandt. Auch ein einfacher Vergleich der Anzahl an Buchstaben wäre unpraktikabel, weil eine Variable sowohl „v“ als auch „diesisteinevariablemitsehrlangemnamen“ heißen kann. Aber man kann auf die insgesamte Dateigröße schauen: Insgesamt haben die Java-Dateien eine Größe von 16.392 Bytes (15.740+652 Bytes), während die Prolog-Datei nur eine Größe von 623 Bytes hat. Die Java-Dateien sind also insgesamt etwa 26-mal größer. Bei diesem kleinen Programm macht das zwar noch nicht den Unterschied, aber Prolog wurde ja für größere Aufgaben entwickelt.

2.5.2.2. Kompilationsdauer

Die zweite Möglichkeit für den Vergleich der zwei Programme ist die Kompilationsdauer. Das ist die Zeit, die Prolog beziehungsweise Java braucht, um meinen Code zu übersetzen, bevor ich ihn anwenden kann. Um diese Zeit abzulesen, müssen einfach beide Programme gestartet und die Zeit am Ende abgelesen werden. Prolog kompiliert immer von neu, bei Java habe ich dafür den Source-Ordner gelöscht.

Prolog gibt aus, dass es zum Kompilieren 0,02 Sekunden brauchte. Java spricht hingegen von einer Build-Dauer von 5 Sekunden und 708 Millisekunden. Beide Werte schwanken jedoch mit jeder erneuten Ausführung (auch, wenn zuvor bei Java der Source-Ordner gelöscht wurde). Diese Schwankungen liegen aber immer im Rahmen – es gibt also keine großen Ausreißer bei der Kompilationsdauer. Trotzdem lässt sich somit über das Java-Programm sagen, dass es deutlich länger zum Kompilieren braucht als das Prolog-Programm, um zu kompilieren.

2.5.2.3. Fazit des Vergleiches

Die Programmiersprache Prolog hat ihren Namen nicht umsonst von „Programming in Logic“ (Bramer, 2013, p. v) (wahlweise auch in deutscher oder französischer Übersetzung). Um logisch zu programmieren, ist Prolog einfach eine der besten Möglichkeiten. Natürlich kann man theoretisch die Logik aus Prolog auch nach Java oder C++ und Co überführen, aber das wäre aufwendig, da die Programmiersprachen einfach unterschiedliche Konzepte verfolgen. Und warum sollte man ein Programm von einer schnellen, dafür zugeschnittenen Programmiersprache in eine in diesem Kontext langsamere Programmiersprache, die dafür gar nicht gedacht ist, überführen? Stattdessen lohnt sich der Rückgriff auf Prolog hier viel mehr.

2.6. Logikrätsel

Ein weiterer Anwendungszweck, mit dem Prolog und die logische Programmierung allgemein häufig in Verbindung gebracht werden, sind die sogenannten Logikrätsel. Es sind unzählige Ausführungen über Prolog im Verbund mit Logikrätseln im Internet zu finden. Das macht auch Sinn: Prolog kann, wie beschrieben, Aussagen, die reiner Logik entstammen, auf ihren Wahrheitswert prüfen. Selbst wenn die Aussagen Lücken aufweisen wird Prolog versuchen, diese zu füllen, sodass diese (unvollständigen Fakten) wahr werden (Dibbs, kein Datum). Und genau das ist auch die Definition von Logikrätseln: Logikrätsel (auch Logicals) sind Rätsel, die nur durch logische Schlussfolgerungen lösbar sind (Rätselstunde, kein Datum). Natürlich ist die Lösung von Logikrätseln nicht der Hauptzweck von Prolog, aber es ist doch sehr anschaulich.

2.6.1. Einstein-Rätsel

Eines der wohl bekanntesten Logik-Rätsel ist das sogenannte „Einstein-Rätsel“, wahlweise auch „Zebrarätsel“ (Rätselstunde, kein Datum) genannt. Doch nicht nur der Name des Rätsels unterscheidet sich von Version zu Version, selbst der Inhalt ist immer anders. Der Grund für die Benennung als „Einstein-Rätsel“ ist das Gerücht, dass Albert Einstein dieses Rätsel erfunden haben soll und meinte, dass es nur 2 Prozent der Menschheit es lösen könnten. Prolog ist zwar kein Mensch, kann es aber auch lösen, durch rein logische Mechanismen.

Im Rätsel ist von einer Häuserzeile mit 5 Häusern die Rede. Alle Bewohner haben unterschiedliche Nationalitäten, Lieblingsspeisen und -getränke sowie unterschiedliche

Haustiere. Zudem ist die Farbe ihrer Häuser unterschiedlich. Gesucht wird nach der Person, die als Haustier einen Fisch hält. Der verwendete Wortlaut des Rätsels liefert 15 Fakten (Stettner, 2020):

1. Der Brite lebt im roten Haus.
2. Der Schwede hält sich einen Hund.
3. Der Däne trinkt gern Tee.
4. Das grüne Haus steht direkt links neben dem weißen Haus (vom Betrachter ausgesehen, der vor den Häusern steht).
5. Der Besitzer des grünen Hauses trinkt Kaffee.
6. Die Person, die eine Banane isst, hat einen Vogel.
7. Der Mann im mittleren Haus trinkt Milch.
8. Der Bewohner des gelben Hauses isst Schokolade.
9. Der Norweger lebt im ersten Haus (vom Betrachter aus gesehen ganz links).
10. Der Brot-Esser wohnt neben der Person mit der Katze.
11. Der Mann mit dem Pferd lebt neben der Person, die Schokolade isst.
12. Der Apfel-Esser trinkt gern Bier.
13. Der Norweger wohnt neben dem blauen Haus.
14. Der Deutsche isst gerne Kartoffeln.
15. Der Brot-Esser hat einen Nachbarn, der Wasser trinkt.

2.6.2. Prolog-Programm

Die Umsetzung des Programmes in Prolog ist stark inspiriert von einem YouTube-Video von „kwoxer“ (kwoxer, 2010). Ich habe sein Programm jedoch angepasst, weil im Video eine andere Version des Rätsels verwendet wurde.

Im Programmcode fällt folgender wiederkehrender Befehl auf: „member([rot,brite,_,_,_],X)“. Diese Codezeile ist nur dann wahr, wenn die Elemente innerhalb der member-Methode Teil der Liste sind. Das Objekt, welches Teil der Liste sein soll, wird in diesem Fall durch das „[rot,brite,_,_,_]“ dargestellt. Diese Liste ist an der Lösungstabelle orientiert: In den drei freien Bereichen würde nun das Lieblingsessen, das Lieblingsgetränk sowie das Haustier stehen. Die angesprochene Liste ist hierbei das zuvor definierte „X“ („X = [_,_,_,_,_]“). Die member-Methode erfüllt also nur true, wenn in der Liste „X“ an Position 1 die Farbe rot steht und an Position 2 die Nationalität Brite (van Noord, kein Datum). Die Verwendung dieser Methode macht insofern Sinn, dass Prolog andere Möglichkeiten berechnet, bis es eine gefunden hat, die alle Bedingungen wahr werden lässt – selbst, wenn diese unvollständig sind.

Die Ausgabe am Ende des Programmes ist folgendermaßen programmiert:

```
write(X),nl,
write('Der '),write(N),write(' hat einen Fisch als Haustier.'),nl.
```

Programmcode 6: Einstein-Rätsel: Ausgabezeile

Zunächst gibt Prolog also nochmal die gesamte Liste „X“ aus, welche im Verlauf des Programmes mehrere geworden sind. Dann soll die Nationalität N des Besitzers des Fisches ausgegeben werden. Für diese Nationalität N ist die letzten Member-Methode wichtig („member([_,N,_,_,fisch],X)“): Hier wird der Variable N der Besitz des Fisches zugeordnet. Die Ausgabe in der write-Zeile ist also praktisch diejenige Listen-Zeile, der der Fisch zugeordnet wurde. Aufgefallen sein mag vielleicht der Unterschied zwischen „N“ und „_“, welche beide für unbekannte Werte (Variablen) verwendet werden. Der

Grund für die Verwendung von „N“ statt „_“ ist dabei, dass ich das N ansprechen kann – im Gegensatz zum Unterstrich. Ansonsten funktionieren beide gleichermaßen als Variable. Aber da ich den Unterstrich nicht ansprechen will, muss ich ihm ja auch keinen ansprechbaren Namen geben.

2.6.3. Und wem gehört jetzt der Fisch? – Die Lösung

Nummer	1	2	3	4	5
Farbe	Gelb	Blau	Rot	Grün	Weiß
Nationalität	Norweger	Däne	Brite	Deutsch	Schwede
Speise	Schokolade	Brot	Banane	Kartoffeln	Apfel
Getränk	Wasser	Tee	Milch	Kaffee	Bier
Tier	Katze	Pferd	Vogel	**Fisch**	Hund

Tabelle 4: Einstein-Rätsel: Lösungstabelle

Anscheinend ist der Deutsche der Fischbesitzer. Das habe zumindest ich erschlossen. Die Frage: Hat Prolog es auch herausgefunden?

```
% y:/Benedikt/Logikprogrammierung/logikraetsel.pl compiled 0.00 sec, 6 clauses
?- run.
[[gelb,norweger,schokolade,wasser,katze],[blau,daene,brot,tee,pferd],[rot,brite,banane,milch,vogel],[gruen,deutsche,kartoffeln,kaffee,fisch],[weiss,schwede,apfel,bier,hund]]
Der deutsche hat einen Fisch als Haustier.
true
```

Programmcode 7: Einstein-Rätsel: Gesamte Ausgabe

```
Der deutsche hat einen Fisch als Haustier.
true
```

Programmcode 8: Einstein-Rätsel: Fischbesitzer-Ausgabe [Zoom]

Auch Prolog hat erkannt, dass der Besitzer des Fisches der Deutsche sein muss. In der Ausgabe ist das „deutsche“ jetzt noch klein geschrieben, weil direkt das Konstantensymbol ausgegeben wurde – und die sind eben klein geschrieben.
Einen interessanten Fakt will ich jedoch noch erwähnen: Man kann sie auf dem oberen Bild zwar schwer erkennen, aber die grüne Zeile zeigt die (bereits im Vergleich zwischen Java und Prolog, siehe dazu 2.5.2.2.) erwähnte Kompilationsdauer. Bei diesem Programm brauchte Prolog exakt 0,00 Sekunden, um die 6 Aussagen (Clauses) zu kompilieren. Die Kompilationsdauer war also so kurz, dass sie im nicht messbaren Bereich liegt. Es ist eher unwahrscheinlich, dass ein Mensch dieses Rätsel schneller gelöst bekommt.
Diese Geschwindigkeit zeigt erneut die Vorteile von Prolog: Es ist einfach für solche logischen Fragestellungen geschaffen. Und das kann ihm keiner streitbar machen.

3. Schluss

3.1. Ausblick – Was kann Prolog noch?

Natürlich kann Prolog noch viel mehr, als der kurze Einblick aus dieser Arbeit gegeben hat. Das trifft auch auf die Logikprogrammierung an sich zu. Beides zusammen, die Theorien, Erklärungen und Geschichten dahinter füllt 500-seitige Bücher. Und es bleibt trotzdem nur ein Auszug. Trotzdem will ich in diesem Abschnitt versuchen, ein paar wenige weitere Prolog-Techniken zu nennen.

Wichtiger Teil von Prolog ist die Prolog-Datenbank. In den Programmen wurde sie immer in der Quelldatei, also im Programmcode, bearbeitet. Tatsächlich ist es aber auch möglich, diese während der Laufzeit zu ergänzen. Dies geht über die Prädikate „assertz“ (für Fakten) und „asserta“ (für Regeln) (Bramer, 2013, pp. 110-111). Die Westhouse-Group betont diese Möglichkeit in ihrer Wertung von Prolog als relevante KI-Programmiersprache (Westhouse-Group, 2021).

Abgesehen von der Bearbeitung während der Laufzeit kann Prolog aber noch sehr viel mehr – und das bezieht sich nicht nur auf die vielen Optionen für Prolog-Bäume, wie etwa Suchalgorithmen (Fuchs, 1990). Grundsätzlich ist es möglich, dass Prolog zum Beispiel Informationen aus externen Dateien liest oder dort welche einfügt. Prolog kann auch Nutzereingaben lesen. Allgemein könnte man sagen, Prolog kann alles, was eine imperative Programmiersprache wie Java auch kann – obwohl es hauptsächlich deklarativ ist! Natürlich fällt diesem Fakt manchmal der deklarative Charakter von Prolog zum Opfer und es wird mehr imperativ. Interessant ist dementsprechend auch, dass Prolog trotz des rekursiven Charakters die Möglichkeit besitzt, Schleifen zu implementieren. Auch für Automaten ist Prolog verwendbar (Weisweber, 1997). Bei all der Datenlogik und der Arbeit mit Datenbanken würde man zwar denken, Prolog würde keine Grafikprogrammierung zulassen. Das ist jedoch falsch! Es gibt Versionen von Prolog, die die Entwicklung von User-Interfaces zulassen. So stellt schon Norbert E. Fuchs in seinem Werk „Kurs in logischer Programmierung“ ein Scroll-Menu mittels Prologs dar. Dabei muss man für die Grafikentwicklung nicht einmal ein spezielles Framework (wie etwa „JavaFX“ für Java) installieren. In diesem Beispiel verwendet er dafür das Prädikat „scroll_menu/4“ der Prolog-Implementation „LPA MacProlog“ (Fuchs, 1990, pp. 33-35). SWI-Prolog selbst besitzt jedoch keine eigenen Möglichkeiten für ein Grafik-Interface und wird diese auch nicht bekommen. Es verweist stattdessen auf verschiedene Externe Programmiersprachen, wie etwa das wiederum in Prolog entwickelte „XPCE“ für GUI-Entwicklung (SWI-Prolog, kein Datum). Trotzdem ist es eher unwahrscheinlich, dass Prolog für grafische Spielentwicklung genutzt wird – Text- und Controllerbasierte Spiele gibt es aber tatsächlich (Weisweber, 1997).

3.2. Allgemeines Fazit

Ja, künstliche Intelligenz und allgemein Automatisierung sind extrem wichtig. Sie sollte in der Informatik unser aller Herzprojekt sein. Ein großer Teil davon sind Logik und Logikprogrammierung. Diese Facharbeit hat die Wichtigkeit davon dargestellt. Denn es wäre unwirtschaftlich, ohne diese zu arbeiten: Wir müssten riesige Datenbanken erzeugen, und dafür riesige Verarbeitungsalgorithmen schreiben. Abgesehen von all dem Aufwand, wären diese große Serverfarmen auch nicht gerade gut für die Umwelt (Lobe, 2019). Zwar werden unsere Datenmengen durch die Verwendung von Logik nicht klein,

aber zumindest kleiner: Ich muss nicht mehr für jedes Teil jede Eigenschaft durchspeichern, stattdessen ordne ich einfach per Regeln diesen Dingen Eigenschaften zu. So sparen wir sehr viel Speicherplatz, und tun nebenbei auch noch was Gutes für die Umwelt (vergleiche für dieses Beispiel „1. Einleitung“ und „2.1.1. Formalisierung“). Das gibt der Logikprogrammierung definitiv ihre Daseinsberechtigung.
Prolog ist wohl die meistverwendete logische Programmiersprache, eine der bekanntesten Versionen von ihr ist SWI-Prolog. Wie wichtig diese Programmiersprache ist, sieht man an vielen Stellen. Zum einen bietet SWI-Prolog so täglich neue Downloads mit der neuesten GitHub-Version an. Das soll der schnellen Beseitigung von Bugs dienen (SWI-Prolog, 2023). Zum anderen arbeiten mittlerweile hunderte Entwickler in ihrer Freizeit an diesem Projekt mit (SWI-Prolog, kein Datum). SWI-Prolog ist aber auch das oberste Ergebnis bei einer Google-Suche nach Prolog und wird in der meisten (modernen) Fachliteratur genutzt. Aber das ist gar nicht der Punkt: Wer von Logikprogrammierung redet, redet von Prolog. Und wer von KI redet, redet unweigerlich auch über die Logikprogrammierung. Die Westhouse-Group (laut Selbstauskunft führende technologische Recruiting-Plattform) bezeichnet so Prolog als eine der einflussreichsten und effizientesten KI-Programmiersprachen – und das neben Python, Java und C++ (Westhouse-Group, 2021). In dieser Facharbeit habe ich zwei dieser Programmiersprachen für logische Prozesse miteinander verglichen. Das Ergebnis: Die Westhouse-Group hat wohl recht, Prolog ist Teil der effizientesten KI-Programmiersprachen, denn das Prolog-Programm lief bei gleichem Aufbau deutlich schneller als das Java-Programm und war zusätzlich dazu auch noch kompakter. Die Relevanz der Logik in der KI erkennen auch verschiedene Forschungseinrichtungen (Aldinger, 2021).
Die Wichtigkeit dieser Programmiersprache haben auch andere erkannt: Im „TIOBE Programming Community Index“, der die Beliebtheit einer Programmiersprache auf Basis der Suchanfragen nach ihr abbilden will, erreichte Prolog im Februar 2023 den Rang 42[9]. Damit gehört es zwar nicht zu den beliebtesten Programmiersprachen, aber schon zu den wichtigeren (TIOBE Programming Community Index, 2023). Und auf Basis all dessen, was Prolog kann, macht das sogar Sinn.

3.3. Persönliches Fazit

Zum Schluss der Facharbeit muss ich einräumen: Als ich erstmals auf die Themenidee Logik, Logikprogrammierung und Prolog gekommen bin, dachte ich mir „interessant, ein weiterer Programmierstil. Vergleichbar zur Rekursion und Iteration.“ Ich lag völlig falsch. Damit meine ich nicht nur, dass das gar nicht vergleichbar ist (hier würde man eher vom Vergleich zwischen deklarativen- und imperativen Paradigmen sprechen). Logik ist so viel mehr als einfach nur eine „weitere“ Programmierart. Unsere Geschichte ist so eng mit der Logik verbunden – und hier rede ich nicht nur von der wichtigen Logikprogrammierung. Selbst der moderne Computer findet seinen Ursprung in der Logik. Aus meiner Sicht hat das Steffen Hölldobler im ersten Kapitel seines Werkes „Logik und Logikprogrammierung“ sehr interessant dargestellt (Hölldobler, 2003). Aber allein dieser generelle Überblick ist so lang wie meine gesamte Facharbeit.

[9] TIOBE macht jedoch darauf aufmerksam, dass ab Listenplatz 20 der Index nur noch inoffiziell veröffentlicht wird, weil sie Programmiersprachen vergessen könnten. In diesem inoffiziellen Teil befinden – oder befanden – sich jedoch auch bekanntere Programmiersprachen, etwa Scratch und Rust (heute beide Top 20).

Nach dem tiefen Einblick in ein völlig neues Konzept, auf dem alle von uns in der Schule behandelten Programmierkonzepte aufbauen, das uns bisher aber trotzdem unbekannt geblieben ist, frage ich mich: Wo wären wir heute ohne die Logik, ohne die Formalisierung von Aristoteles, ohne die Kalkülbildung und Mechanisierung der Ägypter? Wären wir trotzdem so fortschrittlich, würde ich diesen Text trotzdem an einem Computer schreiben?
Ich finde, es gibt nur einen würdigen Abschluss für diese Ausführungen: Ein Zitat unseres aller liebsten Logikers und Halb-Vulkanier Spock aus „Star Trek“: „Logik ist der Beginn von Weisheit, Lieutenant, nicht ihr Ende.“ (Star Trek VI: Das unentdeckte Land, 1991). Und damit hat er so recht.

Literatur- und Quellenverzeichnis

Literaturverzeichnis

Akademie der Wissenschaften und der Literatur Mainz, kein Datum *Entwicklung von Webanwendungen.* [Online]
Available at: https://studiengang-digitale-methodik.pages.gitlab.rlp.net/modul-7/7a/pages/development/
[Zugriff am 07 Februar 2023].

Aldinger, C., 2021. *Künstliche Intelligenz: Wo bleibt die Logik? [Hochschule Harz].* [Online]
Available at: https://www.hs-harz.de/blog/ki-forschung-wo-bleibt-die-logik
[Zugriff am 08 Februar 2023].

Berkholz, C., 2019. *Kapitel 5: Logik-Programmierung [HU Berlin].* [Online]
Available at: https://www2.informatik.hu-berlin.de/logik/lehre/WS18-19/Logik/downloads/LogInf-Handout-LogProg.pdf
[Zugriff am 22 Januar 2023].

Bramer, M., 2013. *Logic Programming with Prolog.* 2. Hrsg. London: Springer-Verlag London.

Brass, S., 2013/2014. *Objektorientiere Programmierung - Kapitel 15: Überladene Methoden [Martin-Luther-Universität Halle-Wittenberg].* [Online]
Available at: http://users.informatik.uni-halle.de/~brass/oop13/jf_overl.pdf
[Zugriff am 04 Februar 2023].

Brooks, K., 2020. *Easy Prolog installation on Windows, Linux, macOS.* [Online]
Available at: https://kaibrooks.medium.com/easy-prolog-installation-on-windows-linux-macos-a5c5e4b3bc45
[Zugriff am 12 Februar 2023].

Dibbs, S., kein Datum *Kapitel 4: Prolog [Leseprobe EDV-Buchversand].* [Online]
Available at: https://www.edv-buchversand.de/productinfo.php?replace=false&cnt=productinfo&mode=2&type=2&id=or-322&index=2&nr=0&preload=false&page=1&view=fit&Toolbar=1&pagemode=none
[Zugriff am 05 Februar 2023].

Fuchs, N. E., 1990. *Kurs in Logischer Programmierung (Springers Angewandte Informatik).* 1. Hrsg. Wien, New York: Springer-Verlag Wien New York.

Fuchs, N. E., 2023. *Prolog Tutorial [Institut für Informatik Universität Zürich].* [Online]
Available at: https://files.ifi.uzh.ch/rerg/arvo/courses/logische_programmierung/ws03/documents/Prolog_Tutorial.pdf
[Zugriff am 23 Januar 2023].

Hölldobler, S., 2003. *Logik und Logikprogrammierung.* 3. Hrsg. Heidelberg: Synchron Wissenschaftsverlag der Autoren/Synchron Publishers.

IONOS, 2021. *Imperative Programmierung: Das älteste Programmierparadigma im Überblick.* [Online]
Available at: https://www.ionos.de/digitalguide/websites/web-entwicklung/imperative-programmierung/
[Zugriff am 07 Februar 2023].

kwoxer, 2010. *Einstein Rätsel - (Prolog 2-007) [YouTube].* [Online]
Available at: https://www.youtube.com/watch?v=WRolU_PoYgw
[Zugriff am 06 Februar 2023].

Lobe, A., 2019. *Clouds voller CO2 [Süddeutsche Zeitung].* [Online]
Available at: https://www.sueddeutsche.de/kultur/internet-co2-fussabdruck-1.4628731
[Zugriff am 06 Februar 2023].

Plasse, W., kein Datum *Weltveränderer Aristoteles [GEOlino].* [Online]
Available at: https://www.geo.de/geolino/mensch/2755-rtkl-weltveraenderer-aristoteles
[Zugriff am 11 Februar 2023].

Rätselstunde, kein Datum *Logical [Rätselstunde].* [Online]
Available at: https://www.raetselstunde.de/logical.html
[Zugriff am 05 Februar 2023].

Star Trek VI: Das unentdeckte Land. 1991. [Film] Regie: Nichlosa Meyer. USA: Paramount Pictures.

Stettner, A., 2020. *Albert Einstein erfand ein Rätsel, das fast niemand lösen kann - und Sie? [Merkur].* [Online]
Available at: https://www.merkur.de/leben/karriere/albert-einstein-erfand-raetsel-fast-niemand-loesen-kann-zr-10018013.html
[Zugriff am 05 Februar 2023].

SWI-Prolog, 2023. *Download daily builds for Windows [SWI-Prolog].* [Online]
Available at: https://www.swi-prolog.org/download/daily/bin/
[Zugriff am 12 Februar 2023].

SWI-Prolog, kein Datum *Graphical applications [SWI-Prolog Wiki].* [Online]
Available at: https://www.swi-prolog.org/Graphics.html
[Zugriff am 11 Februar 2023].

SWI-Prolog, kein Datum *SWI-Prolog Main development repository [GitHub].* [Online]
Available at: https://github.com/SWI-Prolog/swipl-devel
[Zugriff am 22 Januar 2023].

TIOBE Programming Community Index, 2023. *TIOBE Index for February 2023.* [Online]
Available at: https://www.tiobe.com/tiobe-index/
[Zugriff am 06 Februar 2023].

van Noord, G., kein Datum *Predicate member/2 [The SWI-Prolog library].* [Online]
Available at: https://www.swi-prolog.org/pldoc/man?predicate=member/2
[Zugriff am 06 Februar 2023].

Weisweber, W., 1997. *Prolog - Logische Programmierung in der Praxis.* 1. Hrsg. Bonn, Albany: International Thomson Publishing.

Westhouse-Group, 2021. *KI Programmiersprachen: Python, C++, Java, LISP, Prolog [Westhouse-Group].* [Online]
Available at: https://www.westhouse-group.com/ki-programmiersprachen-python-c-java-lisp-prolog/
[Zugriff am 06 Februar 2023].

Wielemaker, J., kein Datum *Jan Wielemaker: Info [LinkedIn].* [Online]
Available at: https://nl.linkedin.com/in/jan-wielemaker-6a270a1b3?original_referer=
[Zugriff am 23 Januar 2023].

Des Weiteren halfen bei der Erstellung dieser Facharbeit diverse Unterseiten der Homepage von SWI-Prolog (swi-prolog.org), die teilweise in Vergessenheit geraten sein können. In dem Fall wurden sie dann aber nicht zitiert. Stack Overflow hat auch einige (ebenfalls nicht zitierte, jedoch informierende) Informationen beigetragen.

Verzeichnis der verwendeten Medien

Verzeichnis Programmcodes

Verzeichnis Grafiken

Verzeichnis Formeln

Verzeichnis Tabellen

Webverweis

In dieser Facharbeit habe ich viele Materialien zu Rate gezogen, die sicherlich zu den Ausführungen dazu gehören sollten. Doch sie hier anzuheften, würde den Rahmen sprengen. Stattdessen liegen sie in verschiedenen Clouds, wo Sie im folgenden Abschnitt Zugriff drauf erhalten.

GitHub

Im Rahmen dieser Facharbeit sind mehrere (kleinere) Softwareprogramme entstanden. Da sie, wie im quantitativen Vergleich zwischen Java und Prolog (vergleiche 2.5.2.) dargestellt, teilweise mehrere hundert Zeilen lang sind, können Sie auf diese über GitHub zugreifen.

GitHub ist wohl das größte netzbasierte Versionsverwaltungssystem für Software-Entwicklungsprozesse – oder einfach nur die größte Cloud für Entwickler, die ihren Code online speichern wollen. So wird auch SWI-Prolog auf GitHub entwickelt (SWI-Prolog, kein Datum). Außerdem ist eine GitHub-Erweiterung in eigentlich jeder größeren IDE vorhanden. IntelliJ oder PyCharm (JetBrains IDE für Python, verwendet in der Schule) besitzen so zum Beispiel eine GitHub-Erweiterung.

Zugriff auf mein Repository zur Facharbeit erhalten Sie, indem Sie dem Link folgen oder den QR-Code mit GitHub-Logo scannen. Die Programme sind dort nach der jeweiligen Kapitelnummer in dieser Facharbeit sortiert.

Benedikt-Heffels/Facharbeit: Facharbeit: Logik und Logikprogrammierung mit Prolog (github.com)[10]

Microsoft OneDrive

Die Vorgaben zur Facharbeit besagen, dass wir Webquellen sichern sollen. Da ein erneutes Hochladen der Abhandlungen aus dem Internet auf GitHub höchstwahrscheinlich illegal wäre und sie mit einer Länge von jeweils etwa 25 Seiten nicht hier angehangen werden könnten, habe ich sie auf den Microsoft OneDrive Server der Schule hochgeladen.

Auch hier erhalten Sie Zugriff, indem Sie dem Link folgen oder den QR-Code mit OneDrive-Logo scannen.

benhef @ Logikprogrammierung mit Prolog

[10] Text-Link: https://github.com/Benedikt-Heffels/Facharbeit

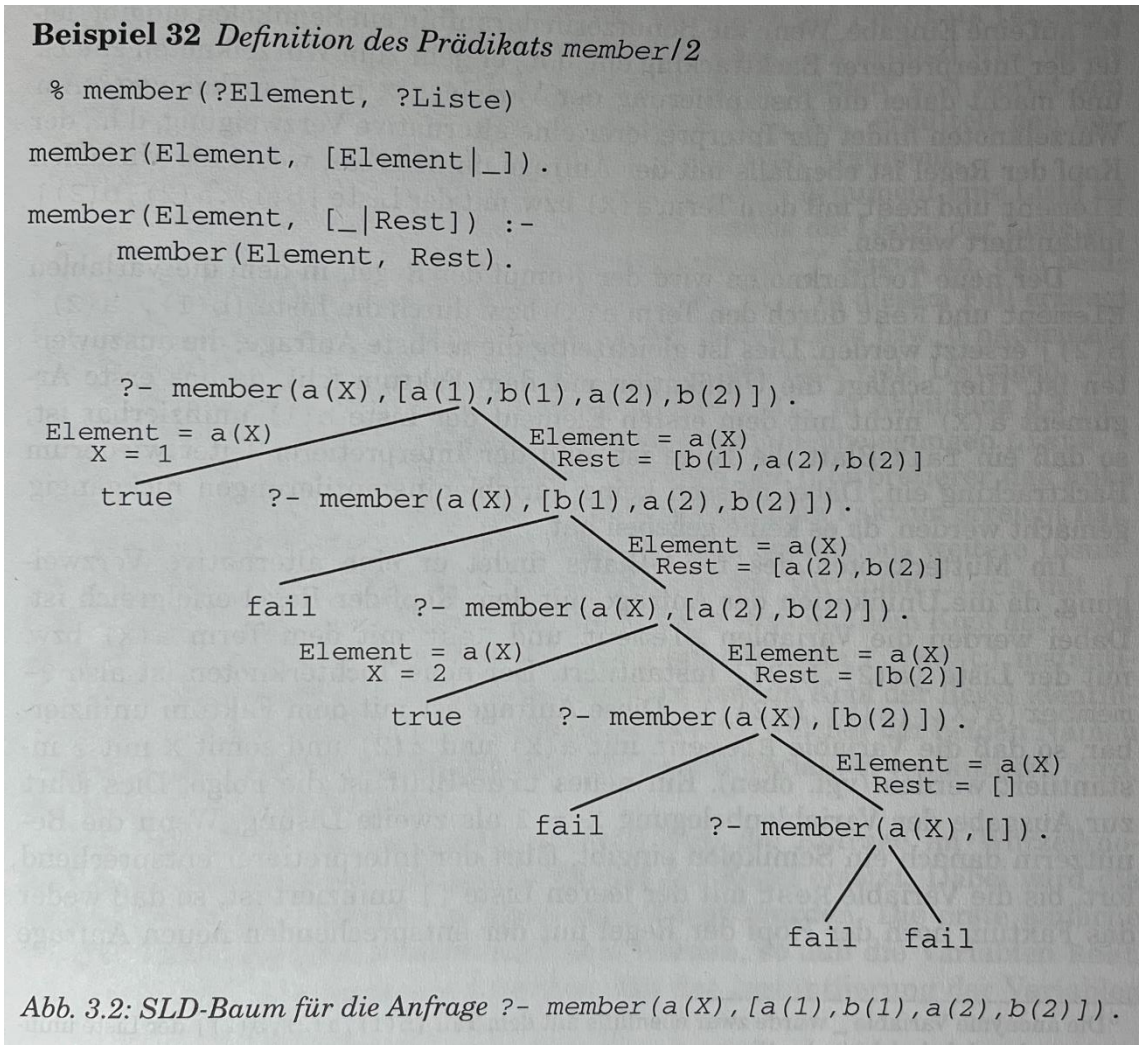
QR-Codes zu GitHub und Microsoft OneDrive.[11]

[11] Quellenverweis:
Logo GitHub: https://th.bing.com/th/id/OIP.wlJw2F4i0dPMZuGQD72loQAAAA?pid=Img-Det&rs=1
Logo MS OneDrive: http://dwpoppymusic.com/uploading-large-files-from-ios-to-onedrive-for-sharing/
QR-Codes erstellt mit QRCode Monkey: https://www.qrcode-monkey.com/de/

Anhang

A1: Backtracking und Bäume

In Kapitel 2.4.3. zum Thema Backtracking ist die Rede von Backtracking und Bäumen. Hier ist der Verweis auf einen Beispiel-Baum sinnvoll.



Grafik 3: Backtracking an beispielhaften SLD-Baum (Weisweber, 1997, p. 63)

Das Bild zeigt den SLD-Baum zur Definition des Prädikates „member/2“. Die 2 steht dabei dafür, dass dem Prädikat zwei Variablen übergeben werden. Ein SLD-Baum wird bei jeder Anfrage in Prolog auf Basis der Anfrage konstruiert. SLD steht dabei für „selected literal with definite clauses“. Die Anfrage an den SLD-Baum ist der Wurzelknoten. Im Anschluss gibt es so viele Tochterknoten, wie es Klauseln zur Definition des Prädikates gibt. Das sieht man auch in der Grafik.

Der im Unterricht angesprochenen Baum-Theorie zu Folge folgt man zunächst immer dem linken Ast. Bei diesem Baum ist der wahr, denn die gesuchte Variable „X“ von „a“ stimmt mit der Zahl 1 des Elementes „a“ überein. Dies wird auch ausgegeben. Leite ich nun wie im Kapitel zum Backtracking beschrieben jenes Backtracking ein, würde Prolog alle Verweise auf diesen Ast löschen und zurück zum letzten Mutterknoten gehen, bei dem man sich anders entscheiden kann. An diesem wird dann auch der andere Ast gewählt, und es wird das nächste Element der Liste verglichen. Da das Element „b(1)“ aber kein „a“ ist, ist dies ein fail-Ast. Im Falle von false (fail-Ast) leitet Prolog automatisch

Backtracking ein und der oben beschriebene Prozess wiederholt sich. Das nächste Objekt ist wieder übereinstimmend und wird erneut ausgegeben. Leite ich nun erneut manuell Backtracking ein, besteht die Liste nur noch aus „[b(2)]“. Das führt nur zu fail-Ästen und Prolog gibt endgültig „false“ aus (Weisweber, 1997, pp. 60-63).

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst habe und keine als die im Literaturverzeichnis angegebenen Hilfsmittel verwendet habe.
Insbesondere versichere ich, dass ich alle wörtlich und sinngemäßen Übernahmen aus anderen Werken (zum Beispiel auch Facharbeiten anderer Schulen) als solche kenntlich gemacht habe.

Heinsberg, den 15.02.2023