Max Rohde, winf2848, Informatik-Seminar SS2007 bei Prof. Dr. Sebastian Iwanowski

Fachhochschule Wedel

Eignung logischer Programmier-sprachen für Spiele-KI-Entwicklung am Beispiel Prolog

Inhaltsverzeichnis

Einleitung

Seit in den 50er Jahren die ersten höheren Programmiersprachen (FORTRAN, COBOL, LISP) entwickelt wurden, hat sich Zahl der Programmiersprachen geradezu exponentiell vergrößert. Trotz der vielen unterschiedlichen Sprachen, gibt es nur recht wenige bedeutende Programmiersprachen-Konzepte. Diese sind neben der prozedualen, funktionalen und Skriptsprachen die logischen Programmiersprachen. Gegenstand dieser Ausarbeitung soll die Untersuchung sein, ob sich die Konzepte logischer Programmiersprachen dazu eigenen, Künstliche Intelligenz in Computerspielen zu implementieren.

Dies geschieht am Beispiel der Sprache Prolog, die in den 70er Jahren entwickelt wurde und sich seitdem besonders im Bereich der Künstlichen Intelligenz großer Beliebtheit erfreut, die sich im Großen und Ganzen aber nie richtig durchsetzen konnte.

Diese Ausarbeitung bezieht sich dabei auf den logischen Kern von Prolog. D. h., auf die Eigenschaften die Prolog von funktionalen oder prozedualen Programmiersprachen unterscheiden.

Deshalb wird z. B. in der Fallstudie, der Implementierung einer Künstlichen Intelligenz für das Spiel Vier Gewinnt, für alle nicht-KI Aspekte ein Java-Programm verwendet, da sich die Spielsteuerung nicht ohne Verletzung logischer Prinzipien in Prolog umsetzen lässt.

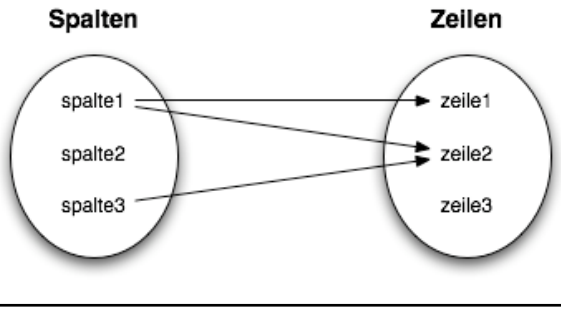
Grundlegende Eigenschaften von Prolog

Folgend soll ein kurz erläutert werden, welches die grundlegenden Eigenschaften der Sprache Prolog sind, sowie welche Abweichungen und Übereinstimmungen mit anderen Programmiersprachen bestehen.

Relational

Eine n-stellige Relation ist definiert als eine Teilmenge des kartesischen Produkts von n Mengen. D. h. jedes Element einer Menge kann mit einer beliebigen Kombination von Elementen der anderen Mengen in Verbindung stehen.

$$R \subseteq M_1 \times ... \times M_i,\ i \geq 1$$



Relation [B1]

In diesem Sinne ist die Relation sehr eng verwandt mit dem Begriff des Prädikats aus der Logik. Dieses ist ebenfalls als kartesisches Produkt definiert mit der Einschränkung, dass die Elemente der Menge Elemente des Diskursuniversums sein müssen und ihnen ein Wahrheitswert zugeordnet werden kann.

Die Definition von Relationen in Prolog ist sehr einfach. Abfragen über den definierten Relationen können dem System einfach gestellt werden. Diese können dabei natürlich mehr als ein Ergebnis zurück liefern.

```
feld(spalte1, zeile1).
feld(spalte1, zeile2).
feld(spalte3, zeile2).
```

Relation in Prolog

```
?-feld(spalte1, X).
Soln: 1
X = zeile1
;
Soln: 2
X = zeile2
;
No More Solutions.
```

Anfrage mit mehreren Ergebnissen

Damit ist Prolog in gewisser Weise ein relationales Datenbanksystem an das Abfragen gestellt werden können. An andere Datenbanksysteme können mittels SQL bzw. genauer DQL (Data Query Language) Anfragen gestellt werden.

```
SELECT * FROM FELDER WHERE spalte='1'

spalte    zeile
------    ------
1         1
1         2
```

Beispiel SQL-Anfrage

Funktionen in imperativen Programmiersprachen sowie in funktionalen Programmiersprachen liefern dagegen immer genau ein Ergebnis zurück[1]. Folgend ein Programm in der funktionalen Programmiersprache Haskell, das versucht, die Relation feld zu definieren:

```
feld :: Spalte -> Zeile
feld 1 = 1
feld 1 = 2
feld 3 = 2
```

Haskell Programm

```
Ok, modules loaded: Feld.
*Feld> feld 1
1
*Feld>
```

Haskell Session

Wie zu erkennen ist, wird hier trotz des Eintrages „`feld 1 = 2`“, nur das Ergebnis „`1`“ zurückgeliefert.

Logisch

Eine Logische Programmiersprache (engl. logic programming language) beschreibt im weitesten Sinne eine Programmiersprache, die sich der mathematischen Logik bedient. Da sich die meisten Programmiersprachen zumindest in ihren Ausdrücken der mathematischen Logik bedienen, bedarf es jedoch einer genaueren Eingrenzung.

```
boolean regen=true;
boolean keinRegenschirm=false;
nass = regen && keinRegenschirm;
```

Mathematische Logik in Java

Eine Logische Programmiersprache kann daher im Gegensatz zu einer imparativen Programmiersprache dadurch genauer spezifiziert werden, dass sie keine Folge von Anwei-

[1] Wie es der Name Funktion laut mathematischer Definition bestimmt.

sungen, sondern eine Menge von Axiomen[2] enthält. Des Weiteren sollten Mechanismen von der Programmiersprache bereitgestellt werden, die den Wahrheitswert von Aussagen aus den Axiomen berechnen können[3].

Ein Prolog-Programm besteht in der Tat nur aus einer Sammlung von Klauseln. Die Dynamik im Programm wird durch Abfragen über der Klausel-Menge erreicht. Näheres dazu im Abschnitt „Von Aussagen zur Prolog-Syntax“.

Deklarativ
Die Grundidee einer deklarativen Programmiersprache ist, die Komplexität der Aufgabe Programmieren für den Entwickler dadurch zu verringern, dass nicht mehr genau spezifiziert werden muss, wie ein Problem gelöst werden soll, sondern allein was gelöst werden soll.

In der imperativen oder prozedualen Programmierung[4] wird genau spezifiziert, in welcher Reihenfolge bestimmte Operationen ausgeführt werden sollen, um zu einem Algorithmus zu gelangen, der ein bestimmtes Problem löst. Obwohl diese Programmiersprachen eine genaue Spezifikation der Algorithmen erfordern, enthalten auch sie deklarative Elemente, beispielsweise bei der Auswertung arithmetischer Ausdrücke, bei denen nicht mehr festgelegt werden muss, wie die Rechenoperationen auf Maschinenebene ausgeführt werden sollen[5].

```
int i=5;
int j=3;
int k=i*j+j;
```

Deklarative Elemente in C

```
sortiert(Ausgangsliste, Sortiert):-
permutation(Ausgangsliste, Sor-
tiert), sortiert(Sortiert).
```

Deklaratives Programm in Prolog

---

[2] Ein Axiom ist eine Tatsache, deren Gültigkeit als gegeben angesehen wird und nicht weiter in Frage gestellt wird. Axiome stellen die "Grundlage der Mathematik" dar, da immer von ihnen ausgehend ein Beweis erbracht werden muss (s. "Mathematische Grundlagen der Informatik", Christoph Meinel, Martin Mundhenk, 2. Aufl. S. 61 f.).

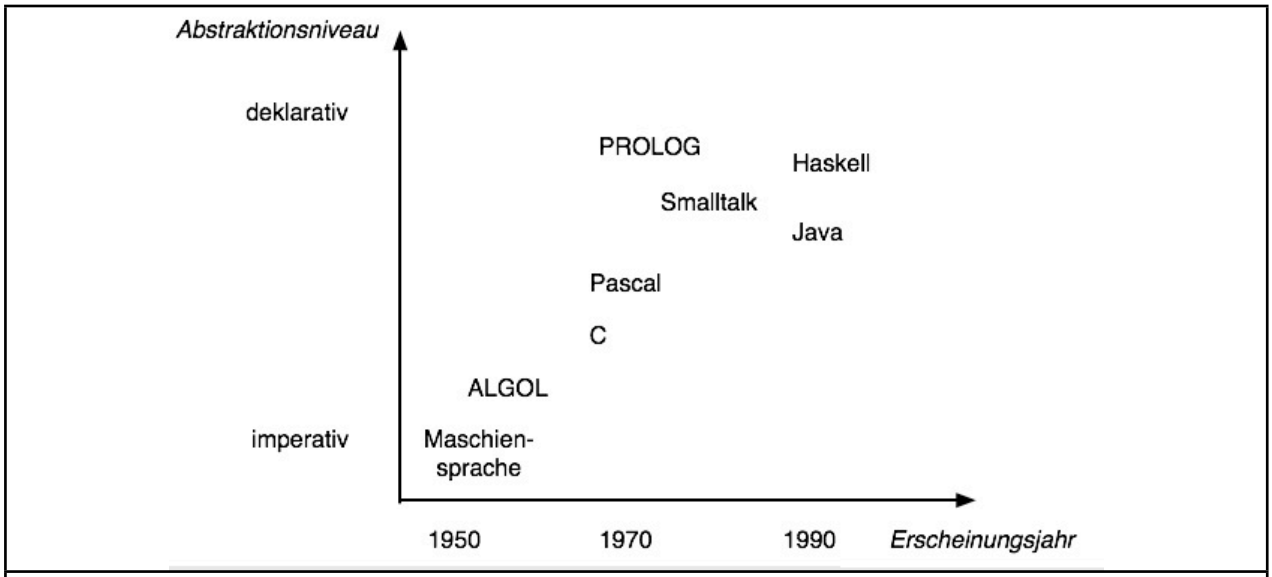
[3] Der Programmierer spezifiziert damit nur das Problem und nicht, wie es gelöst werden soll. Diese Aufgabe fällt der Programmierumgebung zu. In [L2] heißt es dazu passend:

„[...] the programmer supplies a formal specification of the problem to be solved and leaves the computer to decide how to solve it.“

[4] Bekannte Beispiele für derartige Programmiersprachen sind Java, Pascal und C.

[5] Eine in diesem Sinne am wenigsten deklarative Programmiersprache ist Assembler. Dennoch kann man auch hier gewisse deklarative Elemente Elemente feststellen. So muss z. B. nicht festgelegt werden wie genau die Addition oder Shift-Operationen auf Bit-Ebene ausgeführt werden sollen. Damit zeigt sich, dass eine Eingrenzung „deklarativ“ oder „nicht deklarativ“ nicht vorgenommen werden kann. Vielmehr kann nur festgestellt werden, ob eine Programmiersprache mehr deklarative Elemente enthält als eine andere.

In Prolog ist in der Tat eine mehr deklarative Programmierweise möglich als in den klassischen imperativen Programmiersprachen. Dies liegt daran, dass in Prolog grundlegende Algorithmen wie logisches Schließen, Unifikation und Backtracking integriert sind. Genauer beschrieben werden diese Prinzipien im Abschnitt „Wie Prolog arbeitet“.



Entwicklung deklarativer Programmiersprachen [B2]

Die Begriffe logisch und deklarativ werden oft synonym verwendet. Eine deklarativ/logische Programmiersprache lässt sich von einer imparativen vor allem durch zwei Eigenschaften abgrenzen: Sie enthält keine Folge von Anweisungen, sondern eine Menge von Axiomen und Schlussregeln und sie definiert keine Handlungsanweisungen, sondern definiert Beziehungen zwischen Objekten.

Von Aussagen zur Prolog-Syntax

Prolog ist eng verbunden mit der Prädikatenlogik erster Stufe (engl. first order logic)[6]. Ein Prolog-Programm besteht dabei ausschließlich aus Horn-Klauseln. Folgend soll ein Überblick über die wichtigsten syntaktischen Konstrukte der Sprache Prolog vermittelt werden. Die folgenden Erläuterungen beziehen sich auf Prolog nach ISO-Standard ISO/IEC 13211-1 [L6][7].

Übersicht

Zunächst sei eine Übersicht über die Syntax der Sprache Prolog in Backus-Naur Form (BNF) gegeben[8].

```
Program ::= Clause | Clause Program
Query ::= Term

Clause ::= Fact | Rule
Fact ::= Term .
Rule ::= Term :- Terms .

Terms ::= Term | Term , Terms
Term ::= Number | Variable | AtomName | AtomName(Terms)
               | [] | [Terms] | [Terms | Term]
               | not(Term) | once(Term) | ...
Number ::= Digit | Digit Number
Digit ::= 0 | ... | 9
AtomName ::= LowerCaseChar NameChars
Variable ::= UpperCaseChar NameChars
NameChars ::= NameChar | NameChar NameChars
NameChar ::= LowerCaseChar | UpperCaseChar | Digit | _
LowerCaseChar ::= a | ... | z
UpperCaseChar ::= A | ... | Z
```

Prolog Syntax in BNF

Obwohl diese Syntax sehr kompakt ist, ist sie eine nahezu vollständige Abbildung der Sprache Prolog. Dies liegt unter anderem daran, dass die Kontrollstrukturen aus imperativen Programmiersprachen (`if`, `while`, `for`, `...`) entfallen.

Ein Prolog Programm besteht ausschließlich aus Klauseln (s. „`Clause`“ in BNF). Eine Klausel ist definiert als eine Disjunktion von Literalten:

$$\forall x_1 ... \forall x_n : L_1 \vee ... \vee L_m \text{ mit } n \geq 1, m \geq 1 \text{ und } x_{1..n} \text{ als einzige Variablen in } L_{1..n}$$

Horn-Klauseln sind eine Untermenge der Klauseln, die der zusätzlichen Bedingung genügen, dass nur eins der Literalte $L_{1..n}$ positiv, d.h. ohne vorangehendes ¬ ist.

Bei Klauseln werden oft die voranstehenden Allquantoren weggelassen und implizit als gegeben angenommen.

---

[6] Es sind damit keine Aussagen über Aussagen möglich.

[7] Eine Übersicht über den Standard erhält man auch auf [L5]. Hier auch ohne Schweizer Franken.

[8] Die hier vorgestellte Grammatik ist eine leicht modifizierte Version der unter [L4] vorgestellten.

Atome
Ein Atom in Prolog ist eine unteilbare[9] und unveränderbare Einheit, die ein irgendwie geartetes Objekte repräsentieren. Folgend einige Beispiele für Sachverhalte in natürlicher Sprache und wie diese in Prolog-Atome umgesetzt werden.

natürliche Sprache	**Prolog**
Haus	`haus`
Eine Spalte 1	`spalte1`
Spalte 1 und Spalte 2 sind benachbart-	`benachbart_spalte1_spalte2`
fünfundzwanzig	`25`
sieben Komma zwei	`7.2`
Der Text ‚ABC und D‘	`‘ABC und D‘`
Atome in Prolog	

Wie der Grammatik zu entnehmen ist, muss ein Atom immer mit einem Kleinbuchstaben beginnen. Es können beliebig komplexe Sachverhalte durch Atome ausgedrückt werden wie im Beispiel „`benachbart_spalte1_spalte2`“. Nur liegt diese Information dann für das Prolog System unteilbar vor. D.h. es kann nur eine Abfrage „Sind Spalte 1 und 2 benachbart?“ beantwortet werden, nicht jedoch eine wie „Zu welchen Spalten ist Spalte 1 benachbart?“, da Prolog keine Informationen über einzelne Teile des Atoms sowie den Beziehungen zwischen diesen vorliegen.

Das prädikatenlogische Äquivalent zu Atomen sind Konstanten.

Variablen
Variablen sind in Prolog zunächst Platzhalter für beliebig geartete Objekte und Strukturen. Im Unterschied zu den Atomen müssen Variablennamen immer mit einem Großbuchstaben beginnen. Oft verwendet werden einzelne Großbuchstaben wie „`X`“, „`Y`“, „`Z`“. Aber auch aussagekräftigere Namen wie „`Zeile`“ für eine beliebige Zeile oder „`Feld`“ für ein beliebiges Feld sind möglich. Wie erwähnt können die Variablen beliebige Werte annehmen. D.h., der Variable „`Zeile`“ kann problemlos ein Liste mit Gleitkommawerten zugeordnet werden, auch wenn eigentlich eine einzelne Zahl aus dem Wertebereich Integer vorgesehen ist.

Eine besondere Variable ist der Platzhalter „_“, der für einen beliebigen Wert steht, der nach der Zuweisung sofort wieder verworfen wird.

Variablen entsprechen den Variablen in der Prädikatenlogik.

Fakten
Ein Fakt beschreibt ähnlich wie ein Atom ein Sachverhalt. Der Unterschied zu dem Atomen liegt darin, dass durch Fakten Eigenschaften von oder Beziehungen zwischen Objekten ausgedrückt werden können. Folgend einige Beispiele für Sachverhalte in natürlicher Sprache und wie diese mit Prolog-Fakten ausgedrückt werden.

---

[9] Daher auch der der Name Atom, der vom altgriechischen Wort átomos für unteilbar abgeleitet ist.

natürliche Sprache	**Prolog**
Es gibt ein Haus.	`haus.`
Es gibt eine Spalte spalte1.	`spalte(spalte1).`
Spalte 1 und Spalte 2 sind benachbart	`benachbart(spalte1, spalte2).`
1 ist Vorgänger von 2.	`kommt_vor(1, 2).`
Alles ist gut.	`gut(Alles).` oder `gut(_).`
Jeder ist sich selbst am nächsten.	`am_naechsten(X, X).`
Fakten in Prolog	

Durch „`spalte(spalte1).`“ wird damit die Aussage über das Objekt `spalte1` getroffen, dass es sich hierbei um eine Spalte handelt. Beziehungen werden mit Fakten der Stelligkeit größer 1 ausgedrückt. Dadurch können dem Prolog-System Relationen mitgeteilt werden.

Damit sind die Entsprechung für Prolog-Fakten in der Prädikatenlogik die Prädikate. Tritt ein solches Prädikat in der Form „`Term .`“ auf, handelt es sich um eine Aussage, die vom System als wahr angenommen wird.

Regeln
Neben den Fakten, die die axiomatische Grundlage des Systems bilden, können Regeln aufgestellt werden, die die Grundlage für das logische Schließen bilden. Folgend soll anhand eines Beispiels dargestellt werden, wie sich eine Regel in natürlicher Sprache über die Prädikatenlogik in eine Prolog-Regel überführen lässt.

> Wenn X eine Spalte ist und Y eine Zeile, dann bilden sie zusammen ein Feld.

Lässt sich zunächst in Prädikatenlogik erster Ordnung übersetzten.

$$spalte(x) \wedge zeile(y) \Rightarrow feld(x, y)$$

Wie bereits erwähnt, kann Prolog ausschließlich mit Aussagen arbeiten, die als Horn-Klausel vorliegen. Die aufgestellte Formel ist eine Form für Horn-Klauseln:

$$[L_1 \wedge ... \wedge L_m] \Rightarrow S \text{ mit } m \geq 1 \text{ und } x_{1..n} \text{ als einzige Variablen in } L_{1..n},$$

Durch Umformungen kann gezeigt werden, dass die angegebene Formel als Disjunktion von Literalen, von denen nur eines positiv ist, dargestellt werden kann. Die Formel kann daher einfach in eine Prolog-Klausel überführt werden. Der Operator $\wedge$ wird dabei als , geschrieben und der Implikationspfeil $\Leftarrow$ als :-.

```
feld(X, Y) :- spalte(X), zeile(Y).
```

Rekursion
Da es in Prolog keinerlei Kontrollstrukturen gibt, die beispielsweise Wiederholungen erlauben würden, ist die Rekursion wie bei einer funktionalen Programmiersprache von essen-

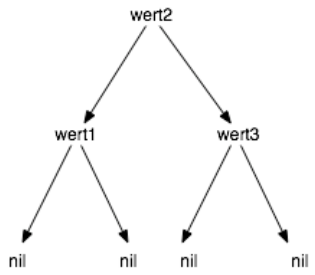
tieller Bedeutung. Dabei können sowohl rekursive Strukturen, als auch die rekursive Auswertung verwendet werden.

Strukturen

Eine rekursive Struktur ist dadurch definiert, dass sie Elemente enthalten kann, die ihr selbst ähneln. Einfache Beispiele sind lineare Liste und binäre Bäume. In Prolog kann ein Prädikat andere Prädikate beinhalten durch die Grammatik-Regel: `AtomName(Terms).`

```
knoten(
     knoten(nil, wert1, nil),
     wert2,
     knoten(nil, wert3, nil)).
```

Binärer Baum in Prolog



Binärer Baum als Graph [B3]

Auswertung

Um die Erfüllbarkeit einer Regel zu ermitteln, kann auf die gleiche Regel Bezug genommen werden. Bei den Fakten ist das Beispiel „`benachbart(spalte1, spalte2).`“ aufgeführt. Obschon es aus dem Sinnzusammenhang zu erschließen ist, erkennt Prolog natürlich nicht, dass die Relation `benachbart/2` symmetrisch ist. Damit aus „`benachbart(spalte1, spalte2).`“ automatisch folgt, dass „`benachbart(spalte1, spalte2).`“ kann folgende Regel aufgestellt werden:[10]

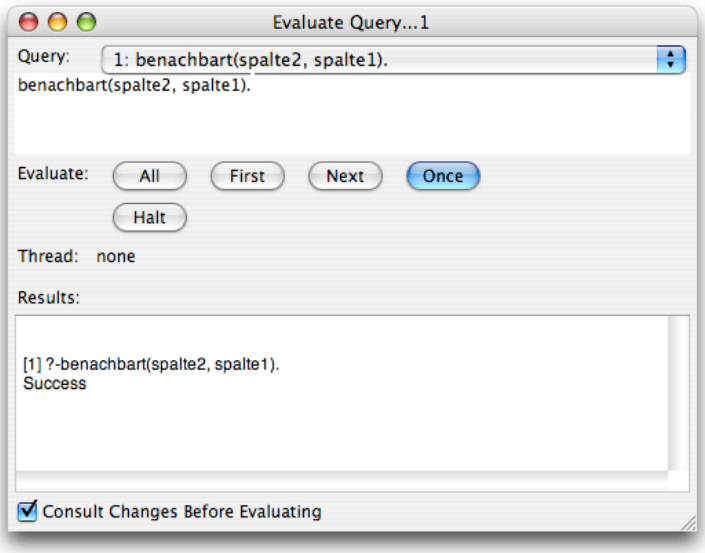
```
benachbart(X, Y) :- benachbart(Y, X).
```

Abfragen

Die Dynamik in einem Prolog-System wird dadurch erzeugt, dass Abfragen (in BNF: `Query`) an das System gestellt werden können. Diese werden mit Erfolg („success“) oder Scheitern („fail“) beantwortet. Folgend ein Beispiel einer Anfrage an das Prolog System XGP [I1][11].

[10] Es sei schon an dieser Stelle darauf hingewiesen, dass die so entstandene Regel zwar deklarativ korrekt ist, sie jedoch kein sehr gutes Prolog-Programm ergibt. Ausführlich wird auf diesen Umstand in Abschnitt „Vor- und Nachteile für die Spieleentwicklung“ eingegangen.

[11] Meist können Abfragen an das System auf zwei Arten gestellt werden: Dass nur die erste gefunden Lösung ausgegeben wird (bei XGP Button „once“) oder alle möglichen Lösungen (bei XGP Button „all“).



Anfrage an Prolog System XGP [B4]

Beinhaltet die Abfrage Variablen, so werden neben der Information, ob sich die Klausel erfüllen lässt, oder nicht, alle möglichen Werte der Variable(n) ausgegeben. Dies würde für „`benachbart(spalte2, X)`“ beispielsweise ergeben:

```
Soln: 1
X = spalte1
;

Soln: 2
X = spalte3
;
No More Solutions.
```

Wie Prolog arbeitet

Nachdem die grundlegenden syntaktischen Konzepte der Sprache Prolog vorgestellt wurden, soll nun kurz erläutert werden, welche zusätzlichen Funktionen ein Prolog-System bieten muss.

Jedes Prolog-System kann aus einer gegebenen Fakten- und Regelmenge mithilfe eines Resolutionsverfahrens logische Schlüsse ziehen.

Dafür wird das Verfahren, Beweis durch Widerspruch angewendet. D. h. für Prolog das ein bestimmter Satz ϕ genau dann wahr sein muss, wenn aus der Vereinigung Menge der Klauseln K, aus denen das Prolog-Programm besteht, und der Menge, die den Satz ¬ϕ beinhaltet, kein Widerspruch (leere Klauselmenge) hergeleitet werden kann.

Damit das Entscheidungsproblem für Prolog lösbar wird, darf die Klauselmenge K ausschließlich aus Horn-Klauseln bestehen. Dadurch wird das Entscheidungsproblem in endlicher Zeit lösbar[12]. In diesem Sinne handelt es sich um Prolog um eine vollständige Programmiersprache[13].

Hier soll jedoch nicht näher auf die theoretische Vorgehensweise eingegangen werden und eher ein grundlegendes Verständnis für die Arbeitsweise von Prolog am Beispiel vermittelt werden.

Für das Entwickeln von Prolog-Programmen ist das Verständnis der Verfahren Unifikation und Backtracking von essentieller Bedeutung, weshalb diese folgend erläutert werden.

Unifikation

Unifikation beschreibt allgemein eine Methode zur Vereinheitlichung prädikatenlogischer Ausdrücke. Dabei wird in Prolog ausgehend von Abfrage festgestellt, ob sich mithilfe der erfassten Regeln und Fakten mittels Substitution ein Term konstruieren lässt, der identisch zur Abfrage ist.

> Substitution: $\sigma = \{ X_1 \rightarrow t_1, ..., X_n \rightarrow t_m \}$ mit Variablen X_n und Termen t_m, $n \geq 0$, $m \geq 0$
>
> Unifikation: Zwei Terme t_1 und t_2 sind unifizierbar, wenn eine Substitution σ existiert, die die beiden Terme identisch macht: $\sigma(t_1) = \sigma(t_2)$
>
> Most General Unifier: Prolog wählt dabei genau die Substitution σ aus, die die minimale Anzahl an Substitutions-Operationen $X \rightarrow t$ aufweist.

In ISO-Prolog ist definiert, dass dafür der Herbrand-Algorithmus verwendet werden soll. Hier soll jedoch nicht der vollständige Algorithmus erläutert werden, sondern lediglich eine Grundvorstellung für den Vorgang anhand von Beispielen vermittelt werden[14].

Es sei zunächst ein kleines Prolog-Programm gegeben:

---

[12] „Endliche Zeit“ kann dabei „sehr lange“ bedeuten.

[13] Diese Aussage sollte nicht ohne gewissen Vorbehalten getroffen werden. Tatsächlich sind dieser theoretischen Vollständigkeit gewissen praktische Grenzen gesetzt. Dies wird jedoch näher im Abschnitt „Vor- und Nachteile für Spiele-KI-Entwicklung“.

[14] Der komplette Algorithmus ist in der Anlage I. beschrieben.

```
spalte(spalte1).
spalte(spalte2).
spalte(spalte3).

zeile(zeile1).
zeile(zeile2).
zeile(zeile3).

nachbar(spalte1, spalte2).
nachbar(spalte2, spalte3).

nachbar(zeile1, zeile2).
nachbar(zeile2, zeile3).

feld(X, Y) :- spalte(X), zeile(Y).

links_von(feld(X1, Y), feld(X2, Y)) :-
     feld(X1, Y),
     feld(X2, Y),
     nachbar(X1, X2).
```

Beispielprogramm

Es soll jetzt betrachtet werden, wie das System arbeitet, wenn folgende Abfrage gestellt wird:

```
?- feld(spalte1, zeile1).
```

Zunächst wird versucht, diejenige Teilmenge T aus allen Klauseln des Programms zu finden für die gilt: Es existiert ein σ, so dass gilt: σ(feld(spalte1, zeile1)) = σ(Kopf(T)). Dafür werden die Fakten und linken Seiten, bzw. der Kopf von Regeln, d. h. alles, was vor dem Terminalsymbol „`:-`“ steht, der Regeln durchsucht. Wichtig ist hierbei, das dabei strikt die Reihenfolge vom Beginn des Programmes bis zum Ende eingehalten wird. Das heißt, die erste Klausel „`spalte(spalte1).`“ wird zuerst, die Klausel „`links_von(...)`“ zuletzt überprüft.

Dabei wird die Regel feld/2 gefunden. Und es ergibt sich:

$T = \{$ `feld(X, Y) :- spalte(X), zeile(Y)` $\}$

$\sigma_{mgu} = \{$ `X` $\rightarrow$ `spalte1`, `Y` $\rightarrow$ `zeile1` $\}$

Die Substitution σ_{mgu} wird auf die Regel in T angewandt. X und Y werden damit mit den Werten `spalte1` und `zeile1` gefüllt.

Da der Kopf einer Klausel nur unter der Bedingung gilt, dass alle Unterziele erfüllt sind (jeder der durch „ , “ getrennten Terme ist ein Unterziel), muss die Gültigkeit der Unterziele nachgewiesen werden. Dies geschieht immer von Rechts nach Links.

Damit muss zuerst `spalte(spalte1)` und danach `zeile(zeile1)` überprüft werden. Für beide findet Prolog nach dem eben beschriebenen Verfahren jeweils die Fakten „`spalte(spalte1).`“ und „`zeile(zeile1).`“ die per Definition gültig sind und damit ist auch die Gültigkeit der Regel gegeben.

Wenn die Anfrage Variablen enthält, wird ähnlich verfahren. Stellt man z. B. die Anfrage

```
?- feld(spalte1, X).
```

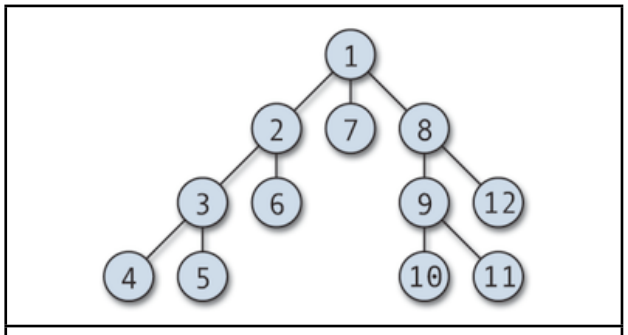
ergibt sich für $\sigma_{mgu}(T)$:

$\sigma_{mgu}(T) = \{$ `feld(spalte1, X) :- spalte(spalte1), zeile(X)` $\}$

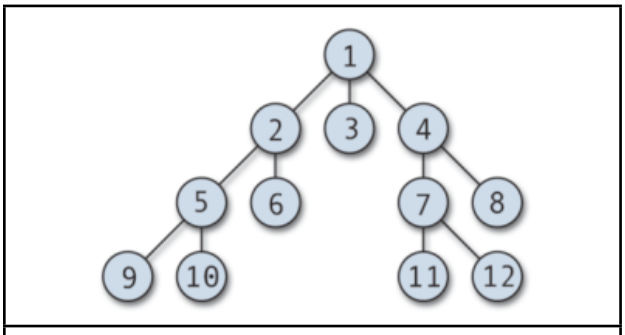
Da der Ausdruck `zeile(X)` für `X=zeile1`, `X=zeile2` und `X=zeile3` wahr wird, werden diese Werte neben der Meldung über den Erfolg als Ergebnis der Anfrage zurück geliefert. Hier wird im Prinzip schon das Backtracking angewendet, das jedoch im nächsten Abschnitt genauer erläutert werden soll.

Backtracking

Backtracking ist eine weitere in die Prolog-Inferenzmaschine eingebaute Funktion. Es führt dazu, dass immer stets *alle* Möglichkeiten[15] für rechte Seiten von Regeln betrachtet werden. Dadurch entsteht ein Lösungsbaum, der mit dem Depth-First-Verfahren (Tiefensuche) durchsucht wird. D.h., es wird immer zunächst ein Ast bis zur endgültigen Feststellung, ob es sich um eine gültige Lösung handelt, verfolgt und nicht wie bei dem Breadth-First-Verfahren (Breitensuche) alle möglichen Äste parallel mit gleicher Tiefe. Folgend die Reihenfolge der Auswertung für die beiden Such-Verfahren. Der Wert der Knoten steht dabei für die Position in der Auswertungsreihenfolge, die mit 1 beginnt.



Tiefensuche [B5]



Breitensuche [B6]

Es soll nun betrachtet werden, wie das Prolog-System bei Auswertung der Anfrage

```
links_von(feld(spalte1, zeile1), feld(Spalte, Zeile)).
```

an das oben aufgeführte Beispielprogramm verfährt. Durch Unifizierung wird dabei zunächst folgender Ausdruck erzeugt:

```
links_von(feld(spalte1, zeile1), feld(Spalte, zeile1)) :-
    feld(spalte1, zeile1),
    feld(Spalte, zeile1),
    nachbar(spalte1, Spalte).
```

Das Unterziel „`feld(spalte1, zeile1)`“ kann nur auf eine Art erfüllt werden. Das nächste Ziel „`feld(Spalte, zeile1)`“ erlaubt jedoch, durch die Variable `Spalte` mehrere Möglichkeiten. Hierbei durchsucht Prolog das Programm von Anfang bis Ende. Eine Klausel

[15] Die Ausnahme ist, wenn die schon erwähnte „nur eine Lösung“-Funktion des Prolog-Systems verwendet wird (bei XGP Button „once“). Hier findet auch Backtracking statt, allerdings nur bis der erste Weg gefunden wurden, der zu einer gültigen Lösungen führt.

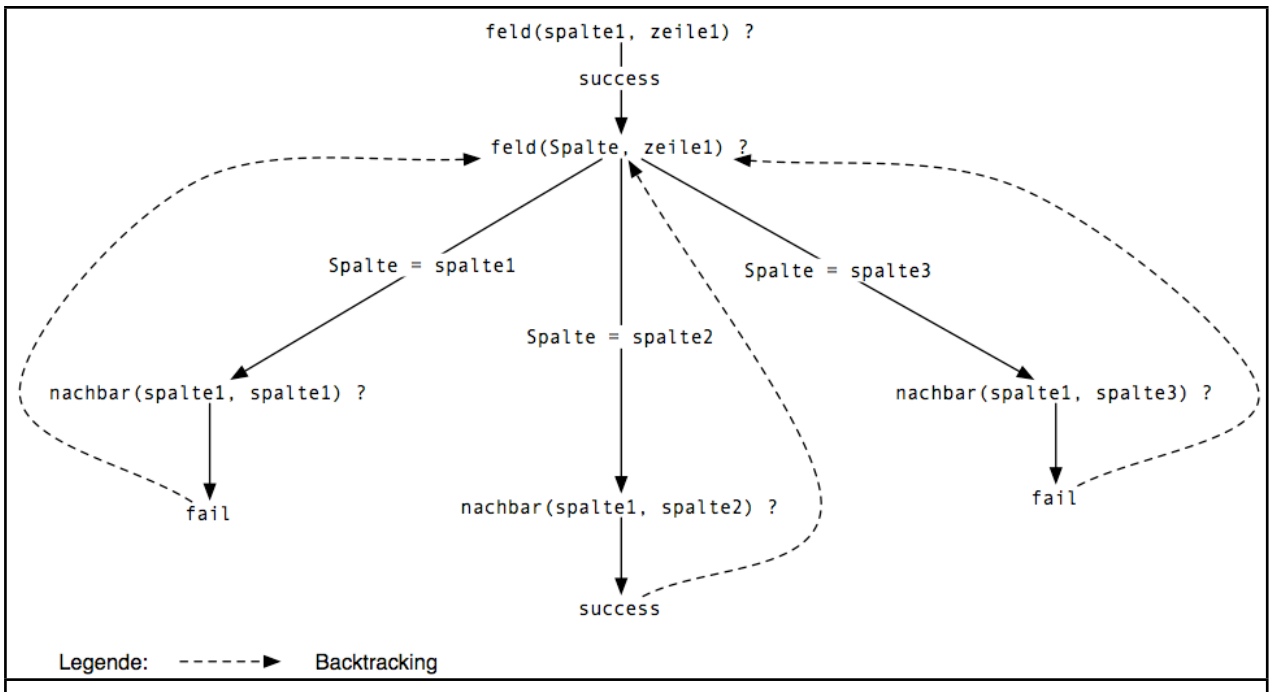
die vor einer anderen deklariert wurde, wird diese demnach zuerst eingesetzt, falls sie sich mit dem Term unifizieren lässt.

Für das Beispielprogramm heißt dies, dass zunächst mit dem Fakt „`feld(Spalte, zeile1) :- spalte(spalte1), zeile(zeile1).`“ unifiziert wird. Demnach wird der Variablen `Spalte` der Wert `spalte1` zugeordnet.

Im nächsten Schritt, wird versucht, das Unterziel „`nachbar(spalte1, spalte1).`“. Dieser befindet sich jedoch nicht in der Datenbasis und kann auch nicht aus dieser abgeleitet werden. Damit scheitert dieses Unterziel. An diesem Punkt wird jedoch nicht der ganze Ausdruck „`links_von(...)`“ als gescheitert angesehen, sondern Prolog kehrt zum vorherigen Unterziel zurück, dass sich noch erfolgreiche erfüllen ließ (backtracking) und versucht es hier, wenn möglich, mit einem anderen Wert.

Im Beispiel ist dies der Ausdruck „`feld(Spalte, zeile1)`“. Hier wurde der Variable `Spalte` durch Unifizierung der Wert `spalte1` zugeordnet. Dies jedoch nur, da der Fakt „`spalte(spalte1).`“ als erstes genannt wurde. Es wird jetzt die nächste passende Regel ausgewählt: „`spalte(spalte2).`“. Dadurch bekommt die Variable Spalte den Wert `spalte2` und das folgende Ziel „`nachbar(spalte1, spalte2)`“ lässt sich erfüllen und für „`links_von(...)`“ kann Erfolg zurückgeliefert werden sowie der Wert `Spalte=spalte2`.

Damit ist die Auswertung allerdings noch nicht abgeschlossen. Auch der Fall „`spalte(spalte3).`“ wird ausprobiert, führt hier allerdings nicht zum Erfolg. Das Backtracking auch ausgelöst wird, wenn alle Unterziele erfolgreiche ausgeführt wurden, zeigt sich, wie gesagt, auch am Beispiel aus dem vorherigen Abschnitt, bei dem als Ergebnis alle Zeilen für `spalte1` ausgegeben wurden.



Backtracking in einem Lösungsbaum [B7]

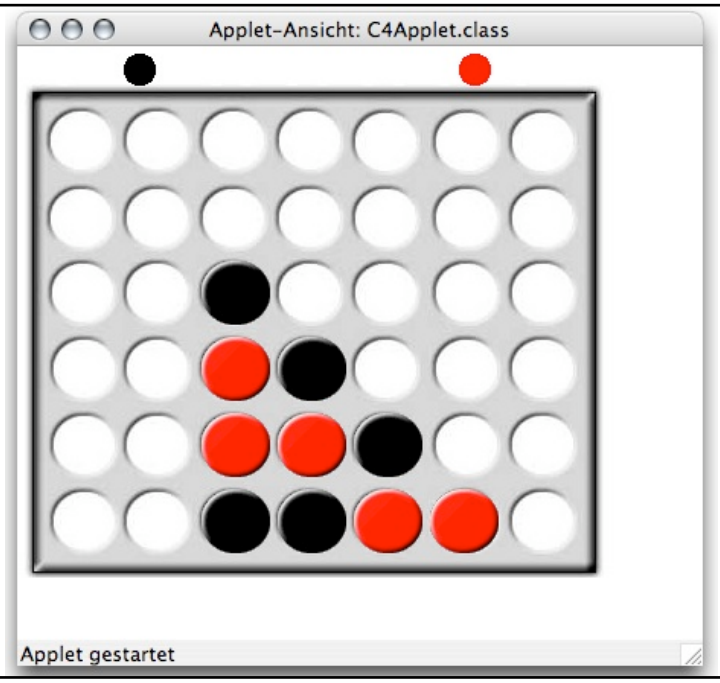
In der Anlage befindet sich der genau spezifizierte Backtracking-Algorithums, der hier jedoch nicht näher erläutert werden soll.

Beispiel für Spiele-KI in Prolog

4 Gewinnt (im Englischen „4 in a row“ oder „connect-four“) ist ein Zwei-Personen-Spiel mit perfekter Information[16]. Es handelt sich um ein „gelöstes“ Spiel. Das heißt, es wurde bewiesen, dass der anfangende Spieler immer gewinnen kann [L7 S.1]. Dennoch ist das Spiel für den menschlichen und einfachen KI-Spieler durchaus anspruchsvoll, weshalb es hier als Beispielanwendung für die Umsetzung einer Spiele-KI mit Prolog verwendet werden soll. Der Quellcode ist als Open Source unter [I6] erhältlich.

Schnittstelle Java/Prolog

Dabei sollen Darstellung sowie IO-Funktionen mittels Java realisiert werden und in Prolog ausschließlich die KI-Logik. Für die Java-Seite wurde [I3] als Vorlage verwendet. Des Weiteren wird die Bibliothek tuProlog eingesetzt, die einen vollständigen ISO-Prolog-Interpretierer in Java bereitstellt. Diese Bibliothek ist als Open Source unter [I4] frei erhältlich.



Ein 4 Gewinnt Spiel gegen die Prolog-KI [B8]

Vom Java Programm wird das Prädikat `zug_sum/4` für jeden im nächsten Spielzug möglichen Spielstein (Spalte, Zeile) aufgerufen:

```
Struct query = new Struct("zug_sum", ((C4Board) b).getSpielfeld(),
spielstein, new Var("X"), new Int(1));
SolveInfo answer = engine.solve(query);
```

Dabei wird über „`((C4Board) b).getSpielfeld()`“ das Spielfeld, sowie über „`spielstein`“ der zu prüfenden Spielstein als vorher generierte Prolog-Struktur übergeben. Das Ergebnis ist ein einzelner Integer-Wert, der die Güte eines Spielzuges darstellt und der in der Variablen „`new Var("X")`“ aufgenommen wird und der über „`answer.getTerm("X")`“ ausgelesen wird. Für Genaueres sei hier auf das Studium des

[16] „Perfekte Information“ bedeutet, dass alle Mitspieler zu jedem Zeitpunkt über alle vorangehenden Entscheidungen informiert sind, die für ihre aktuelle Entscheidung relevant sind. Andere bekannte Beispiel für Spiele mit perfekter Information sind Schach

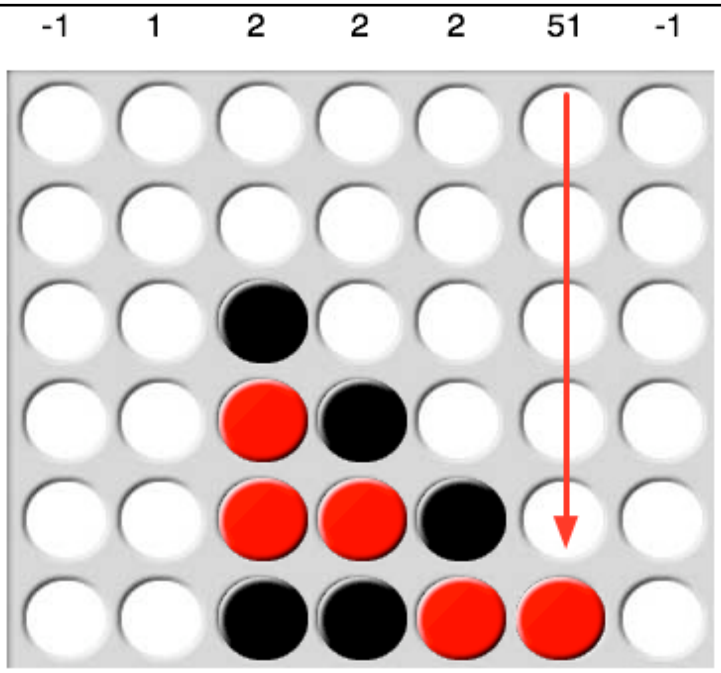
Quellcodes sowie die zu tuProlog bereitgestellte Dokumentation und die Anlage VIII. verwiesen.

Die Regel zug/4

Das Prädikat zug_sum/4 gibt an, wie geeignet oder ungeeignet ein Zug (Setzen eines Steins in einer bestimmte Spalte) aus Sicht der KI ist. Als grober Maßstab wird hier angenommen, dass ein Zug mit der Bewertung -100 sehr ungeeignet scheint, ein Zug mit der Bewertung +100 sehr geeignet erscheint. Um diese Werte zu errechnen, werden alle Lösungen für die Variable Bewertung für die Relation zug/4 addiert.

```
zug_sum(Spielfeld, GesetzterSpielstein, Bewertung, Tiefe).
```

Spielfeld repräsentiert dabei eine Liste aller bisher gesetzten Spielsteine, GesetzterSpielstein steht für den Zug, der geprüft werden soll, Bewertung die Güte dieses Spielzuges und Tiefe die Anzahl von Schritten, die die KI ausprobieren soll[17].

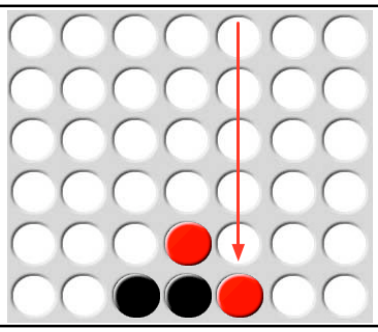


Bewertung einer Spielsituation durch zug_sum/4 [B9]

Intelligenz durch Logik

Durch die Relation `zug/4` werden unterschiedliche Situationen auf dem Spielfeld charakterisiert und mit einer Bewertung verbunden. Beispielhaft sei hier eine Situation aufgezeigt, die bei unerfahrenen Spielern häufig zur Spielentscheidung führt:

[17] Man könnte es an dieser Stelle so auffassen, dass es sich bei Spielfeld, GesetzterSpielstein und Tiefe um Input-Parameter und bei Bewertung um einen Output-Parameter handelt. Demnach könnte die Relation auch „bewerte“ genannt werden. Diese Vorgehensweise würde jedoch den Prinzipien einer logischen Programmiersprache widersprechen. Die Relation ist derart ausgelegt, dass jeder Parameter als Aus- und Eingabeparameter fungieren kann. Es lässt sich die Heuristik aufstellen, dass Verben als Namen für Prädikate ein Indiz dafür sind, dass man sich auf dem „falschen Weg“ im Sinne der logischen Programmierung befindet.



Verhinderung des doppelt offenen Dreiers [B10]

Der rote Spieler muss in der gezeigten Situation in die 5. oder 2. Spalte setzten, da der schwarze Spieler sonst gewinnt. Normalerweise erkennt eine künstliche „Intelligenz“ derartige Situationen, indem sie schlicht alle möglichen eigenen Züge mit den Reaktionen des Gegenspielers bis zu einer gewissen Zugtiefe ausprobiert[18]. Eine solche Vorgehensweise lässt sich problemlos ein Prolog umsetzten (alpha-beta search). Hier soll jedoch ein regelbasierter Ansatz verfolgt werden, der sich für die Vorstellung logischer Programmierung besser eignet. Es werden der KI bestimmte Regeln beigebracht, in der Form: wenn X gilt, ist der Zug Y gut. Dies wird über die schon beschriebene Relation `zug/4` erreicht. Um die Situation des doppelt offenen Dreiers zu erkennen und zu bewerten, wird folgendes Prädikat verwendet:

```
doppelter_dreier(Spielfeld1, Stein1) :-
     gesetzt(Spielfeld1, _, Stein4),
     vierer(Spielfeld1, Stein4 , Stein1, Stein2, Stein3),
     gesetzt(Spielfeld1, _, Stein5),
     vierer(Spielfeld, Stein5 , Stein1, Stein2, Stein3,),
     Stein4 \= Stein5.
```

Dies lässt folgend in natürliche Sprache übersetzten:

> Es liegt ein doppelter Dreier für ein Spielfeld $Spielfeld_1$ vor, an dem der Stein $Stein_1$ beteiligt ist, wenn:
>
> - ein auf dem Spielfeld $Spielfeld_1$ gesetzter Stein $Stein_4$
> - mit den Steinen $Stein_1$, $Stein_2$ und $Stein_3$ auf dem $Spielfeld_1$ eine Vierer-Reihe bildet,
> - ein auf dem Spielfeld $Spielfeld_1$ gesetzter Spielstein $Stein_5$
> - ebenfalls mit den Steinen $Stein_1$, $Stein_2$ und $Stein_3$ auf dem $Spielfeld_1$ eine Vierer-Reihe bildet
> - und $Stein_4$ und $Stein_5$ unterschiedliche Steine sind.

Dies ist eine sehr deklarative und damit leicht nachzuvollziehende Art und Weise, der KI „beizubringen“, wann ein doppelt offener Dreier vorliegt. Dies lässt sich z.B. daran ablesen, dass die Reihenfolge der Unterziele verändert werden kann, ohne dass sich die Bedeutung des Programmes verändert[19]. Es ist insbesondere nicht derart zu interpretieren, dass zuerst ein Spielstein gesetzt wird und danach geprüft wird, ob eine Vierer-Reihe vor-

---

[18] Hierfür gibt es leicht verbesserte Vorgehensweisen, die breite Verbreitung aufweisen, z. B. den auf dem Minmax-Prinzip aufbauenden Alpha-Beta-Algorithmus [s. L8 S. 395 ff.].

[19] Mit der Einschränkung, dass das Unterziel $Stein_4$ \= $Stein_5$ immer zuletzt stehen muss. Auf diesen Umstand wird im Zusammenhang der Problematik mit `not/1` eingegangen.

liegt. Es geht hier nur um die Eigenschaft des Spielsteines; dass er setzbar ist. So könnte auch zuerst nach Steinen $Stein_1$, $Stein_2$, $Stein_3$ und $Stein_4$, die einen Vierer-Reihe bilden und im Anschluss geprüft werden, ob sich $Stein_4$ überhaupt auf dem Spielfeld setzten lässt.

Der beschriebenen Situation wird im Anschluss durch

```
zug(Spielfeld, Spielstein, 15, _):-doppelter_dreier(Spielfeld, Spielstein).
```

eine Bewertung von +15 zugeordnet. D. h. jeder doppelt offene Dreier, der durch den Spielstein erzeugt wird, führt zu einer Erhöhung der Güte-Bewertung dieses Zuges um 15. Der Wert 15 ist hier willkürlich gewählt, kann allerdings auch durch genaue Analysen bestimmt werden.

Die KI verhindert den doppelten Dreier dadurch, dass sie stets die Kosten für die Unterlassung für einen Zug berechnet. Die Kosten für die Unterlassung werden als die Hälfte des Güte-Wertes angenommen, die der Gegner durch Setzen des Steins erreicht[20]:

```
zug([_|SpielfeldTail], spielstein(F1, Spieler), Bewertung / 2, Tiefe) :-
     zahl(Tiefe), Tiefe > 0,
     beide_spieler(Spieler, AndererSpieler),
     zug_sum(SpielfeldTail,
          spielstein(F1,AndererSpieler),Bewertung,Tiefe-1).
```

[20] Der Ausdruck „`[_|SpielfeldTail]`“ bedeutet dabei: Nehme von der übergebenen Liste nur den Teil ohne Kopfelement.

Vor- und Nachteile für Spiele-KI-Entwicklung

In diesem Abschnitt soll diskutiert werden, ob sich die Sprache Prolog für die Entwicklung von Künstlicher Intelligenz in Spielen eignet. Dafür muss zunächst definiert werden, was die wesentlichen Eigenschaften einer Spiele-KI sind.

Für den Begriff Künstliche Intelligenz gibt es keine allgemein gültige Definition. Hier soll eine sehr pragmatische Definition für Spiele-KI verwendet werden:

> Unter Spiele-KI werden alle Techniken verstanden, die den Spieler beim Umgang mit dem Spiel unterstützen, komplexe Zusammenhänge in der Spielwelt simulieren, dem Spieler die Illusion von weiteren menschlichen Mit- oder Gegenspielern bieten oder Objekte der Spielwelt derart steuern, dass sie eine Herausforderung für den Spieler bieten.

Zuerst sollen die Allgemeinen Vor- und Nachteile der Sprache Prolog und folgend die speziell für die Spiele-KI-Entwicklung geltenden aufgeführt werden.

Allgemein

Vorteile

Viele Implementierungen

Es sind viele Open Source-Implementierungen verfügbar, sowohl als vollständige Umgebung, mit denen Prolog kompiliert oder interpretiert werden kann, als auch als vollständig verfügbarer Quellcode für eine Wirtssprache ([I2], [I4]).

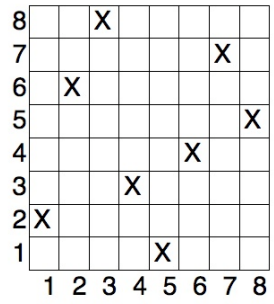
Des Weiteren wurde Prolog standardisiert durch Norm ISO/IEC 13211-1 [L5].

Gut geeignet für Backtracking-Probleme

Probleme, die sich mit Depth-First-Suche mit Backtracking lösen lassen, führen zu sehr kompakten Programmen. Folgend die wichtigsten Prädikate eines Prolog-Programms, um das 8-Damen Problem zu lösen. Der komplette Quellcode ist in der Anlage IV. zu finden[21].

[21] Teile des Programmes sind aus [L1 S. 130 (Sekundärquelle)] entnommen. Es ist anzumerken, dass das hier vorliegende Programm bei Auswahl der Funktion „alle Lösungen“ nicht die erwarteten 12 Lösungen zurück liefert, sondern 92, was darauf zurückzuführen ist, dass alle Lösungen, die durch Spiegelung und Rotation des Brettes entstehen ebenfalls gezählt werden. Eine Anfrage, um die Anzahl der Lösungen zu bestimmen, wäre:

```
?-findall(Ergebnis, damen(8, [], Ergebnis), Ergebnisse), length(Ergebnisse, Anzahl-
Loesungen).
[...] AnzahlLoesungen=92
```



Eine Lösung für das 8-Damen-Problem [B11]

```
sichere_position(_, []).
sichere_position(p(X1, Y1), [p(X2, Y2)|R]):-
     zahl(X1), zahl(Y1), zahl(X2), zahl(Y2),
     X1=\=X2, Y1=\=Y2, % keine gleichen Zeilen/Spalten
     Dx is X1-X2, Dy is Y1-Y2, % keine gleiche Diagonale
     Dx =\= Dy, Dx =\= -Dy,
     sichere_position(p(X1, Y1), R).

sichere_stellung([]).
sichere_stellung([F|R]):-sichere_position(F, R), sichere_stellung(R).

damen(0, Ergebnis, Ergebnis).
damen(N, Start, Ergebnis):-
     zahl(N), zahl(Y),
     sichere_stellung([p(N, Y)|Start]),
     eins_weniger(N, NMinusEins),
     damen(NMinusEins, [p(N, Y)|Start], Ergebnis).
...
?-damen(8, [], Ergebnis).
Ergebnis = [p(8, 7), p(7, 6), p(6, 8), p(5, 4), p(4, 5), p(3, 2), p(2, 3),
p(1, 1)]
```

8-Damen-Problem in Prolog

Ein Java-Programm, dass die gleiche Aufgabe löst, ist als Anlage V. vorhanden. Es ist festzustellen, dass dieses nicht nur wesentlich mehr Codezeilen aufweist, sondern aufgrund der zahlreichen Schleifen schwieriger nachzuvollziehen ist, gesetzt den Fall, dem Betrachter sind Prolog- und Java-Syntax gleichermaßen vertraut.

Relationen allgemeiner als Funktionen

Des Weiteren führt relationale Denkweise zu allgemeineren Lösungen als bei einem funktionalen Ansatz. Bei „reiner“ Prolog-Programmierung wird nicht zwischen Ein- und Ausgabeparametern unterschieden. D. h., eine Regel eins_mehr kann auch dafür verwendet werden, -1 zu berechnen.

Das gegebene Prolog-Programm für das 8-Damen-Problem ist allgemeiner als das Java-Programm. Von ihm wird nicht nur die Anfrage „Gebe mir die Positionen von acht Damen aus, die sich nicht bedrohen.“, sondern z. B. auch die Anfrage „Wie viele Damen bedrohen sich in einer Position nicht?“ beantwortet.

```
?-damen(N, [], [p(1, 2), p(2, 6), p(3, 8), p(4, 3), p(5, 1), p(6,
4), p(7, 7), p(8, 5)]).
N = 8
```

Geeignet für parallele Verarbeitung

Wenn meta-logische Konstrukte nicht verwendet werden[22], eignet sich Prolog prinzipiell ausgezeichnet für paralleles Berechnen auf Mehrprozessorsystemen. Hier können die Äste des Lösungsbaums gleichzeitig durchsucht werden.

Nachteile

Softwaretechnische Mängel

Prolog wurde vor fast 40 Jahren entwickelt und modernere Erkenntnisse der Softwaretechnik wurden beim Sprachentwurf nicht berücksichtigt. So gibt es z. B. nur einen globaler Namensraum für Prädikate, was besonders bei großen Projekten zu Problemen führen kann. Auch Modularisierung ist nur rudimentär möglich und steht beispielsweise dem Java Packages-Konzept um einiges nach.

Moderne Prolog-Implementierungen bieten hier allerdings schon einige Verbesserungen [s. I5].

Wie bei allen Sprachen mit deklarativem Ansatz gestaltet sich das Debugging eher schwierig, da, was das Programm macht, natürlich nicht Schritt für Schritt nachvollzogen werden kann.

Probleme mit Negation

Horn-Klauseln erlauben nicht die Definition einer Alternative beziehungsweise einer Negation. Die Negation kann in Prolog zwar definiert werden, allerdings nur durch direkte Kontrolle des Backtracking[23].

```
not(X) :- X, !, fail.
not(_).

spalte(spalte1). spalte(spalte2).
spalte(spalte3).

nachbar(spalte1, spalte2).
nachbar(spalte2, spalte3).
```

Prolog-Programm mit Prädikat `not/1`

```
?-not(nachbar(spalte1, X)), spal-
te(X).
Failure

?-not(nachbar(spalte1, spalte3)),
spalte(spalte3).
Success
```

Anfrage an Prolog-Programm mit Prädikat `not/1`

[22] Dies sind z. B. `assert/1`, `rectract/1` sowie `!/0`. Auf diese soll hier jedoch nicht weiter eingegangen werden, da hier die logische Eigenschaften von Prolog im Mittelpunkt stehen sollen.

[23] Dies geschieht durch den Cut-Operator !/0. Er führt dazu, dass alle Wertzuweisungen zu Variablen, die in vor dem Cut-Operator stehenden Unterzielen vorgenommen wurden, als unveränderbar angesehen werden. D.h., hier werden im Zuge des Backtracking keine alternativen Möglichkeiten mehr ausprobiert.

Im angeführten Beispiel könnte man erwarten, dass die Anfrage „`not(nachbar(spalte1, X)), spalte(X).`“ die Lösungen `spalte3` und `spalte1` liefert, die, wie die zweite Anfrage zeigt, eine gültige Lösung ist. Es wird jedoch nur `Failure` zurückgeliefert.

Dies liegt daran, dass `not/1` nicht mit uninstanzierten Variablen arbeiten kann. Eine Abhilfe im Beispiel wäre die Abfrage „`spalte(X), not(nachbar(spalte1, X)).`“ zu verwenden. Damit wird die Reihenfolge der Klauseln signifikant für das Ergebnis, was dem deklarativen Ansatz völlig widerspricht (s. auch [L2 S. 126] „The problem with not“).

Probleme mit arithmetischen Auswertungen

Arithemtische Ausdrücke werden in Prolog zunächst nicht ausgewertet. D. h., die Anfrage

```
2+5 = 5+2
```

wird zunächst mit `Failure` beantwortet. Da sich die Ausdrücke `2+5` und `5+2` nicht unifizieren lassen. Eine arithmetische Auswertung muss mithilfe des `is/2`-Operators erzwungen werden. Diese erfordert jedoch (ähnlich wie `not/1`), dass alle verwendeten Variablen instanziert sind und weiter noch, dass es sich bei dem linken Ausdruck um eine Zahl oder eine Variable, die nur eine Zahl enthält, handelt. Damit die Anfrage positiv beantwortet wird, muss sich demnach wie folgt umgestaltet werden:

```
Links is 2+5, Rechts is 5+2, Links=Rechts
```

Die Verwendung des Operators `is/2` führt damit auch dazu, dass Prolog-Prädikate ihre Allgemeingültigkeit verlieren. D. h., dass eine Unterscheidung zwischen Ein- und Ausgabeparametern vorgenommen werden muss. So liefert die Regel

```
plus_1(Zahl, ZahlPlus1) :- ZahlPlus1 is Zahl + 1.
```

Auf die Anfrage „`plus_1(Zahl, 2).`“ keineswegs die erwartete Antwort „`1`“, sondern einen „`instantiation_error`“, da es sich bei der linken Seite um keine instanzierte Variable handelt. Dies ist besonders problematisch, da keine syntaktische Unterscheidung zwischen Ein- und Ausgabeparametern vorgenommen werden kann (wie z. B. in Pascal mit „`var`“ und „`const`“).

Keine rein deklarative Programmierung möglich

Die beiden vorangegangen Punkte zeigen auf, dass die Reihenfolge, in der die Klauseln aufgeschrieben werden, von signifikanter Bedeutung für ein Prolog-Programm ist. Es muss bei einem Prolog-Programm neben der deklarativen Bedeutung auch stets die prozeduale Bedeutung, d.h. in welcher Reihenfolge und wie die Berechnung vonstatten geht, betrachtet werden. So ist z. B. das Programm

```
benachbart(spalte1, spalte2).
benachbart(X, Y) :- benachbart(Y, X).
```

deklarativ korrekt. Die Anfrage „`?-once(benachbart(spalte2, spalte1))`“[24] wird erwartungsgemäß beantwortet. Stellt man jedoch die Anfrage „`?-benachbart(spalte2, X)`“

[24] Bei dem Prädikat `once/1` handelt es sich um eine meta-logische Funktion. Sie sorgt dafür, dass, nachdem die erste Lösung gefunden wurde, die Verarbeitung abgebrochen wird. Sie ist damit ähnlich dem Button „once“ im System XGP.

führt dies zu einer unendlich langen Bearbeitung, da im Zuge der Unifizierung ständig der Ausdruck „`benachbart(X, Y)`“ mit dem Ausdruck „`benachbart(Y, X)`“ ausgetauscht wird.

Damit zeigt sich auch, dass es sich bei Prolog um eine nicht vollständige Abbildung der Prädikatenlogik handelt (s. [L9 S. 141]: „PROLOG is ‚incomplete‘ in that it offers only an approximation to logic programming.“). Dies ist darauf zurückzuführen, dass Prolog nicht nur die Erfüllbarkeit einer Formel bestimmt (success oder fail), was in polynominaler Laufzeit ausgeführt werden könnte, sondern zusätzlich alle möglichen Variablenbelegungen ermittelt.

Sehr großen Einfluss hat die prozeduale Bedeutung auf die Ausführungs-Effizienz (Rechenzeit- und Speicherplatzverbrauch). Folgend zwei Implementierungen des Prädikates `vierer/5`, die sich in der Ausführungs-Effizienz dramatisch voneinander unterscheiden, obschon ihre deklarative Bedeutung identisch ist:[25]

```
vierer(Spielfeld, spielstein(F1, Spieler),  spielstein(F2, Spieler),
spielstein(F3, Spieler),  spielstein(F4, Spieler)):-
     permutation([F1, F2, F3, F4], [V1, V2, V3, V4]),
     reihe(V1, V2, V3, V4),
     member(spielstein(F2, Spieler), Spielfeld),
     member(spielstein(F3, Spieler), Spielfeld),
     member(spielstein(F4, Spieler), Spielfeld).
```

Effiziente Implementierung des Prädikates `vierer/5`.

```
vierer(Spielfeld, spielstein(F1, Spieler),  spielstein(F2, Spieler),
spielstein(F3, Spieler),  spielstein(F4, Spieler)):-
     member(spielstein(F2, Spieler), Spielfeld),
     member(spielstein(F3, Spieler), Spielfeld),
     member(spielstein(F4, Spieler), Spielfeld),
     permutation([F1, F2, F3, F4], [V1, V2, V3, V4]),
     reihe(V1, V2, V3, V4).
```

Ineffiziente Implementierung des Prädikates `vierer/5`.

Dies ist dadurch bedingt, dass in der zweiten Implementierung zuerst drei zufällige Elemente aus der Liste der Spielsteine entnommen werden. Dies ergibt, wenn die Liste der Spielsteine lang wird, sehr viele Möglichkeiten:

> Anzahl der Kombinationen n-ter Ordnung aus N Elementen ohne Wiederholung und mit Berücksichtigung der Anordnung:
>
> N! / (N - n)!

[25] Das Prädikat `member/2` dient dabei, die Zugehörigkeit eines Elements zu einer Liste festzustellen. `permutation/2` erzeugt alle Permutationen einer Liste.

Länge der Liste	Möglichkeiten
3	6
5	60
10	720
20	6840
100	970200

Anzahl der Möglichkeiten für n=3

Hierbei zeigt sich auch ein eklatanter Mangel des Prolog-Ansatzes für deklarative Programmierung. Die Programme sind oft nicht nur ein wenig ineffizient, so dass die Weiterentwicklung der Hardware diese Nachteil mit der Zeit kompensieren würde, es sind oft mathematische Gesetzmäßigkeiten, die viele, unbedachte deklarative Programme praktisch unbrauchbar machen. Der Entwickler ist vielmehr ständig angehalten, die prozeduale Bedeutung seiner Programme im Blick zu behalten. Damit geht jedoch der Vorteil des deklarativen Ansatzes größtenteils verloren.

So zeigt sich auch an der KI für Vier Gewinnt, dass der erste, rein deklarative Versuch noch unter Ausführungs-Effizienz-Gesichtspunkten optimiert werden musste, damit die KI in angemessener Zeit ihre Entscheidung treffen kann. Die optimierte Version und die ursprüngliche können unter [I6] verglichen werden.

Architekturmodell des modernen PCs ungeeignet

Die Architektur des modernen PC ist nicht optimiert für die nicht-numerische und parallele Verarbeitung von Prolog.

Für Spiele-KI-Entwicklung
Vorteile

Schnelle Entwicklung von KI-Prototypen

Ein erfahrener Prolog-Entwickler kann Prototypen für eine Spiele-KI, sofern keine Ein-, Ausgabe- oder Grafikelemente erforderlich sind, schneller entwickeln als ein ähnlich erfahrener C++-Entwickler, da die Prolog-Syntax kompakter ist und so gut wie keine Verwaltungsoperationen (Initialisierung Variablen und explizite Speicherplatzfreigabe) erforderlich sind und sich durch das eingebaute Backtracking Kontrollstrukturen einsparen lassen.

Geeignet für intelligente Agenten

Intelligente Agenten in Spiele-KI sollten logisches Schlussfolgern beherrschen, was bei Verwendung von Prolog nicht mehr implementiert werden muss[26]. Folgend ein Beispiel für einen Agenten, der in sich in einer bestimmten Umwelt für einen Weg entscheiden muss:

[26] Zu einem solchen Agenten gibt es auch einen Artikel in [L10] "A Simple Interference Engine for Rule-Based Architecture", Mike Christian. Zu regelbasierten Systemen mit Inferenzmaschine heißt es dort allgemein: "[The rule-based system] gives you the ability to understand and manipulate behaviors at a high abstraction level, that of goals, and it is natural how we humans think of the game characters behaving.“ (S. 307).

```
ort(weg1).
ort(weg2).
ort(stadt).
ort(brunnen).
verbunden(stadt, weg1).
verbunden(stadt, weg2).
verbunden(weg1, brunnen).
verbunden(weg2, brunnen).

befindet_sich(gegner, weg1).
befindet_sich(ich, stadt).
durstig.

guter_weg(Start, Ueber, Ziel):-
     ort(Start),
     ort(Ueber),
     not(befindet_sich(gegner, Ueber)),
     ort(Ziel).

gehe(Ueber):-
     durstig,
     befindet_sich(ich, Start),
     guter_weg(Start, Ueber, brunnen).
```

Prolog-Implementierung eines Agenten, der einfaches Schlussfolgern beherrscht

Nachteile

Geringe Unterstützung für APIs

Die meisten Prolog-Implementierungen bieten geringe bis meist keine Unterstützung für application programming interfaces (APIs) wie Direct X oder OpenGL, die als essentiell für die Entwicklung moderner Computerspiele angesehen werden können.

Wenig Erfahrung bei Entwicklern

Die meisten Mitglieder eines Spiele-Entwickler-Teams werden eher auf ausgeprägte C++-Erfahrungen zurückgreifen können, was die Produktivität in dieser Sprache signifikant erhöhen sollte, selbst für Aufgaben, für die Prolog prinzipiell besser geeignet ist.

Geringe Performance

„Bottleneck“ bei der Spieleentwicklung (auch der Spiele-KI-Entwicklung) ist oft die Performance. Deklarativ ausgerichtete Prolog-Programme weisen oft eine geringe Performance auf und Performance-Optimierungen gestaltet sich in Prolog eher schwierig und führen zu nicht mehr deklarativen Programmen.

Formale Logik oft zu exakt

In der Prädikatenlogik gilt das Gesetzt des ausgeschlossenen Dritten. D. h., etwas ist entweder wahr oder nicht wahr. Viele reale und Spiel-Situationen erfordern dagegen die Feststellung, dass etwas nur mit einer gewissen Wahrscheinlichkeit eintrifft. Geeignetes Mittel, solche Sachverhalte auszudrücken ist die Fuzzy-Logik[27].

[27] Vgl. dazu Artikel „"An Open-Source Fuzzy Logic Library", Michael Zarozinsk in [L10]. Dort heißt es u. a.: "FFLL can save significant time and money." (S. 101).

Abschließende Bemerkungen

Der schwerwiegenste Nachteil von Prolog ist konzeptioneller Natur. Die deklarative Programmierung ist schlicht nicht möglich. Dies bräuchte nicht weiter problematisch zu sein, da es sich hierbei nicht unbedingt um einen Mangel von Prolog handelt, sondern oft mathematische Gesetzmäßigkeiten (exponentielle Laufzeit!) den deklarativen Ansatz verhindern. Für Probleme, für die es keine effizienteren Algorithmen als den brute force Ansatz gibt, eignet sich Prolog demnach sehr gut, da auch mit einem imparaitven Ansatz keine bessere Performance erzielt werden kann.

Unverzeihlich ist jedoch die Tatsache, dass deklarative und nicht deklarative Elemente nicht unterschieden werden können. So ist es den Prädikaten oft nicht anzusehen, ob Parameter als reine In- oder Outputparameter fungieren müssen[28].

Dennoch kann es in bestimmten Situationen vorteilhaft sein, eine logische Programmiersprache zu verwenden, um kleinere Aspekte einer KI zu implementieren, die logisches Schlussfolgern betreffen. Dafür muss die Sprache, in der das Spiel implementiert ist, eine gute Schnittstelle zu Prolog besitzen. Das hier vorgestellte tuProlog, bei dem Prolog-Code in Java interpretiert wird, bietet eine solche Schnittstelle, ist allerdings im Vergleich zu anderen Prolog-Implementierungen sehr langsam.

Keinesfalls ist es ratsam ein ganzes Spiel oder die gesamte KI-Funktionalität in Prolog zu implementieren. Dagegen sprechen die softwaretechnischen Mängel und die geringe Performance.

[28] Oft wird dies bei der Dokumentation einer Prolog-Funktion versucht. Ein ? steht dabei für In- oder Outputparameter, ein + für reinen Inputparameter. Die Bedeutung dieser Zeichen kann jedoch auch leicht variieren.

Anlagen

I. Herbrand Algorithmus

The Herbrand Algorithm for an MGU

Given a set of equations of the form $t_1 = t_2$ apply in any order one of the following non-exclusive steps:

1. If there is an equation of the form:
   1. `f=g` where `f` and `g` are different atomic terms, or
   2. `f=g` where `f` is an atomic term and `g` is a compound term, or `f` is a compound term and `g` is an atomic term, or
   3. `f(...) = g(...)` where `f` and `g` are different functors, or
   4. $f(a_1, a_2, \ldots, a_N) = g(b_1, b_2, \ldots, b_M)$ where `N` and `M` are different.

   then exit with failure (*not unifiable*).
2. If there is an equation of the form `x = x`, `x` being a variable, then remove it.
3. If there is an equation of the form `c = c`, `c` being a atomic term, then remove it.
4. If there is an equation of the form $f(a_1, a_2, \ldots, a_N) = f(b_1, b_2, \ldots, b_N)$ then replace it by the set of equations $a_i = b_i$.
5. If there is an equation of the form `t = x`, `x` being a variable and `t` a non-variable term, then replace it by the equation `x = t`,
6. If there is an equation of the form `x = t` where:
   1. `x` is a variable and `t` a term in which the variable does not occur, and
   2. the variable `x` occurs in some other equation,

   then substitute in all other equations every occurrence of the variable `x` by the term `t`.
7. If there is an equation of the form `x = t` such that `x` is a variable and `t` a non-variable term which contains this variable, then exit with failure (*not unifiable, positive occurs check*.
8. If no other step is applicable, then exit with success (*unifiable*).

Quelle: [L5]

II. Backtracking-Algorithmus

Sei P eine Wissensbasis bestehend als Regeln und Fakten, sei G eine Anfrage, dann berechnet der folgende Algorithmus eine Variablenbelegung σ, falls G aus P herleitbar ist.

Der Algorithmus für Rückwärtsverkettung benötigt weiterhin zwei Stacks:

– GOALLIST: Liste der aktuellen Literale der Anfrage

– CHOICEPOINTS: Entscheidungspunkte für Backtracking

1. Initialisierung: *GOALLIST* := G; *CHOICEPOINTS* := Ø, σ := Ø;
2. falls *GOALLIST* = Ø
   dann stop, *SUCCESS* = (true, σ);
3. L := top(*GOALLIST*), *GOALLIST* := pop(*GOALLIST*), *CLAUSES* := P;
4. falls *CLAUSES* = Ø
   dann falls *CHOICEPOINTS* := Ø
   dann *SUCCESS* = (false, Ø)
   sonst (L, *GOALLIST*, σ, *CLAUSES*) := top(*CHOICEPOINTS*);
5. *S*:= top(*CLAUSES*), *CLAUSES* := pop(*CLAUSES*), H := Konklusion(*S*), L' := Lσ, H' := Hσ;
6. falls L'τ = H'τ für eine Substitution τ
   dann σ := στ,
   GOALLIST := Prämissen(S) ∪ *GOALLIST*,
   STATE := (L, *GOALLIST*, σ, *CLAUSES*),
   CHOICEPOINTS := push(*STATE*,*CHOICEPOINTS*),
   gehe zu 2
   sonst gehe zu 4;

Quelle: Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Andreas Abecker (Sekundärquelle) (http://www.dfki.uni-kl.de/~aabecker/Mosbach/PrologHandouts.pdf)

III. Türme von Hanoi

Folgenden ein Beispiel, welches das bekannte Rätsel der „Türme von Hanoi“ löst. Die Grundidee für diese Lösung ist entnommen aus [L1 S. 125, Sekundärquelle]. Sie wurde hier jedoch leicht modifiziert. Es ist zu beachten, dass die Moves in umgekehrter Reihenfolge ausgegeben werden. D. h., der letzte Schritt steht am Anfang der Liste.

```
hanoi(0, A, B, _, MovesInit, MovesInit).
hanoi(N, A, B, C, MovesInit, Moves):-
     N > 0,
     M is N-1,
     hanoi(M, A, C, B, MovesInit, Moves2),
     hanoi(M, C, B, A, [move(A, B)|Moves2], Moves).
...
?-hanoi(3, links, rechts, mitte, [], Moves).
Moves = [move(links, rechts), move(mitte, rechts), move(mitte, links), mo-
ve(links, rechts), move(rechts, mitte), move(links, mitte), move(links,
rechts)]
```

Türme von Hanoi in Prolog

IV. 8-Damen-Problem (Prolog)

```
zahl(1). zahl(2). zahl(3).
zahl(4). zahl(5). zahl(6).
zahl(7). zahl(8).

eins_weniger(Zahl, ZahlMinusEins):-
     zahl(Zahl),
     (zahl(ZahlMinusEins);ZahlMinusEins=0),
     Zahl - 1 =:= ZahlMinusEins.

sichere_position(_, []).
sichere_position(p(X1, Y1), [p(X2, Y2)|R]):-
     zahl(X1), zahl(Y1), zahl(X2), zahl(Y2),
     X1=\=X2, Y1=\=Y2, % keine gleichen Zeilen/Spalten
     Dx is X1-X2, Dy is Y1-Y2, % keine gleiche Diagonale
     Dx =\= Dy, Dx =\= -Dy,
     sichere_position(p(X1, Y1), R).

sichere_stellung([]).
sichere_stellung([F|R]):-sichere_position(F, R), sichere_stellung(R).

damen(0, Ergebnis, Ergebnis).
damen(N, Start, Ergebnis):-
     zahl(N), zahl(Y),
     sichere_stellung([p(N, Y)|Start]),
     eins_weniger(N, NMinusEins),
     damen(NMinusEins, [p(N, Y)|Start], Ergebnis).
```

V. 8-Damen-Problem (Java)

Das folgende Programm wurde ohne Änderungen von der Seite (http://spaz.ca/aaron/SCS/queens/) übernommen. Es werden hier allerdings nur die für den Lösungsalgorithmus relevanten Funktionen der Klasse Board angegeben:

```
    /**
     * Adds (or removes) a queen to the given coordinates.
     * If add is true, it adds a queen. If it is false, it removes a queen.
     */
    private void addQueen(int x, int y, boolean add) {
         int i;
         gr = this.getGraphics();
         if (add)
              grid[x][y] = 2;
         else
              grid[x][y] = 0;
         for (i=0;i<8;i++) {
              if (i!=y) changeCell(x,i,add);
              if (i!=x) changeCell(i,y,add);
         }
         for (i=1; legal(x+i,y+i); i++)
              changeCell(x+i,y+i,add);
         for (i=1; legal(x+i,y-i); i++)
              changeCell(x+i,y-i,add);
         for (i=1; legal(x-i,y+i); i++)
              changeCell(x-i,y+i,add);
         for (i=1; legal(x-i,y-i); i++)
              changeCell(x-i,y-i,add);

         drawCell(x,y);
         if (!add)
              for (i=0;i<8;i++)
                   for (int j=0;j<8;j++)
                        if (grid[i][j] == 2)
                             addQueen(i,j,true);
    }
```

```
    /**
     * Add or remove a queen to the given cell
     */
    private void changeCell(int x, int y, boolean add) {
         if (add && grid[x][y] == 0)
              grid[x][y] = 1;
         else if (!add && grid[x][y] == 1)
              grid[x][y] = 0;
         drawCell(x,y);
    }
```

```
    /**
     * Returns true if the coordinates are a legal board position
     */
    private boolean legal(int x, int y) {
         return (x >= 0 && x < 8 && y >= 0 && y < 8);
    }
```

```
/**
 * Solve the 8-queens puzzle with a very simple depth-first search.
 */
public boolean solve(int y) {
     int  i,j;
     boolean r = false;

     for (i=0;i<8;i++) {
          // for each row, try placing a queen
          if (grid[i][y] == 0) {
               addQueen(i,y,true);
               if (y == 7) {
                    return true; //we have placed all queens successfully
               } else {
                    if (solve(y+1)) {
                         return true; // we solved it, return
                    } else {
                         addQueen(i,y,false); // remove the queen
                    }
               }
          }
     }
     // unable to solve down this branch -- retreat!
     return false;
}
```

VI. 4 Gewinn - Prolog-Quellcode

```
spieler(spieler0).
spieler(spieler1).
beide_spieler(X, Y):-spieler(X), spieler(Y), X \= Y.

zahl(0). zahl(1). zahl(2). zahl(3). zahl(4). zahl(5). zahl(6). zahl(7).
zahlenreihe(X, Y) :- zahl(X), zahl(Y), (X+1) =:= Y.
zahlenreihe(X, Y, Z) :- zahl(X), zahl(Y), zahl(Z), (X+1) =:= Y, (Y+1) =:= Z.
nachbarzahlen(X, Y) :- zahlenreihe(X, Y).
nachbarzahlen(X, Y) :- zahlenreihe(Y, X).

spalte(X) :- zahl(X), X > 0, X < 8.
zeile(X) :- zahl(X), X > 0, X < 7.

benachbart(feld(X1, Y1), feld(X2, Y2)):-
     (nachbarzahlen(X1, X2), nachbarzahlen(Y1, Y2));
     (nachbarzahlen(X1, X2), Y1=Y2);
     (X1=X2, nachbarzahlen(Y1, Y2)).

feld(X, Y) :- spalte(X), zeile(Y).

% waagerecht
reihe(feld(X1, Y1), feld(X2, Y1), feld(X3, Y1)) :-
     zahlenreihe(X1, X2, X3),
     feld(X1, Y1), feld(X2, Y1), feld(X3, Y1).
% horizontal
reihe(feld(X1, Y1), feld(X1, Y2), feld(X1, Y3)) :-
     zahlenreihe(Y1, Y2, Y3),
     feld(X1, Y1), feld(X1, Y2), feld(X1, Y3).
% diagonal nach oben
reihe(feld(X1, Y1), feld(X2, Y2), feld(X3, Y3)) :-
     zahlenreihe(X1, X2, X3),
     zahlenreihe(Y1, Y2, Y3),
     feld(X1, Y1), feld(X2, Y2), feld(X3, Y3).
% diagonal nach unten
reihe(reihe(feld(X1, Y1), feld(X2, Y2), feld(X3, Y3))) :-
     zahlenreihe(X1, X2, X3),
     zahlenreihe(Y3, Y2, Y1),
     feld(X1, Y1), feld(X2, Y2), feld(X3, Y3).

reihe(F1, F2, F3, F4) :-
     reihe(F1, F2, F3), reihe(F2, F3, F4).

gesetzt(Spielfeld, [spielstein(feld(X, Y), Spieler)|Spiel-
feld],spielstein(feld(X, Y), Spieler)) :-
     zahlenreihe(OldY, Y),
     feld(X, Y),
     (OldY = 0;
     member(spielstein(feld(X, OldY), _), Spielfeld)),
     not(member(spielstein(feld(X, Y), _), Spielfeld)).

% vierer(Spielfeld, zuTestenderStein, WeitererStein1, WeitererStein2, WeitererS-
tein3)
vierer(Spielfeld, spielstein(F1, Spieler),  spielstein(F2, Spieler),
spielstein(F3, Spieler),  spielstein(F4, Spieler)):-
     permutation([F1, F2, F3, F4], [V1, V2, V3, V4]),
     reihe(V1, V2, V3, V4),
     member(spielstein(F2, Spieler), Spielfeld),
     member(spielstein(F3, Spieler), Spielfeld),
     member(spielstein(F4, Spieler), Spielfeld).

dreier(Spielfeld, spielstein(F1, Spieler), spielstein(F2, Spieler),
spielstein(F3, Spieler)):-
     permutation([F1, F2, F3], [Dreier1, Dreier2, Dreier3]),
```

```
     reihe(Dreier1, Dreier2, Dreier3),
     member(spielstein(F2, Spieler), Spielfeld),
     member(spielstein(F3, Spieler), Spielfeld).

doppelter_dreier(Spielfeld, Dreier1) :-
     dreier(Spielfeld, Dreier1, Dreier2, Dreier3),
     gesetzt(Spielfeld, Sp2, Vierer11),
     vierer(Spielfeld, Vierer11, Dreier1, Dreier2, Dreier3),
     gesetzt(Sp2, _, Vierer21),
     vierer(Spielfeld, Vierer21, Dreier1, Dreier2, Dreier3),
     Vierer21 \= Vierer11.

offener_dreier(Spielfeld, spielstein(F1, Spieler)) :-
     dreier(Spielfeld, spielstein(F1, Spieler), S32, S33),
     vierer(Spielfeld, spielstein(Fehlt, Spieler), spielstein(F1, Spieler), S32,
S33),
     not(member(spielstein(Fehlt, _), Spielfeld)),
     not(gesetzt(Spielfeld, _, spielstein(Fehlt, _))).

% zug(Spielfeld, gesetzerSpielstein, Bewertung, Tiefe)
% Bewertung: 100 mach es, -100 lass es

% wenn nichts Besonderes, dann weder vor- noch nachteilhaft
zug(_,_, 0, _).

% Stein am Rand bringt Kummer und Schand
zug(_, spielstein(feld(X, _), _), -1, _):-X < 3.
zug(_, spielstein(feld(X, _), _), -1, _):-X > 4.
zug(_, spielstein(feld(_, Y), _), -1, _):-Y > 3.

% Siegstein
zug(Spielfeld, Spielstein, 100, _):-
     once(vierer(Spielfeld, Spielstein, _, _, _)).

% alles, was fuer meinen Gegner gut ist, ist fuer mich schlecht
% Direkte Kosten fuer Unterlassung
zug([_|SpielfeldTail], spielstein(F1, Spieler), Bewertung / 2, Tiefe) :-
     zahl(Tiefe), Tiefe > 0,
     beide_spieler(Spieler, AndererSpieler),
     zug_sum(SpielfeldTail, spielstein(F1, AndererSpieler), Bewertung, 0).

% Nicht Zuarbeiten fuer den Gegner
zug(Spielfeld, spielstein(feld(Spalte, _), Spieler), Bewertung / 2, Tiefe) :-
     zahl(Tiefe), Tiefe > 0,
     beide_spieler(Spieler, AndererSpieler),
     gesetzt(Spielfeld, _, spielstein(feld(Spalte, ZeileNeu), _)),
     zug_sum(Spielfeld, spielstein(feld(Spalte, ZeileNeu), AndererSpieler), Neg-
Bewertung, 0),
     Bewertung = - NegBewertung.


% zwei zusammen ist besser als gar nichts
zug(Spielfeld, spielstein(F1, Spieler), 2, _):-
     benachbart(F1, F2),
     member(spielstein(F2, Spieler), Spielfeld).

% offener Dreier
zug(Spielfeld, Spielstein, 6, _):-
     offener_dreier(Spielfeld, Spielstein).

% Dreier mit zwei direkten Ansetzpunkten
zug(Spielfeld, Spielstein, 15, _):-
     doppelter_dreier(Spielfeld, Spielstein).
```

```
% List: Liste der positiven und negativen Bewertungen des Zuges
zug_alle(Spielfeld, Spielstein, List, Tiefe) :-
     findall(X, zug(Spielfeld, Spielstein, X,Tiefe), List).
% Sum: Summe der positiven und negativen Bewertungen des Zuges
zug_sum(Spielfeld, Spielstein, Sum, Tiefe) :-
     gesetzt(Spielfeld, SpielfeldMitStein, Spielstein),
     zug_alle(SpielfeldMitStein, Spielstein, List, Tiefe), sum(List, Sum).


% -=- Hilfsfunktionen -=-

% sum(List, Sum) : List ist Input-Parameter, Sum ist Output!
sum([], 0).
sum([Head|Tail], Sum) :- sum(Tail, SumU), Sum is SumU+Head.

not(X):-call(X), !, fail.
not(_).

assert_list([Head|Tail]):-assert(Head), assert_list(Tail).
assert_list([]).

takeout(X,[X|R],R).
takeout(X,[F|R],[F|S]) :- takeout(X,R,S).
permutation([X|Y],Z) :- permutation(Y,W), takeout(X,Z,W).
permutation([],[]).
```

VII. 4 Gewinnt - Java Schnittstelle

```
public Move getMove(Board b) {
          Move[] moves = b.getPossibleMoves(this);
          Vector<Possibility> vals = new Vector<Possibility>();


          for(Move m: moves) {
               try {
                    Struct feld = new Struct("feld", new Int(m.toInt()+1), new
Int(((C4Board) b).numerOfChipsInColumn(m.toInt())+1 ));
                    Struct spielstein = new Struct("spielstein", feld, playerTerm);
                    Struct query = new Struct("zug_sum", ((C4Board) b).getSpielfeld(),
spielstein, new Var("X"), new Int(1));
                    System.out.println("Query: "+query);
                    long startTime= System.currentTimeMillis();;
                    SolveInfo answer = engine.solve(query);
                    long stopTime = System.currentTimeMillis();
                    System.out.println(" Time:" + (stopTime -  startTime) + " millise-
conds");

                    if (answer != null && answer.isSuccess()) {
                         Term derivative = answer.getTerm("X");
                         Possibility ps = new Possibility(m.toInt(), ((Int)
derivative).intValue());
                         vals.add(ps);
                         System.out.println(ps);
                    } else {
                         System.out.println("Keine Moeglichkeit fuer Spalte:
"+m.toInt());
                    }
               } catch (InvalidTermException e) {
                    e.printStackTrace();
               } catch (NoSolutionException e) {
                    e.printStackTrace();
               } catch (UnknownVarException e) {
                    e.printStackTrace();
               }
          }

          Collections.sort(vals);
          System.out.println("Best Move: "+vals.get(vals.size()-1));

          return new C4Move(this, vals.get(vals.size()-1).column);
     }
```

VIII. Handout Einstieg in Prolog

Woher?

- für Windows/Linux: http://www.swi-prolog.org/ „SWI-Prolog“
- für Mac OS X: http://sourceforge.net/projects/xgp/ „XGP Prolog“
- In Java: http://www.alice.unibo.it:8080/tuProlog/ „tuProlog“

Sollte ein Link nicht funktionieren, hilft wahrscheinlich eine google-Suche nach den angegebenen Begriffen.

Erstes „Programm“

```
farbe(rot).
farbe(gruen).
ist(rasen, gruen).
ist(erdbeere, rot).
ist(portugal, gruen).
ist(portugal, rot).
zweifarbig(X):-farbe(F1), farbe(F2), F1 \= F2, ist(X, F1), ist(X, F2).
/* IO */
alle_farben(Farben):-findall(X, farbe(X), Farben).
alle_farben:-alle_farben(X), write(X), nl.
```

Ein Programm benötigt zunächst keine Ein- oder Ausgabe. Das Programm kann durch Eingabe von Queries getestet werden. Solche wären:

```
:- ist(X, rot).
X = erdbeere, X = portugal
:- zweifarbig(rasen).
nein
:- zweifarbig(X).
X = portugal
```

Prolog und Java

Die Bibliothek tuProlog ist eine Implementierung eines Prolog-Interpretierers in Java. Sie kann ohne weitere Vorbedingungen in jedem Java-Projekt durch Einbinden des Archivs „2P.jar“ verwendet werden.

Dies geschieht in Eclipse durch Rechtsklick auf das Projekt im „Navigator“ und Auswahl von „Build Path / Add External Archives ...“.

Es muss zunächst einen Prolog-Quelle erstellt werden. Dafür kann der oben aufgeführte Code in eine Datei „theory.pl“ kopiert werden.

Es ist ratsam, die Prolog-Quelle zunächst mit einem anderen Prolog-Interpretierer zu testen, da tuProlog recht sparsam über Fehler im Quellcode berichtet.

In das gleiche Verzeichnis wird die Bibliothek „2P.jar“ kopiert. Jetzt kann eine Java-Quelle „Test.java“ mit folgendem Inhalt erstellt werden:

```
import java.io.FileNotFoundException;
import java.io.IOException;
import alice.tuprolog.InvalidTheoryException;
import alice.tuprolog.MalformedGoalException;
import alice.tuprolog.NoMoreSolutionException;
import alice.tuprolog.NoSolutionException;
import alice.tuprolog.Prolog;
import alice.tuprolog.SolveInfo;
import alice.tuprolog.Term;
import alice.tuprolog.Theory;
import alice.tuprolog.UnknownVarException;

public class Test {

	public static void main(String[] args) {
		try {
			Prolog engine = new Prolog();
			Theory t = new Theory(new java.io.FileInputStream("theory.pl"));
			engine.setTheory(t);
			SolveInfo answer = engine.solve("farbe(X).");

			while (answer != null && answer.isSuccess()) {
				Term derivative = answer.getTerm("X");
				System.out.println("Lösung: "+derivative.toString());
				try {
					answer = engine.solveNext();
				} catch (NoMoreSolutionException e) {
					answer = null;
				}
			}
		} catch (FileNotFoundException e) { e.printStackTrace();
		} catch (InvalidTheoryException e) { e.printStackTrace();
		} catch (MalformedGoalException e) { e.printStackTrace();
		} catch (NoSolutionException e) { e.printStackTrace();
		} catch (UnknownVarException e) { e.printStackTrace();
		} catch (IOException e) { e.printStackTrace();
		}
	}

}
```

Compilieren der Quellen: `$ javac -cp 2P.jar Test.java`

Ausführen: `$ java -cp 2P.jar:. Test`

Direkt: `$ java -cp 2P.jar alice.tuprolog.Agent theory.pl 'alle_farben.'`

Literaturangaben

[L1] Helder Coelho, José C. Cotta, „Prolog by Example“, Springer-Verlag New York 1988

[L2] Tony Dodd, „Prolog – A Logical Approach“, I. Title, Oxford University Press 1990

[L4] Michael Winikoff, „W-Prolog“ (http://goanna.cs.rmit.edu.au/~winikoff/wp/, Abruf am 29.6.2007)

[L5] J.P.E. Hodgson „Prolog: The ISO Standard Documents“ (http://pauillac.inria.fr/~deransar/prolog/docs.html, Abruf am 29.6.2007)

[L6] ISO - International Organization for Standardization (http://www.iso.org/)

[L7] Victor Allis „A Knowledge-based Approach of Connect-Four“ (z.B. unter http://www.tomski.com/archive/connect4.pdf, Abruf am 29.6.2007)

[L8] Ivan Bratko, „Prolog - Programmierung für Künstliche Intelligenz“, Addison Wesley Verlag 1987

[L9] Wolfgang Kreutzer, Bruce McKenzie, „Programming for Artificial Intelligence“, Addison-Wesley 1991

[L10] Steve Rabin u. a. , „AI Game Programming Wisdom“, Charles River Media 2002,

Internetquellen

[I1] XGP: http://xgp.sourceforge.net/

[I2] Prolog-Interpretierer in Perl:
http://search.cpan.org/~jjore/AI-Prolog-0.739/lib/AI/Prolog.pm

[I3] „Four in a row Applet“ von Sean Bridges (
http://www.geocities.com/ResearchTriangle/System/3517/C4/C4Conv.html)

[I4] Prolog-Interpretierer in Java: tuProlog (http://sourceforge.net/projects/tuprolog/)

[I5] SWI-Prolog, Prolog-Umgebung (http://www.swi-prolog.org/)

[I6] Webseite zur Ausarbeitung:

http://www.mxdt.de/davread/Studies/Prolog_f_r_Spie/Prolog_f_r_Spie.html oder über

http://www.mxro.de/

Abbildungsverzeichnis