Prolog (Programmiersprache)

aus Wikipedia, der freien Enzyklopädie

Prolog (vom Französischen: ***Pro**grammation en **Log**ique*,[1] dt.: „Programmieren in Logik“) ist eine Programmiersprache, die Anfang der 1970er Jahre maßgeblich von dem französischen Informatiker Alain Colmerauer entwickelt wurde und ein deklaratives Programmieren ermöglicht. Sie ist die wichtigste logische Programmiersprache.

Prolog	
Paradigmen:	logisch, deklarativ, oft auch constraintbasiert
Erscheinungsjahr:	1972
Designer:	Alain Colmerauer
Entwickler:	Philippe Roussell
Typisierung:	schwach, dynamisch
Wichtige Implementierungen:	SICStus, SWI-Prolog, GNU Prolog, YAP-Prolog
Dialekte:	ISO-Prolog, Edinburgh Prolog, BinProlog, Visual/Turbo Prolog; historisch: micro-Prolog
Beeinflusst von:	Q-Systems, Theorembeweiser, Horn-Klauseln
Beeinflusste:	Erlang, Mercury, Mozart/Oz

Erste Implementierungen wichen in ihrer Syntax stark voneinander ab, aber der Edinburgh-Dialekt setzte sich bald als Quasistandard durch. Er war jedoch nicht formal definiert,[2] bis er 1995 zur Grundlage eines ISO-Standards wurde (ISO/IEC 13211-1), der auch ISO-Prolog genannt wird.

Der erste Prolog-Interpreter wurde in Marseille in ALGOL W realisiert. Der erste Ansatz für einen Compiler stammte von David H. D. Warren aus Edinburgh. Dieser hatte als Zielsprache die des Logik-Prozessors Warren’s Abstract Machine und erlaubte deshalb weder dynamische Änderungen noch einen Anschluss rücksetzbarer Prädikate in anderen Programmiersprachen. Der erste voll nutzbare Compiler, der beides erlaubte, wurde von Preben Folkjaer in München entwickelt. Er verwandte einen anderen, von der TU Wien stammenden, Zwischencode der inkrementell compiliert wurde; wurden Prädikate verändert, wurde das Compilat gelöscht und beim nächsten Aufruf neu compiliert.

Inhaltsverzeichnis

Grundprinzip

Prolog-Programme bestehen aus einer Datenbasis, deren Einträge sich Fakten und Regeln nennen. Der Benutzer formuliert Anfragen an diese Datenbasis. Der Prolog-Interpreter benutzt die Fakten und Regeln, um systematisch eine Antwort zu finden. Ein positives Resultat bedeutet, dass die Anfrage logisch ableitbar ist. Ein negatives Resultat bedeutet nur, dass aufgrund der Datenbasis keine Ableitung gefunden werden kann. Dies hängt eng mit der Closed world assumption zusammen (siehe unten).

Das typische erste Programm in Prolog ist nicht wie in prozeduralen Programmiersprachen ein Hallo-Welt-Beispiel, sondern eine Datenbasis mit Stammbauminformationen.

Folgendes Beispiel repräsentiert den Stammbaum einer kleinen Familie. Das erste Faktum in Form einer Aussage *mann(tobias).* liest sich als: Tobias ist ein Mann. *vater(tobias, frank).* definiert das Faktum: Tobias ist der Vater von Frank.

```
mann(adam).
mann(tobias).
mann(frank).
frau(eva).
frau(daniela).
frau(ulrike).
vater(adam,tobias).
vater(tobias,frank).
vater(tobias,ulrike).
mutter(eva,tobias).
mutter(daniela,frank).
mutter(daniela,ulrike).
```

In einem Prolog-Interpreter können nun interaktiv Anfragen an die Datenbasis gestellt werden. Das Ausführen eines Prolog-Programms bedeutet immer das Stellen einer Anfrage. Das System antwortet entweder mit *yes.* oder *no.*, abhängig davon, ob die Anfrage bewiesen werden konnte. Der Interpreter signalisiert mit der Eingabeaufforderung *?-*, dass er eine Anfrage erwartet:

```
?- mann(tobias).
yes.
?- mann(heinrich).
no.
```

Eine Anfrage mit einer Variablen liefert als Antwort zusätzlich Belegungen, mit denen die Anfrage wahr wird. Man nennt eine solche Variablenbelegung Unifikation und sagt, die Variable wird mit diesem Wert unifiziert. Variablen sind in Prolog Token, die mit einem Großbuchstaben beginnen:

```
?- frau(X).
X=eva
X=daniela
X=ulrike
```

no ist die Antwort auf die um die vorher ausgegebenen Antworten reduzierte Faktenliste.

Der Interpreter liefert nur positive Antworten auf Anfragen, die explizit definiert oder folgerbar sind (Closed world assumption). So liegen etwa über *heinrich* keinerlei Informationen in der Datenbasis:

```
?- mann(heinrich).
no.
?- frau(heinrich).
no.
```

Zusätzlich zu Fakten lassen sich in Prolog Regeln formulieren. Der Regeloperator :- ist dabei wie ein umgedrehter Implikationspfeil zu lesen. Beispiel:

```
grossvater(X,Y) :-
    vater(X,Z),
    vater(Z,Y).
```

Die Regel besagt: X ist Großvater von Y, wenn es ein Z gibt, sodass X Vater von Z ist und Z Vater von Y. Damit ist der Großvater väterlicherseits definiert. Eine zweite Regel für den Großvater mütterlicherseits sieht so aus:

```
grossvater(X,Y) :-
    vater(X,Z),
    mutter(Z,Y).
```

Der Operator "," in dieser Regel definiert eine Konjunktion und liest sich "und". Der Term links vom Implikationsoperator nennt sich auch Head oder Konsequenz. Haben zwei Regeln (wie oben) die gleiche Konsequenz, folgt diese, wenn mindestens in einer Regel die Vorbedingung erfüllt ist (Disjunktion).

Durch die Definition von Regeln können auch Fakten geschlossen werden, die nicht explizit in der Datenbasis stehen:

```
?- grossvater(adam,ulrike).
yes.
?- grossvater(X,frank).
X=adam
```

Weitere Techniken

Entscheidend für die Prolog-Programmierung sind die Techniken der Rekursion und die Nutzung von Listen.

Ist die Rekursion in den meisten Programmiersprachen nur eine zusätzliche Variante zur Iteration, ist sie bei der Prolog-Programmierung die einzige Möglichkeit, *Schleifen* zu produzieren. Benötigt man in obigem Beispiel eine allgemeine *Vorfahr*-Relation, wird das wie folgt realisiert (";" zeigt in der Klausel die Disjunktion bzw. das logische "oder"):

```
vorfahr(X,Z) :-
    elternteil(X,Z).

vorfahr(X,Z) :-
    elternteil(X,Y),
    vorfahr(Y,Z).

elternteil(X,Y) :- mutter(X,Y); vater(X,Y).
```

Dies lässt sich wie folgt lesen: X ist ein Vorfahr von Z, wenn X Elternteil von Z ist (Regel 1) oder es ein Y gibt, das Vorfahr von Z ist und gleichzeitig X Elternteil von Y (Regel 2) (Es wurde hier *elternteil* statt *mutter* oder *vater* verwendet.)

Auch Listen sind ein entscheidender Bestandteil von Prolog. Die meisten Prolog-Implementationen bringen dafür viele Basisfunktionen mit ("concat" = Anhängen von Werten, "count" = Anzahl der Werte, etc.), die sich aber auch alle selbst definieren lassen. In einer gedachten Familienstruktur muss die Anzahl der Kinder ja variabel sein. Folgendes wäre denkbar:

```
familie(heinz,jutta,[peter,laura]).
familie(karl,gertrud,[]).
```

Dann ließen sich z. B. mit einer Abfrage alle Männer ohne Kinder anzeigen:

```
?- familie(X, _, []).
X=karl
```

Dabei ist X die Variable, deren verschiedene Werte ausgegeben werden sollen. Der Unterstrich _ ist in Prolog die anonyme Variable, wodurch Prolog veranlasst wird hier jeden Wert zuzulassen. Die eckigen Klammern stehen für die leere Menge, welche die nicht vorhandenen Kinder repräsentiert.

Eine weitere Eigenschaft und Besonderheit gegenüber anderen Programmiersprachen ist, dass Prolog in der Lage ist, während der Laufzeit seine vorhandene Datenbank zu erweitern oder zu löschen. Ein Beispiel für das Löschen eines einzelnen Elements:

```
auto(bmw,rot).
auto(bmw,blau).
autofarbe(Automarke,X) :-
    retract(auto(bmw,_)),
    auto(Automarke,X).
```

Die Abfrage:

```
?- auto(bmw,X).
```

ergibt (anfänglich) ganz normal:

```
X=rot;
X=blau;
No
```

Die Abfrage:

```
?- autofarbe(bmw,X).
```

würde beim ersten Mal:

```
X=blau;
No
```

beim zweiten Mal nur noch:

```
No
```

liefern, da die Informationen:

```
auto(bmw,rot).
auto(bmw,blau).
```

aus der Datenbank gelöscht wurden. Auch *?- auto(bmw,X)* liefert jetzt nur noch *No*. Zum Löschen aller gleichen Elemente (also z. B. *auto()*) auf einmal benutzt man retractall(), zum Ausgeben asserta() (oben in der Datenbank) und assertz() (unten in der Datenbank).

Beispiele

Lösen eines mathematischen Rätsels

```
ABB -  CD = EED
 -     -     *
 FD +  EF =  CE
 =     =     =
EGD *  FH = ???
```

A bis H stehen jeweils für eine Ziffer 0 bis 9, wobei nicht klar ist, welche Zahl an welchen Buchstaben gebunden ist. Gesucht ist die Zahl, die bei den Fragezeichen stehen muss. Dieses Problem ist in Prolog sehr einfach zu lösen. Man schreibt zunächst eine Regel, die bewirkt, dass A bis H später mit allen möglichen Kombinationen von 0 bis 9 belegt werden (Permutation):

```
gen(A,B,C,D,E,F,G,H) :- permutation([A,B,C,D,E,F,G,H, , ],[0,1,2,3,4,5,6,7,8,9]).
```

Nun müssen nur die fünf entstehenden Gleichungen (*ABB – CD = EED*, *FD + EF = CE*, *ABB – FD = EGD*, *CD – EF = FH* und *EED * CE = EGD * FH = **X***) in Prolog-Syntax geschrieben werden:

```
gl1(A,B,C,D,E) :- ((A * 100 + B * 10 + B) - (C * 10 + D)) =:= (E * 100 + E * 10 + D).
gl2(C,D,E,F) :- ((F * 10 + D) + (E * 10 + F)) =:= (C * 10 + E).
gl3(A,B,D,E,F,G) :- ((A * 100 + B * 10 + B) - (F * 10 + D)) =:= (E * 100 + G * 10 + D).
gl4(C,D,E,F,H) :- ((C * 10 + D) - (E * 10 + F)) =:= (F * 10 + H).
gl5(C,D,E,F,G,H,X) :- ((E * 100 + E * 10 + D) * (C * 10 + E)) =:=
    ((E * 100 + G * 10 + D) * (F * 10 + H)), X is ((E * 100 + G * 10 + D) * (F * 10 + H)).
```

Interessiert nur *X*, wird eine Lösungsregel angelegt, die alles zusammenführt und *X* ausgibt:

```
loesung :- gen(A,B,C,D,E,F,G,H), gl1(A,B,C,D,E), gl2(C,D,E,F),
    gl3(A,B,D,E,F,G), gl4(C,D,E,F,H), gl5(C,D,E,F,G,H,X), write(X).
```

Wird nun die Abfrage *loesung.* eingegeben, wird die Lösung ausgegeben. Wie man sieht, benötigt man zur Lösung dieses Problems fast keine Programmierkenntnisse über Schleifen oder ähnliches, sondern gibt nur die Fakten ein und welches Ergebnis man benötigt. Prolog steht in der Abstraktionshierarchie aus genau diesem Grund über imperativen und objektorientierten Sprachen.

Bearbeitung hierarchischer Strukturen

Eine häufig gestellte Aufgabe an Programmiersprachen ist die Verarbeitung hierarchischer Strukturen, wie z. B. SGML oder XML. Insbesondere für XML bildet Prolog eine sehr wirkungsvolle und ausdrucksstarke Alternative zu der verbreitetsten Verarbeitungssprache XSLT.

Ein typischer XML-Baum wie

```
<buch titel="Peer Gynt">
    <autor name="Henrik Ibsen" nat="norwegisch"/>
    ...
</buch>
```

wird unter Prolog als rekursive Liste von Elementen *element(TagName, Attribute, Kinder)* dargestellt.

```
[element(buch, [titel='Peer Gynt'], [
    element(autor, [name='Henrik Ibsen', nat='norwegisch'], []),
    ...])
]
```

Ein sehr einfaches Paradigma (untere drei Klauseln) erlaubt es, jeden Baum rekursiv zu durchlaufen. Folgende Beispiele löschen (oberster Klausel mit *delete*) und konkatenieren (zweiter Klausel von oben mit *concat*) bestimmte Tags. Der erste Unifikator ist die Operation (*delete* oder *concat*), der zweite die zu bearbeitende Struktur, der dritte das spezifizierte Tag, der vierte der Ergebnisbaum. *append* ist ein Befehl zum Konkatenieren von Listen.

```
transform(delete, [element(DelTag, _, _) | Siblings], DelTag, ResTree) :-
    transform(delete, Siblings, DelTag, ResTree).

transform(concat, [Element1, Element2 | Siblings], ConTag, ResTree) :-
    Element1 = element(Contag, Attr, Children1),
    Element2 = element(Contag, _, Children2),
    append(Children1, Children2, Children),
    transform(concat, [element(ConTag, Attr, Children) | Siblings], ConTag, ResTree).

transform(_, [], _, []).

transform(Trans, [element(CTag, Attr, Children) | Siblings], Tag, ResTree) :-
    \+ Tag = CTag,
    transform(Trans, Children, Tag, ResChildren),
    transform(Trans, Siblings, Tag, ResSiblings),
    ResTree = [element(CTag, Attr, ResChildren) | ResSiblings].

transform(_, [Atom], _, [Atom]) :-
    atomic(Atom).
```

Stößt der Backtracker bei der Operation *delete* auf ein Tag, das wie das zu löschende heißt, so wird dieses entfernt und bei den Nachbarn weitergesucht. Ein entsprechender Aufruf ist z. B. *transform(delete, Tree, autor, ResTree).*, der alle Autoren entfernt.

Ähnlich können durch *transform(concat, Tree, paragraph, ResTree).* alle nebeneinanderstehenden Paragraphen miteinander verschmolzen werden. Dazu werden zunächst deren Inhalte konkateniert, daraus eine neue Paragraphstruktur erzeugt und diese weiterverarbeitet.

Planungssysteme

Planungssysteme suchen eine Möglichkeit, von einem Ausgangszustand in einen gewünschten Zielzustand zu gelangen. Sie lassen sich für die Suche von Straßen- oder Verkehrsverbindungen, aber auch für allgemeinere

Problemstellungen einsetzen. Zunächst der allgemeinste Ansatz für eine *blinde Tiefensuche* (d. h. es ist unbekannt, ob der einzelne Schritt auch näher zum Ziel führt):

```
weg(Ziel,Ziel,Zustandsliste) :-
    write(Zustandsliste),nl.                /* Ziel erreicht, Abbruch der Rekursion und Aus
weg(Start,Ziel,Zustandsliste) :-            /* Es gibt einen Weg vom Start zum Ziel, wenn
    operator(Op),                           /* ... es einen Operator gibt, ... */
    anwendbar(Op,Start),                    /* ... der im Startzustand anwendbar ist, ...
    fuehrt_zu(Op,Start,Neu),                /* ... von dort zu einem neuen Zustand fuehrt,
    not(member(Neu,Zustandsliste)),         /* ... der noch nie da war (Verhinderung von Sc
    zulaessig(Neu),                         /* ... und zulaessig ist, ... */
    weg(Neu,Ziel,[Neu|Zustandsliste]).      /* ... und es von dort einen Weg zum Ziel gibt.
```

Nur die Prädikate *operator*, *anwendbar*, *fuehrt_zu* und *zulaessig* sowie die Beschreibung eines Zustands sind problemspezifisch zu formulieren. Aufgerufen wird das Prädikat mit einer Zustandsliste, die den Anfangszustand enthält.

Abhängig vom Problemtyp lässt sich einiges vereinfachen und/oder weglassen; für eine Wegesuche in einem Straßennetz ergibt sich z. B.

```
weg(Ziel,Ziel,Ortsliste):-
    write(Ortsliste),nl.                    /* Ziel erreicht, Abbruch der Rekursion und Aus
weg(Start,Ziel,Ortsliste):-                 /* Es gibt einen Weg vom Start zum Ziel, wenn
    strasse(Start,Neu),                     /* ... es eine Strasse vom Start zu einem neuer
    not(member(Neu,Ortsliste)),             /* ... in dem man noch nicht war (Verhinderung
    weg(Neu,Ziel,[Neu|Ortsliste]).          /* ... und von dem es einen Weg zum Ziel gibt.
```

Bei realen Problemen führt eine blinde Suche selten zum Ziel; man benutzt eine Breitensuche, bei der alle vom Start aus erreichbaren neuen Zustände ermittelt, mit einer *Heuristikfunktion* bewertet und nur der beste (*Heuristische Suche*) oder eine sortierte Liste der besten (*Best-first-Suche*) weiterverfolgt werden. (Die einfache heuristische Suche kann dazu führen, dass nicht immer die optimale Lösung gefunden wird, da bestimmte Lösungsschritte, die fälschlicherweise als ungünstig aussortiert wurden, sich als bessere Lösung ergeben würden.) Die Kunst liegt in der richtigen problemspezifischen Formulierung der Heuristikfunktion. In vielen Fällen hilft die *A-Heuristik*, das ist die Summe aus bisher erbrachtem Aufwand und geschätztem Restaufwand zum Ziel (z. B. zurückgelegte Fahrtstrecke + Luftliniendistanz zum Zielort):

```
weg(Ziel,Ziel,Ortsliste,Strecke):-
    write(Ortsliste),nl,write(Strecke),nl.          /* Ziel erreicht, Abbruch der
weg(Start,Ziel,Ortsliste,Strecke):-                 /* Es gibt einen Weg vom Start
    findall(Ort,strasse(Start,Ort),Neuliste),       /* ...es eine Liste erreichbare
    bewerte(Neuliste,Start,Strecke,Ziel,BewerteteListe),  /* ...von denen jeder bewertet
    sort(BewerteteListe,SortierteListe),            /* ...durch Sortieren der Liste
    member([_,Sgesamt,Neu],SortierteListe),         /* ...der beste gesucht wird,..
    not(member(Neu,Ortsliste)),                     /* ...in dem man noch nicht wa
    weg(Neu,Ziel,[Neu|Ortsliste],Sgesamt).          /* ...und von dem es einen Weg
```

Jedes Element von BewerteteListe hat die Struktur [Heuristikwert,gesamte Fahrtstrecke,Ort]; zur Berechnung der A-Heuristik sind die bisherige Strecke, der letzte Ort und der Zielort (Luftlinie) erforderlich.

Einsteins Rätsel

Dies ist eine Version des Zebrarätsels. Es wurde angeblich von Albert Einstein im 19. Jahrhundert verfasst. Einstein wird oft der Vermerk zugeschrieben, nur 2 % der Weltbevölkerung seien im Stande, das Rätsel zu

lösen. Es existiert jedoch kein Hinweis auf jedwede Autorenschaft. Hier soll es ein Beispiel für ein Problem darstellen, das mit Prolog lösbar ist.

1. Es gibt fünf Häuser mit je einer anderen Farbe.
2. In jedem Haus wohnt eine Person anderer Nationalität.
3. Jeder Hausbewohner bevorzugt ein bestimmtes Getränk, raucht eine bestimmte Zigarettenmarke und hält ein bestimmtes Haustier.
4. Keine der fünf Personen trinkt das gleiche Getränk, raucht die gleichen Zigaretten oder hält das gleiche Tier wie seine Nachbarn.

Frage: Wem gehört der Fisch?

Hinweise:

- Der Brite lebt im roten Haus.
- Der Schwede hält einen Hund.
- Der Däne trinkt gern Tee.
- Das grüne Haus steht direkt links neben dem weißen Haus.
- Der Besitzer des grünen Hauses trinkt Kaffee.
- Die Person, die Pall Mall raucht, hält einen Vogel.
- Der Mann, der im mittleren Haus wohnt, trinkt Milch.
- Der Besitzer des gelben Hauses raucht Dunhill.
- Der Norweger wohnt im ersten Haus.
- Der Marlboro-Raucher wohnt neben dem, der eine Katze hält.
- Der Mann, der ein Pferd hält, wohnt neben dem, der Dunhill raucht.
- Der Winfield-Raucher trinkt gern Bier.
- Der Norweger wohnt neben dem blauen Haus.
- Der Deutsche raucht Rothmans.
- Der Marlboro-Raucher hat einen Nachbarn, der Wasser trinkt.

Lösung:

Jedes Haus ist eine Liste der Form [Farbe, Nationalitaet, Getraenk, Zigarettenmarke, Haustier].

Zuerst vier einfache Hilfsprädikate zur Listenbearbeitung:

```
erstes(E,[E|_]).
mittleres(M,[_,_,M,_,_]).

links(A,B,[A,B|_]).
links(A,B,[_|R]) :- links(A,B,R).

neben(A,B,L) :- links(A,B,L);links(B,A,L).
```

Lösungsprädikat:

```
run :-
    X = [_,_,_,_,_],                          /* Es gibt (nebeneinander) 5 (noch unl
    member([rot,brite,_,_,_],X),              /* Der Brite lebt im roten Haus */
    member([_,schwede,_,_,hund],X),           /* Der Schwede hält einen Hund */
    member([_,daene,tee,_,_],X),              /* Der Däne trinkt gern Tee */
    links([gruen,_,_,_,_],[weiss,_,_,_,_],X), /* Das grüne Haus steht links vom weil
    member([gruen,_,kaffee,_,_],X),           /* Der Besitzer des grünen Hauses tri
    member([_,_,_,pallmall,vogel],X),         /* Die Person, die Pall Mall raucht, h
    mittleres([_,_,milch,_,_],X),             /* Der Mann, der im mittleren Haus wo
```

```
member([gelb,_,_,dunhill,_],X),                 /* Der Besitzer des gelben Hauses rauc
erstes([_,norweger,_,_,_],X),                   /* Der Norweger wohnt im 1. Haus */
neben([_,_,_,marlboro,_],[_,_,_,_,katze],X),    /* Der Marlboro-Raucher wohnt neben de
neben([_,_,_,_,pferd],[_,_,_,dunhill,_],X),     /* Der Mann, der ein Pferd hält, wohnt
member([_,_,bier,winfield,_],X),                /* Der Winfield-Raucher trinkt gern Bi
neben([_,norweger,_,_,_],[blau,_,_,_,_],X),     /* Der Norweger wohnt neben dem blauen
member([_,deutsche,_,rothmans,_],X),            /* Der Deutsche raucht Rothmans */
neben([_,_,_,marlboro,_],[_,_,wasser,_,_],X),   /* Der Marlboro-Raucher hat einen Nach
member([_,N,_,_,fisch],X),                      /* Der mit der Nationalität N hat eine
write(X),nl,                                    /* Ausgabe aller Häuser */
write('Der '),write(N),write(' hat einen Fisch als Haustier.'),nl.   /* Antwort auf di
```

Definite Clause Grammar

Um Regeln für Parser zu schreiben, haben die meisten Prologsysteme einen Präprozessor implementiert. Er erlaubt es, die Regeln in einer besser lesbaren Form zu notieren, die in der Form den Regeln entsprechen, die verwendet wird, um eine kontextfreie Sprache zu beschreiben. Der Präprozessor ergänzt Platzhalter und erzeugt die oben erwähnten Prolog-Logik-Formeln. Durch Übergabe weiterer Attribute ist es möglich, mit Definite Clause Grammars auch komplexere Sprachen als die kontextfreien zu beschreiben.

Prolog aus logischer Sicht

Ein Prolog-Programm ist eine geordnete Liste so genannter Horn-Klauseln, einer eingeschränkten Form der Prädikatenlogik erster Ordnung. Stellt man dem System eine Anfrage (*Query*), versucht es, diese auf der Grundlage dieser Datenbasis mittels Resolution zu beweisen. Das Ergebnis einer Query ist *yes* oder *no*. Seine eigentliche Wirkung entfaltet ein Prolog-Programm streng genommen durch Nebenwirkungen, die während der Beweissuche auftreten. Also kann ein Prolog-System auch als ein sehr effizienter – wenn auch eingeschränkter – automatischer Theorembeweiser verstanden werden. Die einzige in Prolog eingebaute Suchstrategie bei der Beweisfindung ist Tiefensuche mit Backtracking.

Anwendungsgebiete

In den 1980er Jahren spielte die Sprache eine wichtige Rolle beim Bau von Expertensystemen. Die Sprache wird heute noch in den Bereichen Computerlinguistik und Künstliche Intelligenz verwendet. Zum Beispiel sind Sprachverarbeitungskomponenten des durch seinen Auftritt bei Jeopardy! bekannt gewordenen KI-Systems Watson in Prolog geschrieben.[3] Außerdem gibt es einige kommerzielle Anwendungen im Bereich des Systemmanagements, bei denen asynchrone Ereignisse (Events) mit Hilfe von Prolog oder darauf basierenden proprietären Erweiterungen verarbeitet werden. Ein Beispiel hierzu ist das Produkt *Tivoli Enterprise Console* (TEC) von IBM, das auf IBM-Prolog basiert.

Siehe auch

- Axiom, Deduktion
- Schnittregel
- Erlang (begann als Prolog-Interpreter, auch die Syntax ist davon inspiriert, ebenfalls personelle Nähe, denn der Erfinder von Erlang Joe Armstrong hat am Swedish Institute of Computer Science (SICS) gearbeitet)

Literatur

- Rüdeger Baumann, *Prolog Einführungskurs*, Klett Verlag, 2000, ISBN 3-12-717721-6.
- Patrick Blackburn, Johan Bos, Kristina Striegnitz: *Learn Prolog Now!* College Publications, 2006, ISBN 1-904987-17-6.
- David L. Bowen, Lawrence Byrd, Fernando C. N. Pereira, Luís M. Pereira und David H. D. Warren: *DECsystem-10 Prolog User's Manual.* Occasional Paper 27, 1982. Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland. (ASCII-Download (http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/doc/intro/prolog.doc))
- Hans Kleine Büning, Stefan Schmittgen: *PROLOG: Grundlagen und Anwendungen*. B.G. Teubner, Stuttgart 1986, ISBN 3-519-02484-5
- Ivan Bratko: *Prolog. Programmierung für Künstliche Intelligenz*. Addison-Wesley, Bonn 1987, ISBN 3-925118-63-2.
- William F. Clocksin: *Clause and Effect. Prolog Programming for the Working Programmer*. Springer, Berlin 2005, ISBN 3-540-62971-8.
- William F. Clocksin, Christopher S. Mellish: *Programming in Prolog*. Springer, Berlin 2003, ISBN 3-540-00678-8.
- H. Göhner, B. Hafenbrak: *Arbeitsbuch PROLOG*. DÜMMLER, Bonn 1995, ISBN 3-427-46863-1.
- Richard A. O'Keefe: *The Craft of Prolog*. MIT Press, Cambridge 1990, ISBN 0-262-15039-5.
- Esther König, Roland Seiffert: *Grundkurs PROLOG fur Linguisten*. UTB Linguistik, 1989, ISBN 3-7720-1749-5.
- Leon Sterling, Ehud Shapiro: *The Art of Prolog. Advanced Programming Techniques*. MIT Press, Cambridge 1994, ISBN 0-262-69163-9.
- Gerhard Röhner: *Informatik mit Prolog*. Amt für Lehrerbildung (AfL), 2007, ISBN 3-88327-499-2.
- Uwe Schöning: *Logik für Informatiker*. Spektrum, Heidelberg 2000, ISBN 3-8274-1005-3.
- Wilhelm Weisweber: *Prolog. Logische Programmierung in der Praxis*. Thomson, 1997, ISBN 3-8266-0174-2

Weblinks

Tutorials und Kurse

Wikibooks: Prolog – Lern- und Lehrmaterialien

- Learn Prolog Now! (http://www.learnprolognow.org/) Online-Buch mit Prolog-Einführung, auch für Programmieranfänger verständlich geschrieben. (englisch)
- Prologbeispielsammlung (http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html) (englisch)
- Datenbanken in Prolog. Umsetzung von SQL-Abfragen in Prolog. (http://www.philipphauer.de/info/info/datenbanken-in-prolog/)

Prolog-Implementierungen

- BProlog (http://www.probp.com/) kommerzielles Prolog-System (kostenlos für Bildung und Forschung) mit Erweiterungen zur Constraintprogrammierung (CLP), Nebenläufigkeit und interaktive Graphen
- GNU Prolog (http://www.gprolog.org/) Der freie, quelloffene GNU Prolog Compiler (englisch)
- Jekejeke Prolog (http://www.jekejeke.ch/) ist eine reine Interpreterimplementation von Prolog, die vollständig in Java geschrieben wurde. Die Implementation der Sprache hält sich im Wesentlichen an den ISO Core Standard.
- JIProlog (http://www.ugosweb.com/jiprolog/) ist ein als Shareware erhältlicher kommerzieller Prolog-

Interpreter, der ISO-Kompatibilität anstrebt und in Java (J2SE, J2ME) läuft
- Prolog.NET (http://prolog.hodroj.net) Prolog-Entwicklungsumgebung für das .NET-Framework (englisch)
- SICStus Prolog (http://www.sics.se/isl/sicstuswww/site/index.html) kommerzielles, ISO-kompatibles Prolog-System vom Swedish Institute of Computer Science, erlaubt z. B. Constraintprogrammierung (englisch)
- SWI-Prolog (http://www.swi-prolog.org/) freies, quelloffenes (LGPL), umfassendes ISO-kompatibles Prolog-System mit gutem Community-Support (inkl. Editor, Debugger, Profiler und zahlreichen Programmbibliotheken; englisch)
- tuProlog (http://alice.unibo.it/xwiki/bin/view/Tuprolog/) ist ein freier, quelloffener (LGPL) Interpreter für eine Untermenge von Prolog in Java (J2SE, J2ME) und .NET
- YAP Prolog (http://www.dcc.fc.up.pt/~vsc/Yap/) freier, quelloffener, schneller, ISO-kompatibler Prolog-Interpreter
- C# Prolog (http://sourceforge.net/projects/cs-prolog/) , freier, quelloffener Prolog-Interpreter

An Prolog orientierte logische Programmiersysteme

- CIAO (http://www.ciaohome.org/) frei, quelloffen (LGPL), implementiert ISO Prolog, hat Spracherweiterungen für variable Prädikate (HiLog), constraintbasierte, objektorientierte und nebenläufige Programmierung (englisch)
- ECLiPSe Constraint Programming System (http://www.eclipseclp.org/) frei, quelloffen (MPL), Prolog-basiert mit Erweiterungen für Constraintprogrammierung und zusätzliche Suchstrategien (englisch)
- Logtalk (http://logtalk.org/) ist eine freie, quelloffene (Artistic License 2.0) objektorientierte logische Programmiersprache (englisch)
- Mercury, eine stark an Prolog angelehnte Programmiersprache, vereint Elemente aus der funktionalen und der logischen Programmierung (englisch)
- Poplog (http://www.cs.bham.ac.uk/research/projects/poplog/freepoplog.html) ist eine freie, quelloffene (XFree86-Lizenz) integrierte, interaktive, mehrsprachige Programmierumgebung mit inkrementellen Compilern für die KI-Sprachen POP-11, Prolog, Common Lisp und Standard ML, die nicht nur multiparadigmatisches Programmieren, sondern auch das Mischen dieser Programmiersprachen ermöglicht (englisch)
- QuProlog (http://itee.uq.edu.au/~pjr/HomePages/QuPrologHome.html) – ein erweiterter, freier Prolog-Compiler, der v. a. zum Implementieren interaktiver Theorembeweiser dient (englisch)
- XSB (http://xsb.sourceforge.net/) freies, quelloffenes (LGPL), »fast« ISO-Prolog-kompatibles logisches Programmiersystem mit über Prolog hinausgehenden Spracherweiterungen (HiLog; volle, tabulierte Resolution; erweitertes Pattern Matching) und Bibliotheken für GUI-Programmierung, F-Logic und Ontologie-Verarbeitung (englisch)

Einzelnachweise und Anmerkungen

1. A. Colmerauer und P. Roussel: *The birth of prolog. History of programming languages II*. 1996, S. 331–367. (http://cs305.com/book/programming_languages/Conf-01/HOPLII/p331-colmerauer.pdf) (PDF-Dokument)
2. In Ermangelung einer formalen Spezifikation für *Edinburgh Prolog* wurde meist das *DEC-10 PROLOG Manual* von Bowen u.a. (1982) oder *Programming in Prolog* von Clocksin und Mellish herangezogen.
3. Adam Lally, Paul Fodor (31. März 2011): *Natural Language Processing With Prolog in the IBM Watson System (http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/)* (englisch). The Association for Logic Programming. Abgerufen am

18. Oktober 2001.

 Dieser Artikel wurde in die Liste der lesenswerten Artikel aufgenommen.
