Logische Programmierung: PROLOG

Proseminar Programmiersprachen WS 03/04

von
Jochen Frey

nach einem Thema von
Prof. Dr. Gert Smolka

Lehrstuhl für Programmiersysteme
FR Informatik
Universität des Saarlandes

Inhaltsverzeichnis

1 Einleitung

1.1 Geschichte

Das Prinzip der logischen Programmierung hat seine Ursprünge in dem automatisierten Beweisen mathematischer Sätze. In erster Linie wurde diese Entwicklung innerhalb der Logik vorangetrieben, erst später innerhalb der Informatik. Logische Programmierung basiert auf der Syntax der Prädikatenlogik erster Stufe, die in der zweiten Hälfte des 19. Jahrhunderts von Gottlob Frege aufgestellt und später durch Guiseppe Peano und Bertrand Russell weiterentwickelt wurde.
In den 30er Jahren beschäftigten sich Kurt Gödel und Jacques Herbrand mit der Idee von auf Ableitungen basierenden Berechnungen (Ursprünge von *computation as deduction*) .
1965 veröffentlichte Alan Robinson seine Arbeit (Robinson 1965), welche die Fundamente für automatisierte Ableitungen darstellte. Grundlegende Ideen wurden durch ihn eingeführt. Mit diesen Konzepten war es möglich prädikatenlogische Theoreme zu beweisen. 1974 veröffentlichte Robert Kowalski seine Arbeit (*Predicate Logic as a programming Language*, North-Holland, 1974), in der er beschrieb, wie man in dem oben genannten System Berechnungen anstellen konnte. Zur gleichen Zeit arbeitete Alain Colmerauer an einer Programmiersprache zum Beweisen von Theoremen. Beide arbeiteten von 1971 bis 1973 zusammen, was schließlich zur Beschreibung der Programmiersprache PROLOG führte (1973).
PROLOG kann als praktische Umsetzung der Idee von logischer Programmierung gesehen werden. Ihre Anfänge hatte sie als Anwendung innerhalb der natrlichen Sprachverarbeitung, aber später erkannte man, daß PROLOG auch als eigenständige Programmiersprache benutzt werden konnte.
Das Konzept der logischen Programmierung beeinflusste viele Entwicklungen in der Informatik. So führte sie in den 70er Jahren zur Entwicklung von deduktiven Datenbanken. Ein weiterer Aufschwung dieses Konzepts kam im Zuge des Fifth Generation Projects der Japaner, das auf dem Konzept der logischen Programmierung aufbaute. In letzter Zeit führte die Enwicklung zu *constraint logic programming*.

1.2 Was ist PROLOG?

PROLOG gehört zu der Klasse der deklarativen Programmiersprachen. Programmiersprachen untergliedert man grob in prozedurale und deklarative Programmiersprachen. In prozeduralen oder auch imperativen Programmiersprachen wird vom Benutzer festgelegt, wie ein Problem zu lösen ist, also der genaue algorithmische Ablauf. In deklarative Sprachen hingegen steht das eigentliche Problem im Vordergrund. Der Programmierer beschreibt also, was zu berechnen ist im Gegensatz zu wie. Deklarative Sprachen untergliedert man wiederum in funktionale und logische Programmiersprachen. Beiden Ansätzen liegen mathematische Modelle zugrunde; der λ-Kalkül bei funktionalen Sprachen wie z. B. Haskell oder ML und die Prädikatenlogik erster Stufe bei logischen Programmiersprachen wie PROLOG.

Die eigentliche Programmieraufgabe ist es nun, reale Fakten aus der Umwelt in das PROLOG-System zu übersetzen.

2 Grundlagen

2.1 Terme

Die grundlegende Datenstruktur in PROLOG sind Terme. Es gibt einfache Terme (Atome, Zahlen, Variablen) und zusammengesetzte Terme (Strukturen).

2.1.1 Atome

Atome oder Konstanten sind Zeichenfolgen, die mit Kleinbuchstaben beginnen oder mit Apostrophen eingeschlossen sind.

Beispiel 2.1

```
abraham, isaac, 'Bart', 'Hoomer Simpson'
```

2.1.2 Zahlen

Ganze oder reele Zahlen

2.1.3 Variablen

Variablen sind Zeichenfolgen, die mit Grossbuchstaben oder _ beginnen.

Beispiel 2.3

```
X, _x, Ergebnis, _23
```

2.1.4 Zusammengesetzer Term

Ein zusammengesetzter Term besteht aus einem Symbol und seinen Argumenten. Die Argumente sind selbst wieder Terme. Jeder zusammengesetze Term hat eine feste Stelligkeit, welche die Anzahl der Argumente angibt.

Beispiel 2.4

```
father(abraham,isaac)
male(abraham)
parent(X,milcah)
```

2.2 Klauseln

Ein PROLOG - Programm besteht aus einer Folge von Klauseln. Klauseln sind entweder Fakten, Regeln (oder Anfragen).

2.2.1 Fakten

Ein Fakt ist eine prädikatenlogsiche Aussage und besteht aus einem zusammengesetzten Term, der mit einem Punkt abgeschlossen wird. Alle ohne Vorbedingung wahren Aussagen werden als Fakten bezeichnet.

Beispiel 2.5

```
father(abraham,isaac).
mother(sarah,isaac).
male(abraham).
female(sarah).
```

2.2.2 Regeln

Eine Regel besteht aus einem Regelkopf und einem Regelkörper, die durch das Symbol :- getrennt werden. Auch Regeln stellen Aussagen dar, allerdings unter bestimmten Vorbedingungen. Regeln werden ebenfalls mit einem Punkt beendet.

$$H :- B_1, B_2, \ldots, B_n.$$

Diese Regel kann nun auf zwei Arten interpretiert werden:

- **Deklarative Semantik**: H ist erfüllt, wenn B_1 bis B_n erfüllt werden können.
- **Prozedurale Semantik**: H ist wahr, wenn zuerst B_1, dann B_2, ... , dann B_n erfüllt werden können

Für die Abarbeitung von PROLOG-Programmen wird den Programmklauseln eine prozedurale Semantik unterstellt.

Beispiel 2.6

```
son(isaac,abraham) :- father(abraham,isaac), male(isaac).
daughter(X,Y) :- father(Y,X), female(X).
daughter(X,Y) :- mother(Y,X), female(X).
```

Klauseln mit gleichem Namen und gleicher Stelligkeit werden als Prozedur bezeichnet.

2.2.3 Anfragen

An ein PROLOG-Programm können nun Anfragen gestellt werden. Sie dienen der Ermittlung, ob eine Relation gilt, bzw. ob das Ziel einer Anfrage (Zielklausel) eine logische Konsequenz der Programmklauseln ist. Ist dies der Fall, so antwortet PROLOG mit Yes, andernfalls mit No. Die Anwort No bedeutet nicht, dass die Relation nicht gilt, sondern dass sie nicht aus dem Programm abgeleitet werden kann. (Anm.: Eine Anfrage kann auch als Regel ohne Regelkopf interpretiert werden.)

Beispiel 2.7

```
?- father(abraham,isaac).
Yes
?- female(isaac).
No
?- father(abraham,X), male(X).
X=isaac
?- son(X,abraham).
X=isaac
```

3 Operationale Semantik

Die Aufgabe eines PROLOG-Systems ist es also zu beweisen, ob eine Anfrage des Benutzers in Form einer Zielklausel logisch aus den Programmklauseln ableitbar ist. Enthält die Anfrage Variablen, so muss eine gültige Variablenbelegung gefunden werden.

3.1 Unifikation

Kernstück einer solchen Berechnung ist der Unifikationsalgorithmus. Unter Unfikation versteht man das Angleichen zweier Terme durch gültige Variablensubstitutionen.
Sind t_1 und t_2 Terme gilt also:

t_1 unt t_2 sind unifizierbar, falls es eine Substituion s gibt, mit $s(t_1) = s(t_2)$.

Der Unifikationsalgorithmus sucht die allgemeinste Lösung, d.h. die Substition s, die am wenigsten einschränkend ist. Diese Lösung wird als allgemeinster Unifikator bezeichnet.

Beispiel 3.1: Ein Unifikationsalgorithmus

$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$	ersetzten durch $s_1 = t_1, \ldots, s_n = t_n$
$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_n)$ mit $f \neq g$	Fehler
$x = x$	löschen
$t = x$, wobei t keine Variable ist	ersetzen durch $x = t$
$x = t$, wobei x nicht in t vorkommt	Substitution $x = t$ anwenden
$x = t$, wobei x in t vorkommt und $x \neq t$	Fehler

Beispiel 3.2

- t_1 = father(X,isaac) t_2 = father(abraham,Y)
  $\Longrightarrow$ X=abraham
  $\Longrightarrow$ Y=isaac
- t_1 = Y t_2 = father(X,isaac)
  $\Longrightarrow$ Y=father(X,isaac)
- t_1 = father(haran,lot) t_2 = father(abraham,isaac)
  $\Longrightarrow$ nicht unifizierbar
- t_1 = father(X, isaac) t_2 = mother(sarah, isaac)
  $\Longrightarrow$ nicht unifizierbar

3.2 Resolution

Die Resolution stellt die Grundlage für eine automatische Beweisführung dar. Sie basiert auf dem Prinzip der Unifikation und des automatischen Rücksetzens (Backtracking). Grundsätzlich wird zwischen der Breitensuche und der Tiefensuche unterschieden. PROLOG folgt der prozeduralen Strategie der Tiefensuche. Hierbei spielt die Reihenfolge der Klauseln eine entscheidende Rolle. Die Teilziele einer Anfrage werden von links nach rechts bearbeitet. Zu jedem Teilziel wird die im Programmtext erste Klausel ausgewählt und versucht mit dem Teilziel zu unifizieren. Handelt es sich hierbei um eine Regel, so wird das Teilziel durch den Regelkörper ersetzt und versucht zu beweisen. Andernfalls wird versucht das nächste Ziel der Anfrage herzuleiten.
Tritt während der Resolution ein Fehler bei der Unifikation auf, wird also keine passende Programmklausel gefunden, so springt das PROLOG-System durch das eingebaute Rücksetzen auf den letzten Punkt zurück, an dem eine Entscheidung getroffen wurde, hebt die an dieser Stelle gemachten Variablenbindungen auf und wählt die nächste alternative Klausel aus.
Können alle Ziele einer Anfrage erfolgreich bearbeitet werden, so gibt PROLOG die durchgeführten Variablenbindungen zurück und antwortet mit Yes. Schlägt die Bearbeitung eines Ziels fehl und gibt es keine möglichen Alternativen mehr, so antwortet PROLOG mit No.

Beispiel 3.3

```
father(abraham,isaac).
father(haran,lot).
father(haran,milcah).
father(haran,yiscah).

male(isaac).
male(lot).
female(milcah).
female(yiscah).

son(X,Y) :- father(Y,X), male(X).
daughter(X,Y) :- father(Y,X), female(X).
```

An dieses Programm wird nun folgende Anfrage gestellt:

```
?- daughter(X,haran).
```

1. *PROLOG sucht passende Programmklausel*
   Unfikation `daughter(X,Y)` *und* `daughter(X,haran)`
   $\Longrightarrow$ *Substitution Y=haran*
2. *Das ursprüngliche Ziel wird durch den Regelkörper ersetzt*
   $\Longrightarrow$ *neue Zielanfrage:* `?- father(haran,X), female(X).`
3. *linkes Teilziel wird ausgewählt:* `father(haran,X)`
   erste Programmklausel wird ausgewählt: `father(abraham,isaac).`
   $\Longrightarrow$ *Unifikation nicht möglich* $\Longrightarrow$ *Backtracking*

4. *nächste alternative Programmklausel wird ausgewählt:* `father(haran,lot)`
   $\Longrightarrow$ *Substitution X=lot*
5. *rechtes Teilziel wird ausgewählt und Substitution angewendet:* `female(lot)`
   $\Longrightarrow$ *keine passende Programmklausel vorhanden* $\Longrightarrow$ *Backtracking*
6. *Substitution X=lot wird aufgehoben, nächste Klausel im Programmtext ausgewählt:* `father(haran,milcah)`
   $\Longrightarrow$ *Substitution X=milcah*
7. *rechtes Teilziel wird ausgewählt und Substitution angewendet:* `female(milcah)`
   $\Longrightarrow$ *passende Programmklausel vorhanden*
   $\Longrightarrow$ *PROLOG antwortet mit Yes und gibt die Substitution X=milcah aus*
8. *Durch Eingabe von ; wird manuell Backtracking erzwungen um alternative Lösungen zu finden*
9. *Substitution X=milcah wird aufgehoben, nächste Klausel im Programmtext ausgewählt:* `father(haran,yiscah)`
   $\Longleftrightarrow$ *Substitution X=yiscah*
10. *rechtes Teilziel wird ausgewählt und Substitution angewendet:* `female(yiscah)`
    $\Longrightarrow$ *passende Programmklausel vorhanden*
    $\Longrightarrow$ *PROLOG antwortet mit Yes und gibt die Substitution X=yiscah aus*
11. *weitere Eingabe von ; führt zur Ausgabe von No, da keine Alternativen mehr vorhanden sind*

4 Listen

4.1 Grundlagen

Eine Liste ist ein zusammengesetzter Term, der eine Folge von Elementen enthält, die wieder Terme sind. Eine Liste setzt sich zusammen aus einem Listenkopf und einem Listenrumpf, der wieder ein Liste ist. Folgende Notationen von Listen sind in PROLOG alle gleichwertig zu benutzen:

Interner Term	cons-Syntax	Standardsyntax
.(a, [])	[a | []]	[a]
.(a, .(b, []))	[a | [b | []]]	[a,b]
.(a, .(b, .(c, [])))	[a | [b | [c | []]]]	[a,b,c]

4.2 Unifikation von Listen

Die einzelnen Elemente der Listen werden paarweise unifiziert. Beide Listen müssen gleich lang sein.

Beispiel 4.1

- t_1 `= [a,b,c]` t_2 `= [X|Y]`
  $\Longrightarrow$ `X=a`
  $\Longrightarrow$ `Y=[b,c]`

- t_1 = [a,b,c] t_2 = [X,Y|Z]
  $\Longrightarrow$ X=a
  $\Longrightarrow$ Y=b
  $\Longrightarrow$ Z=[c]
- t_1 = [a,X] t_2 = [b,c]
  $\Longrightarrow$ nicht unifizierbar
- t_1 = [a] t_2 = [X|Y]
  $\Longrightarrow$ X=a
  $\Longrightarrow$ Y=[]

4.3 Operationen auf Listen

Zwei grundlegende Operationen auf Listen sind *member* und *append*.

Beispiel 4.2: member

```
member(X, [X|Xs]).
member(X, [Y|Ys]) :- member(X, Ys).
```

- ?- member(b, [a,b,c]).
  Yes
- ?- member(X, [1,2,3]).
  X=1 ;
  X=2 ;
  X=3 ;
  No

Beispiel 4.3: append

```
append([], Ys, Ys).
append([X|Xs], YS, [X|Zs]) :- append(Xs, Ys, Zs).
```

- ?- append([a,b],[c,d], [a,b,c,d]).
  Yes
- ?- append([a,b],[1,2,3], X).
  X=[a,b,1,2,3] ;
  No
- ?- append(X,[3,4],[1,2,3,4]).
  X=[1,2] ;
  No

Mit *append* lässt sich leicht eine Operation schreiben, die das letzte Element einer Liste angibt.

Beispiel 4.4: last

```
last(List, Last) :- append(_, [Last], List).
```

5 Arithmetik

5.1 Grundlagen

Prolog bietet u.a. folgende arithmetische Operatoren:

+, - , * , / , mod

Im Gegensatz zu anderen Programmiersprachen werden in PROLOG arithmetische Ausdrücke nicht sofort evaluiert, sondern sind gewöhnliche zusammengesetzte Terme.

Beispiel 5.1

```
?- X = 2 + 3.
X= 2+3
```

5.2 Evaluation arithmetischer Ausdrücke

Um arithmetisch Ausdrücke explizit zu berechnen, gibt es in PROLOG den Infix-Operator *is*, welcher:

- eine Variable und einen Term als Argumente nimmt,
- den Term berechnet und an die Variable bindet.

Im rechten Argument dürfen keine ungebundenen Variablen vorkommen, da sonst ein Fehler auftritt.

Beispiel 5.2

- ```
  ?- X is 2 +3.
  X=5
  ```
- ```
  ?- X is Y + 3.
  Fehler
  ```

5.3 Vergleichsprädikate

Prolog bietet folgende Vergleichsprädikate:

$<, >, =<, =>, =:=, =\=$

Die Vergleichsprädikate evaluieren ebenfalls ihre Argumente. Es gilt aber auch die Einschränkung, dass keines der Argumente ungebundene Variablen enthalten darf.

Beispiel 5.3

- ```
  ?- 6 * 2 =:= 3 * 4.
  Yes
  ```
- ```
  ?- 2 + 3 > 5.
  No
  ```
- ```
  ?- X > 2.
  Fehler
  ```
  ```

5.4 Probleme

Die oben angesprochene Problematik, dass durch ungebunde Variablen Fehler bei der Berechnung von arithmetsichen Ausdrücken auftreten können, führt dazu, dass keine deklarative Semantik mehr besteht, d. h. der Programmier ist gezwungen über die Resolution nachzudenken, da eine falsche Reihenfolge von Zielanfragen zu Fehlern führt.

Beispiel 5.4

- `?- Y = 2, X is Y + 3.`
  `X=5`
- `?- X is Y+3, Y = 2.`
  `Fehler`

6 Kontrollfluss

Betrachen wir folgendes Beispiel:

Beispiel 6.1

```
if_then_else(P, Q, R) :- P, Q.
if_then_else(P, Q, R) :- not P, R.
```

Diese Lösung kann als ineffizient bezeichnet werden, da unter Umständen unnötige Berechnungen durchgeführt werden. Nehmen wir an, die erste Klausel wird ausgewählt und P kann hergeleitet werden. Die Berechnung wird nun mit Q fortgesetzt. Schlägt Q fehl, kommt es zu Backtracking und es wird versucht P unter einer alternativen Berechnung erneut herzuleiten, obwohl wir schon wissen, dass P bewiesen werden kann.
Schlägt andernfalls P fehl, so wird die unter Klausel ausgewählt und die Berechnung mit not(P) fortgesetzt, was ebenfalls schon bewiesen wurde.

6.1 Cut

Um diese Problematik zu behandeln gibt es in PROLOG den Cut-Operator *"!"*. Der Cut-Operator ist immer wahr und verhindert ein Backtracking, d. h. für sämtliche Ziele vor dem Cut werden keine Lösungen mehr gesucht und es können keine alternativen Klauseln ausgewählt werden. Die Variablenbelegungen vor dem Cut können nicht mehr aufgehoben werden.

Beispiel 6.2

```
if_then_else(P, Q, R) :- P, !, Q.
if_then_else(P, Q, R) :- R.
```

Durch den Cut-Operator kann eine korrekte und effiziente Implementierung von *if_then_else* erzielt werden. Wird z. B. die erste Klausel ausgewählt und P erfolgreich bewiesen, springt die Kontrolle zum Cut und dann zu Q. Schlägt Q nun fehl wird kein Backtracking mehr durchgeführt und die Anfrage mit No beantwortet. Ist P allerdings nicht herleitbar, wird die zweite Klausel ausgewählt und die Berechnung mit R fortgesetzt.

Die Idee des Cut ist es also die Suchbäume zu stutzen, um unnötige Berechnungen zu verhindern.

Man unterscheidet:

- grüne Cuts:
  - schneiden Suchbäume weg, die keine Lösungen enthalten
  - verändern die deklarative Bedeutung des Programms nicht
  - können weggelassen werden
- rote Cuts:
  - schneiden Suchbäume weg, die Lösungen enthalten
  - verändern die deklarative Bedeutung des Programms
  - können nicht weggelassen werden

6.2 Negation

PROLOG bietet dem Anwender das Prädikat *not*, um eine Negation von Aussagen oder Anfragen auszudrücken. *Not* sollte hierbei nicht als die logische Verneinung aufgefasst werden, sondern eher als Fehlschlagen einer Berechnung. Die Anfrage `?- not(X)` gilt dann als bewiesen, wenn X fehlschlägt. Falls X allerdings herleitbar ist, dann schlägt `not(X)` fehl.

Not kann mit dem Cut-Operator wie folgt implementiert werden:

Beispiel 6.3

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

Beispiel 6.4

- ```
  ?- not(3 < 2).
  Yes
  ```
- ```
  ?- not(2 < 3).
  No
  ```

6.3 Probleme

Tritt eine ungebundene Variable im Argument von *not* auf, so führt dies auch hier zu Problemen. Betrachten wir folgende Anfragen:

Beispiel 6.5

- ```
  ?- member(X,[1,2,3,4], not(X=3).
  X=1 ;
  X=2 ;
  X=4
  ```
- ```
  ?- not(X=3), member(X, [1,2,3,4]).
  No
  ```

Durch die Vertauschung beider Ziele kommt es zu unterschiedlichen Lösungen.Wird nämlich zuerst versucht `not(X=3)` zu bearbeiten, wird dies immer fehlschlagen, da X ungebunden ist und somit X=3 immer unifiziert werden kann.
Auch hier ist der Programmierer gezwungen über die Resolution nachzudenken, da unterschiedliche Reihenfolgen der Zielklauseln zu unterschiedlichen Lösungen führen.

Anmerkung:

Prolog2 stellt ein Prädikat zur Verfügung, mit dem es möglich ist die Problematik mit ungebundenen Variablen zu beheben.
freeze(X, Goal) verzögert die Ausführung von Goal bis die Variable X gebunden wird.

7 Beispiel: N-Damen-Problem

7.1 Problemstellung

Auf eine N x N - Schachbrett sollen N Damen so angeordnet werden, dass sie sich nicht gegenseitig schlagen können.

7.2 Lösungsstrategie

Wir gehen davon aus, dass in jeder Spalte und Zeile jeweils immer nur eine Dame steht. Wir bestimmen uns also eine Liste von [1, 2, ... n], wobei das erste Element die Zeile angibt, in der in der ersten Spalte eine Dame steht, das zweite Element die Zeile der zweiten Spalte, usw. . Jetzt bestimmen wir alle Permutationen dieser Liste und prüfen, ob diese Permutation sicher ist, d. h. sich die Damen diagonal nicht schlagen können. Ist dies der Fall, so haben wir eine mögliche Lösung gefunden, andernfalls wird durch Backtracking eine neue Permutation gewählt.

7.3 Implementierung in PROLOG

```
queens(N,Qs) :- range(1,N,Ns), permutation(Ns,Qs), safe(Qs).


range(M,N,[M|Ns]) :- M < N, M1 is M+1, range(M1,N,Ns).
range(N,N,[N]).
```

```
permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).

safe([Q|Qs]) :- safe(Qs), not(attack(Q,Qs)).
safe([]).

attack(X,Xs) :- attack(X,1,Xs).

attack(X,N,[Y|Ys]) :- X is Y+N.
attack(X,N,[Y|Ys]) :- X is Y-N.
attack(X,N,[Y|Ys]) :- N1 is N+1, attack(X,N1,Ys).
```

8 Zusammenfassung

PROLOG stellt ein Programmiersprachenkonzept dar, welches sich grundlegend von anderen unterscheidet. Es handelt sich um deklaratvie Progammierung, basierend auf Prädikatenlogik erster Ordnung. Der Programmierer kann sich hierbei ganz dem eigentlichen Probelm widmen, ohne sich Gedanken über Typen oder den genauen algorithmischen Ablauf machen zu müssen. PROLOG eignet sich daher besonders auf dem Gebiet des *Rapid Prototyping*, da sehr schnell effiziente Programme implementiert werden können. PROLOG bietet effiziente Techniken für Unifikation und Rücksetzen und stellt somit einen zuverlässigen Suchmechanismus zur Verfügung.

PROLOG fehlt es allerdings an bestimmten Elementen zur Strukturierung, wie z. B. Modulen oder die Möglichkeit der Datenkapselung. Dies kann in größeren Projekten zu Unübersichtlichkeit und Programmierfehlern führen. Weitere Schwierigkeiten entstehen durch die problematische Arithmetik sowie spezifische Kontrollmechanismen, wie z. B. den Cut-Operator.

Abschließend sei zu bemerken, dass PROLOG sich besonders für Probleme eignet, die auf logischen Zusammenhängen basieren. Mögliche Anwendungen findet PROLOG daher z. B. auf dem Gebiet der Künstlichen Intelligenz und der Computerlinguistik.

Literatur

[1] Sterling, L.; Shapiro, E.: *The Art of Prolog*. 1. Aufl., MIT, 1986

[2] Mitchell, J.C.: *Concepts in Programming Languages*. 1.Aufl., Cambridge University Press, 2003

[3] Kowalski, R.: Algorithm = Logic + Control, *Communication of the ACM* 22, pp.424-436, 1979