Programmmierkurs Prolog

Sommersemester 2002

Ralf Klinkenberg

Universität Dortmund

Programmierkurs Prolog im Sommersemester 2002

Termin: Mo., 22.07., – Fr., 02.08.2002,
täglich von Mo.–Fr.

Uhrzeit: 9:00–12:00 (Vorlesung) und
13:00–14:30 (Übung) und
ab 14:30 Uhr eigenständige Arbeit im Pool oder zuhau

Ort: GB IV / Raum 113,
Ausnahme: 29.+30.06.2002: GB V / Raum 420

Pool: GB V / Raum U 010

WWW: `http://www-ai.cs.uni-dortmund.de/LEHRE/PROLOG/`

Veranstalter:

Ralf Klinkenberg, Uni Dortmund, LS Informatik VIII

Tel.: 0231/755–5103

EMail: klinkenberg@ls8.cs.uni-dortmund.edu

pkpro000@cs.uni-dortmund.de

Themen des Programmierkurses Prolog (1):

- Theoretische Grundlagen: **Prädikatenlogik**
- **Programmierung in Prolog**: Einführung, Erste Schritte, Syntax, Ausführungsmodell
- Arithmetik, Rekursion, Strukturen, Bäume, Listen, Backtracking, der Cut
- Ein- & Ausgabe, Systemprädikate, Modulsystem, Datenstrukturen und Algorithmen, Debugger & Compiler (1), Entwurf von Prolog-Programmen
- Problemlösen als Suche (Suchprobleme), Spielprobleme

Themen des Programmierkurses Prolog (1):

- Theoretische Grundlagen: **Prädikatenlogik**
- **Programmierung in Prolog**: Einführung, Erste Schritte, Syntax, Ausführungsmodell
- Arithmetik, Rekursion, Strukturen, Bäume, Listen, Backtracking, der Cut
- Ein- & Ausgabe, Systemprädikate, Modulsystem, Datenstrukturen und Algorithmen, Debugger & Compiler (1), Entwurf von Prolog-Programmen
- Problemlösen als Suche (Suchprobleme), Spielprobleme

Themen des Programmierkurses Prolog (1):

- Theoretische Grundlagen: **Prädikatenlogik**
- **Programmierung in Prolog**: Einführung, Erste Schritte, Syntax, Ausführungsmodell
- Arithmetik, Rekursion, Strukturen, Bäume, Listen, Backtracking, der Cut
- Ein- & Ausgabe, Systemprädikate, Modulsystem, Datenstrukturen und Algorithmen, Debugger & Compiler (1), Entwurf von Prolog-Programmen
- Problemlösen als Suche (Suchprobleme), Spielprobleme

Themen des Programmierkurses Prolog (1):

- Theoretische Grundlagen: **Prädikatenlogik**
- **Programmierung in Prolog**: Einführung, Erste Schritte, Syntax, Ausführungsmodell
- Arithmetik, Rekursion, Strukturen, Bäume, Listen, Backtracking, der Cut
- Ein- & Ausgabe, Systemprädikate, Modulsystem, Datenstrukturen und Algorithmen, Debugger & Compiler (1), Entwurf von Prolog-Programmen
- Problemlösen als Suche (Suchprobleme), Spielprobleme

Themen des Programmierkurses Prolog (1):

- Theoretische Grundlagen: **Prädikatenlogik**
- **Programmierung in Prolog**: Einführung, Erste Schritte, Syntax, Ausführungsmodell
- Arithmetik, Rekursion, Strukturen, Bäume, Listen, Backtracking, der Cut
- Ein- & Ausgabe, Systemprädikate, Modulsystem, Datenstrukturen und Algorithmen, Debugger & Compiler (1), Entwurf von Prolog-Programmen
- Problemlösen als Suche (Suchprobleme), Spielprobleme

Themen des Programmierkurses Prolog (2):

- Meta-Prädikate, Meta-Programmierung, Meta-Interpreter
- Expertensysteme, Debugger & Compiler (2)
- Definite Clause Grammars (DCGs)
- Constraint Logic Programming (CLP)

Themen des Programmierkurses Prolog (2):

- Meta-Prädikate, Meta-Programmierung, Meta-Interpreter
- Expertensysteme, Debugger & Compiler (2)
- Definite Clause Grammars (DCGs)
- Constraint Logic Programming (CLP)

Themen des Programmierkurses Prolog (2):

- Meta-Prädikate, Meta-Programmierung, Meta-Interpreter
- Expertensysteme, Debugger & Compiler (2)
- Definite Clause Grammars (DCGs)
- Constraint Logic Programming (CLP)

Themen des Programmierkurses Prolog (2):

- Meta-Prädikate, Meta-Programmierung, Meta-Interpreter
- Expertensysteme, Debugger & Compiler (2)
- Definite Clause Grammars (DCGs)
- Constraint Logic Programming (CLP)

Bratko, Ivan (2001). *Prolog Programming for Artificial Intelligence*. Addison-Wesley Longman, Amsterdam, 3. Auflage, ISBN 0-201-40375-7.

Clocksin, W.F. und Mellish, C.S. (1994). *Programming in Prolog*. Springer-Verlag, 4. Auflage.

O'Keefe, R. (1990). *The Craft of Prolog*. MIT press, Cambridge, MA, USA und London, GB.

Sterling, L. und Shapiro, E. (1994). *The Art of Prolog*. MIT Press, 2. Auflage.

Bratko, Ivan (2001). *Prolog Programming for Artificial Intelligence*. Addison-Wesley Longman, Amsterdam, 3. Auflage, ISBN 0-201-40375-7.

Clocksin, W.F. und Mellish, C.S. (1994). *Programming in Prolog*. Springer-Verlag, 4. Auflage.

O'Keefe, R. (1990). *The Craft of Prolog*. MIT press, Cambridge, MA, USA und London, GB.

Sterling, L. und Shapiro, E. (1994). *The Art of Prolog*. MIT Press, 2. Auflage.

Bratko, Ivan (2001). *Prolog Programming for Artificial Intelligence*. Addison-Wesley Longman, Amsterdam, 3. Auflage, ISBN 0-201-40375-7.

Clocksin, W.F. und Mellish, C.S. (1994). *Programming in Prolog*. Springer-Verlag, 4. Auflage.

O'Keefe, R. (1990). *The Craft of Prolog*. MIT press, Cambridge, MA, USA und London, GB.

Sterling, L. und Shapiro, E. (1994). *The Art of Prolog*. MIT Press, 2. Auflage.

Bratko, Ivan (2001). *Prolog Programming for Artificial Intelligence*. Addison-Wesley Longman, Amsterdam, 3. Auflage, ISBN 0-201-40375-7.

Clocksin, W.F. und Mellish, C.S. (1994). *Programming in Prolog*. Springer-Verlag, 4. Auflage.

O'Keefe, R. (1990). *The Craft of Prolog*. MIT press, Cambridge, MA, USA und London, GB.

Sterling, L. und Shapiro, E. (1994). *The Art of Prolog*. MIT Press, 2. Auflage.

Brockhausen, Peter (2000). *Programmierkurs Prolog*. Uni Dortmund.

Joachims, Thorsten und Lehmke, Stephan (1998). *Programmierkurs Prolog*. Uni Dortmund.

Joachims, Thorsten und Markof, Ingolf (1997). *Programmierkurs Prolog*. Uni Dortmund.

Lehmke, Stephan (2002). *Vorlesung Intelligente Systeme*. FH Gelsenkirchen.

Brockhausen, Peter (2000). *Programmierkurs Prolog*. Uni Dortmund.

Joachims, Thorsten und Lehmke, Stephan (1998). *Programmierkurs Prolog*. Uni Dortmund.

Joachims, Thorsten und Markof, Ingolf (1997). *Programmierkurs Prolog*. Uni Dortmund.

Lehmke, Stephan (2002). *Vorlesung Intelligente Systeme*. FH Gelsenkirchen.

Brockhausen, Peter (2000). *Programmierkurs Prolog*. Uni Dortmund.

Joachims, Thorsten und Lehmke, Stephan (1998). *Programmierkurs Prolog*. Uni Dortmund.

Joachims, Thorsten und Markof, Ingolf (1997). *Programmierkurs Prolog*. Uni Dortmund.

Lehmke, Stephan (2002). *Vorlesung Intelligente Systeme*. FH Gelsenkirchen.

Brockhausen, Peter (2000). *Programmierkurs Prolog*. Uni Dortmund.

Joachims, Thorsten und Lehmke, Stephan (1998). *Programmierkurs Prolog*. Uni Dortmund.

Joachims, Thorsten und Markof, Ingolf (1997). *Programmierkurs Prolog*. Uni Dortmund.

Lehmke, Stephan (2002). *Vorlesung Intelligente Systeme*. FH Gelsenkirchen.

Gliederung

Zum Nachlesen und Vertiefen:

Schöning, Uwe (1987–1995). *Logik für Informatiker*. B·I Wissenschaftsverlag.

Thiele, Helmut (1991–1996). Skript zur Vorlesung *Logische Systeme der Informatik*. Universität Dortmund.

Wagner, Hubert (1996–2001). Skript zur Vorlesung *Logische Systeme der Informatik*. Universität Dortmund.

2.1.1 Elemente der mathematischen Logik

Syntax: Sprache logischer Ausdrücke.

Semantik: Interpretation logischer Ausdrücke —
wann ist ein Ausdruck gültig, wann ungültig?

Modellbegriff: Was macht einen Ausdruck gültig?

Semantische Folgerung:
Was folgt aus einer Menge von Annahmen?

Formales Ableiten: Wie kann man die semantische Folgerung durch regelbasiertes Schließen charakterisieren?

2.1.1 Elemente der mathematischen Logik

Syntax: Sprache logischer Ausdrücke.

Semantik: Interpretation logischer Ausdrücke —
wann ist ein Ausdruck gültig, wann ungültig?

Modellbegriff: Was macht einen Ausdruck gültig?

Semantische Folgerung:
Was folgt aus einer Menge von Annahmen?

Formales Ableiten: Wie kann man die semantische Folgerung durch regelbasiertes Schließen charakterisieren?

2.1.1 Elemente der mathematischen Logik

Syntax: Sprache logischer Ausdrücke.

Semantik: Interpretation logischer Ausdrücke —
wann ist ein Ausdruck gültig, wann ungültig?

Modellbegriff: Was macht einen Ausdruck gültig?

Semantische Folgerung:
Was folgt aus einer Menge von Annahmen?

Formales Ableiten: Wie kann man die semantische Folgerung durch regelbasiertes Schließen charakterisieren?

2.1.1 Elemente der mathematischen Logik

Syntax: Sprache logischer Ausdrücke.

Semantik: Interpretation logischer Ausdrücke —
wann ist ein Ausdruck gültig, wann ungültig?

Modellbegriff: Was macht einen Ausdruck gültig?

Semantische Folgerung:
Was folgt aus einer Menge von Annahmen?

Formales Ableiten: Wie kann man die semantische Folgerung durch regelbasiertes Schließen charakterisieren?

2.1.1 Elemente der mathematischen Logik

Syntax: Sprache logischer Ausdrücke.

Semantik: Interpretation logischer Ausdrücke —
wann ist ein Ausdruck gültig, wann ungültig?

Modellbegriff: Was macht einen Ausdruck gültig?

Semantische Folgerung:
Was folgt aus einer Menge von Annahmen?

Formales Ableiten: Wie kann man die semantische Folgerung durch regelbasiertes Schließen charakterisieren?

PS Menge von Paaren Prädikatensymbol/Arität (Stellenz.).

Beispiel 2.2.1 Prädikatensymbol `member` mit Arität 2: $\boxed{\texttt{member}/2 \in \textbf{PS}}$.

PS Menge von Paaren Prädikatensymbol/Arität (Stellenz.).

Beispiel 2.2.1 Prädikatensymbol `member` mit Arität 2: $\boxed{\texttt{member/2} \in \mathbf{PS}}$.

FS Menge von Paaren Funktionssymbol/Arität.

Beispiel 2.2.2 Funktionssymbol `div` mit Arität 2: $\boxed{\texttt{div/2} \in \mathbf{FS}}$.

Spezialfall Arität $= 0$: Individuenkonstante (Z. B. `pi`).

Definition 2.2.1 (Individuenvariablen und Terme)

1. *Wir fixieren eine Menge* **VAR** *von Individuenvariablen.*
2. *Individuenvariablen und Individuenkonstanten sind Terme.*
3. *Gilt* $f/n \in$ **FS** *mit* $n \geq 1$ *und sind* $T_1, \ldots, T_n$ *Terme, dann ist* $\boxed{f(T_1, \ldots, T_n)}$ *ein Term.*

Definition 2.2.2 (Ausdrücke)

1. *Wenn* $\mathbf{p}/0 \in \mathbf{PS}$*, so ist* $\boxed{\mathbf{p}}$ *ein Ausdruck.*
2. *Gilt* $\mathbf{p}/\mathbf{n} \in \mathbf{PS}$ *mit* $\mathbf{n} \geq 1$ *und sind* $\mathbf{T_1}, \ldots, \mathbf{T_n}$ *Terme, dann ist* $\boxed{\mathbf{p}(\mathbf{T_1}, \ldots, \mathbf{T_n})}$ *ein Ausdruck.*
3. *Sind* $\mathbf{A}$ *und* $\mathbf{B}$ *Ausdrücke, so auch* $\boxed{\neg \mathbf{A}}$, $\boxed{(\mathbf{A} \vee \mathbf{B})}$.
4. *Ist* $\mathbf{V} \in \mathbf{VAR}$ *und ist* $\mathbf{A}$ *ein Ausdruck, so auch* $\boxed{\forall \mathbf{V}\ \mathbf{A}}$.

Die Menge aller Ausdrücke: **AUSD**.

Ausdrücke nach 1 und 2 heißen atomar.

Zusätzliche Operatoren:

$$(\mathbf{A} \rightarrow \mathbf{B}) =_{\text{def}} (\neg \mathbf{A} \vee \mathbf{B})$$

$$(\mathbf{A} \wedge \mathbf{B}) =_{\text{def}} \neg(\neg \mathbf{A} \vee \neg \mathbf{B})$$

$$(\mathbf{A} \leftrightarrow \mathbf{B}) =_{\text{def}} ((\mathbf{A} \rightarrow \mathbf{B}) \wedge (\mathbf{B} \rightarrow \mathbf{A}))$$

$$\exists \mathbf{V}\, \mathbf{A} =_{\text{def}} \neg \forall \mathbf{V} \neg \mathbf{A}$$

Zusätzliche Operatoren:

$$(\mathbf{A} \rightarrow \mathbf{B}) =_{\text{def}} (\neg \mathbf{A} \vee \mathbf{B})$$

$$(\mathbf{A} \wedge \mathbf{B}) =_{\text{def}} \neg(\neg \mathbf{A} \vee \neg \mathbf{B})$$

$$(\mathbf{A} \leftrightarrow \mathbf{B}) =_{\text{def}} ((\mathbf{A} \rightarrow \mathbf{B}) \wedge (\mathbf{B} \rightarrow \mathbf{A}))$$

$$\exists \mathbf{V}\, \mathbf{A} =_{\text{def}} \neg \forall \mathbf{V} \neg \mathbf{A}$$

Klammersparung:

$\leftrightarrow$ trennt stärker als $\rightarrow$, $\vee$ und $\wedge$

$\rightarrow$ trennt stärker als $\vee$ und $\wedge$

$\vee$ trennt stärker als $\wedge$

Beispiel 2.2.3

$$\mathbf{PS} = \{\mathtt{p/2}\}, \quad \mathbf{FS} = \emptyset$$

(2.1) $\mathtt{p}$ reflexiv

$$\forall \mathtt{X}\, \mathtt{p(X,X)}$$

(2.2) $\mathtt{p}$ symmetrisch

$$\forall \mathtt{X}\, \forall \mathtt{Y}\, (\mathtt{p(X,Y)} \rightarrow \mathtt{p(Y,X)})$$

(2.3) $\mathtt{p}$ transitiv

$$\forall \mathtt{X}\, \forall \mathtt{Y}\, \forall \mathtt{Z}\, (\mathtt{p(X,Y)} \wedge \mathtt{p(Y,Z)} \rightarrow \mathtt{p(X,Z)})$$

(2.3) ohne Abkürzungen:

$$\forall \mathtt{X}\, \forall \mathtt{Y}\, \forall \mathtt{Z}\, (\neg\neg(\neg \mathtt{p(X,Y)} \vee \neg \mathtt{p(Y,Z)}) \vee \mathtt{p(X,Z)})$$

Definition 2.3.1 (Interpretation)

Ein Quintupel $\boxed{\mathfrak{J} = [\mathbf{PS}, \mathbf{FS}, \mathfrak{U}, \mathbf{\Phi}, \mathbf{\Pi}]}$ *heiße Interpretation*

$=_{\text{def}}$ 1. $\mathfrak{U}$ *ist eine nicht-leere Menge (Individuenbereich, Universum).*

2. $\mathbf{\Phi}$ *ordnet jedem Paar* $\mathbf{f/n} \in \mathbf{FS}$ *eine Abbildung* $\boxed{\mathbf{\Phi}_{\mathbf{f/n}} : \mathfrak{U}^{\mathfrak{n}} \to \mathfrak{U}}$ *zu; ist* $\mathfrak{n} = 0$, *so gilt:* $\mathbf{\Phi}_{\mathbf{f/n}} \in \mathfrak{U}$.

3. $\mathbf{\Pi}$ *ordnet jedem Paar* $\mathbf{p/n} \in \mathbf{PS}$ *eine Teilmenge* $\boxed{\mathbf{\Pi}_{\mathbf{p/n}} \subseteq \mathfrak{U}^{\mathfrak{n}}}$ *zu; ist* $\mathfrak{n} = 0$, *so gilt* $\mathbf{\Pi}_{\mathbf{p/n}} \subseteq \{\varnothing\}$.

Definition 2.3.2 (Interpretation der Terme)

σ *heiße* $\mathfrak{J}$*-Zustand*

$=_{\text{def}}$ $\sigma : \mathbf{VAR} \rightarrow \mathfrak{U}$

$\mathbf{EL}(\mathsf{T}, \mathfrak{J}, \sigma)$ *ordnet dem Term* T *das durch* $\mathfrak{J}$ *im* $\mathfrak{J}$*-Zustand* σ *festgelegte Individuum zu.*

$\mathbf{EL}(\mathsf{V}, \mathfrak{J}, \sigma) =_{\text{def}} \boxed{\sigma(\mathsf{V})}$, *falls* $\mathsf{V} \in \mathbf{VAR}$

$\mathbf{EL}(\mathsf{f}, \mathfrak{J}, \sigma) =_{\text{def}} \boxed{\Phi_{\mathsf{f}}}$, *falls* $\mathsf{f}/0 \in \mathbf{FS}$

$\mathbf{EL}(\mathsf{f}(\mathsf{T}_1, \ldots, \mathsf{T}_n), \mathfrak{J}, \sigma)$

$=_{\text{def}} \boxed{\Phi_{\mathsf{f}/n}(\mathbf{EL}(\mathsf{T}_1, \mathfrak{J}, \sigma), \ldots, \mathbf{EL}(\mathsf{T}_n, \mathfrak{J}, \sigma))}$,

falls $\mathsf{f}/n \in \mathbf{FS}$ *und* $n \geq 1$.

$$\sigma \langle V := \xi \rangle (W) =_{\text{def}} \begin{cases} \sigma(W), & \text{falls } W \neq V \\ \xi, & \text{falls } W = V \end{cases}$$

Definition 2.3.3 (Erfüllungsdefinition)

$$\sigma, \mathfrak{I} \models p =_{\text{def}} \boxed{\Pi_p = \{\varnothing\}}$$

$$\sigma, \mathfrak{I} \models q(T_1, \ldots, T_n)$$

$$=_{\text{def}} \boxed{[\mathrm{EL}(T_1, \mathfrak{I}, \sigma), \ldots, \mathrm{EL}(T_n, \mathfrak{I}, \sigma)] \in \Pi_q}$$

$\sigma, \mathfrak{I} \models \neg A =_{\text{def}}$ *Es gilt nicht* $\boxed{\sigma, \mathfrak{I} \models A}$

$$\sigma, \mathfrak{I} \models (A \vee B)$$

$=_{\text{def}}$ $\boxed{\sigma, \mathfrak{I} \models A \text{ oder } \sigma, \mathfrak{I} \models B}$

$$\sigma, \mathfrak{I} \models \forall V A$$

$=_{\text{def}}$ *Für jedes* $\xi \in \mathfrak{U}$ *gilt:* $\boxed{\sigma\langle V := \xi\rangle, \mathfrak{I} \models A}$

Definition 2.3.4 (Allgemeingültig)

1. *In* $\mathfrak{I}$ *ist* $\mathbf{A}$ *allgemeingültig* ($\mathfrak{I} \models \mathbf{A}$)
   $=_{\text{def}}$ *Für jeden* $\mathfrak{I}$*-Zustand* $\boldsymbol{\sigma}$ *gilt:* $\boldsymbol{\sigma}, \mathfrak{I} \models \mathbf{A}$.
2. $\mathbf{A}$ *ist (schlechthin) allgemeingültig* ($\models \mathbf{A}$)
   $=_{\text{def}}$ *Für jede Interpretation* $\mathfrak{I}$ *gilt:* $\mathfrak{I} \models \mathbf{A}$.

Beispiel 2.3.1

$$\mathbf{A} =_{\text{def}} \forall \mathtt{X}\, \mathtt{p}(\mathtt{X},\mathtt{X})$$

$$\mathbf{B} =_{\text{def}} \forall \mathtt{X}\, \forall \mathtt{Y}\, (\mathtt{p}(\mathtt{X},\mathtt{Y}) \rightarrow \mathtt{p}(\mathtt{Y},\mathtt{X}))$$

$$\mathbf{C} =_{\text{def}} \forall \mathtt{X}\, \forall \mathtt{Y}\, \forall \mathtt{Z}\, (\mathtt{p}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{p}(\mathtt{Y},\mathtt{Z}) \rightarrow \mathtt{p}(\mathtt{X},\mathtt{Z}))$$

Dann gilt für $\mathfrak{I} = [\textbf{PS}, \textbf{FS}, \mathbb{N} \times \mathbb{N}, \boldsymbol{\Phi}, \boldsymbol{\Pi}]$ mit

$\boxed{\boldsymbol{\Pi}_{\mathtt{p}} = \{ (\mathbf{n}, \mathbf{m}) \mid \mathbf{n}, \mathbf{m} \text{ haben die gleichen Teiler} \}}$,

daß $\mathfrak{I} \models \mathbf{A}$ und $\mathfrak{I} \models \mathbf{B}$ und $\mathfrak{I} \models \mathbf{C}$ (Äquivalenzrelation).

Wählen wir

$\boldsymbol{\Pi}_{\mathtt{p}} = \{ (\mathbf{n}, \mathbf{m}) \mid \mathbf{n}, \mathbf{m} \in \mathbb{N} \text{ und } \mathbf{n} \leq \mathbf{m} \}$, so gilt $\mathfrak{I} \models \mathbf{A}$ und $\mathfrak{I} \models \mathbf{C}$, aber nicht $\mathfrak{I} \models \mathbf{B}$.

Beispiel 2.3.2 Für jeden Ausdruck $\boldsymbol{A}$ gilt

$$\models \boldsymbol{A} \rightarrow \boldsymbol{A}.$$

Definition 2.3.5 (Semantische Äquivalenz)

$\mathbf{A}_1$ *ist mit* $\mathbf{A}_2$ *semantisch äquivalent* ($\mathbf{A}_1 \equiv \mathbf{A}_2$) $=_{\text{def}}$ $(\mathbf{A}_1 \leftrightarrow \mathbf{A}_2)$ *ist allgemeingültig, d. h.* $\models (\mathbf{A}_1 \leftrightarrow \mathbf{A}_2)$.

Theorem 2.3.1 (Semantische Ersetzbarkeit)

Sind $\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3$ *beliebige Ausdrücke und geht* $\mathbf{A}_4$ *aus* $\mathbf{A}_1$ *durch Ersetzung von* $\mathbf{A}_2$ *durch* $\mathbf{A}_3$ *hervor, gilt schließlich*

$$\mathbf{A}_2 \equiv \mathbf{A}_3,$$

so gilt auch $\mathbf{A}_1 \equiv \mathbf{A}_4$.

Beispiel 2.3.3 Es gilt für Ausdrücke A, B, C

$A \vee B \equiv B \vee A$	Kommutativität
$(A \vee B) \vee C \equiv A \vee (B \vee C)$	Assoziativität
$A \vee (A \wedge B) \equiv A$	Absorption
$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$	Distributivit.
$\neg (A \vee B) \equiv \neg A \wedge \neg B$	De Morgansche Regel
$\neg\neg A \equiv A$	Doppelte Negation
$\neg A \rightarrow \neg B \equiv B \rightarrow A$	Kontraposition
$\neg \forall x A \equiv \exists x \neg A$	

Kommt x nicht frei in B vor, so

$\forall x A \vee B \equiv \forall x (A \vee B)$

Definition 2.3.6 (Normalformen)

$\mathbf{A}$ *heiße pränexe Normalform*

$=_{\text{def}}$ $\mathbf{A}$ *ist quantorenfrei oder es gibt*
$\mathbf{V}_1, \ldots, \mathbf{V}_n \in \mathbf{VAR}$,
Quantoren $\mathbf{Q}_1, \ldots, \mathbf{Q}_n \in \{\forall, \exists\}$ *und ein*
quantorenfreies $\mathbf{B}$, *so daß* $\boxed{\mathbf{A} = \mathbf{Q}_1\mathbf{V}_1 \ldots \mathbf{Q}_n\mathbf{V}_n\mathbf{B}}$.
$\mathbf{Q}_1\mathbf{V}_1 \ldots \mathbf{Q}_n\mathbf{V}_n$ *heißt Präfix von* $\mathbf{A}$.

$\mathbf{A}$ *heiße universale Normalform*

$=_{\text{def}}$ $\mathbf{A}$ *ist pränexe Normalform; ist das Präfix von* $\mathbf{A}$ *nicht leer, so kommen darin nur Allquantoren vor.*

Theorem 2.3.2

Zu jedem Ausdruck $\mathbf{A} \in \mathbf{AUSD}$ *kann eine pränexe Normalform* $\mathbf{N}$ *konstruiert werden, so daß*

$$\mathbf{A} \equiv \mathbf{N}.$$

Beweis

Anwenden des semantischen Ersetzbarkeitstheorems. $\square$

Sei $\mathbf{X}$ eine Menge von Ausdrücken.

Definition 2.4.1 (Modell)

$\mathfrak{J}$ heiße Modell von $\mathbf{X}$ ($\mathfrak{J} \models \mathbf{X}$)

$=_{\text{def}}$ Für jedes $\mathbf{A} \in \mathbf{X}$ ist $\mathbf{A}$ in $\mathfrak{J}$ allgemeingültig,

d. h. es gilt $\boxed{\mathfrak{J} \models \mathbf{A}}$.

Menge aller Modelle von $\mathbf{X}$: $\boxed{\mathbf{MOD}(\mathbf{X})}$.

Theorem 2.4.1 (Endlichkeitssatz für Modelle)
Gibt es zu jeder endlichen Teilmenge $\mathbf{X}_{\text{fin}} \subseteq \mathbf{Y}$ *ein Modell* $\mathfrak{I}$, *d. h. mit* $\mathfrak{I} \models \mathbf{X}_{\text{fin}}$, *dann gibt es auch für die gesamte Menge* $\mathbf{Y}$ *ein Modell, etwa* $\mathfrak{I}'$, *d. h. mit* $\mathfrak{I}' \models \mathbf{Y}$.

Definition 2.4.2 (Semantische Folgerung)

Aus $\mathbf{X}$ *folgt (semantisch)* $\mathbf{A}$ ($\mathbf{X} \Vdash \mathbf{A}$)

$=_{\text{def}}$ *Für jede Interpretation* $\mathfrak{J}$ *gilt:*

Wenn $\mathfrak{J}$ *Modell von* $\mathbf{X}$, *so* $\mathfrak{J}$ *Modell von* $\mathbf{A}$.

$$\mathbf{CONS}(\mathbf{X}) =_{\text{def}} \{\mathbf{A} \mid \mathbf{A} \in \mathbf{AUSD} \text{ und } \mathbf{X} \Vdash \mathbf{A}\}.$$

Beispiel 2.4.1

$$\mathbf{PS} = \{=/2\}, \qquad \mathbf{FS} = \{+/2, 0/0\}$$

$$\mathbf{A} =_{\text{def}} \forall X \forall Y \forall Z \ \ (X + Y) + Z = X + (Y + Z)$$

(+ ist assoziativ)

$$\mathbf{B} =_{\text{def}} \forall X \ \ X + 0 = X$$

(Neutrales Element)

$$\mathbf{C} =_{\text{def}} \forall X \exists Y \ \ X + Y = 0$$

(Inverses)

$$\mathbf{D} =_{\text{def}} \forall X \forall Y \ \ X + Y = Y + X$$

(+ ist kommutativ)

$$\mathbf{G} =_{\text{def}} \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$$

$\mathbf{T} \in \mathbf{Cons}(\mathbf{G}) \Leftrightarrow \mathbf{T}$ ist ein Satz der Gruppentheorie.

Definition 2.4.3 (Generalisierte)
Seien $X_1, \ldots, X_n$ *die Variablen, die in* A *frei vorkommen.*

$$\mathbf{Gen}(A) =_{\text{def}} \forall X_1 \ldots \forall X_n A.$$

Lemma 2.4.2
Sei $A \in \mathbf{AUSD}$.

$$\mathbf{Cons}(A) = \mathbf{Cons}(\mathbf{Gen}(A)).$$

2.5.1 Idee

Semantische Folgerung (zu aufwändig) soll durch Syntaktisches Ableiten (Berechnungen über der Sprache **AUSD**) ersetzt werden.

2.5.1 Idee

Semantische Folgerung (zu aufwändig) soll durch Syntaktisches Ableiten (Berechnungen über der Sprache **AUSD**) ersetzt werden.

Logik: Eine Sprache **AUSD** zusammen mit einem (semantischen) Folgerungsoperator $\Vdash$.

Kalkül: Eine Sprache **AUSD** zusammen mit einem (syntaktischen) Beweisbarkeitsoperator $\vdash$.

Rechtfertigung eines Kalküls:

Korrektheit: $\mathbf{X} \vdash \boldsymbol{A} \quad \Longrightarrow \quad \mathbf{X} \Vdash \boldsymbol{A}$

„Alles, was beweisbar ist, folgt auch semantisch“.

Vollständigkeit: $\mathbf{X} \Vdash \boldsymbol{A} \quad \Longrightarrow \quad \mathbf{X} \vdash \boldsymbol{A}$

„Alles, was folgt, ist auch beweisbar“.

Rechtfertigung eines Kalküls:

Korrektheit: $\mathbf{X} \vdash \boldsymbol{A} \quad \Longrightarrow \quad \mathbf{X} \Vdash \boldsymbol{A}$

„Alles, was beweisbar ist, folgt auch semantisch“.

Vollständigkeit: $\mathbf{X} \Vdash \boldsymbol{A} \quad \Longrightarrow \quad \mathbf{X} \vdash \boldsymbol{A}$

„Alles, was folgt, ist auch beweisbar“.

Um den Beweisoperator automatisieren zu können, wählen wir den Resolutionskalkül.

Rechtfertigung eines Kalküls:

Korrektheit: $\mathbf{X} \vdash \boldsymbol{A} \quad \Longrightarrow \quad \mathbf{X} \Vdash \boldsymbol{A}$

„Alles, was beweisbar ist, folgt auch semantisch“.

Vollständigkeit: $\mathbf{X} \Vdash \boldsymbol{A} \quad \Longrightarrow \quad \mathbf{X} \vdash \boldsymbol{A}$

„Alles, was folgt, ist auch beweisbar“.

Um den Beweisoperator automatisieren zu können, wählen wir den Resolutionskalkül.

Da dieser auf einer sehr einfachen Schlußregel beruht, müssen wir die Ausdrücke stark vereinfachen.
$\rightsquigarrow$ Klauselform.

2.5.2 Vorbereitung

Definition 2.5.1 (Termeinsetzung)

$$\mathbf{A}\left[{}^{\mathbf{V}}/_{\mathbf{T}}\right]$$

$=_{\text{def}}$ *Derjenige Ausdruck, der aus* $\mathbf{A}$ *dadurch entsteht, dass die Individuenvariable* $\mathbf{V}$ *in* $\mathbf{A}$ *überall, wo sie frei vorkommt, simultan durch* $\mathbf{T}$ *ersetzt wird.*

2.5.2 Vorbereitung

Definition 2.5.1 (Termeinsetzung)

$$\mathbf{A}\left[{}^{\mathbf{V}}\!/_{\mathbf{T}}\right]$$

$=_{\text{def}}$ *Derjenige Ausdruck, der aus* $\mathbf{A}$ *dadurch entsteht, dass die Individuenvariable* $\mathbf{V}$ *in* $\mathbf{A}$ *überall, wo sie frei vorkommt, simultan durch* $\mathbf{T}$ *ersetzt wird.*

Gebundene Umbenennung: Eine Variable wird überall im Wirkungsbereich eines Quantors, durch den sie gebunden wird, durch eine andere ersetzt.

Lemma 2.5.1 (Widerlegungssystem)

Für jedes $\mathbf{X} \subseteq \mathbf{AUSD}$ *und jedes* $\mathbf{A} \in \mathbf{AUSD}$ *gilt:*

$\mathbf{X} \Vdash\!\!- \mathbf{A}$ *genau dann,*

wenn $\boxed{\mathbf{X} \cup \{\neg\, \mathbf{Gen}(\mathbf{A})\}}$ *kein Modell hat.*

Definition 2.5.2

1. $\mathbf{X}$ *und* $\mathbf{Y}$ *heißen semantisch äquivalent* ($\mathbf{X} \equiv \mathbf{Y}$)

   $=_{\text{def}}$ *Für jede Interpretation* $\mathfrak{J}$ *und jeden* $\mathfrak{J}$*-Zustand* $\boldsymbol{\sigma}$ *gilt:*

   $$\boxed{\boldsymbol{\sigma}, \mathfrak{J} \models \mathbf{X} \text{ g.d.w. } \boldsymbol{\sigma}, \mathfrak{J} \models \mathbf{Y}}.$$

2. $\mathbf{X}$ *und* $\mathbf{Y}$ *heißen Modell-äquivalent* ($\mathbf{X} \cong \mathbf{Y}$)

   $=_{\text{def}}$ $\boxed{\mathbf{MOD}(\mathbf{X}) = \mathbf{MOD}(\mathbf{Y})}$.

3. $\mathbf{X}$ *und* $\mathbf{Y}$ *heißen schwach Modell-äquivalent* ($\mathbf{X} \approxeq \mathbf{Y}$)

   $=_{\text{def}}$ $\boxed{\mathbf{MOD}(\mathbf{X}) \neq \emptyset \text{ gdw. } \mathbf{MOD}(\mathbf{Y}) \neq \emptyset}$.

Theorem 2.5.2

Zu jedem $\mathbf{X} \subseteq \mathbf{AUSD}$ *kann ein* $\mathbf{Y} \subseteq \mathbf{AUSD}$ *konstruiert werden, so dass*

1. $\mathbf{Y}$ *allein aus pränexen Normalformen besteht und*
2. $\mathbf{X}$ *und* $\mathbf{Y}$ *semantisch äquivalent sind.*

2.5.3 Skolemisierung

A sei eine Aussage (Ausdruck ohne freie Variablen) der Form

$$A = \forall X_1 \ldots \forall X_n \exists Y \; B.$$

Wir wählen ein neues Funktionssymbol f/n.
Dann ist A schwach modell-äquivalent zu

$$\forall X_1 \ldots \forall X_n B \left[{}^{Y}/_{f(X_1, \ldots, X_n)} \right].$$

Theorem 2.5.3 (Skolemisierungstheorem)
Zu jedem $\mathbf{X} \subseteq$ **AUSD** *kann ein* $\mathbf{X}' \subseteq$ **AUSD** *konstruiert werden, so dass*

1. $\mathbf{X}'$ *allein aus universalen Normalformen besteht und*
2. $\mathbf{X}'$ *schwach Modell-äquivalent mit* $\mathbf{X}$ *ist.*

Folgerung 2.5.4

Zu jedem $\mathbf{X} \subseteq$ **AUSD** *kann ein* $\mathbf{X}' \subseteq$ **AUSD** *konstruiert werden, so dass*

1. $\mathbf{X}'$ *allein aus quantorenfreien Ausdrücken besteht und*
2. $\mathbf{X}'$ *schwach Modell-äquivalent mit* $\mathbf{X}$ *ist.*

2.5.4 Konjunktive Normalform

Definition 2.5.3

1. $\mathbf{L}$ *heiße Literal*
   $=_{\text{def}}$ $\mathbf{L} \in$ **AUSD** *und* $\mathbf{L}$ *ist ein atomarer oder ein negierter atomarer Ausdruck.*
2. *Eine konjunktive Normalform ist eine Konjunktion aus Alternativen, die ihrerseits aus Literalen bestehen.*

Lemma 2.5.5

Zu jeder Menge $\mathbf{X} \subseteq$ **AUSD** *von quantorenfreien Ausdrücken kann* $\mathbf{X}' \subseteq$ **AUSD** *konstruiert werden, so dass*

1. *alle Ausdrücke* $\mathbf{A}' \in \mathbf{X}'$ *konjunktive Normalformen sind*
2. *und* $\mathbf{X}'$ *semantisch äquivalent mit* $\mathbf{X}$ *ist.*

Theorem 2.5.6

Zu jedem $\mathbf{X} \subseteq$ **AUSD** *kann ein* $\mathbf{X}' \subseteq$ **AUSD** *konstruiert werden, so dass*

1. *alle Ausdrücke aus* $\mathbf{X}'$ *Alternativen von Literalen sind und*
2. $\mathbf{X}'$ *schwach Modell-äquivalent mit* $\mathbf{X}$ *ist.*

2.5.5 Klausellogik

Menge von Alternativen von Literalen

$\Longrightarrow$ Menge von Mengen von Literalen

$$\{\neg \mathtt{p(X)} \vee \mathtt{p(f(X,\ Y))}, \neg \mathtt{q(Y)} \vee \mathtt{p(f(X,\ Y))}\}$$

$$\Longrightarrow \{\{\neg \mathtt{p(X)}, \mathtt{p(f(X,\ Y))}\}, \{\neg \mathtt{q(Y)}, \mathtt{p(f(X,\ Y))}\}\}$$

- Klausel: Menge von Literalen
  (Semantik wie bei Alternative).

- Sei $\mathbf{X}$ eine Menge von Alternativen von Literalen.
  $\mathbf{KLM}(\mathbf{X})$: Die $\mathbf{X}$ zugeordnete Klauselmenge.

- $\Box =_{\text{def}} \emptyset$: Die leere Klausel (ist niemals erfüllt).

2.5.6 Grundresolution

Definition 2.5.4

T *ist ein Grundterm*

$=_{\text{def}}$ T *enthält keine Individuenvariablen.*

Menge aller Grundterme: **GTERM**.

Grundklausel: Klausel, deren Terme alle Grundterme sind.

Definition 2.5.5 (Herbrand-Menge)
Sei A ein quantorenfreier Ausdruck, der genau die Variablen $V_1, \ldots, V_n$ enthält.

$$H(A)$$

$$=_{\text{def}} \left\{ A \left[{}^{V_1, \ldots, V_n}/_{T_1, \ldots, T_n} \right] \middle| \; T_i \in \text{GTERM} \right\}$$

$$H(X) =_{\text{def}} \bigcup_{A \in X} H(A)$$

Grundresolution: Anwenden der Schlußregel

Aus	$\mathbf{K}' \cup \{\boldsymbol{A}\}$
und	$\mathbf{K}'' \cup \{\neg \boldsymbol{A}\}$
wird abgeleitet	$\mathbf{K}' \cup \mathbf{K}''$

Grundresolution: Anwenden der Schlußregel

$$\begin{array}{ll} \text{Aus} & \mathbf{K}' \cup \{A\} \\ \text{und} & \mathbf{K}'' \cup \{\neg A\} \\ \hline \text{wird abgeleitet} & \mathbf{K}' \cup \mathbf{K}'' \end{array}$$

Grundresolutionsbeweis aus Grundklauselmenge

KLM: Sammeln aller Klauseln, die sich aus **KLM** und daraus ableitbaren Klauseln (induktive Definition) ableiten lassen.

Definition 2.5.6

Sei **KLM** *eine Menge von Grundklauseln.*

$\mathbf{KLM} \vdash_{\mathbf{GR}} \mathbf{K}$

$=_{\text{def}}$ $\mathbf{K} \in \mathbf{KLM}$ *oder*

es gibt einen Grundresolutionsbeweis

für **K** *aus* **KLM**

Definition 2.5.6

Sei **KLM** *eine Menge von Grundklauseln.*

$\mathbf{KLM} \vdash_{\mathbf{GR}} \mathbf{K}$

$=_{\text{def}}$ $\mathbf{K} \in \mathbf{KLM}$ *oder*

es gibt einen Grundresolutionsbeweis

für **K** *aus* **KLM**

Sei $\mathbf{X}$ eine Menge von Alternativen von Literalen.

Theorem 2.5.7

$\mathbf{X}$ *hat kein Modell genau dann, wenn*

$\mathbf{KLM}(\mathbf{H}(\mathbf{X})) \vdash_{\mathbf{GR}} \square$.

Definition 2.5.7 (Grundresolutionskalkül)
Sei $\mathbf{X} \subseteqq \mathbf{AUSD}$, $A \in \mathbf{AUSD}$ *und* $\mathbf{X}'$ *eine Menge von Alternativen von Literalen, so dass*

$$\mathbf{X}' \approxeq \mathbf{X} \cup \{\neg A\}.$$

Dann sei

$$\mathbf{X} \mathrel{\underline{\vdash}_{\mathbf{G}}} A =_{\mathrm{def}} \mathbf{KLM}\left(\mathbf{H}\left(\mathbf{X}'\right)\right) \mathrel{\underline{\vdash}_{\mathbf{GR}}} \square.$$

Definition 2.5.7 (Grundresolutionskalkül)
Sei $\mathbf{X} \subseteq \mathbf{AUSD}$, $\mathbf{A} \in \mathbf{AUSD}$ *und* $\mathbf{X}'$ *eine Menge von Alternativen von Literalen, so dass*

$$\mathbf{X}' \approxeq \mathbf{X} \cup \{\neg \mathbf{A}\}.$$

Dann sei

$$\mathbf{X} \mathrel{\vdash_{\!\!\underline{\ \ \mathbf{G}\ \ }}} \mathbf{A} =_{\text{def}} \mathbf{KLM}\left(\mathbf{H}\left(\mathbf{X}'\right)\right) \mathrel{\vdash_{\!\!\underline{\ \ \mathbf{GR}\ \ }}} \square.$$

Theorem 2.5.8 (Korrektheit und Vollständigkeit)
Sei $\mathbf{X} \subseteq \mathbf{AUSD}$ *und* $\mathbf{A} \in \mathbf{AUSD}$. *Dann gilt*

$$\mathbf{X} \Vdash \mathbf{A} \text{ g.d.w. } \mathbf{X} \mathrel{\vdash_{\!\!\underline{\ \ \mathbf{G}\ \ }}} \mathbf{A}.$$

2.5.7 Prädikatenlogische Resolution

Idee (Robinson, 1965):

Substitutionen nach Bedarf ausführen.

Definition 2.5.8 (Substitution)

Eine Abbildung

$$\mathbf{sub} : \mathbf{VAR} \to \mathbf{TERM}$$

nennen wir Substitution.

Sei Z ein Term oder Ausdruck.

$Z\,$**sub**:

ersetze in Z alle Variablen $\mathbf{V}$ simultan durch **sub**$(\,\mathbf{V}\,)$.

Sei Z ein Term oder Ausdruck.

Z **sub**:
ersetze in Z alle Variablen V simultan durch **sub**(V).

Notation **sub** $= \left[{}^{V}/_{T}\right]$:

$$\mathbf{sub}(W) = \begin{cases} T, & \text{falls } W = V \\ W, & \text{falls } W \neq V \end{cases}$$

Sei Z ein Term oder Ausdruck.

Z **sub**:
ersetze in Z alle Variablen $\mathbf{V}$ simultan durch $\mathbf{sub}(\mathbf{V})$.

Notation $\mathbf{sub} = \left[{}^{\mathbf{V}}/_{\mathbf{T}}\right]$:

$$\mathbf{sub}(\boldsymbol{W}) = \begin{cases} \mathbf{T}, & \text{falls } \boldsymbol{W} = \mathbf{V} \\ \boldsymbol{W}, & \text{falls } \boldsymbol{W} \neq \mathbf{V} \end{cases}$$

$\mathbf{sub}_1 \circ \mathbf{sub}_2$: Hintereinanderausführung.

Sei $\mathbf{M}$ eine Menge von Termen oder Ausdrücken.
(Anwendung $\mathbf{M}\,\mathbf{sub}$ elementweise definiert)

Definition 2.5.9 (Unifikator)

1. **sub** *heiße Unifikator für* $\mathbf{M}$
   $=_{\text{def}}$ $\mathbf{M}\,\mathbf{sub}$ *ist einelementig.*
2. **sub** *heiße allgemeinster Unifikator für* $\mathbf{M}$
   $=_{\text{def}}$ **sub** *ist Unifikator für* $\mathbf{M}$ *und für jeden Unifikator* $\mathbf{sub}'$ *von* $\mathbf{M}$ *existiert* $\mathbf{sub}''$*, so dass*

$$\mathbf{sub}' = \mathbf{sub} \circ \mathbf{sub}''\,.$$

Theorem 2.5.9 (Unifikationstheorem)

Hat die Menge $\mathbf{M}$ *von Literalen einen Unifikator, so hat* $\mathbf{M}$ *auch einen allgemeinsten Unifikator.*

Unifikationsalgorithmus

EINGABE:

Eine nicht-leere endliche Menge $\mathbf{M}$ von Literalen.

sub $:=$ die identische Substitution.

WHILE **card** $(\mathbf{M}\,\mathbf{sub}) > 1$ DO

BEGIN Wähle $\mathbf{L}_1, \mathbf{L}_2 \in \mathbf{M}\,\mathbf{sub}$ sowie die erste Stelle, wo sich $\mathbf{L}_1$ und $\mathbf{L}_2$ unterscheiden.

IF keines der Symbole an dieser Stelle ist eine Variable

THEN stop „nicht unifizierbar“

ELSE

BEGIN

Sei **V** die Variable und **T** der Term an dieser Stelle.

IF **V** kommt in **T** vor

THEN stop „nicht unifizierbar“

ELSE **sub** $:=$ **sub** $\circ \left[{}^{\mathbf{V}}/_{\mathbf{T}}\right]$

END;

END;

Gib **sub** aus.

Theorem 2.5.10

Der Unifikationsalgorithmus terminiert für jedes $\mathbf{M}$ *und*

1. *gibt „nicht unifizierbar“ aus*
   g.d.w. $\mathbf{M}$ *nicht unifizierbar ist*
2. *gibt einen allgemeinsten Unifikator* **sub** *für* $\mathbf{M}$ *aus*
   g.d.w. $\mathbf{M}$ *unifizierbar ist*

Prädikatenlogische Resolutionsregel

Sei A ein atomarer Ausdruck.

$\overline{A} =_{\text{def}} \neg A$ und $\overline{\neg A} =_{\text{def}} A$.

$$\begin{array}{ll} \text{Aus} & K' \cup \{L_1, \ldots, L_n\} \\ \text{und} & K'' \cup \{L'_1, \ldots, L'_m\} \\ \hline \text{wird abgeleitet} & \left(K' \cup K''\right) \textbf{sub} \end{array}$$

wobei **sub** allg. Unifikator von

$\left\{L_1, \ldots, L_n, \overline{L'_1}, \ldots, \overline{L'_m}\right\}$ ist.

Prädikatenlogische Resolutionsregel

Sei $\mathbf{A}$ ein atomarer Ausdruck.

$\overline{\mathbf{A}} =_{\text{def}} \neg\mathbf{A}$ und $\overline{\neg\mathbf{A}} =_{\text{def}} \mathbf{A}$.

$$\begin{array}{ll} \text{Aus} & \mathbf{K}' \cup \{\mathbf{L}_1, \ldots, \mathbf{L}_n\} \\ \text{und} & \mathbf{K}'' \cup \{\mathbf{L}_1', \ldots, \mathbf{L}_m'\} \\ \hline \text{wird abgeleitet} & \left(\mathbf{K}' \cup \mathbf{K}''\right) \mathbf{sub} \end{array}$$

wobei **sub** allg. Unifikator von

$\left\{\mathbf{L}_1, \ldots, \mathbf{L}_n, \overline{\mathbf{L}_1'}, \ldots, \overline{\mathbf{L}_m'}\right\}$ ist.

Prädikatenlogischer Resolutionsbeweis aus Klauselmenge KLM: Wie bei Grundresolution.

Definition 2.5.10

Sei **KLM** *eine Menge von Klauseln.*

$\mathbf{KLM} \vdash_{\mathbf{PR}} \mathbf{K}$

$=_{\text{def}}$ $\mathbf{K} \in \mathbf{KLM}$ *oder*

ex. ein prädikatenlog. Resolutionsbeweis

für **K** *aus* **KLM**

Definition 2.5.10

Sei **KLM** *eine Menge von Klauseln.*

$\mathbf{KLM} \vdash_{\mathbf{PR}} \mathbf{K}$

$=_{\text{def}}$ $\mathbf{K} \in \mathbf{KLM}$ *oder*

ex. ein prädikatenlog. Resolutionsbeweis

für **K** *aus* **KLM**

Sei **X** eine Menge von Alternativen von Literalen.

Theorem 2.5.11

X *hat kein Modell genau dann, wenn*

$\mathbf{KLM}(\mathbf{X}) \vdash_{\mathbf{PR}} \square$.

Definition 2.5.11 (Prädikatenlog. Resolutionskalkül)
Sei $\mathbf{X} \subseteqq \mathbf{AUSD}$, $\boldsymbol{A} \in \mathbf{AUSD}$ *und* $\mathbf{X}'$ *eine Menge von Alternativen von Literalen, so dass*

$$\mathbf{X}' \approxeq \mathbf{X} \cup \{\neg \boldsymbol{A}\}.$$

Dann sei

$$\mathbf{X} \vdash \boldsymbol{A} =_{\mathrm{def}} \mathbf{KLM}\left(\mathbf{X}'\right) \vdash_{\overline{\mathbf{PR}}} \square.$$

Definition 2.5.11 (Prädikatenlog. Resolutionskalkül)
Sei $\mathbf{X} \subseteq \mathbf{AUSD}$, $\mathbf{A} \in \mathbf{AUSD}$ *und* $\mathbf{X}'$ *eine Menge von Alternativen von Literalen, so dass*

$$\mathbf{X}' \approxeq \mathbf{X} \cup \{\neg \mathbf{A}\}.$$

Dann sei

$$\mathbf{X} \vdash \mathbf{A} =_{\text{def}} \mathbf{KLM}\left(\mathbf{X}'\right) \mathrel{\vdash_{\overline{\mathbf{PR}}}} \Box.$$

Theorem 2.5.12 (Korrektheit und Vollständigkeit)
Sei $\mathbf{X} \subseteq \mathbf{AUSD}$ *und* $\mathbf{A} \in \mathbf{AUSD}$. *Dann gilt*

$$\mathbf{X} \Vdash \mathbf{A} \text{ g.d.w. } \mathbf{X} \vdash \mathbf{A}.$$

Theorem 2.6.1

Ist X *rekursiv aufzählbar, so ist auch* $\mathbf{Cons}(X)$ *rekursiv aufzählbar.*

Theorem 2.6.1
Ist **X** *rekursiv aufzählbar, so ist auch* **Cons**(**X**) *rekursiv aufzählbar.*

Theorem 2.6.2 (Satz von Church)
Gibt es mindestens ein nullstelliges Funktionssymbol, zwei einstellige Funktionssymbole und ein zweistelliges Prädikatensymbol, so ist für die Sprache **AUSD** *die Allgemeingültigkeit unentscheidbar.*

Idee (Kowalski, 1970):

Durch spezielle Klauselnotation und festgelegte Resolutionsstrategie Logik zum Programmieren nutzen.

2.7.1 Horn-Klauseln

Definition 2.7.1

Eine Klausel $\mathbf{K}$ *heiße* Horn-*Klausel*
$=_{\text{def}}$ $\mathbf{K}$ *enthält höchstens ein positives Literal.*

Was können Horn-Klauseln?

Wir unterscheiden drei Fälle:

Fakten (oder Tatsachenklauseln):

$\mathbf{K}$ besteht aus genau einem positiven Literal, z. B.

$$\mathbf{K} = \{\mathtt{gruen(gras)}\}$$

Regeln (oder Prozedurklauseln):

$\mathbf{K}$ enthält ein positives und mindestens ein negatives Literal, z. B.

$$\mathbf{K} = \{\neg\mathtt{mensch(X)}, \mathtt{sterblich(X)}\}$$
$$\equiv \mathtt{mensch(X)} \rightarrow \mathtt{sterblich(X)}$$

oder

$$\mathbf{K} = \{\neg\mathtt{teilt(2,X)}, \neg\mathtt{teilt(3,X)}, \mathtt{teilt(6,X)}\}$$
$$\equiv \mathtt{teilt(2,X)} \wedge \mathtt{teilt(3,X)} \rightarrow \mathtt{teilt(6,X)}$$

Anfragen (oder Zielklauseln):

$\mathbf{K}$ enthält nur negative Literale, z. B.

$$\mathbf{K} = \{\neg\mathtt{prim(X)}, \neg\mathtt{gerade(X)}\}$$
$$\equiv \neg\left(\mathtt{prim(X)} \wedge \mathtt{gerade(X)}\right)$$

Anfragen (oder Zielklauseln):

$\mathbf{K}$ enthält nur negative Literale, z. B.

$$\mathbf{K} = \{\neg\mathtt{prim(X)}, \neg\mathtt{gerade(X)}\}$$
$$\equiv \neg\,(\mathtt{prim(X)} \wedge \mathtt{gerade(X)})$$

Achtung:

$$\forall\mathtt{X}\,(\neg\mathtt{prim(X)} \vee \neg\mathtt{gerade(X)})$$
$$\equiv \neg\exists\mathtt{X}\,(\mathtt{prim(X)} \wedge \mathtt{gerade(X)})$$

Beispiel 2.7.1 Sei

$$\mathbf{X} = \left\{ \begin{array}{l} \forall \mathtt{V}\ \big(\mathtt{mensch(V)} \rightarrow \mathtt{sterblich(V)}\big)\ , \\ \mathtt{mensch(sokrates)} \end{array} \right\} .$$

Beispiel 2.7.1 Sei

$$\mathbf{X} = \left\{ \begin{array}{l} \forall \mathtt{V}\ (\mathtt{mensch(V)} \rightarrow \mathtt{sterblich(V)})\ , \\ \mathtt{mensch(sokrates)} \end{array} \right\} .$$

Dann gilt

$$\mathbf{X} \Vdash \exists \mathtt{V}\, \mathtt{sterblich(V)}$$

g.d.w.

$$\left\{ \begin{array}{l} \{\neg\mathtt{mensch(V)}, \mathtt{sterblich(V)}\}, \\ \{\mathtt{mensch(sokrates)}\}, \\ \{\neg\mathtt{sterblich(V)}\} \end{array} \right\} \vdash_{\mathbf{PR}} \square$$

Was können Horn-Klauseln nicht?

Alternativen als Konklusion

$$\texttt{elternteil(X,Y)} \rightarrow \texttt{vater(X,Y)} \vee \texttt{mutter(X,Y)}$$

$$\equiv \{\neg\texttt{elternteil(X,Y)}, \texttt{vater(X,Y)}, \texttt{mutter(X,Y)}\}$$

Was können Horn-Klauseln nicht?

Alternativen als Konklusion

$$\texttt{elternteil(X,Y)} \rightarrow \texttt{vater(X,Y)} \vee \texttt{mutter(X,Y)}$$

$$\equiv \{\neg\texttt{elternteil(X,Y)}, \texttt{vater(X,Y)}, \texttt{mutter(X,Y)}\}$$

Negationen

$$\texttt{teil(X,Y)} \wedge \neg\texttt{teil(Y,X)} \rightarrow \texttt{echter\_teil(X,Y)}$$

$$\equiv \{\neg\texttt{teil(X,Y)}, \texttt{teil(Y,X)}, \texttt{echter\_teil(X,Y)}\}$$

2.7.2 Logik-Programme

Notation:

Fakten

Beispiel	dargestellt als
{`mensch(sokrates)`}	`mensch(sokrates).`

Regeln

Beispiel

$\{\neg$`teilt(2,X)`, $\neg$`teilt(3,X)`, `teilt(6,X)`$\}$

dargestellt als

`teilt(6,X)` $\leftarrow$ `teilt(2,X)`, `teilt(3,X)`.

Regeln

Beispiel

$\{\neg$`teilt(2,X)`$, \neg$`teilt(3,X)`$,$ `teilt(6,X)`$\}$

dargestellt als

`teilt(6,X)` $\leftarrow$ `teilt(2,X), teilt(3,X).`

Ziele

Beispiel

$\{\neg$`prim(X)`$, \neg$`gerade(X)`$\}$

dargestellt als

`?-  prim(X), gerade(X).`

Definition 2.7.2 (Logik-Programm)

1. *Eine Klausel* $\mathbf{K}$ *heiße definit (oder Programm-Klausel)*
   $=_{\text{def}}$ $\mathbf{K}$ *enthält genau ein positives Literal.*
   Das positive Literal heißt Kopf von $\mathbf{K}$, *die Menge der negativen heißt Rumpf.*
2. $\mathbf{KLM}$ *heiße Logik-Programm*
   $=_{\text{def}}$ $\mathbf{KLM}$ *ist eine Menge von definiten Klauseln.*

Definition 2.7.2 (Logik-Programm)

1. *Eine Klausel* **K** *heiße definit (oder Programm-Klausel)*
   $=_{\text{def}}$ **K** *enthält genau ein positives Literal.*
   Das positive Literal heißt Kopf von **K**, *die Menge der negativen heißt Rumpf.*
2. **KLM** *heiße Logik-Programm*
   $=_{\text{def}}$ **KLM** *ist eine Menge von definiten Klauseln.*

Definition 2.7.2 (Logik-Programm)

1. *Eine Klausel* **K** *heiße definit (oder Programm-Klausel)*

   $=_{\text{def}}$ **K** *enthält genau ein positives Literal.*

   Das positive Literal heißt Kopf von **K**, *die Menge der negativen heißt Rumpf.*

2. **KLM** *heiße Logik-Programm*

   $=_{\text{def}}$ **KLM** *ist eine Menge von definiten Klauseln.*

2.7.3 SLD-Resolution

„linear resolution with selection function for definite clauses“

Wir definieren eine Resolutionsstrategie $\mathtt{SLD}(\mathbf{P}, \mathbf{Z})$, die für ein Logik-Programm $\mathbf{P}$ und eine Zielklausel $\mathbf{Z}$ bestimmt, ob sich aus $\mathbf{P} \cup \{\mathbf{Z}\}$ die leere Klausel ableiten läßt.

Wir nehmen an, dass sowohl $\mathbf{P}$ als auch $\mathbf{Z}$ geordnet sind.

Auf diese Weise wird das Ausführungsmodell eines Logikprogramms eindeutig festgelegt.

2.7.3 SLD-Resolution

„linear resolution with selection function for definite clauses“

Wir definieren eine Resolutionsstrategie $\mathtt{SLD}(\mathbf{P}, \mathbf{Z})$, die für ein Logik-Programm $\mathbf{P}$ und eine Zielklausel $\mathbf{Z}$ bestimmt, ob sich aus $\mathbf{P} \cup \{\mathbf{Z}\}$ die leere Klausel ableiten läßt.

Wir nehmen an, dass sowohl $\mathbf{P}$ als auch $\mathbf{Z}$ geordnet sind.

Auf diese Weise wird das Ausführungsmodell eines Logikprogramms eindeutig festgelegt.

2.7.3 SLD-Resolution

„linear resolution with selection function for definite clauses“

Wir definieren eine Resolutionsstrategie $\mathtt{SLD}(\mathbf{P}, \mathbf{Z})$, die für ein Logik-Programm $\mathbf{P}$ und eine Zielklausel $\mathbf{Z}$ bestimmt, ob sich aus $\mathbf{P} \cup \{\mathbf{Z}\}$ die leere Klausel ableiten läßt.

Wir nehmen an, dass sowohl $\mathbf{P}$ als auch $\mathbf{Z}$ geordnet sind.

Auf diese Weise wird das Ausführungsmodell eines Logikprogramms eindeutig festgelegt.

2.7.3 SLD-Resolution

„linear resolution with selection function for definite clauses“

Wir definieren eine Resolutionsstrategie $\mathtt{SLD}(\mathbf{P}, Z)$, die für ein Logik-Programm $\mathbf{P}$ und eine Zielklausel Z bestimmt, ob sich aus $\mathbf{P} \cup \{Z\}$ die leere Klausel ableiten läßt.

Wir nehmen an, dass sowohl $\mathbf{P}$ als auch Z geordnet sind.

Auf diese Weise wird das Ausführungsmodell eines Logikprogramms eindeutig festgelegt.

Festlegung: Resolvent von Z und Programmklausel K wird immer bzgl. der beiden ersten Literale gebildet.

Die 'restlichen' Literale von K werden vorne an Z angehängt.

Festlegung: Resolvent von $\mathbf{Z}$ und Programmklausel $\mathbf{K}$ wird immer bzgl. der beiden ersten Literale gebildet.

Die 'restlichen' Literale von $\mathbf{K}$ werden vorne an $\mathbf{Z}$ angehängt.

`RECURSIVE FUNCTION SLD(` $\mathbf{P}, \mathbf{Z}$ `).`

`EINGABE:` Logikprogramm $\mathbf{P}$, Zielklausel $\mathbf{Z}$.

`IF` $\mathbf{Z}$ ist leer
`THEN RETURN true`
`ELSE`
  `BEGIN`
    $\mathbf{L} :=$ Erstes Literal in $\mathbf{Z}$.

```
LOOP
    K := nächste Klausel in P, deren Kopf mit L̄ unifizierb.

    IF ein K konnte gefunden werden
    THEN
        BEGIN
            Z' := Resolvent von K und L.
            IF SLD(P, Z') THEN RETURN true

        END;

    ELSE RETURN false

ENDLOOP

END
```

2.7.4 Antworterzeugung

Problem: $\mathtt{SLD}(\mathbf{P}, \mathbf{Z})$ gibt nur `true` oder `false` zurück.

Für Variablen in der Anfrage hätten wir aber gern eine erfüllende Belegung.

2.7.4 Antworterzeugung

Problem: SLD($\mathbf{P}$, $\mathbf{Z}$) gibt nur `true` oder `false` zurück.

Für Variablen in der Anfrage hätten wir aber gern eine erfüllende Belegung.

Lösung: Für jede in $\mathbf{Z}$ vorkommente Variable $\mathbf{V}$ fügen wir $\mathbf{Z}$ ein Literal $\neg$antw("$\mathbf{V}$",$\mathbf{V}$) hinzu, das bei der SLD-Resolution ignoriert wird. Die erfüllende Belegung wird automatisch während der Resolution für $\mathbf{V}$ substituiert. Am Ende werden die Antwortliterale aus $\mathbf{Z}$ geeignet ausgegeben.

2.7.5 Abwandlungen der Resolutionsstrategie

- Erzeugung aller Antworten
  $\leadsto$ PROLOG-Ausführungsmodell.

2.7.5 Abwandlungen der Resolutionsstrategie

- Erzeugung aller Antworten
  $\rightsquigarrow$ PROLOG-Ausführungsmodell.
- SLD ist nicht vollständig für die Klasse der Hornklauseln (Möglichkeit von Endlosschleifen).

2.7.5 Abwandlungen der Resolutionsstrategie

- Erzeugung aller Antworten
  $\leadsto$ PROLOG-Ausführungsmodell.

- SLD ist nicht vollständig für die Klasse der Hornklauseln (Möglichkeit von Endlosschleifen).

  Breitensuche ist vollständig für die Klasse der Hornklauseln, aber mehr Rechenaufwand; als Programmiersprache ungeeignet.

Wir wollen folgende Situation formalisieren:

„Der Dorfbarbier rasiert alle, die sich nicht selbst rasieren.“

Wir wollen folgende Situation formalisieren:

„Der Dorfbarbier rasiert alle, die sich nicht selbst rasieren.“

Frage: Wer rasiert den Dorfbarbier?

Wir wollen folgende Situation formalisieren:

„Der Dorfbarbier rasiert alle, die sich nicht selbst rasieren.“

Frage: Wer rasiert den Dorfbarbier?

Wir wollen zeigen: Die Situation ist widersprüchlich, d. h. ein solcher Dorfbarbier kann nicht existieren.

2.8.1 Formalisierung

1. „Der Dorfbarbier rasiert alle, die sich nicht selbst rasieren."

$$\forall \mathtt{B} \left(\mathtt{ba(B)} \rightarrow \forall \mathtt{P} \; (\neg \mathtt{ra(P,P)} \leftrightarrow \mathtt{ra(B,P)}) \right)$$

2. „Es gibt keinen Dorfbarbier."

$$\neg \exists \mathtt{B} \; \mathtt{ba(B)}$$

2.8.1 Formalisierung

1. „Der Dorfbarbier rasiert alle, die sich nicht selbst rasieren.“

$$\forall \mathtt{B}\ \big(\mathtt{ba(B)} \rightarrow \forall \mathtt{P}\ (\neg\, \mathtt{ra(P,P)} \leftrightarrow \mathtt{ra(B,P)}\,)\big)$$

2. „Es gibt keinen Dorfbarbier.“

$$\neg\, \exists \mathtt{B}\ \mathtt{ba(B)}$$

2.8.2 Anwendung des Widerlegungssystems

$$\{\forall \mathtt{B}\,\big(\mathtt{ba(B)} \rightarrow \forall \mathtt{P}\,(\,\neg\mathtt{ra(P,P)} \leftrightarrow \mathtt{ra(B,P)}\,)\big)\}$$

$$\Vdash\ \neg\exists \mathtt{B}\ \mathtt{ba(B)}$$

g.d.w.

$$\left\{\begin{array}{l}\forall \mathtt{B}\,\big(\mathtt{ba(B)} \rightarrow \forall \mathtt{P}\,(\,\neg\mathtt{ra(P,P)} \leftrightarrow \mathtt{ra(B,P)}\,)\big)\,, \\ \exists \mathtt{B}\ \mathtt{ba(B)}\end{array}\right\}$$

hat kein Modell.

2.8.3 Umwandlung in Klauselform

$$\left\{ \begin{array}{l} \forall \mathtt{B} \left(\mathtt{ba(B)} \rightarrow \forall \mathtt{P} \left(\neg \mathtt{ra(P,P)} \leftrightarrow \mathtt{ra(B,P)} \right) \right), \\ \exists \mathtt{B}\ \mathtt{ba(B)} \end{array} \right\}$$

($\Rightarrow$ pränexe Normalform)

$$\equiv \left\{ \begin{array}{l} \forall \mathtt{B} \forall \mathtt{P} \left(\mathtt{ba(B)} \rightarrow \left(\neg \mathtt{ra(P,P)} \leftrightarrow \mathtt{ra(B,P)} \right) \right), \\ \exists \mathtt{B}\ \mathtt{ba(B)} \end{array} \right\}$$

($\Rightarrow$ Skolemisierung)

$$\approx \left\{ \begin{array}{l} \mathtt{ba(B)} \rightarrow (\neg \mathtt{ra(P,P)} \leftrightarrow \mathtt{ra(B,P)}) , \\ \mathtt{ba(klaus)} \end{array} \right\}$$

($\Rightarrow$ konjunktive Normalform)

$$\equiv \left\{ \begin{array}{l} \mathtt{ba(B)} \\ \rightarrow (\neg \mathtt{ra(P,P)} \rightarrow \mathtt{ra(B,P)}) \\ \qquad \wedge (\neg \mathtt{ra(P,P)} \leftarrow \mathtt{ra(B,P)}) , \\ \mathtt{ba(klaus)} \end{array} \right\}$$

$$\equiv \left\{ \begin{array}{l} \neg\mathtt{ba(B)} \\ \vee \begin{array}{l} (\mathtt{ra(P,P)} \vee \mathtt{ra(B,P)}) \\ \wedge\ (\neg\mathtt{ra(P,P)} \vee \neg\mathtt{ra(B,P)})\ , \end{array} \\ \mathtt{ba(klaus)} \end{array} \right\}$$

$$\equiv \left\{ \begin{array}{l} (\neg\mathtt{ba(B)} \vee \mathtt{ra(P,P)} \vee \mathtt{ra(B,P)}) \\ \wedge\ (\neg\mathtt{ba(B)} \vee \neg\mathtt{ra(P,P)} \vee \neg\mathtt{ra(B,P)})\ , \\ \mathtt{ba(klaus)} \end{array} \right\}$$

$$\equiv \left\{ \begin{array}{l} \neg\mathtt{ba(B)} \lor \mathtt{ra(P,P)} \lor \mathtt{ra(B,P)}, \\ \neg\mathtt{ba(B)} \lor \neg\mathtt{ra(P,P)} \lor \neg\mathtt{ra(B,P)}, \\ \mathtt{ba(klaus)} \end{array} \right\}$$

($\Rightarrow$ Klauselform)

$$\equiv \left\{ \begin{array}{l} \{\neg\mathtt{ba(B)}, \mathtt{ra(P,P)}, \mathtt{ra(B,P)}\}, \\ \{\neg\mathtt{ba(B)}, \neg\mathtt{ra(P,P)}, \neg\mathtt{ra(B,P)}\}, \\ \{\mathtt{ba(klaus)}\} \end{array} \right\}$$

2.8.4 Anwendung der Resolution

$$\left\{\begin{array}{r}\neg\texttt{ba(B)},\\ \texttt{ra(P,P)},\\ \texttt{ra(B,P)}\end{array}\right\} \qquad \{\texttt{ba(klaus)}\} \qquad \left\{\begin{array}{r}\neg\texttt{ba(B)},\\ \neg\texttt{ra(P,P)},\\ \neg\texttt{ra(B,P)}\end{array}\right\}$$

$$\textbf{sub} = \left[{}^{\texttt{B}}/_{\texttt{klaus}}\right]$$

$$\left\{\begin{array}{r}\texttt{ra(P,P)},\\ \texttt{ra(klaus,P)}\end{array}\right\} \qquad \left\{\begin{array}{r}\neg\texttt{ra(P,P)},\\ \neg\texttt{ra(klaus,P)}\end{array}\right\}$$

$$\textbf{sub} = \left[{}^{\texttt{P}}/_{\texttt{klaus}}\right]$$

$$\square$$

1. Semantik:
   Ein Ziel zu beweisen, bedeutet zu beweisen,
   dass es aus dem Programm folgt.
2. Programmklauseln sind allquantifiziert,
   Zielklauseln existenzquantifiziert.
3. p, q bedeutet „p und q“.
4. Die Negation kommt nicht mehr vor.
5. Der Geltungsbereich einer Variablen
   ist die Klausel, in der sie vorkommt.
6. Die Resolutionsstrategie verwendet Tiefensuche.
7. Ziel und Programmklauselkopf werden unifiziert.

1. Semantik:
   Ein Ziel zu beweisen, bedeutet zu beweisen,
   dass es aus dem Programm folgt.

2. Programmklauseln sind allquantifiziert,
   Zielklauseln existenzquantifiziert.

3. p, q bedeutet „p und q“.

4. Die Negation kommt nicht mehr vor.

5. Der Geltungsbereich einer Variablen
   ist die Klausel, in der sie vorkommt.

6. Die Resolutionsstrategie verwendet Tiefensuche.

7. Ziel und Programmklauselkopf werden unifiziert.

1. Semantik:
   Ein Ziel zu beweisen, bedeutet zu beweisen,
   dass es aus dem Programm folgt.
2. Programmklauseln sind allquantifiziert,
   Zielklauseln existenzquantifiziert.
3. p, q bedeutet „p und q“.
4. Die Negation kommt nicht mehr vor.
5. Der Geltungsbereich einer Variablen
   ist die Klausel, in der sie vorkommt.
6. Die Resolutionsstrategie verwendet Tiefensuche.
7. Ziel und Programmklauselkopf werden unifiziert.

1. Semantik:
   Ein Ziel zu beweisen, bedeutet zu beweisen, dass es aus dem Programm folgt.

2. Programmklauseln sind allquantifiziert, Zielklauseln existenzquantifiziert.

3. p, q bedeutet „p und q“.

4. Die Negation kommt nicht mehr vor.

5. Der Geltungsbereich einer Variablen ist die Klausel, in der sie vorkommt.

6. Die Resolutionsstrategie verwendet Tiefensuche.

7. Ziel und Programmklauselkopf werden unifiziert.

1. Semantik:
   Ein Ziel zu beweisen, bedeutet zu beweisen,
   dass es aus dem Programm folgt.
2. Programmklauseln sind allquantifiziert,
   Zielklauseln existenzquantifiziert.
3. p, q bedeutet „p und q“.
4. Die Negation kommt nicht mehr vor.
5. Der Geltungsbereich einer Variablen
   ist die Klausel, in der sie vorkommt.
6. Die Resolutionsstrategie verwendet Tiefensuche.
7. Ziel und Programmklauselkopf werden unifiziert.

1. Semantik:
   Ein Ziel zu beweisen, bedeutet zu beweisen,
   dass es aus dem Programm folgt.
2. Programmklauseln sind allquantifiziert,
   Zielklauseln existenzquantifiziert.
3. `p, q` bedeutet „`p` und `q`“.
4. Die Negation kommt nicht mehr vor.
5. Der Geltungsbereich einer Variablen
   ist die Klausel, in der sie vorkommt.
6. Die Resolutionsstrategie verwendet Tiefensuche.
7. Ziel und Programmklauselkopf werden unifiziert.

1. Semantik:
   Ein Ziel zu beweisen, bedeutet zu beweisen, dass es aus dem Programm folgt.
2. Programmklauseln sind allquantifiziert, Zielklauseln existenzquantifiziert.
3. p, q bedeutet „p und q“.
4. Die Negation kommt nicht mehr vor.
5. Der Geltungsbereich einer Variablen ist die Klausel, in der sie vorkommt.
6. Die Resolutionsstrategie verwendet Tiefensuche.
7. Ziel und Programmklauselkopf werden unifiziert.

Gliederung

3.1.1 Literatur

- W. F. Clocksin and C. S. Mellish
  Programming in Prolog, 4th edition,
  Springer-Verlag 1994
- Leon Sterling and Ehud Shapiro
  The Art of Prolog, 2nd edition, MIT Press 1994
- Richard A. O'Keefe
  The Craft of Prolog, MIT Press 1990
- Ivan Bratko
  PROLOG Programming for Artificial Intelligence
  2nd edition, Addison-Wesley 1990

3.1.2 Logisches Programmieren

Grundlegende Idee:

Robert Kowalski

Algorithm = Logic + Control

Comm. ACM 22, 1979, pp. 424–436.

Logik: Was ist das Problem?

Kontrolle: Wie wird das Problem gelöst?

What-Type-Language: Der Benutzer spezifiziert das Problem abstrakt; das System berechnet die Lösung.

How-Type-Language: Der Benutzer spezifiziert eine Folge von Operationen, durch die das Problem gelöst wird.

In PROLOG:

- Abstrakte Beschreibung durch Fakten und Regeln.
- Das System stellt das Lösungsverfahren zur Verfügung

  logisch: Unifikation und Resolution.

  technisch: Matching und Nichtdeterminismus (Backtr.).

3.1.3 Geschichte

1965: Resolutionskalkül (J. A. Robinson)

1970–72 Erster PROLOG-Interpreter R. Kowalski, Edinburgh (Theorie; Horn-Klauseln)

A. Colmerauer, Marseille (Implementierung)

1975–79 Erster PROLOG-Compiler (D. Warren).

1980 Borland's Turbo PROLOG.

1982 Beginn des japanischen 5th Generation Projekts.

heute Viele Implementierungen, z. T. erheblich erweitert (z. B. um constraints).

3.1.4 Anwendungen

- Künstliche Intelligenz
- deduktive Datenbanken (Datalog)
- symbolische Mathematik
- *constraint programming*
- Konstruktion von Parsern, Interpretern und Compilern
- Konfigurationsaufgaben (z. B. NT-Netzwerkkonfiguration)

Allgemein:

strukturorientierte Verarbeitung symbolischer Daten.

3.1.5 Drei Perspektiven bzgl. PROLOG

1. Programming in Logic.
   Zu ausdrucksschwach $\leadsto$ Theorembeweiser
2. Database Query Language.
   Zu ausdrucksstark $\leadsto$ Datalog
3. Effiziente strukturorientierte Programmiersprache.

Programmieren in PROLOG umfaßt

1. Deklarieren von Fakten
2. Definieren von Regeln

$\left.\begin{array}{l}\text{1. Deklarieren von Fakten}\\ \text{2. Definieren von Regeln}\end{array}\right\}$ Programm

3. Stellen von Anfragen.

3.1.6 Fakten

Schreibweise:

```
mag(klaus, sabine).
```

Prädikat **Konstante** **Argumente**

3.1.6 Fakten

Schreibweise:

```
mag(klaus, sabine).
```

Prädikat **Konstante** **Argumente**

Prädikate und Konstanten beginnen mit einem Kleinbuchstaben.

3.1.6 Fakten

Schreibweise:

```
mag(klaus, sabine).
```

Prädikat **Konstante** **Argumente**

Prädikate und Konstanten beginnen mit einem Kleinbuchstaben.

Fakten deklarieren Eigenschaften von oder Relationen zwischen Objekten.

Beispiel 3.1.1

```
wertvoll(gold).

weiblich(heike).

vater(klaus,heike).

bruder_von(lukas,heike).
```

Beispiel 3.1.1

```
wertvoll(gold).
weiblich(heike).
vater(klaus,heike).
bruder_von(lukas,heike).
```

Die Bedeutung von Prädikaten und ihren Argumenten muß zu Anfang festgelegt und dann konsequent durchgehalten werden.

Beispiel 3.1.2

Zweistellige Prädikate werden infix verstanden:

```
bruder_von(lukas,heike)
```

$\leadsto$ `lukas bruder_von heike.`

Beispiel 3.1.2

Zweistellige Prädikate werden infix verstanden:

```
bruder_von(lukas,heike)
```

$\leadsto$ `lukas bruder_von heike.`

Die Gesamtheit aller Fakten (und Regeln) nennen wir Datenbasis oder logisches Programm.

3.1.7 Anfragen (oder Ziele)

Schreibweise:

```
?- wertvoll(gold).
```

3.1.7 Anfragen (oder Ziele)

Schreibweise:

```
?- wertvoll(gold).
```

Anfragen bewirken eine Suche in der Datenbasis (Beweis des Ziels aus dem logischen Programm).

Beispiel 3.1.3

```
weiblich(heike).        maennlich(klaus).
vater(klaus,heike).     bruder_von(lukas,heike).
```

```
?- maennlich(klaus).
Yes
?- bruder_von(lukas,heike).
Yes
?- maennlich(lukas).
No
```

Anfragen sind nicht Teil des Programms, sondern werden vom Benutzer an das System gestellt und von diesem beantwortet.

Anfragen sind nicht Teil des Programms, sondern werden vom Benutzer an das System gestellt und von diesem beantwortet.

3.1.8 Variablen

Variablen beginnen mit einem Großbuchstaben oder einem Unterstrich.

```
?- maennlich(Mann).
```

Anfragen sind nicht Teil des Programms, sondern werden vom Benutzer an das System gestellt und von diesem beantwortet.

3.1.8 Variablen

Variablen beginnen mit einem Großbuchstaben oder einem Unterstrich.

```
?- maennlich(Mann).
```

Variablen können mit einem Wert instantiiert werden und stehen dann für diesen Wert.

Anfragen mit Variablen

Beim Versuch, ein Ziel zu beweisen, wird das Ziel mit vorhandenen Fakten gematcht (unifiziert).

Beim matching dürfen Variablen beliebig instantiiert werden, alles nicht-variable muß exakt übereinstimmen.

In Anfragen sind Variablen existenzquantifiziert.

Antworten auf eine Anfrage sind erfüllende Belegungen sämtlicher Variablen.

Anfragen mit Variablen

Beim Versuch, ein Ziel zu beweisen, wird das Ziel mit vorhandenen Fakten gematcht (unifiziert).

Beim matching dürfen Variablen beliebig instantiiert werden, alles nicht-variable muß exakt übereinstimmen.

In Anfragen sind Variablen existenzquantifiziert.

Antworten auf eine Anfrage sind erfüllende Belegungen sämtlicher Variablen.

Anfragen mit Variablen

Beim Versuch, ein Ziel zu beweisen, wird das Ziel mit vorhandenen Fakten gematcht (unifiziert).

Beim matching dürfen Variablen beliebig instantiiert werden, alles nicht-variable muß exakt übereinstimmen.

In Anfragen sind Variablen existenzquantifiziert.

Antworten auf eine Anfrage sind erfüllende Belegungen sämtlicher Variablen.

Anfragen mit Variablen

Beim Versuch, ein Ziel zu beweisen, wird das Ziel mit vorhandenen Fakten gematcht (unifiziert).

Beim matching dürfen Variablen beliebig instantiiert werden, alles nicht-variable muß exakt übereinstimmen.

In Anfragen sind Variablen existenzquantifiziert.

Antworten auf eine Anfrage sind erfüllende Belegungen sämtlicher Variablen.

Beispiel 3.1.4

```
maennlich(thomas).      maennlich(klaus).
weiblich(heike).        weiblich(ingrid).
```

```
?- maennlich(Mann).
Mann = thomas ;
Mann = klaus ;
No
```

Variablen in Fakten

Auch Fakten dürfen Variablen enthalten. Eine solche Variable matcht alles, was in einem Ziel an dieser Argumentstelle steht.

Eine mehrfach vorkommende Variable muß an allen Stellen mit dem gleichen Wert instantiiert werden.

In Fakten sind Variablen allquantifiziert.

Variablen in Fakten

Auch Fakten dürfen Variablen enthalten. Eine solche Variable matcht alles, was in einem Ziel an dieser Argumentstelle steht.

Eine mehrfach vorkommende Variable muß an allen Stellen mit dem gleichen Wert instantiiert werden.

In Fakten sind Variablen allquantifiziert.

Variablen in Fakten

Auch Fakten dürfen Variablen enthalten. Eine solche Variable matcht alles, was in einem Ziel an dieser Argumentstelle steht.

Eine mehrfach vorkommende Variable muß an allen Stellen mit dem gleichen Wert instantiiert werden.

In Fakten sind Variablen allquantifiziert.

Beispiel 3.1.5

```
teilt(1,X).
teilt(X,X).
?- teilt(X,2).
X = 1 ;
X = 2 ;
No
```

3.1.9 Suchstrategie

Für jedes Ziel wird die Datenbasis von vorn nach dem ersten Fakt durchsucht, das das Ziel matcht (Variablen matchen alles).

Die Fundstelle wird markiert (choice point), die Variablen im Ziel (oder Fakt) werden instantiiert.

Es wird eine Antwort ausgegeben.

Wird eine weitere Lösung angefordert, wird die Instantiierung zurückgenommen und ab der markierten Stelle weitergesucht (Wiedererfüllung, REDO).

3.1.9 Suchstrategie

Für jedes Ziel wird die Datenbasis von vorn nach dem ersten Fakt durchsucht, das das Ziel matcht (Variablen matchen alles).

Die Fundstelle wird markiert (choice point), die Variablen im Ziel (oder Fakt) werden instantiiert.

Es wird eine Antwort ausgegeben.

Wird eine weitere Lösung angefordert, wird die Instantiierung zurückgenommen und ab der markierten Stelle weitergesucht (Wiedererfüllung, REDO).

3.1.9 Suchstrategie

Für jedes Ziel wird die Datenbasis von vorn nach dem ersten Fakt durchsucht, das das Ziel matcht (Variablen matchen alles).

Die Fundstelle wird markiert (choice point), die Variablen im Ziel (oder Fakt) werden instantiiert.

Es wird eine Antwort ausgegeben.

Wird eine weitere Lösung angefordert, wird die Instantiierung zurückgenommen und ab der markierten Stelle weitergesucht (Wiedererfüllung, REDO).

3.1.9 Suchstrategie

Für jedes Ziel wird die Datenbasis von vorn nach dem ersten Fakt durchsucht, das das Ziel matcht (Variablen matchen alles).

Die Fundstelle wird markiert (choice point), die Variablen im Ziel (oder Fakt) werden instantiiert.

Es wird eine Antwort ausgegeben.

Wird eine weitere Lösung angefordert, wird die Instantiierung zurückgenommen und ab der markierten Stelle weitergesucht (Wiedererfüllung, REDO).

3.1.10 Konjunktionen

```
?- mag(klaus, X), mag(heike, X).
```

„,“ wird als und gelesen.

3.1.10 Konjunktionen

```
?- mag(klaus, X), mag(heike, X).
```

„,“ wird als und gelesen.

Der Gültigkeitsbereich von Variablen ist die gesamte Konjunktion

$\leadsto$ `X` steht für „alles, was Klaus und Heike mögen“.

Erweiterte Suchstrategie

Variableninstantiierungen bleiben so lange fest, bis sie zurückgenommen werden.

Ist für eine gegebene Variableninstantiierung ein Teilziel nicht erfüllbar, so schlägt dieses fehl (fail), und es wird versucht, das vorherige Teilziel (choice point) wiederzuerfüllen (REDO).

$\rightsquigarrow$ Backtracking

Dabei werden evtl. Instantiierungen zurückgenommen.

Erweiterte Suchstrategie

Variableninstantiierungen bleiben so lange fest, bis sie zurückgenommen werden.

Ist für eine gegebene Variableninstantiierung ein Teilziel nicht erfüllbar, so schlägt dieses fehl (fail), und es wird versucht, das vorherige Teilziel (choice point) wiederzuerfüllen (REDO).

$\rightsquigarrow$ Backtracking

Dabei werden evtl. Instantiierungen zurückgenommen.

Erweiterte Suchstrategie

Variableninstantiierungen bleiben so lange fest, bis sie zurückgenommen werden.

Ist für eine gegebene Variableninstantiierung ein Teilziel nicht erfüllbar, so schlägt dieses fehl (fail), und es wird versucht, das vorherige Teilziel (choice point) wiederzuerfüllen (REDO).

$\rightsquigarrow$ Backtracking

Dabei werden evtl. Instantiierungen zurückgenommen.

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

$\left[{}^{\mathtt{X}}/_{\mathtt{kino}}\right]$

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

$\left[{}^{\mathtt{X}}/_{\mathtt{kino}}\right]$

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

$[{}^{X}/_{kino}]$

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

$\left[{}^{\texttt{X}}/_{\texttt{kino}}\right]$

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

$\left[{}^{\mathtt{X}}/_{\mathtt{kino}}\right]$

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

$\left[{}^{\texttt{X}}/_{\texttt{tanzen}}\right]$

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

$\left[{}^{\mathtt{X}}/\mathtt{tanzen}\right]$

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

$\left[{}^{\mathtt{X}}/_{\mathtt{tanzen}}\right]$

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

Beispiel 3.1.6

```
?- mag(klaus, X),
   mag(heike, X).
```

$\left[{}^{\texttt{X}}/_{\texttt{tanzen}}\right]$

```
mag(klaus, kino).
mag(klaus, tanzen).
mag(heike, tanzen).
mag(heike, fussball).
```

3.1.11 Regeln

```
bruder(X,Y) :-  maennlich(X),
                eltern(X,E1,E2),
                eltern(Y,E1,E2).
```

3.1.11 Regeln

```
bruder(X,Y) :-  maennlich(X),
                eltern(X,E1,E2),
                eltern(Y,E1,E2).
```

„`:-`“ wird als wenn gelesen.

3.1.11 Regeln

```
bruder(X,Y) :-  maennlich(X),
                eltern(X,E1,E2),
                eltern(Y,E1,E2).
```

„:-“ wird als wenn gelesen.

Der Teil vor dem :- heißt Kopf,
der Teil hinter dem :- heißt Rumpf der Regel.

3.1.11 Regeln

```
bruder(X,Y) :-  maennlich(X),
                eltern(X,E1,E2),
                eltern(Y,E1,E2).
```

„:-“ wird als wenn gelesen.

Der Teil vor dem :- heißt Kopf,
der Teil hinter dem :- heißt Rumpf der Regel.

Der Gültigkeitsbereich von Variablen ist die gesamte Regel.

Die Gesamtheit aller Fakten und Regeln für ein Prädikat nennen wir die Klauseln für dieses Prädikat.

Die Gesamtheit aller Fakten und Regeln für ein Prädikat nennen wir die Klauseln für dieses Prädikat.

Erweiterte Suchstrategie

Bei der Suche werden nicht nur Fakten, sondern auch Regelköpfe berücksichtigt.

Die Gesamtheit aller Fakten und Regeln für ein Prädikat nennen wir die Klauseln für dieses Prädikat.

Erweiterte Suchstrategie

Bei der Suche werden nicht nur Fakten, sondern auch Regelköpfe berücksichtigt.

Um eine Regel zu erfüllen, müssen zuerst sämtliche Ziele aus dem Rumpf erfüllt werden — Variableninstantiierung beachten!

```
?- bruder(X,heike).
```

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
```

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{\mathtt{Y}}/_{\mathtt{heike}}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

[Y/heike]

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{\mathtt{Y}}/_{\mathtt{heike}}\right]$

$\left[{}^{\mathtt{X}}/_{\mathtt{klaus}}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

$\left[{}^{X}/_{klaus}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

$\left[{}^{X}/_{klaus}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{\texttt{Y}}/_{\texttt{heike}}\right]$

$\left[{}^{\texttt{X}}/_{\texttt{klaus}}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{\texttt{Y}}/_{\texttt{heike}}\right]$

$\left[{}^{\texttt{X}}/_{\texttt{lukas}}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{\text{heike}}\right]$

$\left[{}^{X}/_{\text{lukas}}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

$\left[{}^{X}/_{lukas}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

$\left[{}^{X}/_{lukas}\right]$

$\left[{}^{E1}/_{klaus}\right]$

$\left[{}^{E2}/_{ingrid}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

$\left[{}^{X}/_{lukas}\right]$

$\left[{}^{E1}/_{klaus}\right]$

$\left[{}^{E2}/_{ingrid}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{heike}\right]$

$\left[{}^{X}/_{lukas}\right]$

$\left[{}^{E1}/_{klaus}\right]$

$\left[{}^{E2}/_{ingrid}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

```
?- bruder(X,heike).
   maennlich(X)
   eltern(X,E1,E2)
   eltern(Y,E1,E2)
```

$\left[{}^{Y}/_{\mathtt{heike}}\right]$

$\left[{}^{X}/_{\mathtt{lukas}}\right]$

$\left[{}^{E1}/_{\mathtt{klaus}}\right]$

$\left[{}^{E2}/_{\mathtt{ingrid}}\right]$

```
bruder(X,Y) :- maennlich(X),
               eltern(X,E1,E2),
               eltern(Y,E1,E2).
eltern(lukas,klaus,ingrid).
eltern(heike,klaus,ingrid).
weiblich(ingrid).
weiblich(heike).
maennlich(klaus).
maennlich(lukas).
```

3.1.12 Erste Schritte mit SWI-PROLOG

- Nach dem Programmstart mit `pl` wartet SWI-PROLOG auf Anfragen.
- Online-Hilfe mit `help`.
- Fakten und Regeln direkt eingeben mit `[user]`. Eingabe beenden mit `C-d`.
- Ein Programm laden mit `['`*Programmname*`']`.
- Nach einer Anfrage werden weitere Lösungen mit `;` angefordert.
- SWI-PROLOG beenden mit `halt.`

3.1.12 Erste Schritte mit SWI-PROLOG

- Nach dem Programmstart mit `pl` wartet SWI-PROLOG auf Anfragen.
- Online-Hilfe mit **`help`**.
- Fakten und Regeln direkt eingeben mit `[user]`. Eingabe beenden mit `C-d`.
- Ein Programm laden mit `['`*Programmname*`']`.
- Nach einer Anfrage werden weitere Lösungen mit `;` angefordert.
- SWI-PROLOG beenden mit **`halt.`**

3.1.12 Erste Schritte mit SWI-PROLOG

- Nach dem Programmstart mit `pl` wartet SWI-PROLOG auf Anfragen.
- Online-Hilfe mit `help`.
- Fakten und Regeln direkt eingeben mit `[user]`. Eingabe beenden mit `C-d`.
- Ein Programm laden mit `['`*Programmname*`']`.
- Nach einer Anfrage werden weitere Lösungen mit `;` angefordert.
- SWI-PROLOG beenden mit `halt.`

3.1.12 Erste Schritte mit SWI-PROLOG

- Nach dem Programmstart mit `pl` wartet SWI-PROLOG auf Anfragen.
- Online-Hilfe mit `help`.
- Fakten und Regeln direkt eingeben mit `[user]`. Eingabe beenden mit `C-d`.
- Ein Programm laden mit `['`*Programmname*`']`.
- Nach einer Anfrage werden weitere Lösungen mit `;` angefordert.
- SWI-PROLOG beenden mit `halt.`

3.1.12 Erste Schritte mit SWI-PROLOG

- Nach dem Programmstart mit `pl` wartet SWI-PROLOG auf Anfragen.
- Online-Hilfe mit `help`.
- Fakten und Regeln direkt eingeben mit `[user]`. Eingabe beenden mit `C-d`.
- Ein Programm laden mit `['`*Programmname*`']`.
- Nach einer Anfrage werden weitere Lösungen mit `;` angefordert.
- SWI-PROLOG beenden mit `halt.`

3.1.12 Erste Schritte mit SWI-PROLOG

- Nach dem Programmstart mit `pl` wartet SWI-PROLOG auf Anfragen.
- Online-Hilfe mit `help`.
- Fakten und Regeln direkt eingeben mit `[user]`. Eingabe beenden mit `C-d`.
- Ein Programm laden mit `['`*Programmname*`']`.
- Nach einer Anfrage werden weitere Lösungen mit `;` angefordert.
- SWI-PROLOG beenden mit `halt.`

3.1.13 Das Wichtigste in Kürze

1. Fakten und Regeln werden in Programmdateien mit der Endung `.pl` gespeichert.
2. Diese werden geladen und dann Anfragen gestellt.
3. Um ein Ziel zu beweisen, wird ein matchendes Faktum oder eine Regel mit matchendem Kopf gesucht.
4. Beim matching werden Variablen passend instantiiert.

3.1.13 Das Wichtigste in Kürze

1. Fakten und Regeln werden in Programmdateien mit der Endung .pl gespeichert.
2. Diese werden geladen und dann Anfragen gestellt.
3. Um ein Ziel zu beweisen, wird ein matchendes Faktum oder eine Regel mit matchendem Kopf gesucht.
4. Beim matching werden Variablen passend instantiiert.

3.1.13 Das Wichtigste in Kürze

1. Fakten und Regeln werden in Programmdateien mit der Endung `.pl` gespeichert.
2. Diese werden geladen und dann Anfragen gestellt.
3. Um ein Ziel zu beweisen, wird ein matchendes Faktum oder eine Regel mit matchendem Kopf gesucht.
4. Beim matching werden Variablen passend instantiiert.

3.1.13 Das Wichtigste in Kürze

1. Fakten und Regeln werden in Programmdateien mit der Endung `.pl` gespeichert.
2. Diese werden geladen und dann Anfragen gestellt.
3. Um ein Ziel zu beweisen, wird ein matchendes Faktum oder eine Regel mit matchendem Kopf gesucht.
4. Beim matching werden Variablen passend instantiiert.

5. Bei einem passenden Fakt ist ein Ziel sofort erfüllt, bei einer Regel müssen zuerst noch die Ziele aus dem Rumpf erfüllt werden.

6. Stehen für die Erfüllung eines Ziels mehrere Alternativen zur Verfügung, so wird ein choice point erzeugt.

7. Schlägt die Erfüllung eines Ziels fehl, so wird am zuletzt erzeugten choice point die nächste Alternative ausprobiert (Backtracking).

5. Bei einem passenden Fakt ist ein Ziel sofort erfüllt, bei einer Regel müssen zuerst noch die Ziele aus dem Rumpf erfüllt werden.
6. Stehen für die Erfüllung eines Ziels mehrere Alternativen zur Verfügung, so wird ein choice point erzeugt.
7. Schlägt die Erfüllung eines Ziels fehl, so wird am zuletzt erzeugten choice point die nächste Alternative ausprobiert (Backtracking).

5. Bei einem passenden Fakt ist ein Ziel sofort erfüllt, bei einer Regel müssen zuerst noch die Ziele aus dem Rumpf erfüllt werden.

6. Stehen für die Erfüllung eines Ziels mehrere Alternativen zur Verfügung, so wird ein choice point erzeugt.

7. Schlägt die Erfüllung eines Ziels fehl, so wird am zuletzt erzeugten choice point die nächste Alternative ausprobiert (Backtracking).

3.2.1 Terme

PROLOG-Programme sind aus Termen aufgebaut. Wir unterscheiden drei Typen.

Konstanten

Zwei Arten von Konstanten:

Atome dienen als Bezeichner.

Beginnen normalerweise mit einem Kleinbuchstaben und enthalten Buchstaben, Ziffern oder _.

Beispiele:

`bruder`   `heike`   `helmut_Kohl`   `nummer5`

Man kann beliebige Zeichenfolgen verwenden, wenn man diese in Hochkommata einschließt:

`'KeineVariable'`   `'12'`   `'rechts-links'`

Auch beliebige Folgen der folgenden Zeichen sind Atome:

`= + - * / \ ~ ^ < > : . ? @ # $ &`

Beispiele:

`=`   `$$$`   `>=`   `==>`   `?-`   `\-/`

Zahlen 1. Ganze Zahlen:

`0 1 999 123456789 -17`

2. Gleitkommazahlen:

`1.2 -15.0 1.3e5 -17.7e-5.`

Zahlen 1. Ganze Zahlen:

`0 1 999 123456789 -17`

2. Gleitkommazahlen:

`1.2 -15.0 1.3e5 -17.7e-5.`

Variablen

Beginnen mit einem Großbuchstaben oder _ und enthalten Buchstaben, Ziffern oder _.

`X L Mensch V_23 _intern`

Sonderfall _: anonyme Variable. Mehrere anonyme Variablen in einer Klausel dürfen verschieden instantiiert werden!

```
maennlich(X) :- eltern(_,X,_).

?- eltern(X,_,_).
X = lukas ;
X = heike ;
No
```

Strukturen

$$\boxed{\mathtt{f}(T_1,\ \ldots,\ T_n).}$$

Funktor **Term** **Argumente** **Arität**

Der Funktor ist ein Atom, die Argumente beliebige Terme (rekursive Definition).

Strukturen

$$\boxed{f(T_1, \ldots, T_n).}$$

Funktor **Term** **Argumente** **Arität**

Der Funktor ist ein Atom, die Argumente beliebige Terme (rekursive Definition).

Einen Funktor zusammen mit seiner Arität schreiben wir f/n. Verschiedenstellige Funktoren gelten als verschieden!

Strukturen können beliebig verschachtelt werden:

```
besitzt(lukas,buch).

besitzt(lukas,buch(momo,ende)).

besitzt(lukas,buch(momo,autor(ende,michael))).
```

3.2.2 Operatoren

Ein- und zweistellige Funktoren können als Operatoren deklariert werden.

Dies ergibt eine bequemere Schreibweise für Terme.

3.2.2 Operatoren

Ein- und zweistellige Funktoren können als Operatoren deklariert werden.

Dies ergibt eine bequemere Schreibweise für Terme.

Achtung: Operatoren bewirken keine Berechnung, sondern lediglich eine andere Schreibweise für Terme.

2+3 und 3+2 sind verschiedene Terme!

Position

Einstellige Operatoren können Präfix- oder Postfixoperatoren sein. Beispiel:

$$\texttt{-X} \rightsquigarrow \texttt{-(X)} \qquad \texttt{X!} \rightsquigarrow \texttt{!(X)}$$

Position

Einstellige Operatoren können Präfix- oder Postfixoperatoren sein. Beispiel:

$$\texttt{-X} \rightsquigarrow \texttt{-(X)} \qquad \texttt{X!} \rightsquigarrow \texttt{!(X)}$$

Zweistellige Operatoren sind immer Infixoperatoren.

$$\texttt{X+1} \rightsquigarrow \texttt{+(X,1)} \qquad \texttt{X >= Y} \rightsquigarrow \texttt{>=(X,Y)}$$

3.2.3 Matching und Termvergleich

Matching von Termen wird bei der Suche nach einer für ein Ziel passenden Klausel gebraucht.

3.2.3 Matching und Termvergleich

Matching von Termen wird bei der Suche nach einer für ein Ziel passenden Klausel gebraucht.

Seien $\mathbf{T_1}$, $\mathbf{T_2}$ Terme.

Rekursive Definition für „matche $\mathbf{T_1}$ und $\mathbf{T_2}$“:

1. $\mathbf{T_1}$ uninstantiierte Variable $\mathbf{V}$
   $\leadsto$ instantiiere $\mathbf{V}$ mit $\mathbf{T_2}$.
2. $\mathbf{T_2}$ uninstantiierte Variable $\boldsymbol{W}$
   $\leadsto$ instantiiere $\boldsymbol{W}$ mit $\mathbf{T_1}$.

3.2.3 Matching und Termvergleich

Matching von Termen wird bei der Suche nach einer für ein Ziel passenden Klausel gebraucht.

Seien T_1, T_2 Terme.

Rekursive Definition für „matche T_1 und T_2“:

1. T_1 uninstantiierte Variable V
   $\rightsquigarrow$ instantiiere V mit T_2.
2. T_2 uninstantiierte Variable W
   $\rightsquigarrow$ instantiiere W mit T_1.

3. Sind T_1, T_2 (instantiiert zu) Konstanten, so matchen T_1, T_2 genau dann, wenn sie identisch sind.

4. (Induktionsschritt:) Sind T_1, T_2 Strukturen, so matche erst die Funktoren und dann die Argumente (Stelle für Stelle).

3. Sind T_1, T_2 (instantiiert zu) Konstanten, so matchen T_1, T_2 genau dann, wenn sie identisch sind.

4. (Induktionsschritt:) Sind T_1, T_2 Strukturen, so matche erst die Funktoren und dann die Argumente (Stelle für Stelle).

3. Sind T_1, T_2 (instantiiert zu) Konstanten, so matchen T_1, T_2 genau dann, wenn sie identisch sind.

4. (Induktionsschritt:) Sind T_1, T_2 Strukturen, so matche erst die Funktoren und dann die Argumente (Stelle für Stelle).

Ausgabe: „Fehlschlag“ oder Terme T_1', T_2', die (nach Variableneinsetzung) identisch sind.

3. Sind T_1, T_2 (instantiiert zu) Konstanten, so matchen T_1, T_2 genau dann, wenn sie identisch sind.

4. (Induktionsschritt:) Sind T_1, T_2 Strukturen, so matche erst die Funktoren und dann die Argumente (Stelle für Stelle).

Ausgabe: „Fehlschlag“ oder Terme T_1', T_2', die (nach Variableneinsetzung) identisch sind.

Achtung! Kein occur check,
also sind ‘unendliche’ Terme möglich!

```
?- a(X,f(X),f(Y,Y)) = a(g(A),B,f(h(B),C)),
|     write(a(X,f(X),f(Y,Y))),nl,write(a(g(A),B,f(h(B),C))).
a(g(_G710), f(g(_G710)), f(h(f(g(_G710))), h(f(g(_G710)))))
a(g(_G710), f(g(_G710)), f(h(f(g(_G710))), h(f(g(_G710)))))

X = g(_G710)
Y = h(f(g(_G710)))
A = _G710
B = f(g(_G710))
C = h(f(g(_G710))) ;

No
```

```
?-  X = f(X).

X = f(f(f(f(f(f(f(f(f(f(...)))))))))) ;

No
```

Vergleich von Termen

```
T1 = T2.
```

T1 und T2 matchen.

Vergleich von Termen

```
T1 = T2.
```

T1 und T2 matchen.

```
T1 \= T2.
```

T1 und T2 matchen nicht.

Durch \= werden keine Variablen instantiiert.

```
T1 == T2.
```

T1 und T2 sind identisch.

Durch == werden keine Variablen instantiiert.

Beim Test == werden Variablen als verschieden gewertet, wenn sie nicht bereits vorher identifiziert wurden.

== ist nicht mit der logischen Sichtweise auf PROLOG vereinbar.

```
T1 == T2.
```

T1 und T2 sind identisch.

Durch == werden keine Variablen instantiiert.

Beim Test == werden Variablen als verschieden gewertet, wenn sie nicht bereits vorher identifiziert wurden.

== ist nicht mit der logischen Sichtweise auf PROLOG vereinbar.

```
T1 == T2.
```

T1 und T2 sind identisch.

Durch == werden keine Variablen instantiiert.

Beim Test == werden Variablen als verschieden gewertet, wenn sie nicht bereits vorher identifiziert wurden.

== ist nicht mit der logischen Sichtweise auf PROLOG vereinbar.

```
T1 == T2.
```

T1 und T2 sind identisch.

Durch == werden keine Variablen instantiiert.

Beim Test == werden Variablen als verschieden gewertet, wenn sie nicht bereits vorher identifiziert wurden.

== ist nicht mit der logischen Sichtweise auf PROLOG vereinbar.

```
?- f(X) == f(Y).

No
?- X = Y, f(X) == f(Y).

X = _G257
Y = _G257 ;

No
```

```
T1 \== T2.
```

T1 und T2 sind nicht identisch.

Durch \== werden keine Variablen instantiiert.

Vergleich gemäß der Standardordnung für (endliche) Terme: `@<`, `@>`, `@=<`, `@>=`.

Durch diese Tests werden keine Variablen instantiiert.

Vergleich gemäß der Standardordnung für (endliche) Terme: `@<`, `@>`, `@=<`, `@>=`.

Durch diese Tests werden keine Variablen instantiiert.

```
compare(Op,T1,T2).
```

`Op` ist das Ergebnis der Vergleichs der Terme `T1`, `T2` gemäß der Standardordnung. Mögliche Werte: `<`, `>`, `=`.

```
?- compare(Op,X,Y).

Op = <
X = _G234
Y = _G235 ;

No
?- compare(=,X,Y).

No
```

```
?- X = Y, compare(Op,X,Y).

X = _G275
Y = _G275
Op = = ;

No
?- compare(Op,1+2,2+1).

Op = < ;

No
```

3.2.4 Programme

Ein PROLOG-Programm ist eine Folge von Termen, jeweils gefolgt von einem `.`

3.2.4 Programme

Ein PROLOG-Programm ist eine Folge von Termen, jeweils gefolgt von einem .

Kommentare werden mit % abgetrennt und reichen bis zum Ende der Zeile oder werden in /*...*/ eingeschlossen.

3.2.5 Das Wichtigste in Kürze

1. PROLOG-Programme sind aus Termen aufgebaut.
2. Terme sind Konstanten (Atome, Zahlen), Variablen oder Strukturen, in denen Terme als Argumente eines Funktors auftreten.
3. Erklärt man einen ein- oder zweistelligen Funktor zum Operator, darf man ihn in Termen prefix, postfix oder infix verwenden.
4. Mit `T1 = T2` wird festgestellt, ob `T1` und `T2` matchen.

3.2.5 Das Wichtigste in Kürze

1. PROLOG-Programme sind aus Termen aufgebaut.
2. Terme sind Konstanten (Atome, Zahlen), Variablen oder Strukturen, in denen Terme als Argumente eines Funktors auftreten.
3. Erklärt man einen ein- oder zweistelligen Funktor zum Operator, darf man ihn in Termen prefix, postfix oder infix verwenden.
4. Mit `T1 = T2` wird festgestellt, ob `T1` und `T2` matchen.

3.2.5 Das Wichtigste in Kürze

1. PROLOG-Programme sind aus Termen aufgebaut.
2. Terme sind Konstanten (Atome, Zahlen), Variablen oder Strukturen, in denen Terme als Argumente eines Funktors auftreten.
3. Erklärt man einen ein- oder zweistelligen Funktor zum Operator, darf man ihn in Termen prefix, postfix oder infix verwenden.
4. Mit `T1 = T2` wird festgestellt, ob `T1` und `T2` matchen.

3.2.5 Das Wichtigste in Kürze

1. PROLOG-Programme sind aus Termen aufgebaut.
2. Terme sind Konstanten (Atome, Zahlen), Variablen oder Strukturen, in denen Terme als Argumente eines Funktors auftreten.
3. Erklärt man einen ein- oder zweistelligen Funktor zum Operator, darf man ihn in Termen prefix, postfix oder infix verwenden.
4. Mit `T1 = T2` wird festgestellt, ob `T1` und `T2` matchen.

5. Mit `T1 == T2` wird festgestellt, ob `T1` und `T2` identisch sind.

6. Kommentare werden mit `%` abgetrennt oder in `/*...*/` eingeschlossen.

5. Mit `T1 == T2` wird festgestellt, ob `T1` und `T2` identisch sind.

6. Kommentare werden mit `%` abgetrennt oder in `/*...*/` eingeschlossen.

Gegeben ein Programm **P** und eine Folge **Z** von Zielen.

1. Wähle das nächste Ziel **L** (ist **Z** leer $\leadsto$ Antwort).
2. Wähle die nächste Klausel **K** im Programm, deren Kopf mit **L** matcht.
3. Wenn es noch eine weitere Klausel gibt, die in Frage kommt, erzeuge einen choice point.
4. Instantiiere Variablen, so daß **L** und der Kopf von **K** matchen.
5. Hänge den Rumpf von **K** vorn an **Z** an.
6. Beginne von vorn.

Gegeben ein Programm **P** und eine Folge **Z** von Zielen.

1. Wähle das nächste Ziel **L** (ist **Z** leer $\leadsto$ Antwort).
2. Wähle die nächste Klausel **K** im Programm, deren Kopf mit **L** matcht.
3. Wenn es noch eine weitere Klausel gibt, die in Frage kommt, erzeuge einen choice point.
4. Instantiiere Variablen, so daß **L** und der Kopf von **K** matchen.
5. Hänge den Rumpf von **K** vorn an **Z** an.
6. Beginne von vorn.

Gegeben ein Programm **P** und eine Folge **Z** von Zielen.

1. Wähle das nächste Ziel **L** (ist **Z** leer $\leadsto$ Antwort).
2. Wähle die nächste Klausel **K** im Programm, deren Kopf mit **L** matcht.
3. Wenn es noch eine weitere Klausel gibt, die in Frage kommt, erzeuge einen choice point.
4. Instantiiere Variablen, so daß **L** und der Kopf von **K** matchen.
5. Hänge den Rumpf von **K** vorn an **Z** an.
6. Beginne von vorn.

Gegeben ein Programm **P** und eine Folge **Z** von Zielen.

1. Wähle das nächste Ziel **L** (ist **Z** leer $\rightsquigarrow$ Antwort).
2. Wähle die nächste Klausel **K** im Programm, deren Kopf mit **L** matcht.
3. Wenn es noch eine weitere Klausel gibt, die in Frage kommt, erzeuge einen choice point.
4. Instantiiere Variablen, so daß **L** und der Kopf von **K** matchen.
5. Hänge den Rumpf von **K** vorn an **Z** an.
6. Beginne von vorn.

Gegeben ein Programm **P** und eine Folge **Z** von Zielen.

1. Wähle das nächste Ziel **L** (ist **Z** leer $\leadsto$ Antwort).
2. Wähle die nächste Klausel **K** im Programm, deren Kopf mit **L** matcht.
3. Wenn es noch eine weitere Klausel gibt, die in Frage kommt, erzeuge einen choice point.
4. Instantiiere Variablen, so daß **L** und der Kopf von **K** matchen.
5. Hänge den Rumpf von **K** vorn an **Z** an.
6. Beginne von vorn.

Gegeben ein Programm **P** und eine Folge **Z** von Zielen.

1. Wähle das nächste Ziel **L** (ist **Z** leer $\leadsto$ Antwort).
2. Wähle die nächste Klausel **K** im Programm, deren Kopf mit **L** matcht.
3. Wenn es noch eine weitere Klausel gibt, die in Frage kommt, erzeuge einen choice point.
4. Instantiiere Variablen, so daß **L** und der Kopf von **K** matchen.
5. Hänge den Rumpf von **K** vorn an **Z** an.
6. Beginne von vorn.

7. Schlägt die Suche fehl oder wird eine weitere Lösung angefordert (redo), fahre am letzten choice point fort (Backtracking).

3.3.1 First Argument Indexing

Wann kommt eine Klausel „in Frage“?

$\rightsquigarrow$ betrachte den Funktor des ersten Arguments.

3.3.1 First Argument Indexing

Wann kommt eine Klausel „in Frage“?

$\leadsto$ betrachte den Funktor des ersten Arguments.

```
eltern(lukas, klaus, ingrid).
eltern(heike,klaus,ingrid).

?- eltern(lukas,V,M).

V = klaus
M = ingrid ;

No
```

```
eltern(sohn(lukas), vater(klaus), mutter(ingrid)).
eltern(sohn(klaus), vater(gustav), mutter(gertrud)).

?- eltern(sohn(lukas),V,M).

V = vater(klaus)
M = mutter(ingrid) ;
 T Redo: (6) eltern(sohn(lukas), _G410, _G411)

No
```

```
eltern(lukas, klaus, ingrid).
eltern(klaus, gustav, gertrud).

?- eltern(K,V,ingrid).

K = lukas
V = klaus ;
 T Redo: (6) eltern(_G377, _G378, ingrid)

No
```

3.3.2 (nicht-)Determinismus

Ein Prädikat heißt deterministisch, wenn nach Erzeugen der ersten Lösung keine choice points zurückbleiben.

3.3.3 Das Wichtigste in Kürze

1. Um ein Ziel zu erfüllen, wird nach passenden Klauseln gesucht.
2. Ob eine Klausel paßt, wird mit first argument indexing ermittelt.
3. Gibt es mehrere passende Klauseln, wird ein choice point erzeugt.
4. Führt die Ausführung eines Prädikats nicht zur Erzeugung von choice points, nennt man dieses deterministisch.

3.3.3 Das Wichtigste in Kürze

1. Um ein Ziel zu erfüllen, wird nach passenden Klauseln gesucht.
2. Ob eine Klausel paßt, wird mit first argument indexing ermittelt.
3. Gibt es mehrere passende Klauseln, wird ein choice point erzeugt.
4. Führt die Ausführung eines Prädikats nicht zur Erzeugung von choice points, nennt man dieses deterministisch.

3.3.3 Das Wichtigste in Kürze

1. Um ein Ziel zu erfüllen, wird nach passenden Klauseln gesucht.
2. Ob eine Klausel paßt, wird mit first argument indexing ermittelt.
3. Gibt es mehrere passende Klauseln, wird ein choice point erzeugt.
4. Führt die Ausführung eines Prädikats nicht zur Erzeugung von choice points, nennt man dieses deterministisch.

3.3.3 Das Wichtigste in Kürze

1. Um ein Ziel zu erfüllen, wird nach passenden Klauseln gesucht.
2. Ob eine Klausel paßt, wird mit first argument indexing ermittelt.
3. Gibt es mehrere passende Klauseln, wird ein choice point erzeugt.
4. Führt die Ausführung eines Prädikats nicht zur Erzeugung von choice points, nennt man dieses deterministisch.

Problem: Mit den Vergleichsoperatoren in 3.2.11 kann man Zahlen vergleichen. Arithmetische Ausdrücke wie z. B. 1+2 können nicht gemäß des von ihnen repräsentierten Zahlenwerts verglichen werden.

3.4.1 Arithmetische Ausdrücke

Induktive Definition: Ein arithmetischer Ausdruck ist

1. eine Zahl oder
2. eine Struktur, deren Funktor ein arithmetischer Operator und deren Argumente arithmetische Ausdrücke sind oder
3. eine Variable, die mit einen arithmetischen Ausdruck instantiiert ist.

3.4.1 Arithmetische Ausdrücke

Induktive Definition: Ein arithmetischer Ausdruck ist

1. eine Zahl oder
2. eine Struktur, deren Funktor ein arithmetischer Operator und deren Argumente arithmetische Ausdrücke sind oder
3. eine Variable, die mit einen arithmetischen Ausdruck instantiiert ist.

3.4.1 Arithmetische Ausdrücke

Induktive Definition: Ein arithmetischer Ausdruck ist

1. eine Zahl oder
2. eine Struktur, deren Funktor ein arithmetischer Operator und deren Argumente arithmetische Ausdrücke sind oder
3. eine Variable, die mit einen arithmetischen Ausdruck instantiiert ist.

3.4.2 Arithmetische Operatoren

Beispiele:

Ausdruck	steht für
`-A`	Negation
`A1 + A2`	Summe
`A1 - A2`	Differenz
`A1 * A2`	Produkt
`A1 / A2`	Division
`A1 ** A2`	Exponenzierung
`A1 // A2`	Integer-Anteil der Division (`A1`, `A2` ganz)
`A1 mod A2`	Modulo (`A1` und `A2` ganz)
`abs(A)`	Absolutwert

3.4.3 Auswertung eines arithmetischen Ausdrucks

`is(?,+)`	`X is Ausd`

`Ausd` wird gemäß den Regeln der Arithmetik ausgewertet; das Ergebnis (Zahl) wird mit `X` gematcht.

3.4.4 Vergleich arithmetischer Ausdrücke

Argumente werden ausgewertet und die Ergebnisse verglichen.

Vergleich	Relation
`A1 =:= A2`	Gleich
`A1 =\= A2`	Ungleich
`A1 < A2`	Kleiner
`A1 > A2`	Größer
`A1 >= A2`	Größer oder Gleich
`A1 =< A2`	Kleiner oder Gleich

3.4.5 Das Wichtigste in Kürze

1. Ein arithmetischer Ausdruck darf keine uninstantiierten Variablen enthalten.
2. `is(?,+)` wertet einen arithmetischen Ausdruck aus und matcht das Ergebnis mit einer Variablen.
3. `=:=`, `=\=`, `<`, `>`, `>=`, `=<` werten zwei arithmetische Ausdrücke aus und vergleichen die Ergebnisse.

3.4.5 Das Wichtigste in Kürze

1. Ein arithmetischer Ausdruck darf keine uninstantiierten Variablen enthalten.
2. `is(?,+)` wertet einen arithmetischen Ausdruck aus und matcht das Ergebnis mit einer Variablen.
3. `=:=`, `=\=`, `<`, `>`, `>=`, `=<` werten zwei arithmetische Ausdrücke aus und vergleichen die Ergebnisse.

3.4.5 Das Wichtigste in Kürze

1. Ein arithmetischer Ausdruck darf keine uninstantiierten Variablen enthalten.
2. `is(?,+)` wertet einen arithmetischen Ausdruck aus und matcht das Ergebnis mit einer Variablen.
3. `=:=`, `=\=`, `<`, `>`, `>=`, `=<` werten zwei arithmetische Ausdrücke aus und vergleichen die Ergebnisse.

Beispiel 3.5.1 (Euklidischer Algorithmus)

gcd(+I,+J,?G) berechnet den größten gemeinsamen Teiler von I und J und matcht das Ergebnis mit G.

IB: Ist J $= 0$, so soll G = I gelten.

```
gcd(I,0,I).
```

IS: Ist J > 0, berechne gcd(J, I mod J, G).

```
gcd(I,J,Gcd) :-
    J > 0, K is I mod J,  gcd(J,K,Gcd) .
```

Beispiel 3.5.1 (Euklidischer Algorithmus)

`gcd(+I,+J,?G)` berechnet den größten gemeinsamen Teiler von `I` und `J` und matcht das Ergebnis mit `G`.

IB: Ist $J = 0$, so soll `G = I` gelten.

```
gcd(I,0,I).
```

IS: Ist $J > 0$, berechne `gcd(J, I mod J, G)`.

```
gcd(I,J,Gcd) :-
    J > 0, K is I mod J,  gcd(J,K,Gcd) .
```

Beispiel 3.5.1 (Euklidischer Algorithmus)

`gcd(+I,+J,?G)` berechnet den größten gemeinsamen Teiler von `I` und `J` und matcht das Ergebnis mit `G`.

IB: Ist $J = 0$, so soll `G = I` gelten.

```
gcd(I,0,I).
```

IS: Ist $J > 0$, berechne `gcd(J, I mod J, G)`.

```
gcd(I,J,Gcd) :-
    J > 0, K is I mod J, gcd(J,K,Gcd).
```

IH

Beispiel 3.5.1 (Euklidischer Algorithmus)

`gcd(+I,+J,?G)` berechnet den größten gemeinsamen Teiler von `I` und `J` und matcht das Ergebnis mit `G`.

IB: Ist $J = 0$, so soll `G = I` gelten.

```
gcd(I,0,I).
```

IS: Ist $J > 0$, berechne `gcd(J, I mod J, G)`.

```
gcd(I,J,Gcd) :-
    J > 0, K is I mod J, gcd(J,K,Gcd).
```

IH

Dank last call optimization ist `gcd` iterativ !

Beispiel 3.5.2 (Fakultät) `fact(+N,?F)` berechnet die Fakultät von `N` und matcht das Ergebnis mit `F`.

IB: Ist N $= 0$, so soll `F = 1` gelten.

```
fact(0,1).
```

IS: Ist N > 0, berechne `fact(N-1, F')`;
es soll `F is N * F'` gelten.

```
fact(N,F) :- N > 0,
    N1 is N - 1, fact(N1,F1),
    F is N*F1.
```

Beispiel 3.5.2 (Fakultät) `fact(+N,?F)` berechnet die Fakultät von `N` und matcht das Ergebnis mit `F`.

IB: Ist `N` $= 0$, so soll `F = 1` gelten.

```
fact(0,1).
```

IS: Ist `N` > 0, berechne `fact(N-1, F')`;
es soll `F is N * F'` gelten.

```
fact(N,F) :- N > 0,
   N1 is N - 1, fact(N1,F1),
   F is N*F1.
```

Beispiel 3.5.2 (Fakultät) `fact(+N,?F)` berechnet die Fakultät von `N` und matcht das Ergebnis mit `F`.

IB: Ist `N` $= 0$, so soll `F = 1` gelten.

```
fact(0,1).
```

IS: Ist `N` > 0, berechne `fact(N-1, F')`;
es soll `F is N * F'` gelten.

```
fact(N,F) :- N > 0,
  N1 is N - 1, fact(N1,F1),
  F is N*F1.
```

Beispiel 3.5.2 (Fakultät) `fact(+N,?F)` berechnet die Fakultät von `N` und matcht das Ergebnis mit `F`.

IB: Ist N $= 0$, so soll `F = 1` gelten.

```
fact(0,1).
```

IS: Ist N > 0, berechne `fact(N-1, F')`;
es soll `F is N * F'` gelten.

```
fact(N,F) :- N > 0,
   N1 is N - 1, fact(N1,F1),
   F is N*F1.
```

`fact` ist nicht iterativ!

Beispiel 3.5.3 (Fakultät iterativ)

Hilfsprädikat `fact(+N,+A,?F)`:

N wird heruntergezählt und dabei $A = N * (N - 1) * \ldots$ akkumuliert.

Um `fact(N,F)` zu beweisen, beweise `fact(N,1,F)`.

```
fact(N,F) :- fact(N,1,F).
```

Ist $N = 0$, soll `F = A` gelten.

```
fact(0,F,F).
```

Ist $N > 0$, berechne `fact(N-1,A*N,F)`.

```
fact(N,A,F) :- N > 0,
   N1 is N-1, A1 is A*N, fact(N1,A1,F).
```

Beispiel 3.5.3 (Fakultät iterativ)

Hilfsprädikat `fact(+N,+A,?F)`:

N wird heruntergezählt und dabei A = N * (N − 1) * ... akkumuliert.

Um `fact(N,F)` zu beweisen, beweise `fact(N,1,F)`.

```
fact(N,F) :- fact(N,1,F).
```

Ist N = 0, soll F = A gelten.

```
fact(0,F,F).
```

Ist N > 0, berechne `fact(N-1,A*N,F)`.

```
fact(N,A,F) :- N > 0,
   N1 is N-1, A1 is A*N, fact(N1,A1,F).
```

Beispiel 3.5.3 (Fakultät iterativ)

Hilfsprädikat `fact(+N,+A,?F)`:

N wird heruntergezählt und dabei $A = N * (N - 1) * \ldots$ akkumuliert.

Um `fact(N,F)` zu beweisen, beweise `fact(N,1,F)`.

```
fact(N,F) :- fact(N,1,F).
```

Ist $N = 0$, soll `F = A` gelten.

```
fact(0,F,F).
```

Ist $N > 0$, berechne `fact(N-1,A*N,F)`.

```
fact(N,A,F) :- N > 0,
    N1 is N-1, A1 is A*N, fact(N1,A1,F).
```

Beispiel 3.5.3 (Fakultät iterativ)

Hilfsprädikat `fact(+N,+A,?F)`:

N wird heruntergezählt und dabei $A = N * (N - 1) * \ldots$ akkumuliert.

Um `fact(N,F)` zu beweisen, beweise `fact(N,1,F)`.

```
fact(N,F) :- fact(N,1,F).
```

Ist $N = 0$, soll F = A gelten.

```
fact(0,F,F).
```

Ist $N > 0$, berechne `fact(N-1,A*N,F)`.

```
fact(N,A,F) :- N > 0,
   N1 is N-1, A1 is A*N, fact(N1,A1,F).
```

Beispiel 3.5.4 (Prädikat mit mehreren Lösungen)

`inbetween(+I,+J,?K)` ist erfüllt,
wenn `K` zwischen `I` und `K` liegt (incl.).

Wenn `I` $\leq$ `J` gilt, ist `K = I` eine Lösung.

```
inbetween(I,J,I) :- I =< J.
```

Wenn `I` $<$ `J` gilt, ist auch `K` mit `between(I+1,J,K)` eine Lösung.

```
inbetween(I,J,K) :-
   I < J, I1 is I+1, inbetween(I1,J,K).
```

Beispiel 3.5.4 (Prädikat mit mehreren Lösungen)

`inbetween(+I,+J,?K)` ist erfüllt,
wenn `K` zwischen `I` und `K` liegt (incl.).

Wenn `I` $\leq$ `J` gilt, ist `K = I` eine Lösung.

```
inbetween(I,J,I) :- I =< J.
```

Wenn `I` < `J` gilt, ist auch `K` mit `between(I+1,J,K)` eine Lösung.

```
inbetween(I,J,K) :-
  I < J, I1 is I+1, inbetween(I1,J,K).
```

Beispiel 3.5.4 (Prädikat mit mehreren Lösungen)

`inbetween(+I,+J,?K)` ist erfüllt,
wenn `K` zwischen `I` und `K` liegt (incl.).

Wenn `I` $\leq$ `J` gilt, ist `K = I` eine Lösung.

```
    inbetween(I,J,I) :- I =< J.
```

Wenn `I` $<$ `J` gilt, ist auch `K` mit `between(I+1,J,K)` eine Lösung.

```
    inbetween(I,J,K) :-
      I < J, I1 is I+1, inbetween(I1,J,K).
```

```
?- inbetween(4,7,N).

N = 4 ;

N = 5 ;

N = 6 ;

N = 7 ;

No
```

3.5.1 Das Wichtigste in Kürze

1. Ein rekursives Prädikat besitzt eine Regel, die einen Aufruf desselben Prädikats als Teilziel enthält.

2. Ein rekursives Prädikat ist immer induktiv aufgebaut:

   Die Induktionsbasis (Abbruchbedingung) ist ein ausreichend einfacher Fall, daß das Ziel ohne rekursiven Aufruf bewiesen werden kann.

   Im Induktionsschritt wird die Induktionshypothese (rekursiver Aufruf) angewendet.

3.5.1 Das Wichtigste in Kürze

1. Ein rekursives Prädikat besitzt eine Regel, die einen Aufruf desselben Prädikats als Teilziel enthält.

2. Ein rekursives Prädikat ist immer induktiv aufgebaut:

   Die Induktionsbasis (Abbruchbedingung) ist ein ausreichend einfacher Fall, daß das Ziel ohne rekursiven Aufruf bewiesen werden kann.

   Im Induktionsschritt wird die Induktionshypothese (rekursiver Aufruf) angewendet.

3.5.1 Das Wichtigste in Kürze

1. Ein rekursives Prädikat besitzt eine Regel, die einen Aufruf desselben Prädikats als Teilziel enthält.

2. Ein rekursives Prädikat ist immer induktiv aufgebaut:

   Die Induktionsbasis (Abbruchbedingung) ist ein ausreichend einfacher Fall, daß das Ziel ohne rekursiven Aufruf bewiesen werden kann.

   Im Induktionsschritt wird die Induktionshypothese (rekursiver Aufruf) angewendet.

3. Ein rekursives Prädikat kann einen iterativen Prozeß definieren, wenn alle rekursiven Aufrufe am Ende der jwlg. Regel stehen und zum Zeitpunkt des rekursiven Aufrufs keine choice points zurückbleiben (last call optimization).

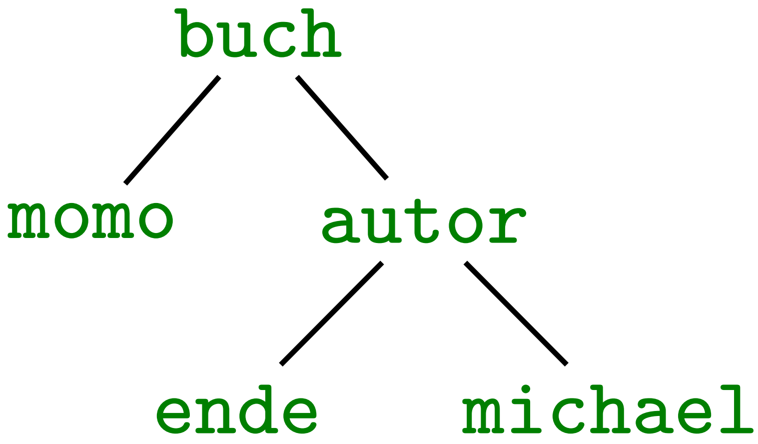
4. Ein nicht-iteratives Prädikat kann durch Einführung eines Akkumulators iterativ gemacht werden.

3. Ein rekursives Prädikat kann einen iterativen Prozeß definieren, wenn alle rekursiven Aufrufe am Ende der jwlg. Regel stehen und zum Zeitpunkt des rekursiven Aufrufs keine choice points zurückbleiben (last call optimization).

4. Ein nicht-iteratives Prädikat kann durch Einführung eines Akkumulators iterativ gemacht werden.

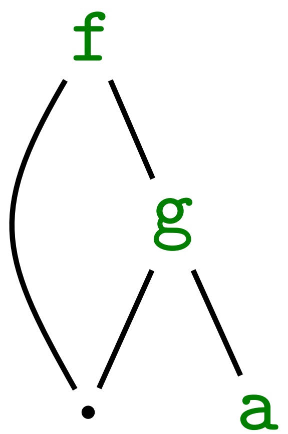
Jede Strukur kann als Baum aufgefaßt werden.

```
buch(momo,autor(ende,michael))
```



Identische Variablen in Strukturen führen zu identischen Knoten.

`f(X,g(X,a))`



3.6.1 Implizit definierte Bäume

```
fahrzeit(hbf,6min).
fahrzeit(bochum,12min).
fahrzeit(essen,20min).
fahrzeit(duesseldorf,40min).
fahrzeit(koeln,70min).
```

Man erhält eine Baumstruktur durch Verwenden **rekursiv** definierter Strukturen.

Gewünschte Baumdarstellung:

```
                    fahrzeit(essen,20min, • , • )

fahrzeit(duesseldorf,40min, • ,_)   fahrzeit(hbf,6min,_, • )

 fahrzeit(bochum,12min,_,_)      fahrzeit(koeln,70min,_,_)
```

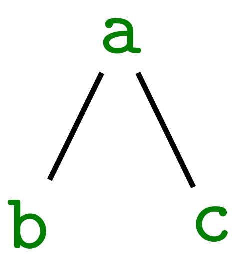
Als Term:

```
fahrzeit(essen,20min,
         fahrzeit(duesseldorf,40min,
                  fahrzeit(bochum,12min,_,_),
                  _
                 ),
         fahrzeit(hbf,6min,
                  _,
                  fahrzeit(koeln,70min,_,_)
                 )
        ).
```

3.6.2 Explizit definierte Bäume

```
bintree(void).
bintree(tree(Element,Left,Right)) :-
          bintree(Left), bintree(Right).
```

Der Baum



wird repräsentiert als

```
tree(a,tree(b,void,void),tree(c,void,void)).
```

3.6.3 Partielle Strukturen

Eine Struktur, die freie Variablen enthält, nennen wir partielle Struktur.

Die Variablen können jederzeit instantiiert werden
$\leadsto$ Flexibilität beim Aufbau der Struktur.

Uninstantiierte Variablen können dupliziert werden
$\leadsto$ 'merken' von Löchern.

Beispiel 3.6.1

```
T =
lochb(tree(a,tree(b,_,Loch),tree(c,_,_)),Loch),
Loch = tree(d,_,_).
```

3.6.3 Partielle Strukturen

Eine Struktur, die freie Variablen enthält, nennen wir partielle Struktur.

Die Variablen können jederzeit instantiiert werden
$\leadsto$ Flexibilität beim Aufbau der Struktur.

Uninstantiierte Variablen können dupliziert werden
$\leadsto$ ‘merken’ von Löchern.

Beispiel 3.6.1

```
T =
lochb(tree(a,tree(b,_,Loch),tree(c,_,_)),Loch),
Loch = tree(d,_,_).
```

3.6.3 Partielle Strukturen

Eine Struktur, die freie Variablen enthält, nennen wir partielle Struktur.

Die Variablen können jederzeit instantiiert werden
$\leadsto$ Flexibilität beim Aufbau der Struktur.

Uninstantiierte Variablen können dupliziert werden
$\leadsto$ 'merken' von Löchern.

Beispiel 3.6.1

```
T =
lochb(tree(a,tree(b,_,Loch),tree(c,_,_)),Loch),
Loch = tree(d,_,_).
```

3.6.3 Partielle Strukturen

Eine Struktur, die freie Variablen enthält, nennen wir partielle Struktur.

Die Variablen können jederzeit instantiiert werden
$\leadsto$ Flexibilität beim Aufbau der Struktur.

Uninstantiierte Variablen können dupliziert werden
$\leadsto$ 'merken' von Löchern.

Beispiel 3.6.1

```
T =
lochb(tree(a,tree(b,_,Loch),tree(c,_,_)),Loch),
Loch = tree(d,_,_).
```

3.6.4 Das Wichtigste in Kürze

1. Jede Struktur kann als Baum aufgefaßt werden.
2. Die Datenstruktur Baum kann implizit durch Verwenden einer rekursiven Datenstruktur implementiert werden.
3. Man kann die Baumstruktur explizit machen durch Verwendung einer Struktur

   ```
   tree(Knoten,LinkerTeilbaum,RechterTeilbaum)
   ```

4. Partielle Strukturen können als Strukturen mit Löchern aufgefaßt werden, wobei man die Löcher explizit repräsentieren kann.

3.6.4 Das Wichtigste in Kürze

1. Jede Struktur kann als Baum aufgefaßt werden.
2. Die Datenstruktur Baum kann implizit durch Verwenden einer rekursiven Datenstruktur implementiert werden.
3. Man kann die Baumstruktur explizit machen durch Verwendung einer Struktur

   ```
   tree(Knoten,LinkerTeilbaum,RechterTeilbaum)
   ```

4. Partielle Strukturen können als Strukturen mit Löchern aufgefaßt werden, wobei man die Löcher explizit repräsentieren kann.

3.6.4 Das Wichtigste in Kürze

1. Jede Struktur kann als Baum aufgefaßt werden.
2. Die Datenstruktur Baum kann implizit durch Verwenden einer rekursiven Datenstruktur implementiert werden.
3. Man kann die Baumstruktur explizit machen durch Verwendung einer Struktur

   `tree(Knoten,LinkerTeilbaum,RechterTeilbaum)`
4. Partielle Strukturen können als Strukturen mit Löchern aufgefaßt werden, wobei man die Löcher explizit repräsentieren kann.

3.6.4 Das Wichtigste in Kürze

1. Jede Struktur kann als Baum aufgefaßt werden.
2. Die Datenstruktur Baum kann implizit durch Verwenden einer rekursiven Datenstruktur implementiert werden.
3. Man kann die Baumstruktur explizit machen durch Verwendung einer Struktur

   `tree(Knoten,LinkerTeilbaum,RechterTeilbaum)`
4. Partielle Strukturen können als Strukturen mit Löchern aufgefaßt werden, wobei man die Löcher explizit repräsentieren kann.

Induktive Definition: eine Liste ist

IB: die leere Liste `[]` oder

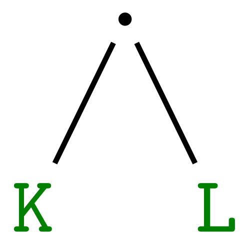
IS: eine Struktur `.(K,L)`, wobei `K` (Kopf der Liste) ein beliebiger Term und `L` (Rumpf der Liste) wieder eine Liste ist.

Induktive Definition: eine Liste ist

IB: die leere Liste [] oder

IS: eine Struktur .(K,L), wobei K (Kopf der Liste) ein beliebiger Term und L (Rumpf der Liste) wieder eine Liste ist.

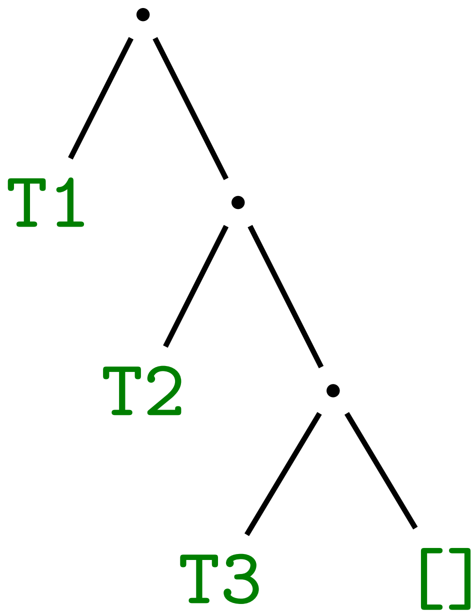
Baumnotation:



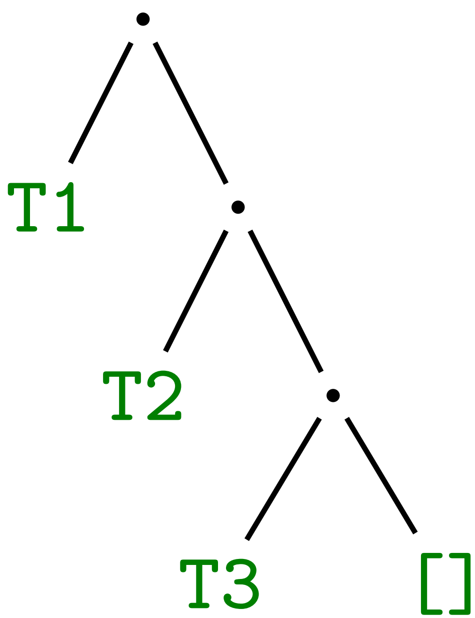
Alternative Notation:

.(K,L)	[K|L]
.(T1,.(T2,.(T3,[])))	[T1,T2,T3]

Baumdarstellung für [T1,T2,T3]:



Baumdarstellung für [T1,T2,T3]:



Alternativ:

Listen können verschachtelt werden:

```
[a,V1,b,[X,Y]]

.  ——  .  ——  .  ——  .  ——  []
|      |      |      |
a      V1     b      .  ——  .  ——  []
                     |      |
                     X      Y
```

Beispiel 3.7.1

```
p([1,2,3]).  p([1,2,3,[4,5,6]]).

?- p([X|Y]).

X = 1
Y = [2, 3] ;

X = 1
Y = [2, 3, [4, 5, 6]] ;

No
```

```
?- p([_,_,_,[_|X]]).

X = [5, 6] ;

No
```

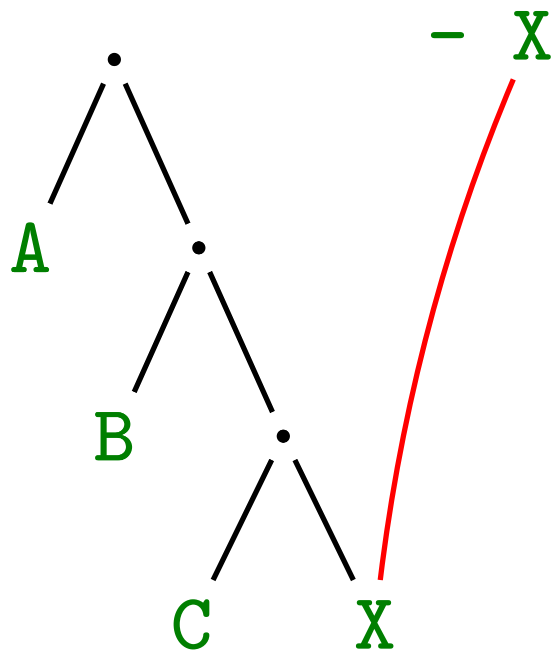
Listen mit ‘Löchern’: Differenzlisten

Eine Differenzliste ist eine partielle Liste, die ein ‘Loch’ am Ende hat, zusammen mit dem Loch.

Listen mit ‘Löchern’: Differenzlisten

Eine Differenzliste ist eine partielle Liste, die ein ‘Loch’ am Ende hat, zusammen mit dem Loch.

Beispiel 3.7.2 `[a,b,c|X]-X`, repräsentiert als Baum:



In einer Differenzliste `FL - D` ist `FL` ein 'Versprechen' für die vollständige Liste und `D` der Anteil, der zur vollen Liste noch fehlt.

In einer Differenzliste `FL - D` ist `FL` ein ‘Versprechen’ für die vollständige Liste und `D` der Anteil, der zur vollen Liste noch fehlt.

Der eigentliche Inhalt, der durch `D=[]` erhalten wird, ist also tatsächlich in gewissem Sinne die Differenz zwischen `FL` und `D`.

In einer Differenzliste `FL - D` ist `FL` ein 'Versprechen' für die vollständige Liste und `D` der Anteil, der zur vollen Liste noch fehlt.

Der eigentliche Inhalt, der durch `D=[]` erhalten wird, ist also tatsächlich in gewissem Sinne die Differenz zwischen `FL` und `D`.

Aneinanderhängen von Differenzlisten:

```
dl_append(A-M, M-R, A-R).
```

Das Wichtigste in Kürze

1. Eine Liste ist eine spezielle Struktur mit Funktor `..`
2. Schreibweise `[K|R]` (statt `.(K,R)`) für eine Liste mit Kopf `K` und Rumpf `R`.
3. Operationen auf Listen sind rekursiv: Behandle zuerst den Kopf, dann (rekursiver Aufruf) den Rumpf.
4. Akkumulatoren können bei der Listenverarbeitung die Effizienz verbessern.

Das Wichtigste in Kürze

1. Eine Liste ist eine spezielle Struktur mit Funktor `..`
2. Schreibweise `[K|R]` (statt `.(K,R)`) für eine Liste mit Kopf `K` und Rumpf `R`.
3. Operationen auf Listen sind rekursiv: Behandle zuerst den Kopf, dann (rekursiver Aufruf) den Rumpf.
4. Akkumulatoren können bei der Listenverarbeitung die Effizienz verbessern.

Das Wichtigste in Kürze

1. Eine Liste ist eine spezielle Struktur mit Funktor
   `..`
2. Schreibweise `[K|R]` (statt `.(K,R)`) für eine Liste mit Kopf `K` und Rumpf `R`.
3. Operationen auf Listen sind rekursiv: Behandle zuerst den Kopf, dann (rekursiver Aufruf) den Rumpf.
4. Akkumulatoren können bei der Listenverarbeitung die Effizienz verbessern.

Das Wichtigste in Kürze

1. Eine Liste ist eine spezielle Struktur mit Funktor `..`
2. Schreibweise `[K|R]` (statt `.(K,R)`) für eine Liste mit Kopf `K` und Rumpf `R`.
3. Operationen auf Listen sind rekursiv: Behandle zuerst den Kopf, dann (rekursiver Aufruf) den Rumpf.
4. Akkumulatoren können bei der Listenverarbeitung die Effizienz verbessern.

5. Eine Differenzliste ist eine partielle Struktur, die anstelle mit der leeren Liste mit einem Loch (uninstantiierte Variable) abgeschlossen wird, das explizit repräsentiert wird. Durch Instantiieren mit dem Loch kann an die Liste etwas hinten angehängt werden.

Bisher haben wir hauptsächlich die logische Semantik eines PROLOG-Programmes betrachtet.

Diese ist wichtig, um logische Zusammenhänge leicht korrekt zu modellieren.

Um PROLOG-Programme effizient zu implementieren, muß aber auch die prozedurale Semantik beachtet werden, die sich aus dem Ausführungsmodell ergibt.

Die folgenden Systemprädikate betreffen die prozedurale Semantik und erlauben, die Ausführung eines PROLOG-Programmes zu beeinflussen.

Bisher haben wir hauptsächlich die logische Semantik eines PROLOG-Programmes betrachtet.

Diese ist wichtig, um logische Zusammenhänge leicht korrekt zu modellieren.

Um PROLOG-Programme effizient zu implementieren, muß aber auch die prozedurale Semantik beachtet werden, die sich aus dem Ausführungsmodell ergibt.

Die folgenden Systemprädikate betreffen die prozedurale Semantik und erlauben, die Ausführung eines PROLOG-Programmes zu beeinflussen.

Bisher haben wir hauptsächlich die logische Semantik eines PROLOG-Programmes betrachtet.

Diese ist wichtig, um logische Zusammenhänge leicht korrekt zu modellieren.

Um PROLOG-Programme effizient zu implementieren, muß aber auch die prozedurale Semantik beachtet werden, die sich aus dem Ausführungsmodell ergibt.

Die folgenden Systemprädikate betreffen die prozedurale Semantik und erlauben, die Ausführung eines PROLOG-Programmes zu beeinflussen.

Bisher haben wir hauptsächlich die logische Semantik eines PROLOG-Programmes betrachtet.

Diese ist wichtig, um logische Zusammenhänge leicht korrekt zu modellieren.

Um PROLOG-Programme effizient zu implementieren, muß aber auch die prozedurale Semantik beachtet werden, die sich aus dem Ausführungsmodell ergibt.

Die folgenden Systemprädikate betreffen die prozedurale Semantik und erlauben, die Ausführung eines PROLOG-Programmes zu beeinflussen.

3.8.1 Der Cut

Das Ziel $\boxed{!}$ ist immer erfüllbar und bewirkt einen Nebeneffekt: Alle choice points, die für das aktuelle Ziel noch existieren,

- sowohl choice points zu Teilzielen
- als auch der choice point, der auf alternative Klauseln zum aktuellen Ziel zeigt,

werden gelöscht.

3.8.1 Der Cut

Das Ziel $\boxed{!}$ ist immer erfüllbar und bewirkt einen Nebeneffekt: Alle choice points, die für das aktuelle Ziel noch existieren,

- sowohl choice points zu Teilzielen
- als auch der choice point, der auf alternative Klauseln zum aktuellen Ziel zeigt,

werden gelöscht.

Der Cut bewirkt, daß das aktuelle Prädikat sich auf alle bisher getroffenen Entscheidungen festlegt.

Grüne Cuts

Ein grüner Cut entfernt choice points, die nicht zu neuen Lösungen führen.

Ein grüner Cut verändert nicht die logische Semantik eines Programms, sondern verhindert nur, daß beim Beweis ‘sinnlose’ Teilziele verfolgt werden.

Auf diese Weise kann man ein Prädikat deterministisch machen, das es eigentlich sein sollte, es aber nicht ist, weil dies vom Compiler nicht erkannt wird.

Grüne Cuts

Ein grüner Cut entfernt choice points, die nicht zu neuen Lösungen führen.

Ein grüner Cut verändert nicht die logische Semantik eines Programms, sondern verhindert nur, daß beim Beweis ‘sinnlose’ Teilziele verfolgt werden.

Auf diese Weise kann man ein Prädikat deterministisch machen, das es eigentlich sein sollte, es aber nicht ist, weil dies vom Compiler nicht erkannt wird.

Grüne Cuts

Ein grüner Cut entfernt choice points, die nicht zu neuen Lösungen führen.

Ein grüner Cut verändert nicht die logische Semantik eines Programms, sondern verhindert nur, daß beim Beweis 'sinnlose' Teilziele verfolgt werden.

Auf diese Weise kann man ein Prädikat deterministisch machen, das es eigentlich sein sollte, es aber nicht ist, weil dies vom Compiler nicht erkannt wird.

Beispiel 3.8.1

```
fact(0,F,F) :- !.

fact(N,A,F) :- N > 0,
    N1 is N-1, A1 is A*N, fact(N1,A1,F).
```

Rote Cuts

Schneiden Lösungen ab.

Können eingesetzt werden, um unerwünschte Lösungen zu verhindern.

3.8.2 Alternative. IF-THEN-ELSE.

Werden Ziele mit ; verknüpft (bindet schwächer als ,), so wird erst versucht, die linke Seite des ; zu beweisen. Schlägt dies fehl, so wird versucht, die rechte Seite des ; zu beweisen.

3.8.2 Alternative. IF-THEN-ELSE.

Werden Ziele mit ; verknüpft (bindet schwächer als ,), so wird erst versucht, die linke Seite des ; zu beweisen. Schlägt dies fehl, so wird versucht, die rechte Seite des ; zu beweisen.

Beispiel 3.8.2

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

Das Konstrukt IF <A> THEN <B> ELSE <C> kann man implementieren als

```
(
    <A>, !, <B>
;
    <C>
).
```

Das Konstrukt IF <A> THEN <B> ELSE <C> kann man implementieren als

```
(
    <A>, !, <B>
;
    <C>
).
```

Hierfür gibt es die spezielle Notation

`( <A> -> <B> ; <C> )`.

Beispiel 3.8.3

```
fact(N,A,F) :-
        (
            N =:= 0 -> A = F
        ;
            N1 is N-1, A1 is A*N, fact(N1,A1,F)
        ).
```

3.8.3 Fail. Negation.

Das Ziel `fail` schlägt immer fehl.

3.8.3 Fail. Negation.

Das Ziel `fail` schlägt immer fehl.

Die Kombination `!, fail` kann genutzt werden, um unerwünschte Alternativen auszuschließen.

3.8.3 Fail. Negation.

Das Ziel `fail` schlägt immer fehl.

Die Kombination `!, fail` kann genutzt werden, um unerwünschte Alternativen auszuschließen.

Beispiel 3.8.4

```
schweizer_garde_geeignet(X) :-
              groesse(X,H),
              H < 184,
              !, fail.
```

Beispiel 3.8.5

```
not(X) :-
        (
            call(X) -> fail
        ;
            true
        ).
```

Beispiel 3.8.5

```
not(X) :-
        (
            call(X) -> fail
        ;
            true
        ).
```

Für `not` gibt es die spezielle Notation `\+`.

Beispiel 3.8.5

```
not(X) :-
        (
            call(X) -> fail
        ;
            true
        ).
```

Für `not` gibt es die spezielle Notation $\boxed{\texttt{\textbackslash+}}$.

Beispiel 3.8.6

```
disjunkt(L1,L2) :- \+ ( member(X,L1), member(X,L2) ).
```

Beispiel 3.8.7 Wenn wir Prädikate mit Nebeneffekten einsetzen (z. B. I/O), können wir alle Lösungen eines Ziels folgendermaßen abarbeiten (failure-driven loop):

```
alle_Loesungen :-
        (
            ziel(X),
            verarbeite(X),
            fail
        ;
            true
        ).
```

3.9.1 Manipulation und Aufruf von Termen

`functor(?T,?F,?A).`	Matcht `T` mit einer Struktur, die den Funktor `F` und die Arität `A` hat.
`arg(+N,+T,?Arg).`	Matcht das `N`-te Argument von Struktur `T` mit `Arg`.

`?Term =.. ?Liste.`	Matcht `Term` mit der Struktur, deren Funktor der Kopf von `Liste` ist und deren Argumente die restlichen Listenelemente bilden.

`?Term =.. ?Liste.`	Matcht `Term` mit der Struktur, deren Funktor der Kopf von `Liste` ist und deren Argumente die restlichen Listenelemente bilden.

`call(+Goal).`	Ruft den Term `Goal` als Ziel auf.

Beispiel 3.9.1

```
mapfunc([], [], _).

mapfunc([K|R], [MK|MR], F) :-
        functor(Ziel,F,2),
        arg(1,Ziel,K),
        arg(2,Ziel,MK),
        call(Ziel),
        mapfunc(R,MR,F).
```

Alternativ:

```
mapfuncx([], [], _).

mapfunc([K|R], [MK|MR], L) :-
        append(L,[K,MK],ZL),
        Ziel =.. ZL,
        call(Ziel),
        mapfunc(R,MR,L).
```

Alternativ:

```
mapfuncx([], [], _).

mapfunc([K|R], [MK|MR], L) :-
        append(L,[K,MK],ZL),
        Ziel =.. ZL,
        call(Ziel),
        mapfunc(R,MR,L).
```

Achtung! `call` ist ‘undurchdringlich’ für den cut.

3.9.2 Compiliert vs. Interpretiert

`:- dynamic(pred/n).`	Erklärt das `n`-stellige Prädikat `pred` als dynamisch. Dieses wird interpretiert und darf manipuliert werden.

3.9.2 Compiliert vs. Interpretiert

`:- dynamic(pred/n).`	Erklärt das `n`-stellige Prädikat `pred` als dynamisch. Dieses wird interpretiert und darf manipuliert werden.

3.9.3 Inspektion der Datenbasis

`clause(+Clause).`	Matcht `Clause` mit einer dynamischen Klausel, falls dies möglich ist.

3.9.4 Alle Antworten

`findall(?T,+Z,?L).`	Matcht L mit einer Liste aller Instanzen von Term T, für die Ziel Z erfüllbar ist. Deterministisch !
`bagof(?T,+Z,?L).`	Instantiiert alle Variablen in Ziel Z, die nicht in T vorkommen, so dass Z erfüllbar ist, und matcht L mit einer Liste aller Instanzen von Term T, für die Z erfüllt ist. Evtl. nichtdeterministisch.
`setof(?T,+Z,?L).`	Wie bagof, aber ohne Duplikate.

3.9.5 Manipulation der Datenbasis

`assert(+Clause).`	Fügt die Klausel `Clause` der Datenbasis hinzu.
`asserta(+Clause).`	Fügt die Klausel `Clause` am Anfang der Datenbasis hinzu.
`assertz(+Clause).`	Fügt die Klausel `Clause` am Ende der Datenbasis hinzu.

`retract(+Clause).`	Entfernt die erste Klausel, die `Clause` matcht, aus der Datenbasis.
`retractall(+Clause).`	Entfernt alle Klauseln, die `Clause` matchen, aus der Datenbasis.

`retract(+Clause).`	Entfernt die erste Klausel, die `Clause` matcht, aus der Datenbasis.
`retractall(+Clause).`	Entfernt alle Klauseln, die `Clause` matchen, aus der Datenbasis.

Das Prädikat, das definiert oder gelöscht wird, muß als dynamisch deklariert sein.

`abolish(+Pred).`	Entfernt das Prädikat `Pred` vollständig aus der Datenbasis. `Pred` darf für `prädikatname` oder für `prädikatname/arität` stehen.

Das Prädikat kann statisch oder dynamisch sein.