051013 VO Theoretische Informatik

Prädikatenlogik III, Logische Programmierung: PROLOG

Ekaterina Fokina

universität wien

Zusammenfassung & Ausblick

Bis jetzt haben wir Folgendes behandelt:

- Formale Logik in der Informatik
- Aussagenlogik
  - Hornlogik
- Prädikatenlogik: Syntax, Semantik, Unentscheidbarkeit

Heute geht es mit:

- Resolution der Prädikatenlogik
- Logische Programmierung
  - Programmiersprache Prolog

Subsection 8

Unentscheidbarkeit der Prädikatenlogik

Unentscheidbarkeit der Prädikatenlogik

Aussagenlogik:

- Endlich viele mögliche Modelle (Belegungen)
- Erfüllbarkeit/Folgerung kann durch Testen aller Belegungen entschieden werden (Wahrheitstafel)

Prädikatenlogik:

- Unendlich viele passende Strukturen
- Unendlich große Strukturen
- Erfüllbarkeit kann nicht so einfach überprüft werden

Church's Theorem

Satz (Church's Theorem)

Es gibt kein Verfahren, das für jede prädikatenlogische Formel F in endlich vielen Schritten entscheidet, ob F erfüllbar ist.

Dazu mehr im 3.Teil der Vorlesung.

Herbrand Universum

Ziel: Anzahl der zu betrachtenden Strukturen einschränken.

Definition

Für eine Formel F in Skolemform ist das **Herbrand Universum** $D(F)$ induktiv wie folgt definiert.

- Alle in F enthaltenen Konstanten sind in $D(F)$
- Mindestens eine Konstante ist in $D(F)$
- Für k-stelliges Funktionssymbol f und $t_1, \ldots, t_k \in D(F)$ ist auch $f(t_1, \ldots, t_k) \in D(F)$

Das Herbrand-Universum ist die Menge aller variablenfreie Terme, die aus den Bestandteilen von F gebildet werden können.

Herbrand Struktur

Intuition:

1. Die Struktur verwendet das Herbrand Universum
2. Die variablenfreien Terme in der Formel werden durch das entsprechende Objekt des Herbrand Universums interpretiert.

Definition

Eine für eine Formel F in Skolemform passende Struktur $\alpha = (U, \varphi, \psi, \xi)$ heißt **Herbrand-Struktur** wenn

1. $U = D(F)$ (verwendet das Herbrand Universum)
2. Für k-stelliges Funktionssymbol f und $t_1, \ldots, t_k \in D(F)$ ist auch $f^{\varphi}(\alpha(t_1), \ldots, \alpha(t_k)) = f(t_1, \ldots, t_k)$

Falls $\alpha \models F$ nennt man α **Herbrand-Modell**.

Beispiel

Formel: $F = \forall z \forall y \left(Q(b) \vee P(b, g(z)) \vee (P(f(h(z)), y) \wedge Q(a))\right)$

Herbrand-Universum:

$D(F) = \{a, b, g(a), f(a), h(a), g(b), f(b), h(b), g(f(a)), \ldots\}$

Herbrand-Struktur:

1. $U = D(F)$
2. Funktionen/Konstanten:
   - $a^\varphi = a$, $b^\varphi = b$
   - $f^\varphi : f^\varphi(a) = f(a)$, $f^\varphi(f(b)) = f(f(b))$, ...
   - $g^\varphi : g^\varphi(a) = g(a)$, $g^\varphi(f(b)) = g(f(b))$, ...
   - $h^\varphi : h^\varphi(a) = h(a)$, $h^\varphi(f(b)) = h(f(b))$, ...
3. Prädikate
   - $Q^\psi = \{a, b\}$
   - $P^\psi = \{(a, b), (f(a), g(h(b)))\}$
4. $z^\xi = y^\xi = h(b)$

Teil 1 & 2 sind für Herbrand-Strukturen vorgegeben.

Satz von Löwenheim-Skolem

Satz (Löwenheim-Skolem)

Eine geschlossene prädikatenlogische Formel in Skolemform ist genau dann erfüllbar wenn sie ein Herbrand-Modell hat.

- Wir können uns also auf Herbrand-Modelle beschränken.
- Das Herbrand Universum ist aber im Allgemeinen unendlich groß.

Satz von Gödel-Herbrand-Skolem

Definition (Herbrand-Expansion)

Sei F eine prädikatenlogische Formel in Skolemform, d.h. F ist von der Form

$$F = \forall x_1 \forall x_2 \dots \forall x_n \, G.$$

Die **Herbrand Expansion** $E(F)$ von F ist die Menge der prädikatenlogischen Formeln

$$E(F) = \{ G_{x_1 \mapsto t_1, \dots, x_n \mapsto t_n} \mid t_1, \dots, t_n \in D(F) \}$$

wobei $G_{x_1 \mapsto t_1, \dots, x_n \mapsto t_n}$ die Formel notiert bei der in G die Variablen x_i durch Elemente t_i des Herbrand Universums ersetzt werden.

Satz (Gödel-Herbrand-Skolem)

Eine geschlossene prädikatenlogische Formel F in Skolemform ist genau dann erfüllbar wenn die Herbrand-Expansion $E(F)$ im aussagenlogischen Sinn erfüllbar ist.

Herbrand Expansion - Beispiel

Beispiel

Formel: $F = \forall x(P(f(x)) \wedge \neg G(x))$

- **Matrixformel:** $P(f(x)) \wedge \neg G(x)$
- **Herbrand Universum:** $D(F) = \{a, f(a), f(f(a)), f(f(f(a)), \ldots\}\}$
- **Herbrand Expansion:**
  $E(F) = \{P(f(a)) \wedge \neg G(a), P(f(f(a))) \wedge \neg G(f(a)), \ldots\}$

Achtung: $P(a) \wedge \neg G(a)$ ist nicht in der Herbrand Expansion von F.

Grundresolutions-Algorithmus

Basierend auf den Satz von Gödel-Herbrand-Skolem.

Grundresolutions-Algorithmus

Gegeben: Formel F in Matrixklauselform

- Initialisiere: $M = \{\}$, $i = 0$
- Iteriere bis M unerfüllbar
  - Erhöhe i um eins
  - Berechne das i-te Element H_i der Herbrand Expansion $E(F)$
  - Setze $M = \{H_1, \ldots H_i\}$
  - Betrachte M als aussagenlogische Formeln und teste auf Erfüllbarkeit
- Ist F unerfüllbar hält das Verfahren nach endlich vielen Schritten (Kompaktheitssatz)
- Ist F erfüllbar endet das Verfahren nie (Endlosschleife)

Grundresolutions-Algorithmus - Beispiel

$$F = \forall x (P(f(x)) \wedge \neg P(x))$$

- **Matrixformel:** $P(f(x)) \wedge \neg P(x)$
- **Matrixklauselform:** $K = \{\{P(f(x))\}, \{\neg P(x)\}\}$
- **D(F):** $\{a, f(a), f(f(a)), \dots\}$
- **E(F):** $\{P(f(a)) \wedge \neg P(a), P(f(f(a))) \wedge \neg P(f(a)), \dots\}$
- Wir können auch gleich mit der Klauselmenge arbeiten:
  E(K): $\{\{P(f(a))\}, \{\neg P(a)\}, \{P(f(f(a)))\}, \{\neg P(f(a))\}, \dots\}$

Grundresolutions-Algorithmus - Beispiel

1. Teste $M = \{\{P(f(a))\}, \{\neg P(a)\}\}$

   $\hookrightarrow$ erfüllbar

2. Teste $M = \{\{P(f(a))\}, \{\neg P(a)\}, \{P(f(f(a)))\}, \{\neg P(f(a))\}\}$

$$\begin{array}{ccc} \{P(f(a))\} & & \{\neg P(f(a))\} \\ & \searrow \quad \swarrow & \\ & \{\} & \end{array}$$

   $\hookrightarrow$ unerfüllbar

Im zweiten Schritt des Grundresolutions-Algorithmus ist M unerfüllbar, daher ist auch F unerfüllbar.

Semi-Entscheidbarkeit der Prädikatenlogik

Die Prädikatenlogik ist **semi-entscheidbar**:

- Ist F unerfüllbar hält das Verfahren nach endlich vielen Schritten. Aber wir haben keine Schranke für die Anzahl der Schritte.
- Wir können in endlich vielen Schritten zeigen das F eine Tautologie ist (wir testen ob $\neg F$ unerfüllbar ist).
- Es gibt kein Verfahren das
  - Testet ob F erfüllbar ist und
  - das für alle erfüllbaren F nach endlich vielen Schritten hält.

Subsection 9

Resolutionskalkül der Prädikatenlogik

Grundresolution:

- berechnet viele Atome und Resolventen die nicht benötigt werden.
- ist schlecht zu optimieren.

Beispiel

In unserem Beispiel:

- Vier Klauseln $\{\{P(f(a))\}, \{\neg P(a)\}, \{P(f(f(a)))\}, \{\neg P(f(a))\}\}$
- Wir benötigen nur zwei $\{\{P(f(a))\}, \{\neg P(f(a))\}\}$

Ziel:

$$\begin{tikzcd}[column sep=small]
P(f(x)) \arrow[d, dash, "x \mapsto a"'] & & \neg P(x) \arrow[d, dash, "x \mapsto f(a)"] \\
P(f(a)) \arrow[dr, dash] & & \neg P(f(a)) \arrow[dl, dash] \\
& \{\} &
\end{tikzcd}$$

oder

$$\begin{tikzcd}[column sep=small]
& & \{\neg P(x)\} \arrow[d, dash, "x \mapsto f(x)"] \\
\{P(f(x))\} \arrow[dr, dash] & & \{\neg P(f(x))\} \arrow[dl, dash] \\
& \{\} &
\end{tikzcd}$$

Unifikation

Idee:

- Für Resolution benötigen wir Literale die bis auf die Negation identisch sind
- Substituiere Variablen in Literalen sodass die Atome gleich werden.

Definition

Sei $\mathcal{L} = \{L_1, \dots, L_n\}$ eine Menge von Literalen. Eine Substitution $s = (x_1 \mapsto t_1, \dots, x_h \mapsto t_h)$ heißt **Unifikator** für $\mathcal{L}$ wenn $L_{1_{x_1 \mapsto t_1, \dots, x_h \mapsto t_h}} = \dots = L_{n_{x_1 \mapsto t_1, \dots, x_h \mapsto t_h}}$.

Beispiel

$\mathcal{L} = \{P(f(x)), P(y)\}$

Unifikator: $x \mapsto f(y), y \mapsto f(f(y))$

- $\{P(f(f(y))), P(f(f(y)))\}$

Allgemeinerer Unifikator: $y \mapsto f(x)$

- $\{P(f(x)), P(f(x))\}$

Unifikation

Definition

Ein Unifikator $s = (x_1 \mapsto t_1, \ldots, x_h \mapsto t_h)$ heißt ein **allgemeinster Unifikator** für $\mathcal{L}$ falls es für jeden weiteren Unifikator s' für $\mathcal{L}$ eine Substitution s'' gibt sodass $s' = s'' \circ s$.

Intuition: Wir suchen möglichst einfache Substitutionen

Beispiel

$\mathcal{L} = \{P(f(x)), P(y)\}$

Unifikator: $x \mapsto f(y), y \mapsto f(f(y))$

- $\{P(f(f(y))), P(f(f(y)))\}$

Allgemeinster Unifikator: $y \mapsto f(x)$

- $\{P(f(x)), P(f(x))\}$

Unifikations-Algorithmus

Gegeben: $\mathcal{L} = \{L_1, \ldots, L_n\}$ eine Menge von Literalen.

1. $\tilde{\mathcal{L}} := \mathcal{L}$, $s := \{\}$
2. Wiederhole bis $\tilde{\mathcal{L}}$ nur ein Literal enthält.
   1. Wähle zwei Literale L_1, L_2.
   2. Seien z_1 und z_2 die ersten Zeichen in denen sich L_1 und L_2 unterscheiden.
   3. Wenn weder z_1 noch z_2 eine Variable ist dann ist $\mathcal{L}$ nicht unifizierbar.
   4. Andernfalls sei z_1 die Variable und t der Term der mit z_2 anfängt
      - Wenn z_1 in t vorkommt dann ist $\mathcal{L}$ nicht unifizierbar.
      - Andernfalls füge die Substitution $z_1 \mapsto t$ zu s hinzu und substituiere z_1 in allen Literalen

- Entscheidet ob $\mathcal{L}$ unifizierbar ist.
- Berechnet einen allgemeinsten Unifikator.

Unifikations-Algorithmus - Beispiel

$\mathcal{L} = \{P(f(x), g(y, h(a, z))), P(f(g(a, b)), g(g(u, v), w))\}$

1. $L_1 = P(f(x), g(y, h(a, z)))$, $L_2 = P(f(g(a, b)), g(g(u, v), w))$
    - $z_1 = x$, $z_2 = g$, $t = g(a, b)$
    - $x \mapsto g(a, b)$
    - $\tilde{\mathcal{L}} = \{P(f(g(a, b)), g(y, h(a, z))), P(f(g(a, b)), g(g(u, v), w))\}$
2. $L_1 = P(f(g(a, b)), g(y, h(a, z)))$, $L_2 = P(f(g(a, b)), g(g(u, v), w))$
    - $y \mapsto g(u, v)$
    - $\tilde{\mathcal{L}} = \{P(f(g(a, b)), g(g(u, v), h(a, z))), P(f(g(a, b)), g(g(u, v), w))\}$
3. $L_1 = P(f(g(a, b)), g(g(u, v), h(a, z)))$,
   $L_2 = P(f(g(a, b)), g(g(u, v), w))$
    - $w \mapsto h(a, z)$
    - $\tilde{\mathcal{L}} = \{P(f(g(a, b)), g(g(u, v), h(a, z)))\}$
4. Da $\tilde{\mathcal{L}}$ nur ein Element hat stoppt das Verfahren

Der Unifikator ist $s = (x \mapsto g(a, b), y \mapsto g(u, v), w \mapsto h(a, z))$.

Prädikatenlogische Resolventen

Für Formeln in Matrixklauselform können wir annehmen das alle Klauseln verschiedene Variablen haben.

- Da alle Variablen allquantifiziert sind und die Klauseln mit Konjunktionen verbunden sind können wir sie umbenennen.

Definition

Seien K_1, K_2, K_3 prädikatenlogische Klauseln sodass K_1 und K_2 keine gemeinsamen Variablen haben. K_3 heißt **Resolvente** von K_1 und K_2 wenn

1. Es gibt Literale $L_1, \dots, L_k$ in K_1 und $L'_1, \dots, L'_k$ in K_2 sodass $\{\neg L_1, \dots, \neg L_k, L'_1, \dots, L'_k\}$ einen allgemeinsten Unifikator s hat.
2. $K_3 = s\big((K_1 \setminus \{L_1, \dots, L_k\}) \cup (K_2 \setminus \{L'_1, \dots, L'_k\})\big)$

Sei $K = K(F)$ die Matrixklauselform einer Formel:

- $Res(K) = K \cup \{R \mid R \text{ ist Resolvente zweier Klauseln in } K\}$

Resolutionssatz

Satz (Resolutionssatz)

Eine prädikatenlogische Formel mit Matrixklauselform K ist unerfüllbar genau dann wenn $\{\} \in Res^\infty(K)$.

Beispiel

$$F = \forall x(P(f(x)) \wedge \neg P(x))$$

- **Matrixklauselform:** $K = \{\{P(f(x))\}, \{\neg P(x)\}\}$
- **Umbenennen**:$K = \{\{P(f(x))\}, \{\neg P(y)\}\}$
- **Unifiziere** $\neg P(f(x))$ und $\neg P(y)$
  $\hookrightarrow$ Unifikator $(y \mapsto f(x))$
  $\hookrightarrow$ Klausel $\{\}$
- Da $\{\} \in Res^\infty(K)$ ist F unerfüllbar.

Prädikatenlogische Resolventen - Beispiel 1

Matrixklauselform $K = \{\{\neg Q(x)\}, \{\neg P(x, g(b))\}, \{Q(c), P(c, g(z)), Q(a)\}, \{Q(c), P(c, g(z)), P(f(h(z)), y)\}\}$

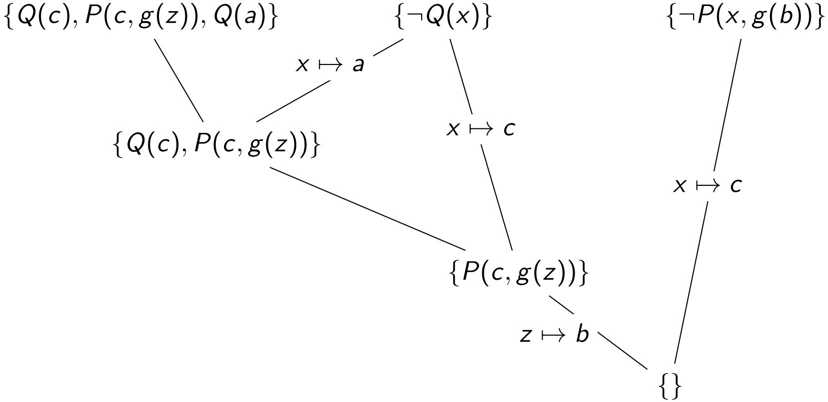
- $Res^0(K) = K$
- $Res^1(K) = Res^0(k) \cup \Big\{\{P(c, g(z)), Q(a)\}, \{Q(c), P(c, g(z))\}, \{P(c, g(z)), P(f(h(z)), y)\}, \{Q(c), Q(a)\}, \{Q(c), P(f(h(b)), y)\}, \{Q(c), P(c, g(z))\}\Big\}$
- $Res^2(K) = Res^1(k) \cup \Big\{\{P(c, g(z))\}, \{P(f(h(b)), y)\}, \{P(c, g(z))\}, \{Q(c)\}, \{Q(a)\}\Big\}$
- $Res^3(K) = Res^2(K) \cup \Big\{\{\}, \dots\Big\}$

Wir haben die leere Klausel $\{\}$ erreicht daher ist die Klauselmenge unerfüllbar.

Prädikatenlogische Resolventen - Beispiel 1

$\{\{\neg Q(x)\}, \{\neg P(x, g(b))\}, \{Q(c), P(c, g(z)), Q(a)\}, \{Q(c), P(c, g(z)), P(f(h(z)), y)\}\}$

Es geht auch wieder kompakter:



Prädikatenlogische Resolventen - Beispiel 2

Matrixklauselform: $K = \{\{Q(a), P(x)\}, \{R(y), \neg P(y)\}, \{S(z, f(z))\}\}$

- K ist erfüllbar (überlegen Sie sich ein Modell)
- Das Herbrand Universum: $D(K) = \{a, f(a), f(f(a)), \dots\}$ ist unendlich.
  $\hookrightarrow$ Grundresolution terminiert nicht (Endlosschleife)

Resolution:

- $Res^0(K) = \{\{Q(a), P(x)\}, \{R(y), \neg P(y)\}, \{S(z, f(z))\}\}$
- $Res^1(K) = Res^0(k) \cup \{\{Q(a), R(x)\}\}$
- $Res^2(K) = Res^1(K)$

Da $\{\} \notin Res^2(K) = Res^\infty(K)$ ist K erfüllbar.

$\hookrightarrow$ Resolution terminiert und zeigt Erfüllbarkeit.

In machen Fällen kann prädikatenlogische Resolution auch Erfüllbarkeit beweisen (auch wenn Grundresolution das nicht kann).

Logiken - Überblick

Es gibt in der Informatik viele verschiedene logische Systeme.

Einige haben wir in der Vorlesung kennengelernt:

- Aussagenlogik kann mehr ausdrücken als Hornlogik.
- Prädikatenlogik erste Stufe kann mehr ausdrücken als Aussagenlogik
- Auf den nächsten Folien: Prädikatenlogik zweiter Stufe kann mehr ausdrücken als Prädikatenlogik erster Stufe

Aber aus großer Ausdruckskraft folgt immer großer Berechnungsaufwand.

- Hornlogik kann sehr effizient berechnet werden.
- Aussagenlogik kann berechnet werden.
- Prädikatenlogik erste und zweiter Stufe kann im Allgemeinen nicht berechnet werden (mehr dazu in der Einheit Berechenbarkeit).

Subsection 11

Prädikatenlogik zweiter Stufe

Prädikatenlogik zweiter Stufe

Unsere Prädikatenlogik heißt auch Prädikatenlogik erster Stufe.
$\hookrightarrow$ Es gibt auch Prädikatenlogiken höher Stufen.

Eine Schwäche von Prädikatenlogik erster Stufe ist, dass Sie nicht über Gruppen/Mengen von Objekten quantifizieren kann.

Beispiel

Aussage: Es gibt eine Gruppe von Freunden, sodass jede Sprache von mindestens einem in der Gruppe gesprochen wird.

Ist die Gruppe durch ein Prädikat *Gruppe*(.) gegeben, können wir das wie folgt formulieren:

$$\forall x \left(Sprache(x) \to \exists y \left(Spricht(y, x) \land Gruppe(y) \right) \right) \land \\ \forall x \, \forall y \left(\left(Gruppe(x) \land Gruppe(y) \right) \to Freunde(x, y) \right)$$

Wir können aber die Existenzaussage "Es gibt eine Gruppe" nicht mit Prädikatenlogik erster Stufe formalisieren.

Prädikatenlogik zweiter Stufe

Prädikatenlogik zweiter Stufe:

- ermöglicht Quantifizierung über Prädikate und Funktionen erster Stufe.
- $\forall P^1 \, \exists x \, P(x)$: Für alle 1-stellige Prädikate P gibt es ein Objekt x sodass $P(x)$ wahr ist
- $\exists P^2 \, \forall x \, P(x, x)$: Es gibt ein 2-stelliges Prädikate P das reflexiv ist

Beispiel

Aussage: Es gibt eine Gruppe von Freunden sodass jede Sprache von mindestens einem in der Gruppe gesprochen wird.

Die Aussage kann jetzt formalisiert werden:

$$\exists Gruppe^1 \, \Big(\forall x \, \big(Sprache(x) \rightarrow \exists y \, (Spricht(y, x) \land Gruppe(y)) \big) \land \\ \forall x \, \forall y \, \big((Gruppe(x) \land Gruppe(y)) \rightarrow Freunde(x, y) \big) \Big)$$

Zusammenfassung & Ausblick

Bis jetzt haben wir Folgendes behandelt:

- Formale Logik in der Informatik
- Aussagenlogik
  - Hornlogik
- Prädikatenlogik

Weiter geht es mit:

- Logische Programmierung
  - Programmiersprache Prolog

Wiederholung: Hornlogik

Definition

Eine Klausel heißt **Hornklausel** wenn sie höchstens ein positives Literal enthält. Eine Formel F in KNF heißt **Hornformel** wenn $K(F)$ nur aus Hornklauseln besteht.

Beispiel für eine Hornformel:

- $(a \vee \neg b \vee \neg c) \wedge (\neg b \vee \neg c) \wedge c$

Unterschiedliche Arten von Hornklauseln:

- Tatsachenklauseln/Fakten: nur ein positives Literal, z.B. c
- Regeln: ein positives Literal, z.B. $(a \vee \neg b \vee \neg c)$ $(\equiv (b \wedge c) \rightarrow a)$
- Zielklausel: nur negative Literale, z.B. $(\neg b \vee \neg c)$

Hornformel kann man genauso für die Prädikatenlogik definieren.

Section 5

Logische Programmierung

Programmierparadigmen

Programmierparadigmen

- sind fundamentale Prinzipien nach denen Programmiersprachen aufgebaut sind.
- sollen Programmierer bei der Erstellung von „gutem" Code unterstützen.
- unterscheiden sich durch die Art
  - wie Sachverhalte modelliert und
  - „Funktionen" berechnet werden.

Eine konkrete Programmiersprache kann aber mehreren Paradigmen gleichzeitig folgen.

Programmierparadigmen

- Imperative Programmierung: Eine Folge von Befehlen gibt vor was in welcher Reihenfolge vom Computer getan werden soll.
- Objektorientierte Programmierung: Daten und darauf arbeitende Routinen werden zu Objekten zusammengefasst.
- Parallele Programmierung: Hat Methoden die es erlauben Programmteile nebenläufig auszuführen.
- Deklarative Programmierung: Es wird nicht angegeben wie etwas ausgerechnet werden soll, sondern es wird spezifiziert was ausgerechnet werden soll.
    - Logische Programmierung: nutzt logische Aussagen
    - Funktionale Programmierung: nutzt math. Funktionen
    - Mengen-Orientierte Abfragesprachen: z.B. SQL für Datenbanken
- ...

Vorteile Deklarativer Programmierung

- Die Spezifikation ist schon das Programm.
- Der Berechnungsmechanismus ist nicht Teil des Programms.
  $\hookrightarrow$ kann leicht ausgetauscht werden.
- Deklaratives „Denken“ ist oft einfacher als prozedurales „Denken“.
- Bei Logischer Programmierung ist der Output eine logische Konsequenz des Programms.
- Deklarative Programme sind (meist) sehr flexibel bezüglich der Fragestellung.

Als ein Beispiel für Deklarative Programmierung betrachten wir:

- Logische Programmierung – Sprache: Prolog

Logische Programmierung

Konventionelle Programmierung

- Prozedurale Denkweise als Grundlage
- Program = Algorithmus + Datenstruktur

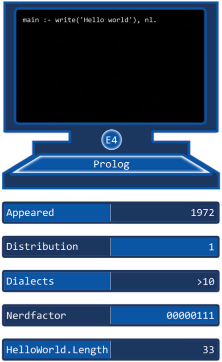
Logische Programmierung

- Algorithmus = Logik + Steuerung
- Logik definiert Wissen und Zielsetzung
- Steuerung definiert die Strategie dieses Wissen einzusetzen.
- Programm = Logik + Datenstruktur + Steuerung
- Der Programmierer sieht aber nur die Logik

Prolog

Programmiersprache **Prolog**:

- Populärste logische Programmiersprache.
- Der Name kommt vom Französischen „Programmation en Logique“.
- In den 1970ern maßgeblich von Alain Colmerauer entwickelt.
- Query-oriented: Berechnungen werden durch eine Frage in Form einer Formel gestartet.
- Prolog Implementierungen sind für die meisten Plattformen frei verfügbar z.B.: `http://www.swi-prolog.org/`.



Johann Weiher – `http://codequartett.de`

Teile eines Prolog-Programms

- Prolog-Programme bestehen aus Aussagen und nicht aus Anweisungen.
- Die Reihenfolge der Aussagen hat (meistens) keine Auswirkung auf die Semantik/Ausgabe des Programms.
- Atome werden aus Prädikaten gebildet.
- Aussagen werden als Horn Klauseln codiert:
  - Fakten
  - Regeln
  - Anfragen (Zielklausel)

Teile eines Prolog-Programms - Fakten

Die simpelsten Bestandteile von Prolog Programmen sind **Fakten**, auch „allgemeine Tatsachen“ genannt. Diese stellen elementare Tatsachen dar, oft wird damit die konkrete Probleminstanz/Eingabe codiert.

Hierzu verwendet man Prädikate und Objekte.

Beispiel

- Aussage: „Otto ist lieb“
  Prolog: `lieb(otto)`
- Aussage: „Otto ist Vater von Karl“
  Prolog: `istvater(otto, karl).`

Erinnerung: Bei Prädikaten ist die Reihenfolge der Argumente wichtig. $istvater(otto, karl)$ ist also nicht das gleiche wie $istvater(karl, otto)$.

Prolog Notation: Prädikate und Konstanten werden klein geschrieben.
Variablen beginnen mir einem Großbuchstaben.

Teile eines Prolog-Programms - Regeln

Regeln erlauben neue Fakten abzuleiten und haben die Form

```
a:-b1, ..., bn.
```

`a` ist der Kopf (head) und `b1,...,bn` der Rumpf (body) der Regel.[1]

```
Kopf :- Rumpf
```

Die Regel liest sich als

Wenn alle `bi` gelten dann gilt auch `a`.

und „entspricht“ damit der Implikation

$$a \leftarrow b1 \wedge \cdots \wedge bn$$

[1] `:-` ist das „neck“-Symbol.

Teile eines Prolog-Programms - Regeln

Beispiel

Regel: Wenn ein Mann Vater eines Kindes ist und dieses Kind Vater (oder Mutter) eines Kindes ist, dann ist er Großvater.

In Prolog:

```
grossvater(Mann):-istvater(Mann,Kind),istvater(Kind,Enkel).
```

`grossvater, istvater` sind Prädikate.
`Mann, Kind, Enkel` sind Variablen.

Das entspricht der logischen Formel:

$$\forall x, y, z(\mathit{Grossvater}(x) \leftarrow (\mathit{Istvater}(x, y) \wedge \mathit{Istvater}(y, z)))$$

Um das Prädiktat grossvater vollständig zu definieren braucht man noch:

```
grossvater(Mann):-istvater(Mann,Kind),istmutter(Kind,Enkel).
```

Teile eines Prolog-Programms - Rekursive Regeln

Wir wollen eine Relation definieren die alle Vorfahren kennt:

```
vorfahre(A,B) :- elternteil(A, B).
vorfahre(A,B) :- elternteil(A, X), elternteil(X, B).
vorfahre(A,B) :- elternteil(A, X), elternteil(X, Y),
                                   elternteil(Y, B).
vorfahre(A,B) :- elternteil(A, X), elternteil(X, Y),
                  elternteil(Y, Z), elternteil(Z,B).
...
```

Mit diesem Schema bräuchten wir (unendlich) viele Regeln.
Besser ist wir verwenden eine rekursive Regel.

```
vorfahre(A, B) :- elternteil(A, B).
vorfahre(A, B) :- elternteil(A, X), vorfahre(X, B).
```

Wichtig: Der rekursive Aufruf muss auf der rechten Seite stehen.
Da Prolog Regeln von Links nach Rechts abarbeitet.

Teile eines Prolog-Programms - Rekursive Regeln

Rekursive Regeln können lange Schleifen oder sogar Endlos-Schleifen erzeugen.

Faustregeln zum Umgang mit Rekursionen:

- Zuerst eine Regel definieren die dem Rekursionsanfang entspricht
- Im Rumpf der rekursiven Regel:
  - Die nicht-rekursiven Prädikate zuerst.
  - Die rekursiven Prädikate ganz rechts.

Teile eines Prolog-Programms - Anfragen / Queries

Berechnungen werden durch **Anfragen** (Queries) gestartet. Diese haben die Form

```
?-b1, ..., bn.
```

Die Anfrage liest sich als

Gilt b1, b2, . . . ,und bn.

Beispiel

Frage: Ist Fritz Großvater?
Prolog: `?-grossvater(fritz).`

Frage: Wer ist Großvater?
Prolog: `?-grossvater(X).`

SWI-Prolog

Freies Prolog System für Windows, Linux, und Mac OS:
`http://www.swi-prolog.org`

Aufruf mit `swipl` oder `prolog` (UNIX) / `swipl-win.exe` (Windows)
Startet in einem Modus in dem nur Anfragen gestellt werden können.

Programm laden: (für Programmdatei program.pl)

- Beim Start mit `prolog -f program.pl`
- Im laufenden Betrieb mit `consult(program.pl).` oder `[program.pl].`

Programm zur Laufzeit schreiben:

- Mit `consult(user).` oder `[user].` starten
- Programm eingeben
- mit Strg+D (Ctrl+D) beenden

Es gibt auch eine online Version: `https://swish.swi-prolog.org/`

Typisches Prolog-Programm

Ein typisches Prolog-Programm besteht aus

- Der **Wissensbasis**:
  - Aufstellen von Fakten
  - Aufstellung von Regeln
- Ergänzenden **Kommentaren** (`/* Kommentar */`)
- **Anfragen** zum starten von Berechnungen.

Eine Wissensbasis:

```
/* Fakten */
istvater(fritz, paul). istvater(paul, karl).
istvater(zeus, herkules). istmutter(alkmene,herkules).
/* Regeln */
grossvater(Mann):-istvater(Mann,Kind),istvater(Kind,Enkel).
```

Anfragen wären:

`?- grossvater(zeus).` **Antwort**: false.

`?- grossvater(X).` **Antwort**: X=fritz

`?- grossvater(X), istvater(X, paul).` **Antwort**: X=fritz

Anfragen Auswerten

```
?- p(X,Y), q(Z,a,X)..., b(X).
```

Eine Anfrage wird von links nach rechts ausgewertet:

- Wenn kein Prädikat übrig ist gib die Unifizierung true aus.
- Versuche das erste Prädikat zu unifizieren:
  - mit einem Fakt
  - mit einem der Köpfe der Regeln.
- Wähle eine der möglichen Unifizierungen aus
  - Wenn mit einem Fakt unifiziert wurde entferne das erste Prädikat
  - Wenn mit einem Regelkopf unifiziert wurde ersetze das erste Prädikat durch den Rumpf der Regel.
  - Wende die Unifikation auf alle Prädikate der Anfrage an.
  - Werte die neue Anfrage aus. Gibt diese false zurück betrachte die nächste Unifikation (backtracking)
- Wenn keine Unfizierung true ergibt gib false aus.

Anfragen Auswerten - Beispiel I

Beispiel

Anfrage: `?- grossvater(zeus).`

Wir haben nur eine Regel um grossvater abzuleiten:
`grossvater(Mann):-istvater(Mann,Kind),istvater(Kind,Enkel).`

Wir setzen `Mann=zeus`
`grossvater(zeus):-istvater(zeus,Kind),istvater(Kind,Enkel).`

Wir suchen einen Wert für `Kind` sodass `istvater(zeus,Kind)` wahr ist.
Die einzige Möglichkeit ist `Kind=herkules`
`grossvater(zeus):-istvater(zeus,herkules),istvater(herkules,Enkel).`

Nun gibt es keinen Wert für `Enkel` sodass `istvater(herkules,Enkel)` wahr ist.
$\hookrightarrow$ **Antwort**: false.

Anfragen Auswerten - Beispiel II

Beispiel

Anfrage: `?- grossvater(X).`

`grossvater(X):-istvater(X,Kind),istvater(Kind,Enkel).`

Wir haben 3 Möglichkeiten `istvater(X,Kind)` wahr zu machen.

1. (X,Kind)=(fritz, paul):
   `grossvater(fritz):-istvater(fritz,paul),istvater(paul,Enkel).`
   Die einzige Möglichkeit für `istvater(paul,Enkel)` ist `Enkel=karl`
   ↪ `X=fritz` ist Lösung
2. (X,Kind)=(paul, karl):
   `grossvater(paul):-istvater(paul,karl),istvater(karl,Enkel).`
   `istvater(karl,Enkel)` ist immer falsch
   ↪ `X=paul` ist keine Lösung
3. (X,Kind)=(zeus, herkules): analog zu (2) `X=zeus` ist keine Lösung

↪ **Antwort**: X=fritz; false

Anfragen Auswerten III

Beispiel

Anfrage: `?- grossvater(X), istvater(X, paul).`

Zuerst werten wir `grossvater(X)` aus und bekommen wieder `X=fritz`:
`?- grossvater(fritz), istvater(fritz, paul).`

Da `istvater(fritz, paul)` wahr ist erhalten wir

Antwort: X=fritz; false

Closed World Assumption

Prolog folgt bei Anfragen der **Closed World Assumption**:

- Wenn sich etwas nicht aus Fakten und Regeln ableiten lässt ist es falsch.

Sie kennen dieses Prinzip vielleicht von

- Datenbanken
- Zugfahrplänen
- Vorlesungsverzeichnissen

Eine Alternative wäre die Open World Assumption, bei der ein Atom nur falsch ist wenn sich das auch ableiten lässt.

Variablen bei Anfragen

Eine Wissensbasis:

```
/* Fakten */
frau(alkmene). frau(aphrodite). frau(harmonia). frau(hera).
mann(zeus). prim(zwei).
```

Wollen wir alle Frauen (oder eine beliebige Frau) ermitteln könnten wir alle Objekte einzeln testen: `?-frau(zwei).`, `?-frau(alkmene).`, ...

Es geht aber natürlich einfacher: `?-frau(X).` (`X` ist eine Variable)

Als Ausgabe erhalten wir:

```
X=alkmene;
X=aphrodite;
X=harmonia;
X=hera;
false.
```

In SWI Prolog wird zunächst nur die erste Antwort ausgegeben. Weitere Antworten können mit ";" abgerufen werden.

Konjunktion bei Anfragen

Eine **Wissensbasis**:

```
mag(utta, otto). mag(utta,milch).
mag(otto,essen). mag(otto,utta). mag(otto,milch).
```

In Anfragen können wir Konjunktion nutzen.

- **Frage:** Mag Otto Utta und mag Utta essen ?
  Prolog: `?- mag(otto,utta), mag(utta,essen).` **Antwort**: false.
- **Frage:** Mag Otto Utta und mag Utta Milch?
  Prolog: `?- mag(otto,utta), mag(utta,milch).` **Antwort**: true.

Das hätten wir aber auch leicht mit zwei getrennten Anfragen herausfinden können. Ein interessanteres Beispiel:

- **Frage:** Gibt es etwas das sowohl Otto als auch Utta mögen
  Prolog: `?- mag(otto,Etwas), mag(utta,Etwas).`
  Antwort: Etwas = milch.

Prolog - Beispiel

Eine Wissensbasis:

```
/* Fakten */
maennlich(uranus). maennlich(kronos).
weiblich(gaia). weiblich(rhea).
/* eltern(X,Y,Z) - X,Y sind Eltern von Z */
eltern(uranus,gaia,kronos). eltern(uranus,gaia,rhea).
```

Wir wollen eine Relation `istbruder` definieren:

```
/* Regel */
istbruder(Bruder,X):-maennlich(Bruder),eltern(Y,Z,Bruder),
                                  eltern(Y,Z,X),Bruder\==X.
```

Anfragen:

`?- istbruder(kronos,rhea).` **Antwort**: true.

`?- istbruder(kronos,Person).` **Antwort**: Person=rhea

`?- istbruder(kronos,X),istbruder(X,kronos).` **Antwort**: false.

Prolog - Beispiel (Anfrage 1 auswerten)

Beispiel

Anfrage: `?- istbruder(kronos,rhea).`

Wir haben nur eine Regel um `istbruder` abzuleiten:

```
istbruder(Bruder,X):-maennlich(Bruder),eltern(Y,Z,Bruder),
eltern(Y,Z,X),Bruder\==X.
```

Wir setzen `(Bruder,X)=(kronos,rhea)`

```
istbruder(kronos,rhea):-maennlich(kronos),eltern(Y,Z,kronos),
eltern(Y,Z,rhea),kronos\==rhea.
```

`maennlich(kronos)` ist wahr. Wir müssen `eltern(Y,Z,kronos)` wahr machen. Daher setzen wir `(Y,Z)=(uranus,gaia)`:

```
istbruder(kronos,rhea):-maennlich(kronos),
eltern(uranus,gaia,kronos), eltern(uranus,gaia,rhea),
kronos\==rhea.
```

Jetzt sind auch `eltern(uranus,gaia,rhea)` und `kronos\==rhea` wahr.
$\hookrightarrow$ **Antwort**: true.

Prolog - Beispiel (Anfrage 2 auswerten)

Beispiel

Anfrage: `?- istbruder(kronos,Person).`

Wir haben nur eine Regel um `istbruder` abzuleiten:

```
istbruder(Bruder,X):-maennlich(Bruder),eltern(Y,Z,Bruder),
eltern(Y,Z,X),Bruder\==X.
```

Wir setzen `Bruder=kronos`

```
istbruder(kronos,X):-maennlich(kronos),eltern(Y,Z,kronos),
eltern(Y,Z,X),kronos\==X.
```

`maennlich(kronos)` ist wahr. Wir müssen `eltern(Y,Z,kronos)` wahr machen. Daher setzen wir `(Y,Z)=(uranus,gaia)`:

```
istbruder(kronos,rhea):-maennlich(kronos),
eltern(uranus,gaia,kronos), eltern(uranus,gaia,X),kronos\==X.
```

Es gibt zwei Möglichkeiten `eltern(uranus,gaia,X)` wahr zu machen `X=kronos` und `X=rhea`. Im ersten Fall ist `kronos\==kronos` falsch im zweiten `kronos\==rhea` wahr.

↪ **Antwort**: Person=rhea; false.

Prolog - Beispiel (Anfrage 3 auswerten)

Beispiel

Anfrage: `?- istbruder(kronos,X),istbruder(X,kronos).`

Wir betrachten zuerst `istbruder(kronos,X)` und bekommen wie bei Anfrage 2 `X=rhea; false.` Wir setzen also `X=rhea`.

```
?- istbruder(kronos,rhea),istbruder(rhea,kronos).
```

Jetzt betrachten wir `istbruder(rhea,kronos)`.

```
istbruder(rhea,kronos):-maennlich(rhea),eltern(Y,Z,rhea),
eltern(Y,Z,kronos),rhea\==X.
```

`maennlich(rhea)` ist falsch und damit auch `istbruder(rhea,kronos)`

$\hookrightarrow$ **Antwort**: false.

Mehr Beispiele

Beispiel (Verwendung von Regeln)

Nehmen wir an, wir wollen folgende Tatsache angeben:

„Otto mag Essen.“

Wir können schreiben:
„Otto mag Brot.“ und “Brot is essbar.“
„Otto mag Wurst.“ und “Wurst is essbar.“
„Otto mag Käse.“ und “Käse is essbar.“

Besser: Eine allgemeine Regel der Form:

„Wenn ein Objekt o essbar ist, dann mag Otto o.“

In Prolog schaut das dann so aus:

```
essbar(brot). essbar(wurst). essbar(kaese).
mag(otto, X) :- essbar(X).
```

Mehr Beispiele

Beispiel (Verwendung von Regeln)

```
essbar(brot). essbar(wurst). essbar(kaese).
mag(otto, X) :- essbar(X).
```

- `?- mag(otto,brot).` **Antwort**: true.
- `?- mag(otto,X).` **Antwort**: X=brot; X=wurst; X=kaese; false.
- `?- mag(otto,brot),mag(otto,kaese).` **Antwort**: true.
- `?- mag(otto,brot),mag(otto,otto).` **Antwort**: false.

Mehr Beispiele

Beispiel (Verkehrsmittel)

Wir betrachten die verschiedenen Verkehrsmittel in einer Stadt:

- Bus, Bahn, Fahrrad, Auto, ...

Wir wollen öffentliche Verkehrsmittel von privaten unterscheiden.

```
/* Fakten */
oeffentlich_vm(bus). oeffentlich_vm(bahn).
privat_vm(fahrrad). privat_vm(auto).
```

Fahrrad fahren und öffentliche Verkehrsmittel sind in der Stadt billig.

```
billig(fahrrad).
/* Regeln */
billig(X):-oeffentlich_vm(X).
```

Mehr Beispiele

Beispiel (Verkehrsmittel)

Jetzt können wir verschiedene Fragen stellen:

- Ist Autofahren billig?
  `?- billig(auto).` **Antwort**: false.
- Welche Verkehrsmittel sind billig?
  `?- billig(X).` **Antwort**: X=bus; X=bahn; X=fahrrad; false.
- Welche privaten Verkehrsmittel sind billig?
  `?- privat_vm(X), billig(X).` **Antwort**: X=fahrrad; false.

Vergleichsoperatoren

Prolog kennt verschiedene Vergleichsoperatoren

Identische Ausdrücke (==)

`A == B` ist true wenn die beiden Ausdrücke ident sind. Verändert den Ausdruck nicht (es werden keine Variablen gebunden).

Beispiele:

- `?- mag(X,brot)==mag(X,brot).` **Antwort**: true.
- `?- mag(X,brot)==mag(Y,brot).` **Antwort**: false.
- `?- mag(X,brot)==mag(otto,brot).` **Antwort**: false.
- `?- mag(X,Y)==mag(Y,X).` **Antwort**: false.

Nicht Identische Ausdrücke (\==)

`A \== B` ist true wenn `A == B` false ist. Verändert den Ausdruck nicht (es werden keine Variablen gebunden).

Beispiele:

- `?- mag(X,brot)\==mag(Y,brot).` **Antwort**: true.
- `?- mag(X,brot)\==mag(X,brot).` **Antwort**: false.
- `?- mag(X,brot)\==mag(otto,brot).` **Antwort**: true.
- `?- mag(X,Y)\==mag(Y,X).` **Antwort**: true.

Vergleichsoperatoren

Unifizierbare Ausdrücke (=)

`A = B` ist true wenn die beiden Ausdrücke unifizierbar sind. Es wird eine Unifizierung ausgewählt.

Beispiele:

- `?- mag(X,brot)=mag(X,brot).` **Antwort**: true.
- `?- mag(X,brot)=mag(Y,brot).` **Antwort**: X=Y.
- `?- mag(X,brot)=mag(otto,brot).` **Antwort**: X=otto.
- `?- mag(X,Y)=mag(Y,X).` **Antwort**: X=Y.
- `?- mag(otto,Y)=mag(utta,X).` **Antwort**: false.

Nicht Unifizierbare Ausdrücke (\=)

`A \= B` ist true wenn die beiden Ausdrücke nicht unifizierbar sind. Verändert den Ausdruck nicht (es werden keine Variablen gebunden).

Beispiele:

- `?- mag(X,brot)\=mag(X,brot).` **Antwort**: false.
- `?- mag(X,brot)\=mag(Y,brot).` **Antwort**: false.
- `?- mag(X,brot)\=mag(otto,brot).` **Antwort**: false.
- `?- mag(X,Y)\=mag(Y,X).` **Antwort**: false.
- `?- mag(otto,Y)\=mag(utta,X).` **Antwort**: true.

Not Operator

Der not Operator drückt aus das etwas nicht bewiesen werden kann.

not

`not(A)` ist true wenn A false ist. Verändert den Ausdruck nicht (es werden keine Variablen gebunden).

Beispiele für `hund(forest).`

- `?- not(hund(forest)).` **Antwort**: false.
- `?- hund(findus).` **Antwort**: false.
- `?- not(hund(findus)).` **Antwort**: true.
- `?- not((hund(forest),hund(findus))).` **Antwort**: true.
- `?- not((hund(forest),not(hund(findus)))).` **Antwort**: false.

Disjunktion

Disjunktion

`A;B` ist true wenn mindestens einer beiden Ausdrücke `A`, `B` true ist.

Beispiel

```
elternteil(A):-vater(A,X).
elternteil(A):-mutter(A,X).
```

kann auch als

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

formuliert werden.

Vorsicht: Konjunktion bindet stärker als Disjunktion.
$\hookrightarrow$ Bei längeren Ausdrücken Klammern verwenden.

Existenzquantoren in Prolog

Aussage: Ein Vater ist ein Mann der ein Kind hat.

Prädikatenlogik:

$$\forall x((Mann(x) \land \exists y\, KindVon(y, x)) \rightarrow Vater(x))$$

Prolog:

```
vater(X) :- mann(X), kindVon(Y,X).
```

Die Variable Y wird für den Existenzquantor verwendet.

Wenn eine Variable in nur einem Prädikat vorkommt brauchen wir Sie nicht bennen sondern können stattdessen "_" schreiben

```
vater(X) :- Mann(X), KindVon(_,X).
```

Prolog - Generate & Test

Beispiel

Drei Buben Fritz, Hans und Karl rudern mit ihren Booten.

- Die Boote haben die Farben rot, grün und blau.
- Der Bub im roten Boot ist der Bruder von Fritz.
- Hans sitzt nicht im grünen Boot.
- Der Bub im grünen Boot hat Streit mit Fritz.

Wer sitzt in welchem Boot?

Lösungsparadigma Generate & Test:

- Die Lösungen werden in ein Prädikat $\texttt{loesung}(\vec{X})$ gespeichert.
- In einem ersten Teil einer Regel $\texttt{generate}(\vec{X})$ generieren wir alle Lösungskandidaten
- In einem zweiten Teil der Regel $\texttt{test}(\vec{X})$ testen wir alle Bedingungen die eine Lösung erfüllen muss.

$\texttt{loesung}(\vec{X})\texttt{:- generate}(\vec{X})\texttt{, test}(\vec{X})\texttt{.}$

Prolog - Generate & Test

Drei Buben Fritz, Hans und Karl rudern mit ihren Booten.

- Die Boote haben die Farben rot, grün und blau.
- Der Bub im roten Boot ist der Bruder von Fritz.
- Hans sitzt nicht im grünen Boot.
- Der Bub im grünen Boot hat Streit mit Fritz.

Wer sitzt in welchem Boot?

1) Wir nutzen ein Prädikat `loesung(B_rot,B_gruen,B_blau)`, das sich als `B_rot` sitzt im roten Boot, etc. liest.

2) Dann generieren wir Lösungskandidaten:

Prolog:

```
 bub(fritz). bub(hans). bub(karl).

loesung(B_rot,B_gruen,B_blau):-
bub(B_rot), bub(B_gruen), bub(B_blau),
```

Prolog - Generate & Test

Drei Buben Fritz, Hans und Karl rudern mit ihren Booten.

- Die Boote haben die Farben rot, grün und blau.
- Der Bub im roten Boot ist der Bruder von Fritz.
- Hans sitzt nicht im grünen Boot.
- Der Bub im grünen Boot hat Streit mit Fritz.

Wer sitzt in welchem Boot?

3) Wir testen Lösungskandidaten:

Prolog:

```
bub(fritz). bub(hans). bub(karl).

loesung(B_rot,B_gruen,B_blau):-
    bub(B_rot), bub(B_gruen), bub(B_blau),
        B_rot\==B_gruen, B_rot\==B_blau, B_gruen\==B_blau,
        B_rot\==fritz, hans\==B_gruen, B_gruen\==fritz.
```

Die Lösung bekommen wir mit `?-loesung(B_rot,B_gruen,B_blau).`

Was wir nicht betrachten haben

Wir haben die Grundkonzpte von Prolog betrachtet.

Einige wichtige Konzpete haben wir aber nicht betrachtet:

- Listen
- Arithmetik
- Input/Ouput
- ...

Ergänzende Materialien (Prolog)

- SWI-Prolog Manual.
  http://www.swi-prolog.org/
- Wikibooks
  Prolog.
  https://en.wikibooks.org/wiki/Prolog
- Logic Programming with Prolog
  Max Bramer
  Springer, 2013, ISBN: 978-1-4471-5486-0
- Learn Prolog Now!
  Patrick Blackburn, Johan Bos, and Kristina Striegnitz
  http://www.learnprolognow.org/

(Andere) Ansätze für logische Programmiersprachen

Prolog folgt dem Ansatz basierend auf **Theorembeweisern**

1. Generiere Problem Repräsentation.
2. Die Lösung wird durch eine Ableitung einer Formel / Query gegeben.

Ansatz der **Model-Generierung**

1. Generiere Problem Repräsentation.
2. Die Lösungen werden durch die Modelle der Repräsentation gegeben.

Beispiele für Model-generierende Sprachen

- SATisfiability Testing: Generiert Modelle für Aussagenlogische Formeln.
- Answer-Set Programming: Generiert „stable models“ für logische Programme.

SAT-Testing

- Eine gängige Methode, um schwierige kombinatorische Probleme [2] effizient zu lösen.
- Man schreibt ein Programm, das jede Probleminstanz in eine aussagenlogische Formel umwandelt, sodass
- durch Testen der Erfüllbarkeit der Formel das ursprüngliche Problem gelöst wird.
- Verwendet hochentwickelte Systeme um Modelle zu aussagenlogischen Formeln zu berechnen.

[2] Wir werden diese Problem später NP-schwer nennen.

SAT-Testing: Beispiel

Problemstellung

Wir betrachten eine Landkarte mit mehreren Länderen und wollen die Ländern mit drei Farben (z.B.: Rot, Grün, Blau) sodass jedes Land genau eine Farbe hat und benachbarte Länder verschiedene Farben haben.

Ein Programm kann jetzt wie folgt eine passende Formelmenge $\mathcal{F}$ erzeugen:

- Für jedes Land ℓ führen wir die Variablen ein:
  - $R_\ell \dots$ Land ℓ ist rot gefärbt.
  - $G_\ell \dots$ Land ℓ ist grün gefärbt.
  - $B_\ell \dots$ Land ℓ ist blau gefärbt.
- Für jedes Land ℓ fügen wir die folgenden Formeln zu $\mathcal{F}$ hinzu:
  - $R_\ell \vee G_\ell \vee B_\ell$
  - $\neg(R_\ell \wedge G_\ell) \wedge \neg(R_\ell \wedge B_\ell) \wedge \neg(B_\ell \wedge G_\ell)$
- Für Nachbarländer ℓ, f fügen wir die folgende Formel zu $\mathcal{F}$ hinzu:
  - $\neg(R_\ell \wedge R_f) \wedge \neg(G_\ell \wedge G_f) \wedge \neg(B_\ell \wedge B_f)$

SAT-Testing: Beispiel

Wenn wir wissen wollen ob wir die Karte mit 3 Farben einfärben können

- testen wir $\mathcal{F}$ auf Erfüllbarkeit.

Wenn wir wissen wollen wie wir die Karte mit 3 Farben einfärben können

- berechnen wir ein Modell für $\mathcal{F}$.
- Färben ein Land ℓ Rot / Grün / Blau wenn R_ℓ / G_ℓ / B_ℓ in dem Modell wahr ist.

Zusammenfassung

Programmiersprachen folgen verschiedenen Programmierparadigmen.

- Klassische Programmiersprachen: Programmiererin gibt an wie etwas berechnet werden soll
  - Imperative Programmierung
  - Objektorientierte Programmierung
  - Parallele Programmierung
  - ...
- Deklarative Programmierung: Programmiererin definiert was berechnet werden soll
  - Beispiel: Logische Programmierung (Prolog)