Technische Universität Berlin

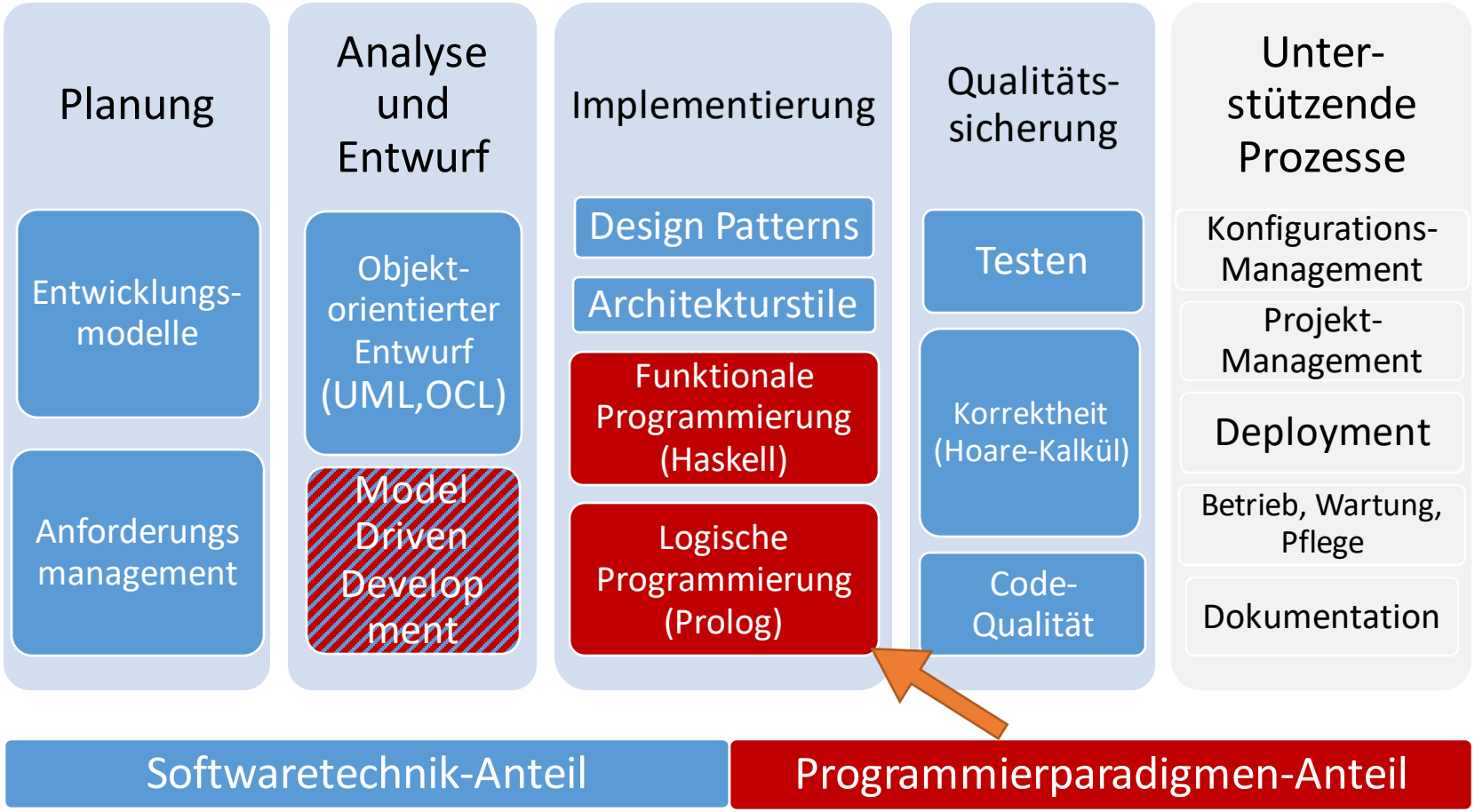
Softwaretechnik und Programmierparadigmen

02 Logische Programmierung

Prof. Dr. Sabine Glesner
Software and Embedded Systems Engineering
Technische Universität Berlin

SESE

Diese VL


Planung
Entwicklungs-modelle
Anforderungs management
Analyse und Entwurf
Objekt-orientierter Entwurf (UML,OCL)
Model Driven Develop ment
Implementierung
Design Patterns
Architekturstile
Funktionale Programmierung (Haskell)
Logische Programmierung (Prolog)
Qualitäts-sicherung
Testen
Korrektheit (Hoare-Kalkül)
Code-Qualität
Unter-stützende Prozesse
Konfigurations-Management
Projekt-Management
Deployment
Betrieb, Wartung, Pflege
Dokumentation
Softwaretechnik-Anteil
Programmierparadigmen-Anteil

Inhalt

Logische Programmierung

- Einführung in Prolog
- Sprach-Syntax
- Unifikation und Resolution
- Rekursion
- Vordefinierte Prädikate

Inhalt

Logische Programmierung

- Einführung in Prolog
- Sprach-Syntax
- Unifikation und Resolution
- Rekursion
- Vordefinierte Prädikate

Logische Programmierung

Weiterer Vertreter der **deklarativen Programmierung**

- **Beschreibung des Problems** steht im Vordergrund (Was)
- Trennt die **Implementierung** (Wie) vom **Algorithmus**
- Lösungsmethode ist vom Interpreter vorgegeben

Beruht auf der mathematischen Logik

- **Axiome** (Fakten und Regeln) beschreiben eine Situation (**Was gilt**)
- Axiome enthalten damit auch die **gesamte Datenbasis**
- **Anfragen** an das System werden vom Benutzer als **Ziel** vorgegeben
- Der **Interpreter** versucht automatisch weitere Regeln **herzuleiten**, um die Frage zu beantworten

Prolog - Einführung

Lösungsmethode in Prolog:

- **Tiefensuche** mit **Unifikation** und **Resolution**
- Ziel: **Variablenbelegungen**, mit denen die Anfrage wahr wird
- **positive Antwort**: Anfrage kann aus Datenbasis abgeleitet werden
  - Eine **gültige Belegung** wird zurückgegeben
- **negative Antwort**: Anfrage kann nicht aus Datenbasis abgeleitet werden
  - Keine gültige Belegung kann gefunden werden
  - Die Antwort ist „no“ bzw. **„false“**

Anwendungsgebiete**:**

- Künstliche Intelligenz (Spracherkennung)
- Datenbanksuche
- Expertensysteme
- Constraint Programming

Prolog - Geschichte

1965: John Alan Robinson entwickelt den Resolution Calculus

1972: Erster Prolog-Interpreter

- von Alan Colmerauer (Frankreich) entwickelt
- Bedeutung des Namens: Programmation en Logique

1983: Warren Abstract Machine

- Erster Prolog-Compiler, von David H.D. Warren entwickelt (Edinburgh)

1995: ISO-Prolog

- Definition eines ISO-Standards basierend auf dem Edinburgh-Dialekt

Es gibt verschiedene Implementierungen. Wir verwenden SWI-Prolog (http://www.swi-prolog.org/).

Inhalt

Logische Programmierung

- Einführung in Prolog
- Sprach-Syntax
- Unifikation und Resolution
- Rekursion
- Vordefinierte Prädikate

Begriffe: Überblick

Ein Programm in Prolog besteht aus einer Menge von Prädikate (Predicates)

Prädikate sind allgemeiner als Funktionen (z.B. in Haskell):

- Sie können durch mehrere Belegungen erfüllt werden (mehrere Ergebnisse)

Prädikate werden durch eine Menge von Klauseln (Clauses) definiert

Eine Klausel besteht aus Literalen (Literals), die durch logische Operatoren verbunden sind

Literale haben als Argumente Terme (Terms), die die Datenstrukturen in Prolog darstellen

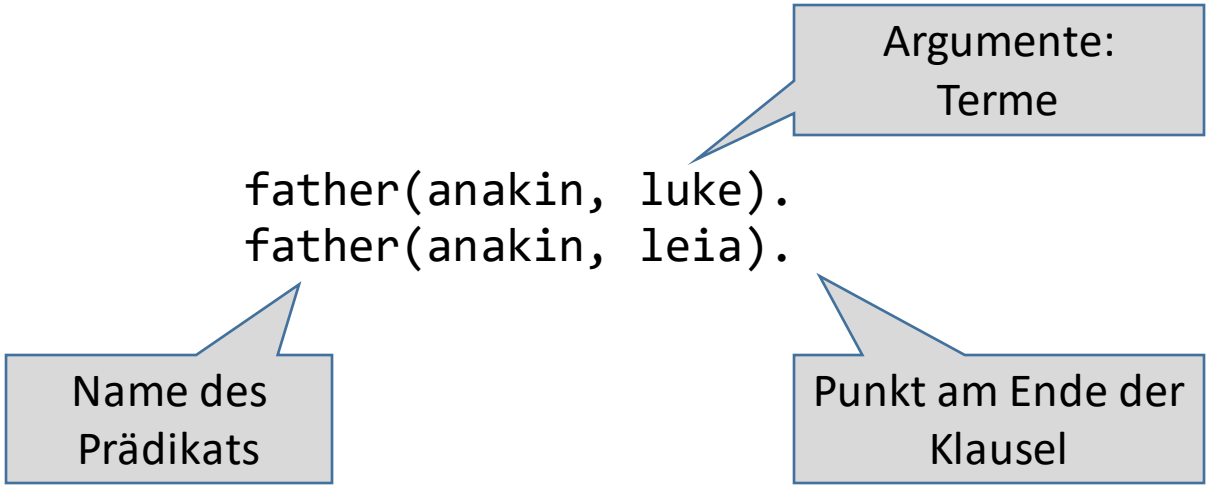
Eine Klausel kann entweder ein Fakt (Fact), eine Regel (Rule) oder eine Anfrage (Query, Goal) sein

Closed-world assumption: Datenbasis und sonst nichts gilt

Fakt

Fakt: Einfachste Form der Klausel

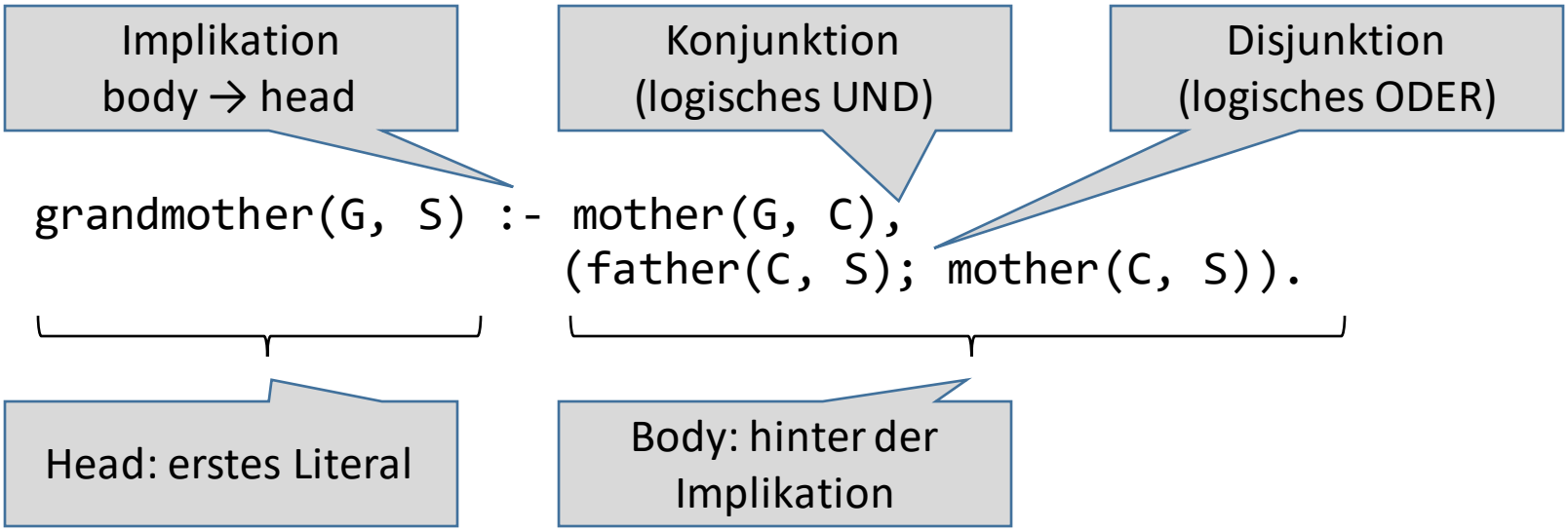
- Besteht aus nur einem Literal
- Aussage, die **wahr** ist (Relation zwischen Argumenten)



Regel

Klausel mit der Form **head** :- **body.**

- body besteht aus Literalen, verbunden durch **logische Operatoren**
- Eine Regel ist eine Implikation aus einer wahren Aussage



Ein Fakt einspricht einer Regel mit **body = true**

```
father(anakin, luke) :- true.
```

Terme

Einzige **Datenstruktur** in Prolog (Prolog besitzt keine statische Typisierung)

- Dienen als **Argumente** von Literalen
- Können folgende Form annehmen:

  Zahlen: 5, 3.5, -12413412.234

  Individuenkonstanten (engl. *atom*): luke, ‘apple‘, australia, -->

  - Starten mit Kleinbuchstaben ODER sind von einfachen Anführungszeichen umschlossen ODER bestehen nur aus Spezialzeichen

  Strings: ”You must unlearn what you have learned.”

  Listen: [1,2,3], [alice, bob, eve]

  - Menge von Termen

  Variablen: X, Y, Person, _Student, _lecture

  - Starten mit Großbuchstaben oder Unterstrich

  Funktoren (Function terms/compound terms):

  lecture(swtpp, „DO12-14“, „EB301“)

Basisterm (*ground term*): Hat keine Variablen

Variablen

Variable Werte

- Können beliebig ersetzt werden
- **Gültigkeitsbereich** (Scope): Die Klausel in der sie definiert wurde
- Jedes Vorkommen einer Variable innerhalb einer Klausel wird mit demselben Wert belegt.
- Variablen, die in einer Regel nur einmal auftauchen (**frei**), nennen sich **Singletons** – sie können nicht mit Werten belegt werden
  - Meist ein Fehler
  - SWI-Prolog gibt eine Warnung aus
- Variablen, die mehrfach auftauchen, also *verwendet* werden, nennt man **gebundene** Variablen

Anonyme Variable: '_' (Unterstrich)

- Für irrelevante Werte

Anfragen

Query/Goal

- Fakten und Regeln, die an den Interpreter zur Lösung gegeben werden
- Die Antwort ist true, wenn eine Belegung gefunden werden kann, sonst false.
- Wenn freie Variablen in der Query verwendet wurden, wird für sie eine Belegung angegeben

SWI-Prolog: Mit [SPACE] kann man den Interpreter nach weiteren Belegungen suchen lassen und diese ausgeben.

Beispiel-Queries

```
father(anakin, luke).
father(anakin, leia).
mother(shmi, anakin).

grandmother(G, S) :- mother(G, C),
                     (father(C, S); mother(C, S)).
```

Anfrage: Ist Anakin der Vater von Luke?

Als Parameter übergeben wir konkrete Werte (Atome).

```
?- father(anakin,luke).
true.
```

Antwort: **true**. Es kann eine Belegung gefunden werden, die Zutrifft.

Beispiel-Queries

```
father(anakin, luke).
father(anakin, leia).
mother(shmi, anakin).

grandmother(G, S) :- mother(G, C),
                     (father(C, S); mother(C,S)).
```

Anfrage: Hat Anakin Kinder?

Diesmal verwenden wir eine anonyme Variable. Es interessiert uns nicht welche Kinder es gibt.

```
?- father(anakin,_).
true.
```

Antwort: **true**. Es kann eine Belegung gefunden werden, in der Anakin der Vater von irgendwem ist.

Beispiel-Queries

```
father(anakin, luke).
father(anakin, leia).
mother(shmi, anakin).

grandmother(G, S) :- mother(G, C),
                     (father(C, S); mother(C,S)).
```

Anfrage: Von wem ist Shmi die Großmutter?

X ist eine freie Variable. Nun werden Belegungen von X gesucht, mit denen die Bedingungen wahr werden.

```
?- grandmother(shmi,X).
X = luke ;
X = leia ;
false.
```

Die Antwort ist zweimal ein Datensatz für den die Aussage zutrifft (**true**). Danach **false**, da es keinen dritten gibt.

Beispiel-Queries

```
father(anakin, luke).
father(anakin, leia).
mother(shmi, anakin).

grandmother(G, S) :- mother(G, C),
                     (father(C, S); mother(C,S)).
```

Anfrage: Ist denn Leia eine Mutter?

```
?- mother(leia,_).
false.
```

Antwort: Nicht in unserem Datensatz (Closed world assumption). Es wird schon beim ersten Aufruf **false** zurückgegeben.

Beispiel-Queries

```
father(anakin, luke).
father(anakin, leia).
mother(shmi, anakin).

grandmother(G, S) :- mother(G, C),
                     (father(C, S); mother(C,S)).
```

Haben Luke und Leia denselben Vater (und wenn ja, wen)?

Alle Vorkommen von X sind gleich.

```
?- father(X,luke),father(X,leia).
X = anakin.
```

Verundet: Beide Literale müssen in der Zielbelegung erfüllt sein.

Funktoren

Funktoren sind Terme, die wiederum aus anderen Termen bestehen

- Beliebig tief geschachtelt möglich

Literal mit Individuenkonstanten

```
persondata(peter, 41, 'Spooner Street', '31', 'Quahog').
```

Funktor zur expliziten Benennung eines Arguments

```
persondata(peter, age(41),
           address('Spooner Street', '31', 'Quahog')).
```

Funktor zur Zusammenfassung von zusammengehörigen Termen (Daten)

Funktoren (2)

Achtung: Funktoren sind keine Funktionen! Sie werden nicht ausgewertet. Beispiel:

```
printPlus1(X) :- write(X + 1).

?- printPlus1(10).
10+1
true.
```

Genau. Es wird 10+1 ausgegeben obwohl da ein '+' steht. Ausgewertet werden nur Ausdrücke mit is (mehr dazu später).

Achtung: Namen von Funktoren können keine Variablen sein

- D.h. Variablen haben keine Argumente

```
?- Y= X (a,b).
```

Fehler!

Arithmetik

Boolesche Operatoren and (,), or (;), implies (:-), not (\+)

Vergleiche

- Zahlen: Numerisch gleich (=:=) , ungleich (=\=), <, >, =<, >=
- Terme:
  - Unifizierbar (=), nicht unifizierbar (\=): Test ob ein Term in den anderen umgewandelt werden kann
  - identisch (==), nicht identisch (\==): Test ob zwei Terme gleich sind.
- Bei Vergleichen müssen alle Variable gebunden sein

```
?- wert(a)=wert(X).
X = a.
true.

?- wert(a)==wert(X).
false.
```

Auf der Rechten Seite des Vergleichs kann X durch a ersetzt werden, dann sind beide gleich. → unifizierbar

Beim Vergleich mit == hingegen sind wirklich gleiche Terme gefragt

Arithmetik (2)

Grundrechenarten +, -, *, /, mod

Zuweisung eines Wertes (numerisch) zu einer Variable mit is

- Bsp: X1 is X + 1
- Variable auf der linken Seite wird der Ausdruck der rechten Seite zugewiesen
- Auf der rechten Seite müssen alle Variablen gebunden sein

➢ Hier wird tatsächlich etwas ausgewertet

➢ Ohne die Verwendung von is sind die Operatoren normale Terme in Infix-Notation!

```
?- X = (5*5-3) mod 2.
X = (5*5-3)mod 2.
```

```
?- X is (5*5-3) mod 2.
X = 0.
```

Listen

Vordefinierte **rekursive Struktur**

Beispiel: [1,2,3,4,5]

Leere Liste: [ ] (in manchen Prolog-Versionen Leerzeichen nötig)

Mit ‘|‘ können **Head** (Element(e)) und **Tail** (Liste) getrennt werden:

[H|T]

[first,second|Rest]

[1,2|[3,4,5]]

Achtung: es gibt auch hierbei keine Typisierung, d.h. folgende Definition einer Liste ist gültig:

[morgen, 34, temperatur(98,“F“),warm]

Auch ist es möglich, eine Liste zu übergeben wenn keine erwartet wird

➢Übergabe „passender“ Parameter ist Verantwortung des Entwicklers

Listenoperationen

Einige vordefinierte Listenoperationen (viele weitere sind verfügbar).

```
is_list(L). /* erfüllt wenn L eine Liste ist */

length(List,Length).  /* Length erhält die Länge von List */

member(E, List). /* erfüllt wenn ein Element ==E  in List
                    enthalten ist */

delete(List1, E, List2). /* List2 entspricht List1 ohne das
                            Element E */

append(L1,L2,L1_2). /* L1_2 entspricht L1 konkateniert mit
                       L2 */
```

Inhalt

Logische Programmierung

- Einführung in Prolog
- Sprach-Syntax
- Unifikation und Resolution
- Rekursion
- Vordefinierte Prädikate

Was steckt hinter Prolog?

Fakten und Regeln in Prolog-Programmen sind Hornklauseln.

- Hornklauseln sind eine echte Teilmenge der Prädikatenlogik

Das Ableiten von neuen Fakten aus der Datenbasis wird durch Resolution ermöglicht

➢ einem Beweisverfahren aus der Logik

- Die Laufzeitumgebung versucht dabei, die Ausdrücke durch Unifikation so weit zu ersetzen, dass sich eine gültige Belegung ergibt

➢ Es wird nicht vorgegeben was getan werden soll, sondern was *gilt*

Unifikation

Unifikation zweier Terme: **Ersetzung** (Substitution) der Variablen in den Termen derart, dass die so entstandenen Zeichenfolgen **gleich** sind

- Zwei Terme sind unifizierbar, wenn:
  - Sie beide dieselbe Individuenkonstante sind
  - Einer von ihnen eine freie Variable ist
  - Beide komplexe Terme sind mit folgenden Eigenschaften:
    - der Funktor ist derselbe
    - die Anzahl Parameter (Stelligkeit) ist gleich
    - deren Argumente sind paarweise unifizierbar

Unifikation (2)

Ein paar unifizierbare Beispiele…

```
?- gleich = gleich.
true.
```

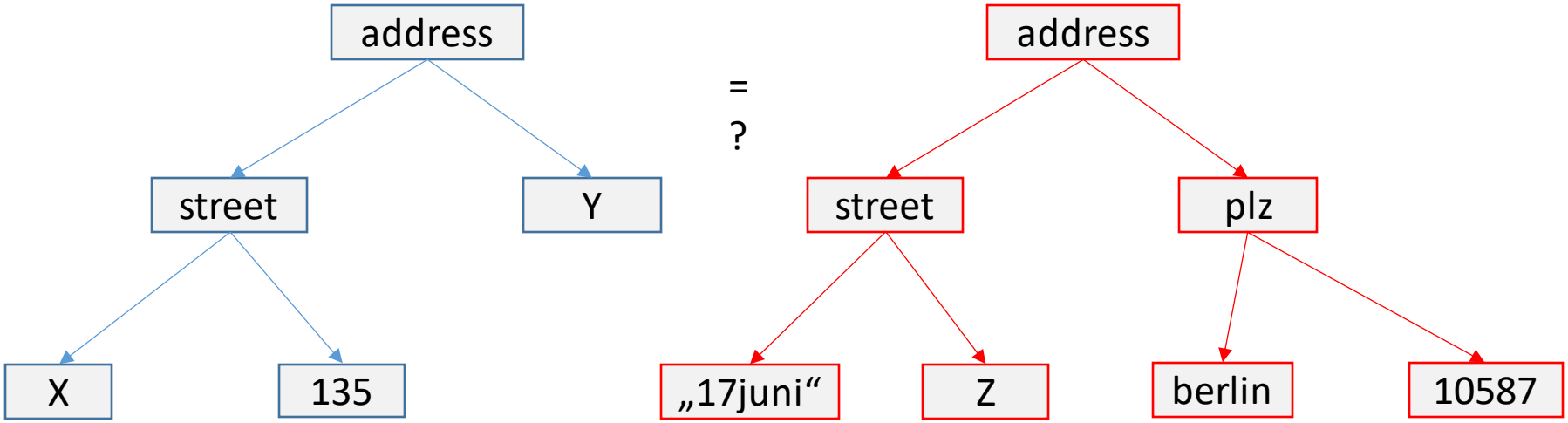
```
?- X = gleich.
true.
```

```
?- father(anakin,luke) = father(anakin,X).
X = luke.
```

```
?- address(street(X,135),Y) = address(street("17juni",Z),plz(berlin,10587)).
X = "17juni",
Y = plz(berlin, 10587),
Z = 135.
```

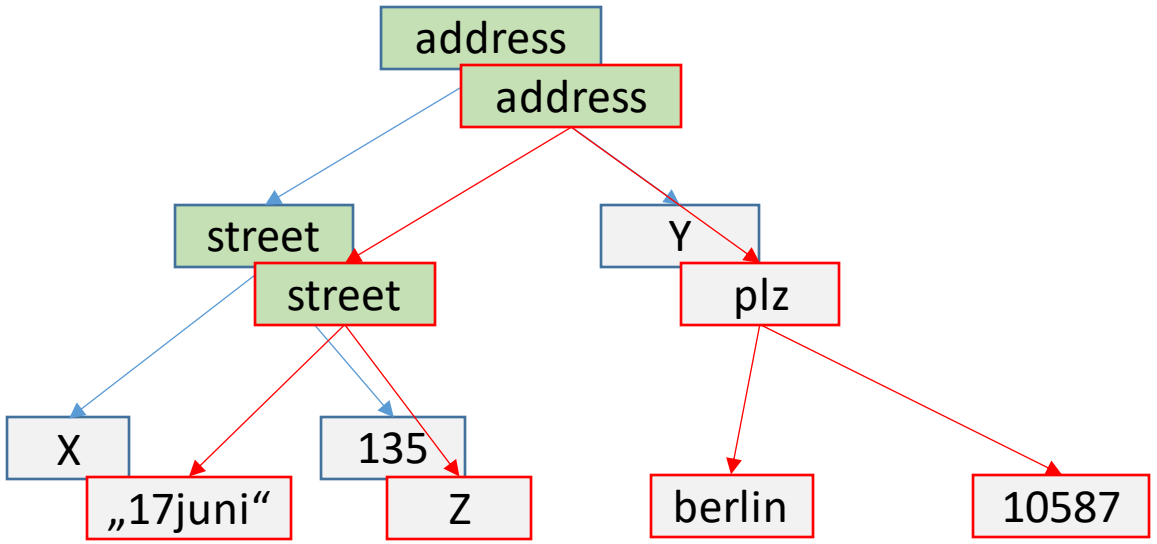
Unifikation (3)

address(street(X,135),Y) = address(street("17juni",Z),plz(berlin,10587)).



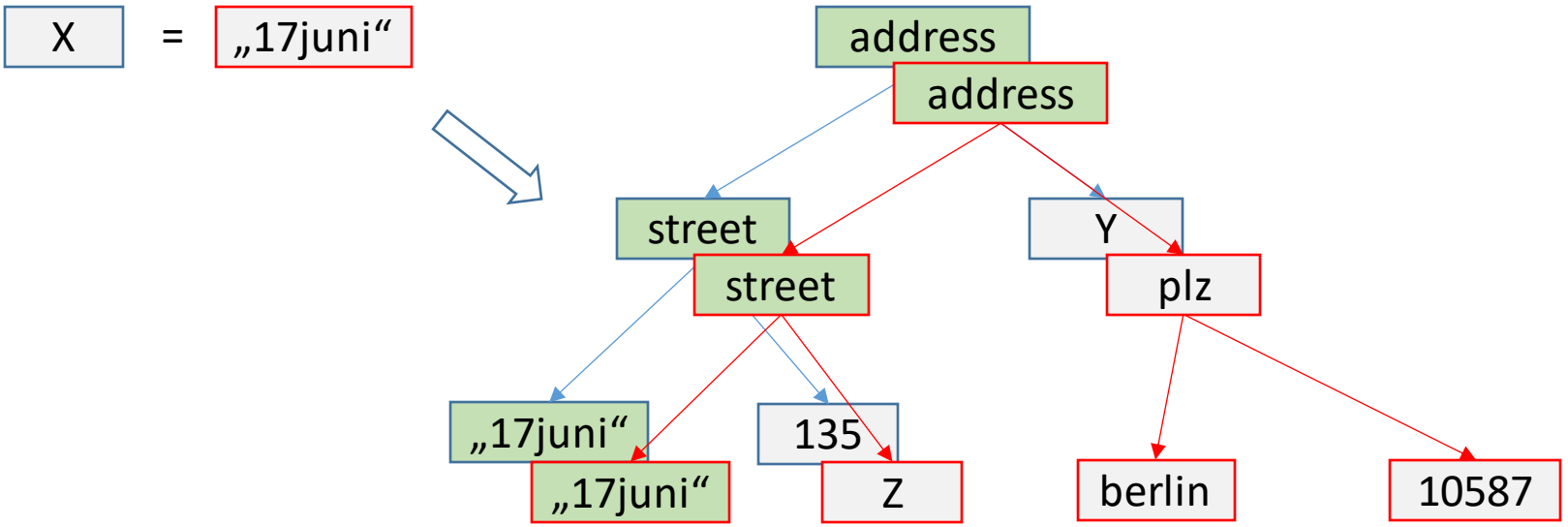
Unifikation (3)

address(street(X,135),Y) = address(street("17juni",Z),plz(berlin,10587)).



Unifikation (3)

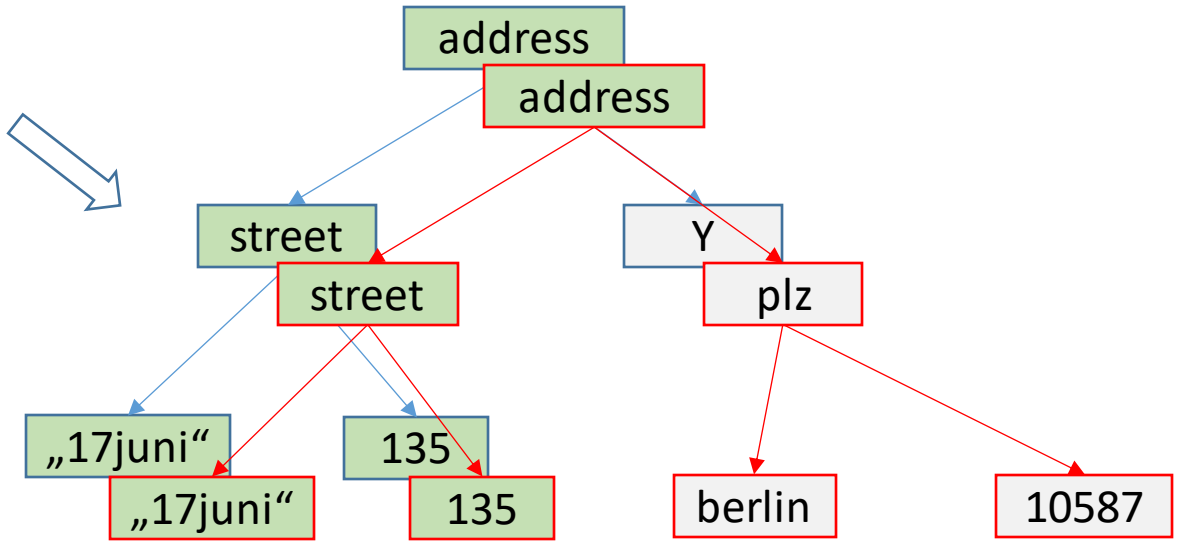
address(street(X,135),Y) = address(street("17juni",Z),plz(berlin,10587)).



Unifikation (3)

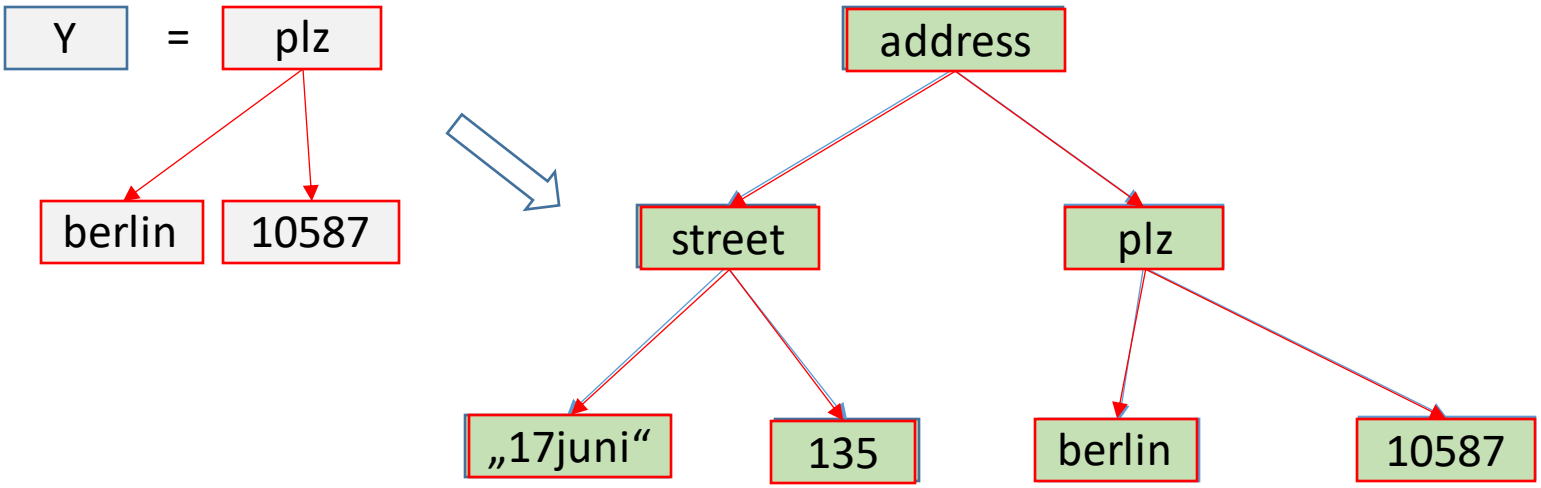
address(street(X,135),Y) = address(street("17juni",Z),plz(berlin,10587)).

135 = Z



Unifikation (3)

address(street(X,135),Y) = address(street("17juni",Z),plz(berlin,10587)).



Unifikation (4)

… und diese sind nicht unifizierbar

```
?- aehnlich = ähnlich.
false.
```

Nicht ganz.

```
?- father(X,luke) = father(anakin,X).
false.
```

Ersetzung von X muss überall gleich sein

```
?- father(X,luke) = man(anakin).
false.
```

Funktor nicht gleich und Parameterzahl unterschiedlich.

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

Zyklischer Term erzeugt unendlichen Baum. Je nach Prolog-Implementierung und Einstellung wird das erkannt oder nicht. Hier Default-Verhalten von SWI-Prolog.

Resolution

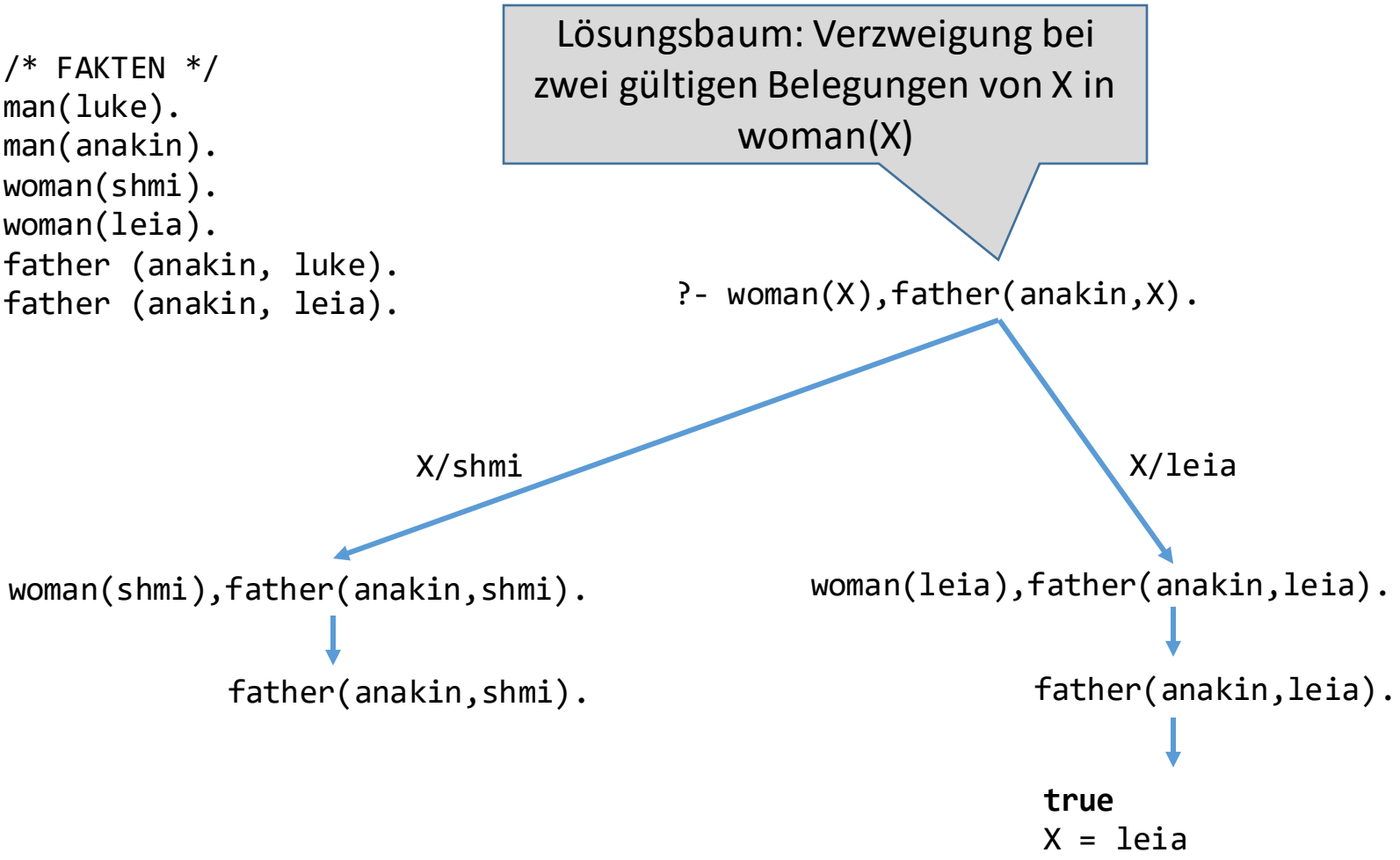
Der Prolog-Interpreter versucht, Anfragen schrittweise durch Unifikation so weit zu substituieren, dass die leere Behauptung übrig bleibt (true)

- Alternative Entscheidungen (mehrere gültige Eingaben) erzeugen einen **Lösungsbaum**
- Baum wird nach Lösung durchsucht (SWI-Prolog: **Tiefensuche**)
- Bei Misserfolg (oder um weitere Lösung zu finden): **Backtracking** zur letzten Entscheidung
- Antwort „false“ möglich durch Closed-World-Assumption: „**Negation by Failure**“

SWI-Prolog (Bearbeitung der Anfrage):
- Auswahl des ersten Literals (links), dann schrittweise Auflösung nach rechts
- Bei Entscheidungen: Zuerst wird die erste (linke/obere) Möglichkeit versucht

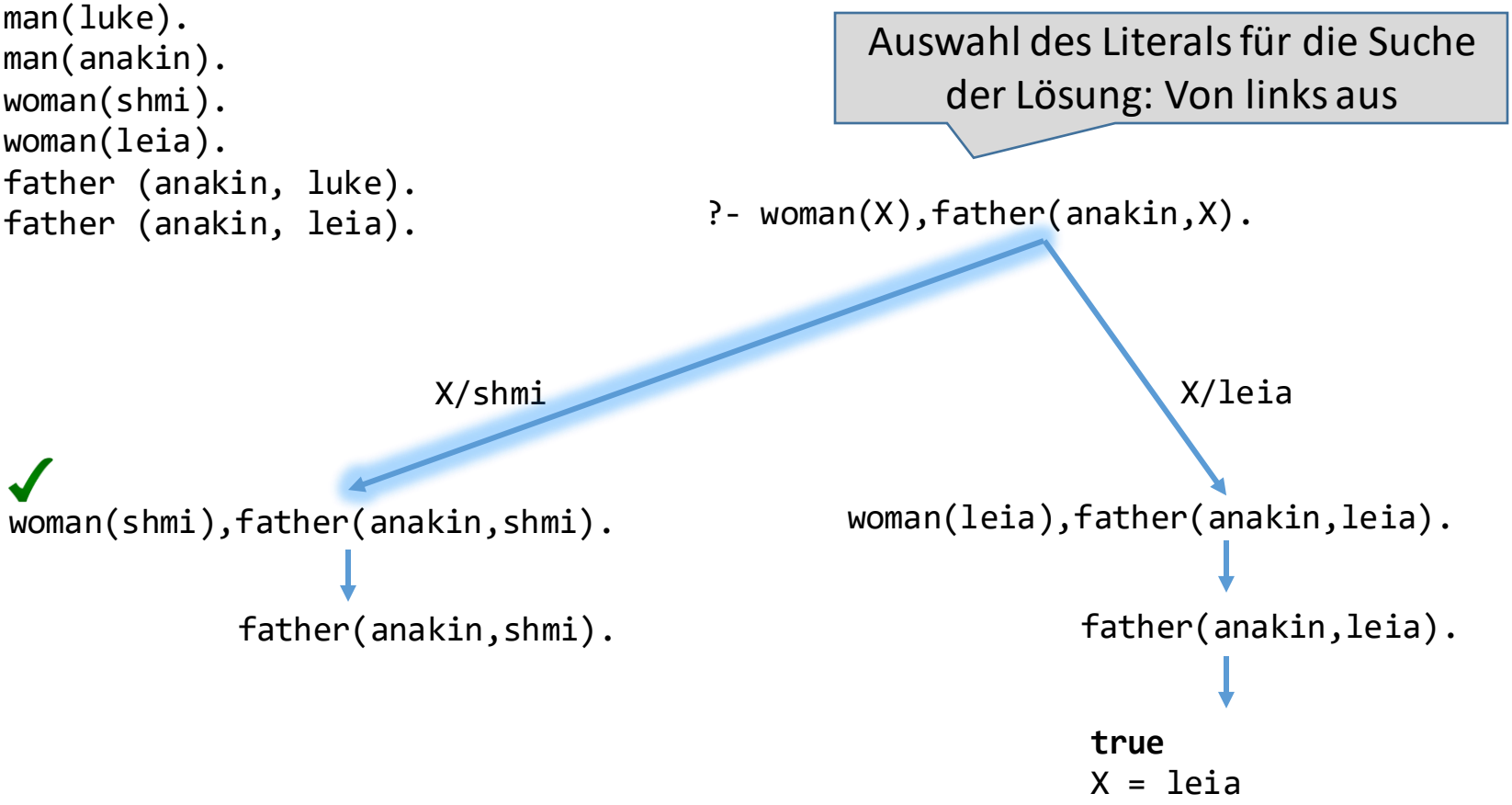
Resolution Beispiel

```
/* FAKTEN */
man(luke).
man(anakin).
woman(shmi).
woman(leia).
father (anakin, luke).
father (anakin, leia).
```



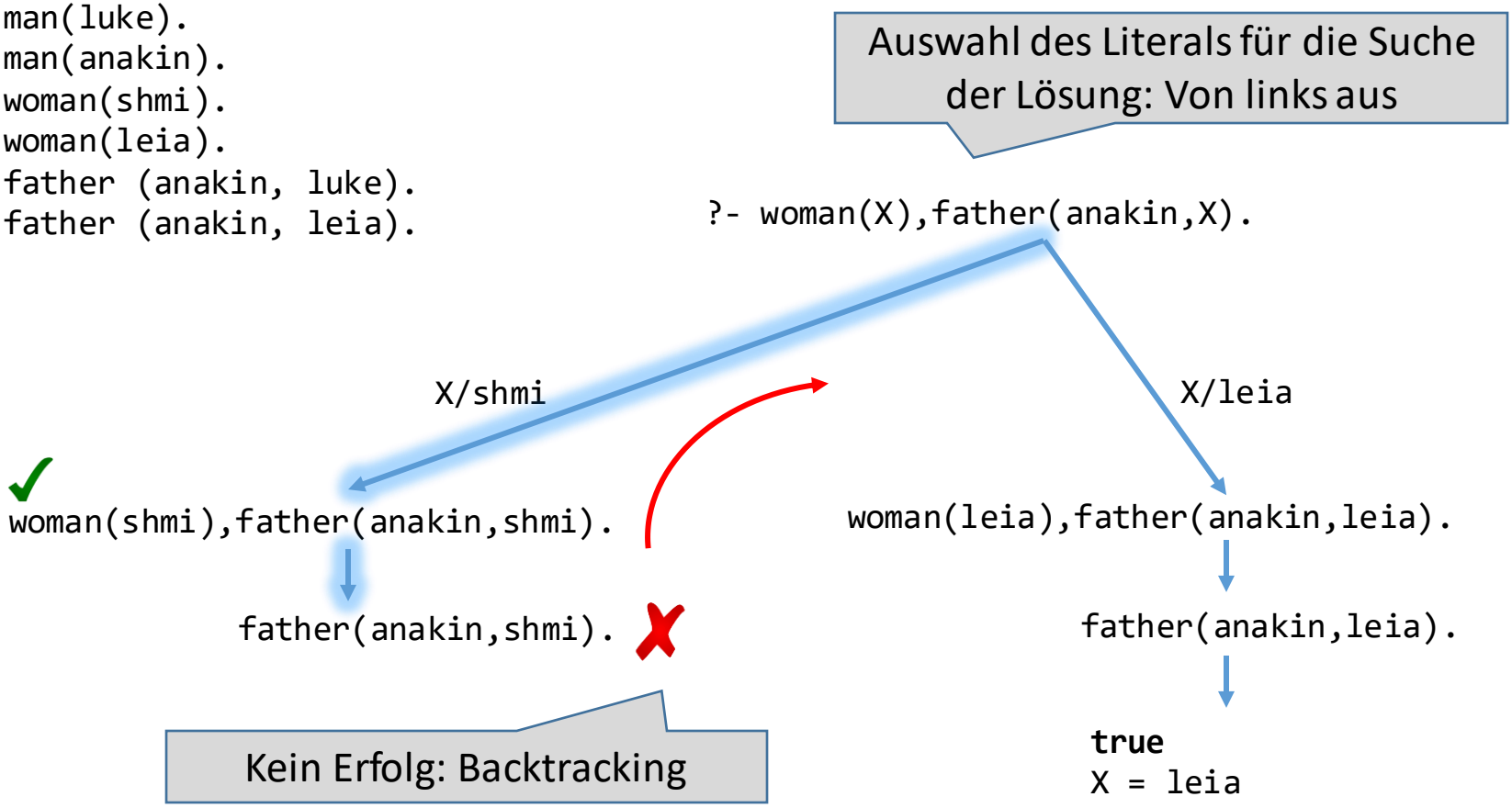
Resolution Beispiel

```
/* FAKTEN */
man(luke).
man(anakin).
woman(shmi).
woman(leia).
father (anakin, luke).
father (anakin, leia).
```



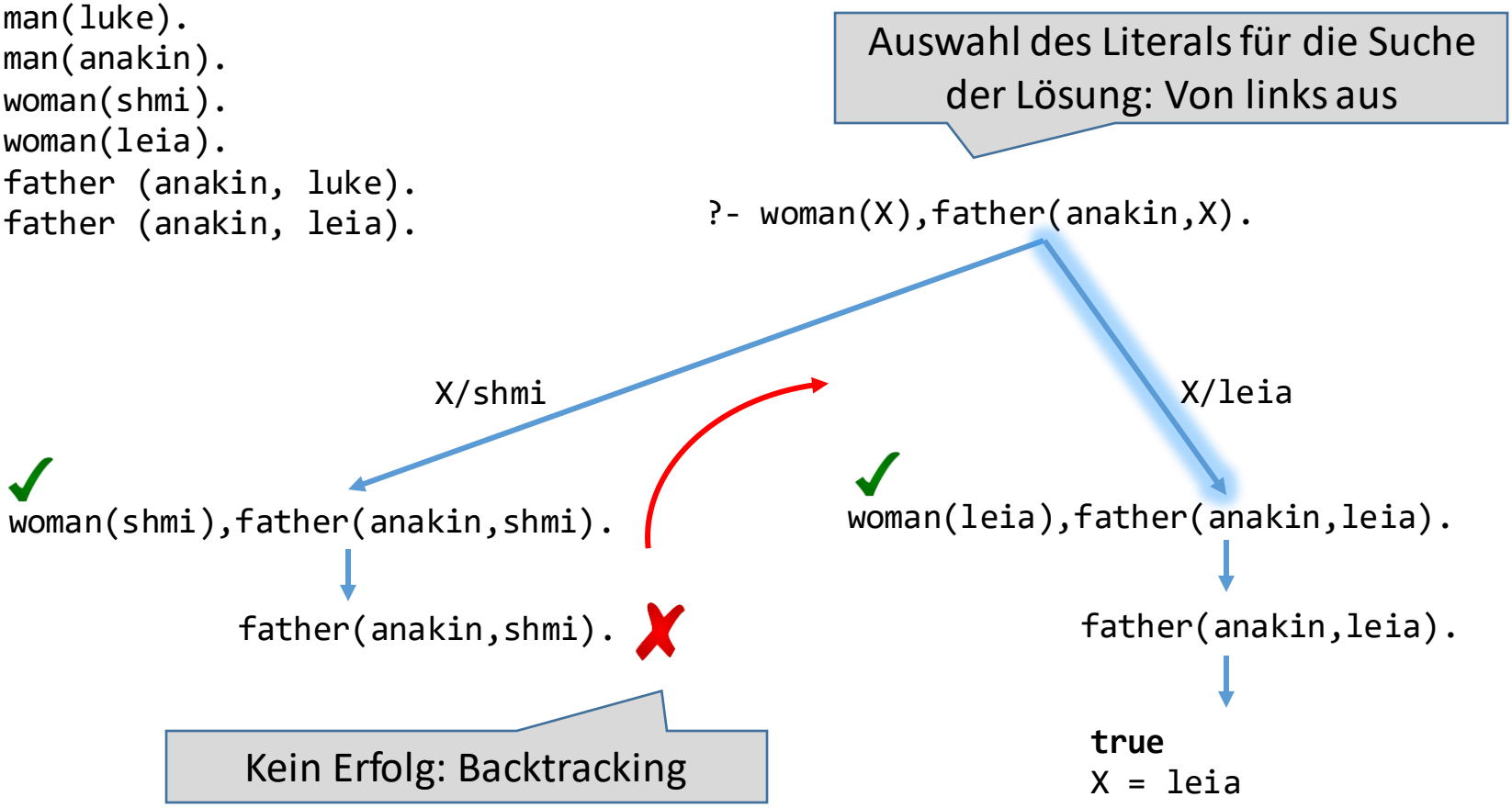
Resolution Beispiel

```
/* FAKTEN */
man(luke).
man(anakin).
woman(shmi).
woman(leia).
father (anakin, luke).
father (anakin, leia).
```



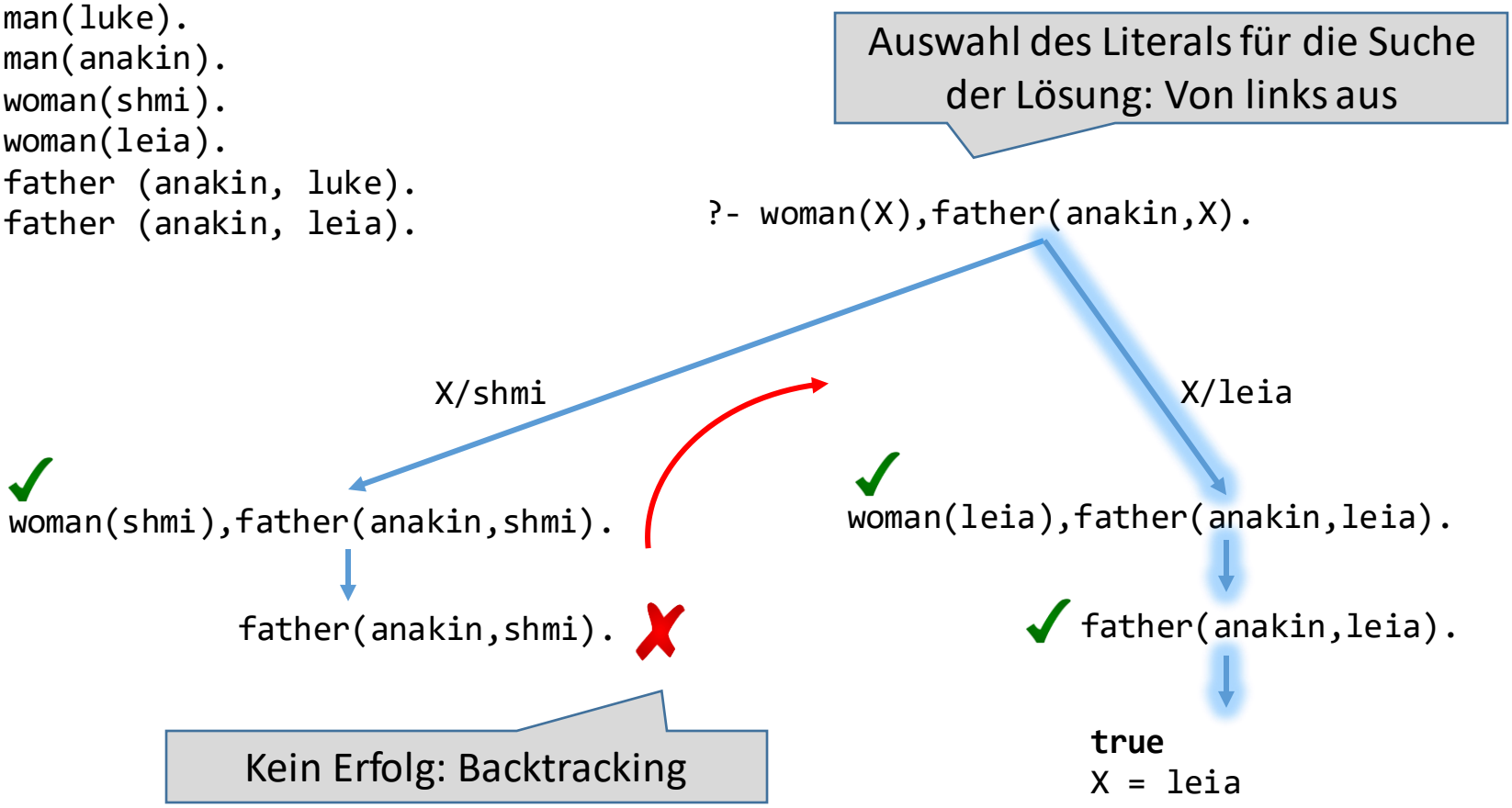
Resolution Beispiel

```
/* FAKTEN */
man(luke).
man(anakin).
woman(shmi).
woman(leia).
father (anakin, luke).
father (anakin, leia).
```



Resolution Beispiel

```
/* FAKTEN */
man(luke).
man(anakin).
woman(shmi).
woman(leia).
father (anakin, luke).
father (anakin, leia).
```



Inhalt

Logische Programmierung

- Einführung in Prolog
- Sprach-Syntax
- Unifikation und Resolution
- Rekursion
- Vordefinierte Prädikate

Rekursion

Prädikate können **rekursiv** definiert werden, d.h. in einem Prädikat kann es selbst wieder auftauchen

Rekursionsanker: Eltern sind direkt Vorfahren

```
ancestor(A,B) :- parent(A,B).
ancestor(A,B) :- parent(A,A_B), ancestor(A_B,B).
```

Rekursionsschritt: Vorfahren können beliebig viele Generationen entfernt sein

➢Prolog wird das Prädikat so oft in sich selbst einsetzen bis eine gültige Belegung gefunden ist.

Rekursion (2)

Wenn zwischen den Rekursionsschritten Datenaustausch nötig ist, kann dies über Variablen geschehen.

Akkumulator: Zwischenrechnungen werden weiter nach „unten“ gegeben.

Ergebnisvariable: Im Rekursionsanker wird das Ergebnis mit dieser Variable verknüpft.

```
cntAncestor(A,B,Cnt,Erg) :- parent(A,B), Erg = Cnt.
cntAncestor(A,B,Cnt,Erg) :- parent(A,A_B), Cnt1 is Cnt+1,
                            cntAncestor(A_B,B,Cnt1,Erg).

?- cntAncestor(shmi, luke, 1, Erg).
Erg = 2
```

Cnt muss mit dem Startwert (1) initialisiert werden, für Erg soll eine Belegung gefunden werden.

In jedem Rekursionsschritt wird dieser Zähler inkrementiert.

Rekursion (3)

Alternative Umsetzung der Rekursion:

Setzen des Startwerts für Erg im Rekursionsanker.

```
cntAncestor2(A,B,1) :- parent(A,B).
cntAncestor2(A,B,Erg) :- parent(A,A_B),
                         cntAncestor2(A_B,B,Erg1), Erg is Erg1 + 1.

?- cntAncestor2(shmi, luke, Erg).
Erg = 2
```

Berechnung des Ergebnisses auf dem Weg nach „oben“

Rekursive Berechnungen

Beispiel: Berechnung der Fakultät.

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

?- fak(5,F).
F = 120
```

Rekursionsschritt:
Fakultät von n-1

Ergebnis wird **nach** dem Rekursionsschritt aktualisiert

Regeln für Rekursion

Nicht-Rekursive Klauseln zuerst!

- Erinnerung: Tiefensuche im Lösungsbaum.
- Es werden die Klauseln nacheinander von oben nach unten auf Unifizierbarkeit geprüft
- Steht die rekursive Klausel oben, ergibt sich ein **unendlicher Lösungsbaum bevor** der Rekursionsanker geprüft wird

Innerhalb einer Klausel: Nicht-rekursive Literale vor rekursivem Literal!

- So wird sichergestellt, dass Manipulationen an den Übergabewerten vor dem Rekursionsschritt geschehen

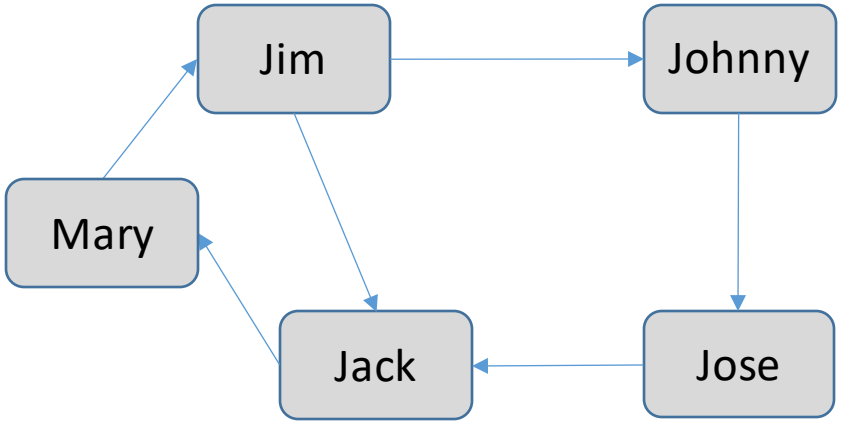
Zyklen in den Daten (Graph):

- Keine Zyklen innerhalb eines Prädikats!
- Besuchte Knoten in weiterem Parameter „merken“ (siehe nächste Folie)

Zyklische Daten

Aufgabe: Es soll herausgefunden werden, über wie viele Ecken zwei Personen sich kennen.

```
friend(jim,johnny).
friend(johnny,jose).
friend(jose,jack).
friend(jack,mary).
friend(mary,jim).
friend(jim,jack).
knows(A,B) :- friend(A,B).
knows(A,B) :- friend(A,C), knows(C,B).
```



Problem: Zyklische Abhängigkeiten ermöglichen unendliche Rekursion: knows(jim,johnny) → jim,johnny,jose,jack,mary,jim,johnny, …..

➢Mit etwas Pech findet man sogar nur einen unendlichen Pfad

Zyklische Daten

Lösung: „Merken“ schon verwendeter Fakten(Daten)

Liste Visited enthält alle schon gefundenen „Zwischenfreunde“

Prüft ob dieser Freund C nicht schon einmal besucht wurde

```
knows(A,B,_) :- friend(A,B).
knows(A,B,Visited) :- friend(A,C),\+ member(C,Visited),
                      knows(C,B,[A|Visited]).
```

Merke Zwischenwert in Liste

```
?- knows(jim,johnny, [ ]).
true ;
false
```

Leere Liste: Am Anfang wurde noch keiner besucht

Inhalt

Logische Programmierung

- Einführung in Prolog
- Sprach-Syntax
- Unifikation und Resolution
- Rekursion
- Vordefinierte Prädikate

Vordefinierte Prädikate

Einige weitere wichtige Prädikate:

```
write(X). /* Druckt X auf die Konsole. Gibt immer true zurück */

findall(X,G,L). /* Alle Variablenbelegungen X, die die Anfrage G
                   erfüllen, sind in L enthalten. */

setof(X,G,L). /* entspricht findall, aber nach X sortiert */
```

Beispiel für findall:

```
?- findall(X,father(anakin,X),L).
L = [luke, leia].
```

Lernziele

- ❑ Was ist logische Programmierung?
- ❑ Was ist die Closed-World Assumption?
- ❑ Wie setzen sich Terme in Prolog zusammen?
- ❑ Was sind Variablen in Prolog und welchen Typ haben sie?
- ❑ Welche Typen gibt es in Prolog?
- ❑ Was unterscheidet einen Funktor von einer Funktion in anderen Programmiersprachen?
- ❑ Was ist der Unterschied zwischen einer Anfrage und einer Regel?
- ❑ Warum gibt es so viele Vergleichsoperatoren?
- ❑ Wann werden arithmetische Ausdrücke ausgewertet?
- ❑ Wie erhalte ich das erste Element einer Liste?
- ❑ Was passiert, wenn man an eine Liste mit Zahlen einen String anhängt?
- ❑ Wie funktioniert Unifikation und wann sind zwei Terme unifizierbar?
- ❑ Wie löst der Prolog-Interpreter die ihm gestellten Anfragen?
- ❑ Wieso sollten rekursive Ausdrücke immer rechts innerhalb einer Klausel und unten bei mehreren Klauseln stehen?
- ❑ Wie vermeidet man eine unendliche Auswertung zyklischer Daten?

Literatur

- Leon Sterling and Ehud Shapiro: The Art of Prolog. MIT Press, 1994
- http://www.learnprolognow.org/
- http://www.swi-prolog.org/
- Für die, die mehr wissen wollen:
  Stuart Russell, Peter Norvig: Künstliche Intelligenz.
  Pearson,2012