

Kapitel 04:

Anfangsgründe von Prolog

Grundlage: Covington, Davis;
sowie Vorlesungsunterlagen von FAU-Inf8
(Beckstein [jetzt FSU Jena], Jaksch, Zinn)



Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Ausblick: Kapitel 04; Teil 1

- **Lernziele:**

- Was ist logik-orientierte Programmierung?
- Worin unterscheidet sich logik-orientierte Programmierung von imperativer Programmierung?
- Welche Methoden bietet Prolog, um ein Problem zu lösen?
- Wie werden Relationen in Prolog umgesetzt?

Inhalt: Teil 1

- Logik-orientierte Programmierung
 - Relationen & Hornklauseln
- Prolog-Notation
- Unifikation
- Backtracking
- Relationen definieren
- Konjunktive Ziele
- Prozedurale Semantik
- Negative Ziele
- Test auf Gleichheit
- Anonyme Variable

Inhalt: Teil 2

- Prolog-Systeme
- Darstellungsalternativen
- Funktoren
- Funktionen
- Arithmetische Ausdrücke
- Rekursion
- Listen
- Rekursion über Listen
- Beispiel: Baumtraversierung
- Strukturen

Logik-orientierte Programmierung

- **Ziel:** Zerlegung eines Berechnungsproblems in zwei Teilprobleme
 - **Was** ist zu berechnen?
 - **Wie** ist die Berechnung durchzuführen?
- Anders gefragt:

Ist es möglich, **Spezifikationen als ausführbare Ausdrücke** einer Programmiersprache zu formulieren?

Die logik-orientierte Programmierung erreicht dies durch Auszeichnung einer **Teilsprache** der formalen Logik, die ausdrucksstark genug ist, um darin **Berechnungsprobleme** auszudrücken, aber gleichzeitig "schwach" genug ist, um eine **effiziente** und **kontrollierbare** prozedurale **Interpretation** zu haben.

Logik-orientierte Programmierung

- Ein Programm in einer logik-orientierten Programmiersprache soll ein **effektives** Programm sein, das auf einem Rechner ausführbar ist:
 - Die Ablaufsteuerung (**wie**) wird erreicht durch die Auswertungs-Reihenfolge der Sprache. Ausgehend von einer **Ziel**-Formel muss es möglich sein, die Anordnung ihrer Teilformeln und der korrespondierenden Teilziele so zu wählen, dass die Berechnung **effizient** erfolgen kann.
 - Das Ziel der Berechnung führt zu einem Ergebnis (**was**), das als eine **logische Folgerung** aus dem (Logik-) Programm gesehen werden kann.
- Das Potential der logik-orientierten Programmierung entstammt einer **prozeduralen Interpretation** der Logik.

Logik-orientierte Programmierung

- Für jede "Computational Logic"
(logik-orientierte Programmierung, automatisches Beweisen,
Wissensrepräsentation/-verarbeitung, Semantic Web, ...) gilt:

Bei der Wahl der logischen (Repräsentations-) Sprache sind die Definitionen zu unterscheiden für:

- die **formale Sprache**
Syntax und Semantik | Ausdruckskraft
 - das **Schlußfolgerungsproblem** (Inferenzproblem)
Entscheidbarkeit | Komplexität
 - die **Problemlösungs-Prozedur**
Korrektheit und Vollständigkeit | (asymptotische) Komplexität
- Wir werden noch sehen: Die Wahl der konkreten logik-sprachlichen Mittel ist stets ein Kompromiss zwischen **Ausdruckskraft** und **Traktabilität**.

Logik-orientierte Programmierung

- **Relationale Programmierung:**

- Im Unterschied zur prozeduralen Programmierung beruht die logik-orientierte Programmierung auf **Relationen**.
- **Weitergehend** als bei der gerichteten prozeduralen Berechnung (wohldefinierte Eingabeparameter führen zu einem eindeutigen Wert) sollen mit relationalen Ausdrücken der Form:

<relation>(<eingabeparameter> , <ausgabeparameter>).

bei hinreichender Angabe von Parametern

- Berechnungen in **beiden Richtungen** erfolgen, und
- Ausdrücke mehr als einen (Ausgabeparameter-) Wert haben können (**Nichtdeterminismus**).

Logik-orientierte Programmierung

- **Prolog**: Sprache für logik-orientierte Programmierung
 - **Herangehensweise**: Zunächst exemplarisch/informell und auf Programmierpraxis gerichtet; der theoretische Hintergrund wird in Kap. 8 ausgearbeitet.
 - Prolog beruht auf einer echten **Teilsprache der Quantorenlogik: Horn-Formeln**

$$(a_1 \wedge \dots \wedge a_n) \rightarrow b$$

Antecedens
Rumpf

Konsequens
Kopf

(Alfred Horn: On Sentences Which are True of Direct Unions of Algebras. Journal of Symbolic Logic, Vol. 16, No. 1 (1951), 14–21.)

Logik-orientierte Programmierung

- Klassisch, d.h. unter Voraussetzung der Wahrheitsdefinitheit (tnd) gilt:

$$\begin{aligned}(a_1 \wedge \dots \wedge a_n) \rightarrow b &\prec \neg(a_1 \wedge \dots \wedge a_n) \vee b \\ &\prec \neg a_1 \vee \dots \vee \neg a_n \vee b \quad (\text{"Klauseln"})\end{aligned}$$

- Unter Berücksichtigung der "Klauselnormalform" (Kap. 8) sagt man auch:
"Horn-Klauseln sind Klauseln mit höchstens einem positiven Literal (Elementaraussageform)."

Logik-orientierte Programmierung

Es gibt vier Arten von Hornklauseln:

1. Einheitsklauseln: \mathbf{b} bzw. $\rightarrow \mathbf{b}$

(kein Antecedens, nur Konsequens) behaupten die Wahrheit des Konsequens: "**Fakten**".

2. Nicht-Einheitsklauseln: $(a_1 \wedge \dots \wedge a_n) \rightarrow b$

(Antecedens und Konsequens) behaupten die Wahrheit des Konsequens, wenn das Antecedens wahr ist: "**Regeln**".

3. Negative Klauseln: $(a_1 \wedge \dots \wedge a_n) \rightarrow$ bzw. $\neg(a_1 \wedge \dots \wedge a_n)$

(nur Antecedens, kein Konsequens) verneinen (negieren) die Wahrheit des Antecedens: "**Anfragen**" (**Ziele**, "**goals**").

4. Leere Klausel ("falsum"): \square

Klauseln der Arten (1) und (2) heißen auch **Definite Klauseln**

Prolog-Notation

- Syntaktische Basiseinheit: **Term**
 - "**Atome**" – Namen für Individuen (Konstanten) und Prädikate; beginnen mit Kleinbuchstaben:
x, erlangen, a_und_b, 'na so was'
 - **Zahlen**
 - **Strukturen**: functor(arg), located_in(atlanta,texas), ...
 - **Variablen** – beginnen mit Großbuchstaben oder mit `_` :
X, Result, _value ;
Variablen gelten (implizit) als **allquantifiziert**.

Prolog-Notation

- **Fakten:** $q.$ statt $\rightarrow q$
- **Regeln:** $r :- p.$ statt $p \rightarrow r$ (if p then r)
 $r :- p, q.$ statt $p \wedge q \rightarrow r$ (if p and q then r)
- **Ziele:** $:- p.$ statt $p \rightarrow$

$:-$ ist ebenfalls ein Funktor in **Infix-Schreibweise**;
statt $a(X) :- b(X).$ könnte man auch schreiben $:-(a(X), b(X)).$

Beispiel 1: Geographische Datenbasis

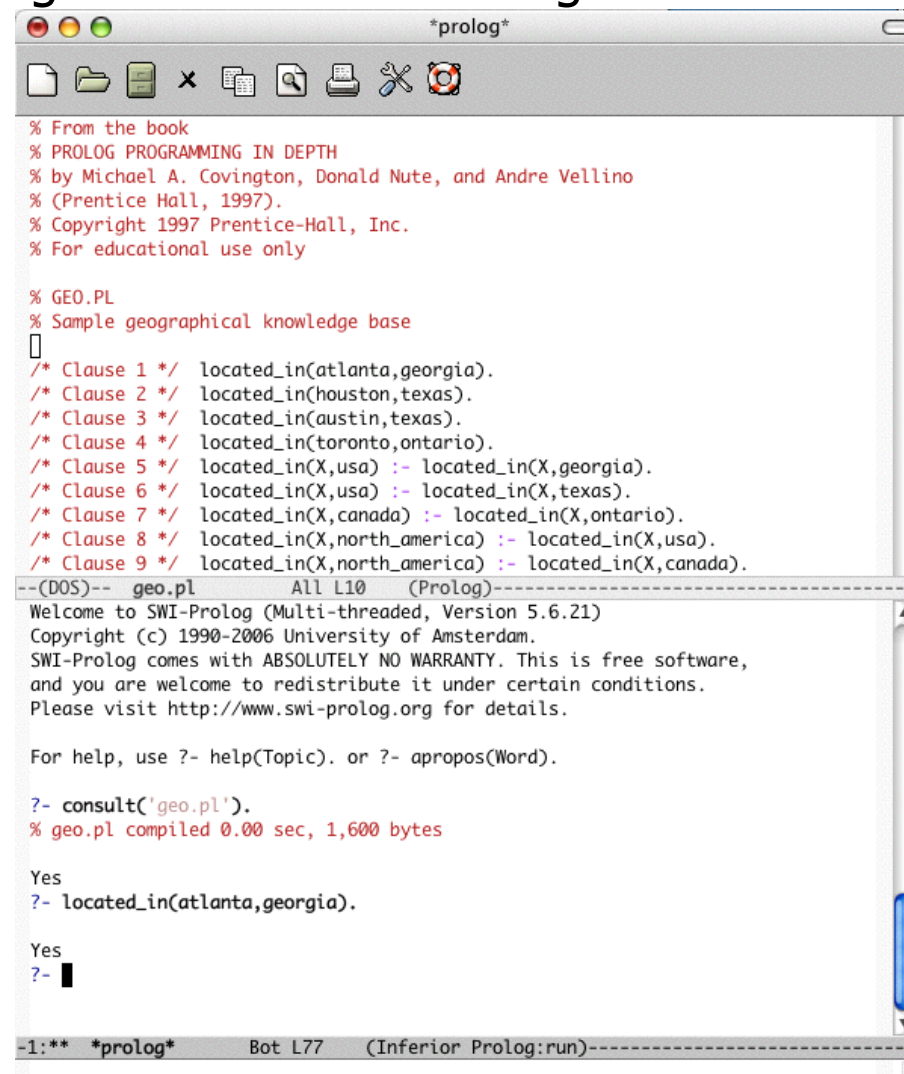
```
% From the book: PROLOG PROGRAMMING IN DEPTH  
% by Michael A. Covington, Donald Nute, and Andre Vellino  
% Copyright 1997 Prentice-Hall, Inc.  
% For educational use only
```

```
% GEO.PL  
% Sample geographical knowledge base
```

```
located_in(atlanta,georgia).          /* Clause 1 */  
located_in(houston,texas).           /* Clause 2 */  
located_in(austin,texas).            /* Clause 3 */  
located_in(toronto,ontario).         /* Clause 4 */  
located_in(X,usa) :- located_in(X,georgia). /* Clause 5 */  
located_in(X,usa) :- located_in(X,texas). /* Clause 6 */  
located_in(X,canada) :- located_in(X,ontario). /* Clause 7 */  
located_in(X,north_america) :- located_in(X,usa). /* Clause 8 */  
located_in(X,north_america) :- located_in(X,canada). /* Clause 9 */
```

Beispiel 1: Geographische Datenbasis

- Ein erster Programmmlauf: SWI-Prolog in Emacs



```
*prolog*
% From the book
% PROLOG PROGRAMMING IN DEPTH
% by Michael A. Covington, Donald Nute, and Andre Vellino
% (Prentice Hall, 1997).
% Copyright 1997 Prentice-Hall, Inc.
% For educational use only

% GEO.PL
% Sample geographical knowledge base
[]
/* Clause 1 */ located_in(atlanta,georgia).
/* Clause 2 */ located_in(houston,texas).
/* Clause 3 */ located_in(austin,texas).
/* Clause 4 */ located_in(toronto,ontario).
/* Clause 5 */ located_in(X,usa) :- located_in(X,georgia).
/* Clause 6 */ located_in(X,usa) :- located_in(X,texas).
/* Clause 7 */ located_in(X,canada) :- located_in(X,ontario).
/* Clause 8 */ located_in(X,north_america) :- located_in(X,usa).
/* Clause 9 */ located_in(X,north_america) :- located_in(X,canada).

--(DOS)-- geo.pl All L10 (Prolog)-----
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.21)
Copyright (c) 1990-2006 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- consult('geo.pl').
% geo.pl compiled 0.00 sec, 1,600 bytes

Yes
?- located_in(atlanta,georgia).

Yes
?- █

-1:** *prolog* Bot L77 (Inferior Prolog:run)-----
```

Beispiel 1: Geographische Datenbasis

- Anfrage bedingt Regelanwendung:

?- located_in(atlanta,usa).

Yes

?- trace.

Yes

[trace] ?- located_in(atlanta,usa).

Call: (?) located_in(atlanta, usa) ? <CR> creep

Call: (8) located_in(atlanta, georgia) ? creep

Exit: (8) located_in(atlanta, georgia) ? creep

Exit: (?) located_in(atlanta, usa) ? creep

Yes

- No: **Erfolglose Anfrage** ("failure")
(Erfolg ("success"): explizit **zugesichert** oder **ableitbar** =?= "wahr")
 - später: Closed World Assumption (CWA))

?- located_in(atlanta,texas).

No

Beispiel 1: Geographische Datenbasis

- Anfragebeantwortung durch Variablen-Instantiierung:

```
?- located_in(X,texas).
```

```
X = houston ;           % Benutzereingabe!
```

```
X = austin ;
```

```
No
```

- "Trick" zur Auflistung aller Ergebnisse:

```
?- located_in(X,texas), write(X), nl, fail.
```

```
houston
```

```
austin
```

```
No
```

Beispiel 1: Geographische Datenbasis

- **Reversibilität.** (nicht immer gegeben)

?- located_in(austin,X).

X = texas ;

X = usa ;

X = north_america ;

No

- Abfrage **aller** Argumente: Was liegt worin?

?- located_in(X,Y).

..... % Ausgabe aller Paare

?- located_in(X,X).

No % failure!

Unifikation: Beispiel 1

- **Anfragebeantwortung**

- Der erste Schritt besteht in einem **Mustervergleich**:
- Versuch der Passung ("**Unifikation**") der Anfrage mit einer Elementaraussage ("Fakt") oder dem Kopf (Konsequenz) einer Regel;
- ordnet Variablen – sofern vorhanden – einen Wert zu, um die Passung herzustellen: **Variablen-Instantiierung**

- Im Beispiel:

?- located_in(austin,north_america).

unifiziert mit dem Kopf der Klausel 8 durch **Instantiierung** von X mit austin .

Unifikation: Beispiel 1

Die rechte Seite der Regel wird dann zum neuen **Ziel**:

Ziel: `?- located_in(austin,north_america).`

Klausel 8: `located_in(X,north_america) :- located_in(X,usa).`

Instantiierung: `X = austin`

Neues Ziel: `?- located_in(austin,usa).`

Die neue Anfrage kann dann mit Klausel 6 **unifiziert** werden:

Ziel: `?- located_in(austin,usa).`

Klausel 6: `located_in(X,usa) :- located_in(X,texas).`

Instantiierung: `X = austin`

Neue Anfrage: `?- located_in(austin,texas).`

Unifikation: Beispiel 1

- Beispiel (Forts.):
Diese Anfrage passt auf Klausel 3 und daher **erfolgreich terminiert** wird.
Hätte es eine Anfrage gegeben, die nicht unifizierbar ist, wäre die Berechnung gescheitert (Termination mit "failure")
- Beachte:
X musste **zweimal instantiiert** werden – zuerst beim Aufruf von **Klausel 8**, dann erneut beim Aufruf von **Klausel 6**.

Unifikation

- **Prinzip:** Bei **gleichbenannten** Variablen handelt es sich nicht um dieselbe Variable, wenn sie nicht in **derselben Klausel** oder **derselben Anfrage** vorkommen!
 - Variablen-Instantiierung in Prolog korrespondiert mit der **Parameter-Übergabe** in prozeduralen Programmiersprachen (Bindung), **nicht mit der Speicherung** eines Werts in (der Lokation) einer Variablen.
 - Eine **uninstantiierte** Variable kann mit einer **anderen unifiziert** werden: "**Sharing**".
In diesem Fall haben wir zwei Namen für dieselbe Variable (Koreferenz); wird die eine gebunden, so **simultan** auch die andere.

Unifikation

- Die Unifikationsoperation versucht, zwei Terme syntaktisch **zur Übereinstimmung zu bringen**.
Zwei Terme passen übereinander ("match"), **wenn sie gleich sind, oder**, wenn sie Variablen enthalten, die Variablen derart instantiiert werden können, dass die Terme **gleich werden**.
 - Da beide Terme Variablen enthalten können, handelt es sich um einen **bidirektionalen Mustervergleich**.
 - Variablen, die in den Termen enthalten sind, werden an ihr "Gegenstück" gebunden.
- **Beispiel:** Die Unifikation von
 $f(a, X, g(X))$ und $f(a, b, Y)$
führt zu den Bindungen
 $X = b$ und $Y = g(b)$

Beispiel 1: Unifikation

- **Definition: Passung** (Matching)

1. Sind $term_1$ und $term_2$ Konstanten, so besteht eine Passung dann und nur dann, wenn es sich um **dasselbe Atom** oder **dieselbe Zahl** handelt.
2. Ist $term_1$ eine Variable und $term_2$ ein beliebiger Term, so besteht eine Passung und $term_1$ wird mit $term_2$ instantiiert; analog für den umgekehrten Fall. Sind beide Terme Variablen, so werden sie **wechselseitig instantiiert** ("value sharing").
3. Sind $term_1$ und $term_2$ zusammengesetzte Terme, dann besteht Passung dann und nur dann, wenn
 - sie **denselben Funktor** und **dieselbe Stelligkeit** (Parameteranzahl) haben,
 - zwischen **allen** korrespondierenden Argumenten **Passung** besteht, und
 - die Variablen-Instantiiierungen **kompatibel** sind.
4. Zwischen zwei Termen besteht Passung genau dann, wenn sich eine Passung gemäß 1–3 ergibt.

Beispiel 1: Backtracking (Zurücksetzen)

- Wenn eine Anfrage mit **mehreren Regeln** unifiziert werden kann – **wie wird entschieden**, welche benutzt werden soll ?

Die Unifikation von

?- located_in(austin,usa).

mit Klausel 5 erzeugt:

?- located_in(austin,georgia).

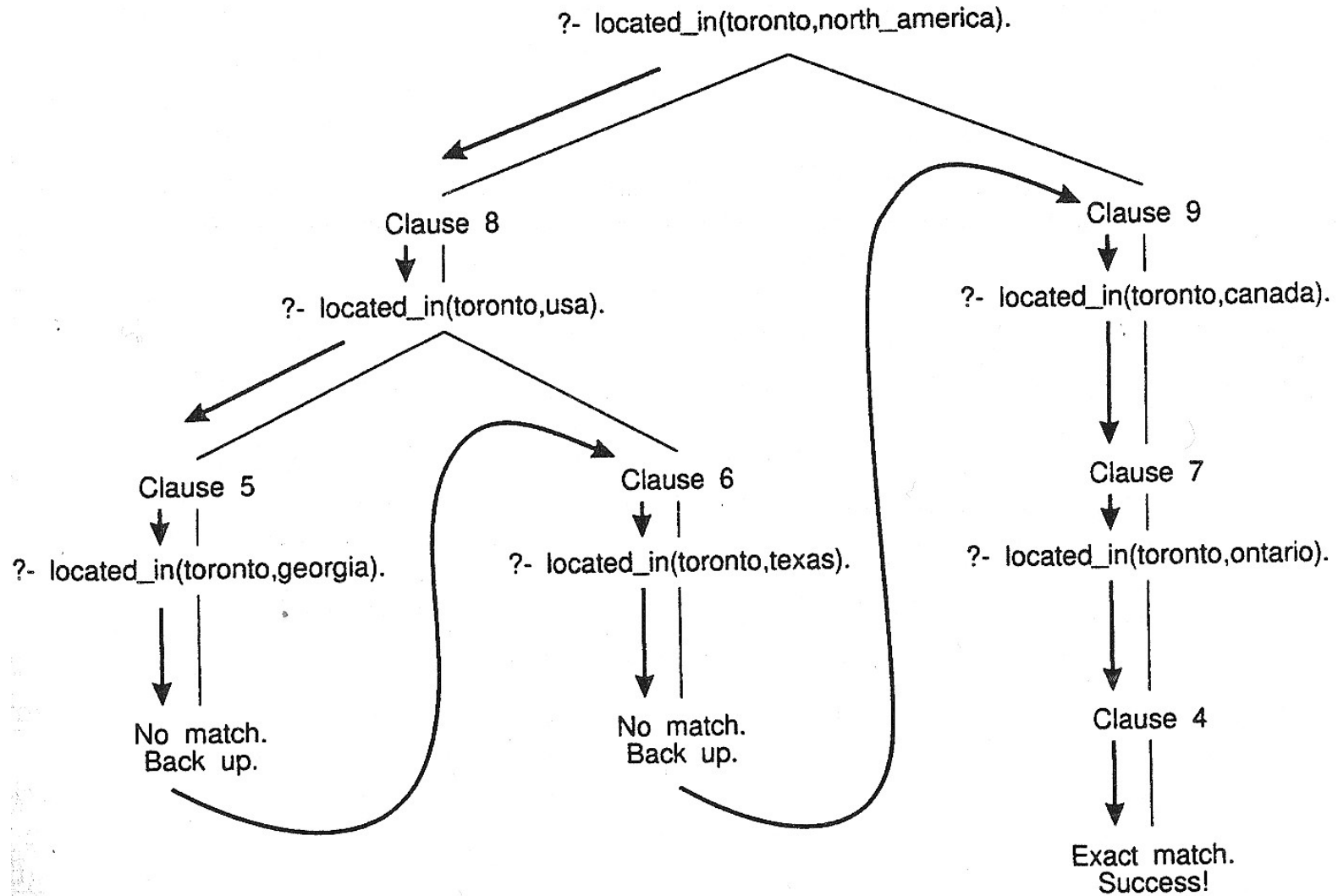
und **scheitert**. Unifikation mit Klausel 6 jedoch erzeugt

?- located_in(austin,texas).

und ist **erfolgreich**. Klausel 5 führt in eine **Sackgasse!**

- Es ist nicht von vorneherein bekannt, welche Klausel erfolgreich sein wird, aber das Prolog-System kann aus Sackgassen **zurücksetzen** ("**back up**"). Die Regeln werden in der gegebenen **Reihenfolge** getestet; führt eine Regel nicht zum Erfolg, wird **zurückgesetzt** und die nächste probiert, etc.

Backtracking



© Covington

Backtracking

- ("Chronologisches") Backtracking geht immer zu der **zuletzt unversuchten** Alternative zurück.
- Wird eine Antwort gefunden ("success"), so wird die Verarbeitung **terminiert**, es sei denn, der Benutzer fragt nach **Alternativen** – in diesem Fall wird **Backtracking** versucht, um einen anderen erfolgreichen Pfad zu suchen.
- Diese Suchstrategie in Bäumen (Suchraum) heißt **Tiefensuche** ("depth-first search").
- Grundsätzlich werden auf diese Weise **alle Lösungen** gefunden, lediglich die **Reihenfolge** kann sich mit der Anordnung der Klauseln in der Wissensbasis bzw. ihrer Indexierung ändern.

Beispiel 2: Familienbeziehungen

- Datenbasis („Wissensbasis“) (© Davis)

```
father(paul, rob).           % paul ist Vater von rob
father(rob, bev).
father(rob, theresa).
father(jeff, aaron).
mother(mary, rob).
mother(dorothy, bev).
mother(dorothy, theresa).
mother(theresa, aaron).
```

Relationen definieren: Beispiel 2

- Neue Relationen auf der Basis vorhandener definiert:
Für die Elternrelation werden zwei **Regeln** benötigt

`parent(M, C) :- mother(M, C).`

`parent(F, C) :- father(F, C).`

`?- parent(X, theresa).`

liefert erst `X = dorothea` (Regel 1), dann `X = rob` (Regel 2)

- **Disjunktive Ziele**

Ohne auf die `parent`-Relation zurückzugreifen, könnte die Anfrage auch direkt ein disjunktives Ziel enthalten:

`?- mother(X, theresa); father(X, theresa).`

Die obigen Regeln kann man auch **zusammenfassen**:

`parent(X, Y) :- mother(X, Y) ; father(X, Y).`

Konjunktive Ziele: Beispiel 2

- **Konjunktive Ziele**

Wer ist bevs Vater und wer ist dessen Vater?

Großvater väterlicherseits: Finde F und G , so dass F Vater von bev und G Vater von F ist!

?- father(F , bev), father(G , F).

resultiert in $F = \text{rob}$ und $G = \text{paul}$

- Vertauschung der Teilziele in der Anfrage liefert **dasselbe Ergebnis**, jedoch wird die Berechnung **verlangsamt**, da dann das erste Teilziel zwei uninstantiierte Variable hat !
- Alle Großväter (väterlicher- und mütterlicherseits):
grandfather(G,C) :- father(F,C), father(G,F).
bzw. grandfather(G,C) :- father(G,F), father(F,C).
grandfather(G,C) :- mother(M,C), father(G,M).
- Großeltern allgemein:
grandparent(G,C) :- parent(G,X), parent(X,C).

Prozedurale Semantik

- Aus Programmiersprachen-Perspektive werden definite Klauseln als **Prozedurdefinitionen** interpretiert
 - **Regeln: Konsequens** ist Prozedurname mit Variablenliste (Formalparameter),
Antecedens ist Prozedurrumpf.
- Ein Hauptprogramm besteht aus einer Menge von **Bedingungen**, die (ohne Konklusion) zu erfüllen sind:
 - Erfüllen von Bedingungen = Ausführen von Zielen als Prozeduraufrufe.
- Ausführung von Prozeduraufrufen
 - Parameterübergabe: Bindung der Parameter durch **Unifikation**; die Variablenbindungen werden im gesamten Rumpf "substituiert".
 - Ausführung des Prozedurrumpfs: Ausführung der **Folge von Prozeduraufrufen** im Rumpf.
 - Dabei ist jeweils einer **auszuwählen** und eine Prozedur zu suchen, deren Kopf damit **unifiziert**; dann wird der Prozeduraufruf durch den Rumpf der gefundenen Prozedur unter Berücksichtigung der Variablenbindungen "ersetzt".
- Bei Sackgassen erfolgt **Backtracking**.

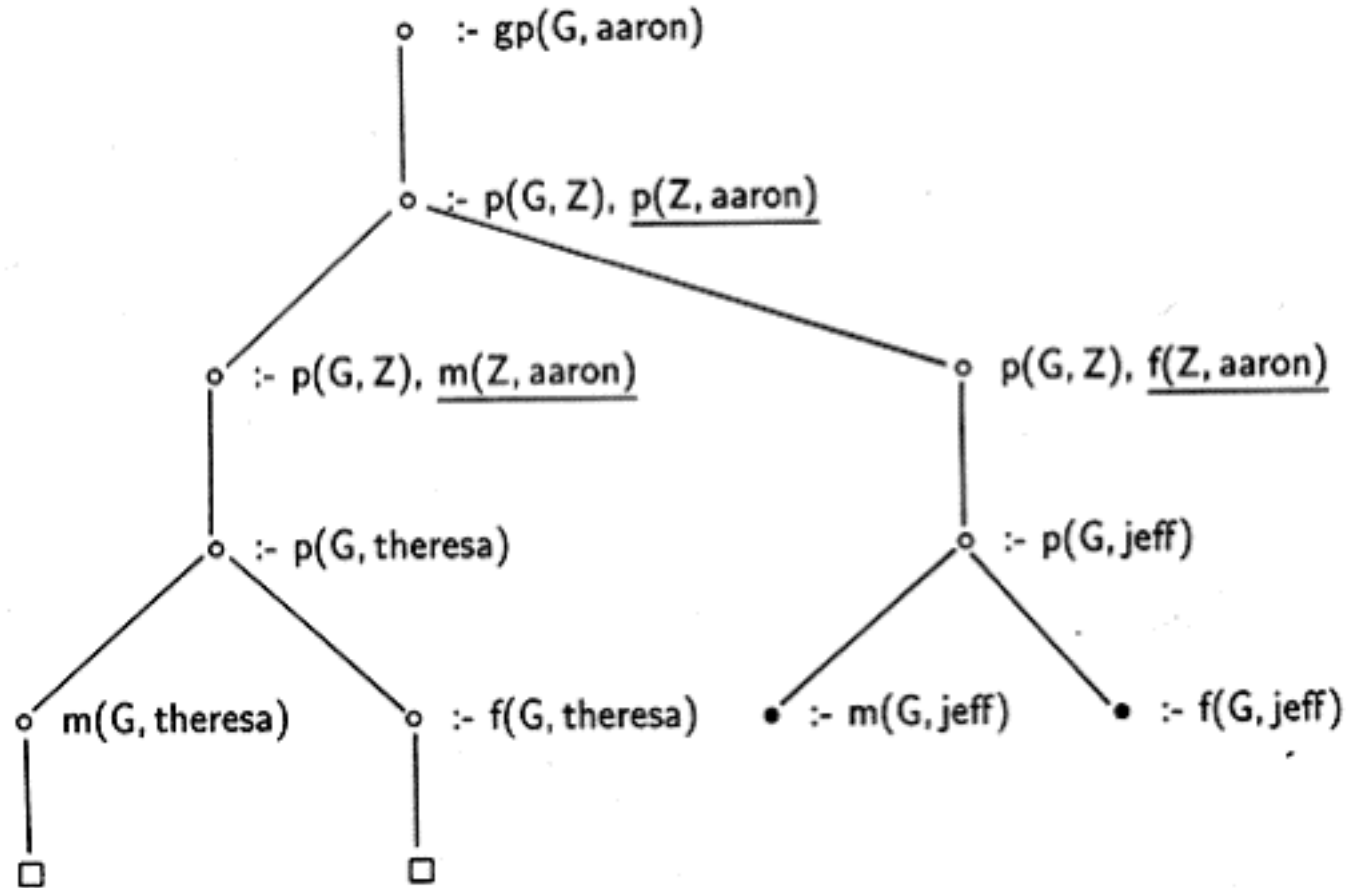
Prozedurale Semantik

- **Ziel:** Hat aaron ein Großelternteil?
?- grandparent(GP, aaron).
- Es gibt nur **eine Regel** mit grandparent/2 als Konsequens:
grandparent(G,C) :- parent(G,X), parent(X,C).
- Mit den Bindungen G = GP und C = aaron wird der Aufruf **ersetzt durch**
?- parent(GP, X), parent(X, aaron).
- Auswahl eines parent-Aufrufs, hier des zweiten
(Heuristik hier: Werterversorgung statt Reihenfolge).
- Aus den beiden Regeln für parent/2 werde die erste ausgewählt:
parent(M, C) :- mother(M, C).
- Mit den Bindungen M = X und C = aaron **resultiert der Aufruf**
?- parent(GP, X), mother(X, aaron).

Prozedurale Semantik

- Werde wiederum der **zweite Aufruf** ausgewählt.
Mit `mother(theresa, aaron)`. in der Datenbasis (leerer Rumpf!)
resultiert `X = theresa` und es bleibt noch
 ?`- parent(GP, theresa)`.
- Wird erneut die erste definierende Klausel für `parent/2` gewählt,
so erhalten wir
 ?`- mother(GP, theresa)`.
- Fakt `mother(dorothy, theresa)`. in der Datenbasis liefert die
erste Lösung `GP = dorothy`.
- **Weitere Lösungen** (hier `GP = rob`) werden durch **Backtracking**
gefunden.

Prozedurale Semantik



Prozedurale Semantik: Nichtdeterminismus

- **Nichtdeterminismus** in zweifacher Hinsicht:
- Die **Reihenfolge der Ausführung** der Prozeduraufrufe im Rumpf einer Prozedur ist frei.
- Bei Ausführung eines Prozeduraufrufs kann **jede Prozedurdefinition**, deren Kopf mit dem Aufruf unifiziert, ausgewählt werden.
- Das ursprüngliche Ziel und die dadurch sukzessive bestimmten Teilziele **spannen einen Suchraum auf**, der mehrere mögliche Berechnungen umfaßt; er enthält **alle möglichen Antworten** für das Ziel.

Prozedurale Semantik: Nichtdeterminismus

- Jeder Pfad vom Ziel zu einer \square repräsentiert eine **erfolgreiche Berechnung**;
verschiedene Pfade können in **verschiedenen Bindungen** für die **Variablen** des (ursprünglichen) Ziels resultieren.
- Wird eine **andere Reihenfolge** zur Erfüllung der Teilziele gewählt, ergibt sich ein **anderer Suchraum**.
- In der folgenden Abbildung wird an jedem **Auswahlpunkt** entschieden, das jeweils **erste Teilziel zuerst** zu bearbeiten.

Prozedurale Semantik

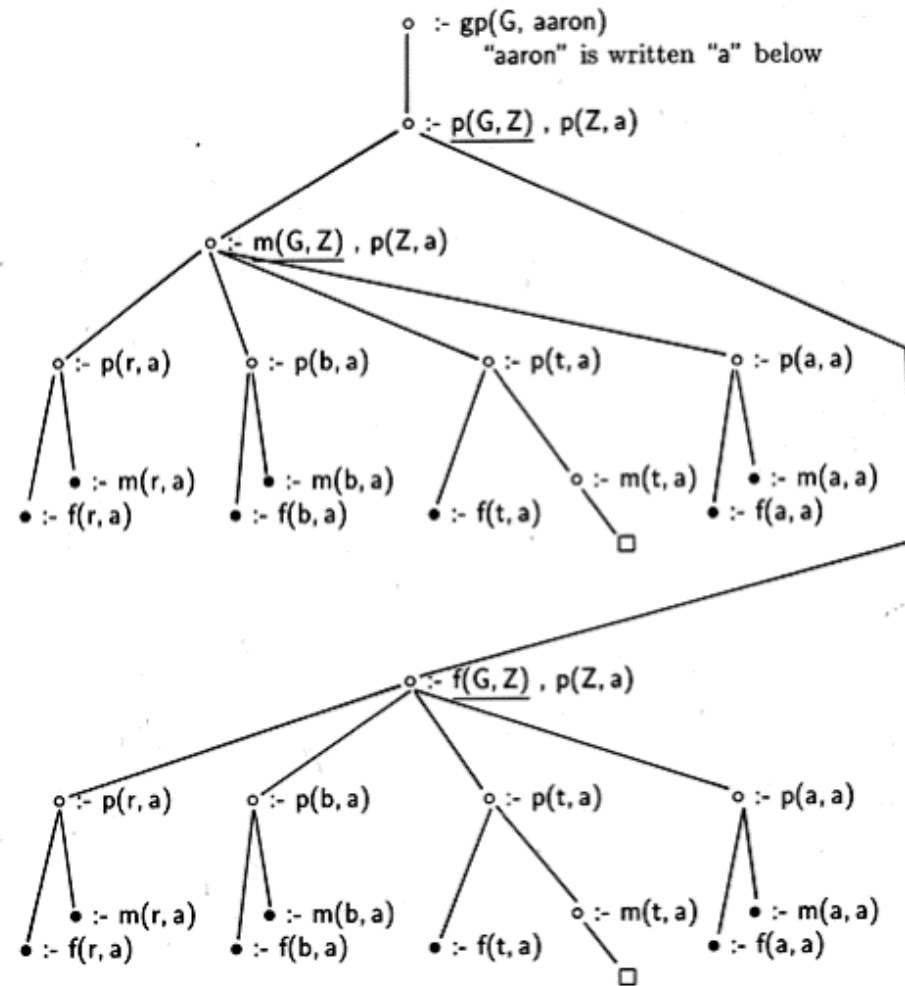


Figure 3.2 Search space obtained by changing the order of the subgoals. (©1985 IEEE)

Negative Ziele

- **Definition:** Prolog-Prädikat \neg "nicht-beweisbar"
– **Negation as Failure – (NAF) :**
Ist g ein Ziel, so ist $\neg g$ erfolgreich, wenn g scheitert,
und scheitert, wenn g erfolgreich ist.
- **Im Familien-Beispiel:**

?- father(rob,theresa).

Yes

?- \neg father(rob,theresa).

No

?- father(rob, paul).

No

?- \neg father(rob, paul).

Yes

Negative Ziele

- Verwendung in **Regeln**, z.B.
`non_parent(X, Y) :- \+ father(X, Y), \+ mother(X, Y).`
- Prolog untersteht der Annahme, dass die Datenbasis **vollständig** ist: "**Closed World Assumption**" (**CWA**).
 - Nur unter dieser Annahme entspricht \+ "nicht" bzw. "not"; genau genommen steht es für das **Scheitern einer Anfrage!**
 - Daher liefert `non_parent/2` "Yes" mit beliebigen Namen als Argument, die **nicht** in der Datenbasis **vorkommen**.
- **Zu beachten:**
 - \+ liefert keine Werte für die Variablen
 - \+ kann nicht in Fakten oder Regel-Köpfen vorkommen – dies würde eine Redefinition von \+ bewirken

Test auf Gleichheit

- Ein **Geschwister-Prädikat** für das Familien-Beispiel:
(Geschwister seien mittels **derselben Mutter** definiert)

$\text{sibling}(X, Y) :- \text{mother}(M, X), \text{mother}(M, Y).$

- Damit erhalten wir: $X = \text{rob}, Y = \text{rob}; X = \text{bev}, Y = \text{bev}; X = \text{bev}, Y = \text{theresa}; X = \text{theresa}, Y = \text{bev}; X = \text{aaron}, Y = \text{aaron}.$
- Um die Lösungen auszuschließen, in denen X und Y **mit demselben Wert instantiiert** sind, wird ein entsprechender **Test** benötigt.
- Das **Identitätsprädikat** $X == Y$ ist erfolgreich, wenn X und Y schon mit demselben Wert instantiiert sind.

Test auf Gleichheit

- **Neues Geschwister-Prädikat:**

$\text{sibling}(X, Y) :- \text{mother}(M, X), \text{mother}(M, Y), \backslash+ X == Y.$

- Im Unterschied dazu versucht das **Unifikationsprädikat** $X = Y$ ("Gleichheitsprädikat"), seine beiden Argumente erst zu unifizieren; es steht also für den **expliziten Aufruf** der **Unifikationsoperation**.

- Mit $X = a$ bekommt die Variable X den Wert a .
- Sind **beide** Argumente instantiiert, verhalten sich '=' und '==' gleich.

Anonyme Variable

- Das Zeichen '_' steht für eine **unbenannte Variable**, die auf **jeden Ausdruck** passt, aber **nie einen Wert** annimmt.
- Kommen in einer Klausel **mehrere anonyme Variablen** vor, so gelten sie als **verschieden**.
- Anonyme Variablen werden üblicherweise verwendet, wenn es nur um die **Passung, aber nicht um den Wert** an der betreffenden Stelle geht.
- **Im Familien-Beispiel:** Mit `mother(dorothy, _)`. kann man herausfinden, ob dorothy Mutter ist, aber nicht, **wessen Mutter**.

Ausblick: Kapitel 04; Teil 2

- **Lernziele:**

- Wie werden Funktionalitäten, die man aus anderen Programmiersprachen kennt, in Prolog verwirklicht? (Arithmetische Ausdrücke, Rekursion, Listen, Strukturen)

Inhalt

- Prolog-Systeme
- Darstellungsalternativen
- Funktoren
- Funktionen
- Arithmetische Ausdrücke
- Rekursion
- Listen
- Rekursion über Listen
- Beispiel: Baumtraversierung
- Strukturen

Prolog-Systeme

- **Prolog-Systeme** bestehen mindestens aus einem
 - **"Interpreter"**:
Schleife: Ziel einlesen – Ausführen – Resultat ausgeben.
 - Ausführung von Zielen durch einen speziellen automatischen **Theorembeweiser** auf der Basis des **Resolutionsverfahrens**.
 - Datenbasen werden in **Dateien** erstellt und eingelesen; ausnahmsweise auch on-line
(`consult(user). <ctrl>-D`).
 - Spezielle Ein-/Ausgabe-Pseudoprädikate für Benutzerprogramme.
 - Hinzu können kommen
 - **Programmierungsumgebung**: Tracer/Debugger, Editor, ...
 - **Compiler**: (i.d.R.) Übersetzung in den Code einer abstrakten Maschine ("WAM": Warren Abstract Machine).

Darstellungs–Alternativen

- Bei der Wahl der Prädikate ist es – **abhängig vom Zweck (!)** des Programms – oft von Vorteil, "**einfache**" **Konzepte** zu wählen und mit ihnen dann weitere zu definieren;
im Familienbeispiel etwa

```
parent(Parent, Child).  
male(Person).  
female(Person).  
father(X, Y) :- parent(X, Y), male(X).  
mother(X, Y) :- parent(X, Y), female(X).
```

- Es können aber auch **längere Tupel** im Stil der **relationalen Datenbanksysteme** definiert werden ("data record"):

```
person(Name, Gender, Father, Mother).  
male(X) :- person(X, male, _, _).  
father(Father, Child) :- person(Child, _, Father, _).
```

Darstellungs–Alternativen

- Repräsentationsvarianten durch **Strukturierung** ("Substrukturen")

birthday(linda, 27, aug, 1966).

birthday(paula, 2, sep, 1963).

birthday(george, 14, feb, 1965).

birthday(tim, 23, feb, 1966).

- **Alternativ:** Verschachtelte Strukturen

birthday(linda, date(27, aug, 1966)).

birthday(paula, date(2, sep, 1963)).

....

?- birthday(linda, Date).

 Date = date(27, aug, 1966)

?- birthday(Person, date(Day, sep, _)).

 Person = paula,

 Day = 2

- date2year(date(_, _, Year), Year).

?- birthday(linda, Date), date2year(Date, Year).

 Date = date(27, aug, 1966),

 Year = 1966

Funktoren

- **Funktionsterme** werden **nicht ausgewertet** (wie Prozeduren in prozeduralen Programmiersprachen), sondern
- **Funktoren sind Konstruktoren:**
 - Jeder variablenfreie Funktionsterm wird als **Bezeichner** eines Elements aus dem jeweiligen Gegenstandsbereich angesehen;
 - Funktionsterme dienen zur "Konstruktion" neuer Elemente aus gegebenen.
- **Funktionsterme** dienen somit als "**Record**"–**Strukturen**: s.u. (einzige Art von Datenstrukturen in Prolog!)
- Die **Unifikation** spielt eine "doppelte Rolle"
 - **Selektion** durch Auswahl / **Aufnahme** der Argumente von Funktionen
 - **Konstruktion** durch Instantiierung von Variablen mit komplexen Termen
 - Terme mit Variablen stehen für partiell spezifizierte **Datenstrukturen** (Variable als Platzhalter).

Funktionen

- Also: **Funktoren sind Bezeichner** ("Köpfe") von Datenstrukturen, keine Bezeichner für Operationen, die auf den Argumenten ausgeführt werden!
- ABER: **Funktionen** können durch **Relationen** dargestellt werden:
 - Die **Eindeutigkeit** von Funktionswerten ist in Prolog grundsätzlich **nicht gegeben**; vielmehr muss der Programmierer dafür Sorge tragen, dass die **relationale Definition** einer Funktion für jede Eingabe genau eine Ausgabe liefert (Funktionen als **rechtseindeutige Relationen**).
 - Die **Relationen–Syntax** erfordert die Einführung von **Hilfsvariablen** (Ausgabeparameter); **Lesbarkeit wird erschwert**.

Funktionen

- **Beispiel:** Definition der **Additionsfunktion als (rekursive) Relation**

$\text{add}(0, Y, Y).$

$\text{add}(\text{succ}(X), Y, \text{succ}(Z)) \text{ :- } \text{add}(X, Y, Z).$

- Der atomare Term 0 repräsentiert die Null.
- succ ist ein **uninterpretiertes Funktionssymbol**; der Term $\text{succ}(0)$ repräsentiert (für uns!) die 1, kann jedoch nicht zu dem Term 1 **ausgewertet** werden!

Funktionen

- Für **funktionale Berechnung**, d.h. Auswertung von Termen ("Reduktion" von Termen auf andere, z.B. $\text{succ}(0) \Rightarrow 1$), wird eine **Logik mit Gleichheit** benötigt:
 - Erst damit könnte man schließen, ob zwei Terme **dasselbe Objekt** bezeichnen.
 - Allgemeine Beweisprozeduren für Logik mit Gleichheit sind **nicht effizient genug** für die logik-orientierte Programmierung.
 - "Gleichungslogik" und Termersetzungssysteme: s. Kreuzer / Kühling, Kap. 7.
- Im Beispiel: Die Relation $\text{add}/3$ entspricht der zweistelligen Funktion $+$ in einer Logik mit Gleichheit:

$$0 + Y = Y.$$

$$\text{succ}(X) + Y = \text{succ}(X + Y).$$

Funktionen

- **Anfragen** im add-Beispiel ... "Strichlisten als succ-Terme"

?- add(0, succ(0), Result).

Result = succ(0)

Yes

?- add(succ(succ(0)), succ(succ(0)), Result).

Result = succ(succ(succ(succ(0))))

Yes

- Hier ist die Unifikation mit der zweiten Klausel der Definition $\text{add}(\text{succ}(X), Y, \text{succ}(Z))$ **erfolgreich** mit $X = \text{succ}(0)$, $Y = \text{succ}(\text{succ}(0))$, $\text{Result} = \text{succ}(Z)$
- Damit **wird** der Rumpf der zweiten Klausel **zu** $\text{add}(\text{succ}(0), \text{succ}(\text{succ}(0)), Z)$
- Erneute Unifikation mit der zweiten Klausel **ergibt** $X1 = 0$, $Y1 = \text{succ}(\text{succ}(0))$, $Z = \text{succ}(Z1)$

Funktionen

- **Einsetzung** in den Rumpf: $\text{add}(0, \text{succ}(\text{succ}(0)), Z1)$
- Dieses Ziel ist mit der **Einheitsklausel** $\text{add}(0, Y, Y)$ **unifizierbar**:
 $Y2 = \text{succ}(\text{succ}(0)), Z1 = Y2$
- Damit terminiert die Ausführung und **wir erhalten**:
 $\text{Result} = \text{succ}(Z)$
 $= \text{succ}(\text{succ}(Z1))$
 $= \text{succ}(\text{succ}(Y2))$
 $= \text{succ}(\text{succ}(\text{succ}(\text{succ}(0))))$
- Die Rekursion führt **nicht in eine Endlosschleife**, weil ein Parameter (hier: X) in jedem Schritt "**vermindert**" wird:
 $\text{succ}(X) \Rightarrow X$

Arithmetische Ausdrücke

- Arithmetische Ausdrücke: **Auswertbare Terme**

Wann werden sie ausgewertet?

- Rechte Seite von **is**, z.B. $X \text{ is } 3+7$.

(is/2-Prädikat: **arithmetische Funktion**)

- Beide Seiten bei Vergleichen mit $:=$ $\backslash=$ $<$ $>$ $=<$ $>=$
(**arithmetische Relationen**)

- **Beispiele:**

$X = 3+7$. $\text{--> } X = 3+7$ % Unifikation, **keine Auswertung!**

$X \text{ is } 3+7$. $\text{--> } X = 10$

$10 \text{ is } 3+7$. --> Yes

$3+7 \text{ is } 3+7$. --> No

$X \text{ is } Y+2$. ERROR: is/2: Arguments are not sufficiently instantiated

$5+6 < 3*4$. --> Yes

$2+3 := 1+4$. --> Yes

$2+3 \backslash= 1+4$. --> No

Arithmetische Ausdrücke

- Operatoren

$N + N$	Addition
$N - N$	Subtraktion
$N * N$	Multiplikation
N / N	Gleitkomma-division
$I // I$	Ganzzahl-division
$I \text{ rem } I$	Dezimalbruch-Rest
$I \text{ mod } I$	Modulo
$-N$	Vorzeichen-Tausch
$I \gg J$	Bit-shift I um J bits nach rechts
$I \ll J$	Bit-shift I um J bits nach links
$I \wedge I$	Bitweises und
$I \vee I$	Bitweises oder
$\backslash I$	Bitweises Komplement (alle Bits von I umkehren)

I, J: Ganzzahlen; N: Ganzzahlen oder Gleitkommazahlen

Operatoren

- Funktionsterme können in **Präfix**–, **Infix**– und **Postfix**–Notation geschrieben werden.
- Festzulegen: **Vorrang und Assoziativität** der Operatoren

Priorität	Assoziativität	Operator
500	yfx	+ -
400	yfx	* /
200	xfy	**
200	fy	-

- **Benutzerdefinierte Operatoren:**

Die Direktive `op` definiert/modifiziert Operator mit Namen `Atom`
:- `op(Priority, Associativity, Atom)`.

Rekursion

- **Beispiel: Fakultätsfunktion** $n!$, $n \in \mathbb{N}_0$

$$n! = n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$$

Also $0! = 1$ und $n! = n * (n - 1)!$ für alle $n > 0$

In der funktionalen Programmierung:

$\text{fac}(N) := \text{if } N \leq 0 \text{ then } 1 \text{ else } N * \text{fac}(N-1).$

... ausführliche Diskussion in „Algorithmen und Datenstrukturen“

$\text{fac}(0, 1).$

$\text{fac}(N, V) :-$

$N > 0,$

$N1 \text{ is } N-1,$

$\text{fac}(N1, V1),$

$V \text{ is } N * V1.$

Rekursion

- **Ausführung von fac(3, X). mit Trace:**

```
?- trace(fac).  
%   fac/2: [call, redo, exit, fail]  
Yes  
[debug] ?- fac(3, X).  
T Call: (? ) fac(3, _G313)  
T Call: (8) fac(2, _G346)    % zurückgestellte Verarbeitungsschritte  
T Call: (9) fac(1, _G365)    % ... bis  
T Call: (10) fac(0, _G384)   % Abbruchbedingung (Einheitsklausel) erreicht  
T Exit: (10) fac(0, 1)  
T Exit: (9) fac(1, 1)       % Aufarbeitung der zurückgestellten Schritte:  
T Exit: (8) fac(2, 2)       %   Ausführung der Multiplikationen  
T Exit: (? ) fac(3, 6)
```

X = 6

- **Linear rekursiver** Berechnungsprozess: N zurückgestellte Schritte (Kellerspeicher, "Stack").

Rekursion

- Die Berechnung der Fakultätsfunktion ist auch in einem **iterativen** (statt einem linear rekursiven) **Prozess** möglich:
- Einführung einer "Akkumulator"-Variablen (V1), an die in jedem Schritt das bis dahin erreichte Zwischenergebnis (Multiplikationen) gebunden wird.
- Einführung eines Zählers (N1).

```
fac_iter(N, V) :- fac_iter(N, 1, V).           % fac_iter/2

fac_iter(0, V, V).                             % fac_iter/3
fac_iter(N, VO, V) :-
    N > 0,
    V1 is VO*N,
    N1 is N-1,
    fac_iter(N1, V1, V).           % Endrekursion !
```

Rekursion

- **Ausführung von fac_iter(3, X). mit Trace:**

```
?- trace(fac_iter/3).
```

```
%    fac_iter/3: [call, redo, exit, fail]
```

```
Yes
```

```
[debug] ?- fac_iter(3, X).
```

```
T Call: (8) fac_iter(3, 1, _G313)
```

```
T Call: (9) fac_iter(2, 3, _G313)
```

```
T Call: (10) fac_iter(1, 6, _G313)
```

```
T Call: (11) fac_iter(0, 6, _G313)
```

```
T Exit: (11) fac_iter(0, 6, 6)      % Nur noch: Ergebnis(se) durchreichen
```

```
T Exit: (10) fac_iter(1, 6, 6)
```

```
T Exit: (9) fac_iter(2, 3, 6)
```

```
T Exit: (8) fac_iter(3, 1, 6)
```

```
X = 6
```

- **Endrekursion** ("tail recursion"): "Tail recursion optimisation" bzw. "Last call optimisation" – als Schleife ausführbar
- **Iterativer** Berechnungsprozess: **konstanter Speicherbedarf**

Rekursion

- **Beispiel:** Fibonacci–Zahlen

Fibonacci (Leonardo von Pisa, 1170/80 – nach 1240):

- **Wachstum einer Kaninchenpopulation** nach folgender Vorschrift:

- Zu Beginn gibt es **ein Paar** geschlechtsreifer Kaninchen.
- Jedes neugeborene Paar wird im zweiten Lebensmonat geschlechtsreif.
- Jedes geschlechtsreife Paar wirft **pro Monat ein weiteres Paar**.
- Die Tiere befinden sich in einem abgeschlossenen Raum, so dass kein Tier die Population verlassen und keines von außen hinzukommen kann.



Liber abaci

- Das erste Paar erzeugt seinen Nachwuchs bereits im ersten Monat. Jeden Folgemonat kommt eine Anzahl von neugeborenen Paaren hinzu, die **gleich der Anzahl** der Paare ist, die **bereits im vorletzten Monat gelebt haben**, da genau diese geschlechtsreif sind und sich nun vermehren...

Rekursion

- **Rekursionsgleichungen**

$$F_0 = 1$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

ergibt die **Fibonacci-Folge**: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

fib(0, 1).

fib(1, 1).

fib(N, Fib) :-

 N > 1,

 N1 is N-1,

 N2 is N-2,

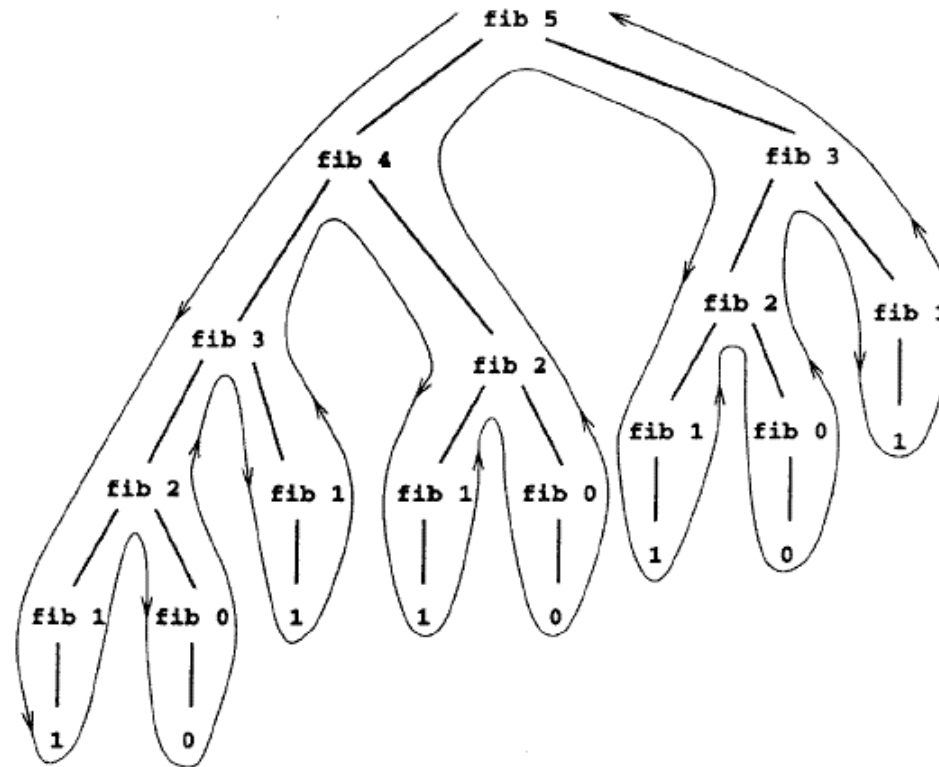
 fib(N1, Fib1),

 fib(N2, Fib2),

 Fib is Fib1 + Fib2.

Rekursion

- Statt eines Trace: Baum der Berechnungsschritte



- Anmerkung: hier abweichende Definition: $\text{fib}(0,0)$ statt $\text{fib}(0,1)$!
© Abelson/Sussman

Rekursion

- Einführung von zwei **Akkumulatorvariablen**: F0, F1
- Simultane Bindung von F0 an den Wert von F0 + F1 und F1 an F0 unter Verwendung einer Hilfsvariablen F2 (denn: X is X+1 ist falsch!)

```
fib_iter(X, Result) :- fib_iter(X, 1, 1, Result).
```

```
fib_iter(0, F0, F1, F0).
```

```
fib_iter(1, F0, F1, F1).
```

```
fib_iter(N, F0, F1, Result) :-
```

```
    N > 1,
```

```
    N1 is N-1,
```

```
    F2 is F0 + F1,
```

```
    fib_iter(N1, F1, F2, Result). %Endrekursion!
```

Rekursion

- **Ausführung von fib_iter(5, X). mit Trace:**

```
[debug] ?- fib_iter(5, X).  
  T Call: (7) fib_iter(5, _G313)  
  T Call: (8) fib_iter(5, 1, 1, _G313)  
  T Call: (9) fib_iter(4, 1, 2, _G313)  
  T Call: (10) fib_iter(3, 2, 3, _G313)  
  T Call: (11) fib_iter(2, 3, 5, _G313)  
  T Call: (12) fib_iter(1, 5, 8, _G313)  
  T Exit: (12) fib_iter(1, 5, 8, 8)  
  T Exit: (11) fib_iter(2, 3, 5, 8)  
  T Exit: (10) fib_iter(3, 2, 3, 8)  
  T Exit: (9) fib_iter(4, 1, 2, 8)  
  T Exit: (8) fib_iter(5, 1, 1, 8)  
  T Exit: (7) fib_iter(5, 8)  
X = 8
```

Listen

- Der **Listenkonstruktor** in Prolog ist der Punkt '.'
- Der Term $.(X, Y)$ repräsentiert die aus X und Y gebildete **Liste**;
- In Prolog gibt es hierfür die besondere Notation $[X|Y] \equiv .(X, Y)$
 - $[]$: die leere Liste
 - $.(X, []) \equiv [X | []] \equiv [X]$
 - Beispiel für explizite Verwendung des Konstruktors:
 $.(a, .(b, [])) \equiv [a,b]$
- $[Head | Tail]$: Liste bestehend aus "Kopf" **Head** und Restliste **Tail**
 - Beispiel: $[a, b | X]$ repräsentiert die Liste(n) mit der Folge von "Elementen" a,b,\dots , d.h. mit der **Liste $[a, b]$ als Kopf**.
- Prolog-Programm, das die induktive Definition von Listen wiedergibt:
`list([]).`
`list([Head | Tail]) :- element(Head), list([Tail]).`

Listen

- **Beispiele:**

- $[a]$: einelementige Liste
- $[a, b, c, d]$: Liste mit vier Elementen

- Sei

?- $X = [1, 2, 3 \mid [4, 5 \mid [6, 7]]]$.

$X = [1, 2, 3, 4, 5, 6, 7]$

?- $Y = [1, [a, b], 3 \mid [4, 5 \mid [6, 7]]]$.

$Y = [1, [a, b], 3, 4, 5, 6, 7]$

- Der **Rest** (Tail) einer Liste **ist immer eine Liste**,
der Kopf (Head) einer Liste ist ein "Element":

$[a \mid []] = [a]$

$[a \mid [b, c, d]] = [a, b, c, d]$

$[a, b, c \mid [d, e, f]] = [a, b, c, d, e, f]$

Listen

- Listen können **konstruiert** und **zerlegt** werden durch **Unifikation**:

Unifiziere	mit	Ergebnis
$[a, b, c]$	X	$X = [a, b, c]$
$[X, Y, Z]$	$[a, b, c]$	$X = a, Y = b, Z = c$
$[X, b, Z]$	$[a, Y, c]$	$X = a, Y = b, Z = c$
$[[a, b], c]$	$[X, Y]$	$X = [a, b], Y = c$
$[a(b), c(X)]$	$[Z, c(a)]$	$X = a, Z = a(b)$

- Die leere Liste kann nicht in Kopf und Rest zerlegt werden.

Listen

- Der Term $[X|Y]$ unifiziert mit jeder nichtleeren Liste, wobei X mit dem Kopf und Y mit dem Rest instantiiert wird.

Unifiziere	mit	Ergebnis
$[X Y]$	$[a, b, c, d]$	$X = a, Y = [b, c, d]$
$[X Y]$	$[a]$	$X = a, Y = []$
$[X, Y Z]$	$[a, b, c]$	$X = a, Y = b, Z = [c]$
$[X, Y Z]$	$[a, b, c, d]$	$X = a, Y = b, Z = [c, d]$
$[X, Y, Z A]$	$[a, b, c]$	$X = a, Y = b, Z = c, A = []$
$[X, Y, Z A]$	$[a, b]$	scheitert
$[X, Y, a]$	$[Z, b, Z]$	$X = Z = a, Y = b$
$[X, Y Z]$	$[a W]$	$X = a, W = [Y Z]$

Rekursion über Listen

- **Länge**

```
list_length([], 0).  
list_length([_ | Rest], Len) :-  
    list_length(Rest, Len0),  
    Len is Len0 + 1.
```

- **Summe einer Liste von Zahlen**

```
list_sum([], 0).  
list_sum([X | Rest], Sum) :-  
    list_sum(Rest, Sum0),  
    Sum is Sum0 + X.
```

- **"Mapping" von Listen**, z.B. elementweises Quadrieren

```
map_sqr([], []).  
map_sqr([X | XTail], [Y | YTail]) :-  
    Y is X*X,  
    map_sqr(XTail, YTail).
```

```
?- map_sqr([1, 2, 3, 4], Y).  
Y = [1, 4, 9, 16]
```

Rekursion über Listen

- **Elementrelation** $\text{mem}(X,Y)$ (member : i.d.R. Systemprädikat)

- Y leer: Scheitern
- X ist erstes Element von Y : Erfolg
- sonst : X ist Element von Y , wenn X Element des Rests von Y ist.

$\text{mem}(X, [X | Xs]).$

$\text{mem}(X, [Y | Ys]) :- \text{mem}(X, Ys).$

?- $\text{mem}(2, [1, 2, 3]).$ --> Yes

?- $\text{mem}(X, [1, 2, 3]).$ --> $X = 1 ; X = 2 ; X = 3$

?- $\text{mem}(2, X).$ --> $X = [2 | _G822];$

$X = [_G821, 2 | _G824];$

$X = [_G821, _G823, 2 | _G826];$

$X = [_G821, _G823, _G825, 2 | _G828];$

Rekursion über Listen

- **Verkettung:**

append([], Ys, Ys).

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

- **Beispiele:**

?- append([1,2], [a,b], [1,2,a,b]). --> Yes

?- append([1,2], [a], [1,2,a,b]). --> No

?- append([1,2,3], [a,b,c], X). --> X = [1,2,3,a,b,c]

?- append([1,2,3], X, [1,2,3,a,b]). --> X = [a,b]

?- append([1,2], [a|X], [1,2,a,b]). --> X = [b]

?- append(X, [a,b], [1,2,3,a,b]). --> X = [1,2,3]

?- append(X, Y, [1,2,3,a,b,c]). --> X = [], Y = [1,2,3,a,b,c];

X = [1], Y = [2,3,a,b,c];

X = [1,2], Y = [3,a,b,c];

X = [1,2,3], Y = [a,b,c];

X = [1,2,3,a], Y = [b,c];

X = [1,2,3,a,b], Y = [c];

X = [1,2,3,a,b,c], Y = []

Rekursion über Listen

- Aufgrund der **deklarativen Definition** können Variablen problemlos an **allen Argumentpositionen** eingesetzt werden:

- ?- append([1,2,3], X, Y). --> X = _G768, Y = [1,2,3|X]

- ?- append(X, [1,2,3], Y).

-->

X = [], Y = [1,2,3] ;

X = [_G097], Y = [_G097,1,2,3] ;

X = [_G097,_G098], Y = [_G097,_G098,1,2,3] ;

X = [_G097,_G098,_G099], Y = [_G097,_G098,_G099,1,2,3] ;

...

- ?- append(X, Y, Z).

-->

X = [], Y = Z = _G753 ;

X = [_G097], Y = _G753, Z = [_G097|Y] ;

X = [_G097,_G098], Y = _G753, Z = [_G097,_G098|Y] ;

X = [_G097,_G098,_G099], Y = _G753, Z = [_G097,_G098,_G099|Y] ;

Rekursion über Listen

- **Listen umkehren:**

Klassisches rekursives, aber sehr ineffizientes Verfahren

- Zerlege die Ausgangsliste in Kopf und Rest;
- Drehe rekursiv den Rest der Ausgangsliste um;
- Bilde eine Liste, deren Kopf der Kopf der Ausgangsliste ist;
- Verkette der umgedrehten Rest der Ausgangsliste mit der im vorhergehenden Schritt erzeugten Liste.

```
reverse([], []).
```

```
reverse([Head | Tail], Result) :-
```

```
    reverse(Tail, ReversedTail),
```

```
    append(ReversedTail, [Head], Result).
```

- **Ineffizienz:** Bei einer achtelementigen Liste werden 45 Schritte benötigt: 9 Aufrufe von `reverse`, gefolgt von 36 Aufrufen von `append`.
- **Übung:** Effiziente Lösung (Akkumulatorvariable)

Rekursion über Listen

- Einebenen:

```
flatten([], []).
```

```
flatten(X, [X]) :-  
    atomic(X),  
    X \== [].
```

```
flatten([H|Tail], List) :-  
    flatten(H, FlatH),  
    flatten(Tail, FlatTail),  
    append(FlatH, FlatTail, List).
```

```
| ?- flatten([1, [a,b], 3, [4,5|[6,7]]], Flat).  
Flat = [1,a,b,3,4,5,6,7]
```

Beispiel: binäre Baumtraversierung

```
operate(X) :- write(X),write(' ').
```

```
prefix([]).
```

```
prefix([Value,Left,Right]) :-  
    operate(Value),  
    prefix(Left),  
    prefix(Right).
```

```
infix([]).
```

```
infix([Value,Left,Right]) :-  
    infix(Left),  
    operate(Value),  
    infix(Right).
```

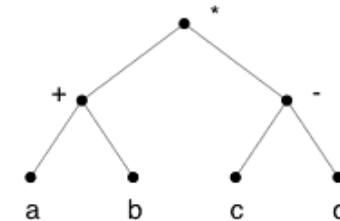
```
postfix([]).
```

```
postfix([Value,Left,Right]) :-  
    postfix(Left),  
    postfix(Right),  
    operate(Value).
```

```
traverse(prefix,Tree) :- prefix(Tree).
```

```
traverse(infix,Tree) :- infix(Tree).
```

```
traverse(postfix,Tree) :- postfix(Tree).
```



```
tree([*, [+,[a,[],[]],[b,[],[]]],[-, [c,[],[]],[d,[],[]]]).
```

```
| ?- tree(X),prefix(X).
```

```
* + a b - c d
```

```
| ?- tree(X),infix(X).
```

```
a + b * c - d
```

```
| ?- tree(X),postfix(X).
```

```
a b + c d - *
```

Strukturen

- Prolog-Terme, bestehend aus einem Funktor, gefolgt von einem oder mehr Termen als Argumenten, heißen auch **Strukturen**:
 - Der Funktor muss ein Atom sein,
 - Die Argumente können Terme beliebiger Art sein (vgl. oben).

- **Beispiel:**

```
person(name(vorname('Kurt'), nachname('Gödel')),
       geschlecht(m),
       geburtsdatum(tag(28), monat(4), jahr(1906)),
       geburtsort(stadt('Brünn'),
                  staat('Österreich-Ungarn'))).
```

- **Strukturen** haben Vieles **mit Listen gemeinsam**; sie werden aber anders (**kompakter**) gespeichert.

Strukturen

- Strukturen können mit dem Prädikat '=..' ("univ") in inhaltsäquivalente Listen konvertiert werden:

?- a(b, c, d) =.. X.

X = [a, b, c, d]

Yes

?- X =.. [w, x, y].

X = w(x, y)

Yes

?- alpha =.. X.

X = [alpha]

Yes

- Im Unterschied zu Strukturen sind Listen **in Kopf und Rest zerlegbar**;
- Strukturen können mit anderen Strukturen mit demselben **Funktor** und derselben **Stelligkeit unifiziert** werden.

Strukturen

- Zur **Dekomposition** von Strukturen gibt es die **System-Prädikate**:

- **functor**(S, F, A) : unifiziert F und A mit dem Funktor und der Stelligkeit der Struktur S

?- functor(a(b, c), X, Y).

X = a

Y = 2

- **arg**(N, S, X) unifiziert X mit dem N -ten Argument der Struktur S

?- arg(2, a(b, c, d), What).

What = c

Strukturen

- Eine wichtige Eigenschaft von Prolog ist die Möglichkeit, **partielle Datenstrukturen** zu verarbeiten.
 - Variablen sind an **jeder** Stelle einer Struktur zulässig; unterbestimmte Werte werden durch ungebundene Variablen repräsentiert ("**unbound Variable**" *ist kein Fehler*!).
 - Diese Stellen werden ggf. später durch Unifikationen instantiiert.
- **Beispiel:**

```
halb1(X) :- X = struct(3, _).
```

```
halb2(X) :- X = struct(_, 5).
```

```
?- halb1(A).
```

```
A = struct(3, _G484)
```

```
?- halb1(A), halb2(A).
```

```
A = struct(3, 5)
```