2 Die deklarative Programmiersprache PROLOG

PROLOG ermöglicht

- deklarative Programmierung,
- kompakte Programme und
- agile Softwareentwicklung, rapid Prototyping.

Die zugrundeliegende Auswertungsstrategie der *SLDNF–Resolution* benutzt

- Tiefensuche,
- Backtracking und
- Unifikation.

Die Programmiersprache PROLOG wurde Anfang der 1970er Jahre maßgeblich von den Informatikern Robert Kowalski und Alain Colmerauer entwickelt.

Top–Down–Auswertung von PROLOG: SLDNF–Resolution

- Genau wie konventionelle Programmiersprachen wird auch PROLOG top–down ausgewertet: der Aufruf eines Prädikats sucht nach einer anwendbaren Regel mit dem Prädikat im Regelkopf und ruft dann nacheinander die Statements des Regelrumpfes auf.
- Anders als in konventionellen Programmiersprachen kann es viele solche Regeln geben; diese werden dann nacheinander – vergleichbar mit den verschiedenen Optionen eines Case–Statements – benutzt.
  Die Auswertung eines Aufrufs einer Regel kann fehlschlagen; dann wird die nächste anwendbare Regel benutzt (Backtracking). Dies erfolgt bis schließlich die komplette Berechnung erfolgreich abgeschlossen ist.
- Da die Argumente einer Regel auch lediglich partiell instantiiert sein können, erfolgt die Paramenterübergabe mittels Unifikation; dies erweitert die Standardmethode der Paramenterübergabe geeignet.

- Ein negierter Aufruf ist erfolgreich, falls der entsprechende positive Aufruf fehlschlägt (Negation–as–Finite–Failure).
- Mittels Backtracking ist es möglich die Liste aller Antworten zu einem gegebenen Aufruf zu berechnen. Dies entspricht der Anfrageauswertung in relationalen Datenbanken mit SQL.

In praktischen PROLOG–Systemen gibt es eine ganze Reihe von vordefinierten Built–in– und auch Meta–Prädikaten (d.h. Prädikaten, deren Argumente selbst wieder Prädikate sein können).

Außerdem gibt es Seiteneffekte – hauptsächlich zum I/O und zum Zugriff auf die interne Faktendatenbank (assert, retract).

Datenstrukturen, Operationen und Kontrollstrukturen

- Die Beschränkung auf wenige Basisdatentypen und einen einzigen komplexen Datentyp, nämlich die Strukturen (Terme), der generisch ist und alle anderen Datentypen subsumiert, standardisiert die Datenstrukturen ganz entscheident.
- Es gibt keine expliziten Typdeklarationen.
- Es gibt eine große Kollektion von generischen Operationen, die auf alle Strukturen (Terme) anwendbar sind – und somit auf alle Datentypen.
- Häufig werden Meta–Prädikate verwendet.
- Auch Kontrollstrukturen basieren auf Meta–Prädikaten. Zusätzlich zu den Standardkontrollstrukturen, wie Verzweigungen (if–then–else), Schleifen (for, while), und Rekursion, können benutzerdefinierte Kontrollstrukturen in Form von Meta–Prädikaten gebildet werden.

Aspekte des Software Engineering

PROLOG unterstützt Abstraktion und kompakten Code, und es stimuliert damit Refaktorisierungen:

- Der generische Typ der Terme mit seinen generischen Operationen unterstützt Abstraktion und die Wiederverwendung von Code.
- Benutzerdefinierte Kontrollstrukturen erlauben weitere Abstraktion.
- Unifikation, implizites Backtracking und der Verzicht auf explizite Typedeklarationen führen zu sehr kompaktem Code und unterstützen Rapid Prototyping.
- Deklarativität macht den Code lesbarer und somit besser erweiterbar.

Der Wechsel von konventionellen Programmiersprachen auf das Paradigma der Logikprogrammierung ist schwierig und erfordert meist sehr viel Training.

2.1 Grundlegende Strukturen

PROLOG–Atom:

- Folge von Buchstaben, Ziffern und dem Unterstrich "_", beginnend mit einem Kleinbuchstaben,
  z.B. `book, george, a, schalke_04`
- Folge von Sonderzeichen
  wie "`+, -, *, /, <, >, =, :, ., &, _, ~`",
  z.B. `<--->, ======>, ..., .:.`
- Folge von Zeichen in Hochkommatas,
  z.B. '`George`', '`FC Bayern`', '`Schalke 04`'

Zahl (Number):

- ganze Zahl (Integer), z.B. 520, -1
- reele Zahl (Real), z.B. 3.14, 2.3e+8

Konstante: PROLOG–Atom | Number

z.B. book, 0, 123

Variable:

Folge von Buchstaben, Ziffern und dem Unterstrich "_",
beginnend mit einem Großbuchstaben oder mit "_",
z.B. X, Y, Book, _, _A, _a

Funktor (Funktions– bzw. Prädikatensymbol):

PROLOG–Atom

z.B. +, –, $\leq$, ancestor, true

Struktur: Konstante | Variable | Funktor(Struktur, . . . , Struktur)

z.B. +(1,7), ancestor('William','George')

Listen sind spezielle Strukturen zum Funktor ".": [a,b,c,d] = .(a, [b,c,d]).

Die Präfix–Notation +(1,7) ist äquivalent zur Infix–Notation 1+7.

Manche Folgen von Sonderzeichen haben eine vordefinierte Bedeutung in PROLOG. Beispiele:

- “`:-`” ist der Regeloperator, der für einen Folgepfeil “$\leftarrow$” nach links steht.
- “`-->`” wird in Definite Clause Grammars verwendet.

Bemerkung zu Atomen:

- PROLOG–Strukturen der Form $p(t_1, \ldots, t_n)$ mit einem Prädikatensymbol p und n Argumenten ($n \in \mathbb{N}_0$) heißen in der Logik bzw. in DATALOG Atome.
- PROLOG–Atome sind dagegen spezielle Formen von Konstanten oder Funktoren.

Die Funktoren “,” bzw. “;” werden in Regelrümpfen als Junktoren für Konjunktion bzw. Disjunktion benutzt

2.1.1 Klauseln

Eine PROLOG–Klausel (Regel, Fakt, Goal) ist von der Form

Kopf :– Rumpf.

“`:-`” spricht man dabei als “falls” aus. Falls der Rumpf leer ist (Fakt), so kann man “`:-`” weglassen und kurz “Kopf” schreiben.

Der Regelrumpf kann folgende Junktoren enthalten:

- Konjunktion “,”,
- Disjunktion “;”, und
- Negation “`\+`”, “`not`”.

Eine negierte Formel kann man als “`\+ F`” oder als “`not(F)`” schreiben.

PROLOG–Klauseln sind spezielle Strukturen.

Gegeben seien zwei Strukturen A und B.

- *Regel:* A `:-` B, Infix–Notation für `:-`(A, B)
  A heißt *Kopf*, und B heißt *Rumpf*, `:-` steht für eine Implikation $\leftarrow$.
  Meist ist der Rumpf B eine Konjunktion $B_1, \ldots, B_n$ (Funktor: ",")
  es können aber auch Disjunktion ";" und Negation "\+", "not" auftreten
- *Fakt:* A
  Regel ohne Rumpf
- *Goal:* `:-` B oder auch `?-` B
  Regel ohne Kopf, Kurzschreibweise für `:-` (B)

Der Operator `:-` ist sowohl binär als auch unär

Beispiel (PROLOG–Klauseln)

- Prädikatensymbole: `likes`, `woman`, `nice`, `loves`
- Regeln:

```
likes(george, Y) :-
   woman(Y), likes(Y, books).
likes(mary, Y) :-
   nice(Y); loves(Y, cats).
```

- Fakten:

```
woman(mary).
likes(mary, books).
loves(george, cats).
```

- Goal:

```
?- likes(george, Y).
```

Das Prädikat `likes` besteht aus 2 Regeln und einem Fakt. Diese decken unterschiedliche Fälle ab:

- Die erste Regel besagt, daß George alle Frauen mag, die Bücher mögen.
- Die zweite Regel besagt, daß Mary alles, was schön (`nice`) ist, und alle Lebewesen, die Katzen lieben, mag.
  Ein Fakt besagt außerdem, daß Mary Bücher mag.

In der prozeduralen Programmierung könnte man eine Fallunterscheidung in einem `case`–Statement ausdrücken.

Die `likes`–Regel zu Mary mit dem disjunktiven Rumpf könnte man in zwei separate Regeln zerlegen. Die Bündelung der Information zu Mary – zu der man eventuell auch noch das Fakt hinzunehmen könnte – erscheint aber softwaretechnisch auch sehr sinnvoll. Da uns momentan keine Information zu `nice` vorliegt, ist der erste Teil der Regel inaktiv.

Die Bündelung liefert folgende Regel mit einem disjunktiven Regelrumpf:

```
likes(mary, Y) :-
   ( nice(Y)
   ; loves(Y, cats)
   ; Y = books ).
```

Diese ist äquivalent zu drei einzelnen Regeln:

```
likes(mary, Y) :-
   nice(Y).
likes(mary, Y) :-
   loves(Y, cats).
likes(mary, books).
```

Die Reihenfolge der Regeln und Fakten zum selben Prädikatensymbol ist – wie in klassischen Programmiersprachen auch – wichtig.

DSL–Notation

Wenn man geeignete Operatoren (Infix, Präfix, Postfix) definiert, dann kann man die PROLOG–Regeln etwas natürlicher notieren:

```
X has_ancestor Z :-
   X has_ancestor Y and
   Y has_parent Z.
george likes Y :-
   Y is_a_woman and Y likes books.
mary likes Y :-
   Y is_nice or Y loves cats.
mary is_a_woman.
mary likes books.
george loves cats.
```

Man könnte `A :- B` sogar als `B implies A` schreiben.

Zur Operationalisierung der obigen Notation muß kann man die Junktoren `and` und `or` wie folgt definieren:

```
X and Y :-
   X, Y.
X or Y :-
   ( X
   ; Y ).
```

Man führt die beiden neuen Junktoren also auf die bekannten Junktoren zurück.

Die spezielle Schreibweise bei der Disjunktion "`;`" wird verwendet, um diese klar von der viel häufiger verwendeten Konjunktion abzuheben; damit erkennt man sie auch bei flüchtigem Hinsehen sofort.

Die Menge aller Regeln und Fakten für ein Prädikatensymbol bezeichnet man als die *Definition* des Prädikatensymbols oder als das Prädikat:

```
likes(george, Y) :-
   woman(Y), likes(Y, books).
likes(mary, Y) :-
   nice(Y); loves(Y, cats).
likes(mary, books).
```

Sie entspricht einer *Methode* bei prozeduralen oder objekt–orientierten Programmiersprachen.

Die einzelnen Regeln decken meist unterschiedliche Fälle ab.

Ableitung: Top–Down, Backward Chaining

- Ein PROLOG–Interpreter versucht ein Goal durch Inferenz zu beweisen.
  Dies erfolgt Top–Down vom Goal ausgehend durch DFS–Durchlauf des *SLD–Baumes*.
- Es wird immer zunächst nur eine Antwort berechnet.
  Durch die Eingabe von “;” kann man Backtracking anstoßen um weitere Antworten zu finden.
- Wendet man die berechneten Antworten (Substitutionen) auf das Anfrage–Atom an, so erhält man die abgeleiteten Atome.
- Bei der SLD–Ableitung werden die Fakten und Regeln in der Reihenfolge ihres Auftretens im Programm benutzt.
  Es ist meist von Vorteil, die Fakten nach vorne zu stellen, gefolgt von den nicht–rekursiven Regeln.

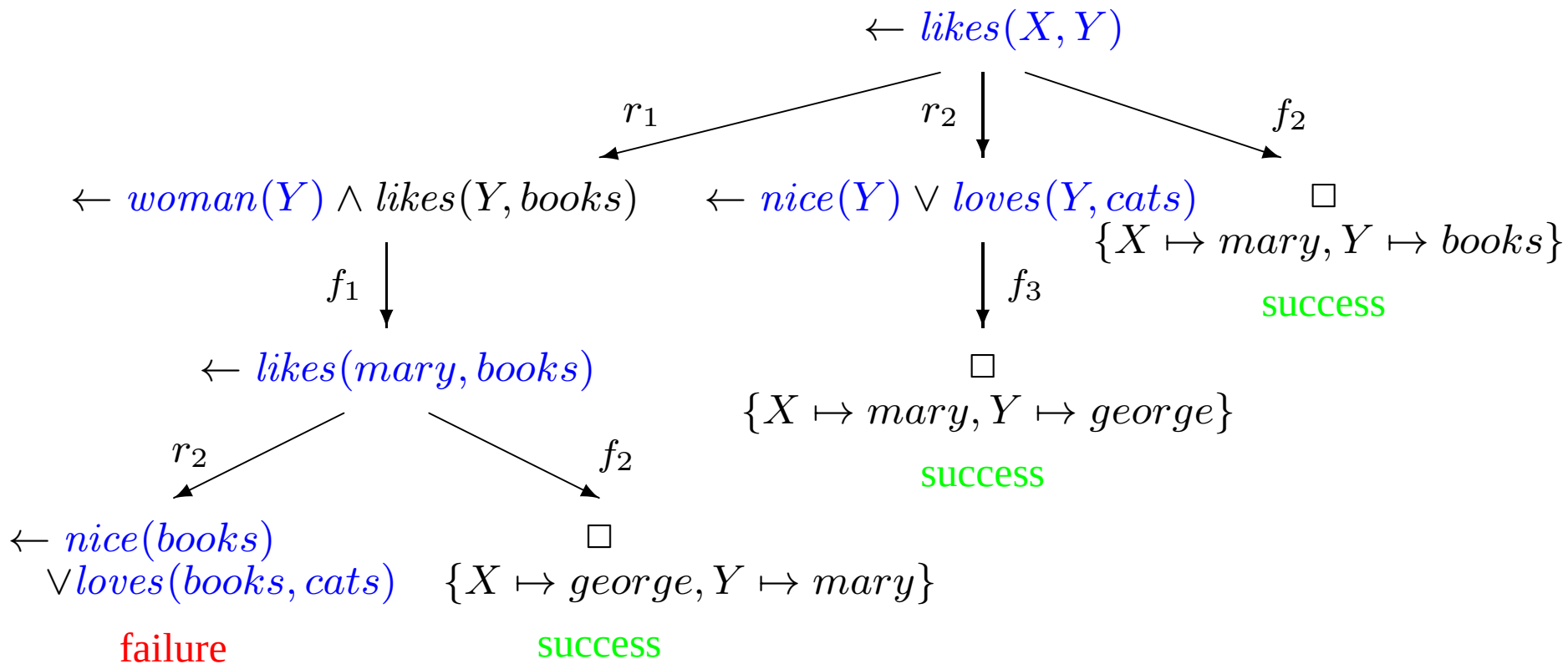
Beispiel (PROLOG)

```
?- likes(george, Y).
Y = mary
?- likes(X, Y).
X = george, Y = mary ;
X = mary, Y = george ;
X = mary, Y = books
```

Die abgeleiteten Atome sind also $likes(george, mary)$, $likes(mary, george)$ und $likes(mary, books)$.

Bei der zweiten Anfrage wurde zwei Mal Backtracking durch Eingabe von ";" hinter den berechneten Antworten angestoßen.

SLD–Baum



Der SLD–Baum zu Anfrage $\leftarrow likes(X, Y)$ hat 3 erfolgreiche Äste.

- Die Knoten eines SLD–Baumes sind mit Goals markiert, die Wurzel mit der Anfrage.
- Für jeden Knoten werden alle möglichen Regeln (bzw. Fakten) betrachtet, deren Regelkopf mit dem ersten Atom des Goals mittels einer geeigneten Substitution unifiziert werden kann. Dann wird das erste Atom des Goals durch den angepaßten Regelrumpf ersetzt. Bei disjunktiven Goals (“;”) können jeweils die ersten Atome der Disjunktionsglieder selektiert werden.
- Wenn man in einem Blatt des SLD–Baumes das leere Goal □ erhält, so war der Ableitungsast erfolgreich (success), und es wird die berechnete Substitution zurückgeliefert; andernfalls war die Ableitung erfolglos (failure).
- Später werden wir sehen, daß SLD–Bäume auch unendliche Äste haben können.

Anfragen in PROLOG

- Die Auswertung von PROLOG–Regeln erfolgt *Top–Down* mittels der Methode der SLDNF–Resolution: d.h., aus einer Anfrage an das Kopfatom einer Regel werden Unteranfragen an die Rumpfatome erzeugt.
- Dies entspricht dem Vorgehen bei der Abarbeitung von prozeduralen Programmiersprachen.
- Falls ein Rumpfatom mehrere Antworten liefert, so wird zuerst mit einer solchen Antwort weitergerechnet (Tiefensuche); falls die Berechnung später fehlschlägt, so kann mit der nächsten Antwort fortgefahren werden (*Backtracking*).
- In der Praxis sind viele PROLOG–Prädikate aber funktional; d.h., aus einer Eingabe wird genau eine Ausgabe erzeugt.

Aus der Anfrage

$$\leftarrow likes(X, Y)$$

wird mittels der Ersetzung $X \mapsto george$ und der Regel

$$r_1 = likes(george, Y) \leftarrow woman(Y) \wedge likes(Y, books)$$

eine Folge von Unteranfragen erzeugt:

$$\leftarrow woman(Y) \wedge likes(Y, books)$$

Aufgrund des Fakts f_1 wird die erste Unteranfrage mit $Y \mapsto mary$ beantwortet. Der neue Wert für Y wird in der zweiten Unteranfrage sofort berücksichtigt, so daß wir also

$$\leftarrow likes(mary, books)$$

zu beantworten haben. Aufgrund des Fakts f_2 ist die Antwort dafür ja.

Kontrollprimitive, welche über die PL1 hinausgehen:

- `assert` und `retract` verändern die interne Datenbank von PROLOG; `clause` sucht in dieser.
- `call` ruft Goals auf, die vorher zusammengebaut werden können.
- Der Cut "`!`" friert die bisherige Variablenbelegung in der aktuellen Regel ein. Er schneidet damit Äste des SLD–Baumes ab, so daß sie nicht durchsucht werden.
- `fail` ist ein Goal, das immer fehl schlägt.
- `true` ist ein Goal, das immer erfolgreich ist.
- Meta–Prädikate wir `findall` und `maplist` realisieren komplexe Kontrollstrukturen.
- ...

Anfrage mit einem Meta–Prädikat

Mit dem Meta–Prädikat `findall/3` kann man alle Antworten auf eine Anfrage bestimmen:

```
?- findall( [X, Y],
       likes(X, Y),
       Pairs ).
Pairs = [[george,mary], [mary,george], [mary,books]].
```

Das DDK erlaubt sogar folgende äquivalente Mengenschreibweise:

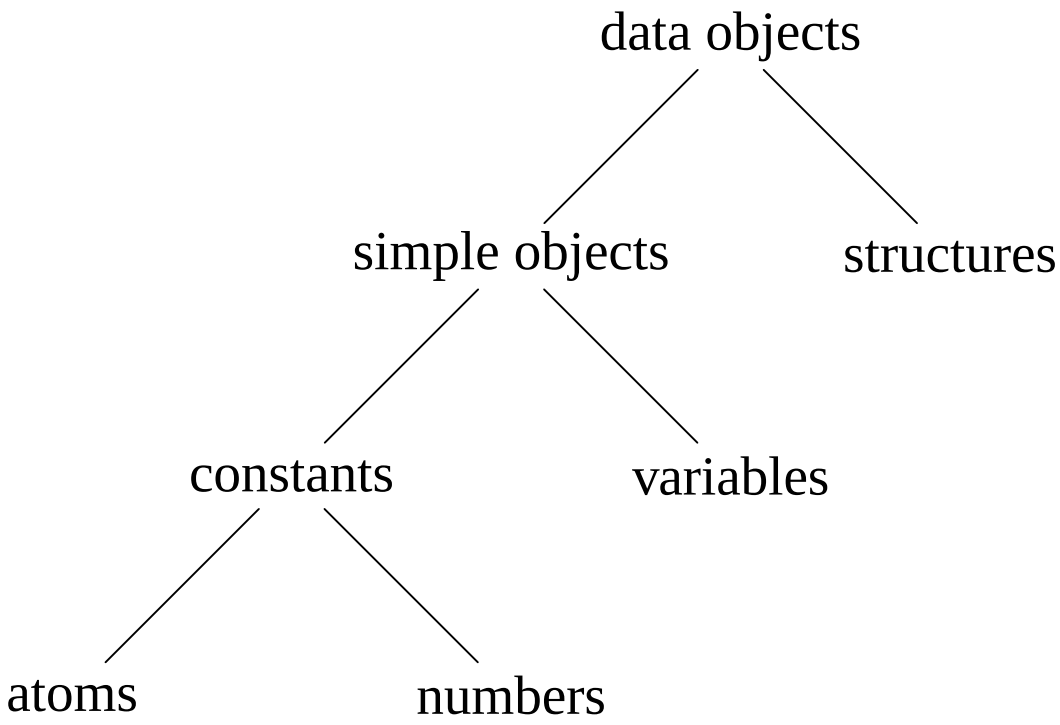
```
?- Pairs <= { [X, Y] | likes(X, Y) }.
```

`Pairs` wird als die Liste aller Paare `[X, Y]` bestimmt, für die `likes(X, Y)` gilt.

Literatur

- W.F. Clocksin, C.S. Mellish:
  Programming in PROLOG,
  Springer, 5th Edition, 2003.
- I. Bratko:
  PROLOG – *Programming for Artificial Intelligence*,
  Addison–Wesley, 4th Edition, 2011.
- J.W. Lloyd:
  Foundations of Logic Programming,
  Springer, 2nd Edition, 1987.

2.1.2 Datentypen



Im folgenden werden einige PROLOG–Datentypen mit Operationen vorgestellt.

Arithmetische Terme als Beispiele für Strukturen

- `log(N)` ist eine Struktur (ein Term) mit dem unären Funktionssymbol `log` und einem Argument `N`; `N` ist ein Variablensymbol.
- Arithmetische Terme werden oft in *Infix–Notation* geschrieben.
  Der komplexe arithmetische Term $3 \cdot X^2 + 2 \cdot X + 1$ kann in PROLOG in Infix–Notation geschrieben werden als

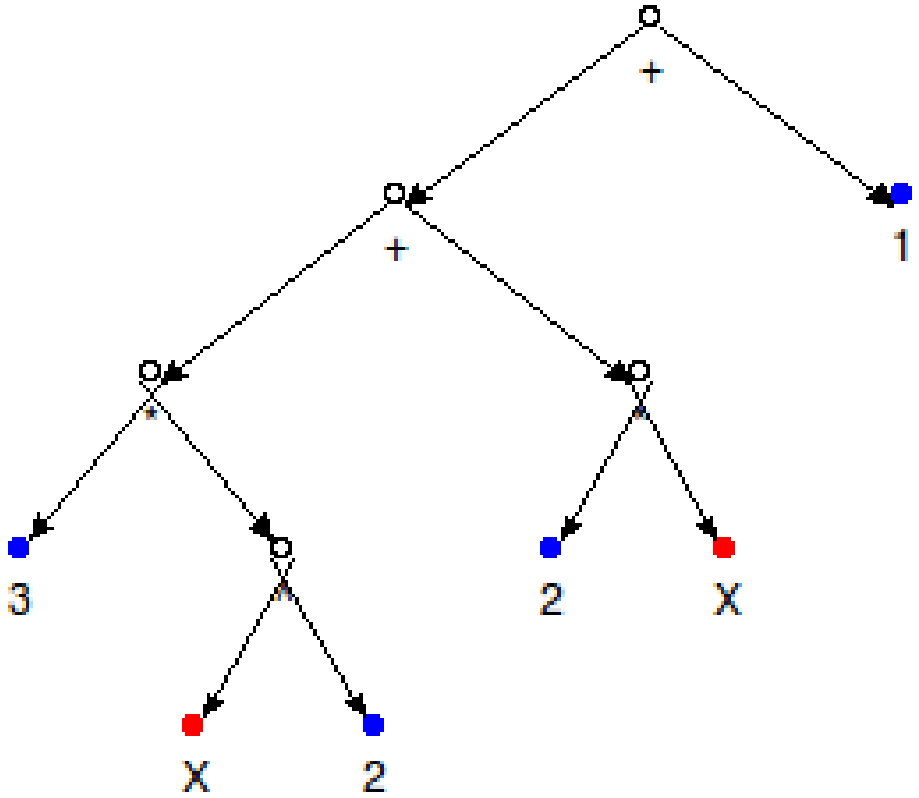
  ```
  3*X^2 + 2*X + 1.
  ```

  Er hat das oberste, binäre Funktionssymbol `+`, und er sähe in *Präfix–Notation* wie folgt aus:

  ```
  +(+(*(3,^(X,2)),2*X),1).
  ```

  Dabei sind `1`, `2` und `3` Zahlen (numbers), und `X` ist ein Variablensymbol.
  `*` und `^` sind weitere, eingebettete, binäre Funktionssymbole.

- Der zu $3 \cdot X^2 + 2 \cdot X + 1$ gehörige *Operatorbaum* ist folgender:



Die Klammerung $(3 \cdot X^2 + 2 \cdot X) + 1$ wird von PROLOG vorgenommen.

- Der komplexe arithmetische Term

  $$\frac{1}{\sqrt{2 \cdot \pi \cdot \sigma^2}} \cdot e^{-\frac{(X-\mu)^2}{2 \cdot \sigma^2}}$$

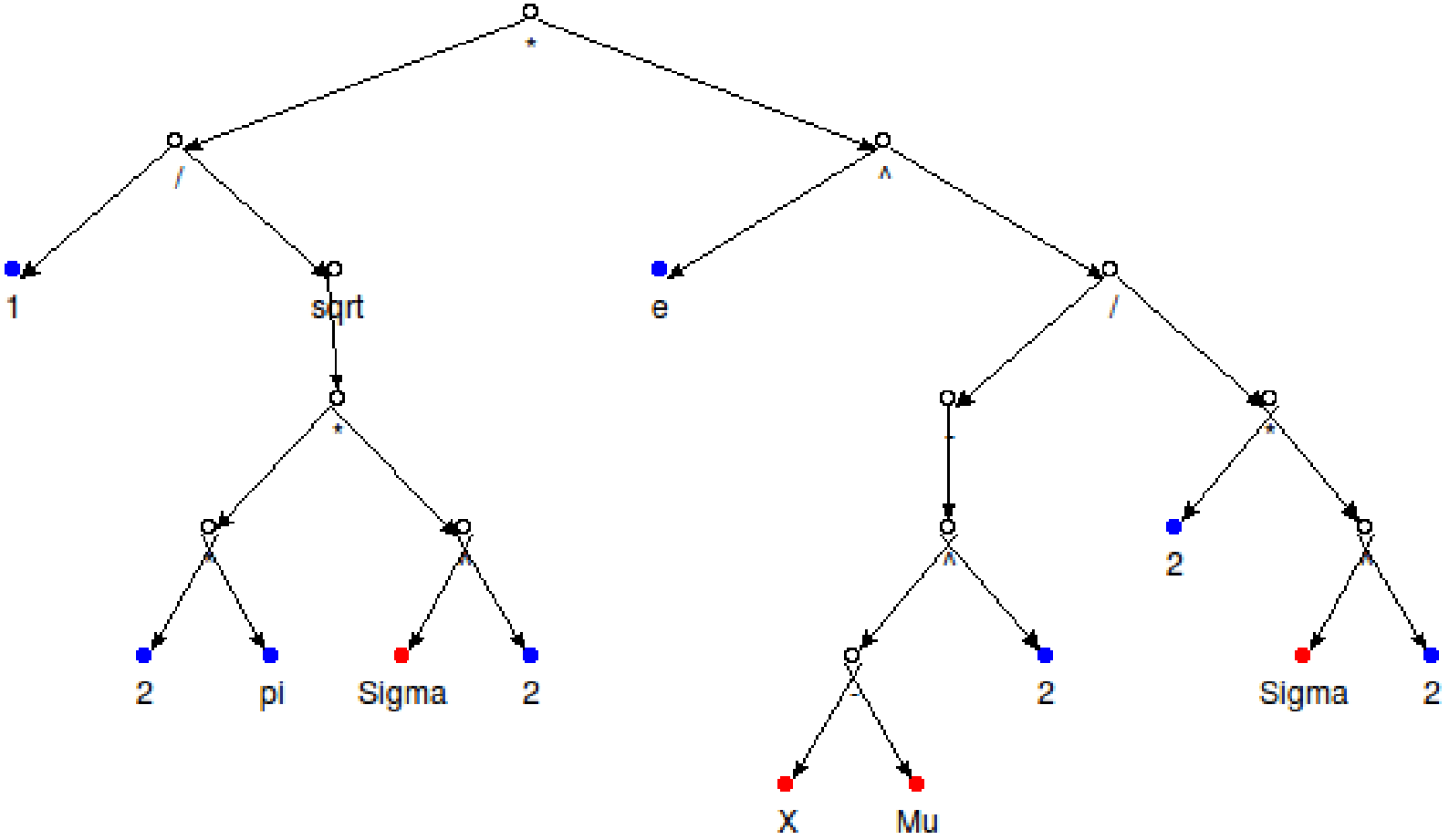
  zur Gaußverteilung kann in PROLOG geschrieben werden als

  ```
  1 / sqrt(2*pi*Sigma^2) *
     e^(-((X-Mu)^2)/(2*Sigma^2)).
  ```

  Hier sind `pi` und `e` PROLOG–Atome, die bei der Auswertung als π und e interpretiert werden.

  Natürlich müssen die Variable `X` und die statistischen Parameter `Sigma` (Standardabweichung) und `Mu` (Mittelwert) vor der Auswertung instanziiert werden.

- Der zugehörige *Operatorbaum* ist folgender:



Arithmetische Auswertung

Im Gegensatz zum Unifikationsprädikat `=/2` wertet das Prädikat `is/2` den arithmetischen Ausdruck zu seiner rechten aus und unifiziert das Ergebnis mit dem Ausdruck zu seiner linken:

```
?- X = 3, Y is e^X + sin(X).
X = 3, Y = 20.2267.
```

Der entsprechende Aufruf mit `=/2` würde `Y` dagegen lediglich mit dem Term `e^3 + sin(3)` unifizieren:

```
?- X = 3, Y = e^X + sin(X).
X = 3, Y = e^3 + sin(3).
```

Arithmetische Terme werden bei Zuweisungen mit dem Prädikat `=/2` also nicht ausgewertet:

```
?- X = 2,
   Y = 3 * X^2 + 2 * X + 1,
   Z is 3 * X^2 + 2 * X + 1.
X = 2,
Y = 3 * 2^2 + 2 * 2 + 1,
Z = 17
```

Erst der Aufruf `Z is ...` mit dem Prädikat `is/2` wertet den arithmetischen Term aus.

Symbolisches Differenzieren

Das Prädikat `diff/3` differenziert beim Aufruf `diff(F, X, DF)` einen Funktionsterm `F` symbolisch nach der Variablen `X` – die beim Aufruf durch eine Konstante instanziiert sein muß – und liefert `DF` als Ergebnis:

```
?- diff(e^x + sin(x), x, F).
F = e^x + cos(x).
?- diff(x^7 + 3 * x^6, x, F).
F = 7 * x^6 + 3 * 6 * x^5.
```

Dasselbe Prädikat `diff/3` kann bei einem anderen Bindungsmuster zum symbolischen Integrieren benutzt werden.

```
diff(X, X, 1) :-
   !.
diff(C, X, 0) :-
   atomic(C).
diff(X^N, X, N*X^M) :-
   N > 0, M is N - 1.
diff(sin(X), X, cos(X)).
diff(e^X, X, e^X).
diff(F+G, X, DF+DG) :-
   diff(F, X, DF), diff(G, X, DG).
diff(C*F, X, C*DF) :-
   atomic(C),
   !,
   diff(F, X, DF).
```

Listen sind spezielle PROLOG–Strukturen mit dem binären Funktor ".".

Liste: .(a, .(b, .(c, .(d, [ ]))))

äquivalente, abkürzende Schreibweise: [a,b,c,d]

leere Liste: [ ] (in JAVA: `null`)

Listenkonstruktion:

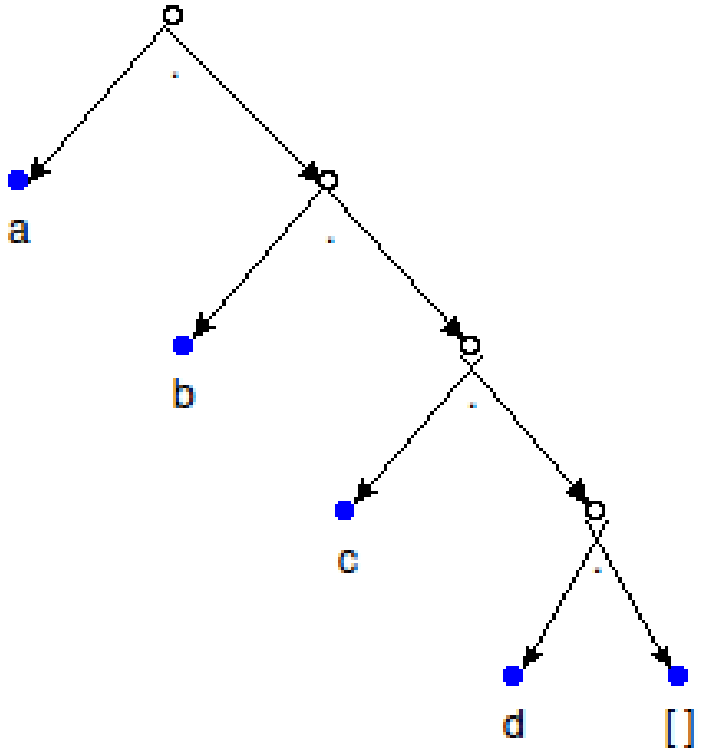
Falls X und Xs bereits gebunden (mit Werten belegt) sind, so ist [X|Xs] die Liste, die man erhält, wenn man das Element X an die Liste Xs vorne anhängt.

Für X = a und Xs = [b,c,d] ist [X|Xs] = [a,b,c,d].

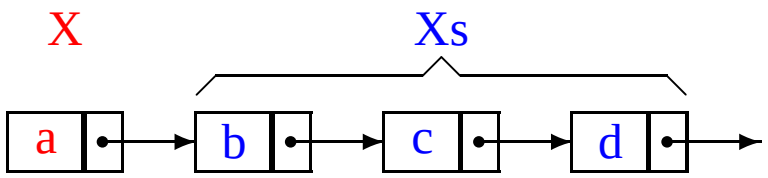
Listenzerlegung:

Falls X und Xs ungebunden sind, so liefert der Aufruf [X|Xs] = [a,b,c,d] die Belegungen X = a und Xs = [b,c,d]; man kann eine Liste also in ihren Kopf X und den Rest Xs zerlegen.

Entsprechend ihrer ausführlichen Schreibweise .(a, .(b, .(c, .(d, [ ])))), hat die Liste [a,b,c,d] den folgenden Operatorbaum:



Listenoperationen



Im PROLOG–Aufruf `[X|Xs] = List` stehen `Xs` und `List` für Listen und `X` für ein Listenelement. Im Vergleich zur gängigen JAVA–Listenklasse gilt:

```
class List { int key; List next; ... }
```

- Falls `List` gebunden ist und `X, Xs` frei, so entspricht der PROLOG–Aufruf den beiden JAVA–Zuweisungen `X = List.key;` `Xs = List.next` zur Listenzerlegung.
- Falls `List` frei ist und `X, Xs` gebunden, so entspricht der PROLOG–Aufruf den beiden JAVA–Zuweisungen `List.key = X; List.next = Xs` zur Listenkonstruktion.
- In PROLOG sind gemischte Fälle möglich, die dann mittels Unifikation behandelt werden (Erklärung folgt später).

In PROLOG sind die beiden Aufrufe `X = Y` und `Y = X` gleichwertig.

Listenkonstruktion und Listenzerlegung erfolgen also mittels desselben Aufrufs `List = [Head|Tail]`.

Die folgenden beiden Prädikate bestimmen das erste Listenelement `Head` und den Rest `Tail` der Liste.

```
list_to_head(List, Head) :-
   List = [Head|_].
list_to_tail(List, Tail) :-
   List = [_|Tail].
```

Um eine "`singleton variable warning`" zu vermeiden, wird der jeweils nicht benötigte Listenteil mittels einer anonymen Variable "_" (Wildcard) bezeichnet.

Die Listenschreibweise

$$[X_1,...,X_n]$$

nennt man *Syntactic Sugar*. Eine Liste ist eigentlich eine stark verschachtelte binäre Struktur:

$$.(\underbrace{X_1}_{\texttt{Head}}, \underbrace{.(X_2,...,.(X_n,[]))}_{\texttt{Tail}}).$$

Somit ist `List = [Head|Tail] = .(Head, Tail)`.

Für `List = [a,b,c,d] = .(a,.(b,.(c,.(d,[]))))` bestimmt der Aufruf `List = [Head|Tail]`, oder

```
.(a,.(b,.(c,.(d,[])))) = .(Head, Tail),
```

mittels *Unifikation* `Head = a` und `Tail = .(b,.(c,.(d,[])))`.

Die Listenkonstruktion aus einem gegebenen Kopf und einem Rest erfolgt – wie bereits gezeigt – auch mittels des Aufrufs `List = [Head|Tail]`:

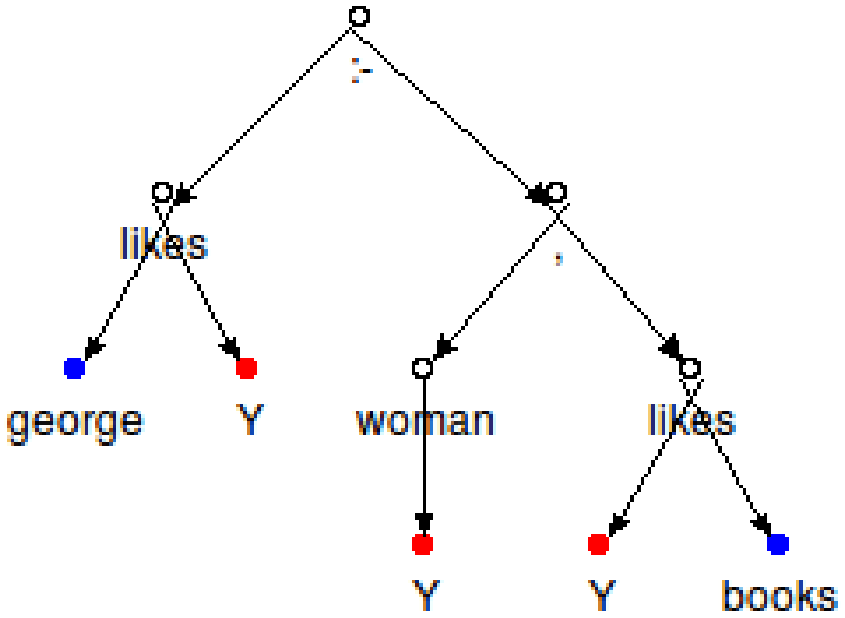
```
head_and_tail_to_list(Head, Tail, List) :-
   List = [Head|Tail].
```

Dasselbe Prädikat kann man auch benutzen, um eine gegebene Liste in Kopf und Rest zu zerlegen:

```
?- head_and_tail_to_list([a], [b,c,d], List).
List = [a,b,c,d].
?- head_and_tail_to_list(Head, Tail, [a,b,c,d]).
Head = [a], Tail = [b,c,d].
```

Regeln sind selbst auch spezielle PROLOG–Strukturen mit dem binären Infix–Funktor ‘`:-`’.

```
likes(george,Y) :-
   woman(Y), likes(Y,books).
```



Anders als in relationalen Datenbanken können auch **komplexe Datenstrukturen** – wie XML–Elemente – direkt durch verschachtelte Strukturen repräsentiert werden:

```
<component name="engine">
   <part name="sparkplug" quantity="4"/>
   <part name="cylinder" quantity="4"/>
   <part name="valve" quantity="4"/> ...
</component>

component( name(engine),
   part(name(sparkplug), quantity(4)),
   part(name(cylinder), quantity(4)),
   part(name(valve), quantity(4)) )
```

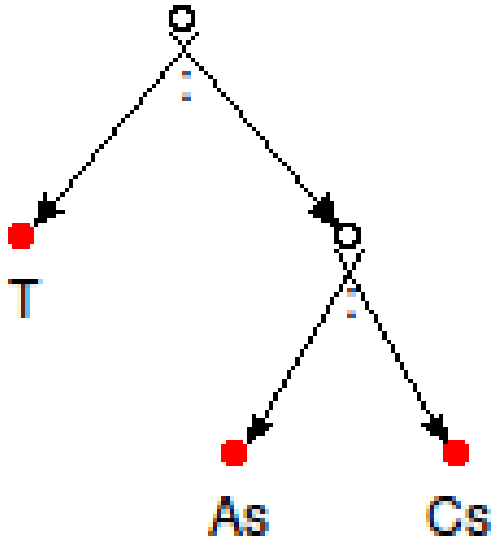
Dabei sind `component`, `part` und `name` Funktoren. Da unterschiedliche Teile unterschiedlich viele Komponenten haben können, schwankt die Stelligkeit von `component`. Dagegen sind `part` und `name` hier immer 2– bzw. 1–stellig.

Im Falle von XML ist ein generisches Document Object Model (DOM) auf der Basis von PROLOG–Strukturen noch eleganter, wobei die Attribut/Wert–Paare bzw. Unterelemente jeweils in PROLOG–Listen repräsentiert werden:

```
<component name="engine">
   <part name="sparkplug" quantity="4"/>
   <part name="cylinder" quantity="4"/>
   <part name="valve" quantity="4"/> ...
</component>

component:[name:engine]:[
   part:[name:sparkplug, quantity:4]:[ ],
   part:[name:cylinder, quantity:4]:[ ],
   part:[name:valve, quantity:4]:[ ] ]
```

Ein Term `T:As:Cs` repräsentiert ein XML–Element mit

- dem Tag `T`,
- der Liste `As` von Attribut/Wert–Paaren und
- der Liste `Cs` von Unterelementen.



Falls `As` leer ist, so kann man kurz `T:Cs` schreiben.

Z.B. kann ein XML–Element

```
<component name="engine">
   <part name="cylinder"/>
</table>
```

als komplexer Term in Field Notation repräsentiert werden:

```
component:[name:engine]:[
   part:[name:cylinder]:[] ].
```

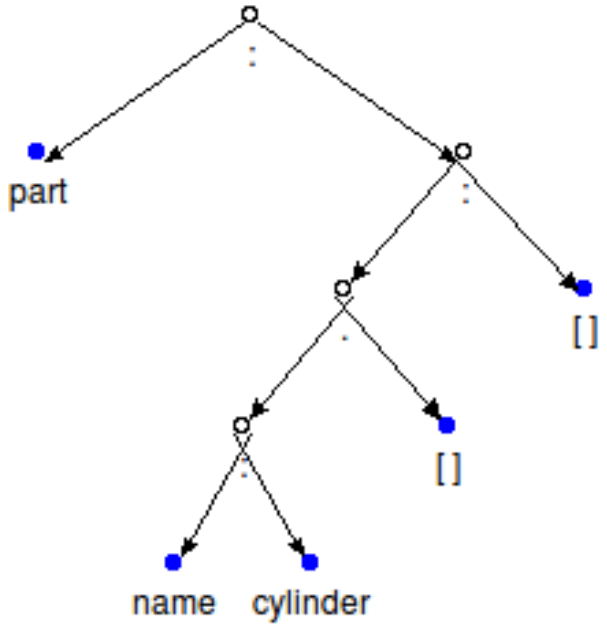
Diese Infix–Form benutzt den binären Funktor ”`:`”.

Der Sub–Term ”`name:engine`” könnte äquivalent auch in Präfix–Form als ”`:(name, engine)`” geschrieben werden.

Oben ist die Liste der Unterelemente des `part`–Elements leer.

Eine nicht–leere Liste wird bekanntlich als `.(X, Xs)` repräsentiert, wobei `X` das erste Element ist und `Xs` den Rest der Liste repräsentiert.

Also hat der Term `part:[name:cylinder]:[]`, der äquivalent ist zu `:(part, :(.(:(name,cylinder), []), []))`, den folgenden Operatorbaum:



Bindungsmuster von Operationen

Viele PROLOG–Prädikate können mit unterschiedlichen Bindungsmustern aufgerufen werden.

Ein Bindungsmuster $(\odot_1, \ldots, \odot_n)$ drückt dabei für ein n–stelliges Prädikat aus, ob eine Argumentposition gebunden oder frei ist:

- `+`: gebunden,
- `-`: frei,
- `?`: gebunden oder frei.

Man schreibt auch $p(\odot_1 X_1, \ldots, \odot_n X_n)$ um auszudrücken, daß sich $\odot_i$ auf das Argument X_i bezieht.

Atome und Zahlen

Die Konstante `'1'` und die natürliche Zahl `1` sind verschiedene Objekte.

Man kann sie aber mittels `atom_number(?Atom, ?Number)` ineinander konvertieren:

```
?- atom_number('1', N).
N = 1
?- atom_number(A, 1).
A = '1'
```

Das Bindungsmuster `(?,?)` kann nicht exakt ausdrücken, daß dabei mindestens eines der beiden Argumente gebunden sein muß: `(+,+)`, `(+,-)`, `(-,+)`.

Atome und Term–Strukturen

- `atom_codes(?X, ?Codes)` und `name(?X, ?Codes)` konvertieren ein Atom oder eine Zahl in die Liste der entsprechenden ASCII–Codes. Bei der Umkehrabbildung erzeugt `atom_codes` immer ein Atom, während `name`, falls möglich, eine Zahl erzeugt. `number_codes(?N, ?Codes)` konvertiert zwischen Zahlen und ASCII–Codes.

```
?- name(abc, Codes_1), name(123, Codes_2).
Codes_1 = [97, 98, 99],
Codes_2 = [49, 50, 51]
```

- Für fest vorgegebene Atome oder Zahlen kann man die Konversion zu ASCII–Codes noch einfacher durchführen:

```
?- Codes = "abc".
Codes = [97, 98, 99]
```

- Mittels `concat(?Atom_1, ?Atom_2, ?Atom_3)` kann man zwei Atome zu einem neuen Atom konkatenieren.
- Mittels `term_to_atom(?Term, ?Atom)` kann man einen Term in ein entsprechendes Atom konvertieren, und umgekehrt.

```
?- concat('f(a,', 'b)', A),
   term_to_atom(T, A).
A = 'f(a,b)',
T = f(a, b)
```

Die anonyme Variable "_"

Man benutzt "_", wenn man sich für eine Argumentposition nicht interessiert. Jedes Vorkommen von "_" in einem Programm steht für eine andere Variable.

Eingabe und Ausgabe von Termen

- Mittels `read(-Term)` kann man einen Term von der Konsole einlesen. Die konkreten Namen der Variablen gehen dabei verloren.
- Mittels `write(+Term)` und `writeq(+Term)` kann man einen Term auf die Konsole schreiben. `writeq` umfaßt dabei Konstanten, die mit einem Großbuchstaben beginnen (und die beim erneuten Einlesen der Ausgabe mit einer Variablen verwechselt würden), mit Hochkommatas.

```
?- read(Term), writeln(Term), writeq(Term).
|: f(X, 'Y', c).
f(_G255, Y, c)
f(_G255, 'Y', c)
Term = f(_G255, 'Y', c)
```

- Im DDK ist `writeln(Term)` definiert als `write(Term), nl`.

Einlesen von Termen mit Variablen

- Mittels `read_term(-Term, [variable_names(Vars)])` kann man einen Term zusammen mit den enthaltenen Variablennamen von der Konsole einlesen.
- Mittels `var(+X)` bzw. `nonvar(+X)` kann man testen, ob `X` eine ungebundene Variable ist oder nicht.

```
?- var(Term),
   read_term(Term, [variable_names(Vars)]),
   nonvar(Term).
|: f(X, 'Y', c).
Term = f(_G270, 'Y', c), Vars = ['X'=_G270]
```

Nach dem Einlesen ist `Term` keine ungebundene Variable mehr.

Einlesen von Dateien, Ein–/Ausgabe–Umlenkung

- Mittels `dread(txt, File, Name)` kann man eine Text–Datei in ein PROLOG–Atom `Name` einlesen.
- Mittels `dwrite(txt, File, Name)` kann man ein PROLOG–Atom `Name` in eine (neue) Text–Datei ausgeben.
- `dread/3` und `dwrite/3` werden wir später noch für andere Datei–Typen kennen lernen.
- Man kann den Ein–/Ausgabe–Strom mittels `see(File)` bzw. `tell(File)` umlenken.
  `seen` bzw. `told` setzt dann wieder auf die Konsole zurück.

2.2 Programmierstrukturen

Basis–Programmierstrukturen in PROLOG sind die Konjunktion `A,B` (Hintereinanderausführung) und die Disjunktion `A;B` (alternative Ausführung) von Statements `A` und `B`.

Die folgenden komplexen Programmierstrukturen sind sehr charakteristisch für PROLOG:

- Rekursion,
- relationales Programmieren.

Bei der Programmierung werden ständig eingesetzt:

- Backtracking und Cut (Behandlung von Alternativen),
- Unifikation (z.B. zur Parameterübergabe),
- Meta–Prädikate (für komplexe Kontrollstrukturen).

Beim *relationalen* Programmieren können mehrere Antworten auf eine Anfrage generiert werden:

- In PROLOG wird diese Generierung explizit mittels Backtracking angestoßen.
- In relationalen Datenbanken werden bei SQL–Anfragen implizit immer alle Antworten – z.B. mittels eines Nested–Loop–Joins – generiert.
- In DATALOG werden ebenfalls implizit alle Antworten im Rahmen einer Iteration generiert.

PROLOG unterstützt auch das *funktionale* Programmieren:

- Meta–Prädikate (z.B. `findall`) haben – analog zu den Funktionalen der Mathematik (z.B. dem Integral: $\int f(x)\,dx$) – Prädikate als Parameter.
- Man kann auch die Komposition von Prädikaten unterstützen – in der Mathematik kennt man die Komposition $f(g(x))$ von Funktionen.

2.2.1 Rekursion

Wir werden das Programmieren mit Rekursion anhand folgender typischer Programmbeispiele illustrieren:

- Türme von Hanoi,
- Fakultät,
- Listendurchlauf (in JAVA: while–Schleife): `sum`, `member`, `append`,
- Sortierverfahren: Quicksort, Mergesort.

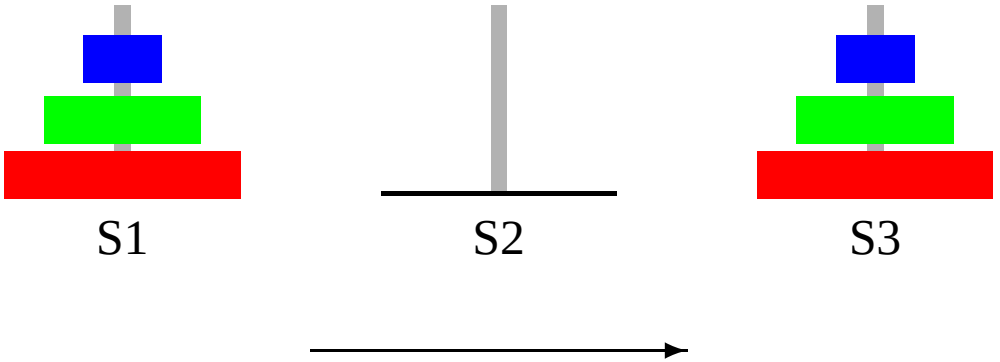
In PROLOG ist Rekursion sehr verbreitet. Auch der Listendurchlauf, der in herkömmlichen Programmiersprachen meist *iterativ* implementiert wird, wird in PROLOG rekursiv implementiert.

Aber man kann das rekursive Durchlaufen einer Liste oft mittels eines Meta–Prädikats verbergen und damit den Code übersichtlicher gestalten.

Türme von Hanoi

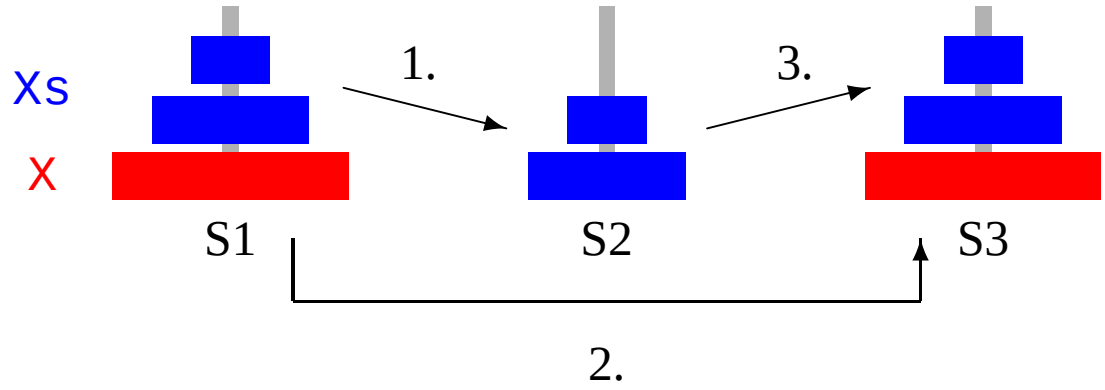
Das Problem besteht darin einen sich nach oben verjüngenden Stapel von Scheiben von Position S1 nach S3 unter Zuhilfenahme einer Zwischenablage S2 zu bewegen.

- Es darf immer nur die oberste Scheibe eines Stapels verschoben werden, und
- es darf nie eine größere Scheibe auf einer kleineren liegen.



Rekursive Lösung:

1. Da X die größte Scheibe ist, können wir den Rest–Stapel Xs ohne X zu beachten von S1 unter Zuhilfenahme von S3 nach S2 verschieben.
2. Dann verschieben wir X von S1 nach S3.
3. Nun können wir den Rest–Stapel Xs ohne X zu beachten von S2 unter Zuhilfenahme von S1 nach S3 verschieben.



Komplexität: $T(N) = 2 \cdot T(N-1) + 1 \longrightarrow T(N) = 2^N - 1.$

Implementierung in PROLOG:

```
/* move(+[S1,S2,S3], +Xs) <-
      move the discs in Xs
      from S1 to S3 using S2. */

move([S1,S2,S3], [X|Xs]) :-
   move([S1,S3,S2], Xs),
   concat(['move disc ',X,
      ' from ',S1,' to ',S3], Text),
   write(Text), nl,
   move([S2,S1,S3], Xs).
move(_, []).
```

Die größte, d.h. die unterste Scheibe `X` ist in der Liste `[X|Xs]` die erste.
Damit können wir den Stapel gut in `X` und den oberen Rest–Stapel `Xs` zerlegen.

```
?- move([s1,s2,s3], [3,2,1]).
move disc 1 from s1 to s3
move disc 2 from s1 to s2
move disc 1 from s3 to s2
move disc 3 from s1 to s3
move disc 1 from s2 to s1
move disc 2 from s2 to s3
move disc 1 from s1 to s3
Yes
```

Natürlich wird die unterste Scheibe `X` erst bewegt, nachdem die darüber liegenden Scheiben aus `Xs` weg bewegt wurden.

Rekursive Fakultätsberechnung

Die Fakultät kann rekursiv berechnet werden als $N! = N \cdot (N-1)!$
Dabei startet man mit $N! = 1$, für $N < 2$.

```
/* factorial(+N, -F) <-
      F is computed as the factorial of N. */

factorial(N, 1) :-
   N < 2,
   !.
factorial(N, F) :-
   M is N - 1,
   factorial(M, G),
   F is N * G.
```

Beim Aufruf “`X is T`” wird der arithmetische Ausdruck “`T`” ausgewertet, und das Ergebnis wird an “`X`” zugewiesen.

Bemerkungen:

- Abgesehen von den nicht erforderlichen Typ–Deklarationen entspricht die PROLOG–Implementierung ziemlich genau der JAVA–Implementierung, die ebenfalls sehr elegant rekursiv erfolgen kann.
- Würde man das Prädikat `is` in den Auswertungen `M is N - 1` und `F is N * G` durch `=` ersetzen, so würde die obige Implementierung anstelle der Fakultät nur das entsprechende Produkt `F` liefern, das man dann aber nachträglich weiter auswerten könnte:

```
?- factorial(4, F), F is G.
F = 4 * (4-1) * ((4-1)-1) * (((4-1)-1)-1),
G = 24
```

  Die Vergleiche `N < 2` funktionieren auch für arithmetische Terme `N`.
- Auf die Bedeutung des Cut "`!`" in der ersten Regel werden wir später eingehen.

Rekursive Summenberechnung

```
sum([X|Xs], Sum) :-
   sum(Xs, Sum_2),
   Sum is X + Sum_2.
sum([], 0).
```

Bei der Summenberechnung über eine Liste `[1,2,3,4]` berechnet die erste Regel zuerst

- mittels `sum(Xs, Sum_2)` die Summe `Sum_2 = 9` der hinteren Elemente `Xs = [2,3,4]`;
- diese wird dann mittels `Sum is X + Sum_2` zu `X = 1` addiert, so daß sich `Sum = 10` ergibt:

Die zweite Regel liefert für die leere Liste von Summanden den Wert `0`.

In der Praxis wird aber meist eine effizientere, end–rekursive Implementierung mittels Akkumulatoren verwendet.

```
sum(Xs, Sum) :-
   sum(Xs, 0, Sum).

sum([X|Xs], Acc, Sum) :-
   Acc_2 is Acc + X,
   sum(Xs, Acc_2, Sum).
sum([], Acc, Acc).
```

Die Listenelemente `X` werden von links beginnend auf einen Akkumulator, der mit `0` initialisiert wurde, addiert (`Acc_2 is Acc + X`). Das Hilfsprädikat `sum/3` wird am Ende seiner ersten Regel rekursiv aufgerufen.

Die zweite Regel für `sum/3` schiebt am Ende der Berechnung den Akkumulator direkt in die Ausgabe, die auf allen Rekursionsebenen dieselbe ist.

Iteration als Rekursion

- In PROLOG kann man einer Variable nur einmal einen Wert zuweisen; es gibt aus sprachtechnischen Gründen kein *destruktives Assignment*.
- Schleifen, in denen iterativ ein Wert aktualisiert werden soll, implementiert man deswegen meist rekursiv: in jedem rekursiven Aufruf verwendet man eine *frische Variable* zur Speicherung des aktualisierten Werts.

In `sum/3` wird eine frische Variable `Acc_2` benutzt:

- aus dem alten Wert `Acc` wird mittels des Aufrufs `Acc_2 is Acc + X` ein neuer Wert `Acc_2` berechnet, der dann rekursiv weiter bearbeitet wird.

Der rekursive Aufruf steht am Ende der Regel (Endrekursion), und die Rückgabevariable `Sum` ist auf allen Rekursionsstufen dieselbe.

Das rekursive Listen–Prädikat member/2

```
/* member(?X, +L) <-
      X is a member of the list L. */

member(X, [X|_]).
member(X, [_|Xs]) :-
   member(X, Xs).

?- member(a, [c, b, a]).
Yes
```

X ist ein Element der Liste L, falls

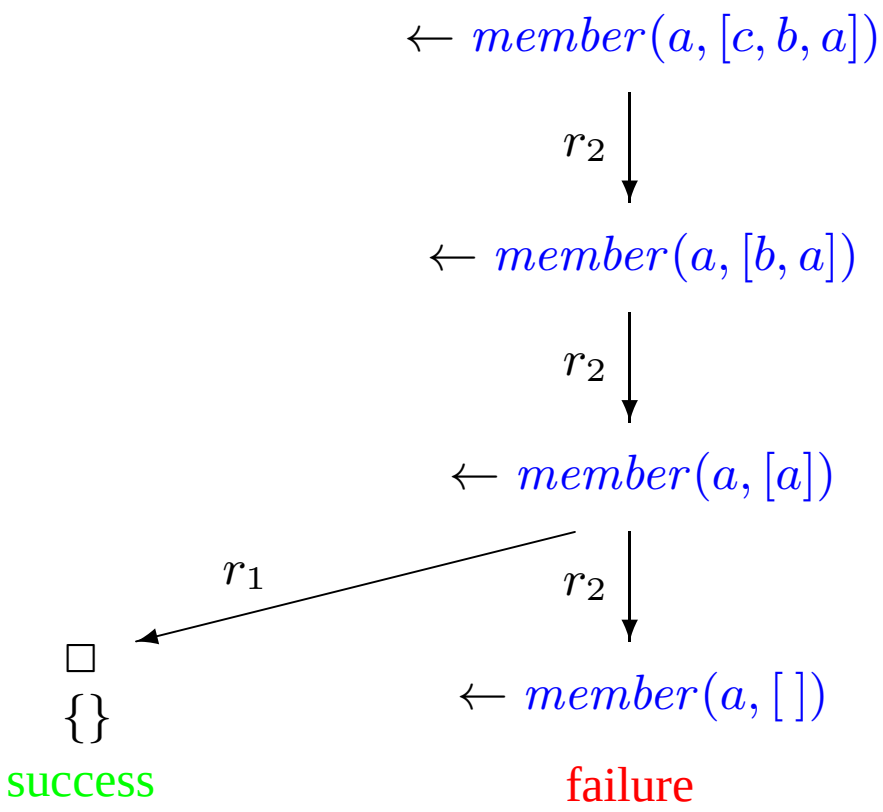
- X das erste Element der Liste ist (erste Regel) oder
- im Rest Xs der Liste vorkommt (zweite Regel).

Langform:

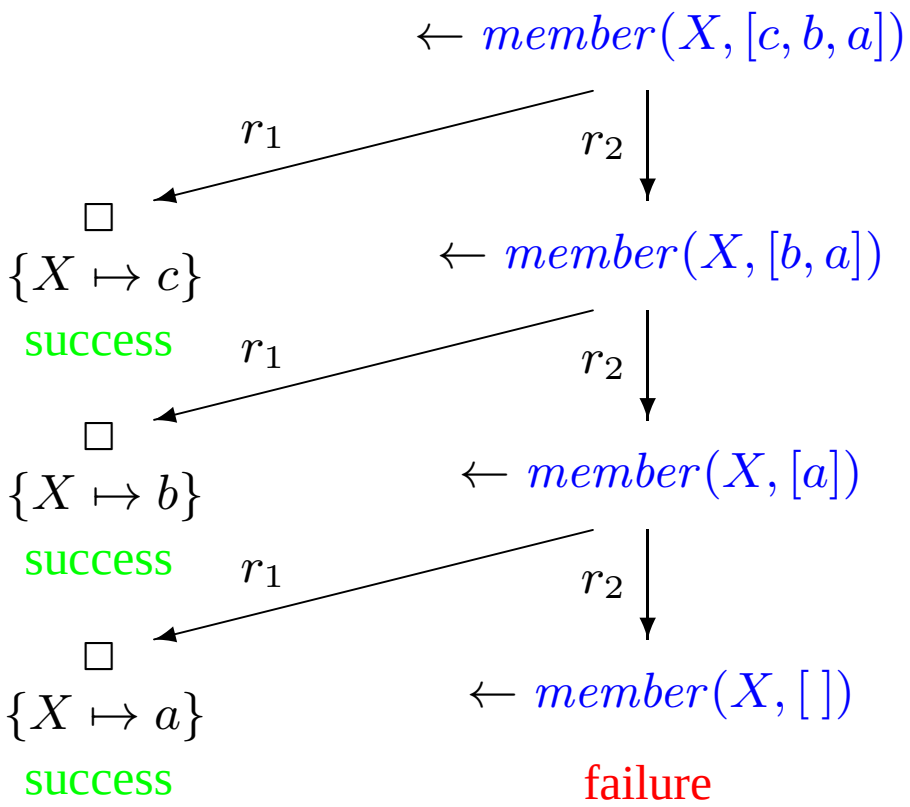
```
/* member(?X, +L) <-
      X is a member of the list L. */

member(X, L) :-
   L = [X|_].              % X  = L.key
member(X, L) :-
   L = [_|Xs],             % Xs = L.next
   member(X, Xs).
```

Man kann die Listenzerlegungen `L = [X|_]` und `L = [_|Xs]` aus dem Regelkopf in den Regelrumpf ziehen.

Der Aufruf `member(a, [c, b, a])` führt zu folgendem SLD-Baum:

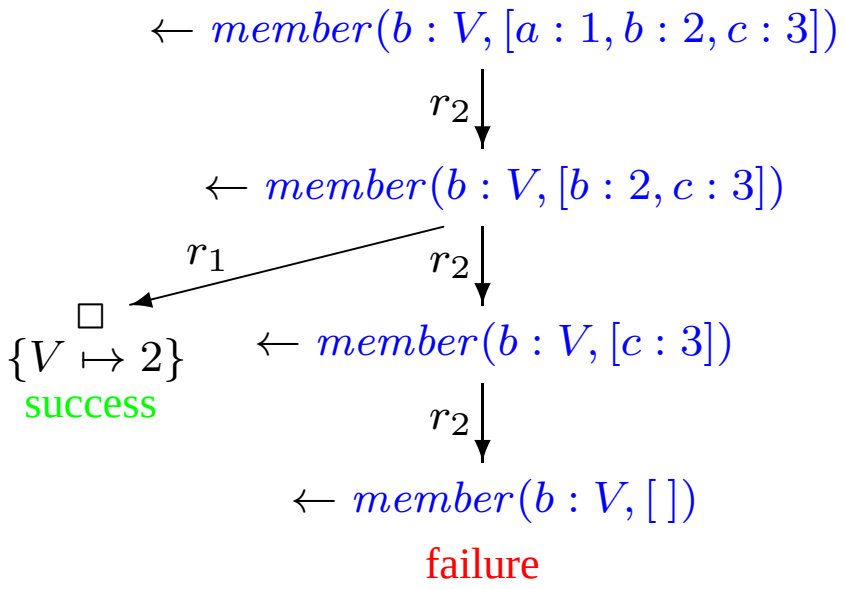


Der Aufruf `member(X, [c, b, a])` führt zu folgendem SLD–Baum:



Mittels des PROLOG–Aufrufs `member(+A:-V, +As)` und Backtracking kann man z.B. auch den Wert `V` eines Attributs `A` aus einer Liste `As` von Attribut/Wert–Paaren (vgl. XML) bestimmen:

```
?- member(b:V, [a:1, b:2, c:3]).
V = 2
```



Der PROLOG–Aufruf `members(+Xs, +Ys)` testet alle Elemente der Liste `Xs` of Mitgliedschaft in der Liste `Ys`. Gegebenenfalls werden Variablen in `Xs` geeignet belegt.

```
/* members(+Xs, +Ys) <-
      */

members([X|Xs], Ys) :-
   member(X, Ys),
   members(Xs, Ys).
members([], _).

?- members([b, c], [a, b, c]).
Yes
?- members([a(X), b(X)], [a(1), a(2), b(1), b(2), b(3)]).
X = 1 ;
X = 2
```

Oft kann man das rekursive Durchlaufen einer Liste mittels eines Meta–Prädikats verbergen und damit den Code übersichtlicher gestalten – wie hier mittels `checklist/2`:

```
members(Xs, Ys) :-
   checklist( member_inverse(Ys), Xs ).

member_inverse(Ys, X) :-
   member(X, Ys).

?- Xs = [a(X), b(X)],
   Ys = [a(1), a(2), b(1), b(2), b(3)],
   members(Xs, Ys).
X = 1 ;
X = 2
```

Dabei benötigen wir hier – aus technischen Gründen – ein weiteres Prädikat `member_inv/2`, das die Argumentreihenfolge von `member/2` umdreht.

Vielleicht noch etwas eleganter ist hier die Verwendung des Meta–Prädikats `do/2` – dann ist kein Hilfsprädikat erforderlich:

```
members(Xs, Ys) :-
   foreach(X, Xs) do member(X, Ys).

?- Xs = [a(X), b(X)],
   Ys = [a(1), a(2), b(1), b(2), b(3)],
   members(Xs, Ys).
X = 1 ;
X = 2
```

Hier wird für jedes Element von `Xs` getestet, ob es auch ein Element von `Ys` ist.

2.2.2 Prädikatenorientiertes, relationales Programmieren

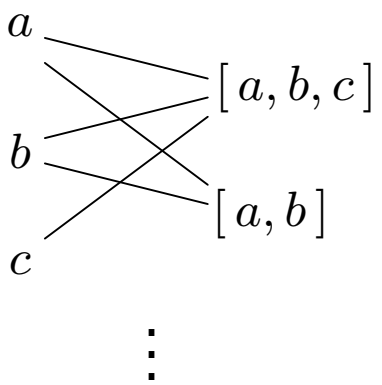
member : Element $\times$ List $\mapsto$ Boolean

```
?- member(b, [a, b, c]).
Yes

?- member(X, [a, b, c]).
X = a ;
X = b ;
X = c
```

Beim Aufruf `member(X, Y)` sind `X` und `Y` nicht funktional voneinander abhängig. Stattdessen realisiert `member/2` eine n:m–Beziehung (Relation) zwischen Listen und ihren Elementen.

Das Prädikat `member/2` definiert eine unendliche, virtuelle Relation:

member	
Element	List
a	$[a, b, c]$
b	$[a, b, c]$
c	$[a, b, c]$
a	$[a, b]$
...	...



Man dürfte `member(X, Xs)` sogar aufrufen, wenn beide Argumente ungebunden sind. Dann würden allerdings unendlich viele Antworten generiert. Falls die Liste `Xs` bekannt ist, so generiert der Aufruf bei Backtracking der Reihe nach alle Elemente `X` von `Xs`.

Auch das Prädikat `append/3` kann mit unterschiedlichen Bindungsmustern aufgerufen werden:

$$\text{append} : \text{List} \times \text{List} \times \text{List} \mapsto \text{Boolean}$$

```
?- append([a], [b, c], Zs).
Zs = [a, b, c]

?- append(Xs, Ys, [a, b, c]).
Xs = [ ], Ys = [a, b, c] ;
Xs = [a], Ys = [b, c] ;
Xs = [a, b], Ys = [c] ;
Xs = [a, b, c], Ys = [ ]
```

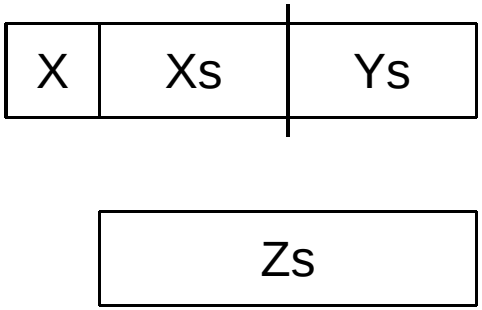
Bindungsmuster

- `+`: gebunden, `-`: frei, `?`: gebunden oder frei
- Der Aufruf `append(+Xs, +Ys, -Zs)` ist funktional; er realisiert eine Funktion

  $$\mathsf{append} : \mathsf{List} \times \mathsf{List} \mapsto \mathsf{List}$$

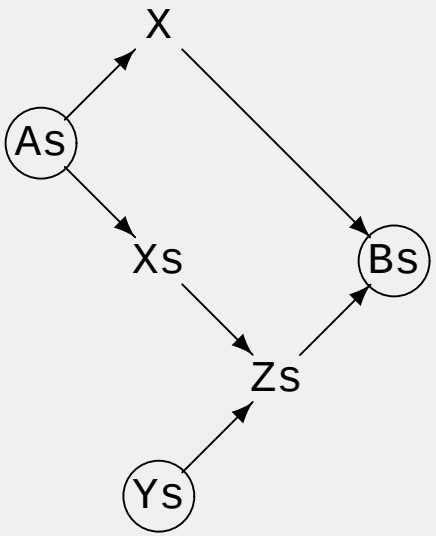
- Der Aufruf `append(+Xs, -Ys, +Zs)` ist zwar relational, aber es gibt immer maximal eine Liste `Ys` als Antwort (partielle Funktion); analog verhält es sich mit `append(-Xs, +Ys, +Zs)`.
- Der Aufruf `append(-Xs, -Ys, +Zs)` ist relational.

```
/* append(Xs, Ys, Zs) <-
      appending the lists Xs and Ys
      yields the list Zs. */

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



Langform:

```
/* append(Xs, Ys, Zs) <-
      appending the lists Xs and Ys
      yields the list Zs. */

append(Xs, Ys, Zs) :-
   Xs = [],
   Zs = Ys.
append(As, Ys, Bs) :-
   As = [X|Xs],              % Listenzerlegung
   append(Xs, Ys, Zs),
   Bs = [X|Zs].              % Listenkonstruktion
```



Die Resultatsliste `Zs` entsteht durch Kopieren der Elemente von `Xs` und `Ys`.

Die angegebenen Implementierungen (Kurzform und Langform) ermöglichen einen weiteren interessanten Aufruf, der zwei aufeinanderfolgende Elemente `X` und `Y` einer Liste `Zs` bestimmt:

```
?- Zs = [a, b, c, d],
   append(_, [X,Y|_], Zs).
X = a, Y = b ;
X = b, Y = c ;
X = c, Y = d
```

Im Prinzip werden zwei Listen `Xs` und `Ys` gesucht, deren Konkatenation `Zs` ergibt; von der zweiten Liste `Ys` werden die ersten beiden Elemente `X` und `Y` ausgewählt – woraus sich auch ergibt, daß `Ys` mindestens zwei Elemente enthalten muß.

Unifikation

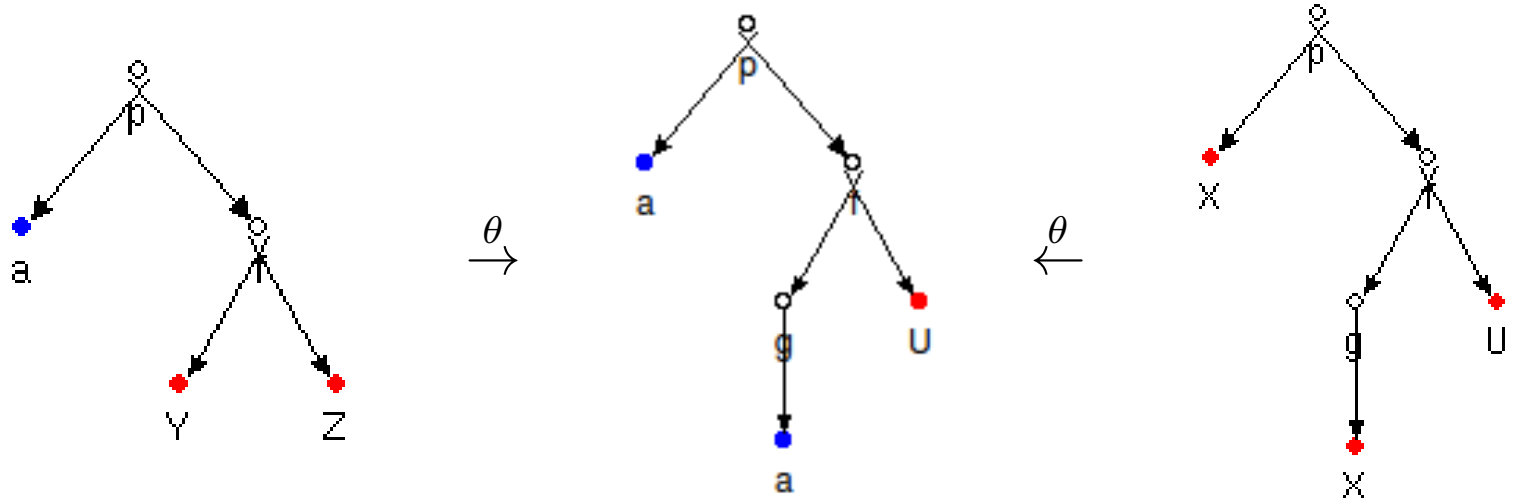
Wir suchen eine Substitution θ zu zwei Termen τ und τ', so daß gilt $\tau\theta = \tau'\theta$. Offensichtlich kann man nur unifizieren, wenn beide Terme denselben Funktor haben.

- Für $\tau = p(a, Y)$ und $\tau' = p(X, b)$ erhalten wir
  $$\theta = \{ X \mapsto a,\ Y \mapsto b \}.$$
- Für $\tau = p(a, X)$ und $\tau' = p(X, b)$ gibt es keinen Unifikator, denn die beiden Zuordnungen $X \mapsto a$ und $X \mapsto b$ sind unvereinbar.
  Deswegen schlägt der PROLOG–Aufruf `?- p(a,X) = p(X,b).` fehl, da die zwei Vorkommen der Variable X dasselbe Objekte bezeichnen.
- Aber wenn man zuerst X in $p(a, X)$ durch Y ersetzt, dann kann man sie unifizieren. Wenn $p(a, X)$ ein Fakt ist, dann ist der PROLOG–Aufruf `?- p(X,b).` erfolgreich mit $\theta = \{ X \mapsto a \}$, da die zwei Vorkommen der Variable X unterschiedliche Objekte bezeichnen können.

Für die Terme

$$\tau = p(a, f(Y, Z)), \quad \tau' = p(X, f(g(X), U)),$$

erhalten wir $\theta = \{ X \mapsto a, Y \mapsto g(a), Z \mapsto U \}$.



Die Blätter der Bäume zu den Termen werden im Rahmen der Unifikation von links nach rechts abgearbeitet. Der Termbaum zu $\tau\theta = \tau'\theta = p(a, f(g(a), U))$ ist in der Mitte dargestellt.

Parameterübergabe mittels Unifikation

- Zur Bearbeitung des Aufrufs τ eines Prädikats p werden alle Regeln zu diesem Prädikat herangezogen.
- Eine Regel kann benutzt werden, wenn der Regelkopf τ' mit dem Aufruf τ unifiziert werden kann.
- Zum Beispiel werden

  τ = `append([a, b, c], [d], Ws)` und
  τ' = `append([X|Xs], Ys, [Z|Zs])`

  mittels $\theta = \{$ `X` $\mapsto$ `a`, `Xs` $\mapsto$ [`b`, `c`], `Ys` $\mapsto$ [`d`], `Ws` $\mapsto$ [`Z`|`Zs`] $\}$ unifiziert.
- τ und τ'' = `append([], Ys, Ys)` können dagegen nicht unifiziert werden, da die leere Liste `[]` nicht mit `[a, b, c]` unifiziert.

  Die leere Liste ist eine Konstante und somit ihr eigener Funktor.
  Die andere Liste hat den Funktor "`.`".

Funktionales Programmieren

Man kann jede Funktion

$$\mathsf{f} : \mathsf{Domain} \mapsto \mathsf{Range}$$

auch als Relation $\mathsf{r_f} : \mathsf{Domain} \times \mathsf{Range} \mapsto \mathsf{Boolean}$ realisieren mit

$$f(x) = y \iff r_f(x, y) = true.$$

In diesem Sinne ist das relationale Programmieren also allgemeiner als das funktionale Programmieren.

Im DDK kann man funktional programmieren:

```
?- Zs <= append([c,d], [e,f]).
Zs = [c,d,e,f]
```

Dieser funktionale Aufruf wird behandelt wie der relationale Aufruf `append([c,d], [e,f], Zs)`.

Bei der *Komposition* von Prädikaten spart die funktionale Schreibweise unübersichtliche (temporäre) Kettvariablen (unten: `Zs`) ein:

```
?- Xs <= append([a,b], append([c,d], [e,f])).
Xs = [a,b,c,d,e,f]
```

Das binäre Prädikat `<=` ist im DDK so implementiert, daß die Berechnung auf die relationale Standard–Schreibweise herunter–compiliert wird:

```
?- append([c,d], [e,f], Zs),
   append([a,b], Zs, Xs).
Zs = [c,d,e,f],
Xs = [a,b,c,d,e,f]
```

Interessanterweise kann man die beiden Aufrufe sogar vertauschen:

```
?- append([a,b], Zs, Xs),
   append([c,d], [e,f], Zs).
Zs = [c,d,e,f],
Xs = [a,b,c,d,e,f]
```

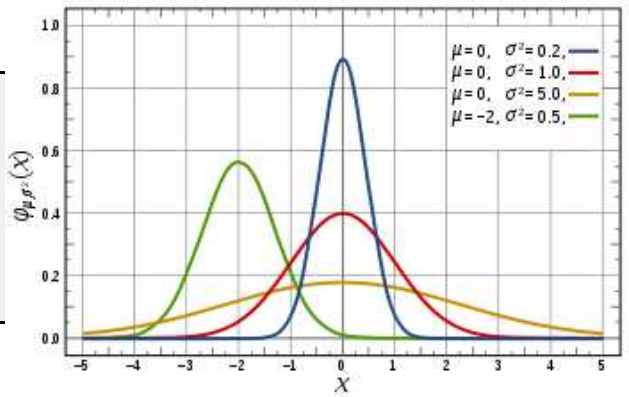
Dann hängt der erste Aufruf `append([a,b], Zs, Xs)` eine variable Liste `Zs` an `[a,b]` an, so daß die Resultatsliste `Xs` am Ende eine Andockstelle hat.

Der zweite Aufruf `append([c,d], [e,f], Zs)` berechnet erst den Wert von `Zs` und füllt somit die Andockstelle.

Damit ist die Resultatsliste `Xs` komplett.

Die folgende Regel definiert z.B. die Gaußverteilung:

```
gauss(Mu, Sigma, X, Y) :-
   Y is 1 / sqrt(2*pi*Sigma^2) *
      e^(-((X-Mu)^2)/(2*Sigma^2)).
```



Dann kann man Berechnungen durchführen:

```
?- A <= gauss(0, 1, 0), B <= gauss(0, 1, 2),
   C <= gauss(0, 1, gauss(0, 1, 2)).
A = 0.398942, B = 0.053991, C = 0.398361
```

Ein funktionaler Aufruf $\mathtt{X}_{n+1}$ `<=` $\mathtt{p(X}_1\mathtt{,...,X}_n\mathtt{)}$ (Zuweisung) mit Variablen oder Konstanten X_i wird wie ein relationaler Aufruf $\mathtt{p(X}_1\mathtt{,...,X}_n\mathtt{,X}_{n+1}\mathtt{)}$ behandelt. Eingebettete komplexe Terme X_i ($1 \leq i \leq n$) werden vor dem relationalen Aufruf ebenfalls ausgewertet.

Der folgende Aufruf ist dagegen unsinnig und schlägt fehl, da die Unifikation `gauss(Mu, Sigma, X) = 1 / sqrt(2*pi*Sigma^2) * ...` fehlschlägt:

```
?- gauss(Mu, Sigma, X) =
      1 / sqrt(2*pi*Sigma^2) *
         e^(-((X-Mu)^2)/(2*Sigma^2)),
   A <= gauss(0, 1, 0).
No
```

Hier wird nicht etwa eine Funktion definiert, sondern es wird versucht die beiden Terme zur linken und rechten des Prädikatensymbols `=` zu unifizieren.
Diese Unifikation ist natürlich schon deswegen unmöglich, da die beiden Terme die unterschiedlichen Funktionssymbole `gauss` bzw. `*` besitzen.

2.2.3 Backtracking und der Cut

- Beim relationalen Programmieren können Aufrufe prinzipiell verschiedene Rückgabewerte liefern.
- Dies kann man mit dem Verhalten nicht–deterministischer Turing–Maschinen vergleichen.
- In der Praxis sind die meisten Aufrufe aber funktional.
- Bei nicht–funktionalen Prädikaten sind die verschiedenen Rückgabewerte meist erwünscht, und man kann sie mittels Backtracking bestimmen.
- Wenn das nicht so ist, dann kann man Teilbäume des SLD–Baumes mithilfe des Cuts abschneiden.

Wirkung des Cuts

- Der Aufruf des Cuts “ ! ” friert alle Entscheidungen, die während der Bearbeitung in einer Regel bisher getroffen wurden, ein.
- Insbesondere können beim Backtracking auch keine alternativen Regeln zum gleichen Prädikat mehr benutzt werden.
- Ein Cut in der Mitte einer Regel friert lediglich die vorher erzeugten Variablenbelegungen ein.
- Ein Cut am Ende einer Regel unterbindet z.B. komplett das Backtracking innerhalb dieser Regel.

  Wenn die Regel einmal abgearbeitet ist, dann werden die erzeugten Variablenbelegungen später nicht mehr verändert.

Der Cut in `factorial/2`

Der Cut in der ersten Regel von `factorial/2` ist zunächst nicht erforderlich.

```
/* factorial(+N, -F) <-
      F is computed as the factorial of N. */

factorial(N, 1) :-
   N < 2,
   !.
factorial(N, F) :-
   M is N - 1,
   factorial(M, G),
   F is N * G.
```

Er verhindert aber, daß bei einem eventuellen Backtracking die zweite Regel für Werte $N \leq 0$ benutzt werden kann, was zu einer Endlosschliefe führen würde.

Backtracking mittels `findall/3`:

- Der folgende Aufruf berechnet die Fakultäten der ganzen Zahlen zwischen 1 und 3:

```
?- findall( F,
        ( between(1, 3, N),
          factorial(N, F) ),
        Fs ).
Fs = [1, 2, 6].
```

- `between(I, K, N)` generiert bei Backtracking die ganzen Zahlen `N` zwischen `I` und `K`.
- Der Cut in der ersten Regel von `factorial/2` verhindert die Endlosschliefe beim Backtracking.

Das Meta–Prädikat findall/3

- Finden aller Lösungen für ein Goal:

```
findall( X,
   goal(X),
   Xs )
```

Es wird die Liste `Xs` aller `X` berechnet, für die `goal(X)` erfolgreich ist. Im DDK kann man sogar folgende mathematische Mengenschreibweise verwenden; das Infix–Prädikat `<=` wertet den Mengenausdruck zu seiner rechten aus und weist das Resultat per Unifikation `Xs` zu:

```
Xs <= { X | goal(X) }
```

- Im Fakultätsbeispiel:

```
Fs <= { F | between(1, 3, N), factorial(N, F) }.
```

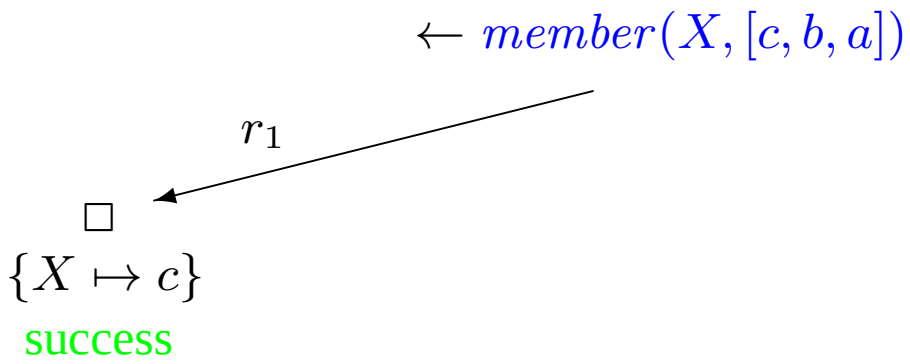
Der Cut in `member/2`

Ein zusätzlicher Cut in der ersten Regel von `member/2`

- würde bei Aufrufen `member(X, Xs)` mit einer freien Variable `X` und einer vorgegebenen Liste `Xs` dazu führen, daß nur noch das erste Listenelement zurückgegeben würde;
- außerdem wären Aufrufe mit einem gebundenen Argument `X` nur noch maximal einmal erfolgreich:

```
member(X, [X|_]) :-
   !.
member(X, [_|Xs]) :-
   member(X, Xs).
```

Beim Aufruf `member(X, [c, b, a])` würde die Wahl der ersten Regel eingefroren und damit der zweite SLD–Teilbaum komplett abgeschnitten:



Also würde nur das erste Element der Liste gefunden.

Der Cut ist nur für Aufrufe mit einem gebundenen ersten Argument sinnvoll, und zwar dann, wenn man nur wissen möchte, ob dieses ein Element der Liste ist – ohne, daß man feststellen möchte, ob es wiederholt vorkommt.

Der Cut in der Maximum–Funktion

Das folgende Prädikat berechnet das Maximum Z zweier Zahlen X und Y:

```
/* max(+X, +Y, ?Z) <-
      */

max(X, Y, Z) :-
   X >= Y,
   !,
   Z = X.
max(_, Y, Y).
```

Bei Aufrufen der Form `max(X, Y, Y)` mit gebundenen Argumenten `X > Y`, wie `max(2, 1, 1)`, ist der Cut erforderlich, da sonst die zweite Regel erfolgreich aufgerufen und fälschlicherweise Y als das Maximum berechnet würde.

Aus demselben Grund darf die Bedingung `Z = X` erst nach dem Cut geprüft werden.

Will man den Cut aus der ersten Regel entfernen, so muß man in der zweiten Regel `X < Y` testen, und man kann die Bedingung `Z = X` in den Kopf der ersten Regel ziehen:

```
max(X, Y, X) :-
   X >= Y.
max(X, Y, Y) :-
   X < Y.
```

Falls man das Maximum über komplexeren Wertebereichen berechnen will, dann ist das Regelschema mit Cut aber vorzuziehen, da der komplexe Vergleich dann nur einmal ausgeführt wird.

Das folgende Prädikat `max/4` vergleicht die beiden Elemente `X` und `Y` mittels eines Prädikats `Compare/2`, das als Parameter übergeben wird:

```
/* max(+Compare, +X, +Y, ?Z) <-
      */

max(Compare, X, Y, Z) :-
   apply(Compare, [X, Y]),
   !,
   Z = X.
max(_, _, Y, Y).
```

Der Aufruf `apply(Compare, [X, Y])` wendet `Compare` an, so als ob `Compare(X, Y)` aufgerufen würde. Letzteres darf man in PROLOG nur nicht schreiben, da Variablen (`Compare`) als Prädikatensymbole nicht erlaubt sind. `apply(P, [`X_1`, ..., `X_n`])` ist äquivalent zu `call(P, `X_1`, ..., `X_n`)`.

Lexikographischer Listenvergleich:

```
geq_lex([X|_], [Y|_]) :-
   X > Y.
geq_lex([X|Xs], [X|Ys]) :-
   geq_lex(Xs, Ys).
geq_lex(_, []).

?- max(geq_lex, [2, 1], [2, 3, 0, 5], M).
M = [2, 3, 0, 5]
```

Für ein Prädikat `greater_lex/2` müßte man die letzte Regel ändern in

`greater_lex(Xs, []) :- Xs \= [].` oder in

`greater_lex([_|_], []).`

wobei "`[_|_]`" für eine Liste steht, die mindestens ein Element enthält.

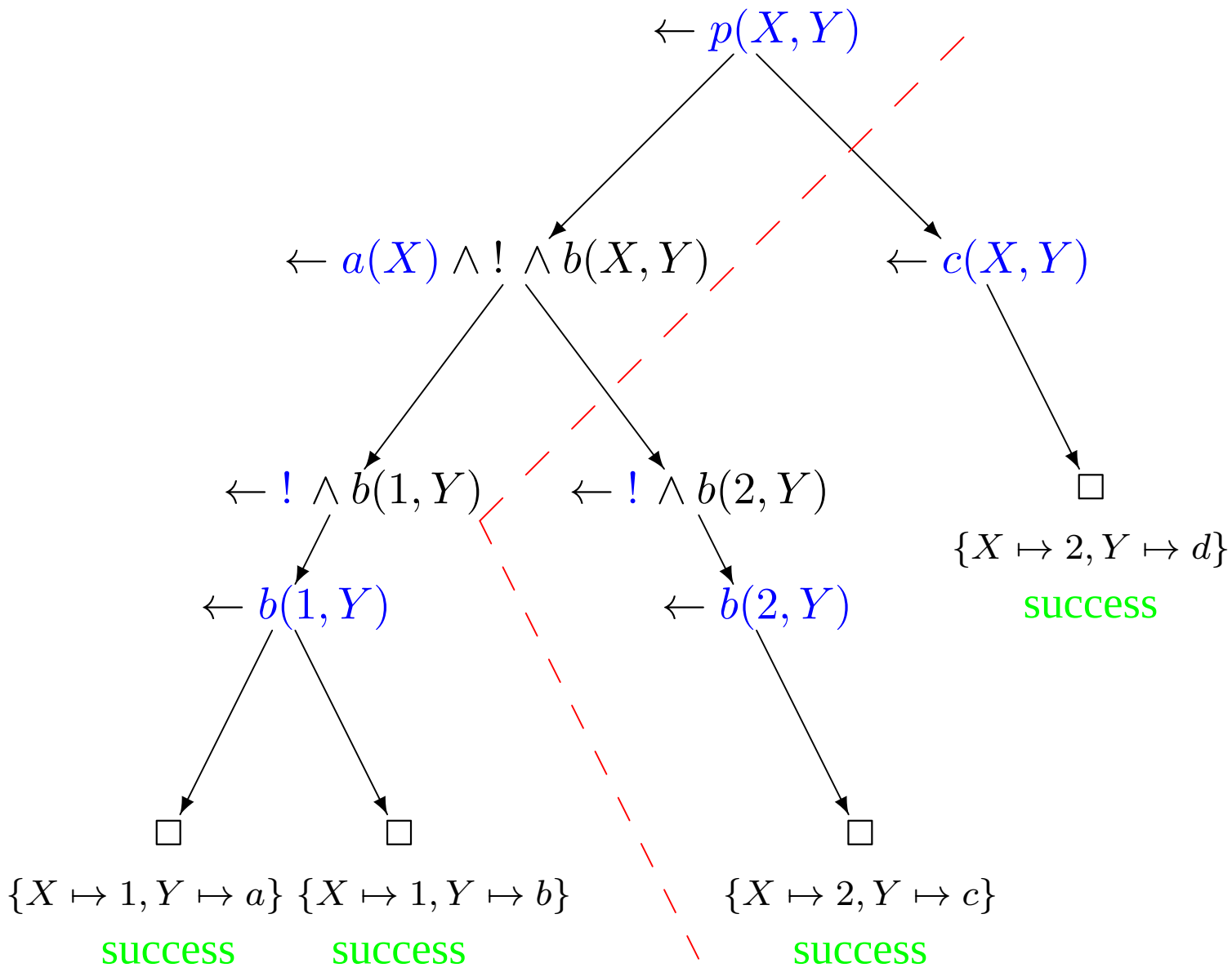
Wirkung des Cuts auf den SLD–Baum

Wir betrachten das PROLOG–Programm

```
p(X, Y) :- a(X), !, b(X, Y).
p(X, Y) :- c(X, Y).
a(1).  b(1, a).  b(1, b).
a(2).  b(2, c).
c(2, d).
```

Die folgende Anfrage liefert nur zwei Ergebnisse, da die Wahl der Regel und die von `a` zuerst vorgenommene Belegung $\texttt{X} \mapsto \texttt{1}$ vom Cut eingefroren wird:

```
?- p(X, Y).
X = 1, Y = a ;
X = 1, Y = b ;
No
```


$\leftarrow p(X,Y)$
$\leftarrow a(X) \wedge ! \wedge b(X,Y)$
$\leftarrow c(X,Y)$
$\leftarrow ! \wedge b(1,Y)$
$\leftarrow ! \wedge b(2,Y)$
$\square$
$\{X \mapsto 2, Y \mapsto d\}$
success
$\leftarrow b(1,Y)$
$\leftarrow b(2,Y)$
$\square$
$\square$
$\square$
$\{X \mapsto 1, Y \mapsto a\}$
$\{X \mapsto 1, Y \mapsto b\}$
$\{X \mapsto 2, Y \mapsto c\}$
success
success
success

Würde man in der ersten Regel den Cut entfernen, so würde man bei Backtracking 4 Antworten erhalten:

```
?- p(X, Y).
X = 1, Y = a ;
X = 1, Y = b ;
X = 2, Y = c ;
X = 2, Y = d ;
No
```

Würde man den Cut zwar beibehalten aber die beiden Regeln vertauschen, so würde man bei Backtracking drei Antworten erhalten, da dann die Antwort $\{ \mathtt{X} \mapsto 2, \mathtt{Y} \mapsto \mathtt{d} \}$ zuerst generiert würde bevor der Cut zuschlägt:

```
?- p(X, Y).
X = 2, Y = d ;
X = 1, Y = a ;
X = 1, Y = b ;
No
```

Falls `p(X, Y)` nur mit gebundenem Argument `X` aufgerufen werden soll, so kann man die Wahl der ersten Regel auch ohne Cut einfrieren, indem man die zweite Regel erweitert:

```
p(X, Y) :- a(X), b(X, Y).
p(X, Y) :- not(a(X)), c(X, Y).
```

Da der Aufruf `a(X)` für gebundenes `X` deterministisch ist, erhalten wir:

```
?- p(1, Y).
Y = a ;
Y = b ;
No
?- p(2, Y).
Y = c ;
No
```

`(If -> Then ; Else)`

Dieselbe Wirkung könnte man auch mit einem `(If -> Then ; Else)`–Statement erreichen – der Hauptfunktor ist hier "`;`":

```
p(X, Y) :-
  ( a(X) ->
    b(X, Y)
  ; c(X, Y) ).
```

Falls `If` (hier: `a(X)`) erfolgreich ist, dann kann nur `Then` (hier: `b(X, Y)`) ausgewertet werden. Andernfalls wird `Else` (hier: `c(X, Y)`) ausgewertet.

Ein `(If -> Then)`–Statement (ohne `Else`–Teil) schlägt dagegen fehl, falls der `If`–Teil fehlschlägt.

2.3 Die interne PROLOG–Datenbank

Im folgenden werden wir die interne PROLOG–Datenbank kennen lernen.

Diese enthält alle momentan verfügbaren Fakten und Regeln.

Man kann darin suchen, einfügen und löschen, und man kann sie sogar mit PROLOG–Methoden selbst analysieren.

- Laden: `consult/1, [...]`
- Anzeigen: `listing/0, listing/1`
- Einfügen und Löschen: `assert/1` bzw. `retract/1`
- Suchen: `clause/2`
- Auswerten: `call/1`

Start von SWI–PROLOG:

```
% pl
Welcome to SWI-Prolog
(Multi-threaded, 64 bits, Version 5.8.2)
Copyright (c) 1990-2009 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).
```

Laden und Anzeigen

```
?- consult('abc.pl').
% abc.pl compiled 0.00 sec, 876 bytes
Yes
?- listing.
a :- b.
a :- c.
b.
Yes
```

Nach dem Konsultieren kann man sich die Regeln und Fakten mittels `listing` anzeigen lassen – das ist aber nur sinnvoll, wenn es nicht zu viele sind.

Auswerten

Nach dem Aufruf `call(A)` versucht PROLOG das Goal `A` mit den vorhandenen Regeln und Fakten zu beweisen.

```
?- call(a).
Yes
?- call(b).
Yes
?- a.
Yes
```

Falls `A` bereits ein Prädikatensymbol enthält, so kann man zur Auswertung auch einfach nur `A` aufrufen. Ansonsten kann man mittels `A =.. [P,X1,...,Xn]` ein Goal aufbauen und dann mittels `call(A)` aufrufen, oder man ruft direkt `call(P,X1,...,Xn)` auf.

Im folgenden Beispiel werden sowohl das Prädikatensymbol, welches dreistellig sein muß, als auch seine Argumente zur Laufzeit erst von der Konsole eingelesen:

```
add(X, Y, Z) :- Z is X + Y.
sub(X, Y, Z) :- Z is X - Y.

?- read(P), current_predicate(P/3),
   read(X), read(Y), call(P, X, Y, Z).
|: add.
|: 1.
|: 2.
P = add, X = 1, Y = 2, Z = 3
```

Wenn man ein Prädikat (im Beispiel: `c/0`) aufruft, für das keine Fakten oder Regeln bekannt sind, dann bekommt man eine Fehlermeldung:

```
?- c.
ERROR: toplevel: Undefined procedure:
c/0 (DWIM could not correct goal)
```

Diesen Fehlerfall kann man vermeiden, indem man das Prädikat `c/0` in der PROLOG–Datei (mittels einer Direktive) als `dynamic` deklariert;
dann liefert der Aufruf ohne Fehler einfach nur `No` bzw. `false`:

```
:- dynamic c/0.
a :- b.
a :- c.
b.
```

```
?- c.
No
```

Laden mehrerer Dateien

```
?- ['abc.pl', 'def.pl'].
% abc.pl compiled 0.00 sec, 876 bytes
% def.pl compiled 0.00 sec, 543 bytes
Yes
```

Anstelle von `consult(File)` kann man auch `[Files]` verwenden; damit kann man gleich mehrere Dateien auf einmal laden.

In der Regel läd man eine PROLOG–Anwendung über eine Datei, die dann Direktiven "`:- [...].`" oder "`:- consult(...).`" zum Konsultieren weiterer Dateien enthält.

Einfügen

```
% pl
Welcome to SWI-Prolog (Version 5.8.2) ...
?- assert(a :- b), assert(a :- c), assert(c).
Yes
?- listing(a).
:- dynamic a/0.
a :- b.
a :- c.
Yes
```

Eine Regel wird hier als Term mit dem Funktor `:-` angegeben:
die Infix–Notation `a :- b` ist äquivalent zu `:-(a, b)`.
Das Prädikat `a/0` ist `dynamic`, da Regeln dazu eingefügt wurden.

Löschen

```
?- retract(a :- c).
Yes
?- listing.
a :- b.
b.
Yes
?- a.
Yes
?- retract(b).
Yes
```

Nach dem Löschen der Regel `a :- c` kann man `a` immer noch aus den verbleibenden Regeln und Fakten herleiten, nach dem Löschen von `b` nicht mehr.

`retractall(X)` löscht alle Fakten bzw. Regeln, die mit `X` unifizieren.

Suchen

```
tc(X, Z) :- arc(X, Z).
tc(X, Z) :- arc(X, Y), tc(Y, Z).
arc(a, b).  arc(b, c).
```

Mittels `clause/2` kann man nach Regeln (bzw. Fakten) in der PROLOG–Datenbank suchen. Dabei wird nichts neu abgeleitet.

```
?- clause(tc(X, Z), Body).
Body = arc(X, Z) ;
Body = arc(X, Y), tc(Y, Z)
?- clause(Head, Body).
ERROR: Arguments are not sufficiently instantiated
```

Aus dem ersten Argument muß das Kopf–Prädikatensymbol (mit Stelligkeit) der gesuchten Regeln hervorgehen.

Man kann allerdings mittels `current_predicate/1` alle dem System momentan bekannten Prädikatensymbole sowie deren Stelligkeit ermitteln. `current_predicate/2` bestimmt auch ein zugehöriges Atom.

```
?- current_predicate(Predicate).
Predicate = tc/2 ;
Predicate = arc/2

?- current_predicate(Predicate, Atom).
Predicate = tc,  Atom = tc(_G2305, _G1606) ;
Predicate = arc, Atom = arc(_G2305, _G1606)
```

Damit kann man dann alle dem System momentan bekannten Regeln ermitteln – und z.B. bei Bedarf analysieren.

```
predicate_to_clause(Predicate/N, Head :- Body) :-
   current_predicate(Predicate, Head),
   functor(Head, _, N),
   catch( clause(Head, Body), _, fail ).

?- predicate_to_clause(P, C).
P = tc/2,  C = tc(X, Z) :- arc(X, Z) ;
P = tc/2,  C = tc(X, Z) :- arc(X, Y), tc(Y, Z) ;
P = arc/2, C = arc(a, b) :- true ;
P = arc/2, C = arc(b, c) :- true
```

Der Funktor und die Stelligkeit eines Atoms werden dabei mittels `functor/3` bestimmt. Fehlermeldungen können mittels `catch/3` abgefangen werden.

Seiteneffekte

Man kann auch mittels eines *Failure–Driven–Loops* alle Antworten für eine Anfrage `?- p(X)` in die interne PROLOG–Datenbank einfügen:

```
loop :-
   p(X), assert(a(X)), writeln(X),
   fail.
loop.
```

Die erste Regel scheitert zwar für alle Antworten für den Aufruf `p(X)` bei `fail`, aber es werden vor dem Backtracking immer die entsprechenden Fakten `a(X)` in die PROLOG–Datenbank eingefügt (Seiteneffekte). Auch die Konsolenausgaben mittels "`writeln(X)`" können natürlich nicht zurückgenommen werden.

Am Ende ist das Prädikat `loop` aufgrund der zweiten Regel erfolgreich.

Beispiel:

Der Aufruf von `loop` setzt die bekannten Fakten für `p` in entsprechende Fakten für `a` um:

```
p(1).
p(2).

?- loop.
1
2
Yes
?- a(X).
X = 1;
X = 2;
No
```

Denselben Effekt könnte man eleganter mittels des Meta–Prädikats `forall/2` erzielen.

```
loop :-
   p(X), assert(a(X)), writeln(X),
   fail.
loop.
```

Der Aufruf `forall(G1, G2)` generiert mittels des Goals `G1` über Backtracking Variablenbelegungen und ruft für diese das Goal `G2` auf:

```
loop :-
   forall( p(X), ( assert(a(X)), writeln(X) ) ).
```

Globale Variablen im DDK

Mittels `assert/1` und `retract/1` kann man globale Variablen mit destruktivem Assignment simulieren.

Eine solche Variable ist durch eine Referenz gegeben. Sie wird durch ein Fakt `global_variable(Ref, Value)` in der PROLOG–Datenbank realisiert.

- Mittels `global_variable_set(Ref, Value)` wird eine globale Variable `Ref` mit einem Wert `Value` initialisiert bzw. aktualisiert.
- Mittels `global_variable_get(Ref, Value)` wird der Wert der globalen Variable `Ref` abgefragt.
- Mittels `global_variable_destroy(Ref, Value)` wird der Wert der globalen Variable `Ref` abgefragt, und sie wird deaktiviert.

Dieses Konzept sollte man eigentlich nur für sich wenig ändernde Variablen – wie z.B. Konfigurationsparameter (Flags) – verwenden.

Mittels `global_variable_set(Ref, Value)` wird eine globale Variable zur Referenz `Ref` auf den neuen Wert `Value` aktualisiert; falls sie noch nicht existierte, so wird sie deklariert und mit `Value` initialisiert.

```
% global_variable_set(Ref, Value) <-

global_variable_set(Ref, Value) :-
   retract(global_variable(Ref, _)),
   !,
   asserta(global_variable(Ref, Value)).
global_variable_set(Ref, Value) :-
   asserta(global_variable(Ref, Value)).
```

Falls schon ein Wert vorhanden ist, so wird dieser gelöscht und einer neuer Wert gespeichert. Andernfalls wird nur der neue Wert gespeichert.

Mittels `global_variable_destroy/2` wird der Wert `Value` der globalen Variable zur Referenz `Ref` abgefragt, und die globale Variable wird deaktiviert. Falls kein Wert gespeichert war, so wird `0` zurückgegeben.

```
% global_variable_destroy(Ref, Value) <-

global_variable_destroy(Ref) :-
   global_variable_destroy(Ref, _).

global_variable_destroy(Ref, Value) :-
   retract(global_variable(Ref, Value)),
   !.
global_variable_destroy(_, 0).
```

`global_variable_destroy/1` deaktiviert die globale Variable lediglich.

Damit könnte man im DDK die Summenberechnung $Sum = \sum_{X=I}^{J} X$ in der von der prozeduralen/objekt–orientierten Programmierung bekannten Art und Weise implementieren:

```
% sum_with_counter(I, J, Sum) <-

sum_with_counter(I, J, Sum) :-
   global_variable_set(sum_with_counter, 0),
   ( for(X, I, J) do
        global_variable_get(sum_with_counter, A),
        B is A + X,
        global_variable_set(sum_with_counter, B) ),
   global_variable_destroy(sum_with_counter, Sum).
```

Der Aufruf `for(X, I, J) do Goal` mit dem Meta–Prädikat `for-do` realisiert eine Schleife: er ruft `Goal` für alle Werte `X` von `I` bis `J` auf.

Es gibt auch das Prädikat `global_variable_add/2` zur Addition eines Wertes `Value` auf eine globale Variable zu einer Referenz.

```
% global_variable_add(Ref, Value) <-

global_variable_add(Ref, Value) :-
   retract(global_variable(Ref, Value_1)),
   !,
   Value_2 is Value_1 + Value,
   asserta(global_variable(Ref, Value_2)).
global_variable_add(Ref, Value) :-
   asserta(global_variable(Ref, Value)).
```

Falls bisher noch kein Wert vorhanden war, so wird automatisch der angegebene Wert gespeichert.

Damit könnte man die Summenberechnung kürzer formulieren.

```
% sum_with_counter(I, J, Sum) <-

sum_with_counter(I, J, Sum) :-
   ( for(X, I, J) do
        global_variable_add(sum_with_counter, X) ),
   global_variable_destroy(sum_with_counter, Sum).
```

Die Initialisierung der globalen Variable zur Referenz `sum_with_counter` kann entfallen – wenn man sicher ist, daß sie woanders nicht benutzt wurde.

Außerdem wurden Abfrage, Addition und Aktualisierung als ein einziges Prädikat `global_variable_add/2` extrahiert (Refaktorisierung).

Im DDK gibt es ein Infix–Auswertungsprädikat `<==/2` als Erweiterung von `is/2`, das Referenzen auf globale Variablen in (arithmetischen) Ausdrücken geeignet behandeln kann.

Damit könnte man die Summenberechnung sogar in der von der prozeduralen Programmierung bekannten Schreibweise implementieren:

```
% sum_with_counter(I, J, Sum) <-

sum_with_counter(I, J, Sum) :-
   @sum <== 0,
   ( for(X, I, J) do
        @sum <== @sum + X ),
   Sum <== @sum.
```

Hier wird die globale Variable zur Referenz `@sum` am Ende nicht deaktiviert. Deswegen wird sie am Anfang immer mit `0` initialisiert.

Bei einem Aufruf `X <== Exp` wird eine Referenz – oben `@sum` – in einem Ausdruck `Exp` durch den Wert der entsprechenden globalen Variable ersetzt.

Sei `Value` das Resultat der Auswertung des aus `Exp` entstandenen Ausdrucks.

- Wenn `X` nur aus einer Referenz besteht, dann wird `Value` der neue Wert der entsprechenden globalen Variable;
- andernfalls wird `X` mit `Value` unifiziert.

Hierbei werden die globalen Variablen bei der ersten Zuweisung deklariert und initialisiert – oben mittels `@sum <== 0`.

Das Endresultat kann dann wieder auf eine (klassische) Variable zugewiesen werden – oben mittels `Sum <== @sum`.

Man kann anstelle des Meta–Prädikats `for-do` natürlich auch einen Failure–Driven–Loop verwenden.

```
% sum_with_counter(I, J, Sum) <-

sum_with_counter(I, J, Sum) :-
   @sum <== 0,
   ( between(I, J, X),
        @sum <== @sum + X, fail
   ; Sum <== @sum ).
```

Dabei generiert `between(I, J, X)` bei Backtracking die ganzen Zahlen zwischen `I` und `J`.

Zur Addition einer vorgegebenen Liste `Xs` von Zahlen verwendet man `member(X, Xs)` als Generator.

```
sum_with_counter(Xs, Sum) :-
   @sum <== 0,
   ( member(X, Xs),
        @sum <== @sum + X, fail
   ; Sum <== @sum ).
```

Eleganter könnte man das mittels des Meta–Prädikats `forall/2` realisieren.

```
sum_with_counter(Xs, Sum) :-
   @sum <== 0,
   forall( member(X, Xs),
      @sum <== @sum + X ),
   Sum <== @sum.
```

Kritische Bewertung

Algorithmen mit Variablen, deren Werte sich häufig ändern, sollte man rekursiv unter Verwendung von frischen Variablen realisieren, und nicht mit globalen Variablen und destruktivem Assignment.

- `assert` und `retract` sind teure und nicht–deklarative Operationen.
- Deswegen ist für die fortgeschrittene Programmierung von einer derartigen Implementierung von Schleifen – wie bei der Summenberechnung – mit globalen Variablen und destruktivem Assignment abzuraten.
- Stattdessen sollte man für die Summenberechnung die – bereits gezeigte – rekursive Implementierung mittels Akkumulatoren verwenden.
- Diese verwendet anstelle des destruktiven Assignments eine frische Variable zur Speicherung der aktualisierten Zwischensumme.

2.4 Datenbanken und PROLOG

Man kann in PROLOG mittels ODBC auf relationale Datenbanken zugreifen oder die Daten direkt in der internen PROLOG–Datenbank ablegen.

Das in PROLOG implementierte deduktive Mini–DBMS DDBASE realisiert eine komfortable Kopplung zwischen den beiden Systemen. Dazu können

- Tabellen,
- das Data Dictionary,
- Integritätsbedingungen und
- Anfragen

behandelt werden.

PROLOG kann auch als Datenbanksprache verwendet werden:

- Man kann in PROLOG die *Tabellen* einer relationalen Datenbank repräsentieren. Die Tupel einer Tabelle werden zu PROLOG–Fakten mit demselben Prädikatensymbol – gewöhnlich dem Namen der Tabelle.
- Das *Data Dictionary* einer relationalen Datenbank kann ebenfalls in Form von PROLOG–Fakten repräsentiert werden.
  Wir werden sehen wie dies mit Hilfe von PROLOG–Termen gemacht werden kann, die einer XML–Repräsentation des Data Dictionary entsprechen.
- *Integritätsbedingungen* und *Sichten* (Views) können als PROLOG–Regeln repräsentiert werden. Konjunktive *Anfragen* (Queries) werden in Form von PROLOG–Goals gestellt, die dann auf der Basis der PROLOG–Regeln ausgewertet werden.

2.4.1 ODBC–Zugriff auf eine relationale Datenbank

Mittels `odbc_connect/3` kann man sich mit einem relationalen Datenbanksystem verbinden, um später ODBC–Anfragen stellen zu können.

```
/* odbc_connect(+DSN, +Connection) <-
      Creates a Connection to the relational
      data source DSN
       - as user seipel
       - with password dietmar. */

odbc_connect(DSN, Connection) :-
   odbc_connect(DSN, _, [
      user(seipel), password(dietmar),
      alias(Connection), open(once) ]).
```

Oft kann man z.B. `odbc_connect(mysql, mysql)` aufrufen.

Mittels `odbc_query/2` und `odbc_query/4` kann man ODBC–Anfragen in String–Form an ein relationales Datenbanksystem stellen:

```
% odbc_string_query_employee_project(Row) <-

odbc_string_query_employee_project(Row) :-
   odbc_query(mysql, 'use company;'),
   concat([ 'select e.lname, p.pname ',
      'from employee e, works_on w, project p ',
      'where e.ssn = w.essn ',
      'and w.pno = p.pnumber;' ], Sql),
   Options = [types([atom, atom])],
   odbc_query(mysql, Sql, Row, Options).
```

Dabei können die gewünschten PROLOG–Datentypen für die selektierten SQL–Attribute angegeben werden.

Die Anfrageergebnisse werden als Terme mit dem Funktor `row` zurückgegeben:

```
?- odbc_string_query_employee_project(Row).

Row = row('Zelaya', 'Computerization') ;
Row = row('Jabbar', 'Computerization') ;
Row = row('Wallace', 'Newbenefits') ;
...
```

Oft wird man mittels `findall/3` direkt die Liste all dieser Terme bestimmen:

```
?- findall( Row,
      odbc_string_query_employee_project(Row),
      Rows ).
Rows = [row('Zelaya', 'Computerization'), ...]
```

Mittels `ddbase_query/3` kann man ODBC–Anfragen in Term–Form an ein relationales Datenbanksystem stellen:

```
% odbc_term_query_employee_project(Tuple) <-

odbc_term_query_employee_project(Tuple) :-
   Sql = [
      use:[company],
      select:[e^lname, p^pname],
      from:[employee-e, works_on-w, project-p],
      where:[e^ssn = w^essn and w^pno = p^pnumber] ],
   ddbase_query(odbc(mysql), Sql, Tuple).
```

Aus dem Term `Sql` werden SQL–Statements erzeugt, die dann intern auf der Basis von `odbc_query/2,4` ausgewertet werden.

Die folgende Regel speichert alle Antwort–Tupel `[X1, X2]` als PROLOG–Fakten der Form `employee_project(X1, X2)`. Vorher wird die PROLOG–Datenbasis noch von allen bisherigen `employee_project/2`–Fakten bereinigt.

```
odbc_term_query_employee_project :-
   retractall(employee_project(_, _)),
   forall( odbc_term_query_employee_project(Tuple),
      ( writeln(Tuple),
        Tuple = [X1, X2],
        assert(employee_project(X1, X2) ) ) ),
   listing(employee_project/2).
```

`forall(Goal_1, Goal_2)` führt `Goal_2` für alle Belegungen aus, die `Goal_1` produziert.

Laden einer MySQL–Tabelle in ein PROLOG–Modul:

```
?- mysql_table_to_prolog_module(company:employee, c),
   c:listing.

:- dynamic employee/7.

employee('Smith',   '1111', '09-JAN-55', 'M', '30000',   '3333', 5).
employee('Wong',    '3333', '08-DEC-45', 'M', '40000',   '5555', 5).
employee('Zelaya',  '2222', '19-JUL-58', 'F', '25000',   '7777', 4).
employee('Wallace', '7777', '20-JUN-31', 'F', '43000',   '5555', 4).
employee('Narayan', '6666', '15-SEP-52', 'M', '38000',   '3333', 5).
employee('English', '4444', '31-JUL-62', 'F', '25000',   '3333', 5).
employee('Jabbar',  '8888', '29-MAR-59', 'M', '25000',   '7777', 4).
employee('Borg',    '5555', '10-NOV-27', 'M', '55000', '$null$', 1).

Yes
```

2.4.2 Das deduktive Mini–DBMS DDBASE

Wir können aus DDBASE heraus über ODBC auf relationale Datenbanken zugreifen.

- Das Schema einer Datenbank kann in XML–Form aus dem Data Dictionary extrahiert werden.
- Aufgrund von Schemabeschreibungen können Tabellen in ein PROLOG–Modul geladen und dort verwaltet (Anfragen, Updates) und untersucht werden (auf Anomalien, Integritätsbedingungen, etc.).
- Basierend auf dem Datenbankschema kann man in DDBASE atomare und konjunktive Anfragen in verschiedenen Formen (DATALOG, Field Notation, SQL–artig) direkt an die Datenbank stellen.

Das Datenbankschema – Data Dictionary

Zur Schemabeschreibung verwenden wir – unter anderem – PROLOG–Fakten der folgenden Kurzform:

```
schema( table:[name:T, database:D]:[
   attributes:[A1,...,An],
   primary_key:[B1,...,Bm],
   not_null:[C1,...,Ck],
   foreign_keys:[A->Tf:Af, ...]] ).
```

Diese enthalten die Attributliste, den Primärschlüssel, sowie die Not Null– und die Fremdschlüsselbedingungen.

Basierend auf den Integritätsbedingungen aus diesen Schema–Fakten können in DDBASE Updates in der PROLOG–Datenbank kontrolliert werden.

Data Dictionary als PROLOG–Fakten (Kurzform)

```
schema( table:[name:employee, database:company]:[
   attributes:['LNAME', 'SSN', 'BDATE', ..., 'SUPERSSN', 'DNO'],
   primary_key:['SSN'],
   foreign_keys:[
      'SUPERSSN'->employee:'SSN', 'DNO'->department:'DNUMBER']] ).
schema( table:[name:works_on, database:company]:[
   attributes:['ESSN', 'PNO', 'HOURS'],
   primary_key:['ESSN', 'PNO'],
   foreign_keys:[
      'ESSN'->employee:'SSN', 'PNO'->project:'PNUMBER']] ).
schema( table:[name:department, database:company]:[
   attributes:['DNAME', 'DNUMBER', 'MGRSSN', 'MGRSTARTDATE'],
   primary_key:['DNUMBER'], not_null:['DNUMBER', 'DNAME'],
   foreign_keys:[
      'MGRSSN'->employee:'SSN']] ).
schema( table:[name:project, database:company]:[
   attributes:['PNAME', 'PNUMBER', 'PLOCATION', 'DNUM'],
   primary_key:['PNUMBER'], not_null:['PNUMBER', 'PNAME'],
   foreign_keys:[
      'DNUM'->department:'DNUMBER']] ).
```

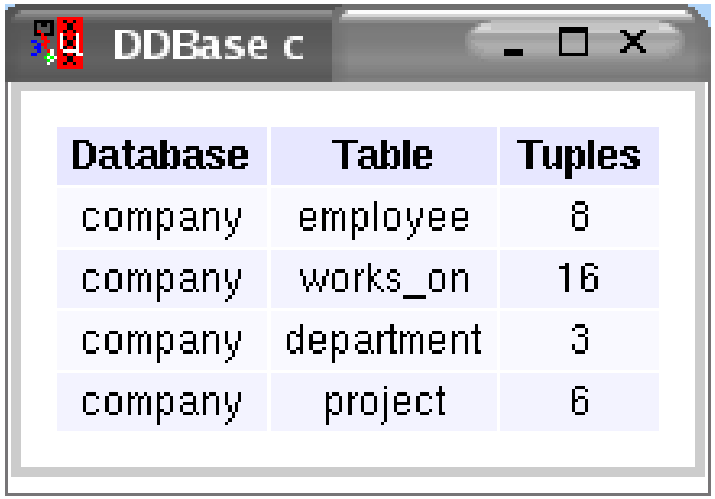
Data Dictionary als PROLOG–Term (Langform)

```
table:[name:employee, database:company]:[
   attribute:[name:'LNAME',
      type:'varchar(15)', is_nullable:'NO']:[],
   attribute:[name:'SSN', ...]:[],
   attribute:[name:'BDATE', ...]:[],
   ...
   attribute:[name:'SUPERSSN', ...]:[],
   attribute:[name:'DNO', ...]:[],
   primary_key:[ attribute:[name:'SSN']:[] ],
   ...
   foreign_key:[ attribute:[name:'DNO']:[],
      references:[table:'department]:[
         attribute:[name:'DNUMBER']:[] ] ] ]
```

Falls eine Datenbankverbindung `mysql` offen ist, so kann man über ODBC die Schema–Fakten zur Datenbank `company` in das PROLOG–Modul `c` laden:

```
?- ddbase_load(odbc(mysql), company, c).
```

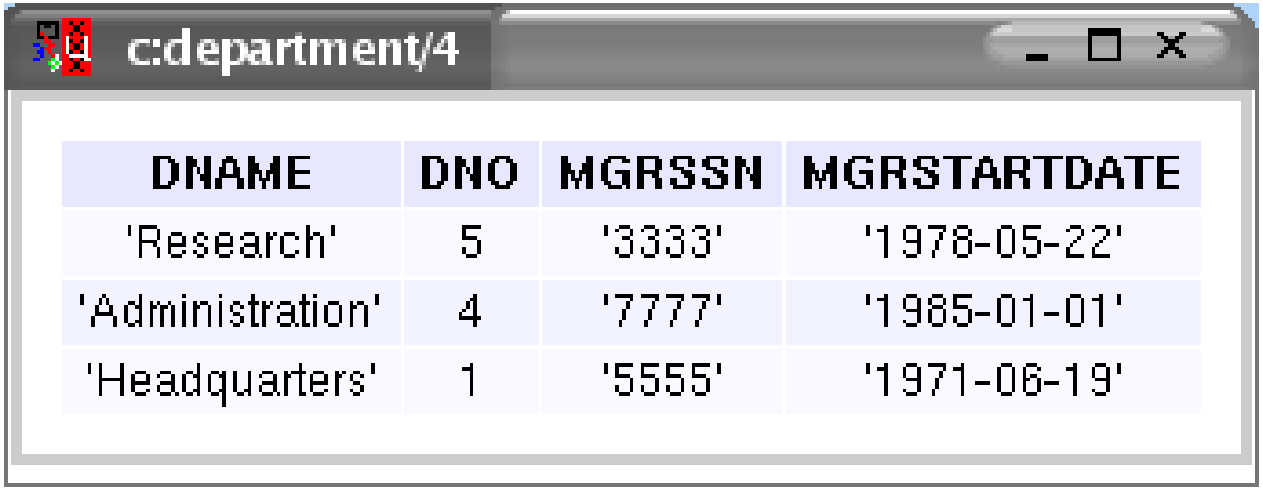
Die zugehörigen Tabellen kann man dann mit `ddbase_load_tables(c)` laden. Mittels `ddbase_show_tables(+Module)` erhält man einen Überblick über die Tabellen eines PROLOG–Moduls:



Database	Table	Tuples
company	employee	8
company	works_on	16
company	department	3
company	project	6

Anzeigen von einzelnen DDBASE–Tabellen:

```
?- ddbase_facts_to_display(c:department/4).
```



DNAME	DNO	MGRSSN	MGRSTARTDATE
'Research'	5	'3333'	'1978-05-22'
'Administration'	4	'7777'	'1985-01-01'
'Headquarters'	1	'5555'	'1971-06-19'

Die Attribute werden den `schema`–Fakten entnommen. Falls es für eine Tabelle kein `schema`–Fakt gibt, so werden die Attributfelder leer gelassen.

Anomalie–Tests

Man kann z.B. mittels rekursiver Regeln die indirekten Vorgesetzten ermitteln (was in SQL nicht geht) und testen, ob jemand sein eigener indirekter Vorgesetzter ist:

```
superior(X, Y) :-
   direct_supervisor(X, Y).
superior(X, Y) :-
   direct_supervisor(X, Z), superior(Z, Y).

direct_supervisor(E, S) :-
   employee(_, E, _, _, _, S, _).

?- c:superior(X, X).
No
```

Außerdem kann man z.B. testen,

- ob der Vorgesetzte eines Mitarbeiters in einer anderen Abteilung beschäftigt ist, und
- ob ein Mitarbeiter an einem Projekt einer anderen Abteilung arbeitet.

```
employee_department(E, D) :-
   employee(_, E, _, _, _, _, D).

supervisor_is_in_another_department(E-De, S-Ds) :-
   employee_department(E, De),
   direct_supervisor(E, S), employee_department(S, Ds),
   De \= Ds.

employee_works_on_outside_project(E-De, P-Dp) :-
   employee_department(E, De),
   works_on(E, P, _), project(_, P, _, Dp),
   De \= Dp.
```

Meta–Prädikate

Mittels des generischen Meta–Prädikats `transitive_closure` kann man in DDBASE die zwei Regeln mit Rekursion

```
superior(X, Y) :-
   direct_supervisor(X, Y).
superior(X, Y) :-
   direct_supervisor(X, Z), superior(Z, Y).
```

für `superior` durch eine einzige, nicht–rekursive, viel kompaktere und abstraktere Regel ersetzen:

```
superior(X, Y) :-
   transitive_closure(direct_supervisor, X, Y).
```

Integritätsbedingungen

- Primärschlüsselbedingung für Employee:

```
primary_key_violation(employee, X, Y) :-
   X = employee(_, SSN, _, _, _, _, _), call(X),
   Y = employee(_, SSN, _, _, _, _, _), call(Y),
   X \= Y.
```

- Fremdschlüsselbedingung für Employee:

```
foreign_key_violation(
      employee('DNO'), department('DNUMBER'), X) :-
   X = employee(_, _, _, _, _, _, DNO), call(X),
   not(department(_, DNO, _, _)).
```

Auch hier wird zur kompakteren Formulierung ein Meta–Prädikat, nämlich `call/1`, verwendet (um nicht wiederholt komplette `employee/7`–Atome zitieren zu müssen).

Field Notation (FN)

Wie in anderen *Programmiersprachen* werden die Argumente t_i eines Atoms $p(t_1, \ldots, t_n)$ bei Position übergeben. Z.B. ist in

```
works_on(S, P, H),
```

die erste Position $t_1 =$ `S` die Sozialversicherungsnummer (social security number, SSN) eines Angestellten, der an einem Projekt mit der Nummer $t_2 =$ `P` (zweite Position) für $t_3 =$ `H` Stunden (hours, dritte Position) gearbeitet hat.

Im *Datenbank*–Kontext von DDBASE können wir aber die Field Notation benutzen, um auf Argumente – auf eine abstraktere Art und Weise – mittels der entsprechenden Attributnamen aus dem Datenbankschema zugreifen zu können.

```
works_on('PNO':P, 'ESSN':S)
```

drückt aus, daß der Angestellte mit der Sozialversicherungsnummer `S` am Projekt mit der Nummer `P` gearbeitet hat – unabhängig von der Reihenfolge der Argumente –, und es is nicht notwendig die Stundenzahl aufzulisten.

Semantische Bedingungen in Field Notation (FN):

- Kein Angestellter sollte mehr als sein Manager verdienen:

```
trigger(salary, X, Y) :-
   employee('SSN':X, 'SALARY':S1, 'SUPERSSN':Y),
   employee('SSN':Y, 'SALARY':S2),
   S1 > S2.
```

- Welcher Angestellte arbeitet an einem fremden Projekt ?

```
trigger(employee_works_on_outside_project, E, P) :-
   works_on('ESSN':E, 'PNO':P),
   employee('SSN':E, 'DNO':D1),
   project('PNUMBER':P, 'DNUM':D2),
   D1 \= D2.
```

FN abstrahiert von den Argumentpositionen: `employee('SSN':E, 'DNO':D1)` entspricht `employee(_, E, _, _, _, _, D1)`.

Integritätsbedingungen in DDBASE

Schließlich kann man die Integritätsbedingungen wie

- Primärschlüsselbedingungen,
- Fremdschlüsselbedingungen, und
- Not–Null–Bedingungen

für ein DDBASE–Modul mittels des Aufrufs

```
ddbase_incorrect(Module)
```

testen und erhält eine Liste der Verletzungen.

SQL–Anfragen vs. PROLOG

Department–Namen der Angestellten in SQL:

SELECT LNAME, DNAME
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DNO = DEPARTMENT.DNUMBER

In PROLOG:

```
department_name(Lname, Dname) :-
   employee(Lname, _, _, _, _, _, DNO),
   department(Dname, DNO, _, _).
```

Nested–Loop–Join

- Für alle `employee`–Fakten werden alle `department`–Fakten durchsucht, um Join–Partner zu finden.
- Die Join–Bedingung wird durch gleiche Variablen (`DNO`) in den entsprechenden Argumentpositionen ausgedrückt.
- Die selektierten Attribute formen das Kopfatom der PROLOG–Regel.

Name des Vorgesetzten in SQL:

SELECT E.LNAME, S.LNAME
FROM EMPLOYEE E, EMPLOYEE S
WHERE E.SUPERSSN = S.SSN

In PROLOG:

```
supervisor_name(E_Name, S_Name) :-
   employee(E_Name, _, _, _, _, SSN, _),
   employee(S_Name, SSN, _, _, _, _, _).
```

Aggregation

Das folgende SQL–Statement findet für alle Projekte die Summe der geleisteten Arbeitsstunden:

```
SELECT PNAME, SUM(HOURS)
FROM PROJECT, WORKS_ON
WHERE PROJECT.PNUMBER = WORKS_ON.PNO
GROUP BY PNAME
```

Im DDK behandeln wir die Aggregation mit einem drei–stelligen Meta–Prädikat `ddbase_aggregate/3`:

```
?- ddbase_aggregate( [Pname, sum(Hours)],
      ( project(Pname, PNO, _, _),
        works_on(_, PNO, Hours) ),
      Pairs ).
```

Updates in DDBASE

Operationen:

- `ddbase_insert(+Module:Atom)`
- `ddbase_delete(+Module:Atom)`

Beim Update werden die Intergritätsbedingungen getestet.

```
?- ddbase_insert(c:department(
       'Computers', 1, '1234', '2007-04-16')).
Atom not inserted, because of IC violation: ...
```

Im Erfolgsfall würde das Fakt in das PROLOG–Modul `c` eingefügt mittels `assert(c:department('Computers', 1, '1234', ...))`.

Nach einer Einfügung oder Löschung in der Datenbank werden die Primär– und Fremdschlüsselbedingungen aus dem Data Dictionary geprüft.

```
?- ddbase_insert(c:works_on('6666', 10, 3)),
   ddbase_insert(c:works_on('6666', 10, 4)),
   ddbase_delete(c:works_on('6666', 10, 3)).
```

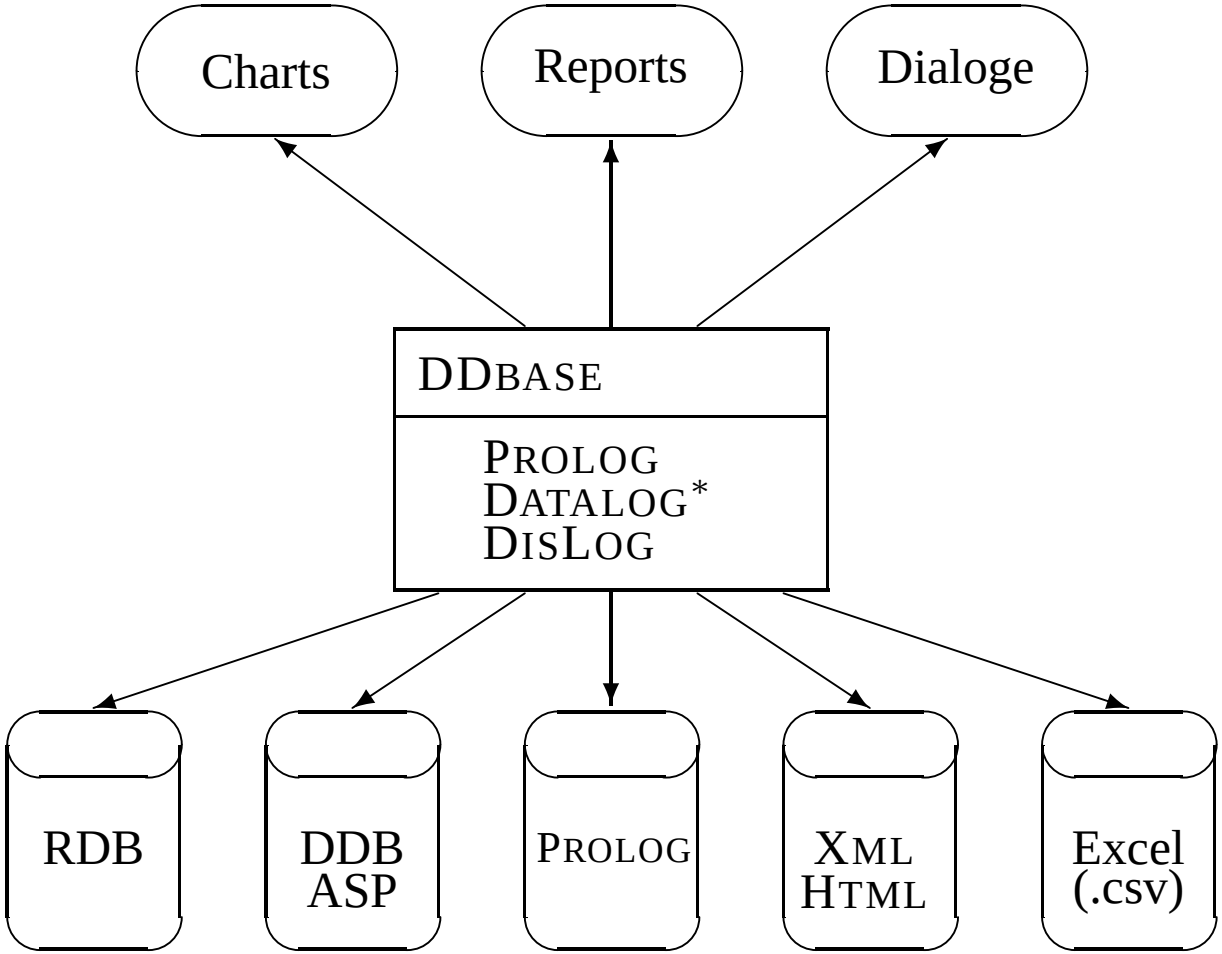
Der zweite Update wird zurückgewiesen, da er die Primärschlüsselbedingung von `works_on` verletzen würde.

Alle Tuple aus allen Relationen einer Datenbank können in einem Schritt gelöscht werden. Das Data Dictionary bleibt dabei unverändert.

```
   ?- ddbase_drop_database(c).
```

DDBASE als Mediator

Schnittstellen zu vielen anderen Systemen



ddbase_call/3

Man kann aus DDBASE heraus auf die verschiedensten Typen von Datensammlungen zugreifen:

```
ddbase_call(Type, Source, Object).
```

Falls eine Datenbankverbindung `Connection` offen ist, so kann man über ODBC auf die Tupel einer Datenbankrelation zugreifen:

```
?- Connection = mysql,
   Type = odbc(Connection),
   Source = company:department,
   ddbase_call(Type, Source, Tuple).

Tuple = ['Headquarters', 1, '1111', date(1971,6,19)] ;
Tuple = ['Administration', 4, '3333', date(1985,1,1)] ;
Tuple = ['Research', 5, '2222', date(1978,5,22)]
```

Man kann auch auf Excel–Dateien im `csv`–Format zugreifen.

```
Name:MatNr
Smith:1234
Miller:5678
```

Der Typ `csv(Separator)` gibt dabei den Separator der Datei an:

```
?- Type = csv(:),
   Source = 'test.csv',
   ddbase_call(Type, Source, Tuple).

Tuple = ['Name', 'MatNr'] ;
Tuple = ['Smith', '1234'] ;
Tuple = ['Miller', '5678']
```

ddbase_call/2

Der Aufruf `ddbase_call(odbc(Connection),Database:Goal)` wertet ein atomares Goal in einer Datenbank über eine ODBC–Verbindung aus:

```
?- ddbase_call( odbc(mysql),
      company:works_on(SSN, PNO, Hours) ),
SSN = '1111', PNO = 20, Hours =  0.0 ;
SSN = '2222', PNO =  2, Hours = 10.0 ;
SSN = '2222', PNO =  3, Hours = 10.0 ;
SSN = '3333', PNO = 20, Hours = 15.0 ;
...
SSN = '8888', PNO = 30, Hours =  5.0

?- ddbase_call( odbc(mysql),
      company:works_on('1111', PNO, Hours) ).
PNO = 20, Hours = 0.0 ;
No
```

Durch vorbelegte Argumente kann man bereits in der Datenbank selektieren.

Durch die Verbindung von `ddbase_aggregate/3` und `ddbase_call/2` kann man strukturierte Reports erstellen, die in SQL nicht möglich sind:

```
?- ddbase_aggregate(
       [ SSN,
         list_to_comma_atom((PNO, Hours)),
         sum(Hours) ],
       ddbase_call( odbc(mysql),
          company:works_on(SSN, PNO, Hours) ),
       Tuples ),
   Attributes = ['SSN', 'Workload', 'Hours'],
   xpce_display_table(Attributes, Tuples).
```

SSN	Workload	Hours
1111	(20, 0.0)	0
2222	(2, 10.0), (3, 10.0)	20
3333	(20, 15.0), (30, 20.0)	35
4444	(1, 32.5), (2, 7.5)	40
5555	(3, 40.0)	40
6666	(1, 20.0), (2, 20.0)	40
7777	(10, 10.0), (30, 30.0)	40
8888	(10, 35.5), (30, 5.0)	40.5

Das Aggregationsprädikat `list_to_comma_atom/2` wandelt eine Liste `[(2,10.0),(3,10.0)]` in ein PROLOG–Atom `'(2,10.0),(3,10.0)'` ohne Klammern "`[]`" um, so daß es in einer XPCE–Tabelle schöner dargestellt werden kann.

Anfragen in Field Notation: `ddbase_call/4`

In jedem Fall ist eine gewisse Kenntnis der angefragten Relationenschemata erforderlich. Man kann aber – wie in SQL– über Attributnamen selektieren, ohne die genaue Attribut–Reihenfolge und –Anzahl einer Tabelle zu kennen:

```
?- ddbase_call( odbc(mysql),
      company:employee, ['SEX':'F'], Pairs ).

Pairs = [..., 'SSN':'3333', ..., 'SEX':'F', ...] ;
Pairs = [..., 'SSN':'6666', ..., 'SEX':'F', ...] ;
Pairs = [..., 'SSN':'7777', ..., 'SEX':'F', ...] ;
No
```

Die Selektion und die Ergebnistupel werden als Assoziationslisten von Paaren `A:V` aus Attributnamen `A` und zugehörigen Werten `V` repräsentiert.

Speziell für Tabellen mit sehr vielen Attributen ist es sehr hilfreich, daß man kein Ergebnisatom mit der korrekten Anzahl von Argumenten angeben muß.

Aus den Antworten auf eine Anfrage kann man Terme der Form `T:As:[]` bilden, welche XML–Elemente mit Attributen – aber ohne Unterelemente – repräsentieren:

```
?- findall( employee:Pairs:[],
      ddbase_call( odbc(mysql),
         company:employee, ['SEX':'F'], Pairs ),
      Es ),
   dwrite(xml, user, employees:Es).

<employees>
   <employee ... SSN="3333" ... SEX="F" .../>
   <employee ... SSN="6666" ... SEX="F" .../>
   <employee ... SSN="7777" ... SEX="F" .../>
</employees>
```

Die Liste `Es` all dieser Terme kann man mit einem weiteren Tag einpacken und in XML ausgeben. Dabei steht ein Term `T:Es` (oben: `T=employees`) für ein XML–Element mit Unterelementen – aber ohne Attributliste.

Join–Anfragen in Term–Form: `ddbase_query/3`

Join–Anfragen können wir in Term–Form analog zu SQL–Anfragen formulieren. Der Aufruf

```
ddbase_query(odbc(Connection), Sql, Tuple)
```

beantwortet eine Anfrage `Sql` über eine offene Datenbankverbindung:

```
?- Sql = [
       use:[company],
       select:[e^lname, p^pname],
       from:[employee-e, works_on-w, project-p],
       where:[e^ssn = w^essn and w^pno = p^pnumber] ],
   ddbase_query(odbc(mysql), Sql, Tuple).
```

Dies ermöglicht zusätzlich `order by` und `group by` mit Aggregation in der relationalen Datenbank:

```
?- Sql = [
      use:[company],
      select:[p^pname, sum(e^salary)],
      from:[employee-e, works_on-w, project-p],
      where:[e^ssn = w^essn and w^pno = p^pnumber],
      group_by:[p^pname] ],
   ddbase_query(odbc(mysql), Sql, Tuple).

Tuple = ['Computerization', 50000.0] ;
Tuple = ['Newbenefits', 93000.0] ;
Tuple = ['ProductX', 55000.0] ;
Tuple = ['ProductY', 95000.0]
```

Konjunktive Anfragen in DATALOG–Form: `ddbase_query/2`

Es gibt auch eine Erweiterung auf konjunktive Anfragen in DATALOG–Form im Stile von `ddbase_call/2`:

```
?- Rule = ( work_on_same_project(X,Y,P) :-
      works_on(X,P,_), works_on(Y,P,_), X \= Y, P < 3 ),
   findall( Rule,
      ddbase_query(odbc(mysql), company:Rule),
      Rules ),
   dwrite(pl, user, Rules).

work_on_same_project(4444, 6666, 1) :-
   works_on(4444, 1, X1), works_on(6666, 1, X2),
   4444 \= 6666, 1 < 3.
work_on_same_project(2222, 4444, 2) :-
   works_on(2222, 2, X1), works_on(4444, 2, X2),
   2222 \= 4444, 2 < 3.
...
```

Typen der Resultatsattribute

- In `ddbase_query/3` werden die Typen der Resultatsattribute aus dem SQL–Statement `Sql` abgeleitet.
- In einer erweiterten Fassung `ddbase_query/4` können diese Typen in einem Optionsteil explizit angegeben werden:

```
ddbase_query(
   odbc(Connection), Sql, Tuple, Options).
```

Konjunktive Anfragen in Field Notation

Eine Erweiterung auf konjunktive Anfragen in Field Notation ist denkbar und geplant.

2.5 GUI–Programmierung in XPCE–PROLOG

Benutzerdialog zur Eingabe von Angestellten

```
ask_employee :-
   new(Dialog, dialog('Define Employee')),
   send_list(Dialog, append, [
      new(N1, text_item('First Name')),
      new(N2, text_item('Last Name')),
      new(S, menu('Sex')),
      new(A, int_item('Age', low := 18, high := 65)),
      new(D, menu('Department', cycle)),
      button(cancel, message(Dialog, destroy)),
      button(enter, message(@prolog, assert_employee,
         N1?selection, N2?selection, S?selection,
         A?selection, D?selection ) ),
      button(show, message(@prolog, employee_table)) ]),
   send_list(S, append, [male, female]),
   send_list(D, append, [research, development, marketing]),
   send(Dialog, default_button, enter),
   send(Dialog, open).
```

Die Methoden für die Buttons `Enter` bzw. `Show` sind folgende:

```
assert_employee(First_Name, Last_Name, Sex, Age, Department) :-
   format('Adding ~w ~w ~w, age ~w, working at ~w~n',
      [Sex, First_Name, Last_Name, Age, Department]),
   assert(
      employee(First_Name, Last_Name, Sex, Age, Department)).

employee_table :-
   findall( [First_Name, Last_Name, Sex, Age, Department],
      employee(First_Name, Last_Name, Sex, Age, Department),
      Rows ),
   Attributes =
      ['First_Name', 'Last_Name', 'Sex', 'Age', 'Department'],
   xpce_display_table(Attributes, Rows).
```

Tabelle in XPCE–PROLOG

First_Name	Last_Name	Sex	Age	Department
John	Smith	male	52	research
Frank	Wong	male	62	research
Alicia	Zelaya	female	49	marketing
Jennifer	Wallace	female	31	development

XML–basierte GUI–Spezifikation (XXUL, verwendet XPCE):

```
<frame title="Define Employee" id="frm">
   <dialog>
      <textbox label="First Name" id="fst"/>
      <textbox label="Last Name" id="lst"/>
      <radiobox label="Sex" rows="1" id="sex">
         <di label="Male"/> <di label="Female"/>
      </radiobox>
      <intbox label="Age" low="18" high="65" id="age"/>
      <menubox label="Department" id="dpt">
         <di label="Research"/> <di label="Development"/>
         <di label="Marketing"/>
      </menubox>
      <dbox orient="horizontal">
         <button label="Cancel" oncommand="quit(frm)"/>
         <button label="Enter"
            oncommand="assert_employee(fst, lst, sex, age, dpt)"/>
         <button label="Show" oncommand="employee_table"/>
      </dbox>
   </dialog>
</frame>
```

Börsen–Charts in XPCE–PROLOG

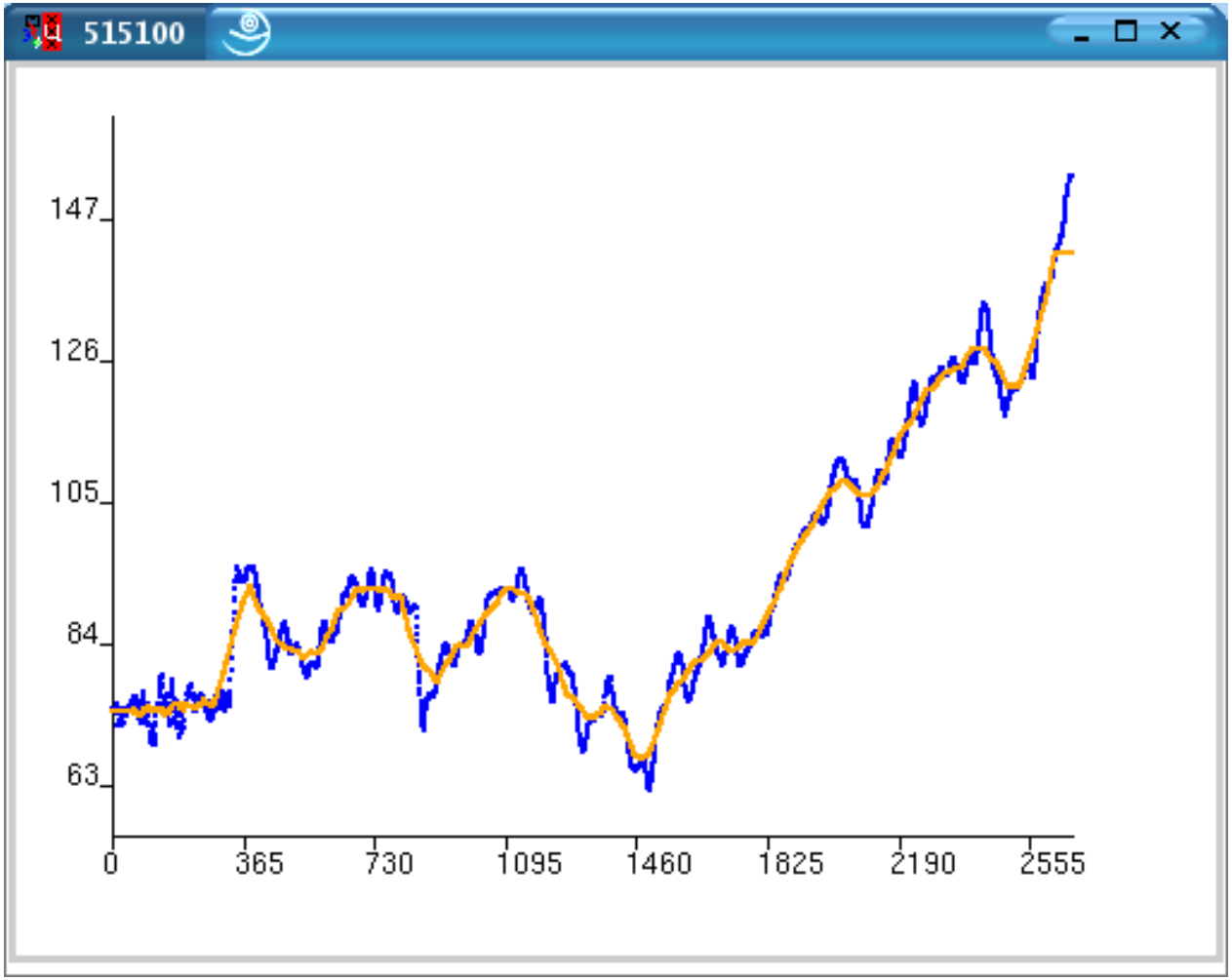
MySQL–Datenbankaufruf:

```
stock_values_for_wkn(Wkn, Values) :-
   stock_values_for_wkn_cached(Wkn, Values),
   !.
stock_values_for_wkn(Wkn, Values) :-
   Statement = [
      use:[stock],
      select:[value], from:[finanztreff],
      where:[wkn=Wkn] ],
   mysql_select_execute_odbc(Statement, Tuples),
   flatten(Tuples, Values),
   assert(stock_values_for_wkn_cached(Wkn, Values)).
```

Nach dem Datenbankaufruf werden die gefundenen Werte als Cache in der internen PROLOG–Datenbank gehalten.

Charts in XPCE–PROLOG:

blau: 20–Tage–Linie, orange: 100–Tage–Linie



Aggregation gleitender Durchschnitte

Der Aufruf `numbers_to_floating_averages(+N, +Xs, -Ys)` berechnet den gleitenden Durchschnitt einer Liste von Zahlen, wobei jeweils `N` Werte zusammengefaßt werden:

```
?- Xs = [1, 2, 3, 4, 5, 6],
   numbers_to_floating_averages(3, Xs, Ys).
Ys = [2, 2, 3, 4, 5, 5]
```

Damit die Resultatsliste `Ys` dieselbe Länge hat wie `Xs` wird am Anfang und am Ende mit gleichen Werten aufgefüllt.

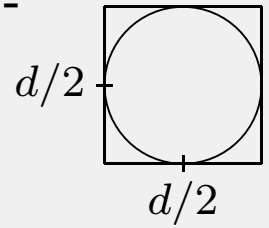
Die linke, obere Ecke des Bildes hat die Bild–Koordinaten $(0, 0)$, und die y–Achse läuft nach unten. Deswegen ist der Ursprung des Chart–Koordinatensystems $(40, 320)$.

```
stock_values_for_wkn_to_picture_with_axis :-
   Parameters = parameters:[
      origin:[40, 320], x:[0, 2650], y:[55, 160],
      step:[365, 21], size:[400, 300] ]
   picture_with_axis(Picture, Parameters),
   Points_1 = [
      1-73.9, 100-74.5, 1000-84.9, 2000-106.5, ... ],
   transform_points_to_plot_points(
      Parameters, Points_1, Points_2),
   send_points(Picture, colour(blue), 2, Points_2),
   send(Picture, open).
```

Die Punkte müssen vor dem Zeichnen transformiert und skaliert werden:

```
transform_points_to_plot_points(
      Parameters, Points_1, Points_2) :-
   maplist( transform_point_to_plot_point(Parameters),
      Points_1, Points_2 ).

transform_point_to_plot_point(
      Parameters, A1-B1, A2-B2) :-
   [X0, Y0] := Parameters/origin/content::'*',
   [X1, X2] := Parameters/x/content::'*',
   [Y1, Y2] := Parameters/y/content::'*',
   [X4, Y4] := Parameters/size/content::'*',
   A2 is X0 + X4 * (A1 - X1) / (X2 - X1),
   B2 is Y0 - Y4 * (B1 - Y1) / (Y2 - Y1).
```

```
send_points(Picture, Colour, Diameter, Points) :-
   checklist( send_point(Picture, Colour, Diameter),
      Points ).

send_point(Picture, Colour, Diameter, X-Y) :-
   Offset = [-Diameter/2,-Diameter/2],
   vector_add([X,Y], Offset, [A,B]),
   send(Picture, display,
      new(Circle, circle(Diameter)), point(A,B) ),
   send(Circle, colour, Colour),
   send(Circle, fill_pattern, Colour).
```



Der Referenzpunkt eines Kreises ist die linke untere Ecke des einfassenden, achsenparallelen Rechtecks. Deswegen wird der Mittelpunkt um den halben Durchmesser in x– und y–Richtung verschoben.

2.6 Datenstrukturen, Kontrollstrukturen und Algorithmen

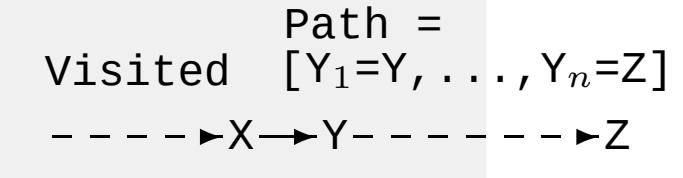
Wir werden Datenstrukturen, Kontrollstrukturen und Algorithmen für folgende Anwendungen in PROLOG behandeln:

- Suche in Graphen
- Sortierverfahren
- Binäre Suchbäume
- Das Anti–Trust–Control–Problem
- Stücklisten–Auflösung
- Medizinische Diagnoseregeln

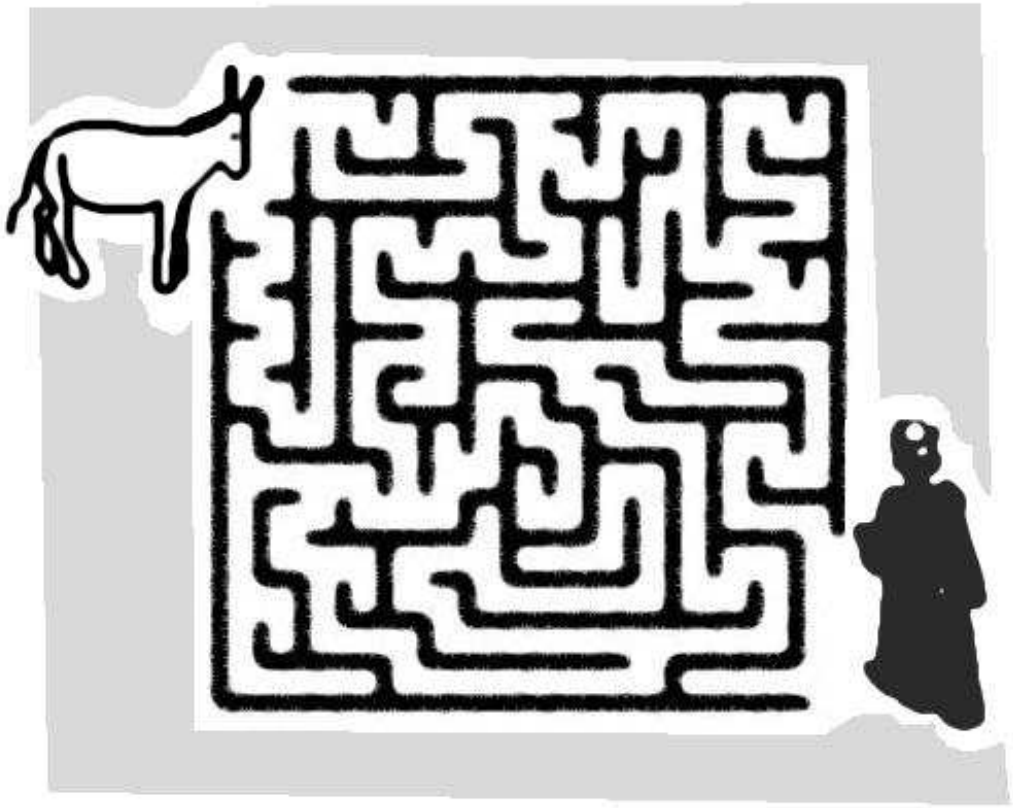
2.6.1 Suche in Graphen

graph_search : Node $\times$ Nodes $\mapsto$ Boolean

```
% graph_search(+Node, -Path) <-

graph_search(X, Path) :-
   graph_search(X, [X], Path).

graph_search(X, _, [X]) :-
   graph_sink(X).
graph_search(X, Visited, [X|Path]) :-
   graph_edge(X, Y),
   not(member(Y, Visited)),
   write(user, '->'), write(user, Y),
   graph_search(Y, [Y|Visited], Path).
```



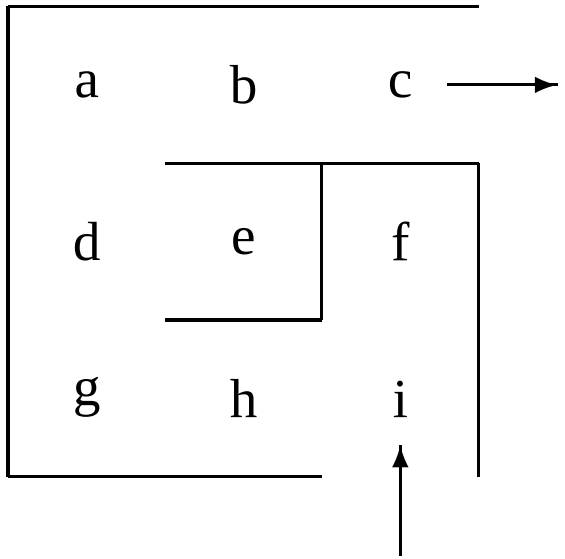
Labyrinth:



Zeige Bileam den Weg zu seiner Eselin!

Repräsentation des Graphen mit PROLOG–Fakten:

Labyrinth:




```
graph_arc(i, f).
graph_arc(i, h).
graph_arc(h, g).
graph_arc(g, d).
graph_arc(d, e).
graph_arc(d, a).
graph_arc(a, b).
graph_arc(b, c).

graph_source(i).
graph_sink(c).
```

Die folgende Regel symmetrisiert das Prädikat `graph_arc/2`:

```
graph_edge(X, Y) :-
   ( graph_arc(X, Y)
   ; graph_arc(Y, X) ).
```

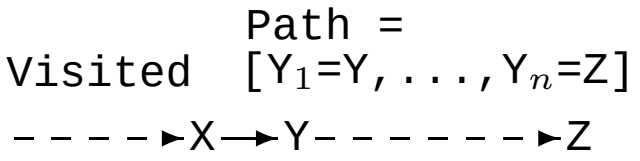
Dadurch ist es nicht nötig, für die Kanten die Rückrichtung explizit anzugeben:

```
graph_edge(i, f).
graph_edge(f, i).
graph_edge(i, h).
graph_edge(h, i).
...
```

- Das Prädikat `graph_search/2` benutzt Tiefensuche, und es berechnet einfache Pfade (ohne doppelte Knoten).
- Beim Aufruf `graph_search(+Node, -Path)` können alle einfachen Pfade von Node zu einer Senke (`graph_sink`) über Backtracking berechnet werden:

```
?- graph_search(i, Path).
->f->h->g->d->e->a->b->c
Path = [i, h, g, d, a, b, c]
?- graph_search(e, Path).
->d->a->b->c
Path = [e, d, a, b, c] ;
->g->h->i->f
No
```

Berechnung einfacher Pfade



- Beim Aufruf `graph_search(X, Visited, [X|Path])` mit einem gebundenen Argument `X` und einer Liste `Visited` von bereits besuchten Knoten, wird
  - eine Kante von `X` zu einem anderen bisher nicht besuchten Knoten `Y` benutzt, und dann
  - ein Weg `Path` von `Y` zu einer Senke `Z` berechnet, der `Y` und die Knoten in `Visited` nicht benutzt.
- Die *Terminierung* wird dadurch gewährleistet, daß Knoten nicht mehrfach besucht werden dürfen.
- Falls `X` bereits eine Senke ist, so ist `Path` die leere Liste.

Würde man eine weitere Kante `graph_arc(e, b)` in den Graphen einfügen (die Wand zwischen e und b einreisen), so gäbe es einen weiteren einfachen Pfad `[e, b, c]` von e zur Senke c.

Man kann alle Antworten mittels Backtracking und `findall/3` bestimmen:

```
...
graph_arc(e, b).

?- findall( Path,
      graph_search(e, Path),
      Paths ).
...
Paths = [ [e, d, a, b, c], [e, b, c] ]
```

2.6.2 Sortierverfahren

Eine ganze Reihe von Sortierverfahren läßt sich sehr elegant rekursiv mittels der Divide–and–Conquer–Idee implementieren.

Wir werden im folgenden

- Quicksort und
- Mergesort

auf Listen behandeln. Dabei werden die beiden Meta–Prädikate

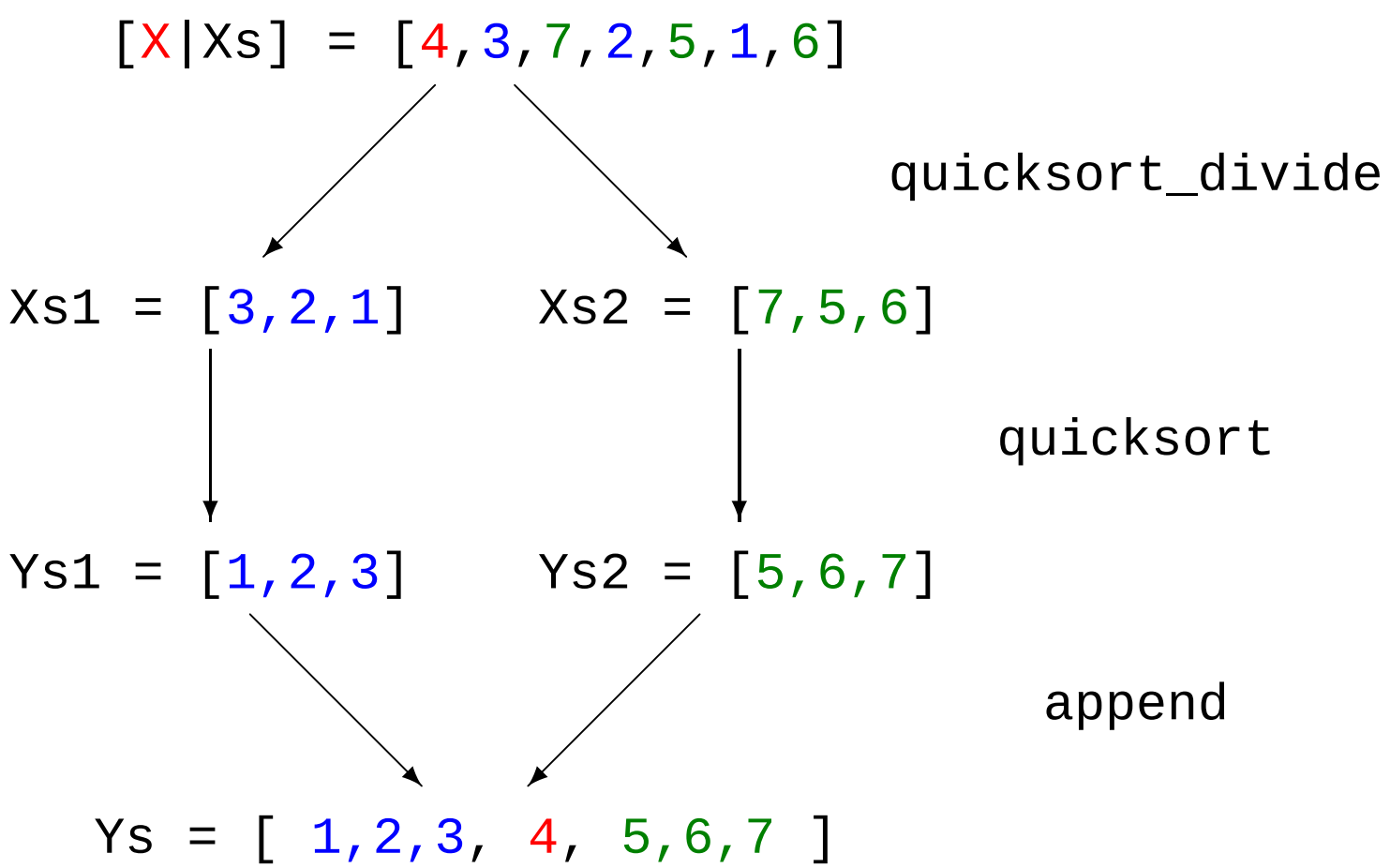
- `findall/3` und
- `sublist/3`

verwendet.

Quicksort

```
% quicksort(List, Sorted_List) <-

quicksort([], []) :-
   !.
quicksort([X|Xs], Ys) :-
   quicksort_divide(X, Xs, Xs1, Xs2),
   quicksort(Xs1, Ys1),
   quicksort(Xs2, Ys2),
   append(Ys1, [X|Ys2], Ys).
```

Der eigentliche Aufwand steckt in der Aufteilung (Divide); der Merge–Schritt ist einfach (append).


[X|Xs] = [4,3,7,2,5,1,6]
quicksort_divide
Xs1 = [3,2,1]
Xs2 = [7,5,6]
quicksort
Ys1 = [1,2,3]
Ys2 = [5,6,7]
append
Ys = [1,2,3, 4, 5,6,7]

```
quicksort_divide(X, Xs, Xs1, Xs2) :-
   findall( X1,
      ( member(X1, Xs),
        X1 < X ),
      Xs1 ),
   findall( X2,
      ( member(X2, Xs),
        X2 > X ),
      Xs2 ).
```

Mittels des Meta–Prädikats `sublist/3` kann man `quicksort_divide/4` noch eleganter implementieren:

```
quicksort_divide(X, Xs, Xs1, Xs2) :-
   sublist( >(X), Xs, Xs1 ),
   sublist( <(X), Xs, Xs2 ).
```

- Der erste Aufruf von `sublist/3` testet für jedes `X1` aus `Xs`, ob `>(X, X1)` gilt, was äquivalent ist zum Infix–Aufruf `X > X1` (bzw. zu obigem Aufruf `X1 < X`); in diesem Fall wird `X1` in die Liste `Xs1` der kleineren Elemente aufgenommen.
- Der zweite Aufruf von `sublist/3` bestimmt analog die Liste `Xs2` der größeren Elemente.

Mergesort

```
% mergesort(List, Sorted_List) <-

mergesort(Xs, Xs) :-
   ( Xs = []
   ; Xs = [_] ),
   !.
mergesort(Xs, Ys) :-
   middle_split(Xs, Xs1, Xs2),
   mergesort(Xs1, Ys1),
   mergesort(Xs2, Ys2),
   mergesort_merge(Ys1, Ys2, Ys).
```

Die Aufteilung (Divide mittels `middle_split`) ist einfach; der eigentliche Aufwand steckt im Merge–Schritt.

```
mergesort_merge([], Xs, Xs) :-
   !.
mergesort_merge(Xs, [], Xs) :-
   !.
mergesort_merge([X1|Xs1], [X2|Xs2], [X|Xs]) :-
   ( X1 < X2 ->
     X = X1,
     mergesort_merge(Xs1, [X2|Xs2], Xs)
   ; X = X2,
     mergesort_merge([X1|Xs1], Xs2, Xs) ).
```

`middle_split(Xs, Xs1, Xs2)` splittet eine Liste `Xs` in zwei etwa gleich große Teile `Xs1` und `Xs2`.

Hier wird die Kontrollstruktur `If -> Then ; Else` verwendet.

2.6.3 Binäre Suchbäume

```
% search_in_binary_tree(Key, Tree) <-

search_in_binary_tree(Key, Tree) :-
   binary_tree_parse(Tree, Root, Lson, Rson),
   ( Key = Root
   ; Key < Root ->
     search_in_binary_tree(Key, Lson)
   ; Key > Root ->
     search_in_binary_tree(Key, Rson) ).
```

Die Suchbäume können dabei in einer XML–artigen Termnotation in PROLOG vorliegen (`binary_tree_mode=xml`) oder in einer einfachen – aber schwerer zu lesenden – Listennotation.

Kapselung des Zugriffs:

Die folgende Methode unifiziert `Tree` mit einem leeren Suchbaum. Sie kann zur Initialisierung und zum Test verwendet werden.

```
% binary_tree_empty(Tree) <-

binary_tree_empty(Tree) :-
   ( dislog_variable_get(binary_tree_mode, xml) ->
     Tree = node:[]:[]
   ; Tree = [] ).
```

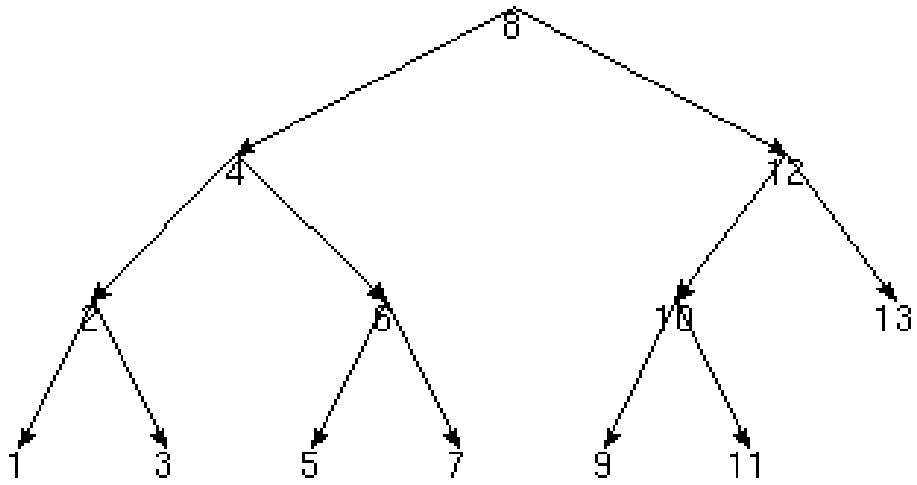
Dabei ist "`:`" ein binärer, assoziativer Operator, und X:Y:Z wird wie X:(Y:Z) geklammert.

Die folgende Methode zerlegt einen Suchbaum in die Wurzel und die beiden Teilbäume; sie schlägt für den leeren Baum bewußt fehl:

```
% binary_tree_parse(Tree, Key, Lson, Rson) <-

binary_tree_parse(Tree, Key, Empty, Empty) :-
   ( dislog_variable_get(binary_tree_mode, xml) ->
     Tree = node:[key:Key]:[]
   ; Tree = [Key] ),
   binary_tree_empty(Empty).
binary_tree_parse(Tree, Key, Lson, Rson) :-
   ( dislog_variable_get(binary_tree_mode, xml) ->
     Tree = node:[key:Key]:[Lson, Rson]
   ; Tree = [Key, Lson, Rson] ).
```

Suchbaum in XML–Darstellung

```
<node key="8">
   <node key="4">
      <node key="2">
         <node key="1"/>
         <node key="3"/>
      </node>
      <node key="6">
         <node key="5"/>
         <node key="7"/>
      </node>
   </node>
   <node key="12">
      <node key="10">
         <node key="9"/>
         <node key="11"/>
      </node>
      <node key="13"/>
   </node>
</node>
```



Visualisierung in PROLOG mittels

```
binary_tree_to_picture(Picture, Tree)
```

Suchbaum – verschiedene Darstellungen

XML–Darstellung:

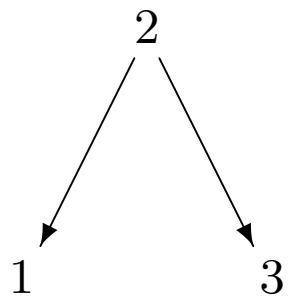
```
<node key="2">
   <node key="1"/>
   <node key="3"/>
</node>
```

PROLOG–Darstellung:

```
node:[key:2]:[
   node:[key:1]:[],
   node:[key:3]:[] ]
```

alternative PROLOG–Darstellung:

```
[ 2,
   [1],
   [3] ]
```

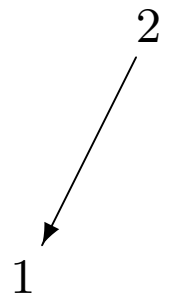


Der leere Baum wird in XML als leerer Knoten

```
<node/>
```

dargestellt. Falls ein Knoten eines Suchbaumes nur genau einen Sohn hat, so wird als anderer Sohn der leere Baum eingetragen:

```
<node key="2">
   <node key="1"/>
   <node/>
</node>
```



Dies gilt auch für die alternative PROLOG–Darstellung

```
[ 2, [1], [] ]
```

Eine entsprechende JAVA–Klasse wäre folgende:

```
Class node { int key ; node lson, rson; ... }
```

Auf der XML–Datenstruktur kann man viele Operationen sehr elegant mit Hilfe von Pfadausdrücken implementieren.

Zur Suche nach einem vorgegebenen Schlüssel `Key` in einem Suchbaum `Tree` genügt z.B. ein einziger Pfadausdruck:

```
search_in_binary_tree(Key, Tree) :-
   dislog_variable_get(binary_tree_mode, xml),
   Key := Tree/descendant::node@key.
```

Dabei wird mittels der Achse `descendant` ein beliebiger Knoten selektiert. Davon wird das Attribut `key` bestimmt und auf den Wert `Key` getestet.

Eigentlich realisiert `X := Exp` die Auswertung des Pfadausdrucks `Exp` und die Zuweisung des Resultats auf `X`. Falls `X` aber bereits gebunden ist, so wird zusätzlich getestet, ob das Resultat mit `X` übereinstimmt (unifiziert).

Einzelne Einfügeoperation

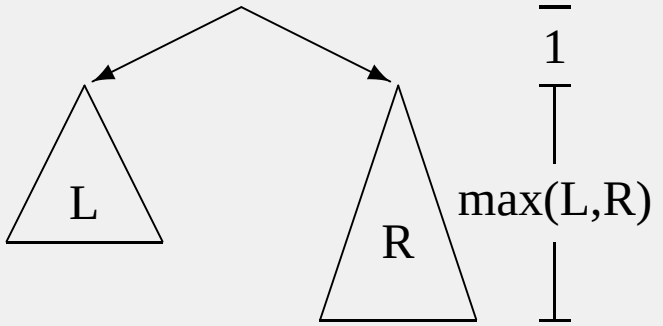
```
% insert_into_binary_tree(Key, Tree, New_Tree) <-

insert_into_binary_tree(Key, Tree, New_Tree) :-
   binary_tree_parse(Tree, Root, Lson, Rson),
   ( Key = Root -> % Key is already in Tree
     New_Tree = Tree
   ; Key < Root ->
     insert_into_binary_tree(Key, Lson, L),
     binary_tree_parse(New_Tree, Root, L, Rson)
   ; Key > Root ->
     insert_into_binary_tree(Key, Rson, R),
     binary_tree_parse(New_Tree, Root, Lson, R) ).
insert_into_binary_tree(Key, Tree, New_Tree) :-
   binary_tree_empty(Tree),
   binary_tree_parse(New_Tree, Key, Tree, Tree).
```

Aufbau eines Suchbaums

```
% keys_to_binary_tree(Keys, Tree) <-

keys_to_binary_tree(Keys, Tree) :-
   binary_tree_empty(Empty),
   insert_into_binary_tree_(Keys, Empty, Tree).


% insert_into_binary_tree_(Keys, Tree, New_Tree) <-

insert_into_binary_tree_([], Tree, Tree).
insert_into_binary_tree_([Key|Keys], Tree, New_Tree) :-
   insert_into_binary_tree(Key, Tree, Tree_2),
   insert_into_binary_tree_(Keys, Tree_2, New_Tree).
```

Berechnung der Höhe eines Suchbaums

```
% binary_tree_to_height(Tree, H) <-

binary_tree_to_height(Tree, -1) :-
   binary_tree_empty(Tree),
   !.
binary_tree_to_height(Tree, H) :-
   binary_tree_parse(Tree, Root, Lson, Rson),
   binary_tree_to_height(Lson, L),
   binary_tree_to_height(Rson, R),
   max(L, R, M),
   H is M + 1.
```



Funktional könnte man in PROLOG auch `H is max(L,R) + 1` schreiben.

2.6.4 Hierarchische Daten

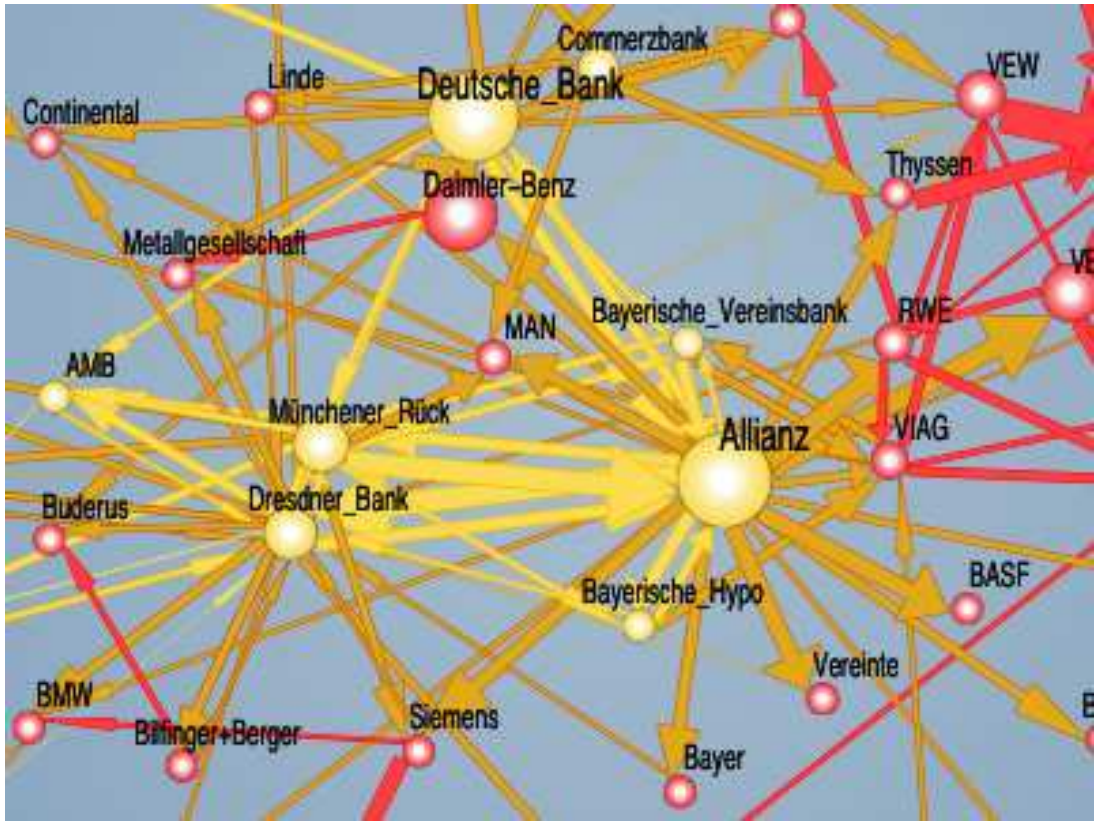
In diesem Abschnitt werden Anfragen über hierarchische Daten (Firmennetzwerk, Stücklistengraph) gestellt.

Diese können in PROLOG elegant mit Hilfe von Meta–Prädikaten (für Kontrollstrukturen) und Backtracking formuliert werden:

- `findall/3`,
- `ddbase_aggregate/3`,
- `maplist/3` (Filter),
- `do/2` (Schleife),
- `transitive_closure/3` (Graphsuche).

Auch eine Bibliothek für Multimengen wird verwendet.

Das deutsche Firmennetzwerk 1996



Das Anti–Trust–Control–Problem

Datenbankschema:

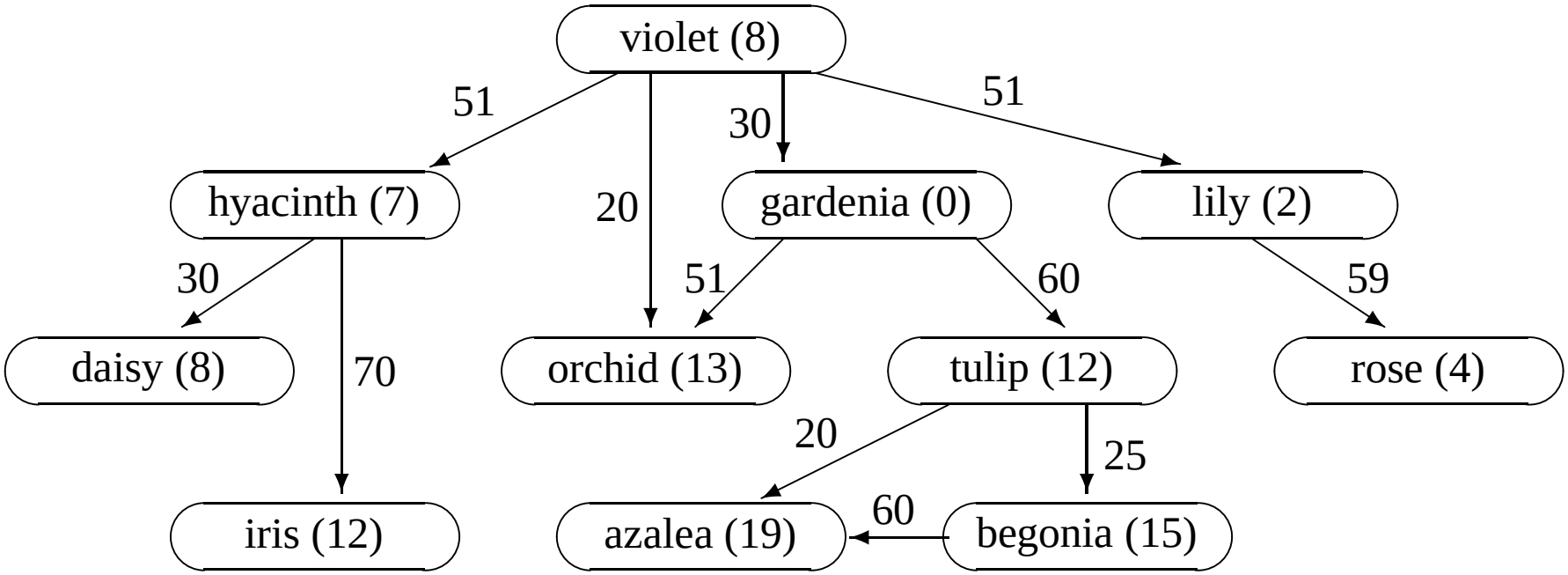
- `has_shares(Company_1, Company_2, P)`:
  Firma `Company_1` hält `P` Prozent der Aktien von Firma `Company_2`.
- `company(Market, Company, P)`:
  Firma `Company` hält `P` Prozent der Aktien des Marktes `Market` selbst.
- `trust_limit(Market, P)`.

Der gesamte Marktanteil "`Total`" einer Firma "`Company`" berechnet sich rekursiv aus den Marktanteilen der (direkten und indirekten) Tochterfirmen:

- `market(Market, Company, Total)`.

Es liegt ein Trust vor, wenn "`Total`" das Trust–Limit des Marktes übersteigt.

Graphische Darstellung der Eigentumsbeziehungen:



- Die Prozentzahlen an den Kanten geben die Beteiligung einer Firma an einer Tochterfirma an (`has_shares`).
- Die Prozentzahlen in Klammern geben den direkten Marktanteil einer Firma an (`company`).

Relationale Repräsentation der Eigentumsbeziehungen:

HAS_SHARES		
violet	hyacinth	51
violet	orchid	20
violet	gardenia	30
violet	lily	51
hyacinth	daisy	30
hyacinth	iris	70
gardenia	orchid	51
fiat	opel	2
gardenia	tulip	60
lily	rose	59
tulip	azalea	20
tulip	begonia	25
begonia	azalea	60

COMPANY		
cars	fiat	35
cars	ford	30
cars	opel	33
toys	azalea	19
toys	begonia	15
toys	daisy	8
toys	gardenia	0
toys	hyacinth	7
toys	iris	12
toys	lily	2
toys	orchid	13
toys	rose	4
toys	tulip	12
toys	violet	8

TRUST–LIMIT	
cars	38
gasoline	40
toys	20

Wir benutzen für das Anti–Trust–Control–Problem das folgende rekursive PROLOG–Programm:

```
controls(Company_1, Company_2) :-
   has_shares(Company_1, Company_2, N),
   N > 50.

market(Market, Company, Total) :-
   company(Market, Company, Quota),
   findall( C,
      controls(Company, C),
      Companies ),
   maplist( market(Market),
      Companies, Quotas ),
   sum([Quota|Quotas], Total).
```

`maplist` wendet hier `market/3` rekursiv auf die kontrollierten Firmen an.

Falls der Graph der Eigentumsbeziehungen kein Baum ist, so kann man eine Dauerschleife oder Doppeltzählungen verhindern, indem man für jeden besuchten Knoten `V` in der PROLOG–Datenbank `assert(visited(V))` ausführt:

```
market(Market, Company, Total) :-
   company(Market, Company, Quota),
   assert(visited(Company)),
   findall( C,
      ( controls(Company, C),
        not(visited(C)) ),
      Companies ),
   maplist( market(Market),
      Companies, Quotas ),
   sum([Quota|Quotas], Total).
```

Außerdem könnte man auch direkt die transitiv kontrollierten Firmen mittels einer Graphsuche berechnen, und daraus dann `Total`:

```
market(Market, Company, Total) :-
   company(Market, Company, _),
   assert(visited(Company)),
   ddbase_aggregate( [sum(Quota)],
      ( transitively_controls(Company, C),
        company(C, Market, Quota) ),
      [[Total]] ).
```

```
transitively_controls(C1, C2) :-
   ( C1 = C2
   ; transitive_closure(arc, C1, C2) ).
arc(C1, C2) :-
   controls(C1, C2), not(visited(C2)), assert(visited(C2)).
```

Dann wird das rekursive Finden aller transitiv kontrollierten Firmen von `transitively_controls/2` anstelle von `market/3` erledigt.

In den beiden letzten Fällen muß man nach jedem Lauf von `market/3` die Hilfsfakten `visited(V)` wieder aus der PROLOG–Datenbank entfernen.

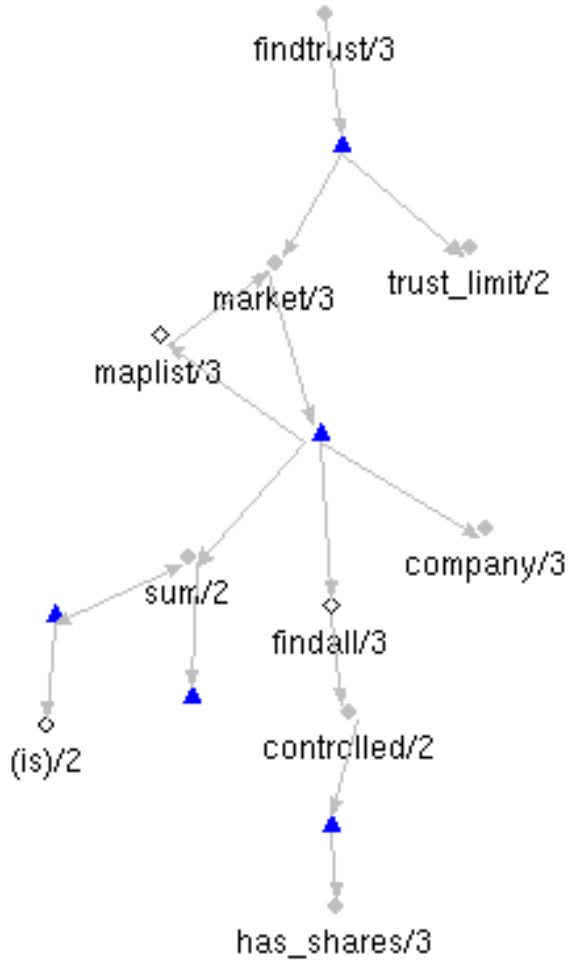
```
?- retractall(visited(_)).
Yes
```

Nun kann man Trusts berechnen:

```
findtrust(Market, Company, Total) :-
   market(Market, Company, Total),
   trust_limit(Market, Threshold),
   Total > Threshold.
```

Aufgrund der Implementierung muß beim Aufruf keines der Argumente von `findtrust/3` gebunden sein.

Die Abhängigkeiten zwischen den Prädikaten (grau) und Regeln (blau) in Programmalternative 1 erkennt man am verallgemeinerten Rule/Goal–Graphen – die arithmetischen Vergleichsprädikate wurden weggelassen:



Alternativ könnte man das Problem auch in DATALOG$^{\mathsf{agg}}$ mit Aggregationsfunktionen folgendermaßen lösen:

$$
\begin{aligned}
&market(Market, Company, Quota) \leftarrow \\
&\qquad company(Market, Company, Quota\_1) \wedge \\
&\qquad market\_indirect(Market, Company, Quota\_2) \wedge \\
&\qquad Quota \ \ is \ \ Quota\_1 + Quota\_2.
\end{aligned}
$$

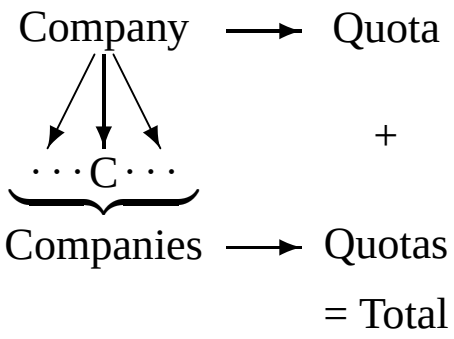
$$
\begin{aligned}
&market\_indirect(Market, Company, \langle\, Quota \,\rangle_{sum}) \leftarrow \\
&\qquad controls(Company, C) \wedge \\
&\qquad market(Market, C, Quota).
\end{aligned}
$$

Die PROLOG–Regel

```
market(Market, Company, Total) :-
   company(Market, Company, Quota),
   findall( C,
      controls(Company, C),
      Companies ),
   maplist( market(Market),
      Companies, Quotas ),
   sum([Quota|Quotas], Total).
```

berechnet für "`Market = toys`" und "`Company = violet`" folgendes:

- Aufruf von `company`: `Quota = 8`,
- Aufruf von `findall`:
  `Companies = [hyacinth, lily]`,
- Aufruf von `maplist`: `Quotas = [19, 6]`,
- Aufruf von `sum`: `Total = 33` (`= 8 + 19 + 6`).



Dabei werden im Rahmen von des Aufrufs von

```
maplist( market(Market),
   Companies, Quotas ),
```

für “`Market = toys`” und “`Companies = [hyacinth, lily]`” die beiden Unteranfragen

- `market(toys, hyacinth, Quota)` und
- `market(toys, lily, Quota)`

gestellt und mit “`Quota = 19`” bzw. “`Quota = 6`” gelöst.

Danach ist “`Quotas = [19, 6]`”.

Die PROLOG–Regel

```
findtrust(Market, Company, Total) :-
   market(Market, Company, Total),
   trust_limit(Market, Threshold),
   Total > Threshold.
```

berechnet für “`Market = toys`” und “`Company = violet`” folglich

- `Threshold = 20` und
- `Total = 33`.

Der Test “`33 > 20`” ist dann erfolgreich.

PROLOG–Sitzung

Die erhaltenen Ergebnisse sehen wir in der folgenden PROLOG–Sitzung:

```
?- findtrust(toys, gardenia, Total).
Total = 25
?- findtrust(toys, violet, Total).
Total = 33
?- findtrust(toys, tulip, Total).
No
?- findtrust(Market, begonia, Total).
Market = toys, Total = 34
```

Das Meta–Prädikat maplist/3

Der Aufruf `maplist(Predicate, Xs, Ys)` wendet `Predicate` auf alle Elemente der Liste `Xs` an und produziert eine neue, gleichlange Liste `Ys`.

Man könnte das generische Prädikat `maplist/3` in PROLOG auch rekursiv wie folgt programmieren:

```
maplist(_, [], []) :-
   !.
maplist(Predicate, [X|Xs], [Y|Ys]) :-
   call(Predicate, X, Y),
   maplist(Predicate, Xs, Ys).
```

Dabei kann `Predicate` ein Atom – möglicherweise mit Argumenten – sein, d.h. `p(`T_1`,...,`T_n`)` oder `p`, für ein Prädikatensymbol `p`.

Für solch ein fest vorgegebenes Atom `p(`T_1`,...,`T_n`)` könnte man die Iteration auch wie folgt als Listenrekursion programmieren:

```
p_it(_,...,_, [], []).
p_it(T_1,...,T_n, [X|Xs], [Y|Ys]) :-
   p(T_1,...,T_n, X, Y),
   !,
   p_it(T_1,...,T_n, Xs, Ys).
```

Diesen PROLOG–Code müßte man aber für jedes Atom `p(`T_1`,...,`T_n`)` extra schreiben.

Dann hätten die beiden Aufrufe `maplist(p(`T_1`,...,`T_n`),Xs,Ys)` und `p_it(`T_1`,...,`T_n`,Xs,Ys)` denselben Effekt.

Das Meta–Prädikat do/2

Der Aufruf

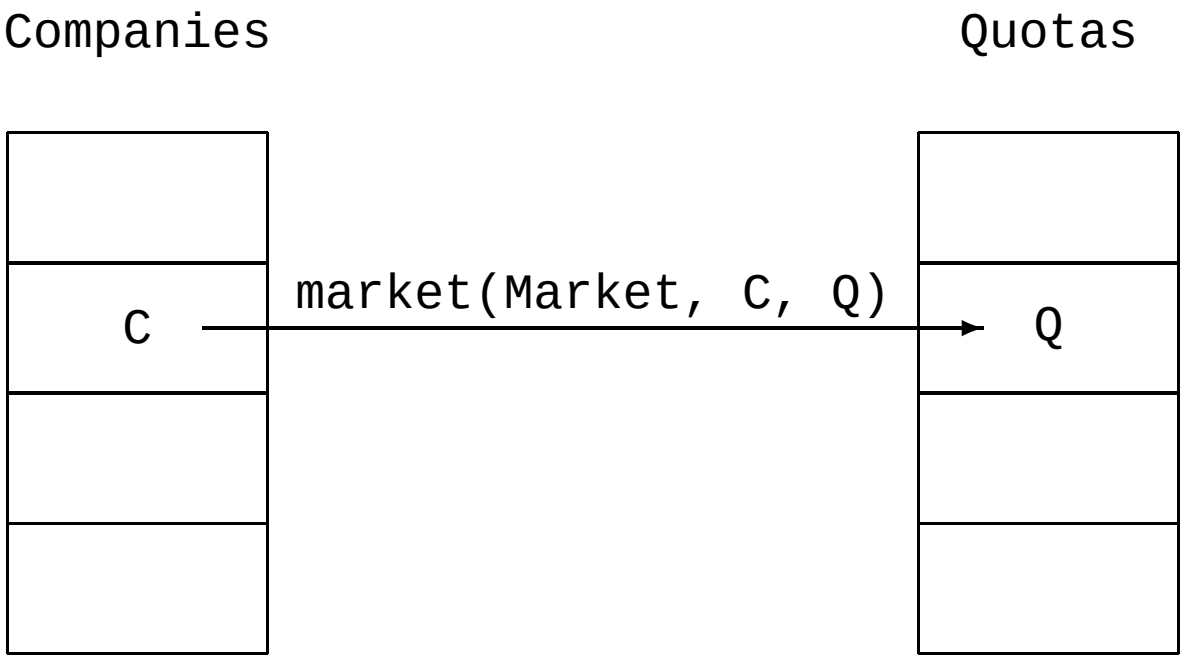
```
maplist( market(Market),
   Companies, Quotas )
```

ist äquivalent zu

```
( foreach(C, Companies), foreach(Q, Quotas) do
     market(Market, C, Q) )
```

Die Liste `Companies` wird sukzessive in `Quotas` transformiert:

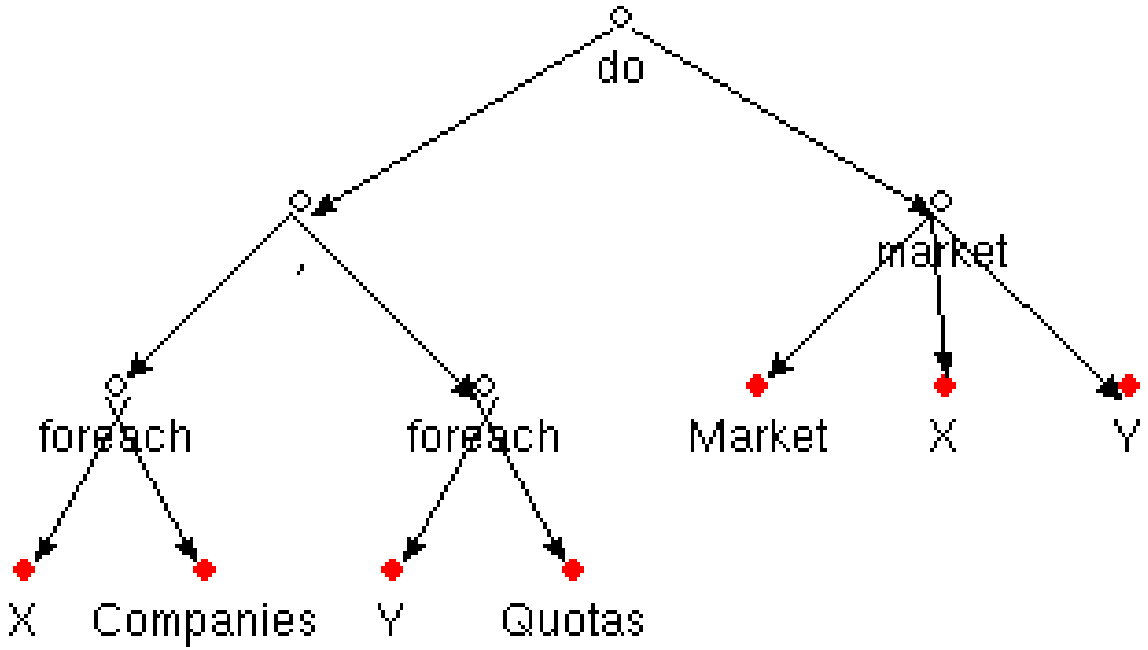
```
( foreach(C, Companies), foreach(Q, Quotas) do
    market(Market, C, Q) )
```



Der PROLOG–Term

```
( foreach(X, Companies), foreach(Y, Quotas) do
    market(Market, X, Y) )
```

hat folgenden Term–Graphen:



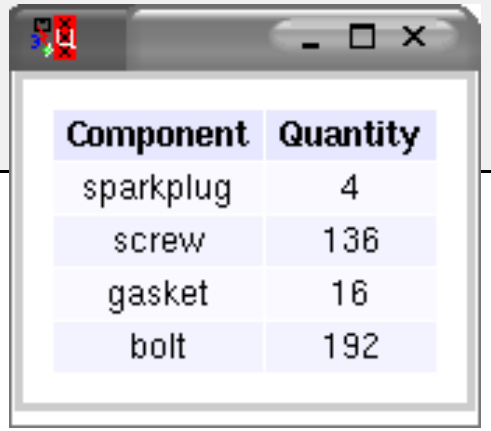
Stücklisten–Auflösung

Um neben den Kosten eines Teils auch noch seine Stücklisten–Auflösung zu berechnen, benutzen wir das folgende PROLOG–Programm.

```
costs(Product, Total_Cost) :-
   parts_of_list(Product:1, Parts),
   maplist( cost,
      Parts, Costs ),
   sum(Costs, Total_Cost).

cost(Component:Quantity, Cost) :-
   price(Component, Price),
   Cost is Price * Quantity.
```

Das Ergebnis sieht dann wie folgt aus:

```
?- parts_of_list(engine:1, Parts).
Parts = [
   sparkplug:4, screw:136, gasket:16, bolt:192]

?- costs(engine, Total_Cost).
Total_Cost = 744
```



Component	Quantity
sparkplug	4
screw	136
gasket	16
bolt	192

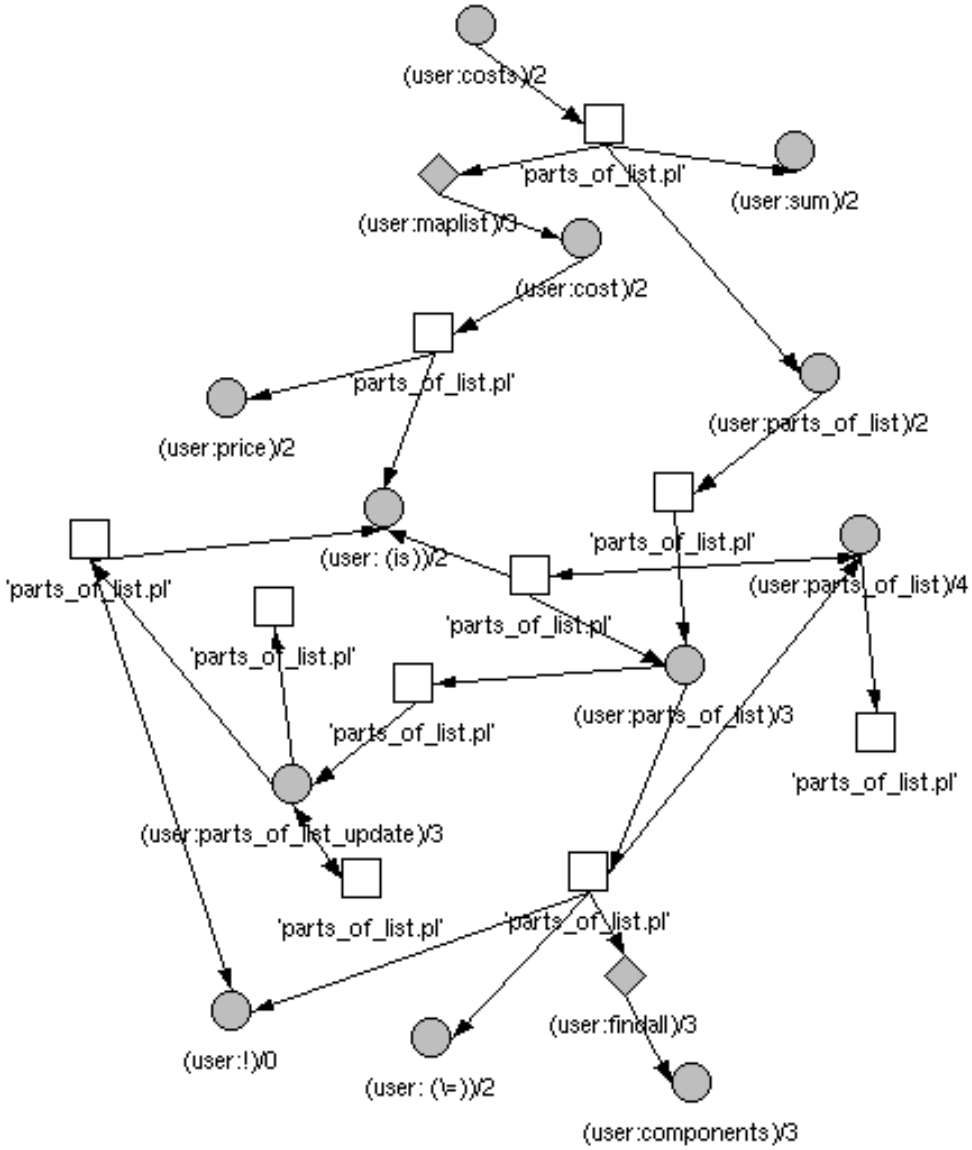
Berechnungen von parts_of_list/2

1. Alternative: aus [CeGoTa 90]

```
parts_of_list(Component:Quantity, Parts) :-
   parts_of_list(Component:Quantity, [ ], Parts).

parts_of_list(C:Q, Parts_1, Parts_2) :-
   findall( C2:Q2,
      components(C, C2, Q2),
      Parts ),
   Parts \= [ ],
   !,
   parts_of_list(Q, Parts, Parts_1, Parts_2).
parts_of_list(C:Q, Parts_1, Parts_2) :-
   multiset_insert(C:Q, Parts_1, Parts_2).
```

```
parts_of_list(F, [C:Q|Ps0], Ps1, Ps3) :-
   Q2 is F * Q,
   parts_of_list(C:Q2, Ps1, Ps2),
   parts_of_list(F, Ps0, Ps2, Ps3).
parts_of_list(_, [ ], Ps, Ps).

% multiset_insert(X:M, Multiset_1, Multiset_2) <-

multiset_insert(X:M, [X:M1|Ps], [X:M2|Ps]) :-
   !,
   M2 is M1 + M.
multiset_insert(X:M, [P|Ps1], [P|Ps2]) :-
   multiset_insert(X:M, Ps1, Ps2).
multiset_insert(X:M, [], [X:M]).
```

Rule/Goal–Graph in VISUR



4.32 Finding all Solutions to a Goal

findall(*+Var, +Goal, -Bag*)
Creates a list of the instantiations *Var* gets successively on backtracking over *Goal* and unifies the result with *Bag*. Succeeds with an empty list if *Goal* has no solutions. `findall/3` is equivalent to `bagof/3` with all free variables bound with the existence operator (`^`), except that `bagof/3` fails when goal has no solutions.

bagof(*+Var, +Goal, -Bag*)
Unify *Bag* with the alternatives of *Var*, if *Goal* has free variables besides the one sharing with *Var* bagof will backtrack over the alternatives of these free variables, unifying *Bag* with the corresponding alternatives of *Var*. The construct *+Var^Goal* tells bagof not to bind *Var* in *Goal*. `bagof/3` fails if *Goal* has no solutions.

The example below illustrates `bagof/3` and the `^` operator. The variable bindings are printed together on one line to save paper.

```
2 ?- listing(foo).

foo(a, b, c).
foo(a, b, d).
foo(b, c, e).
foo(b, c, f).
foo(c, c, g).

Yes
3 ?- bagof(C, foo(A, B, C), Cs).

A = a, B = b, C = G308, Cs = [c, d] ;
A = b, B = c, C = G308, Cs = [e, f] ;
A = c, B = c, C = G308, Cs = [g] ;

Yes
4 ?- bagof(C, A^foo(A, B, C), Cs).

A = G324, B = b, C = G326, Cs = [c, d] ;
A = G324, B = c, C = G326, Cs = [e, f, g] ;

Yes
5 ?-
```

[34] Please note that the semantics have changed between 3.1.1 and 3.1.2

setof(*+Var, +Goal, -Set*)
Equivalent to `bagof/3`, but sorts the result using `sort/2` to get a sorted list of alternatives without duplicates.

4.33 Invoking Predicates on all Members of a List

All the predicates in this section call a predicate on all members of a list or until the predicate called fails. The predicate is called via call/[2..], which implies common arguments can be put in front of the arguments obtained from the list(s). For example:

```
?- maplist(plus(1), [0, 1, 2], X).

X = [1, 2, 3]
```

we will phrase this as "*Predicate* is applied on ..."

checklist(*+Pred, +List*)
Pred is applied successively on each element of *List* until the end of the list or *Pred* fails. In the latter case the `checklist/2` fails.

maplist(*+Pred, ?List1, ?List2*)
Apply *Pred* on all successive pairs of elements from *List1* and *List2*. Fails if *Pred* can not be applied to a pair. See the example above.

sublist(*+Pred, +List1, ?List2*)
Unify *List2* with a list of all elements of *List1* to which *Pred* applies.

4.34 Forall

forall(*+Cond, +Action*)
For all alternative bindings of *Cond Action* can be proven. The example verifies that all arithmetic statements in the list *L* are correct. It does not say which is wrong if one proves wrong.

```
?- forall(member(Result = Formula, [2 = 1 + 1, 4 = 2 * 2]),
                  Result =:= Formula).
```

2. Alternative:

```
parts_of_list(C:Q, Parts) :-
   parts_of_list(C:Q),
   collect_parts_of_list_found(Parts_2),
   multiset_normalize(Parts_2, Parts).

parts_of_list(C:Q) :-
   not(components(C, _, _)),
   !,
   assert(parts_of_list_found(C:Q)).
parts_of_list(C:Q) :-
   forall( components(C, C1, Q1),
      ( Q2 is Q * Q1, parts_of_list(C1:Q2) ) ).

collect_parts_of_list_found(Parts) :-
   findall( Part,
      retract(parts_of_list_found(Part)),
      Parts ).
```

```
% multiset_normalize(+M1, ?M2) <-

multiset_normalize(M1, M2) :-
   multiset_to_elements(M1, Xs),
   maplist( multiset_to_element_with_multiplicity(M1),
      Xs, M2 ).

multiset_to_elements(Multiset, Xs) :-
   findall( X,
      member(X:_, Multiset),
      Xs_2 ),
   sort(Xs_2, Xs).

multiset_to_element_with_multiplicity(Multiset, X, X:M) :-
   findall( N,
      member(X:N, Multiset),
      Ns ),
   sum(Ns, M).
```

Das Meta–Prädikat `bagof/3` gruppiert bei Backtracking nach existentiellen Variablen (unten: `X`) – `findall/3` tut dies nicht.

Damit kann man `multiset_normalize/2` sogar noch eleganter berechnen – ohne zuerst mit `multiset_to_elements/2` die Liste der unterschiedlichen Elemente berechnen zu müssen:

```
multiset_normalize(M1, M2) :-
   findall( X:M,
      multiset_to_element_with_multiplicity(M1, X:M),
      M2 ).

multiset_to_element_with_multiplicity(Multiset, X:M) :-
   bagof( N, member(X:N, Multiset), Ns ),
   sum(Ns, M).
```

3. Alternative:

```
parts_of_list(C:Q, [C:Q]) :-
   not(components(C, _, _)),
   !.
parts_of_list(C:Q, Parts) :-
   findall( Ps,
      ( components(C, C1, Q1),
        Q2 is Q * Q1,
        parts_of_list(C1:Q2, Ps) ),
      Pss ),
   append(Pss, Parts_2),
   multiset_normalize(Parts_2, Parts).
```

Wenn man die Bibliothek für Multisets voraussetzt, dann ist die dritte Alternative die eleganteste.

Man kann das Prädikat `append/2` zu Konkatenation einer Liste von Listen wie folgt auf der Basis von `append/3` implementieren; in der Praxis wird aber meist eine effizientere, end–rekursive Implementierung mittels Akkumulatoren verwendet:

```
% append(Xss, Zs) <-

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

Beim Aufruf `append([[1,2], [3,4], [5,6,7]], Ys)` wird in der ersten Regel `Xs = [1,2]` und `Xss = [[3,4], [5,6,7]]` gesetzt.

- `append(Xss, Ys)` berechnet rekursiv `Ys = [3,4,5,6,7]`.
- `append(Xs, Ys, Zs)` berechnet dann `Zs = [1,2,3,4,5,6,7]`.

Die erste Regel wird nacheinander mit folgenden ersten Argumenten aufgerufen:

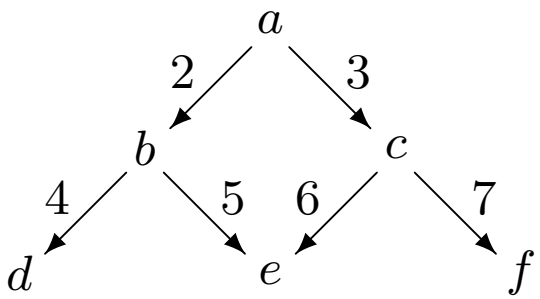
- `[[1,2], [3,4], [5,6,7]]`,
- `[[3,4], [5,6,7]]`,
- `[[5,6,7]]` und
- `[]`.

Die zweite Regel (Abbruchregel) dient am Rekursionsende dazu aus der leeren Liste von Listen die leere Liste zu erzeugen.

Umgekehrt könnte man `append/3` natürlich auch mittels `append/2` ausdrücken – falls man eine von `append/3` unabhängige Implementierung für `append/2` hätte:

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

Beispiel (Stücklisten–Auflösung)



Wir betrachten die Argumente der Aufrufe

```
parts_of_list(C:Q, Parts).
```

Für (a:1, Parts_a) gibt es innerhalb von `findall/3` zwei Unteraufrufe:

- (b:2, Parts_b) führt wiederum zu (d:8, Parts_d) und (e:10, Parts_e) mit den Antworten Parts_d = [d:8] bzw. Parts_e = [d:10].
  Aus Pss_b = [[d:8], [e:10]] wird schließlich Parts_b = [d:8, e:10].
- (c:3, Parts_c) wird analog mit Parts_c = [e:18, f:21] beantwortet.

Aus Pss_a = [[d:8, e:10], [e:18, f:21]] machen `append/2` und `multiset_normalize/2` schließlich Parts_a = [d:8, e:28, f:21].

2.6.5 Diagnoseregeln in DATALOG*

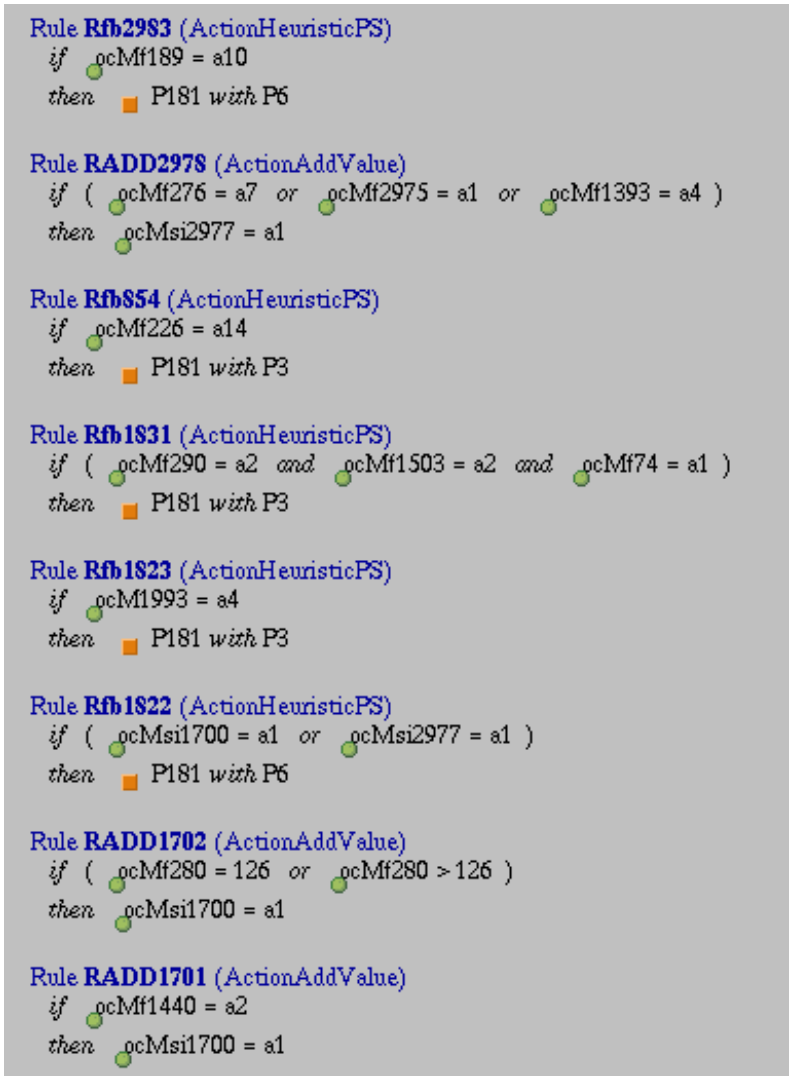
Die folgenden D3–Diagnoseregeln weisen Zwischendiagnosen `Msi...` einen Wert `a...` zu. Dies geschieht durch Verknüpfung der Antworten auf die Fragen `Mf...` während eines interaktiven Untersuchungs–Dialogs:

```
% Rule RADD1702 (ActionAddValue)
d3_finding('Msi1700' = a1) :-
   d3_condition('Mf280' >= 126).

% Rule RADD2978 (ActionAddValue)
d3_finding('Msi2977' = a1) :-
   ( d3_condition('Mf276' = a7)
   ; d3_condition('Mf2975' = a1)
   ; d3_condition('Mf1393' = a4) ).
```

Die folgenden D3–Diagnoseregeln weisen Diagnosen `P...` aufgrund der Zwischendiagnosen und der Antworten auf die gestellten Fragen Scores `Pk` zu:

```
% Rule Rfb1822 (ActionHeuristicPS)
d3_diagnosis('P181' = 'P6') :-
   ( d3_condition('Msi1700' = a1)
   ; d3_condition('Msi2977' = a1) ).

% Rule Rfb1831 (ActionHeuristicPS)
d3_diagnosis('P181' = 'P3') :-
   d3_condition('Mf290' = a2),
   d3_condition('Mf1503' = a2),
   d3_condition('Mf74' = a1).
```

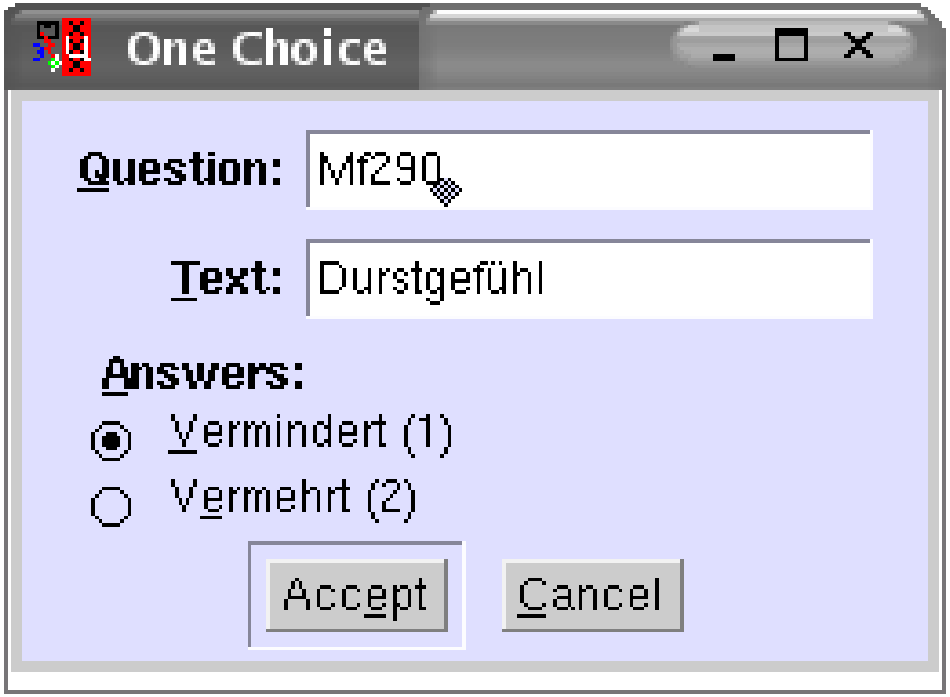
Die zu einer Diagnose abgeleiteten Scores werden später akkumuliert.


Rule Rfb2983 (ActionHeuristicPS)
if ocMf189 = a10
then P181 with P6
Rule RADD2978 (ActionAddValue)
if (ocMf276 = a7 or ocMf2975 = a1 or ocMf1393 = a4)
then ocMsi2977 = a1
Rule Rfb854 (ActionHeuristicPS)
if ocMf226 = a14
then P181 with P3
Rule Rfb1831 (ActionHeuristicPS)
if (ocMf290 = a2 and ocMf1503 = a2 and ocMf74 = a1)
then P181 with P3
Rule Rfb1823 (ActionHeuristicPS)
if ocM1993 = a4
then P181 with P3
Rule Rfb1822 (ActionHeuristicPS)
if (ocMsi1700 = a1 or ocMsi2977 = a1)
then P181 with P6
Rule RADD1702 (ActionAddValue)
if (ocMf280 = 126 or ocMf280 > 126)
then ocMsi1700 = a1
Rule RADD1701 (ActionAddValue)
if ocMf1440 = a2
then ocMsi1700 = a1

Die Auswertung der DATALOG*–Regeln wurde in PROLOG implementiert.
Sie erfolgt zwar Bottom–Up, aber nicht wie sonst für DATALOG üblich:

- Es werden nach jeder Iteration alle Scores `Pk` bzw. `Nk` zur selben Diagnose *akkumuliert*, bevor die nächste Iteration startet.
- Außerdem bedingen Atome `d3_condition(Qid` $\odot$ `V)` zu noch nicht gestellten Fragen `Qid` den Aufruf `d3_dialog(Qid = Val)` eines geeigneten PROLOG–Dialogs, durch den die Antwort `Val` auf die Frage ermittelt wird.
- Dieser Wert `Val` wird dann als `d3_finding(Qid = Val)` in die DATALOG*–Datenbasis aufgenommen (`assert`) und mittels des Vergleichsoperators $\odot$ mit `V` verglichen.

Benutzerdialog in D3 für die Frage `Mf290`



```
d3_condition(C) :-
   C =.. [Comparator, Qid, V],
   ( d3_finding(Qid = Val) ->
     true
   ; d3_dialog(Qid = Val),
     assert(d3_finding(Qid = Val)) ),
   call(Comparator, Val, V).
```

Der Aufruf "`f(`t_1`,...,`t_n`) =.. Xs`" zerlegt dabei einen Term in die Liste "`Xs = [f,`t_1`,...,`t_n`]`" bestehend aus seinem Funktionssymbol und seinen Argumenten.

2.7 Vergleich mit DATALOG

PROLOG und DATALOG unterscheiden sich syntaktisch (Sprachumfang) und semantisch (Auswertungsweise):

- Sprachumfang: Verwendung von Funktionssymbolen (für komplexe Terme), Built–Ins, Seiteneffekten, Default Negation und Disjunktion.
- Auswertungsweise:
  - PROLOG: tupel–orientiert, Top–Down (zielgerichtet), Backward Chaining
  - DATALOG: mengen–orientiert, Bottom–Up, Forward Chaining.

Die existierenden DATALOG–Systeme unterscheiden sich auch; sie stellen unterschiedliche Erweiterungen des Kern–DATALOG zur Verfügung.

Die meisten DATALOG–Systeme erlauben nur sehr eingeschränkte Built–In–Prädikate und keine Seiteneffekte.

PROLOG	DATALOG
Top–Down, Backward Chaining	Bottom–Up, Forward Chaining
Tiefensuche	(normalerweise) Breitensuche
tupel–orientiert	mengen–orientiert
an Auswertungsreihenfolge gebunden	nicht an Auswertungsreihenfolge gebunden
Spezialprädikate	keine Spezialprädikate
Funktionssymbole	keine Funktionssymbole

PROLOG ist eine mächtige Datenbank–Programmiersprache:

- Die Verarbeitung ist zwar zielgerichtet, aber sie ist wegen der tupel–orientierten Vorgehensweise für große Datenmengen manchmal ineffizient. Außerdem terminiert PROLOG selbst für Programme ohne Funktionssymbole nicht immer.
- PROLOG ist mehr prozedural als deklarativ. Die Reihenfolge der Regeln und ihrer Rumpfliterale ist für die Semantik und die Effizienz entscheidend. Die Ausführung von Programmen wird durch Spezialprädikate gesteuert.

DATALOG ist eine deklarative Einschränkung von PROLOG.

- Es ist nicht so flexibel und ausdruckstark in der Programmierung.
- Es kann aber mengen–orientiert und effizient ausgewertet werden, und terminiert für Programme ohne Funktionssymbole immer.

Deswegen wird in deduktiven Datenbankanwendungen in der Praxis eine Kombination von PROLOG und DATALOG angestrebt.

- Dabei spielt PROLOG die Rolle der universellen Programmiersprache, in der z.B. die Benutzerschnittstellen implementiert werden.
- In PROLOG können dann Aufrufe an DATALOG–Tools eingebettet werden. DATALOG fungiert als deklarative Datenbankanfragesprache.

Diese Situation kann man z.B. mit der Einbettung von SQL in JAVA im Rahmen von JDBC vergleichen.

- Man kann allerdings auch direkt aus PROLOG heraus über ODBC auf SQL–Datenbanken zugreifen.

Deklarative vs. prozedurale Programmierung

- DATALOG kann natürlich – als Untersprache von PROLOG – auch mittels Backward Chaining ausgewertet werden.
- DATALOG ist aber – selbst mit seinen Erweiterungen um Aggregatsfunktionen, Built–In–Prädikate, Negation und Disjunktion – auf die *rein deklarative Programmierung* beschränkt.
- In der universellen Programmiersprache PROLOG kann man dagegen auch *prozedural* – unter Anbindung anderer Tools – und mit GUI programmieren.
- Dadurch wird es möglich neben Forward und Backward Chaining auch noch andere deklarative *Inferenzmethoden aus der KI* in PROLOG unter Einbindung von DATALOG zu realisieren.

Da PROLOG und DATALOG beide Turing–vollständig sind, kann man mit beiden natürlich dasselbe programmieren – nur auf unterschiedliche Art und Weise und verschieden elegant.

Es gibt viele DATALOG–Programme, für die beide Systeme – nach außen hin – die gleichen Ergebnisse produzieren.

- Außerdem gibt es Varianten der DATALOG–Auswertung (Magic Sets $\rightarrow$ zielgerichtete Auswertung, allerdings nicht tupel–orientiert), die für weitere Anwendungen ein PROLOG–ähnliches Verhalten produzieren.
- Umgekehrt kann man PROLOG auch mengen–orientiert auswerten (QSQ, allerdings nicht bottom–up).

PROLOG und ein geeignet erweitertes DATALOG, welches wir DATALOG* nennen, können zusammen verwendet werden und sich gegenseitig aufrufen.

Auswertungsweise

Das *Backward Chaining* bearbeitet Regeln entgegen der Richtung des Regelpfeils ausgehend von einer Anfrage $\leftarrow Q$:

- Mittels Unifikation wird eine Instanz

  $$A \leftarrow B_1 \wedge \ldots \wedge B_m$$

  einer Regel ermittelt, die zur Beantwortung der Anfrage herangezogen werden kann; d.h., es wird eine passende Instanz mit $Q = A$ gesucht. Dann werden Unteranfragen $B_1, \ldots, B_m$ gestellt.

- Dies erfolgt tupel–orientiert – vergleichbar mit der Auswertung in Programmiersprachen.

Somit erfolgt die Beantwortung einer Anfrage *top–down* und zielgerichtet.

Die Berechnung endet, wenn alle – rekursiv gestellten – Unteranfragen abgearbeitet sind. Dann hat man eine Antwort auf die Ausgangsanfrage ermittelt.

Das *Forward Chaining* schließt in Richtung des Regelpfeils:

- Falls die Rumpfatome $B_1, \ldots, B_m$ einer Grundinstanz

  $$A \leftarrow B_1 \wedge \ldots \wedge B_m$$

  einer Regel bereits abgeleitet wurden, dann kann man auch das Kopfatom A ableiten.

- Dies erfolgt mengen–orientiert für alle Grundinstanzen – vergleichbar mit der Auswertung eines SELECT–Statements in SQL.

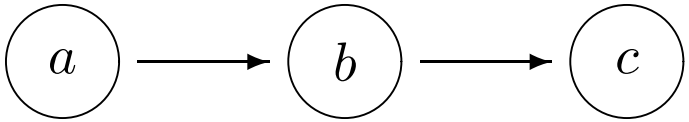
So werden *bottom–up*, ausgehend von den anfangs bekannten Fakten, immer neue Fakten abgeleitet. In der Reinform der Bottom–Up–Auswertung werden erst am Ende der Berechnung die zur Anfrage passenden abgeleiteten Fakten selektiert.

Es gibt allerdings Verfahren, die die Anfrage schon geeignet in die Bottom–Up–Auswertung einbringen (Magic Sets), so daß möglichst nur für die Beantwortung der Anfrage relevante Fakten abgeleitet werden.

Beispiel (Forward vs. Backward Chaining)

Für das DATALOG–Programm $P_{tc} = \{ r_1, r_2, f_1, f_2 \}$ mit

$$r_1 = tc(X, Z) \leftarrow arc(X, Z),$$
$$r_2 = tc(X, Z) \leftarrow arc(X, Y) \wedge tc(Y, Z),$$
$$f_1 = arc(a, b),$$
$$f_2 = arc(b, c),$$



zur Berechnung der transitiven Hülle eines azyklischen Graphen liefert die *Bottom–Up–Auswertung* die folgenden neuen Fakten:

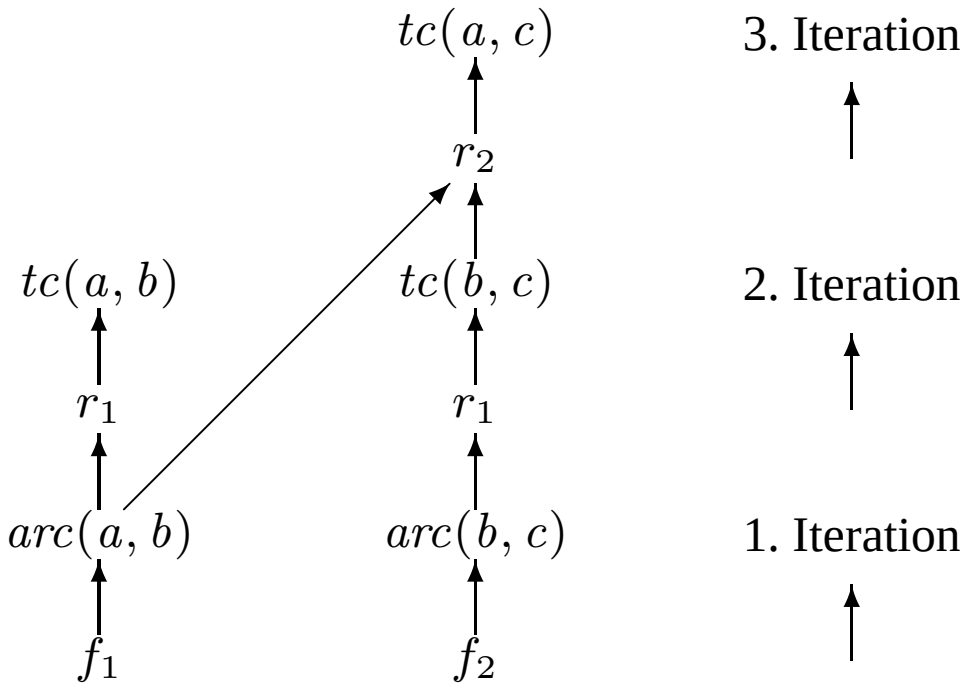
1. Iteration: $arc(a, b),\ arc(b, c),$
2. Iteration: $tc(a, b),\ tc(b, c),$
3. Iteration: $tc(a, c).$

Die Auswertung terminiert nun, da die 4. Iteration keine neuen Fakten mehr liefert.

Beim Forward Chaining wird z.B. in Iteration 3 mittels der Grundinstanz

$$tc(a, c) \leftarrow arc(a, b) \wedge tc(b, c)$$

der Regel r_2 das neue Fakt $tc(a, c)$ abgeleitet, da $arc(a, b)$ bereits in Iteration 1 abgeleitet wurde und $tc(b, c)$ in Iteration 2:



Wir werden nun sehen, daß hier das Backward Chaining dieselben Ergebnisse produziert wie das Forward Chaining.

- Der SLD–Baum zum Goal $\leftarrow tc(X, Z)$ aus dem Backward Chaining hat 3 success–Äste und 4 failure–Äste.

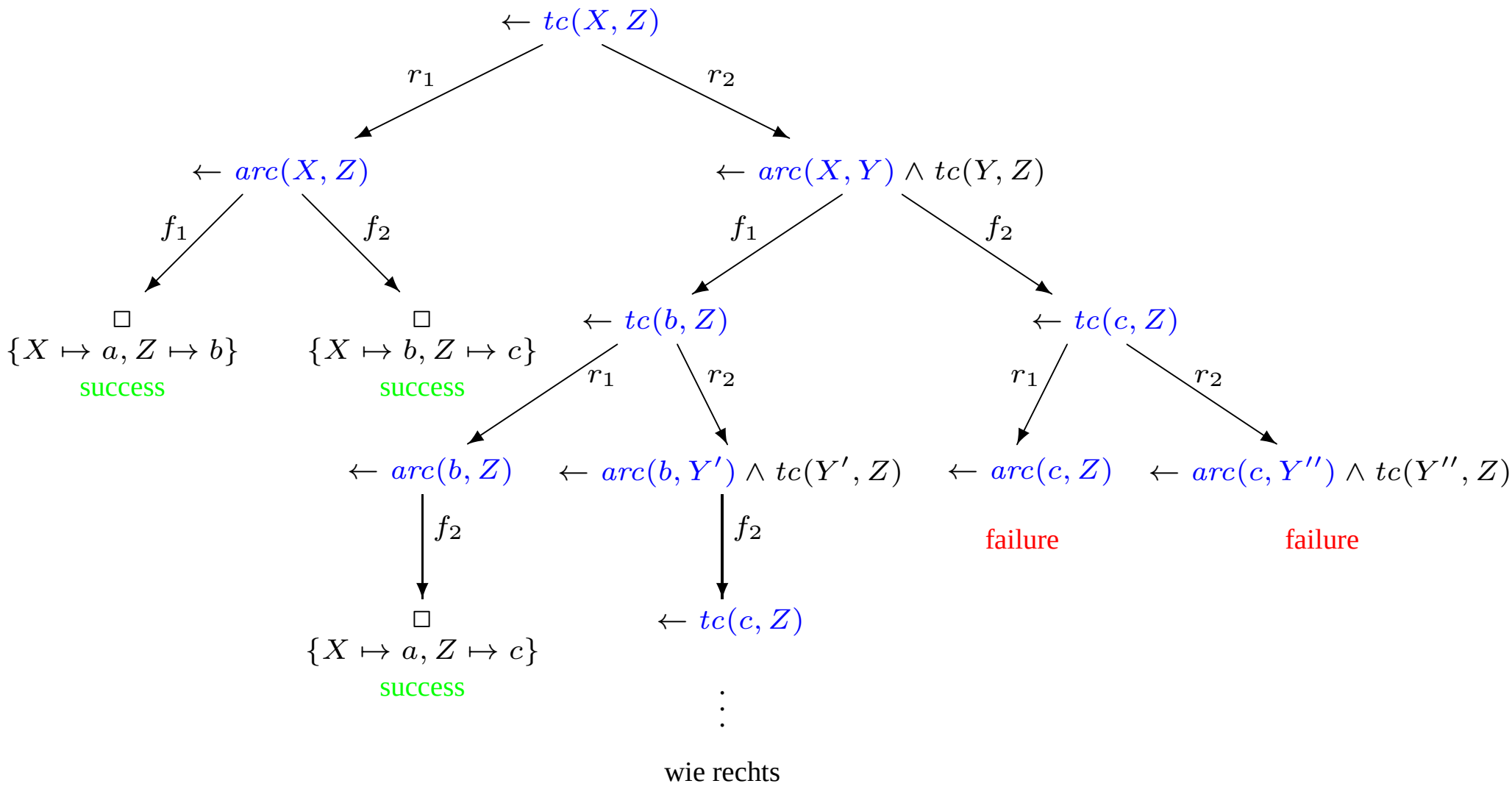
  Die success–Äste berechnen genau die 3 Substitutionen zu den beim Forward Chaining berechneten tc–Atomen A_i:

  $$\theta_1 = \{X \mapsto a, Z \mapsto b\} \text{ zum Atom } A_1 = tc(a, b),$$
  $$\theta_2 = \{X \mapsto b, Z \mapsto c\} \text{ zum Atom } A_2 = tc(b, c),$$
  $$\theta_3 = \{X \mapsto a, Z \mapsto c\} \text{ zum Atom } A_3 = tc(a, c).$$

- Der SLD–Baum zum Goal $\leftarrow arc(X, Z)$ berechnet genau die 2 Substitutionen zu den beim Forward Chaining in der ersten Iteration berechneten arc–Atomen.

Das Backward Chaining terminiert hier auch unter Backtracking, da die SLD–Bäume keine unendlichen Äste enthalten.

SLD–Baum


$\leftarrow tc(X, Z)$
r_1
r_2
$\leftarrow arc(X, Z)$
$\leftarrow arc(X, Y) \wedge tc(Y, Z)$
f_1
f_2
f_1
f_2
$\square$
$\{X \mapsto a, Z \mapsto b\}$
success
$\square$
$\{X \mapsto b, Z \mapsto c\}$
success
$\leftarrow tc(b, Z)$
$\leftarrow tc(c, Z)$
r_1
r_2
r_1
r_2
$\leftarrow arc(b, Z)$
$\leftarrow arc(b, Y') \wedge tc(Y', Z)$
$\leftarrow arc(c, Z)$
$\leftarrow arc(c, Y'') \wedge tc(Y'', Z)$
f_2
f_2
failure
failure
$\square$
$\{X \mapsto a, Z \mapsto c\}$
success
$\leftarrow tc(c, Z)$
wie rechts

Terminierung

- Das Forward Chaining von DATALOG terminiert immer, da keine Funktionssymbole erlaubt sind.
- Das Backward Chaining von PROLOG terminiert nicht immer, und man muß bei der PROLOG–Programmierung generell auf Terminierungsaspekte achten. Zum Beispiel terminieren PROLOG–Anfragen zur Berechnung der transitiven Hülle zyklischer Graphen nicht ohne weiteres.
- Bei DATALOG kann Nicht–Terminierung nur in der Erweiterung um Funktionssymbole auftreten. Dann gibt es Beispiele, wie etwa

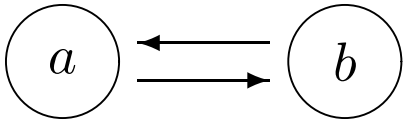
  $$P = \{\ p(f(X)) \leftarrow p(X),\ \ p(a)\ \},$$

  für die einzelne PROLOG–Anfragen terminieren, die komplette DATALOG–Auswertung aber nicht, da unendlich viele Fakten erzeugt würden.

Beispiel (Terminierung)

Für das DATALOG–Programm $P_{tc} = \{ r_1, r_2, f_1, f_2 \}$ mit

$$r_1 = tc(X, Z) \leftarrow arc(X, Z),$$
$$r_2 = tc(X, Z) \leftarrow arc(X, Y) \wedge tc(Y, Z),$$
$$f_1 = arc(a, b),$$
$$f_2 = arc(b, a),$$



zur Berechnung der transitiven Hülle eines zyklischen Graphen liefert die *Bottom–Up–Auswertung* die folgenden neuen Fakten:

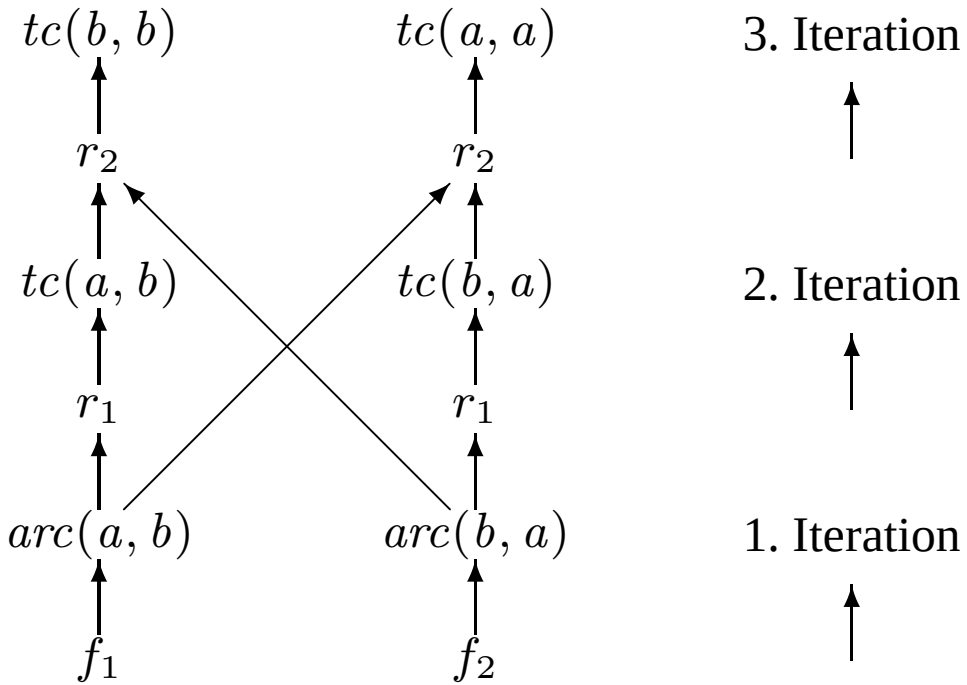
1. Iteration: $arc(a, b),\ arc(b, a),$
2. Iteration: $tc(a, b),\ tc(b, a),$
3. Iteration: $tc(a, a),\ tc(b, b).$

Die Auswertung terminiert nun, da die 4. Iteration keine neuen Fakten mehr liefert.

Beim Forward Chaining wird z.B. in Iteration 3 mittels der Grundinstanz

$$tc(a, a) \leftarrow arc(a, b) \wedge tc(b, a)$$

der Regel r_2 das neue Fakt $tc(a, a)$ abgeleitet, da $arc(a, b)$ bereits in Iteration 1 abgeleitet wurde und $tc(b, a)$ in Iteration 2:



Das Backward Chaining von PROLOG liefert für die folgende Anfrage – bei Backtracking – immer wieder abwechselnd die Antworten $Y \mapsto b$ und $Y \mapsto a$:

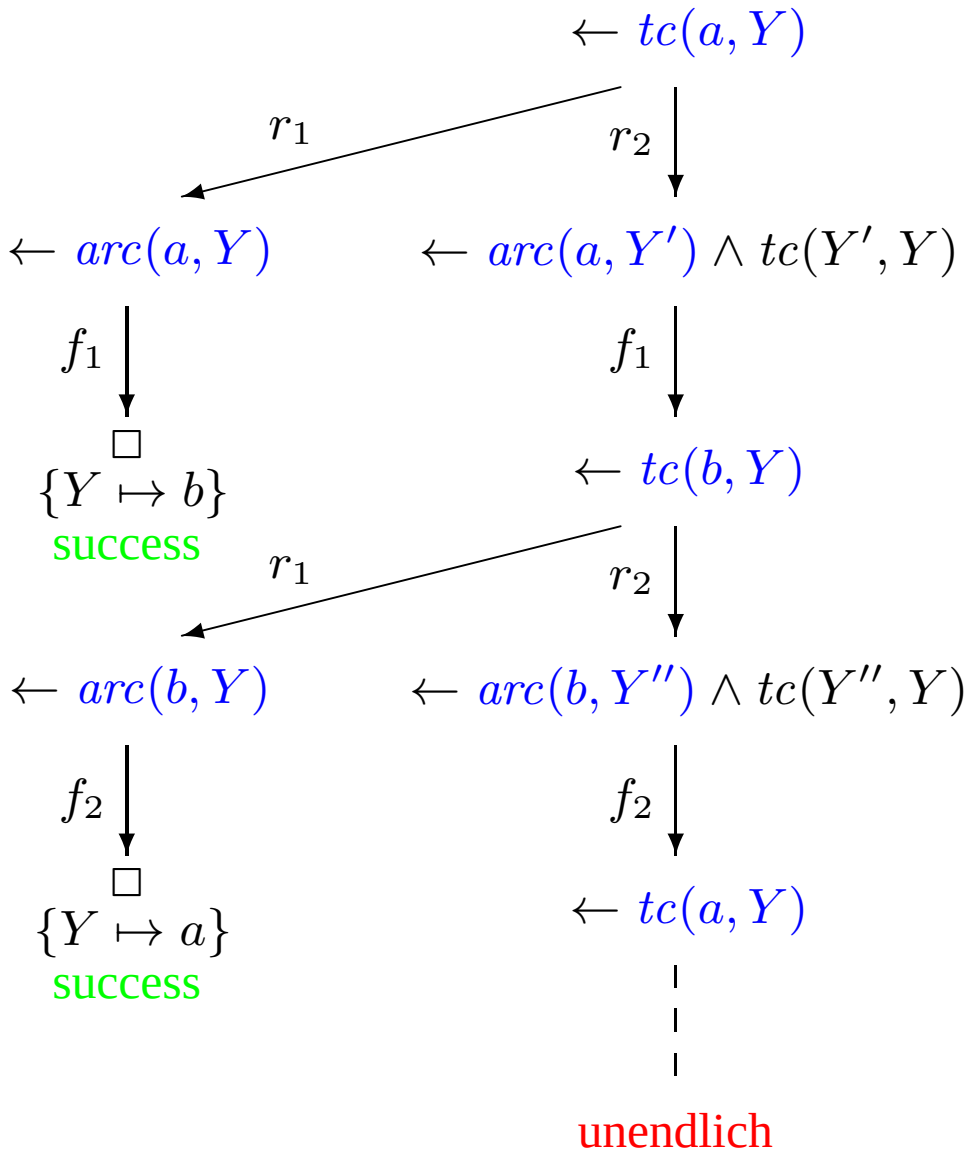
```
?- tc(a, Y).
```

Die folgende PROLOG–Anfrage terminiert nicht, da der zugehörige SLD–Baum einen unendlichen Ast hat, der immer wieder dieselben Anfragen $\leftarrow tc(a, Y)$ und $\leftarrow tc(b, Y)$ stellt:

```
?- findall( Y,
      tc(a, Y),
      Ys ).
```

Falls man sich merken würde, welche Anfragen schon gestellt wurden (Tabling, Memoing), dann würde die Top–Down–Auswertung allerdings terminieren. Tabling erfolgt aber nur in der PROLOG–Erweiterung XSB–PROLOG.

SLD–Baum


$\leftarrow tc(a, Y)$
r_1
r_2
$\leftarrow arc(a, Y)$
$\leftarrow arc(a, Y') \wedge tc(Y', Y)$
f_1
f_1
$\square$
$\{Y \mapsto b\}$
success
$\leftarrow tc(b, Y)$
r_1
r_2
$\leftarrow arc(b, Y)$
$\leftarrow arc(b, Y'') \wedge tc(Y'', Y)$
f_2
f_2
$\square$
$\{Y \mapsto a\}$
success
$\leftarrow tc(a, Y)$
unendlich

Beispiel (Terminierung bei Funktionssymbolen)

Wir betrachten das Logikprogramm $P = \{\ p(a),\ \ p(f(X)) \leftarrow p(X)\ \}$.

- Hier leitet das Forward Chaining unendlich viele Fakten ab:
  $p(a),\ p(f(a)),\ p(f(f(a))), \ldots$
- Dagegen terminiert das Backward Chaining für Anfragen ohne Variablen. Ausgehend von einer Anfrage, z.B. $Q = \leftarrow p(f(f(a)))$, wird aufgrund der zweiten Regel eine einfachere Unteranfrage, $Q' = \leftarrow p(f(a))$, mit einem Funktionssymbol f weniger gestellt, etc.
- Für eine Anfrage mit Variablen wie z.B. $Q = \leftarrow p(X)$ leitet das Backward Chaining unter Backtracking ebenfalls unendlich viele Antworten ab:
  $X \mapsto a,\ X \mapsto f(a),\ X \mapsto f(f(a)), \ldots$
  Würde man die Reihenfolge der Regeln vertauschen, so würde man eine Dauerschleife erhalten, die keine Antworten produziert.

Offensichtlich gibt es auf Anfragen, die von a verschiedene Konstanten bzw. von f verschiedene Funktionssymbole enthalten, keine Antworten.

Wir betrachten das Logikprogramm $P = \{\ p(f(f(a))),\ \ p(X) \leftarrow p(f(X))\ \}$.

- Hier terminiert das Forward Chaining; es leitet die Fakten $p(f(f(a))),\ p(f(a)),\ p(a)$ ab.
- Dagegen terminiert das Backward Chaining selbst für Anfragen ohne Variablen meist nicht. Für die Anfrage $Q = \leftarrow p(f(f(f(a))))$ wird aufgrund der zweiten Regel die komplexere Unteranfrage mit einem Funktionssymbol f mehr: $Q' = \leftarrow p(f(f(f(f(a)))))$, gestellt, etc.
- Für eine Anfrage mit Variablen wie z.B. $Q = \leftarrow p(X)$ leitet das Backward Chaining unter Backtracking zuerst ebenfalls die Antworten $X \mapsto f(f(a)),\ X \mapsto f(a),\ X \mapsto a$ ab. Danach gerät es allerdings in eine Dauerschleife, in der immer komplexere Unteranfragen gestellt werden. Würde man die Reihenfolge der Regeln vertauschen, so würde man sofort eine Dauerschleife erhalten, ohne Antworten zu produzieren. Die genaue Analyse dieses Verhaltens würde an dieser Stelle zu weit führen.

Ein praktisches Beispiel, in dem der Rumpf eine einfachere Termstruktur enthält als der Kopf, ist die zweite Regel von `member/2`:

```
member(X, .(X, _)).
member(X, .(_, Xs)) :-
   member(X, Xs).
```

Die Listennotation `[X|Xs]` entspricht bekanntlich einer Termstruktur `.(X, Xs)` mit dem binären Listenfunktor "`.`".
Der rekursive Aufruf im Rumpf der zweiten Regel enthält im zweiten Argument eine einfachere Termstruktur `Xs` als der Kopf, in dem `.(_, Xs)` steht.

Hier terminiert die SLD–Resolution von PROLOG für Anfragen mit einem gebundenen zweiten Argument.

Selbst wenn man die Regeln mit einem endlichen Domänen–Prädikat bereichtsbeschränkt machen würde, dann würde das Forward Chaining von DATALOG$^{\text{fun}}$ unendlich viele Fakten (Listen mit Wiederholungen) ableiten.

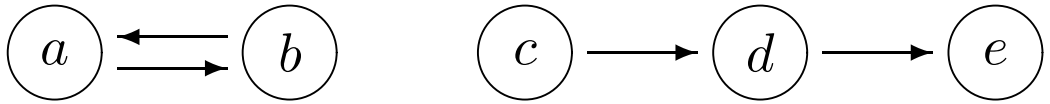
Effizienz

- Das Forward Chaining von DATALOG wird mengen–orientiert durchgeführt und ist deswegen effizienter als das tupel–orientierte Backward Chaining von PROLOG.
- DATALOG kann auf der Basis hoch–optimierter Indexstrukturen relationaler Datenbanken ausgewertet werden.
- Außerdem gibt es eine Reihe von Optimierungsmethoden für DATALOG zur Vermeidung von Redundanzproblemen.
- Andererseits leitet DATALOG immer alle möglichen Fakten her.
  Die zielgerichtet Auswertung von PROLOG ist zur Beantwortung konkreter Anfragen deswegen zunächst effizienter.
- Die DATALOG–Auswertung kann aber mittels der Magic Sets–Methode ebenfalls zielgerichtet durchgeführt werden (siehe Kapitel 5).

Beispiel (Effizienz)

Wenn wir das DATALOG–Programm P_{tc} um die Fakten

$$f_3 = arc(c, d),$$
$$f_4 = arc(d, e),$$



erweitern, die mit der Anfrage $\leftarrow tc(a, Y)$ nicht zu tun haben, so terminiert die Bottom–Up–Auswertung immer noch, aber sie liefert folgende für die Anfrage irrelevanten Fakten:

1. Iteration: $arc(c, d),\ arc(d, e),$
2. Iteration: $tc(c, d),\ tc(d, e),$
3. Iteration: $tc(c, e).$

Die Top–Down–Auswertung der Anfrage $\leftarrow tc(a, Y)$ liefert immer noch denselben SLD–Baum. Sie arbeitet immer nur mit relevanten Fakten – sie ignoriert also die Fakten f_3 und f_4 – und liefert auch nur relevante Antworten, aber sie terminiert unter Backtracking nicht.

Sprachumfang: Disjunktion und Konjunktion

- Der Regelrumpf von PROLOG–Regeln ist normalerweise konjunktiv:

  ```
  a :- b, c.
  ```

  In DATALOG haben Regeln immer einen konjunktiven Regelrumpf.

  PROLOG–Regeln mit Disjunktion im Regelrumpf kann man als mehrere dazu äquivalente Regeln schreiben.
  Allerdings werden dabei aus einer komplexeren Regel wie

  a `:-` `(`b_1`;` $\ldots$ `;`b_n`),(`c_1`;` $\ldots$ `;`c_m`)`

  schon $n \cdot m$ Regeln a `:-` b_i`,`c_j`.`
  Mit Hilfsprädikaten könnte man auf $n + m + 1$ Regeln kommen:

  a `:-` b`,`c`.`

  b `:-` b_i`.` $(1 \leq i \leq n)$ $\quad$ c `:-` c_j`.` $(1 \leq j \leq m)$

- PROLOG und DATALOG$^{\text{not}}$ erlauben Default Negation im Regelrumpf.

- PROLOG–Regeln haben immer einen atomaren Regelkopf.

  Regeln mit einem konjunktiven Regelkopf wie

  ```
  a,b :- c.
  ```

  kann man als zwei dazu äquivalente Regeln schreiben:

  ```
  a :- c.
  b :- c.
  ```

  Regeln mit einem disjunktivem Regelkopf wie

  ```
  a;b :- c.
  ```

  sind weder in PROLOG noch in DATALOG möglich.

- Die disjunktive Erweiterung DISLOG der Logikprogrammierung erlaubt Disjunktion im Regelkopf (unvollständiges Wissen) und Default Negation im Regelrumpf.

Sprachumfang: Default Negation

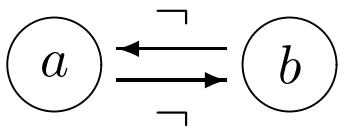
- Negation ohne Rekursion (stratifizierte Negation) ist weniger kompliziert:

  $r_1 = a \leftarrow \mathsf{not}\ b, \quad r_2 = b.$

  Da r_2 das Atom b ableitet, leitet r_1 das Atom a nicht ab (Negation–as–Failure). Es gibt genau eine sinnvolle Interpretation: das *perfekte* Modell, in dem b wahr und a falsch ist.

- Negation in Verbindung mit Rekursion ist kompliziert:

  $a \leftarrow \mathsf{not}\ b, \quad b \leftarrow \mathsf{not}\ a.$



  Es gibt drei sinnvolle Interpretationen: in den beiden *stabilen* Modellen ist a wahr und b falsch bzw. b wahr und a falsch, und im *wohlfundierten* Modell sind a und b undefiniert.

- Es gibt sogar praktische Anwendungen mit Negation und direkter Rekursion:

  $a \leftarrow \mathsf{not}\ a.$

Die Negationsbehandlung (Semantik und Auswertungsmethoden) werden wir in den Kapiteln 6 und 7 kennenlernen.

DATALOG/PROLOG–Systeme

Für die unterschiedlichen Untersprachen bzw. Erweiterungen von PROLOG sind unterschiedliche Auswertungsmethoden/Systeme besser geeignet:

Sprache	System
nicht–rekursives PROLOG	PROLOG
rekursiv–absteigendes PROLOG ohne Negation	PROLOG
DATALOG	DATALOG
stratifiziertes $\text{DATALOG}^{\text{not},\text{agg}}$	DATALOG
$\text{DATALOG}^{\text{not}}$	Smodels
$\text{DATALOG}^{\text{not},\vee}$	dlv

Eigenschaften der bekannten Methoden/Systeme:

- Rekursion kann in PROLOG zur Nicht–Terminierung führen
- Aggregation ist nur in einigen DATALOG–Systemen wie CORAL, $\mathcal{LDL}$, $\mathcal{L}$ola und ADITI möglich.
- Die zielgerichtete und effiziente Auswertung von Anfragen ist bei DATALOG mittels der Magic Sets–Methode möglich.
- Falls Negation und Disjunktion beliebig eingesetzt werden sollen, so muß man Systeme wie Smodels oder dlv verwenden.

DATALOG* integriert viele der bekannten Methoden/Systeme. Es ist auf der Basis von PROLOG implementiert und kann deswegen bidirektional mit PROLOG kommunizieren. Für DATALOG$^{\mathsf{not},\vee}$–Anfragen wird dlv eingebunden.