Alternative Programmierkonzepte (T3INF4271)

Logische Programmierung

00
Allgemeines
zur Vorlesung

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Antonius van Hoof

AVH 2021

Lerninhalte (Laut Modulplan)

- **Logik und Programmierung**
- **Die Programmiersprache PROLOG**
- **Unifikation**
- **Automatische Beweisverfahren**
- **Constrained Based Programming / Optimierung**
- **Wissensrepräsentation**
- **Expertensysteme**
- **Suche / Planung**
- **Language Processing / DSL**
- **Dazu noch:**
  - **Data Management**
  - **Web Development**
  - **Machine Learning / Probabilistic Prolog**

Benutzung dieser Folien ist nur im Rahmen Ihres DHBW-Studiums erlaubt

Literatur (Die Prolog Klassiker)

- Programming in Prolog: Using the ISO Standard 5th edition,
  Clocksin, W., Mellish, C.S. (1981, 2003),
  Heidelberg, Springer
- Clause and Effect: Prolog Programming For The Working Programmer,
  Clocksin, W., (1997),
  Heidelberg, Springer
- The Art of PROLOG: Advanced Programming Techniques,
  Sterling, L., Shapiro, E., (1986),
  Cambridge Mass., MIT Press
- The Craft of Prolog,
  O'Keefe, R. (1990),
  Cambridge Mass., MIT Press
- PROLOG Programming for Artificial Intelligence,
  Bratko, I., (1986, 2012),
  Harlow, Pearson

Literatur

- Logic Programming with Prolog (Second Edition),
  Max Bramer, (2010, 2013),
  Heidelberg, Springer
- Language Processing with Perl and Prolog: Theories, Implementation, and Application,
  Pierre M. Nugues, (2014),
  Heidelberg, Springer
- Constraint-Programmierung : Grundlagen und Anwendungen,
  Thom Frühwirth; Slim Abdennadher, (1997),
  Heidelberg, Springer
- Constraint Handling Rules,
  Thom Frühwirth, (2009),
  Cambridge University Press
- Constraint logic programming using ECLiPSe,
  Krzysztof R. Apt and Mark Wallace, 2007,
  Cambridge University Press

Literatur (Der Hintergrundklassiker)

- Logic for Problem Solving,
  Kowalski, R., (1979, 2014),
  [2014, Frühwirth, T., Hrg.],
  Norderstedt, Book on Demand Verlag

Lesenswertes und Sehenswertes im Web

- http://www.learnprolognow.org/
  Learn Prolog Now! is an introductory course to programming in Prolog.
  (auch als Buch: Blackburn, Bos, Striegnitz, (2006),Learn Prolog now!, Kings College, London)

- https://www.risc.jku.at/education/courses/ws2019/logic-programming/
  Introduction to Prolog, Temur Kutsia

- https://www.metalevel.at/prolog
  The Power of Prolog, online book and videos (YouTube-Channel) on modern developments in prolog.

- https://www.youtube.com/watch?v=G_eYTctGZw8
  Michael Hendricks on Production Prolog
  (Using Prolog for real world application development)

(Alle Links zuletzt geprüft am 21.09.2021)

Prüfungsleistung

- **Aufgaben während und am Ende der Vorlesungsreihe**
- **Sind Prolog Programmieraufgaben**
- **Sind als Prolog Quelldateien (SWI Prolog) gezipt einzureichen (zip oder 7zip)**
- **Jede Aufgabe als selbstständige Quelldatei**
- **Jede Quelldatei muss als Header-Kommentar enthalten:**
  - Autor
  - Aufgabenr.
  - Angaben zu Hilfestellung Dritter (z.B. Kommilitonen, Autoren, Webquellen)
- **Beurteilt werden:**
  - Korrektheit und Vollständigkeit(!) der Lösung
  - Angemessenheit der Programme bzgl. der Möglichkeiten von Prolog (Qualität)
  - Maß der Dokumentation des Codes

Alternative Programmierkonzepte (T3INF4271)

Logische Programmierung

01
Einführung

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Antonius van Hoof

Was ist Logische Programmierung?

- **Robert Kowalski:**
  - **Algorithm = Logic + Control**
- **Benutzung eines logischen Kalküls**
  - Zur Darstellung der dem Programmierer bekannten Sachverhalten
    → das Programm
  - Zur Ableitung neuer logischen Aussagen
    → das Resultat, die Ausgabe
- **Control: Ableitungsverfahren (Beweisverfahren)**
  - i.A. Resolution (Widerspruchsverfahren, Robinson, 1965)
  - Weitere Techniken um eine praktische Sprache zu erhalten (Entscheidbarkeit, Effizienz, Seiteneffekte, usw.)
- **Wir liefern die Logic – Prolog kümmert sich um Control**

Logische Programmierung in der Praxis

- **Durchgesetzt hat sich in der Praxis die Entscheidung für:**
  - Eine Untermenge der Klausel der Prädikatenlogik der ersten Ordnung, sogenannte **Horn Klausel** oder auch **Definite Klausel**
  - Ein Resolutionsverfahren für diese Klausel, das spezielle Eigenschaften hat (linear, Selektionsfunktion; wird später erklärt)
  - Einen speziellen Umgang mit fehlenden Informationen (Closed World Assumption, Negation as Failure)
- **Die diverse Implementierungen dieser Entscheidung machen die Familie der Prolog-Sprachen aus**
  - Davon gibt es diverse (kommerzielle und freie Prologs)
  - Wir benutzen in der Vorlesung i.A. SWI-Prolog
    weil frei, gut dokumentiert, breite Anwenderkreis

Prolog

- **Der Name "Prolog" steht für "PROgramming in LOGic"**
  - Prolog's Entwicklungsgeschichte beruht auf Forschung über Theorembeweiser und automatische Deduktionsverfahren aus den 1960er und 1970er Jahren
- **Das "erste" Prolog war "Marseille Prolog" entwickelt von Alain Colmerauer (1972)**
  - Es wurde explizit entwickelt zur Unterstützung natürlicher Sprachverarbeitung
  - Erste Prolog-*Interpreter* wurde in ALGOL W realisiert (Marseille, Colmerauer)
  - Der erste *Compiler* wurde von David H. D. Warren in Edinburgh entwickelt. Dieser hatte als Zielsprache die des Logik-Prozessors WAM, Warren Abstract Machine (Warren 1983), eine virtuelle Maschine. Das gleiche gilt auch heute noch für die meisten Prolog-Compiler
  - WAM Implementierung in C oder ähnlicher Sprache

Ein erster Vergleich der Sprachparadigmen: Concatenate lists a and b

Imperatives/OO Paradigma

```
list procedure cat(list a, list b)
{
   list t = list u = copylist(a);
   while (t.tail != nil) t = t.tail;
   t.tail = b;
   return u;
}
```

Funktionales Paradigma

```
cat(a,b) ≡
 if a = nil then b
else cons(head(a), cat(tail(a),b))
```

Logisches/Deklaratives Paradigma

```
cat([], Z, Z).
cat([H|T], L, [H|Z]) :- cat(T, L, Z).
```

Nach W. Clocksin

Was macht Prolog anders als andere Sprachen?

- **Prolog kennt neben wenigen Funktionen vor allem Relationen**
  - Relationen haben, anders als Funktionen, keine Richtung
    - Man kann rein relationale Programme unterschiedlich benutzen
  - Paradigma der **deklarativen** Programmierung
  - Relationsargumente sind typefrei
    (Typisierung ist jedoch möglich und manchmal nützlich)
- **Prolog Variablen sind unveränderlich (immutable)**
  - Funktionieren wie mathematische Variablen
  - Zuweisung (und Neu-Zuweisung) von Werten ist nicht möglich (sondern lediglich (auch teilweise) **Instanzierung** von Variablen mittels konkreter Werten)
  - Vermeidet Seiteneffekte (z.B. gut für nebenläufige Progr.)
- **Prolog ist Homo-Ikonisch** (siehe: https://en.wikipedia.org/wiki/Homoiconicity)
  - Kein Unterschied zwischen Daten und Meta-Daten
  - Meta-Interpretation ist denkbar einfach
    Triska: "You can define an interpreter for pure Prolog in two lines of Prolog code"
  - Programme können Aussagen über Programme machen (Programverifikation, Code Quality Control)

Was macht Prolog anders als andere Sprachen?

- **Prolog hat keine klassische Kontrollstrukturen**
  - Kein if, case, for, while
  - Primat der Rekursion (mit Tail Call Optimization, TCO)
- **Prolog kann aufgrund von Unifikation mit partiellen Datenstrukturen (und aufgrund der Homo-Ikonizität mit partiellen Programmen) arbeiten**
  - Technik der Differenz-Strukturen und der DCGs
- **Prolog lässt leicht höhere Ordnung Programmierung zu**
  - Eigenschaften und Beziehungen von Relationen
  - Große Mächtigkeit
- **Prolog Programme ≈ RDB mit Regeln**
  - Dynamische Prädikate (assert & retract)
  - Möglichkeit, Programme zur Laufzeit zu ändern
    („with great power comes great responsibility“)

Gemeinsamkeiten mit funktionalen Sprachen

- **Prolog lässt Arbeit in einer REPL zu**
  (Ist eine Gemeinsamkeit mit funktionalen Progr. Sprachen)
  - Read, Evaluate, Print, Loop → Interaktivität
  - Makro Programmierung / Syntactic Sugar
  - Sowohl für Programme als Abfragen
- **Arbeit mit (Quasi-)Closures, Map & Fold**
  - Beispiel:

    ```
    ?- maplist(plus(1),[1,2,3],L).
    L = [2, 3, 4].
    ```

  - Aber nur quasi, deswegen Extras die nicht funktional sind:

    ```
    ?- maplist(plus(N),[1,2,3],[4,5,6]).
    N = 3.

    ?- maplist(plus(2),L,[3,4,5]).
    L = [1, 2, 3].
    ```

- **Co-Routining** (asynchrone Funktionsausführung) **und Lazy Evaluation**

Weiterentwicklung der Logischen Programmierung

- **Aus Gründen der Effizienz wurde in Prolog die Entscheidung getroffen, Arithmetik nicht relational, sondern funktional zu implementieren**
  - D.h. Funktionsargumente müssen bei Aufruf der Funktion alle vollständig instanziiert sein
  - Damit ist die Verwendung logischer (Un-)Gleichungen nicht mehr möglich
    - Bsp.: 5 is (Y + 3) liefert einen Laufzeitfehler wenn Y bei Aufruf keinen konkreten numerischen Wert hat
    - Besser wäre, wenn abgeleitet würde im Beispielsfall, dass Y = 2
- **Erweiterungen von Prolog (als Bibliothek oder direkt in Prologs Kontrollstrukturen implementiert) machen jedoch relationale Arithmetik möglich (clpfd, clpqr (clpb))**
  → Constraint Programming
- **Constraint Programming ist inzwischen eine eigene Disziplin (auch in anderen Sprachen unterstützt)**
  - Passt aber am Besten in Prolog & Co

Eigenschaften von guten Prolog Programmen

- **Gute Prolog Programme sind:**
  - Kurz (im Vergleich zu anderen Sprachen)
  - Dadurch besser überschaubar/überprüfbar
  - Möglichst Pure im Kern
    - Vollständig deklarativ
    - Monoton
    - Volle Nutzung von Nicht-Determinismus
    - Dadurch vielseitig nutzbar
  - Effizient, vor allem im Laufzeitverhalten
    - Natürlich kann auch Prolog nichts ändern an die asymptotische Eigenschaften des zu lösenden Problems
  - Schön!

Richard O'Keefe: **"Elegance is not optional"**
[The Craft of Prolog, S.4ff]

Was hält Programmierer von Prolog fern?

- **Die logische Denkart, die vielen Informatikern fremd ist, und damit einhergehend eine steile Lernkurve**
- **Ein kleines Ökosystem**
  - Kleine Community der Programmierer
  - Wenig frei verfügbare Programm Code
  - Nur langsame Weiterentwicklung von Compilern und Werkzeugumgebungen
- **Eine ungewöhnliche Syntax**
- **Programversagen ist oft schwierig zu debuggen**
  - Was tun, wenn statt eine erwartete Antwort das „false.“ kommt
  - Gute Debugger aber keine direkt aussagekräftige Stack Traces
  - Allerdings extra: deklaratives Debuggen und Testen
- **Laufzeitperformanz ist manchmal ein Problem**
  (öfters als bei anderen Sprachen)
  - Dies aber meistens, weil „falsch“ programmiert wird (siehe vorige Folie)
  - (Und der Compiler noch optimiert werden kann. U.a. damit verdienen kommerziellen Prologs ihr Geld)

Wird Prolog dann wirklich (d.h. außerhalb der Science Community) benutzt?

- **Ja!! Aber recht exklusiv: wird wg. „competitive edge" wenig über gesprochen/publiziert**
- **Beispiele:**
  - NASA: speech application (Clarissa):
    Procedure browser in support of ISS Astronauts
  - Telecom: Ericsson AB
    Network Resource Manager for configuring and managing multi-vendor IP Backbone networks
  - Biotech: Pyrosequencing AB (biotage.com)
    sequencing in genetics, drug discovery, microbiology, SNP and mutation analysis, forensic identification using mtDNA, pharmacogenomics, and bacterial and viral typing
  - Knowledge Elicitation: Kyndi (kyndi.com)
    Automatic Text Reading, Cognitive Search
  - Finance: SecuritEase (securitease.com)
    Business Rule Management Systems, Stock Broking System for world-wide operating banks
    [see a slide presentation: https://dtai.cs.kuleuven.be/CHR/files/Elston_SecuritEase.pdf]

Praktische Übung (nicht zum Einreichen)

Um für sich selbst einzuschätzen, wie gut Ihre Prolog-Kenntnisse sind, sollen Sie folgendes kleine Programm schreiben (Sie dürfen zur zweit zusammenarbeiten):

- Schreiben Sie eine Prozedere namens **`isa_set/1`** um zu bestimmen, ob eine Liste eine Menge ist (nach mathematischer Definition). Zusätzliche Anforderung soll sein, dass eine Menge sortiert ist. [Sie sollen also mit "ordered sets" arbeiten]
- Schreiben Sie eine Prozedere **`list_2_set/2`** um eine Liste von Elementen in eine Menge zu überführen
- Implementieren Sie Prozederen für die üblichen Operation auf Mengen:
  `set_set_union/3, set_set_intersection/3, set_subtract_diff/3`
- Zusätzlich: Implementieren Sie eine Prozedere für Untermengen, **`subset_set/2`** und Potenzmenge **`set_powerset/2`**

- Sie dürfen in Prolog mitgelieferten Prozederen benutzen – **aber keine die bereits explizit für Mengen oder geordneten Mengen definiert sind.**
- Machen Sie Notizen zu den Problemen, die Sie bei der Ausführung der Übung begegnen: Benennen und kurz erläutern, damit wir sie später besprechen können

Alternative Programmierkonzepte (T3INF4271)

Logische Programmierung

02
Prolog
Syntax & Semantik

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Antonius van Hoof

AVH 2021

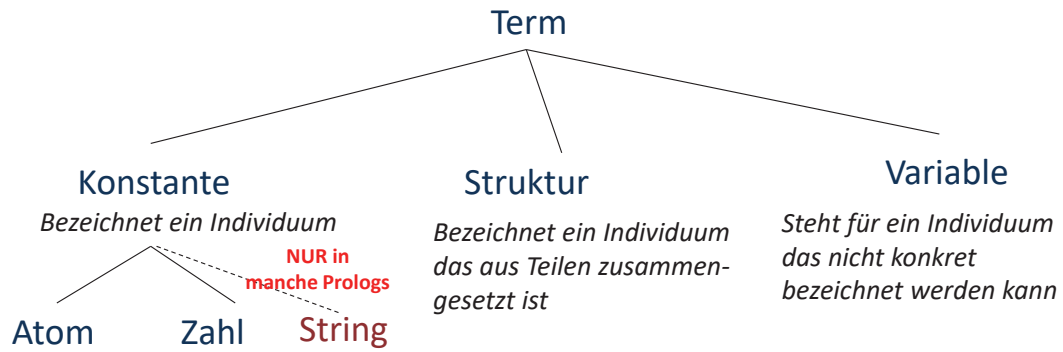
Entwicklung der Prolog Syntax und Sprachumfang

- **Erste Definition der Syntax, sowie von "eingebauten" Prädikaten durch Colmerauers Marseille Prolog**
- **Populär wurde jedoch eine effiziente Prolog Implementierung in C, entwickelt an der Uni Edinburg, C-Prolog bzw. Edinburgh-Prolog**
  - Abweichend von Marseille Syntax und Built-ins
  - Wurde aufgrund seiner Verbreitung in 80-er Jahren de-facto Standard
  - Grundlage für viel spätere Standardisierung
- **1995: Standard** ISO/IEC 13211-1
  - Letzte offizielle Ausgabe: ISO/IEC 13211-2:2000 Normung von Module-Support
  - Wurde seither leicht korrigiert/erweitert (Cor.1:2007, Cor.2:2012, and Cor.3:2017)
- **Viele neue Features und Verbesserungen NICHT im Standard!!**
- **Neue de-facto Standards: Sicstus und SWI**

Prolog Syntax

- **Prolog-Programme bestehen aus Fakten und Regeln**
  - Hierzu nachher mehr. Fürs erste reicht Ihr Wissen aus der Informatik-Vorlesung
- **Fakten und Regeln sind aufgebaut aus Termen**
- **Termen werden zusammengesetzt aus Schriftzeichen (Characters), die zu unterscheiden sind in**
  - Buchstaben (Groß und Klein),
  - Ziffern, und
  - Sonderzeichen
- **Ein Term ist Konstante, ein Variable oder eine Struktur**

Komplette Syntax von Prolog Termen



Atom	Zahl	String
alpha17	0	"hello"
john_smith	1	
dyspepsia	57	
+	1.618	
=/=	2.04e-27	
'12Q&A'	-13.6	
:-		
?-		

Struktur	Variable
likes(john, mary)	X
book(dickens, Z, cricket)	Gross_pay
f(x)	Diagnosis
[1, 3, g(a), 7, 9]	_257
-(+(15, 17), t)	_
15 + 17 - t	

Nach W. Clocksin

Atom

- **Ein Atom beginnt immer mit einem Kleinbuchstaben und kann gefolgt werden von weiteren Buchstaben und Zahlen und das Sonderzeichen "_".**
- **Alternativ kann eine beliebige Zeichenkette zu Atom deklariert werde mittels einfachen Anführungszeichen.**
- **Als Sonderfall gilt noch, dass eine Ansammlung von manchen Sonderzeichen (allerdings ohne ! Und %) ebenso als Atom gilt. Ein spezieller Sonderfall sind dort ":-" und "?-"** (sollten Ihnen bekannt vorkommen)

- **Prolog-Prädikat zum Testen ob ein Term ein Atom ist: atom/1:**
  - ?- atom(foobar23).
    true.

Number

- **Eine Zahl ist eine Folge von Ziffern mit ggf. Vorzeichen und Punkt. Auch wissenschaftliche Notation ist erlaubt**
- **Prolog kennt Integers und Floats**
  - Beispiele:
    1232 -354 1.2345 -23.897 2.5e+012 1e-014
- **Es gibt in einigen Prologs nicht-ISO supported Rationals (u.a. in SWI Prolog)**
  - Beispiel:
    ?- X = 1r3, Y = 1r7, Z is X+Y.
    X = 1r3, Y = 1r7, Z = 10r21.
- **Prolog-Prädikate für Zahlentest:**
  - number/1
  - integer/1
  - float/1
  - rational/1, rational/3

String
(nicht ISO, aber derzeit z.B. in SWI Prolog und ECLiPSe)

- **Ein String ist alles, was zwischen doppelten Anführungszeichen steht**
  - Beispiel: "Dies ist eine Zeichenkette, sie kann auch Sonderzeichen enthalten"
- **Prolog-Prädikat für Stringtest:**
  - string/1
- **Prolog-Prädikat zum Testen ob Term eine Konstante, also ein Atom oder Zahl (oder String), ist: atomic/1**

- **N.B.: klassische Zeichenkettenbehandlung (ISO):**
  - Syntactic sugar für Liste von Character Codes (Edinburgh Prolog)
  - ?- X = "hello".
    X = [104,101,108,108,111].
  - Alternativ in SWI (und konform Marseille Prolog):
    set_prolog_flag(double_quotes,chars).
    ?- X = "hello".
    X = [h, e, l, l, o].

Variable

- **Eine Variable ist ein Term, der immer mit einem Großbuchstaben oder mit '_' anfängt**
  - Beispiele: Xantippe _var22
- **Variablen die mit _ anfangen werden beim Einlesen von Programmdateien besonders behandelt**
  (Unterdrückung von Warnungen vor Gebrauch von Singleton)
- **Variablen die nur aus '_' bestehen sind anonyme Variablen**
  - Jede _ steht in einem Fakt oder einer Regel für eine *andere* Variable.
    Somit bedeutet foo(_,_) etwas anderes als foo(X,X) oder foo(_X,_X).
  - Falls man sie in einer Abfrage (REPL) benutzt, wird deren Instanziierung in der REPL nicht angezeigt

Variablenprüfung

- **Prolog-Prädikate zum Testen ob ein Term eine Variable ist: var/1, nonvar/1:**
  - ?- var(X).
    true.
  - ?- nonvar(X).
    false.

- **Prolog-Prädikat zum Testen, ob ein Term Variablen enthält oder nicht: ground/1:**
  - ?- ground(foo(bar,baz)).
    true.
  - ?- ground(foo(bar,X)).
    false.

Type Prüfungen

- **Problem: die bisher vorgestellten Type-Prädikate sind nicht monoton:**
  - ?- integer(X), X=1.
    false
  - ?- X = 1, integer(X).
    Erfolg (X = 1.)

- **Ideal wäre eine _si (sufficiently instantiated) version der Prädikate (gibt es derzeit nur in Scryer Prolog)**
  - Würde instantiation errors werfen, statt 'false'

- **Besser für Type-Prüfungen:**
  :- use_module(library(error)). *(can be autoloaded)*
  - das Prädikat must_be/2 checkt und wirft ggf. errors

Strukturen

- **Eine Struktur ist ein compound Term (zusammengesetzter Term)**
- **Eine Struktur besteht aus einen Funktor gefolgt von Komponenten**
- **Die Komponente heißen Argumente**
- **Argumente können selbst wieder *Terme* sein (Bemerke die Rekursivität der Definition)**
- **Die Komponenten sind kommagetrennt und stehen zwischen Klammern**
  - Beispiele:
    foo(bar,baz)
    foo(X,bar(a,Y))
    foo([bar(a,Z)|umu],cheesy)

    (Ja, eine Liste ist auch eine Struktur: wir kommen später darauf zurück)

Strukturen

- **Ein Funktor mit seiner Wertigkeit (Arity) stellt meistens (Ausnahme: Funktionen in arithmetischer Kontext) ein FOPL-Prädikat (d.h. eine Relation) dar**
- **Daher sprechen wir in Prolog von Prädikaten**
  (Bzw. von *eingebauten* Prädikaten für solche Funktoren, die bereits im Sprachumfang enthalten sind)
- **Prolog-Prädikat zum Testen ob Term eine Struktur (ein Compound) ist: compound/1**
- **Weitere Prädikate um Compounds zu analysieren oder zu bauen:**
  - functor/3
  - arg/3
  - =../2

Beispiele

```
?- functor(foo(bar,baz),Functor,Arity).
Functor = foo,
Arity = 2.

?- functor(foo,F,A).
F = foo,
A = 0.          [Bemerkung: somit ist ein Atom implizit auch eine Struktur]

?- arg(2,foo(bar,baz),Arg).
Arg = baz.

?- arg(2,foo(X,Y),umu).
Y = umu.

?- foo(bar,baz) =.. L.
L = [foo,bar,baz].

?- Term =.. [hello,lovely,world].
Term = hello(lovely,world).
```


Funktor, Operator, Präzendenz

- **Funktore können ggf. als Operatoren dargestellt werden. Operatoren können immer als Funktor geschrieben werden.**
  - Beispiel:
    ?- 3 > 2.
    true.

    ?- >(3,2).
    true.

- **Operatoren sind ein- oder zweistellig, infix, präfix oder postfix.**
- **Zur Vermeidung von Klammern haben Operatoren eine Präzedenz (Punkt-Strich-Regel)**

Beispiele für eingebaute Operatoren (Prädikate)

```
Precedence Type  Name
_____________________________________________________________
| 1200 |xfx  |-->, :-                                        |
| 1200 | fx  |:-, ?-                                         |
| 1150 | fx  |dynamic,    discontiguous,     initialization,  |
|      |     |meta_predicate,  module_transparent, multifile,|
|      |     |thread_local, volatile                         |
|      |     |                                               |
| 1100 |xfy  |;, |                                           |
| 1050 |xfy  |->, op*->                                      |
| 1000 |xfy  |,                                              |
|  900 | fy  |\+                                             |
|  900 | fx  |~                                              |
|  700 |xfx  |<, =, =.., =@=,  =:=, =<, ==, =\=,  >, >=, @<, |
|      |     |@=<, @>, @>=, \=, \==, is                      |
|  600 |xfy  |:                                              |
|      |     |                                               |
|  500 | yfx |+, -, /\, \/, xor, ><                          |
|  500 | fx  |?                                              |
|  400 | yfx |*, /, //, rdiv, <<, >>, mod, rem               |
|  200 |xfx  |**                                             |
|  200 |xfy  |^                                              |
|  200 | fy  |+, -, \                                        |
|_____|_____|_______________________________________________|
```


Umgang mit Operatoren

- **Definition prüfen mittels current_op/3**
  - op(?Precedence, ?Type, ?Name). *
  - Beispiele:
    ?- current_op(P,T,:-).
    P = 1200,
    T = fx .

    ?- current_op(P,xfx,Op).
    P = 990,
    Op = (:=)

    ?- current_op(_,_,und).
    false.

* Für die Bedeutung von ?, siehe SWI Prolog Manual Kap. 4.1 Notation of Predicate Descriptions

Umgang mit Operatoren

- **Operatoren selbst (re-)definieren mittels op/3**
  - Op(+Precedence,+Type,:Name). *
  - Beispiel:

```
?- op(600,yfx,und).
true.

?- op(500,yfx,[oder,xoder]).
true.

?- a und b oder c = X und Y.
X = a,
Y = b oder c.

?- a und b oder c = X oder Y.
false.
```

* Für die Bedeutung von + und :, siehe SWI Prolog Manual Kap. 4.1 Notation of Predicate Descriptions

Die spezielle Struktur "Liste" compound term "list"

- **Experiment (in SWI Prolog):**

```
?- functor([a,b,c],Functor,Arity).
Functor = '[|]',
Arity = 2.

?- arg(1,[a,b,c],Val).
Val = a.

?- arg(2,[a,b,c],Val).
Val = [b, c].

?- arg(3,[a,b,c],Val).
false.

?- [a,b,c] = '[|]'(a,'[|]'(b,'[|]'(c,[]))).
true.
```

N.B.: andere Prologs verwenden statt [|] als Funktor den punkt (.)

```
functor([a,b,c],Functor,Arity).
Functor = .
Arity = 2.
```

Die spezielle Struktur "Liste"

- **Die Notation einer Liste mittels eckigen Klammern ist syntactic sugar für die 2-stellige Relation [|]**
- **Es gilt:**
  - Das erste Argument von [|] kann ein beliebiger Term sein
  - Das zweite Argument muss eine Listenstruktur sein
  - [] ist eine besondere Listenstruktur, für die leere Liste

- **Der Tail-Operator | kann in Listen angewendet werden**
  Links von | stehen Elementen der Liste
  Rechts steht eine Liste, die man den Tail der ursprüngliche Liste nennt
  - Beispiele:

```
?- [a,b,c] = [a|[b,c]].
true.

?- [a,b,c] = [a,b|[c]].
true.

?- [a,b,c] = [a,b,c|[]].
true.

?- [a,b,c] = [a|[b|[c|[]]]].
true.
```


Die spezielle Struktur "Pair"

- **Kann man darstellen mit dem infix-Operator (-)/2**
- **Beispiele:**
  - A-B
  - 2-a
  - geschlecht-m
- **Unterschied zu anderen infix-Operatoren (Evtl. auch selbst definierten):**
  - Werden, wenn gebraucht, von Prolog als Key-Value Paare interpretiert, z.B. von **keysort/2**
  - Es gibt eine eigene **library(pairs)** mit nützlichen Prädikaten, die mit Paaren als Key-Value pairs operieren

Analogien zu Datenstrukturen in anderen Sprachen

- **Listen**
  - Eingebaute Struktur
- **Assoziative Listen**
  - **library(assoc)** in fast allen Prologs vorhanden
  - Sind de facto spezielle Listen, implementiert als AVL-Baum
- **Dictionaries (Dicts, Hash maps)**
  - Eingebaut in SWI-Prolog (ab Version 7)
  - Nicht Standard in anderen Prologs
- **Arrays**
  - Gibt es nicht! Braucht man nicht, weil:
  - Prädikate können beliebig viele Argumente haben!
  - Die sind in konstanter Zeit zugreifbar
- **Structs / Klassen**
  - Gibt es nicht! Braucht man nicht, weil:
  - Lassen sich mittels Listen und Prädikaten darstellen (natürlich ohne Vererbung)

Programmklausel sind selbst Prolog Terme!

Nachfolgend werden wir die Struktur von Prolog Programmen betrachten. Bedenke aber bitte:

- **Prolog ist Homo-Ikonisch:**
  - Programm ist selbst eine Prolog-Datenstruktur
  - Besteht aus Prolog-Terme
  - Kennung des Endes eines Programm-Terms: '.' (ist nicht selbst Bestandteil des Terms)
  - Wichtigste Programm-Operatoren: ':-', ',' und ';' Diese sind selbst Bestandteil des Terms
- **Prolog-Terme (und damit Programm!) können während der Laufzeit hinzugefügt und weggenommen werden**
  - ACHTUNG: dies ist nicht Pure Prolog → Seiteneffekte

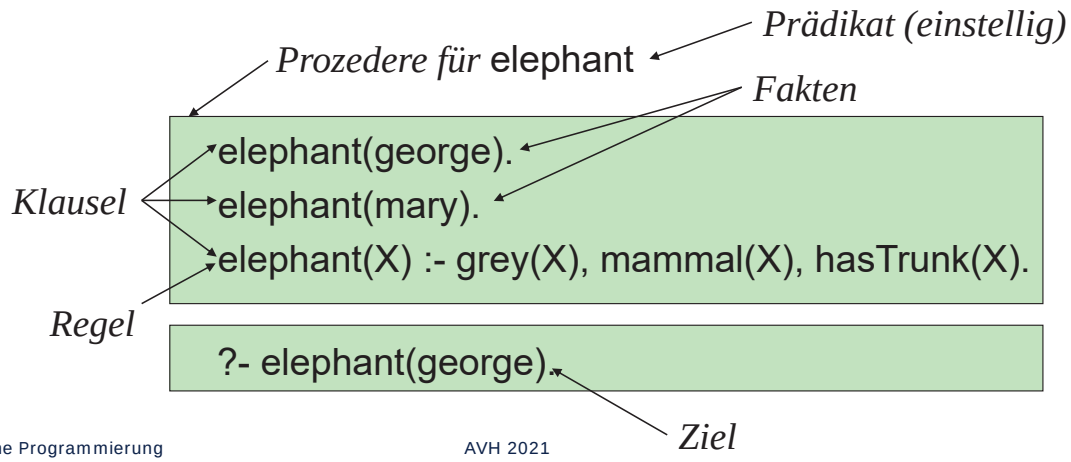
Falls Sie verwirrt sind

- **Da Prolog-Strukture insgesamt sehr unterschiedlich aussehen können, ist einem vielleicht (bei Programmen von Dritten) nicht immer klar, wie die Funktor-Argument Struktur tatsächlich ist**
  - Die Darstellung von Strukturen ist oft "syntactis sugar"
  - Die interne Darstellung für Programmausführungs kann durchaus anders sein

- **write_canonical/1 ist ein Prädikat, das aufzeigt, wie ein Term Prolog-intern dargestellt wird (nützlich im REPL)**
  - ?- write_canonical(5 >= 3).
    >=(5,3)
  - ?- write_canonical([a,b,c|[d|[e|[]]]]).
    [a,b,c,d,e]
  - ?- write_canonical(member(X,[Y|Z]) :- member(X,Z)).
    :-(member(_,[A|B]),member(A,B))

Die Struktur von Prolog Programmen: Fakten, Regeln, Ziele/Queries

- **Programme bestehen aus Prozederen**
- **Prozederen bestehen aus (definiten) Klauseln**
- **Klauseln bestehen aus Fakten oder Regeln**
- **Programme werden ausgeführt durch die Formulierung von Zielen**



Syntax von Fakten und Regeln

- **Ein Fakt ist eine Struktur gefolgt von einem Punkt '.'**
  - Beispiele:
    foo. bar(X). X >> Y. [a,b,c].
- **Ein Regel ist eine Struktur gefolgt von ':-' gefolgt von ein oder mehreren komma- oder semikolon-getrennten Strukturen gefolgt von einem Punkt.**
  - Beispiele:
    foo(X) :- bar(Z), umu(X,Z).
    lulu(A) :- a(A); b(A); b(C).
- **Es dürfen zur rechten Hand von :- sowohl Kommata als auch Semikola auftreten. Zur Disambiguierung kann man Klammern verwenden (sinvoll!!)**
- **Die Struktur zur linken Hand von :- nennt man Kopf (Head) der Klausel. Alles zur rechten Hand von :- nennt man Rumpf (Body) der Klausel.**

Syntax von Zielen (Goals)

- **Ziele sind Regel die keine Struktur vor dem ':-' haben**
  - Beispiele:
    :- foo(X), umu(b).
    :- member(X,[a,b,c]).
- **Zielen können in dieser Weise in Prolog Dateien (Programmen) geschrieben werden (Aufrufe im Code)**
- **In der REPL können nur Ziele formuliert werden**
  Diese werden dann kenntlich gemacht durch '?-'
  - Beispiele:
    ?- foo(a).
    ?- member(X,L), X >= 3.

Wie funktioniert Prolog? (Semantik)

- **Programme in Prolog (Algorithmen) bestehen in Wesentlichem aus Logik + Steuerung (Kowalski 1979)**
- **Logik:**
  - Prolog-Fakte und –Regel sowie –Ziele sind alle grundsätzlich (prädikat-)logische Formel im Klauselformat
  - Programmausführung ist grundsätzlich ein Beweisverfahren (Resolution) Dabei spielt Unifikation eine zentrale Rolle
- **Steuerung:**
  - Das Beweisverfahren ist nicht-deterministisch
  - Prolog baut (implizit) einen Beweisbaum auf
  - Dabei spielt Backtracking eine zentrale Rolle
- **Die genannten Konzepte behandeln wir im folgenden der Reihe nach**

Logik: Klausel-Format

- **Jede FOPL-Aussage kann in CNF umgeformt werden**
  - CNF: **C**onjunctive **N**ormal **F**orm
  - Alternative Benennung: **C**lausal **N**ormal **F**orm
- **Eine Aussage in CNF ist eine Menge von Klauseln, die mittels Konjunktion verbunden sind**
- **Eine Klausel ist eine (implizit) universal (all-) quantifizierte Menge von Literalen, die als Disjunktion aufzufassen ist**
- **Ein Literal ist ein Prädikat (mit entsprechenden Argumenten), oder dessen Negation**

Schreibweise von Klauseln

- **Man lässt die Konjunktionen weg und schreibt nur die Klauseln (Disjunktionen von Literalen) selbst**
- **Aufgrund der Eigenschaften der Disjunktionsoperator kann man die Disjunktionen ordnen:**
  - Man schreibt zuerst die positive Literalen, und anschließend die negative Literalen
  - Beispiel:

$$\alpha \vee \beta \vee \neg\gamma \vee \neg\delta \vee \neg\varepsilon$$

- **Nach dieser Umordnung kann man den Klausel logisch äquivalent umformen zu eine Implikation (Entfernung der Negation)**
  - Beispiel von oben:

$$\alpha \vee \beta \leftarrow \gamma \wedge \delta \wedge \varepsilon$$

- **Da links vom Pfeil alles disjunktiv ist und rechts alles konjunktiv, kann man diese weglassen, bzw. nur Kommata benutzen**
  - Beispiel von oben:

$$\alpha, \beta \leftarrow \gamma, \delta, \varepsilon$$

Hornklausel (Definite Clauses)

- **Falls links vom ← kein oder nur ein Literal steht, so nennt man solch einen Klausel Horn-Klausel**
  - Das heißt: der ursprüngliche Klausel hat kein oder nur ein positives Literal gehabt
  - Genannt nach dem Amerikanischen Logiker Alfred Horn, der sich mit dieser Art von Klauseln befasst hat
  - Alternative Benennung: Definite Clause
  - Rechts vom ← dürfen beliebige viele (auch keine) Literalen stehen
- **Somit gibt es vier Arten von Definite Clauses:**
  - $\alpha \leftarrow \beta, \gamma, \delta$
  - $\leftarrow \beta, \gamma, \delta$ Negierte Aussage
  - $\alpha \leftarrow$ Wahre Aussage
  - $\leftarrow$ Leere Klausel (Falsum, Resolvente)

Warum Horn-Klausel?

- **Weil sie zwei wichtige Eigenschaften aufweisen:**
  - Sie haben eine sehr einfache Struktur
  - Obwohl eine reine Untermenge von Prädikatenlogik, sind sie allgemein genug um alle mögliche Berechnungen anzustellen (Horn-Logik ist Turing-Komplett)
- **Dadurch erlauben sie:**
  - Schnelle Inferenzmechanismen (Resolution-Verfahren)
  - Effiziente Berechnungen bei absolut ausreichender Ausdrucksstärke (Expressiveness)
- **Das bedeutet:**
  - In Prolog können wir sehr kurze, kompakte Programme schreiben, die allgemein und effizient sind
  - Überdies können wir sogar logisch argumentieren *über* Prolog-Programme (wichtig für Erklären und Debuggen)

Horn Klausel und Prolog

- **In Prolog wird statt ← der Turnstile :-**
  (in der REPL als ?-) geschrieben
- **Somit erhalten wir vier Arten von Prolog Klauseln:**

$\alpha \leftarrow \beta,\gamma,\delta$	Regel (Rule)	a :- b, c, d.	
$\leftarrow \beta,\gamma,\delta$	Ziel (Goal)	:- b, c, d.	(?- b,c,d.)
$\alpha \leftarrow$	Fakt (Fact)	a.	(a :- true.)
$\leftarrow$	Empty Clause (leere Resolvente)		

- **Die Empty Clause kann nicht in der Programmierung verwendet werden, sondern ist das Resultat eines erfolgreiches Ziel**
- **Prolog (bzw. alle logische Programmiersprachen verwenden dafür Resolution)**

FOPL, Definite Clauses und Prolog

- **Jede FOPL Aussage kann zu Klausel umgeformt werden**
- **Es kann jedoch nicht jede beliebige FOPL Aussage zu eine Definite Clause umgeformt werden**
- **Somit stellen die Definite Clauses eine Untermenge von FOPL dar!**
- **Damit ist in Prolog nur eine Untermenge von FOPL ausdrückbar**
- **Diese Einschränkung wurde gemacht aus Gründen der Recheneffizienz**
- **Das Rechenverfahren ist SLD-Resolution**

Logik: Resolution

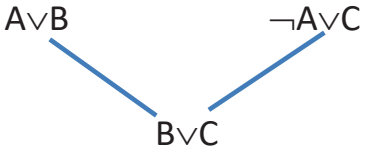
- **Resolution bedeutet „algorithmischer Test für die Unerfüllbarkeit einer Formel, der auf rein syntaktischen Umformungsregeln beruht".**
- **Die Resolutionsableitung besteht aus mehreren einzelnen Schritten, den sogenannten Resolutionsschritten.**
- **Gegeben sind zwei geeignete Klausel (in einer Mengen von evtl. weiteren Klauseln). Aus diesen wird eine dritte Formel, ebenso wieder eine Klausel, abgeleitet. Diese wird als Resolvente bezeichnet und zur Klauselmenge hinzugefügt.**

Logik: Resolution, Beispiel

- **Klausel1: $A \vee B$**
  Klausel2: $\neg A \vee C$
  Resolvente: $B \vee C$

- **Es werden aus beiden Formeln ein kontradiktorische Aussage (hier: das Literal A im Klausel1 und das Literal $\neg A$ in Klausel2) entfernt und die beiden Klauseln mit $\vee$ verbunden**

- **Graphisch:**



Resolution als Widerspruchsverfahren

- **Ziel der Resolution ist es, nach einer endlichen Anzahl von Resolutionsschritten die leere Klausel (Empty Clause / Leere Resolvente) abzuleiten**
- **Damit ist die Unerfüllbarkeit der ursprünglichen Formelmenge gezeigt.**
  (Resolution ist somit ein Widerspruchsverfahren, wie auch das Verfahren der semantischen Tableaux, das Sie in Informatik I kennengelernt haben)
- **Wenn man beweisen will, dass eine Formel aus eine Menge gegebener Klausel folgt, dann fügt man die *Negation* dieser Formel als Klausel zu der Menge hinzu.**
- **Falls man die leere Resolvente ableiten kann, dann ist die Menge zusammen mit negierter Formel nicht erfüllbar (= inkonsistent).**
- **Das ist dann ein Widerspruchsbeweis: die unnegierte Formel folgt somit aus der ursprünglichen Menge.**

SLD Resolution

- **In Prolog stellen die Fakte und Regel die gegebene Klauselmenge dar. Ein Ziel ist de facto eine negierte Klausel. Das Beweisverfahren ist eine spezielle Variante der Resolution: SLD-Resolution.**
- **SLD Resolution (*Selective Linear Definite* clause resolution) ist eine spezielle Form der Resolution. Sie ist korrekt und Widerspruchsvollständig für Horn-Logik.**
- **Das Verfahren ist Linear: Die Resolvente eines Resolutionsschrittes wird sofort wieder für den nächsten Resolutionsschritt (als einer der beiden zu resolvierenden Klauseln) genommen.**
- **Das Verfahren ist Selektiv: Es wird jeweils aufgrund einer Strategie ein Literal zur Resolution ausgewählt (Selection Strategie). Das Verfahren operiert nur mit Definite Clauses.**
- **SLD-Resolution liefert damit einen linearen Beweisbaum.**
- **Prolog hat als Strategie, jeweils das erste Literal des letzten Resolventen für Resolution zu wählen.**

SLDNF-Resolution

- **SLDNF-Resolution ist SLD-Resolution mit zusätzlich das Prinzip des Negation als Fehlschlag (Negation as Failure).**
- **Sie ist eine Kombination von**
  - SLD-Resolution zur Ableitung positiver Literale und
  - Negation als Fehlschlag zur Ableitung negativer Literale
- **Bei der SLDNF-Resolution gilt:**
  - Für jedes positive Literal, das in einer SLDNF-Ableitung auftaucht, wird ein Beweis mit SLD-Resolution geführt
  - für jedes negative Literal ein separater Beweis durch Negation als Fehlschlag.
  - Ein negatives Ziel darf nur selektiert werden, wenn es variablenfrei ist (bzw. die Variablen bereits instanziert sind).
  - Wenn solch ein Literal ausgewählt wurde, dann wird ein Unter-Beweis (Subproof) für das entsprechende positive Literal versucht: not(P) ist nur dann erfolgreich wenn ein SLDNF-Beweis für P nicht gelingt

Logik: Unifikation

- **Wie können wir feststellen ob zwei Literalen „gleich" (abgesehen von der Negation) sind, damit wir sie resolvieren können?**
- **Für Literalen ohne freie Variablen ist dies einfach: sie sollten identisch sein.**
- **Wenn allerdings freie Variablen in den Literalen auftauchen (häufiger Fall in prädikatenlogischen Klauseln), wie kann man dann vorgehen?**
- **Die Antwort liefert darauf die Unifikation: die meist allgemeine Substitution von Variablen, die beide Literalen gleich macht.**

Unifikation: technisch

- **Eine Substitution σ so dass $t\sigma = u\sigma$ ist eine *Unifier* von *t* und *u***
- ***Unifikation*: bestimme die Menge der Unifier für *t* und *u*.**
- **Es seien σ_1 and σ_2 Substitutionen. Wir nennen σ_2 allgemeiner als σ_1 wenn: für irgendeine substitution τ *gilt, dass* $\sigma_1 = \sigma_2\tau$**
- **Beispiel: Es sei**
  $\sigma_1 = \{X/f(g(a, h(Z))), Y/g(h(X), b), Z/h(X)\}$
  und
  $\sigma_2 = \{X/f(g(X, Y)), Y/g(Z, b)\}$

  ***dann ist* σ_2 allgemeiner als σ_1, weil $\sigma_1 = \sigma_2\tau$**
  mit $\tau = \{X/a, Y/h(Z), Z/h(X)\}$.

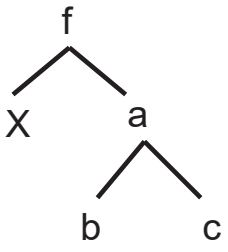
Unifikation: allgemeinster Unifier

Es seien *t1* and *t2* zwei Terme :

- **Eine Substitution σ ist eine *Unifier* für *t1* und *t2* gdw. $t1\sigma = t2\sigma$.**
- ***t1* and t2 sind *unifizierbar* wenn sie einen Unifier haben.**
- **Eine Substitution σ ist ein allgemeinster Unifier (*most general unifier*) wenn es ein Unifier ist, der allgemeiner ist als jede andere Unifier.**
- **Beispiele:**
  - Die Terme *f(Y, h(a))* und *f(h(X), h(Z))* sind unifizierbar mit der substitution *{Y/h(X), Z/a}*.
  - Auch die Substitution *{X/k(W), Y/h(k(W)), Z/a}* ist ein Unifier für beide Terme, aber der vorige ist mehr allgemein.
  - *{Y/h(X), Z/a}* ist ein allgemeinster Unifier für beide Terme.
  - Die Terme *f(X, X)* und *f(a, b)* sind nicht unifizierbar.
- **Bei Unifikation zweier Terme wird ein allgemeinster Unifier gesucht**

Term-Unifikation ist Baum-Unifikation

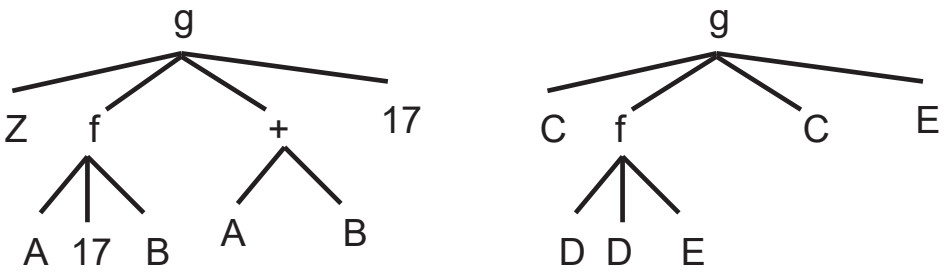
- **Terme können als Bäume dargestellt werden**
- **Beispiel:**
  f(X, a(b,c))
  kann dargestellt werden als



- **Ein Unifikationsalgorithmus geht mittels rekursiver Abstieg (recursive descent) der beiden Term-Bäume vor. Dabei gilt:**
  - Konstante unifizieren wenn sie identisch sind
  - Variablen unifizieren mit beliebigen Termen, auch mit anderen Variablen
  - Compounds unifizieren wenn ihre Funktoren und Komponenten unifizieren.

Ein Beispiel detailliert durchgespielt

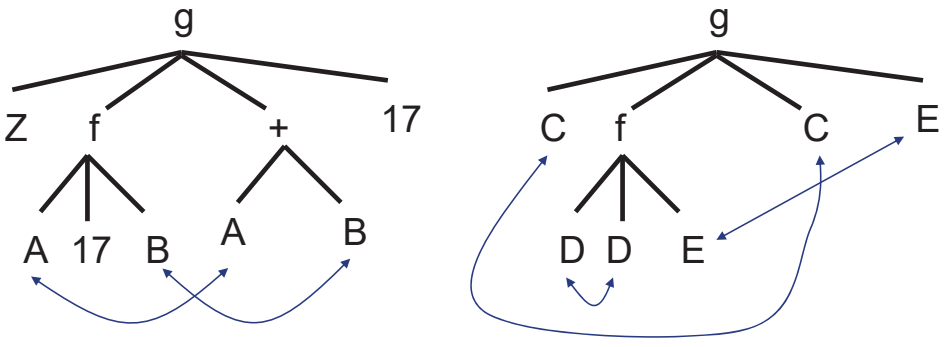
Sind die Terme
g(Z, f(A, 17, B), A+B, 17) und
g(C, f(D, D, E), C, E) unifizierbar?



Beispiel wurde entwickelt von W. Clocksin

Ein Beispiel detailliert durchgespielt

Zuerst festhalten wo welche Variable im Term nochmal auftritt.



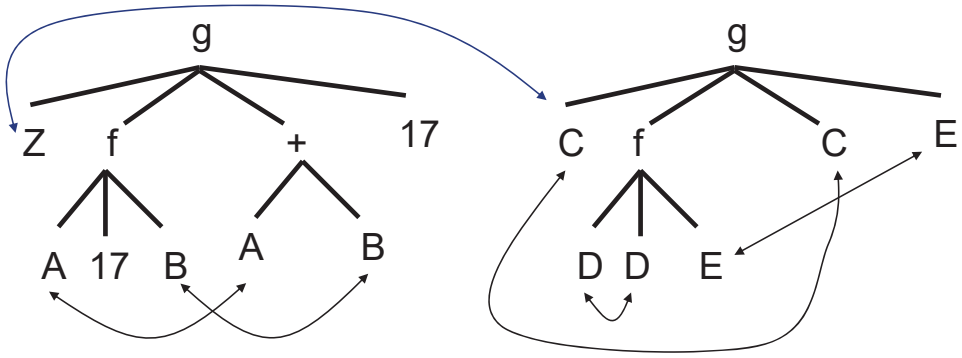
Beispiel wurde entwickelt von W. Clocksin

Benutzung dieser Folien ist nur im Rahmen Ihres DHBW-Studiums erlaubt

Ein Beispiel detailliert durchgespielt

Z/C, C/Z

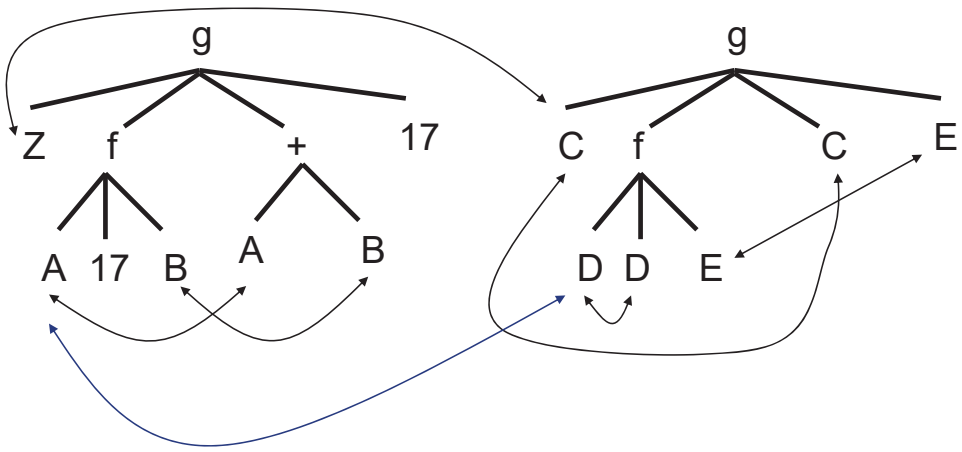
Jetzt rekursiv absteigend vorgehen. Wir gehen top-down, left-to-right, aber die genaue Reihenfolge ist egal, solange man systematisch und vollständig vorgeht



Beispiel wurde entwickelt von W. Clocksin

Ein Beispiel detailliert durchgespielt

Z/C, C/Z, A/D, D/A

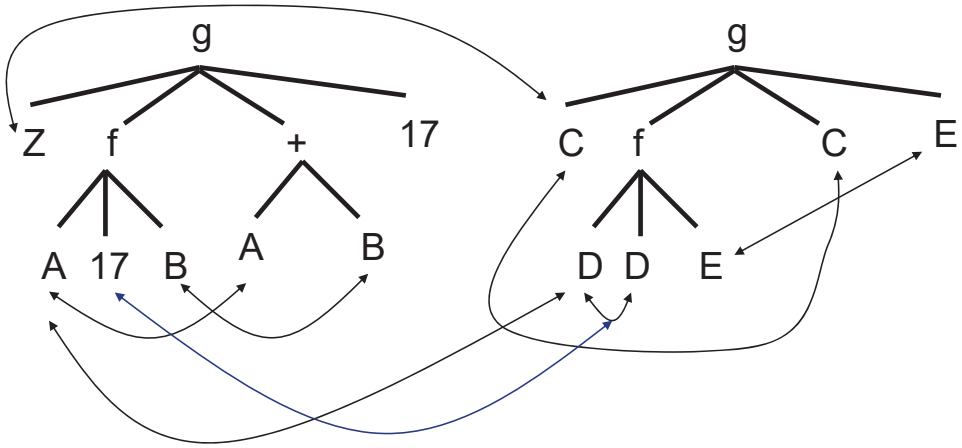


Beispiel wurde entwickelt von W. Clocksin

Benutzung dieser Folien ist nur im Rahmen Ihres DHBW-Studiums erlaubt

Ein Beispiel detailliert durchgespielt

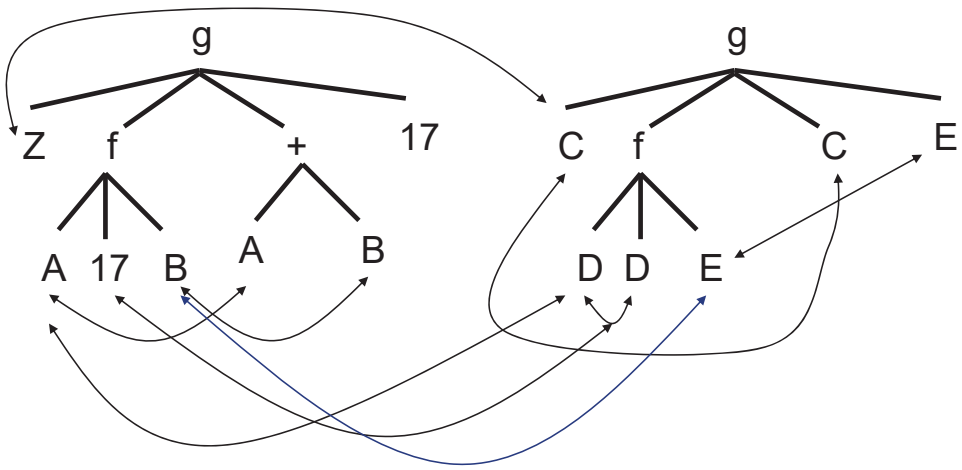
Z/C, C/Z, A/17, D/17



Beispiel wurde entwickelt von W. Clocksin

Ein Beispiel detailliert durchgespielt

Z/C, C/Z, A/17, D/17, B/E, E/B

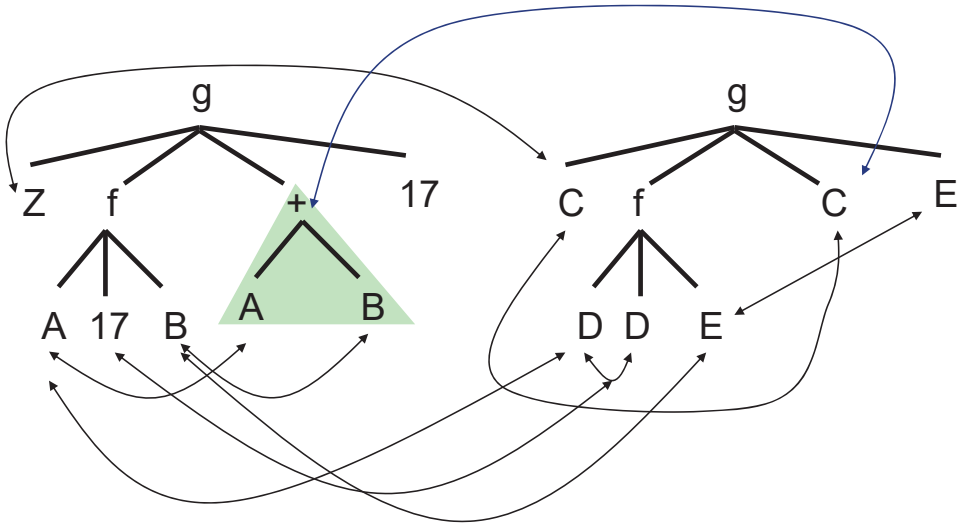


Beispiel wurde entwickelt von W. Clocksin

Benutzung dieser Folien ist nur im Rahmen Ihres DHBW-Studiums erlaubt

Ein Beispiel detailliert durchgespielt

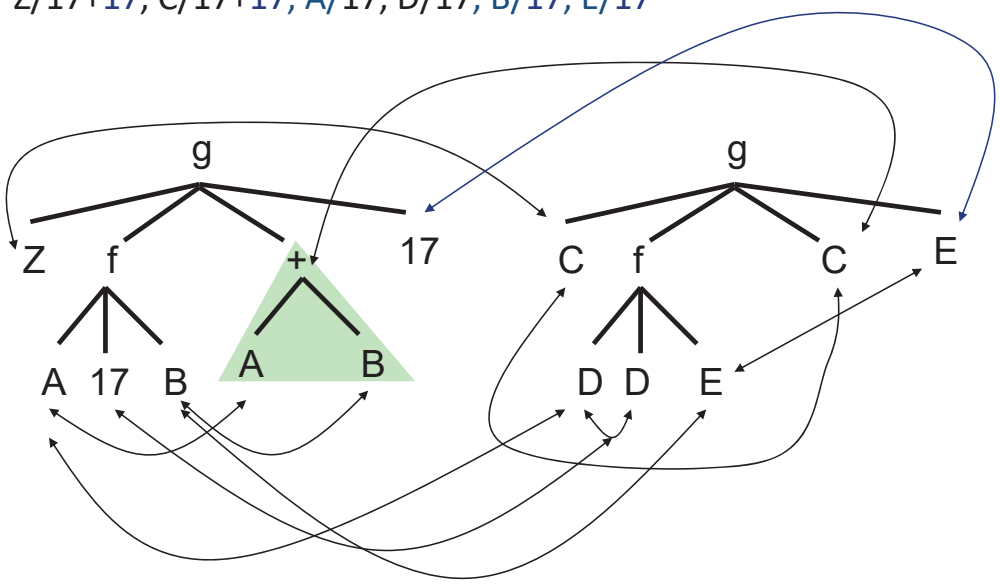
Z/17+B, C/17+B, A/17, D/17, B/E, E/B



Beispiel wurde entwickelt von W. Clocksin

Ein Beispiel detailliert durchgespielt

Z/17+17, C/17+17, A/17, D/17, B/17, E/17



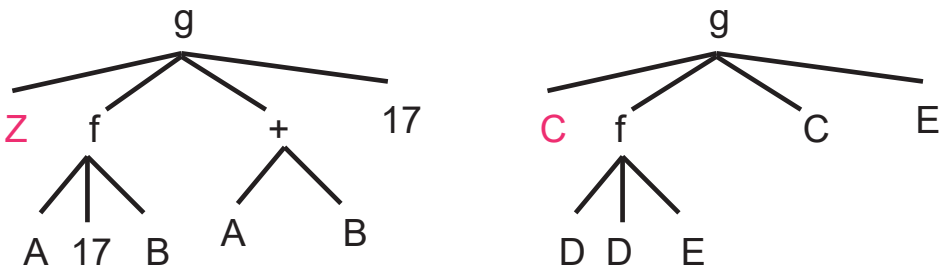
Beispiel wurde entwickelt von W. Clocksin

AVH 2021

Benutzung dieser Folien ist nur im Rahmen Ihres DHBW-Studiums erlaubt

Gleiche Beispiel – Alternative Methode

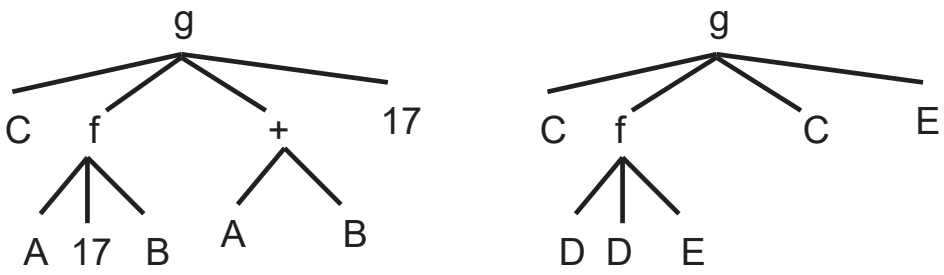
Z/C



Beispiel wurde entwickelt von W. Clocksin

Gleiche Beispiel – Alternative Methode

Z/C

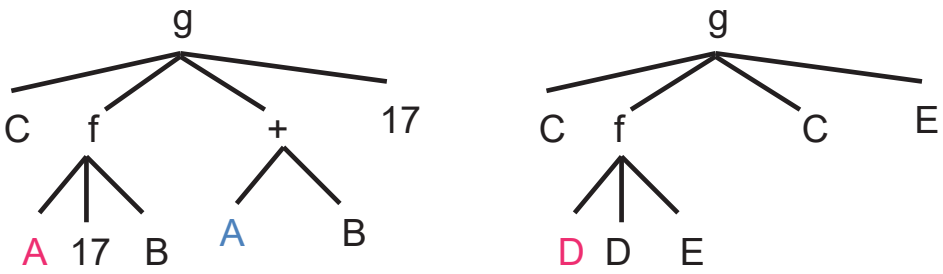


Beispiel wurde entwickelt von W. Clocksin

Benutzung dieser Folien ist nur im Rahmen Ihres DHBW-Studiums erlaubt

Gleiche Beispiel – Alternative Methode

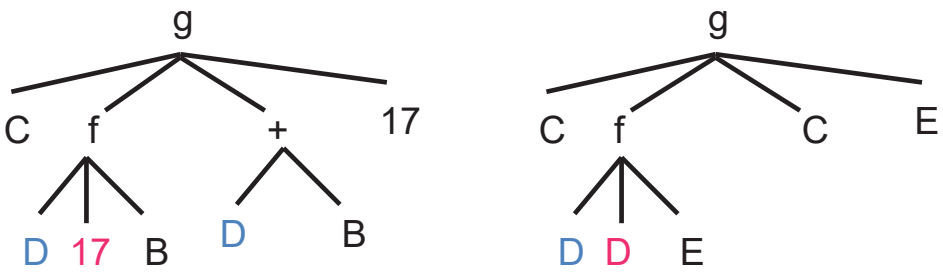
A/D, Z/C



Beispiel wurde entwickelt von W. Clocksin

Gleiche Beispiel – Alternative Methode

D/17, A/D, Z/C



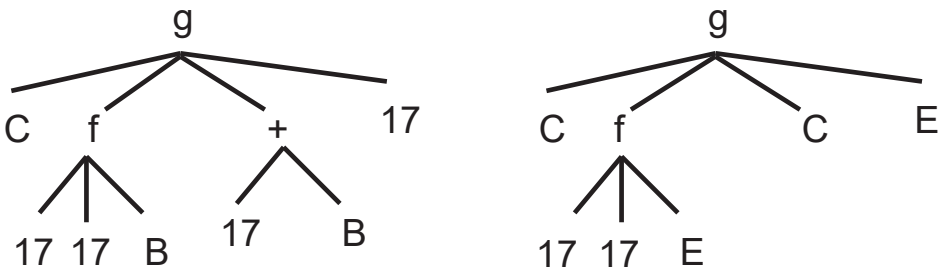
Beispiel wurde entwickelt von W. Clocksin

AVH 2021

Benutzung dieser Folien ist nur im Rahmen Ihres DHBW-Studiums erlaubt

Gleiche Beispiel – Alternative Methode

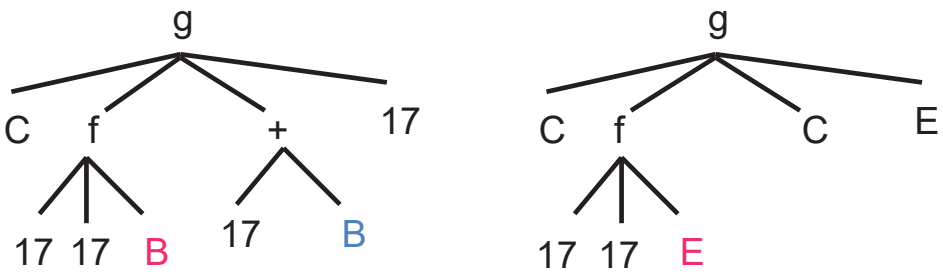
D/17, A/17, Z/C



Beispiel wurde entwickelt von W. Clocksin

Gleiche Beispiel – Alternative Methode

B/E, D/17, A/17, Z/C



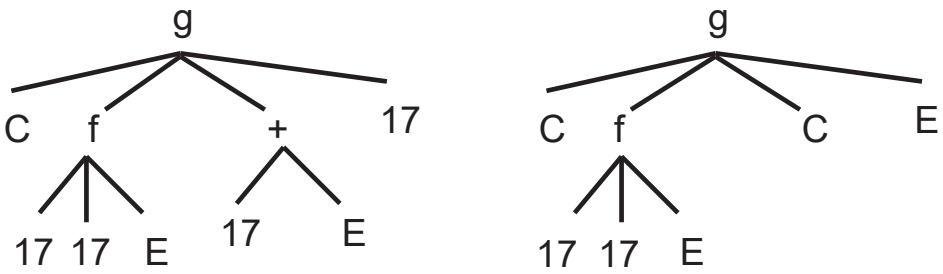
Beispiel wurde entwickelt von W. Clocksin

AVH 2021

Benutzung dieser Folien ist nur im Rahmen Ihres DHBW-Studiums erlaubt

Gleiche Beispiel – Alternative Methode

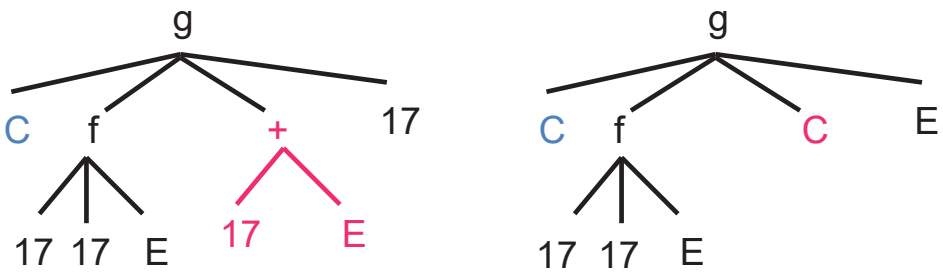
B/E, D/17, A/17, Z/C



Beispiel wurde entwickelt von W. Clocksin

Gleiche Beispiel – Alternative Methode

C/17+E, B/E, D/17, A/17, Z/C

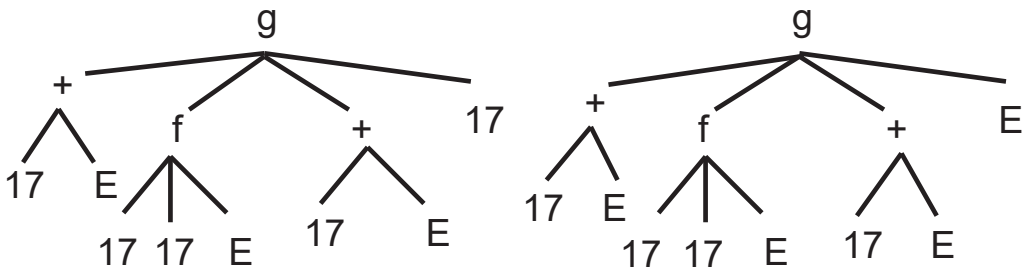


Beispiel wurde entwickelt von W. Clocksin

Benutzung dieser Folien ist nur im Rahmen Ihres DHBW-Studiums erlaubt

Gleiche Beispiel – Alternative Methode

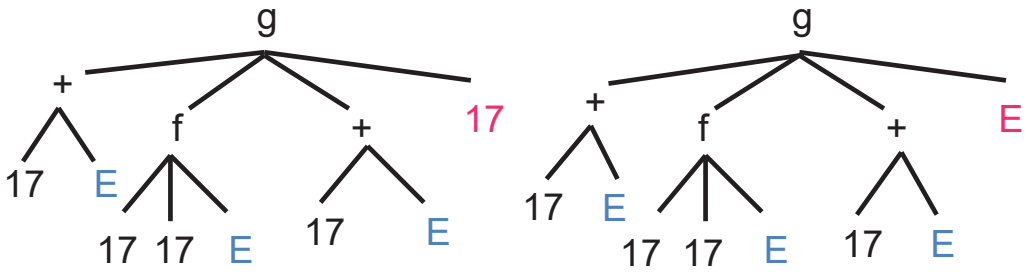
C/17+E, B/E, D/17, A/17, Z/17+E



Beispiel wurde entwickelt von W. Clocksin

Gleiche Beispiel – Alternative Methode

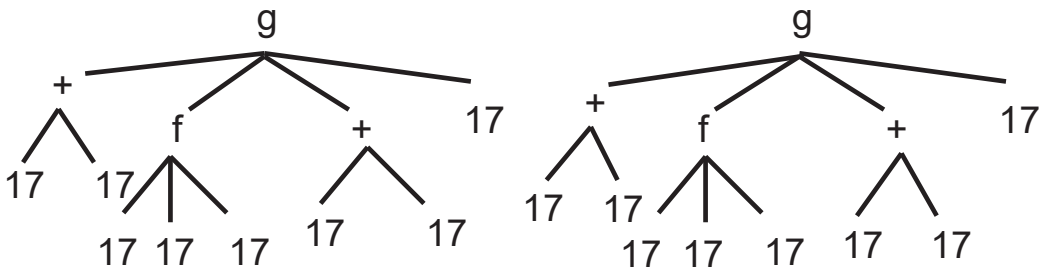
E/17, C/17+E, B/E, D/17, A/17, Z/C



Beispiel wurde entwickelt von W. Clocksin

Gleiche Beispiel – Alternative Methode

E/17, C/17+17, B/17, D/17, A/17, Z/C



Beispiel wurde entwickelt von W. Clocksin

Misslingen eines Unifikationsversuches

- **Unifikation zweier Terme schlägt fehl, wenn:**
  - Beide Terme nicht-identische Atome sind
    Beispiel: foo ≠ fo
  - Beide Terme Compounds mit nicht-identischen Funktoren oder ungleicher Argumentzahl sind
    Beispiel: foo(a,b) ≠ fo(a,b)
    foo(a,b) ≠ foo(a,b,c)
  - Der eine Term eine Variable ist und der andere ein Term ist, der diese Variable enthält
    Beispiel: X ≠ foo(bar(baz(X)))
- **Besonders das Checken der letzten Bedingung (Occurs Check) ist relativ kostspielig, weil für den nicht-variablen Term dessen ganzen Term-Baum traversiert werden muss um zu prüfen ob der Variable nicht darin vorkommt. Dies muss auch nachträglich geprüft werden, wenn später im Verfahren eine andere Variable im Term mit der entsprechenden Variable unifiziert werden soll (und dann nicht darf!).**
  Beispiel: foo(X,X) ≠ foo(bar(Y),Y).

Occurs Check in Prolog

- **Prolog implementiert aus Effizienzgründen die Unifikation standardmäßig OHNE Occurs-Check!**
- **Es ist nicht definiert, wie Prolog sich verhalten soll, wenn z.B. ein Term X mit einem Term f(X) zum unifizieren vorliegt.**
- **Manche Prologs produzieren eine Endlosschleife, andere (auch SWI) lassen eine inkorrekte Unifikation gelingen! Das führt dann zu inkorrekten Resultaten der Prolog-Abfrage.**
- **SWI-Prolog (wie andere Prologs auch) bietet jedoch auch noch ein Prädikat unify_with_occurs_check/2 an. Man kann diese jedoch auch selbst in Prolog implementieren, wie die folgende Folien zeigen**

Unifikationsalgorithmus als Prolog Code !!! (1/3)

```
/* unify(Term1, Term2) :-
Term1 and Term2 are unified with the occurs check.
   See Stirling and Shapiro, The Art of Prolog, Page 152. */

unify (X, Y) :-
       var(X) , var(Y), X=Y.
unify(X,Y) :-
       var(X), nonvar(Y), not_occurs_in(X,Y), X=Y.
unify(X,Y) :-
       var(Y), nonvar(X), not_occurs_in(Y,X), Y=X.
unify(X,Y) :-
       nonvar(X), nonvar(Y), atomic(X) , atomic(Y), X=Y.
unify(X,Y) :-
       nonvar(X), nonvar(Y),
       compound (X) , compound(Y), term_unify(X,Y).
```

Unifikationsalgorithmus als Prolog Code !!! (2/3)

```
not_occurs_in(X,Y) :-
      var(Y), X \== Y.
not_occurs_in(X,Y) :-
      nonvar(Y), atomic(Y).
not_occurs_in(X,Y) :-
      nonvar(Y), compound(Y), functor(Y,F,N),        not_occurs_in(N,X,Y).


not_occurs_in(N,X,Y) :-
      N>0, arg(N,Y,Arg), not_occurs_in(X,Arg),
      N1 is N-1, not_occurs_in(N1,X,Y).
not_occurs_in(0,X,Y).
```


Unifikationsalgorithmus als Prolog Code !!! (3/3)

```
term_unify(X,Y) :-
      functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).


unify_args(N,X,Y) :-
      N>0, unify_arg(N,X,Y), N1 is N-1,
      unify_args(N1,X,Y).
unify_args(0,X,Y).


unify_arg(N,X,Y) :-
      arg(N,X,ArgX), arg(N,Y,ArgY), unify(ArgX,ArgY).
```

Nochmals: Wie funktioniert Prolog? (Semantik)

- **Programme in Prolog (Algorithmen) bestehen in Wesentlichem aus Logik + Steuerung (Kowalski 1979)**
- **Logik:**
  - Prolog-Fakte und –Regel sowie –Ziele sind alle grundsätzlich (prädikat-)logische Formel im Klauselformat
  - Programmausführung ist grundsätzlich ein Beweisverfahren (Resolution)
    Dabei spielt Unifikation eine zentrale Rolle
- **Steuerung:**
  - Das Beweisverfahren ist nicht-deterministisch
  - Prolog baut (implizit) einen Beweisbaum auf
  - Dabei spielt Backtracking eine zentrale Rolle

Steuerung: Nicht-Determinismus

- **Das Beweisverfahren kann nicht-erfolgreich sein (semi-deterministisch)**
- **Wenn es erfolgreich ist, dann wird lediglich eine von möglicherweise mehreren Lösungen ausgegeben**
- **Wenn eine Lösung nicht „akzeptabel" ist, dann kann eine weitere Lösung gesucht werden.**
- **Ob es (mehr) Lösungen gibt kann nur durch ausschöpfende Suche im ganzen Lösungsraum (sämtliche mögliche Resolutionsschritten) herausgefunden werden**
- **Dafür sind mehrere Strategien möglich**
- **Prolog implementiert eine Top-Down, Links-vor-Rechts Strategie (Tiefensuche)**

Steuerung: Beweisbaum

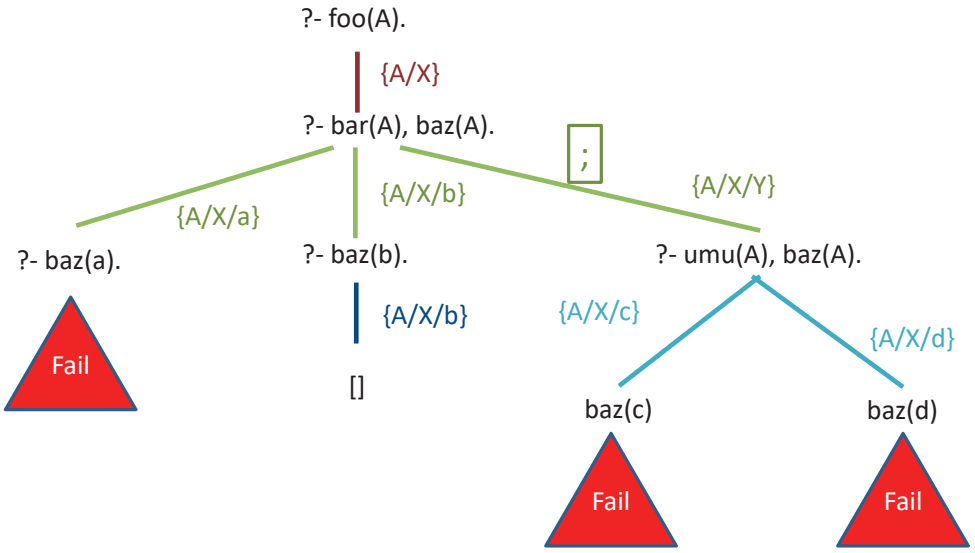
- Prolog baut impliziet einen Beweisbaum auf
- (Nur impliziet: man kann ihn „nachzeichnen", es gibt keine Speicherung des Baumstrukturs)
- Der Baum ist ein SLD(NF)-Baum
- Aufgrund der gewählten Tiefensuche-Stragegie ist der Beweisbaum Prolog-intern realisiert als Goal-Stack:
  Goals werden auf einem Stapel gelegt und weggenommen.
- Auf den Goal-Stack kommt zunächst die Anfrage.
- Ein Resolutionsschritt führt zur Wegnahme eines Goals und evtl. Aufbringung neuer Goals
- Wenn der Goal-Stack im Laufe einer Anfrage leer wird, so ist die Anfrage erfolgreich.

Steuerung: Backtracking

- Überall wo es mehrere Auswahlmöglichkeiten für einen Resolutionsschritt gibt, setzt Prolog ein **Auswahlpunkt (Choice Point)**
- Falls ein Goal sich nicht mit einem Fakt oder Regel resolvieren lässt, geht Prolog zurück im SLD-Baum (bzw. weiter herunter im Stack) zum letzten Auswahlpunkt und setzt mit der nächsten Alternative fort
- Falls an einem Auswahlpunkt keine Alternativen mehr vorhanden sind, geht Prolog weiter zurück zum vorigen Auswahlpunkt.
- Falls Prolog zurückgehen muss und kein Auswahlpunkt mehr hat, ist die ursprüngliche Abfrage nicht erfolgreich
- Dieses Vorgehen nennt sich **Backtracking**

Beispiel SLD-Baum

```
foo(X) :- bar(X), baz(X).
bar(a).
bar(b).
bar(Y) :- umu(Y).
umu(c).
umu(d).
baz(b).
```



Nicht-Logische Konsequenzen der Steuerung

- **Die Reihenfolge der Klauseln einer Prozedere/eines Prädikats ist relevant**
  - Nicht nur für die Effizienz
  - Sondern auch für die Effektivität: manche Beweisen, die es de facto/logischerweise gibt, werden evtl. nicht von Prolog gefunden (wenn nicht-monotone Goals aufgerufen werden)
- **Die Anordnung von Termen im Body einer Klausel ist relevant**
  - Nicht nur für die Effizienz
  - Sondern auch für die Effektivität: manche Beweisen, die es de facto/logischerweise gibt, werden evtl. nicht von Prolog gefunden (wenn nicht-monotone Goals aufgerufen werden)
- **Effektivität kann man jedoch wieder herstellen durch die Beschränkung auf Pure Prolog (dazu später mehr), aber nicht die Effizienz**

Beispiele für logisches Fehlverhalten

```
foo(X) :- bar(X).
foo(a).

bar(X) :- foo(X).
bar(b).

?- foo(a).
endlosschleife
```

```
succ1(X,Y) :- plus(X,1,Y), number(X),number(Y).
succ2(X,Y) :- number(X),plus(X,1,Y),number(Y).
succ3(X,Y) :- number(Y),plus(X,1,Y),number(X).

?- succ2(X,3).
false.
?- succ3(2,X).
false.
```

Problem: number/1 ist kein Pure Predicate, weil sein Verhalten nicht logisch korrekt ist (aber schon ISO-konform)
→
Liefert 'false' wenn Argument nicht instanziert ist, statt ein Instanzierungsfehler!

Die Lehre daraus:

- **Rein logisches deklaratives Denken ist sehr wichtig**
- **Aber prozedurales (SLDNF, Depth-First) Verständnis gehört auch dazu!**

- **Prozedurales Verständnis wird noch wichtiger, wenn noch außerlogische Konzepte dazu kommen:**
  - Cut !
  - Failure fail
  - Oder ;
  - Funktionale Arithmetik is
  - Seiteneffekte I/O, assert, retract

- **Hierzu im nächsten Kapitel mehr!**

Philosophie/Strategie: Pure Prolog

- **Obwohl in der Praxis nur für kleine Programmteile, selten aber für komplette Programme erreichbar, sollten Sie Pure Prolog anstreben**
- **Pure Prolog: Code-Ausführung korrespondiert im Ergebnis mit Prädikatenlogik:**
  - Goal succeeds im Prolog-Programm wenn prädikatenlogische Term logisch aus der Menge der Programmklauseln ableitbar ist
  - Prädikatenlogische Term ist aus der Menge der Programmklauseln logisch ableitbar wenn Goal succeeds
- **Pure Prolog ist monoton und Seiteneffektfrei:**
  - Goal succeeds auch noch wenn Programm mit Klauseln erweitert wird
  - Programm arbeitet nicht mit irgendwelchen globalen Zustände bzw. Variablen
  - Programm arbeitet nicht explizit mit externen Ein- und Ausgaben (Stdin, StdOut usw.)

Warum Pure Prolog?

- **Prädikate können in beliebige Richtung benutzt werden**
- **Programme sind einfacher änderbar im Sinne der Neuorganisation der Reihenfolge der Klausel**
  - Sowohl im Programm als auch in Abfragen
  - Änderung der Effizienz ohne Gefahr für die Effektivität
- **Man kann das Programm rein deklarativ betrachten**
  - Hilfreich für Programmverständnis und Debugging
- **Programm ist auch in Ordnung für andere Resolutionsverfahren**
  - Keine Änderung notwendig, wenn z.B. Prolog neu implementiert würde mit bottom-up oder breiten-orientierte Resolutionsverfahren
  - "Wir kümmern uns um die Logik, Prolog kümmert sich um die Control"
- **Programm ist automatisch thread safe**
  - Wichtig für multi-threading
  - Grund für steigendes Interesse in funktionale und logische Sprachen

Prolog Konventionen

- **Weil Prolog Relationale Logik darstellt gibt es bei Verwendung von Prädikaten mehrere Arten und Weisen, wie man dies tun kann**
  - Beispiele:
  - ?- length([a,b,c],3).
    true.
  - ?- length([a,b,c],4).
    false.
  - ?- length(L,3).
    L = [_51610, _51616, _51622].
  - ?- length([a,b,c],N).
    N = 3.
  - ?- length(L,N).
    L = [],
    N = 0 ;
    L = [_54698],
    N = 1 ;
    L = [_54698, _55730],
    N = 2 usw.
- **Das klappt jedoch nicht immer.**
  Der Programmierer muss angeben was in einer Abfrage möglich ist!

Prolog Konvention: Prädikat-Dokumentation

Determinism	Predicate behaviour
`det`	Succeeds exactly once without a choice point
`semidet`	Fails or Succeeds exactly once without a choice-point
`failure`	Always fails
`nondet`	No constraints on the number of times the predicate succeeds and whether or not it leaves choice-points on the last success.
`multi`	As `nondet`, but succeeds at least one time.
`undefined`	Well founded semantics third value. See undefined/0.

```
%!      length(+List:list, -Length:int) is det.
%!      length(?List:list, -Length:int) is nondet.
%!      length(?List:list, +Length:int) is det.
%
%       True if List is a list of length Length.
%
%       @compat iso
```

++	Argument is ground at call-time, i.e., the argument does not contain a variable anywhere.
+	Argument is fully instantiated at call-time, to a term that satisfies the type. This is not necessarily *ground*, e.g., the term `[_]` is a *list*, although its only member is unbound.
-	Argument is an *output* argument. It may be unbound at call-time, or it may be bound to a term. In the latter case, the predicate behaves as if the argument was unbound, and then unified with that term after the goal succeeds. For example, the goal `findall(X, Goal, [T])` is good style and equivalent to `findall(X, Goal, Xs), Xs = [T]`[3] Determinism declarations assume that the argument is a free variable at call-time. For the case where the argument is bound or involved in constraints, `det` effectively becomes `semidet`, and `multi` effectively becomes `nondet`.
--	Argument is unbound at call-time. Typically used by predicates that create'something' and return a handle to the created object, such as open/3 which creates a *stream*.
?	Argument is bound to a *partial term* of the indicated type at call-time. Note that a variable is a partial term for any type.
:	Argument is a meta-argument. Implies +.
@	Argument will not be further instantiated than it is at call-time. Typically used for type tests.
!	Argument contains a mutable structure that may be modified using setarg/3 or nb_setarg/3.

Inline Dokumentation in SWI-Prolog:

PlDoc

Siehe:
https://www.swi-prolog.org/PlDoc.txt