# Alternative Programmierkonzepte (T2INF4271)

**Logische Programmierung**

**03**
**Prolog**
**Außerlogische Konzepte**
**Metalogische Konzepte**

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

AVH 2021



## Außerlogische Konzepte

- **In diesem Kapitel behandeln wir die folgenden außerlogische Konzepte:**

| | | |
|---|---|---|
| 1 | **Cut** | ! |
| 2 | **Failure** | fail |
| 3 | **If-Then-Else** | -> ; |
| 4 | **Endloswiederholung** | repeat |
| 5 | **Funktionale Arithmetik** | is, arithmetic_function |
| 6 | **Seiteneffekte** | I/O, assert, retract |

- **Anwendung dieser Konzepte kann bei unsachgemäßer Verwendung die Logik eines Programmes zerstören**
- **Richtig angewendet verbessern sie die Effizienz der Beweisführung und vermitteln korrekt mit der Außenwelt**
- **Anschließend behandeln wir meta-logische Konzepte**

## Der Cut (!)



- **Manchmal weiß der Programmier bereits, dass es keinen Sinn macht bestimmte Alternative (Auswahlpunkte / choice points) weiter zu verfolgen, da sie voraussehbar nicht zum Erfolg (leere Resolvente) führen**
- **Deswegen wurde in Prolog aus Effizienzgründen der Cut (!) eingeführt**
- **Wenn Prolog in einer Klausel einen Cut begegnet, dann:**
  - ist dieser automatisch erfolgreich (als Prolog Goal),
  - werden alle Auswahlpunkte zwischen dem Goal direkt vor dem Cut und den Kopf der Klausel gelöscht, mögliche Auswahlpunkte bzgl. des Prädikats im Kopf mit eingeschlossen
- **Ein erfolgreicher Cut in einer Klausel einer Prozedere entfernt damit die Auswahlmöglichkeit aller weiteren Klausel der gleichen Prozedere**





## Beispiel richtiger Benutzung eines Cut

```
foo(N,M) :-
        nonvar(N),
        N > 0,
        !,
        umu(M).
foo(N,M) :-
        nonvar(N),
        N =< 0,
        bar(M).
```

- **foo(4,X) kann mit beiden Klauseln unifizieren**
- **Wenn überhaupt, dann kann nur die erste Klausel zum Erfolg führen (abhängig davon ob umu(M) beweisbar ist).**
- **Dies kann nach N >0 festgestellt werden, daher ist das Platzieren eines Cuts hier logisch einwandfrei und kürzt nur den nicht-erfolgreichen Beweisteil ab.**

## Probleme mit Cut (1)

- Leider kann das Setzen eines Cut die Korrektheit / Vollständigkeit der Beweisführung beeinträchtigen
- Beispielprogramm:

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

- Es gibt dann inkorrekterweise keinen Beweis mehr für **?- foo(X).**
- Sie würden es bemerken? Häufig ist es bei falscher Anwendung des Cut jedoch so, dass zwar eine korrekte Lösung gefunden wird, aber effektiv weitere Lösungen ausgeschlossen werden.





## Probleme mit Cut (2)

- Gut gemeint ist leider nicht immer gut genug
- Beispiel:

```
%% minimum(+A:int,+B:int,?Min:int) is semidet.
minimum(A,B,A) :-
        nonvar(A), nonvar(B),
        A =< B,
        !.
minimum(A,B,B) :-
        nonvar(A),nonvar(B).
```

- Was ist hier das Problem?
- Wie kann man es beheben?

## Red Cuts, Green Cuts

- **GREEN CUT**
  **Cuts, die:**
  - die Effizienz des Programmes positiv beeinflussen
  - jedoch keinen Einfluss haben auf
    - die logische Korrektheit (die richtige Lösungen)
    - die Vollständigkeit (alle Lösungen)
  - des Programmes, nenn man „green Cuts“

- **RED CUT**
  **Cuts die die Logik der Anwendung nicht gewährleisten nennt man dagegen „red Cuts“.**

- **Red Cuts sind zu vermeiden, weil schlecht, aber wie?**
- **Die beste Lösung ist, zuerst mal komplett auf Cut zu verzichten!!**





## Failure (fail)


- **Das Prädikat fail schlägt als Ziel immer fehl**
- **Prolog bietet damit dem Programmierer die Möglichkeit, die Beweisführung in einem bestimmten Bereich des Beweisbaums fehlschlagen zu lassen**
- **Das ist angebracht wenn die Resolutionswahl einer Klausel sich für den Programmierer von vorne herein als die falsche herausstellt**

- **Beispiel:**

```
mutter(Mutti,Kind) :- mann(Mutti), fail.
mutter(Mutti,Kind) :- gezeugt(Mutti,V,Kind).
```

- **Aber: auch hier gibt es oft bessere Lösungsalternative statt Verwendung von 'fail'**

## Failure als Lösungssammler

- Technisch gesehen forciert ein fail ein Backtracking zum letzten Auswahlpunkt. Dieser kann jedoch auch noch vor dem Kopf der Regel liegen
- Das kann man sich zu nutzen machen um alle Lösungen für ein Prädikat zu finden und schreiben zu lassen:

```
write_all(Goal) :-
        Goal,
        write(Goal), nl, fail.
write_all(_) :-
        write('no more solutions'), nl.
```

- Wegen **fail** backtrackt Prolog zurück zum Goal (**write** und **nl** haben keine Auswahlpunkte)
  Falls es eine alternative Lösung gibt, dann wird dieser wiederum geschrieben usw.
- Aufgrund der letzten Klausel ist write_all immer erfolgreich (bzw. ist **write_all** deterministisch)





## Cut / Fail Paarung (1)

- Cut und fail können sehr gut kombiniert werden um eine Beweisführung komplett abzubrechen

- Beispiel:

```
make_cnf(Formel,CNF) :-
        \+ is_wff(Formel),
        !, fail.
make_cnf(Formel,CNF) :-
        is_implication(Formel),
        !,
        rm_impl(Formel,ImplFree),
        make_cnf(ImplFree,CNF).
make_cnf(Formel,CNF) :-
        …….
```

## Cut/Fail Paarung (2)

- Negation ist bereits in Prolog eingebaut ( \+ ). Mann könnte sie aber leicht selber programmieren:

```
my_not( Goal) :-
      Goal,
      !, fail.
my_not( _Goal).
```





## If-Then-Else

- Mittels Cut kann man elegant eine bedingte Ausführung/Beweisführung programmieren:

```
if_then_else(Condition,TrueCode,_FalseCode) :-
        Condition, !, TrueCode).
if_then_else(_Condition,_TrueCode,FalseCode) :-
        FalseCode.
```

- Allerdings braucht man dieses Prädikat nicht:
  Das Konstrukt ist als eigenes syntaktisches Konstrukt in Prolog eingebaut mittels der Operatoren „`->`“ und „`;`“
  verwendbar im Body einer Klausel
- Obige Code schreibt sich dann als:

```
Condition -> TrueCode ; FalseCode
```

- Aufgepasst:
  `->` bindet schwach. Im Zweifel Klammern setzen!

DHBW Duale Hochschule Baden-Württemberg Stuttgart Campus Horb

## Endloswiederholung repeat

- Das repeat Prädikat stammt noch aus Zeiten der Prolog-Compiler ohne LCO (Last Call Optimization)
- Heute kann man repeat meistens vermeiden durch Verwendung von Tail Recursion (Selbstaufruf eines Prädikats als letztes Goal
- Es gibt allerdings insb. bei interaktiven Prolog-Programmen (Abholen von User/Datei Input) durchaus noch Verwendung für das Prädikat

- Das repeat-Goal ist immer erfolgreich. Es lässt allerdings nicht zu, dass Prolog in seiner Beweisführung beim Backtracking über das Statement weiter zurückgeht, sondern erzwingt dass die Goals, die nach dem repeat Statement stehen, frisch bewiesen werden





DHBW Duale Hochschule Baden-Württemberg Stuttgart Campus Horb

## Repeat und fail

- Man kann erzwingen dass Prolog backtrackt durch verwendung von fail/0
- Vernünftige Benutzung (ohne Prolog zu „killen“) von repeat sieht vor, dass man einen Weg am fail vorbei als Abbruchbedingung programmiert
- Dabei kann dann oft -> mit ; helfen
- Beispiel:

```
repeat(X,Z) :-
      repeat,
      writeln(X),
      read(Y),    % read/1 liest einen Prolog Term aus stdin
      ( Y == stop ->
            writeln('gotto stop')
            ;
            do_something_with(Y,Z), fail
      ).
```

DHBW Duale Hochschule Baden-Württemberg Stuttgart Campus Horb

## Funktionale Arithmetik

- Aus Gründen der Effizienz wurden neben der logischen Relationen, ausgedrückt von Prädikaten, noch Funktionen in Prolog eingebaut, allerdings nur für Arithmetik (im Gegensatz zu funktionalen Sprachen)
- Syntaktisch sehen Funktionen genauso aus wie Relationen. Man kann sie (die arithmetische) allerdings erkennen, weil sie auftreten im Kontext von `is/2`
- Arithmetische Funktionen setzen voraus, dass die rechte Hand von `is/2` vollständig instanziiert (`ground`) ist
- Links von is/2 gehört eine Zahl, oder eine Variable
- Prolog kennt die standard arithmetische Funktionen. Abhängig von der Prologimplementierung gibt es mehr oder weniger gut ausgestattete Arithmetikbibliotheken





DHBW Duale Hochschule Baden-Württemberg Stuttgart Campus Horb

## Selbst arithmetische Funktionen definieren

- SWI-Prolog erlaubt die Definition von eigenen arithmetischen Funkionen mittels `arithmetic_function/1`
- Beispiel aus der Hilfe von SWI-Prolog:

```
:- arithmetic_function(mean/2).
mean(A, B, C) :-
C is (A+B)/2.
```

- Benutzung der Definition:

```
?- A is mean(4, 5).
A = 4.500000
true
```

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

## Einwurf: Deklarative Arithmetik

- **Die in Prolog eingebaute Arithmetik ist prozedural, nicht deklarativ bzw. relational**
  - ?- X is 2+3.
    X = 5.
  - ?- 5 is 2 + X.
    ERROR: Arguments are not sufficiently instantiated
  - ?- 4 + 1 is 5.
    false.

- **Deklarative/Relationale Arithmetik gibt es Rahmen der in allen Prolog verfügbare constraint libraries clpfd (finite domains, die integer) und clpqr (rationale und reele Zahlen)**
  - X #= 2+3.
    X = 5.
    ?- 5 #= 2 + X.
    X = 3.
    ?- 2+3 #= 9–X.
    X = 4.
  - Aber auch:
    ?- X #< 5.
    X in inf..4.
    X #> 5.
    X in 6..sup.
  - X in 0..5, X #\= 2.
    X in 0..1\/3..5.





## Sonstige Funktionen

- **Nicht-arithmetische Funktionen sind nicht weiter im Prolog-Angebot enthalten**
- **Man kann sie jedoch mittels Prologs Makro-Mechanismus definieren und benutzen**
- **… und so Prolog um die meisten Features funktionaler Sprachen (wie LISP, Haskell) erweitern**
- **Das ist jedoch ein Thema, das weit über eine Einführung in Prolog hinausgeht**
- **Für Interessierte: siehe**
  - http://www.swi-prolog.org/pack/list?p=func
  - http://www.swi-prolog.org/pack/list?p=function_expansion
  - http://www.swi-prolog.org/pack/list?p=lambda

## Seiteneffektprogrammierung

- **Rein logische bzw. funktionale Programme sind frei von Seiteneffekten:**
  - Eine Relation/Funktion ist in seinem Verhalten nicht modifizierbar durch externen/globalen Zuständen (d.h. Zuständen die nicht Argumente der Relation oder Funktion sind)
  - Eine Relation/Funktion interagiert mit seiner Umgebung ausschließlich mittels seiner Argumente (Relation, Instanziierung) oder Rückgabewerte (Funktion)
- **Konsequenz:**
  - Werte von Variablen können nicht verändert (bei Relationen: nur weiter instanziiert) werden
  - Ein-/Ausgabe (read/write/load/save) ist nicht ohne weiteres möglich





## Seiteneffektprogrammierung in Prolog

- **Um eine hantierbare Sprache zu sein, hat Prolog diverse Prädikate eingebaut, die eine Eingabe/Ausgabe ermöglichen**
- **Prologs Input und Output Management umfasst:**
  - ISO Input und Output Streams:
    **open/4** und **close/1** bzw. **close/2**, **seek/4**, usw.
  - Primitive character I/O:
    **put/1**, **put/2**, **get/**1, **get/2**, usw.
  - Prolog-Term reading und writing:
    **read/1**, **read/2**,
    **write/1**, **write/2**, **writef/1**, **writef/2**,
    **writeln/1**, **writeln/2**,
    **print/1**, **print/2**, **prompt/2**, **prompt1/2**, usw.
- Diese Goals produzieren Seiteneffekte: beim Backtracking werden die Resultate nicht ungetan gemacht! Der Kontrollfluss beim Backtracking geht gleich über sie weiter zurück zu früheren Goals. (Siehe Beispiel write_all)
- **Siehe für weitere Information z.B. das SWI Prolog Handbuch Kap. 4.17-4.20, sowie Kap. 4.32**

## Einwurf: Deklarative Input

- **In den letzten Jahren gibt es in allen Prologs die Bibliothek Pure I/O: library(pio)**
  - SWI manual:
    "This library provides pure list-based I/O processing for Prolog, where the communication to the actual I/O device is performed transparently through coroutining."

- **Die Hauptprädikat sind phrase_from_file/2 und /3**
  - Man kann dies nur verwenden in Zusammenhang mit DCGs
  - Beispiele folgen nach der Behandlung von DCGs





## Weitere Seiteneffektprogrammierung

- **Während der Ausführung eines Prolog-Programms können Horn-Klausel hinzugefügt oder weggenommen werden!**
- **In dieser Weise ändert sich faktisch die Definition von logischen Prädikaten und somit die Möglichkeiten der logischen Beweisführung**
- **Mehr Information zur Philosophie dahinter:**
  http://www.swi-prolog.org/pldoc/man?section=db

- Dies ist sogar für systemintern definierte Prädikate möglich
  (mittels redefine_system_predicate/1).
  Umdefinieren von Systemprädikaten sollte i.A. aber unterlassen werden
  (Es ist eigentlich nur vorgesehen um Inkompatibilitäten zwischen unterschiedlichen Prologimplementierungen abzufangen – Stichwort: Programmportierung)

## Dynamisch Klausel einem Programm hinzufügen

**Die Prolog-Pradikate womit man Fakte, bzw. Regel hinzufügen kann sind:**

- **assert/1 und assertz/1**
  **(es gibt auch versionen hiervon /2)**
  **fügen Klausel für ein Prädikat am Ende der Klauseldefinitionen für das Prädikat an**
- **asserta/1 (bzw. asserta/2)**
  **macht das Gleiche, jedoch fügt es die Klausel am Anfang, vor bereits vorhandenen Prädikatklauseln an**

- Dies funktioniert grundsätzlich für solchen Prädikate, die mittels des Prolog-Prädikats **dynamic/1** als dynamisch veränderbar deklariert werden
- Beim Backtracking werden die Folgen **nicht** ungetan!





## Dynamisch Klausel einem Programm wegnehmen

**Klausel können gezielt entfernt werden:**

- **retract/1**
  **entfernt eine Klausel, die mit dem Argument von retract unifiziert. mit:**
- **retractall/1**
  **entfernt alle Klausel, die mit dem Argument unifizieren.**
  **(So können komplette Definitionen (sämtliche Fakte und Regel) eines Prädikats entfernt werden)**
- **abolish/1**
  **entfernt alle Klausel eines Prädikats mit den angegebenen Argumentzahl (z.B. abolish(foo/2)) (Was natürlich mit einem geeigneten retractall-Ziel auch geleistet werden kann)**

- Dies funktioniert grundsätzlich für solchen Prädikate, die mittels der Prolog-Operator **dynamic/1** als dynamisch veränderbar deklariert werden
- Beim Backtracking werden die Folgen **nicht** ungetan!

## Sinnvolle Benutzung von assert/retract

- **In Prinzip überall wo insbesondere Fakte über die repräsentierte Realität sich ändern: klassische Datenupdates wie in relationale Datenbanken (Man redet dann auch von Prolog DB-Funktionen)**
- **Es vereinfacht in verschiedenen KI-Anwendungen die Programmierung. Als Beispiel für die Verwendung der Prädikate finden Sie in den Unterlagen ein kleines Planungsprogramm (Blocksworld)**
- **Häufig ist dieser Mechanismus allerdings nicht notwendig. So lässt sich das Blocksworld Programm so abändern, dass es ohne assert und retract auskommt (dadurch, dass die Welt und die Pläne als Argumente in den Prädikaten „mitgenommen“ werden.**





## „Schlechte“ Benutzung von assert/retract

- **Zitat aus dem SWI-Prolog Handbuch:**

  *Typically it is a **bad idea** to use any of the predicates in this section **for realising global variables that can be assigned to**. Typically, first consider representing data processed by your program as terms passed around as predicate arguments*
  *(Kap. 4.14)*

- **Generell ist Seiteneffektprogrammierung mit größter Vorsicht zu genießen: Versuche immer, ob eine andere Lösung möglich ist**
  (Beispiel für Suche nach Alternativen aus Informatik I: fibonacci und ackermann)

## Meta-Logische Konzepte

- **Prolog enthält mehrere Prädikate die es erlauben eine Logik höherer Ordnung zu programmieren**
- **Eine Logik höherer Ordnung trifft nicht nur Aussagen über Eigenschaften von bzw. Relationen zwischen Objekte, sondern auch über Eigenschaften von bzw. Relationen zwischen Eigenschaften und Relationen**
- **Somit kann man in Prolog Fakten bzw. Regeln bzgl. Klausel formulieren**
- **Die wichtigsten Prolog-Prädikate dazu sind:**
  - **findall/3, findall/4, setof/3, bagof/4**
  - **clause/2, clause/3**
  - **univ und Co: =../2, functor/3, arg/3**
  - **call/1, call/2-N**





## findall & Co: Einsammeln von Lösungen (1)

- Man will oft nicht nur eine Lösung eines Goals haben, sondern alle
- Prolog erlaubt natürlich dem Programmierer mittels Seiteneffektprogrammierung dieses zu erreichen:

```
find_all(X,Goal,_) :-
     assertz('$one_solution'('$no_solution')),
     Goal,
     asserta('$one_solution'(X)),
     fail.
find_all(X,_,Result) :-
     retract('$one_solution'(X)),
     reap(X,Tmp), !, reverse(Tmp,Result).

reap(X,[]) :-
     X == '$no_solution',!.
reap(X,[X|Sol]) :-
     X \== '$no_solution',
     retract('$one_solution'(X1)),
     reap(X1,Sol).
```

- **Aber es geht natürlich besser:**
  - Das Prädikat **findall/3** leistet obiges

## findall & Co: Einsammeln von Lösungen (2)

- **findall/3 schlägt nicht fehl: Wenn es keine Lösung gibt, dann wird das 3. Argument mit [] instanzieert.**
- **Mit bagof/3 geht es leicht anders:**
  - bagof/3 schlägt fehl, wenn es keine Lösung gibt
  - bagof/3 liefert alle Lösungen, die bei Bindungen restlicher Variablen im Goal möglich sind
  - bagof/3 liefert (wenn es Lösungen gibt) nur das gleiche Resultat wie findall/3 wenn es keine weitere Variablen im Goal gibt, oder wenn diese alle existentiell gebunden sind

`bagof(C,foo(A,B,C),ALL)` $\subseteq$ `bagof(C,A^B^foo(A,B,C),ALL)`

  - **setof/3** liefert die gleiche Lösungen wie bagof/3, aber in sortierter Reihenfolge ohne Duplikate (eine Menge, eben)
    und schlägt fehl, wenn es keine Lösung gibt.





## clause: Klausel finden

- **Manchmal will man wissen ob es (eine) Klausel gibt, die mit einem Goal unifizieren, ohne diesen Goal beweisen zu wollen**
- **Das macht Prolog Klausel zu Daten (homo-iconic)**
- **Das ist möglich mit clause/2 bzw. clause/3:**
  - clause(*Head*,*Body*) ist erfolgreich, wenn *Head* unifiziert werden kann mit der Head einer Klausel und *Body* mit der Body der gleichen Klausel
  - clause/2 liefert alternative Klausel beim Backtracking
  - Wenn der Klausel ein Fakt ist, dann wird *Body* unifiziert mit dem Atom *true*

- **Kombiniert mit findall & Co kann man so alle anwendbare Klausel von Prolog auffinden lassen**
- **Das kann man sehen als Vorstufe zu Meta-Interpretation**

## Univ & Co (1)

- **functor/3, arg/3 und =../2 machen es möglich:**
  - Klausel zu analysieren
  - Klausel auseinander zu nehmen in ihren Bestandteilen
  - Klausel aus Bestandteilen zusammen zu setzen

- **Zusammen mit clause, assert, retract, usw. kann man Prolog-Prozederen voll dynamisch:**
  - erstellen,
  - überarbeiten, modifizieren,
  - erweitern,
  - kürzen, löschen





## Univ & Co (2)

- **functor/3 und arg/3 haben wir bereits analytisch verwendet. Man kann sie aber auch „umgedreht“ d.h. synthetisch/konstruktiv verwenden:**
  - ?- functor(P,foo,2), arg(1,P,a), arg(2,P,b).
    P = foo(a, b).
- **Arbeiten mit functor/3 und arg/3 ist manchmal ein wenig umständlich. Und manchmal möchte man auch arbeiten ohne voraus zu wissen wie viele Argumente ein Prädikat haben soll. Dafür gibt es den Univ- Operator:**
  - *Analytisch:*
    ?- foo(a,b,c,d) **=..** L.
    L = [foo,a,b,c,d].
  - *Synthetisch:*
    ?- P **=..** [foo,a,[b,c,d],f,X].
    P = foo(a, [b, c, d], f, X).

## call: Meta-Interpretation

- **Man kann explizit einen Goal aufrufen mittels call/1**
  **Dies ist erfolgreich, wenn das Goal es ist**
- **Definition wäre sehr gradlinig:**
  - call(Goal) :- Goal.
- **Statt eines einzelnen Goal können (eingeklammert) mehrere als Argument an call übergeben werden:**
  - call((G1,G2;G3)).
- **call/2-8 erlaubt es, vor dem Aufruf von einem Goal diesem noch beliebig weiteren Argumenten (bis zu 7) anzuhängen:**
  - call(foo(a,B),1,X,[c,d],Z) wird umgeformt zu call(foo(a,B,1,X,[c,d],Z)) und führt so zum Aufruf: foo(a,B,1,X,[c,d],Z)
  - Damit ist es einfach map-, filter- und fold-Prädikate zu realisieren (maplist/2-5 und foldl/4-7 gibt es als library Prädikate)





## Map, Filter, Fold (Reduce) [Selbst gebaut mit call/n]

```
map(_,[],[]) :- !.
map(F,[H|T],[R|Rs]) :-
       call(F,H,R),
       map(F,T,Rs).
```

```
?- map(plus(1),[1,2,3],R).
R = [2,3,4] .
```

```
filter(_,[],[]) :- !.
filter(Test,[H|T],Trues):-
       call(Test,H),
       !,
       Trues = [H|R],
       filter(Test,T,R).
filter(Test,[_|T],Trues) :-
       filter(Test,T,Trues).
```

```
?- filter(=<(2),[1,2,3],R).
R = [2,3] .
```

```
fold(_,Acc,[],Acc) :- !.
fold(F,Acc,[H|T],R) :-
       call(F,Acc,H,Acc2),
       fold(F,Acc2,T,R).
```

```
?- fold(plus,0,[1,2,3],R).
R = 6 .
```

SWI-Prolog:
library(apply) definiert:
**maplist/2-5** und
**foldl/4-7**

Sowie Alternativen für filter:
**include/3**
**exclude/3**
**Partition/4,5**

## Prolog Meta-Interpretation

- **Man kann in Prolog selbst ein Interpreter für Prolog schreiben: das ist dann ein Prolog Meta-Interpreter!!**

- **Beispiel einer sehr einfachen Meta-Interpreter:**

```
solve(true):- !.
solve((FirstQuery,RestOfQueries)):-
    !,
    solve(FirstQuery),
    solve(RestOfQueries).
solve(Query):-
    clause(Query,Tail),
    solve(Tail).
```





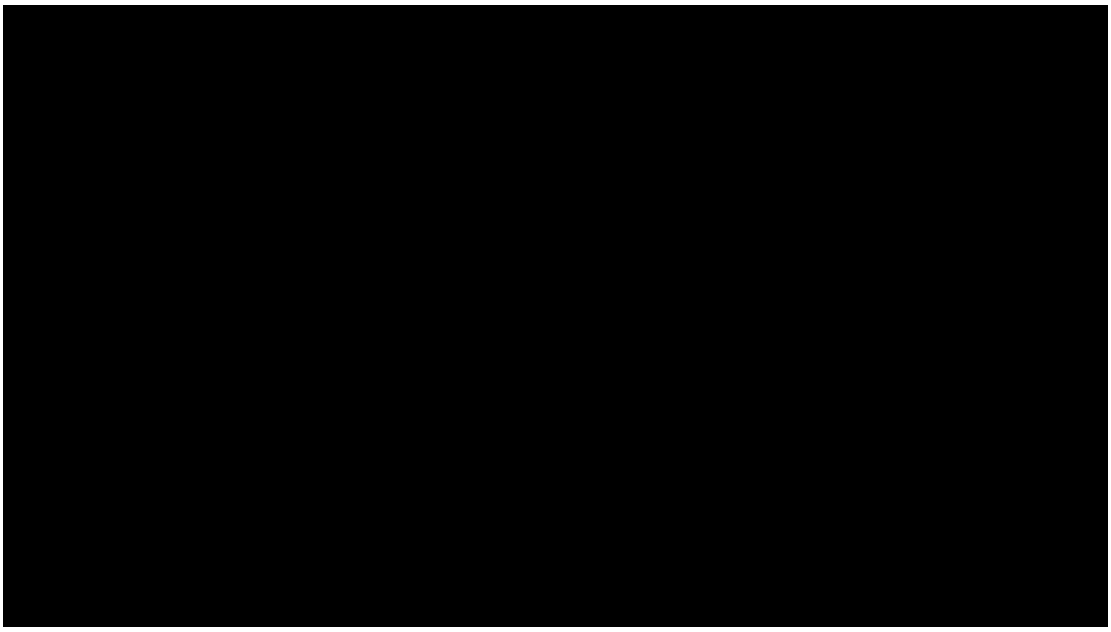
# Alternative Programmierkonzepte (T2INF4271)

**Logische Programmierung**

**04**
**Prolog**
**Prolog Programmierpraxis**

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

## Wie programmiert man in Prolog?



https://www.youtube.com/watch?v=GH4Tg5CYa1k

Mehr info: https://www.youtube.com/watch?v=fQUVWo209WA





## Do & Don‘t

- **Vermeide immer *linksrekursive Aufrufe*: die führen zu Endlosschleifen**
  - `foo(A,B) :- foo(B,C), bar(A,C)`
- **Linksrekursivität kann auch via andere Prädikataufrufe generiert werden: gegenseitig direkt rekursive Aufrufe**
  - `foo(A,B) :- umu(B,C), bar(A,C).`
  - `umu(X,Y) :- foo(X,Z), cheesy(Y,Z).`
- **Man kann zwar dieses Verhalten vermeiden mittels "Tabling" (SLG Resolution, dazu später mehr), aber dies ist sehr speicherintensiv und daher nur mit Vorsicht zu genießen**
- **Versuche immer, rekursive Aufrufe am Ende einer Regel zu stellen: *LCO (Last Call Optimization)*, bwz. TCO**
  **Dies kann man u.a. erreichen mittels extra Prädikatargumente:**
  - Akkumulatoren
  - Zähler (aufwärts und abwärts)

## Vermeide wo möglich "Defaulty" Code

- **Defaulty Code ist Code, die so ist, dass Prolog nicht eindeutig entscheiden kann, welche Klausel zu selektieren ist**
  - Es gibt also mehrere Auswahlmöglichkeiten (Choice Points), die Prolog administrieren muss

- **Beispiel**

**Defaulty:**

```
foo([],true).
Foo(_,false).
```

**Nicht defaulty:**

```
append([],L,L).
append([H|T],L,[H|R] :- append(T,L,R).
```

- **Defaulty Code ist weniger effizient:**
  **Klauselindexierung ist schlechter und Choice Points sind zu verwalten**
  **→ Macht ausführung langsamer**
- **Defaulty Code kann auch schnell semantisch inkorrekt werden und/oder deklaratives (nicht-deterministisches) Verhalten von Prolog verhindern**
  - Siehe z.B.: https://www.youtube.com/watch?v=KJ8uZiWpomg





## Wichtige Prädikate, die Sie kennen sollten
**(neben bereits behandelten)**

- **Viele Prädikate, die mit Listen arbeiten:**
  - `member/2, memberchk/2, append/3, reverse/2`
  - `select/3, sort/2, nth0/3,4, nth1/3,4, last/2`
  - `maplist/2-5, foldl/4-7`
- **Viele Prädikate, die Termen vergleichen:**
  - `dif/2`
  - `compare/3, zcompare/3`
  - `=/2, ==/2, @>/2, @>=/2, @</2, @=</2`
- **Deklaratives if (alternative zum nicht pure Prolog -> ; Konstrukt):**
  - `if_/3` (in library(reif))
- **DCGs**
  - `phrase/2,3`
- **Constraint Logic Programming Prädikate, insb. aus library(clpfd)**
  - `#=, #<, #>, usw.`
  - `in/2, ins/2, label/1, labeling/2`

## Differenzstrukturen

- **Durch Prologs Umgang mit Variablen und Unifikation kann man rekursive Datenstrukturen mit „Löchern“ (zur Aufrufzeit nicht instanziierten Variablen) verwenden**
- **Oft einsetzbar ist die Benutzung von *Differenzlisten***
  **Diese erlauben Listenkonkatenation ohne Verwendung von append/3 und dies in konstanter Zeit!**
- **Differenzlisten gibt es "versteckt" in DCGs (Dazu später mehr)**
- **Ein weiteres Beispiel solcher „offener“ Strukuren stellt die Implementierung eines (Key,Value) Store (als Liste oder auch als binärer Suchbaum), auch Dictionary genannt, dar**





## Programm zu langsam?

- **Arbeite *mit* Prolog, nicht *dagegen*!**
  - Beschreibe (deklarativ), nicht befehlen (imperativ/prozedural)
- **Don't do it!**
  - Suche einen anderen (deklarativen) Ansatz
    - z.B. div. Ansätze für fibonacci und Fakultätsfunktion (siehe Informatik I)
  - Implementiere einen anderen Algorithmus
    - z.B. bei Suchen wechseln zu A* (hierzu später mehr)
    - Evtl. überlegen, ob hierfür eine andere Sprache nicht geigneter ist
- **Don't do it again!**
  - Implementiere memoization (Zwischenlösungen speichern)
  - Benutze tabling/SLG-Resolution (dazu später mehr)