Programmiertechniken in der Computerlinguistik I

Universität Zürich

Computerlinguistik

Wintersemester 2001/2002

- **Dozent**
  Simon Clematide ‹siclemat@ifi.unizh.ch›
- **Übungsbetreuung**
  Sonja Brodersen ‹broder@ifi.unizh.ch›

- **Web**
  http://www.ifi.unizh.ch/cl/siclemat/lehre/ws0102/pcl1

Programmiertechniken in der Computerlinguistik I

Wintersemester 2001/2002

Inhaltsverzeichnis

Aufbau der Lehrveranstaltung

1. Quartal: PROLOG-Einführungskurs

- Einführung in Programmiersprache PROLOG
- Programmiertechniken (Listen, Rekursion…)

2. Quartal: Erste computerlinguistische Anwendungen

- Elementare Verfahren, um die Struktur eines Satzes (entsprechend einer Grammatik) zu berechnen/verarbeiten

Semesterende: Mini-Test

- fakultativ (ersetzt *nicht* die Akzess-Prüfung)
- Testat erleichtert allfälliges Wechseln der Universität

üben, üben, üben…

Programmieren lernen ohne selbst am Computer zu arbeiten ist illusorisch!

Übungsaufgaben

- normalerweise wöchentlich (Aufwand gut 2h pro Woche)
- betreute Übungsstunde (ev. noch 1 Spezial-Übungsstunde)
- schriftliche Abgabe (mit kommentierter Rückgabe)
  - E-Mail (Programmtext bitte direkt in Mail, nicht als Datei anhängen!):

    ```
    Subject: Prologkurs Uebung 1
    To: broder@ifi.unizh.ch
    ```
- Kurzbesprechung jeweils in der nächsten Stunde
  - Abgabe von Musterlösungen

Lösen der Übungsaufgaben

Offizielle Übungsstunde (Prolog unter Windows 95)

- Uni Irchel, Winterthurerstr. 190, Gebäude 11, Raum Y01-F08 (Eingang gegenüber dem Eingang zum Studentenladen)
  - Montag 10.15-12.00h
  - Betreuung durch Sonja Brodersen

Weitere Übungsmöglichkeiten (Prolog unter MacOS)

- In den Übungsräumen kann gearbeitet werden, wenn sie nicht reserviert sind! (normalerweise ab 20h bis 22h, sowie Mi/Do 10-12h). **Wichtig**: UniAccess Login und Passwort verfügbar haben!
  - Rämistr. 74, Raum U 107
  - IFI, Raum 27-G-25/28

Lösen der Übungsaufgaben

Zuhause am Computer

- Abgabe des professionellen Prolog-Systems "SICStus PROLOG" auf CD-ROM
  - Plattformen: Aktuelle Version für Win95/98/NT/2000, Linux, MacOS X; ältere Version für MacOS (68k, PPC), Win 3.11; andere Plattformen auf Anfrage
- Kosten für Einzellizenz
  - CHF 50.– für Nicht-Mitglieder der Fachschaft Computerlinguistik
  - CHF 25.– für Mitglieder der Fachschaft Computerlinguistik

Frei erhältliche Alternative zu SICStus Prolog (für Win und Linux)

SWI-Prolog: `http://www.swi.psy.uva.nl/projects/SWI-Prolog`

- Achtung: Jedes Prologsystem ist unterschiedlich! Wir verwenden und vermitteln hier am Lehrstuhl nur SICStus Prolog!

Folien und Übungsblätter

Die Folien und Übungsblätter sind im WWW verfügbar.

- Adresse: http://www.ifi.unizh.ch/cl/siclemat/lehre/ws0102/pcl1
- Format: PDF-Dateien für Adobe Acrobat
- Programm zum Lesen der PDF-Dateien (Adobe Acrobat Reader 4; bei Version z.T. Probleme beim Betrachten/Drucken)
  - auf der SICStus-CD-ROM (jeweils Unterverzeichnis READER)
  - Neueste Version ab WWW:

    `http://www.adobe.com/prodindex/acrobat/readstep.html`
- Für die ohne Drucker:
  - Kopiervorlage des Skripts befindet sich in IFI-Bibliothek beim CL-Gestell

Übungs- und Lösungsblätter werden in Stunde verteilt.

- Wer will, bitte in Liste eintragen und nächstens CHF 3.– mitbringen

Einstiegsliteratur I

Kombinierte Einführungen zu Prolog und NLP (Natural Language Processing) in Buchform

- Esther **König**/Roland **Seiffert**: Grundkurs PROLOG für Linguisten. Tübingen: Francke, 199 Seiten,1989. (UTB 1525). DM 25.– *[Dt. Einführung in Prolog und simpelstes NLP. Äusserst verständlich geschrieben! "Prolog für Dummies"]*
- Clive **Matthews**: An Introduction to Natural Language Processing through Prolog. London: Longman, 306 Seiten, 1998. USD 34.– *[Gute engl. Prolog-Einführung und grundlegendendste NLP-Anwendungen. Empfehlenswert, und man lernt die engl. Fachtermini (mit Glossar)!]*
- Wilhelm **Weisweber**: Prolog: Logische Programmierung in der Praxis: Thomson, 384 Seiten, 1997. DM ca. 100.– *[Sehr sorgfältige dt. Prolog-Einführung und wichtigen Syntax-Anwendungen. Etwas teuer!]*

Einstiegsliteratur II

Frei im Web erhältlich

- Christof **Rumpf**: Grundkurs Prolog WS '99/00. WWW:

  `http://asw-i8.phil-fak.uni-duesseldorf.de/~rumpf/gkpro99/prologws99.htm`

  [Dt. Vorlesungsunterlagen zu einer sehr guten Prolog-Einführung mit einfachen NLP-Anwendungen wie maschinelle Übersetzung.]
- Stefan **Müller**: Prolog und Computerlinguistik: Teil I - Syntax. 1998 (Z.T. Übersetzung aus einem engl. Skript). 183 Seiten. WWW:

  `http://www.dfki.de/~stefan/Pub/prolog.html`

  [Dt. Einführung in Syntaxanalyse mit fundierter Theorie und Prolog-Implementationen. Sehr gute Qualität, aber z.T. erst in PCL II sinnvoll!]
- Ulf **Nilsson**, J. **Maluszynski**:Logic, Programming and Prolog

  `http://www.ida.liu.se/~ulfni/lpp/`

Standardliteratur

Computerlinguistik mit Prolog

- Michael A. **Covington**: Natural Language Processing for Prolog Programmers. Prentice Hall, 350 Seiten, 1994. USD 66.– *[Sehr sorgfältig aufgebaute Programme! Ausgezeichneter Programmier- und Analysestil!]*
- Gerald **Gazdar**/Chris **Mellish**: Natural Language Processing in PROLOG: An Introduction to Computational Linguistics. Addison-Wesley, 504 Seiten, 1989. *[Gute Beispiele, etwas weniger verständlich geschrieben als Covington!]*

Kommentar

- Die Bücher behandeln unterschiedlichste Bereiche der Computerlinguistik (Syntax, Morphologie, Semantik, Pragmatik) und präsentieren die Ansätze, wie sie in den 80er-Jahren entwickelt wurden. Als Grundlage immer noch nützlich!

Prologliteratur

Der Prolog-Klassiker in 4. Auflage

- W.F. **Clocksin**/C.S. **Mellish**: Programming in Prolog. Springer, 282 Seiten, 1994. USD 37.–

Ein IFI-Produkt

- Norbert E. Fuchs: Kurs in logischer Programmierung. Wien: Springer, 224 Seiten, 1990. DM 53.– *[Dt. Prologkurs. Mehrere Exemplare in der Hauptbibliothek Irchel!]*

SICStus-Prolog Homepage

- Handbuch mit Prädikatsindex
- Mit Verweisen auf Infos, Bibliotheken, Programme usw.

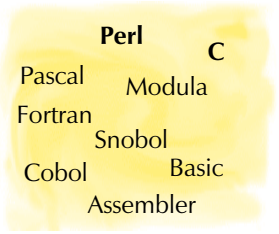
  `http://www.sics.se/sicstus`

Einführung

Übersicht

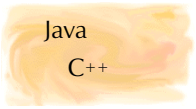
- Programmiersprachen
- Etwas Geschichte
- Was heisst Programmieren?
- Arten von Schlussfolgern
  - Natürlich
  - Formal
  - Mechanisch
- Programmieren und Schlussfolgern in PROLOG

Arten von Programmiersprachen



Prozedural/Imperativ

Was muss ich tun, um das Problem zu lösen?



Objektorientiert

Funktional

Logisch/Deklarativ

Was weiss ich über das Problem?

Kurzgeschichte

Entwicklung von Prolog

- **1972** Erster Prolog-Interpreter (Marseille: Gruppe Colmerauer)
- **1977** DEC-10: Prolog-Compiler: Edinburgh Standard
- **1983** WAM (*Warren Abstract Machine*): Einfaches Ausführungsmodell
- **1983-1993** Starke Verbreitung von Prologs verschiedenster Art (*Golden Age*)
  - Modul-Systeme, um grössere Projekte einfacher verwalten zu können
  - Einbettung von Prolog in "normale" Programme
  - Objektorientierung, …
- **1995** ISO-Prolog-Standard Teil I(ISO/IEC 13211-1:1995)
- … ?

Prolog heisst Programmieren in Logik

Prolog = Programmieren in **Log**ik

- Was heisst "Programmieren"?
  - Text schreiben, den eine (reale oder virtuelle) Maschine mechanisch ausführen kann
  - Problem lösen (*problem solving*): Für eine Klasse von Aufgaben soll ein allgemeines Verfahren entwickelt werden, das die Lösung berechnet.
- Was heisst "in Logik"?
  - Text wird in einem logischen Formalismus notiert.
  - Berechnung der Problemlösung durch Schlussfolgern/Inferenz aus einer deklarativen Problembeschreibung
    - Welche Objekte bestehen?
    - Welche Eigenschaften haben die Objekte?
    - In welchen Beziehungen stehen die Objekte?

"Natürliches" Schliessen

Sokrates ist ein Mensch.	***Prämisse***
Alle Menschen sind sterblich.	***Prämisse***
---	---
Sokrates ist sterblich.	***Folgerung***

- **Natürliche Sprache:** Deutsch
- **Natürlicher Schluss:** +/- intuitiv

Formalisiertes Schliessen

mensch(sokrates)	***A***
$\forall x$ mensch(x) $\rightarrow$ sterblich(x)	***A*** $\rightarrow$ ***B***
---	---
sterblich(sokrates)	***B***

- **Formalisierte Sprache:** Prädikatenlogik
- **Schlussregel:** Modus Ponens (Formale Korrektheit)

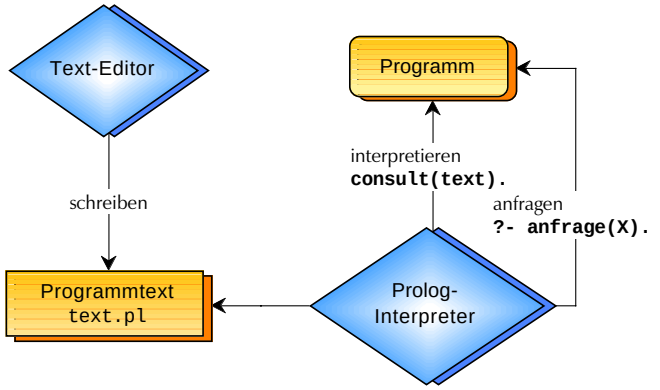
Mechanisches Schliessen

`mensch(sokrates).` `sterblich(X) :- mensch(X).`	***Datenbasis***
---	---

`?- sterblich(X).`
`X = sokrates`
`yes`
	Beweisanfrage
	Instantiierung

- **Formale Sprache:** Prolog (Hornklauellogik)
- **Schlussregel:** Resolution (Formale Korrektheit +Mechanisches Schliessen)

Programmierumgebung



Fakten, Regeln, Anfragen

Übersicht

- Programmieren in Prolog
- Fakten
- Eigenschaften, Gegenstände und Beziehungen
- Prädikate
- Regeln
- Variablen
- Anfragen
- Formalisierung

Programmieren mit Prolog

A. Wissensbasis für Problem erstellen:

- **Fakten** und
- **Regeln** modellieren Miniwelt.

```
mensch(sokrates).
sterblich(X) :-
   mensch(X).
```

B. Anfrage für Wissensbasis stellen:

- **Inferenzmaschine** versucht,
- **Anfragen** an Hand der Wissensbasis zu beweisen.

```
?- sterblich(sokrates).
yes
?- sterblich(platon).
no
```


Fakten modellieren Eigenschaften

Familie Meiers Miniwelt Teil I

- Hans und Klara Meier mit Tochter Gabi, Sohn Kevin und Hund Fido

… zuerst der Hund…

"Fido heisst der Hund."
Faktum in Umgangssprache formuliert

X ist ein Hund.
Eigenschaftsschema

```
hund(fido).
```
Faktum als Prolog Programm-Kode

Das einzelne Faktum

- modelliert die Hunde der Miniwelt.
- definiert das **Prädikat** für die Eigenschaft "ist ein Hund".

Fakten modellieren Eigenschaften

Familie Meiers Miniwelt Teil II

- Hans und Klara Meier mit Tochter Gabi, Sohn Kevin und Hund Fido

… dann die Menschen …

"Hans, Klara, Gabi und Kevin sind Personen."

X ist eine Person.

```
person(hans).
person(klara).
person(gabi).
person(kevin).
```

Die obigen vier Fakten

- modellieren die Personen der Miniwelt.
- definieren das Prädikat für die Eigenschaft "ist eine Person".

Fakten modellieren Eigenschaften

Familie Meiers Miniwelt Teil III

- Hans und Klara Meier mit Tochter Gabi, Sohn Kevin und Hund Fido

"Hans, Kevin und Fido sind männlich."

X ist männlich.

```
maennlich(hans).
maennlich(kevin).
maennlich(fido).
```

"Klara und Gabi sind weiblich."

X ist weiblich.

```
weiblich(klara).
weiblich(gabi).
```

Die obigen Fakten

- modellieren Männliches und Weibliches der Miniwelt.
- definieren je ein Prädikat für die Eigenschaften "ist männlich" und "ist weiblich".

Fakten modellieren Beziehungen

Familie Meiers Miniwelt Teil IV

- Hans und Klara Meier mit Tochter Gabi, Sohn Kevin und Hund Fido

"Gabi und Kevin sind Kinder von Hans und Klara."

X ist Kind von Y.

Beziehungsschema (Relationsschema)

```
kind(kevin, hans).
kind(kevin, klara).
kind(gabi, hans).
kind(gabi, klara).
```

Die 4 Fakten

- modellieren die Kinderbeziehung der Miniwelt.
- definieren das **Prädikat** für die Beziehung "X ist Kind von Y".

Beziehungen (Relationen)

Die Reihenfolge der Objekte in den einzelnen Fakten spielt eine Rolle!

- Wahr: Kevin ist Kind von Hans.
- Falsch: Hans ist Kind von Kevin.

```
kind(kevin, hans).
kind(kevin, klara).
kind(gabi, hans).
kind(gabi, klara).
```

Beziehungen zwischen beliebig vielen Objekten sind möglich!

- Beispiel: Beziehung mit 4 Beteiligten

"Hans gibt Kevin morgen ein Geschenk."

```
gibt(hans, kevin, geschenk, morgen).
```


Schreibweisen

Namen von

- Gegenständen (Objekten, Entitäten) `kevin`
- Eigenschaften `maennlich`
- Beziehungen (Relationen) `kind`

beginnen mit Kleinbuchstaben.

Für Eigenschaften und Beziehungen notiert man zuerst

- Name von Eigenschaft/Beziehung, danach direkt
- in runden Klammern die Namen der beteiligten Objekte
- (durch Kommata getrennt).

```
eigenschaft(objekt)
```

```
beziehung(objekt1, objekt2, … objektn)
```

Regeln modellieren Eigenschaften

Mit Regeln können neue Prädikate aus anderen Prädikaten definiert werden.

- Beispiel: "X ist eine Frau"

"Frauen sind weibliche Personen."

Regel in Umgangssprache formuliert

Jemand ist eine Frau, **falls** dieser *Jemand* eine Person ist **und** derselbe *Jemand* weiblich ist.

Regel in kanonischer Form

```
frau(Jemand) :-
  person(Jemand),
  weiblich(Jemand).
```

Regel als Prolog Programm-Kode

Regeln modellieren Relationen

Mit Regeln können neue Prädikate aus anderen Prädikaten definiert werden.

- Beispiel: "X ist Mutter von Y"

"Mütter sind weibliche Personen mit Kindern."

Regel in Umgangssprache formuliert

X ist Mutter von *Y*, **falls gilt:**
X ist weiblich **und**
Y ist Kind von X.

Regel in kanonischer Form

```
mutter(Mutter, Kind) :-
  weiblich(Mutter),
  kind(Kind, Mutter).
```

Regel als Prolog Programm-Kode

Regeln haben Kopf und Rumpf

```
frau(Jemand) :-
```

Kopf (*Head*)

```
person(Jemand),
weiblich(Jemand).
```

Rumpf (*Body*)

Aus Sicht der Logik

- Rumpf enthält Bedingungen, unter denen der Kopf wahr ist.
- Mehrere Bedingungen sind mit logischen Operatoren (logisches 'Und') verknüpft.

Aus Programmiersicht

- Kopf definiert ein Prädikat, das die Prädikate des Rumpfs benutzt.

Variablen in Fakten und Regeln

Schreibweise

- Variablen beginnen immer mit Grossbuchstaben!

Bedeutung

- In Fakten stehen Variablen für alle Objekte.

```
objekt(X).
```

- In Regeln stehen die Variablen, die im Kopf erscheinen für jedes Mögliche (*allquantifiziert*); Variablen, die nur im Rumpf erscheinen, stehen für irgendein Mögliches (*existenzquantifiziert*).

Jedes *X* ist Grosskind von **jedem** *Y*, falls gilt:
X ist Kind von irgend **einem** Z und
dieses Z ist Kind von Y.

```
gross_kind(X, Y) :-
  kind(X, Z),
  kind(Z, Y).
```

Einfache Anfragen über Familie Meier

Wenn Prolog die Wissensbasis über Familie Meier konsultiert (interpretiert) hat, kann es Anfragen beantworten.

- **Ja-Nein-Fragen:** "Ist Fido ein Hund?"
- **Ja-Nein-Fragen:** "Ist Kevin ein Hund?"

```
?- hund(fido).
yes

?- hund(kevin).
no
```

- **Ergänzungs-Fragen:** "Wer ist alles eine Frau?"

```
?- frau(Wer).
Wer = klara ? ;
Wer = gabi ? ;
no
```

- *Die Inferenzmaschine kann mehr als eine Lösung erschliessen. Mit Strichpunkt werden Alternativen aufgezählt!*

Zusammengesetzte Anfragen

Einfache Anfragen können mit logischem UND verknüpft werden:

- Ist der Hund von Meiers weiblich?

```
?- hund(Hund), weiblich(Hund).
no
```

- Ist der Hund von Meiers männlich?

```
?- hund(Hund), maennlich(Hund).
Hund = fido ?
yes
```


Formalisierung von Regeln

Regeln können ganz verschieden formuliert sein...

- Frauen sind weibliche Personen.
- Wer eine Person und weiblich ist, ist eine Frau.
- Wenn eine Person weiblich ist, dann ist sie eine Frau.
- Alle weiblichen Personen sind Frauen.
- …

Kanonisch

X ist eine Frau, **falls** gilt:
dieses ***X*** ist eine Person ist **und**
dieses ***X*** ist weiblich.

```
frau(X) :-
  person(X),
  weiblich(X).
```


Formalisierung

Fakten

- Der Anwendungszweck des Programms bestimmt, was an grundlegenden Fakten modelliert wird.

Prädikate

- Hans ist eine Person. (Nomen)
- Klara ist weiblich. (Adjektiv)
- Hans liebt Fido. (Verb)

```
person(hans).
weiblich(klara).
liebt(hans, fido).
```

Sprechende Namen

- Möglichst aussagekräftige Bezeichner helfen beim Verständnis von Programmen!
- Prolog selbst sind sprechende Namen schnuppe!

Syntax und Datenstrukturen

Übersicht

- Woraus besteht ein Prolog-Programm?
- Syntaxdiagramme für Bildungsregeln
  - Programm, Klausel, Anfrage, Term
- 3 Sorten von Termen
  - atomare Terme, Variablen, komplexe Terme
- Termklassifikation in Prolog
  - atomic/1, atom/1, number/1, integer/1, float/1, var/1, compound/1
- Prädikate vs. komplexe Objektbezeichnungen
- Kommentare

Bestandteile von Prolog-Programmen

Ein Programm besteht aus

- **Fakten**

  Fido ist ein Hund.
  Hans ist Gabis Vater.

```
hund(fido).
vater(hans, gabi).
```

- **Regeln**

  Hunde sind bissig.

```
bissig(X) :- hund(X).
```

An ein solches Programm werden Anfragen gestellt

- **Anfragen**

  Ist Fido bissig?
  Welche Kinder hat Hans?

```
?- bissig(fido).
?- vater(hans, Kinder).
```


Legende Syntaxdiagramme

Nicht-Terminales Symbol

- Klasse von Ausdrücken der Sprache

Terminales Symbol

- Wörtlicher Text der definierten Sprache

NT

Definiertes Nicht-Terminales Symbol

- durch nachfolgenden Diagramm definiertes Symbol

Übergang (mit beliebigem Leertext)

- Leerzeichen, Zeilenenden, Kommentare

Übergang (kein Leertext erlaubt)

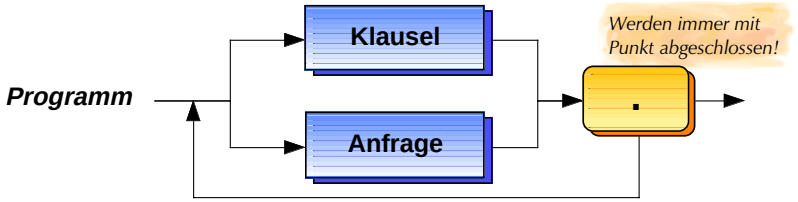
- verbundene Elemente müssen direkt folgen

Lesen von Syntaxdiagrammen

Terminale und Nicht-Terminale Symbole sind durch Einbahnstrassen verbunden

- **Start**: rechts neben dem kursiv geschriebenen Nicht-Terminal ohne Kästchen
- **Weiterfahren**: immer in Pfeilrichtung
- **Schluss**: wo ein Pfeil ins Weisse zeigt
- **Rundes Kästchen**: die Zeichenkette im Kästchen muss wörtlich angetroffen werden
- **Eckiges Kästchen**: das Syntaxdiagramm des im Kästchen erwähnten Nicht-Terminals muss durchlaufen werden
- **Ziel**: vom Start bis zum Schluss durchkommen

Syntax von Programmen



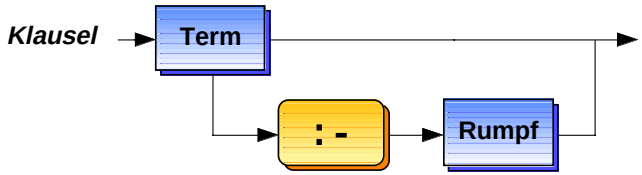
Klauseln (*clauses*)

- Definieren **Prädikate** *(predicates)* durch Fakten und Regeln

Anfragen (*queries*)

- Spezifizieren **Beweisziele** *(goals)*

Syntax von Klauseln



Klausel (*clause*) ist Überbegriff für Fakten und Regeln.

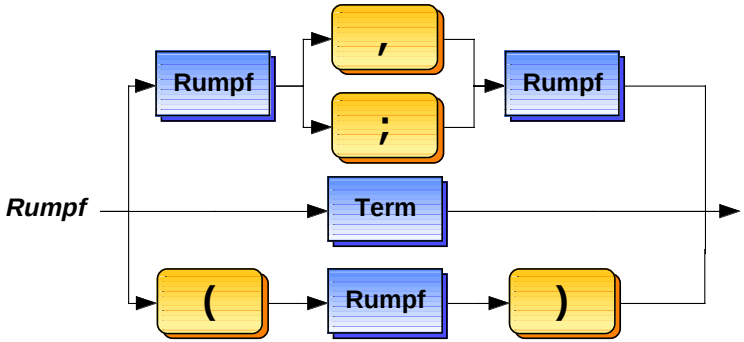
Fakten (*facts*)

- bestehen aus einem Term

Regeln (*rules*)

- bestehen aus einem Term – genannt Kopf (*head*) –, dem Terminal-Symbol ":–" und einem Rumpf (*body*)

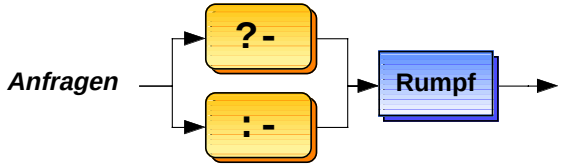
Syntax von Rümpfen



Rümpfe

- Beliebig komplexe Verschachtelung ist möglich!

Syntax von Anfragen

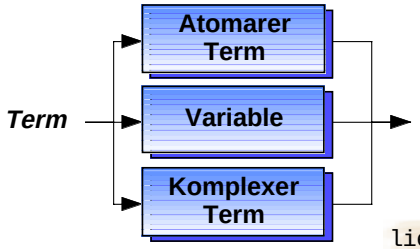


Anfragen (*queries*) sind sofort zu beweisen!

- "?-" für **interaktive** Anfragen: Prolog-Interpreter kann mehrere Antworten (yes/no mit ev. Variablenbelegungen) ausgeben.
- ":-" für **Anweisungen** (**Direktiven**) in Dateien. Höchstens *eine Lösung* wird berechnet und keine Antwort erzeugt!
  - Per Direktiven werden z.B. andere Prolog-Dateien automatisch aus Prolog-Dateien konsultiert.

```
:- consult(datei2).
```

Syntax von Termen



`klara`

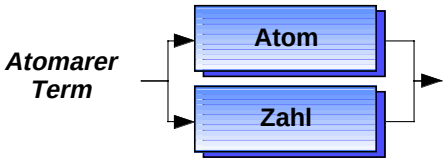
`X`

`liebt(paul, mutter(gabi))`

Terme (*terms*) sind

- atomare Terme (*atomic terms*)
- Variablen (*variables*)
- komplexe Terme (*compound terms*)

Atomare Terme



Atomare Terme (*atomic terms*) sind

- Atome (*atoms*)
- Zahlen (*numbers*)

Das eingebaute Prädikat **atomic**(*T*) ist wahr, falls *T* ein atomarer Term ist.

```
?- atomic(fido).
yes
```

```
?- atomic(23).
yes
```

```
?- atomic(X).
no
```


Atome

Lexikalische Bildungsregeln

- **Normale Atome**: Kleinbuchstabe, gefolgt von beliebig vielen Klein-, Grossbuchstaben, Ziffern und "_" `klara` `a4711_b23_`
- **Zitierte Atome**: beliebige Zeichen zwischen zwei Hochkommata
  - Um Hochkommata in solchen Atomen zu schreiben, muss man sie verdoppeln!

  `'Ich bin ein Atom.'`

  `'Atom mit ''nem Hochkomma.'`
- **Symbolatome**: beliebige Folge aus `+-*/\^<>=`~:.?@#$&` `+` `>=`
- **Sonderatome**: `!`, `;`, `[]`, `{}`

X ist Atom – X ist Zahl

Klassifikationsprädikat für Atome

Das eingebaute Prädikat **atom**(*T*) ist wahr, falls *T* ein Atom ist.

```
?- atom(fido).
yes
```

```
?- atom('23').
yes
```

```
?- atom(hund(fido)).
no
```

```
?- atom(23).
no
```

```
?- atom(X).
no
```

Klassifikationsprädikat für Zahlen

Das eingebaute Prädikat **number**(*T*) ist wahr, falls *T* eine Zahl ist.

```
?- number(23).
yes
```

```
?- number(-4.5).
yes
```

Zahlen

Entsprechend der internen Repräsentation werden auf Rechnern meist 2 Zahlentypen unterschieden.

- Ganzzahlen (*integers*) `12` `-20`
- Gleitpunkt- oder Gleitkommazahlen (*floats*) `123.45` `-2.0e4`

Das eingebaute Prädikat **integer**(*T*) ist wahr, falls *T* eine Ganzzahl ist.

```
?- integer(3).
yes
```

```
?- integer(2.0).
no
```

Das eingebaute Prädikat **float**(*T*) ist wahr, falls *T* eine Gleitpunktzahl ist.

```
?- float(-23.e4).
yes
```

```
?- float(2).
no
```


Variablen

Lexikalische Bildungsregeln

- **Normale Variablen**: Grossbuchstabe oder "_" (Unterstrich), gefolgt von beliebig vielen Gross-, Kleinbuchstaben, Ziffern und "_".

  `Futter` `_futter4FIDO`

- **Spezielle anonyme Variablen**: "_"
  - Jeder einzelne Unterstrich in einer Klausel bezieht sich auf eine andere anonyme Variable.
  - Mit der anonymen Variable drücken wir aus, dass uns das betreffende Objekt nicht interessiert.

```
vater(Vater) :-
  kind(_, Vater),
  maennlich(Vater).
```


Variablen als Platzhalter

Variablen sind Platzhalter.

- Wem alles die Eigenschaft 'Frau zu sein' zugesprochen wird, hängt davon ab, wer eine Person und weiblich ist.

```
frau(Jemand) :-
  person(Jemand),
  weiblich(Jemand).
```

Innerhalb einer Klausel oder einer Anfrage stehen gleiche Variablen immer für das Gleiche.

- Es wäre schlecht, wenn das Frausein aus der Personenhaftigkeit und Weiblichkeit unterschiedlicher Wesen bestehen könnte.

Achtung: Variablen mit unterschiedlichem Namen können manchmal auch für das Gleiche stehen.

Variable oder nicht Variable?

Das eingebaute Prädikat **var**(*T*) ist wahr, falls *T* eine Variable ist.

```
?- var(X).
yes
```

```
?- var(hund).
no
```

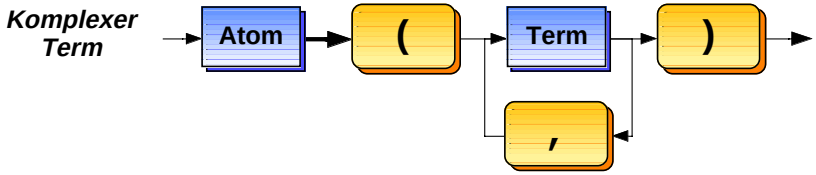
- Ob ein Term eine Variable ist, kann nicht textuell entschieden werden, da Variablen während dem Beweis zu anderen Termen instantiiert werden können.

```
liebt(katrin, Alles).
liebt(mauz, whiskas).
```

```
?- liebt(katrin, Was), var(Was).
true ?
yes
```

```
?- liebt(mauz, Was), var(Was).
no
```

Syntax von komplexen Termen



Ein komplexer Term (*compound term*) besteht aus einem Atom, direkt gefolgt von einem oder mehreren durch Kommata getrennte Terme in Klammern.

- Komplexe Terme enthalten wiederum Terme. D.H. beliebige Schachtelung ist möglich!

X ist komplexer Term

Das eingebaute Prädikat **compound**(*T*) ist wahr, falls *T* ein komplexer Term ist.

```
?- compound(hund(fido)).
yes
```

```
?- compound(hund(X)).
yes
```

```
?- compound(1).
no
```

```
?- compound(X).
no
```


Funktor und Argumente

Termanalyse

- **Funktoren** stehen vor den Klammern.
- **Argumente des Funktors** stehen zwischen den Klammern.
- Argumente einer Ebene werden von links nach rechts durchnummeriert.

  `vater(hans, schwester(kevin))`
  - `hans` ist 1. Argument des Funktors `vater`
  - `schwester(kevin)` ist 2. Argument von `vater`
  - `kevin` ist 1. Argument des Funktors `schwester`

Stelligkeit von Termen

Argumentanzahl eines Terms heisst Stelligkeit (*arity*)

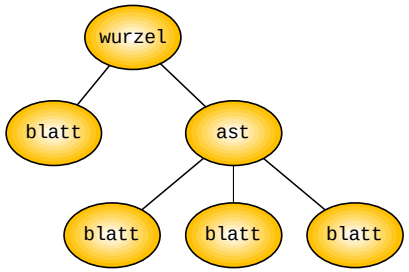
- Kurznotation im Prolog-Slang
  - Funktor und Stelligkeit durch Schrägstrich getrennt
  - ohne die Argumente zu nennen

n	*n-stelliger Term*	*Funktor/Stelligkeit*
0	`fido`	`fido/0`
1	`hund(fido)`	`hund/1`
2	`frisst(fido, X)`	`frisst/2`
3	`gibt(hans, fido, fleisch)`	`gibt/3`
…	…	…
n	`funktor(a_1, a_2, …, a_n)`	`funktor/n`

Terme als Bäume

Terme können graphisch als Bäume dargestellt werden.

- Funktor: Verzweigung
- Argumente: Äste
- Stelligkeit: Anzahl Äste
- Atomare Terme: Blätter




```
wurzel(blatt,ast(blatt,blatt,blatt))
```


Verwendung von Termen

Atome werden verwendet als Name von

- Objekten
- Relationen und Eigenschaften

Zahlen werden verwendet als Name für

- numerische Grössen

Variablen werden verwendet als Platzhalter für

- Terme

Komplexe Terme werden verwendet für

- Prädikatsausdrücke
- komplexe Objektbezeichnungen

Komplexe Objektbezeichnungen

Nicht alle Objekte, die wir bezeichnen, müssen einen Namen tragen.

"Klara liebt den Vater von Kevin."

```
liebt(klara, vater(kevin)).
```

- `vater` ist hier kein Prädikatsname, sondern Teil einer komplexen Objektsbezeichnung – ein Funktionsausdruck.

Prädikatsausdrücke und komplexe Namen sind gleich gebaut, bedeuten aber verschiedenes!

"der Vater von Kevin"

```
vater(kevin)
```

Komplexer Name

"Kevin ist Vater."

```
vater(kevin).
```

Prädikatsausdruck

Kommentare

```
/* Bla bla.
   Bla bla bla. */
```

Zwischen / und */*

```
% Bla bla.
% Bla bla bla.
```

Ab % bis Zeilenende

Kommentare erhöhen die Verständlichkeit für Menschen

- Prolog-Interpreter ignorieren Kommentare
  - Sie werden wie ein Leerzeichen aufgefasst!
- ein Programm ohne Kommentare ist nur sehr schwer verständlich
  - Sogar für AutorIn des Programms, wenn etwas Zeit verstrichen ist!
- besser zu viel als zu wenig kommentieren

Beweisen mit Prolog

Übersicht

- Unifikation
- Substitution
  - Instanz und Variablenbindung
- Wie werden Anfragen bewiesen?
  - Beweisziele
  - Passende Klauseln
  - Beweisregel für Fakten
  - Beweisregel für Regeln
  - Backtracking
- Beweisbaum/Suchbaum

Das Problem

Wie kann Prolog aus Fakten und Regeln…

```
person(hans).
person(gabi).
person(klara).
```

```
weiblich(klara).
weiblich(gabi).
```

```
frau(X) :-
  person(X),
  weiblich(X).
```

…Anfragen beantworten?

Anders gesagt

Wie beweist Prolog, dass 'frau(klara)' und 'frau(gabi)' aus obiger Wissenbasis folgen?

```
?- frau(Wer).
Wer = klara ? ;
Wer = gabi ? ;
no
```

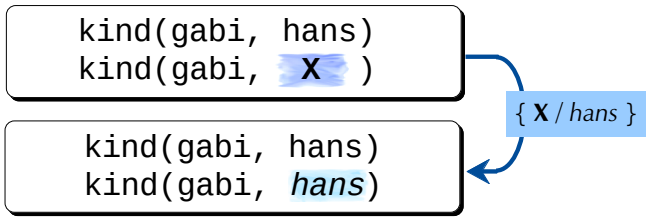

Ingredienzen

Was braucht Prolog zum Beweisen?

- **Termmanipulation**
  - Unifikation
  - Substitution
- **Beweisregeln**
  - für Fakten und für Regeln
- **Suchstrategie**
  - von oben nach unten
- **Backtracking**
  - Entscheidungspunkte

Termmanipulation: Unifikation

Unifikation versucht, zwei Terme gleich zu machen, indem Variablen so weit wie nötig ersetzt werden.



Bei Ersetzung (*Substitution*) von **X** durch *hans* werden die beiden Terme gleich.

Termmanipulation: Substitution

Im Term *T* eine Variable *V* durch Term *S* **substituieren**, heisst **alle** Vorkommen von *V* in *T* durch *S* ersetzen.

Schematisch

$$T' = T\{V/S\}$$

kind(gabi, hans) = kind(gabi,X) {X/hans}

Instanz

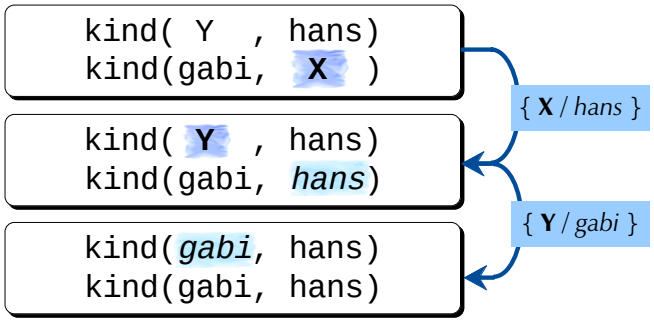
T' heisst **Instanz** von *T*. *T* wurde **instantiiert** zu *T'*.

Variablenbindung

Wenn beim Beweisen eine Variable *V* durch einen Term *S* ersetzt wird, spricht man davon, dass *V* an *S* **gebunden** wurde.

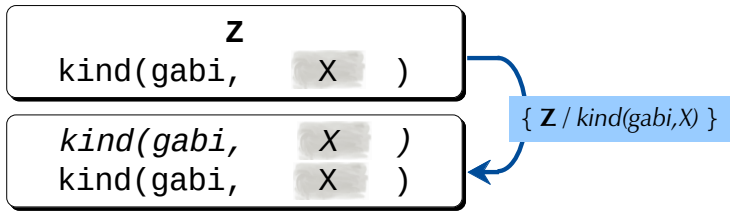
Unifikation durch Substitution

Manchmal müssen mehrere Variablen substituiert werden, um die beiden Terme identisch zu machen.



Unifikation durch Substitution

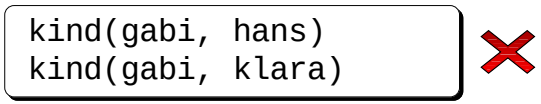
Es brauchen nicht *alle* Variablen substituiert zu werden.



- ***Aber*** *es müssen alle* Vorkommen *einer Variable ersetzt werden.*

Nicht unifizierbare Terme

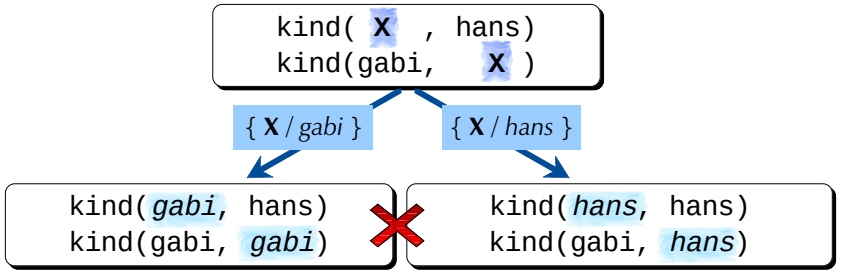
Manchmal gibt es keine Möglichkeit, Variablen zu ersetzen, damit zwei Terme identisch werden:



Die Unifikation **scheitert** *(unification failure)*.

Nicht unifizierbare Terme

Manchmal gibt es keine Möglichkeit, Variablen so zu ersetzen, damit zwei Terme identisch werden:



Unifizierbarkeit

Zwei Terme *T* und *U* sind unifizierbar, genau dann wenn gilt:

- *T* und *U* sind identische atomare Terme,
- **oder** *T* oder *U* ist eine Variable,
  - *substituiere alle Vorkommen der Variable*
- **oder** *T* und *U* sind komplexe Terme, wobei gilt:
  - *T* und *U* haben identische Hauptfunktoren,
  - **und** *T* und *U* haben dieselbe Stelligkeit,
  - **und** die einzelnen Argumente sind paarweise unifizierbar.

Unifizierbar oder nicht?

`p4711 = p4711`

{ }

Ja. (dasselbe Atom)

`X = fido`

{ X / *fido* }

Ja. (ein Variable)

`X = Y`

{ X / *Y* } { Y / *X* }

Ja. (2 Variablen: 2 Möglichkeiten)

`kind(gabi,X) = kind(Y,hans)`

{ X / *hans*, Y / *gabi*}

Ja. (gleicher Funktor kind/2 und paarweise unfizierbare Argumente)

Unifikation in Prolog

Das eingebaute zweistellige Prädikat = ist wahr, wenn seine Argumente unifizierbar sind.

```
?- hans = klara.
no
```

nicht unifizierbar

```
?- hans = hans.
yes
```

unifizierbar

```
?- kind(Y,hans) =
   kind(Y,hans).
true ?
yes
```

unifizierbar, keine Variablenbindung

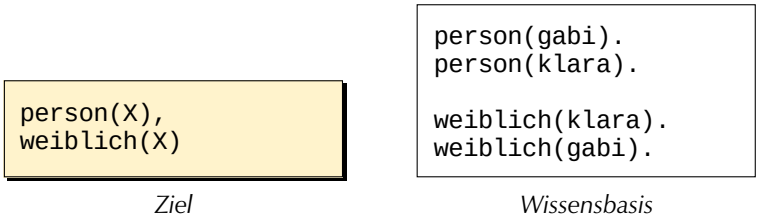
```
?- kind(gabi, X) =
   kind(Y, hans).
X = hans,
Y = gabi ?
yes
```

unifizierbar, mit Variablenbindung

Wie wird eine Anfrage bewiesen?

Nimm die Anfrage als Beweisziel (*goal*).

```
?- person(X), weiblich(X).
```

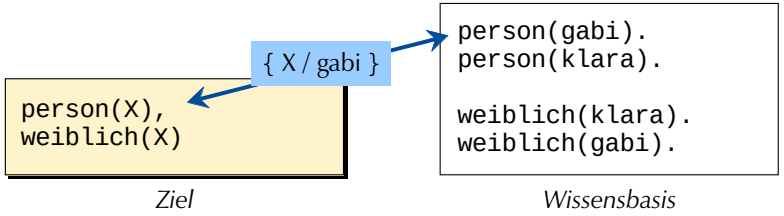


Passendes Fakt suchen

A. Suche ein Fakt, das mit dem ersten Term des Ziels unifiziert.

- Suchrichtung ist von oben nach unten!

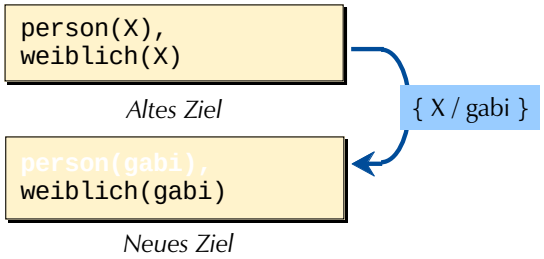
B. Merke dir die Substitution.



Beweisregel für Fakten

A. Mache die Substitution in allen Termen des Ziels.

B. Lösche den ersten Term des Ziels.

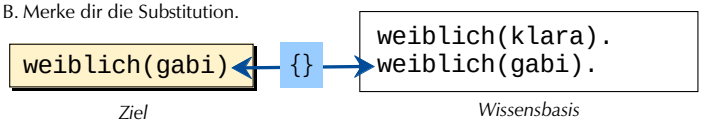


Da Capo Al Fine

Passende Klausel suchen

A. Suche ein Fakt, das mit dem ersten Term des Ziels unifiziert.

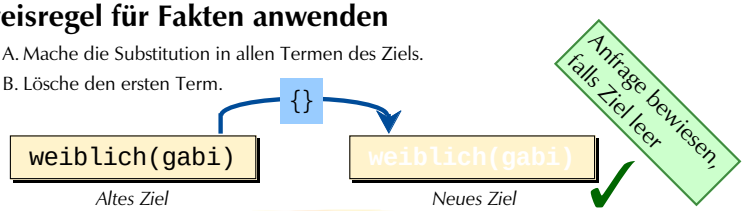
B. Merke dir die Substitution.



Beweisregel für Fakten anwenden

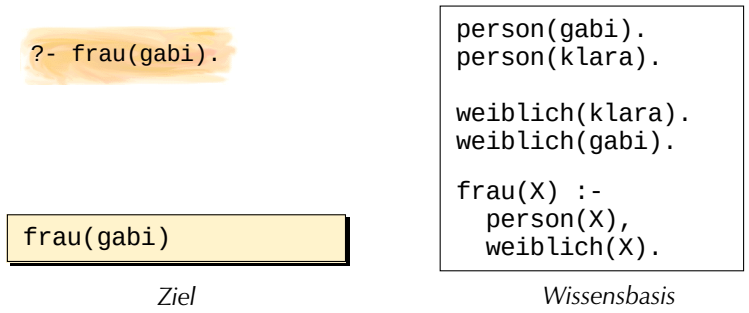
A. Mache die Substitution in allen Termen des Ziels.

B. Lösche den ersten Term.



Wie wird eine Anfrage bewiesen?

Nimm die Anfrage als Beweisziel.

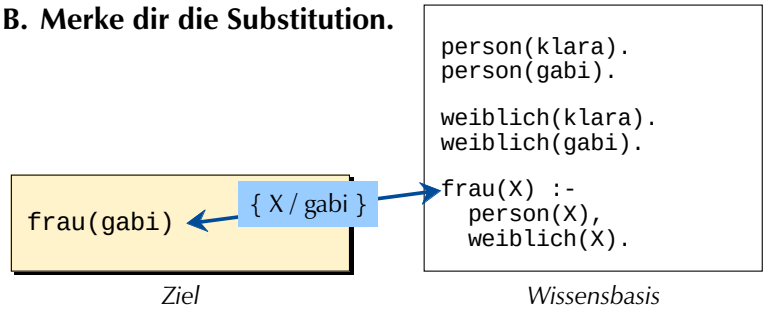


Passende Regel suchen

A. Suche einen Regelkopf, der mit dem ersten Term des Ziels unifiziert.

- Suchrichtung ist von oben nach unten!

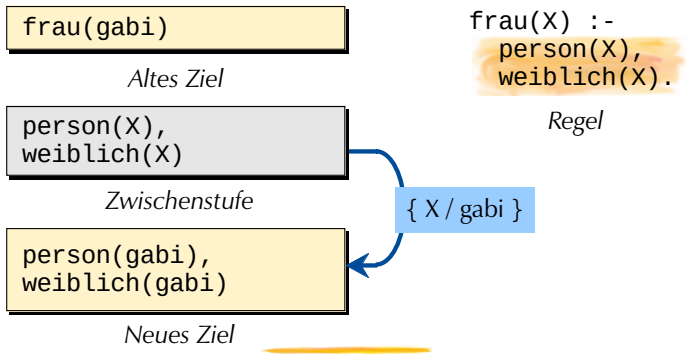
B. Merke dir die Substitution.



Beweisregel für Regeln

A. Ersetze den ersten Term des Ziels durch den Regelrumpf.

B. Mache die Substitution in allen Termen des Ziels.



Beweisen mit Fakten und Regeln

Beweisverfahren im Überblick

I. Suche von oben nach unten die erste passende Klausel für den ersten Term des Ziels.
   a. Falls Klausel ein Faktum ist, wende die Beweisregel für Fakten an.
      i. Wenn das Ziel leer ist, dann ist der Beweis gelungen. Ende
      ii. Wenn das Ziel nicht leer ist, beweise es gemäss I.
   b. Falls Klausel eine Regel ist, wende die Beweisregel für Regeln an.
      i. Beweise Ziel gemäss I.
   c. Falls keine passende Klausel mehr gefunden werden kann, gelingt der Beweis nicht.

- *ad c) Durch Backtracking werden alternative passende Klauseln aufgespürt.*

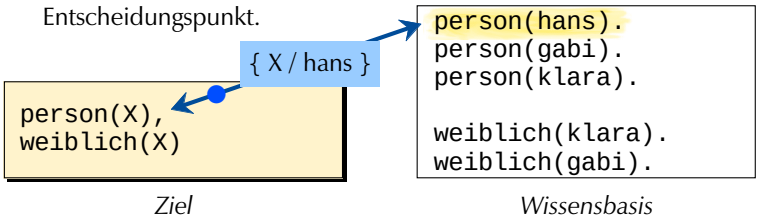
Wie wird eine Anfrage bewiesen?

Nimm die Anfrage als Beweisziel.

```
?- person(X), weiblich(X).
```

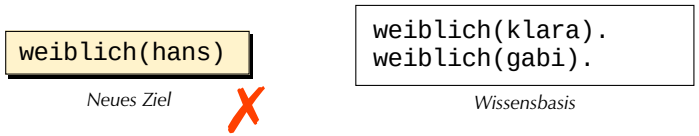
Suche passende Klausel.

- Bei mehreren passenden Klauseln merkt sich Prolog einen Entscheidungspunkt.



Sackgasse

Durch Anwenden der Beweisregel für Fakten entsteht ein neues Ziel.



Aber: Beweis in der Sackgasse

- Keine passende Klausel für neues Ziel!

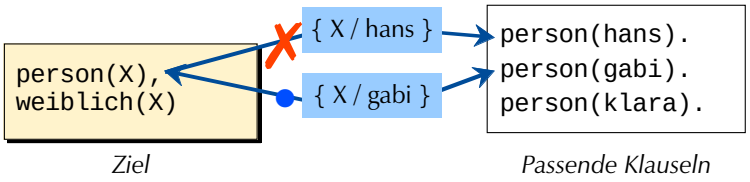
Deshalb: Backtracking (Rückverfolgen)

- Gehe zum nächsten Enscheidungspunkt mit alternativen Klauseln zurück!

Backtracking

Beweise das Ziel, wo als letztes ein Entscheidungspunkt (*decision point*) gesetzt wurde.

- Markiere bearbeiteten Entscheidungspunkt als erledigt.
- Setze einen neuen Entscheidungspunkt, falls immer noch mehrere Klauseln passen.
  - *Hinweis: Durch Backtracking werden Variablenbindungen rückgängig gemacht.*



Manuelles Backtracking

Die manuelle Eingabe des Strichpunkts am Prompt des Prolog-Interpreters löst Backtracking aus.

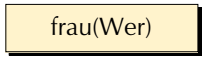
```
?- person(X), weiblich(X).
X = gabi ? ;
X = klara ? ;
no
```

- Die für jeden gelungenen Beweis erforderlichen Variablenbindungen werden vom Prolog-Interpreter jeweils herausgeschrieben.

Visualisierung als Beweis-/Suchbaum

Bestandteile und Legende

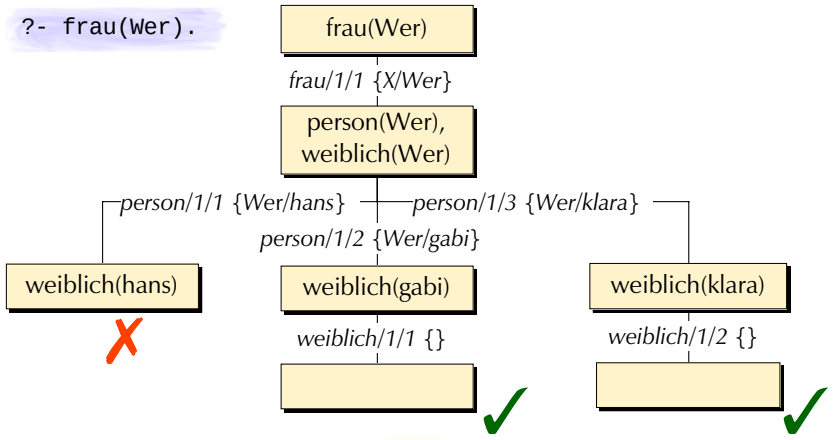
- Beweisziele
  - «Aktuelles Beweisziel: frau(Wer)»



- Beweis gefunden

- Passende Prädikatsklausel mit Substitution
  - «Beweis für ersten Term des Ziels durch 2. Klausel des einstelligen Prädikats *person*, wobei die Variable *Wer* durch *gabi* ersetzt wurde.» *person/1/2 {Wer/gabi}*
  - *Substitutionen von Variablen, die beim Beweisen mit Regeln entstanden sind, aber in Zielen nie auftauchen, können weggelassen werden.*
- Sackgasse

Visualisierung als Beweis-/Suchbaum



Occur Check

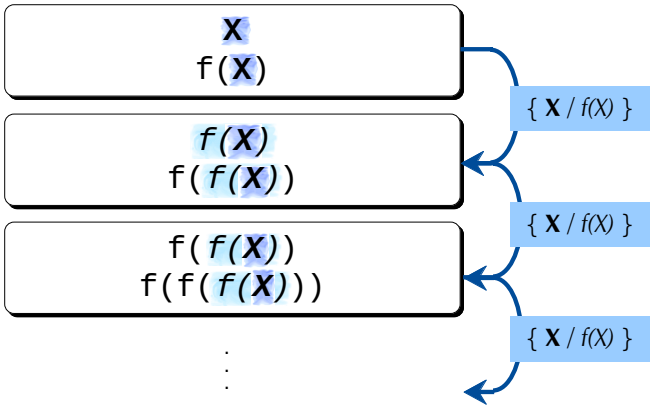
Übersicht

- Beim Unifizieren kann ein Problem entstehen…

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

- Verbot: Occurs Check
- Alternative: Zyklische Terme

Substitutionen ad infinitum…



Verbot: Occur Check

Eine Variable darf nicht an einen Term gebunden werden, in der dieselbe Variable vorkommt.

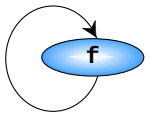
- Dieser sog. **Occur Check** (Vorkommenstest) wird aus Effizienzgründen von Prolog-Interpretern nicht gemacht.
  - Programmiernde müssen sich eigenhändig um diese (seltenen) Fälle kümmern.
- Prolog-Implementationen, die den ISO-Prolog-Standard erfüllen, bieten ein besonderes Prädikat für die Unifikation mit *Occur Check* an: **unify_with_occur_check/2**.

```
?- use_module(library(terms)). % Konsultiere Bibliothek
yes
?- unify_with_occur_check(X,f(X)).
no
```


Alternative: Zyklische Terme

Zyklische Terme

- sind in SICStus Prolog zugelassen.
- entstehen durch Unifikationen, die den *Occur Check* verletzen.
  - Das ergibt Terme, die nicht mehr als Bäume darstellbar sind!
- erfordern etwas vorsichtigen Umgang, damit man sich beim Verarbeiten dieser Strukturen nicht unendlich tief verliert.




```
?- X = f(X).
X = f(f(f(f(f(f(f(f(f(...))))))))))
yes
```

Fehlersuche mit Prolog

Übersicht

- Auflisten der Wissenbasis
- Kästchenmodell (*Byrds Box*)
  - Komplexe Ziele: Nebeneinander kleben
  - Unterziele: Ineinander schachteln
- Trace-Modus
- Debug-Modus
  - *Spy*-Punkte verwalten
- Ausnahmen (*Exceptions*)
- Fehlermeldungen

listing/0 und listing/1

Anzeigen, was Prolog beim Interpretieren eines Programmes verstanden hat.

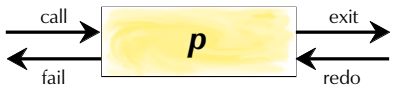
- Das Prädikat `listing` ist nützlich, wenn man wissen will, mit welcher Wissensbasis Prolog eigentlich beweist.
- Mit `listing/0` werden im Normalfall alle benutzerdefinierten Prädikate ausgegeben.
- Mit `listing/1` werden die Klauseln des Prädikat ausgegeben, das als Argument spezifiziert wurde

```
?- listing.
```

```
?- listing(q/0).
```


Kästchenmodell

```
?- p.
```



Ein einfaches Ziel *p* kann als Kästchen mit vier *Ports* (Ein- und Ausgänge) dargestellt werden.

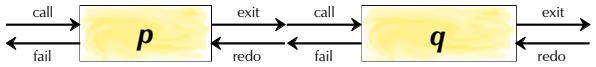
- Zwei Eingänge
  - *call* — *p* soll zum ersten Mal bewiesen werden
  - *redo* — *p* soll über Backtracking ein weiteres Mal bewiesen werden
- Zwei Ausgänge
  - *exit* — *p* konnte bewiesen werden
  - *fail* — *p* konnte nicht bewiesen werden

Kästchenmodell: Konjunktion

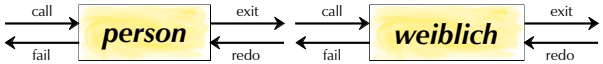
```
?- p, q.
```

Konjunktiv verknüpfte Ziele ergeben nebeneinander verhängte Kästchen.

- Der Beweis beginnt mit dem ersten *call* (ganz links).
- Die Beweis gelingt mit dem letzten *exit* (ganz rechts).
- Mittleres *exit* wird mit *call* verbunden
- Mittleres *fail* wird mit *redo* verbunden



Ein Beispiel




```
?- person(Wer), weiblich(Wer).
```

- call: person(Wer)
- exit: person(hans)
- call: weiblich(hans)
- fail: weiblich(hans)
- redo: person(hans)
- exit: person(klara)
- call: weiblich(klara)
- exit: weiblich(klara)

Wer = klara

```
person(hans).
person(klara).
person(gabi).
person(kevin).
weiblich(klara).
weiblich(gabi).
```

Wissensbasis

trace/0 und notrace/0

Prolog kann dies selbst als *Tracing* ausgeben.

- Einschalten mit `trace`; Ausschalten mit `notrace`
- Nützlich beim Suchen von Programmierfehlern
  - Leider momentan kleine Unterschiede zwischen MacOS 3.6 und Vers. 3.8.4
  - Stationen zwischen exit und nächstem Entscheidungspunkt werden unterschlagen!

```
| ?- trace.
{The debugger will first creep -- showing everything (trace)}
yes
{trace}
| ?- person(Wer), weiblich(Wer).
   1  1  Call: person(_187) ?
   1  1  Exit: person(hans) ?
   2  1  Call: weiblich(hans) ?
   2  1  Fail: weiblich(hans) ?
   1  1  Redo: person(hans) ?
   1  1  Exit: person(klara) ?
   2  1  Call: weiblich(klara) ?
   2  1  Exit: weiblich(klara) ?
```

Strichpunkt erzwingt Backtracking

```
Wer = klara ? ;
   1  1  Redo: person(klara) ?
   1  1  Exit: person(gabi) ?
   2  1  Call: weiblich(gabi) ?
   2  1  Exit: weiblich(gabi) ?
```

```
Wer = gabi ? ;
   1  1  Redo: person(gabi) ?
   1  1  Exit: person(kevin) ?
   2  1  Call: weiblich(kevin) ?
   2  1  Fail: weiblich(kevin) ?
no
```

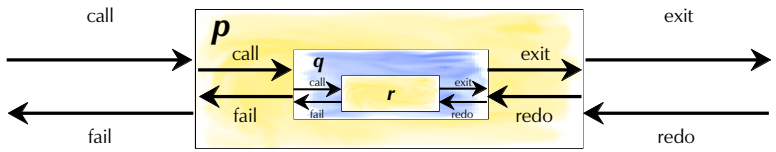

Kästchenmodell: Verschachtelung

Unterziele, die bei Regeln durch Ersetzen des Rumpfs entstehen:

- Verschachtelung der Kästchen
- Das Ursprungsziel gelingt mit dem äussersten *exit*.

```
p :- q.
q :- r.
r.
```

```
?- p.
```

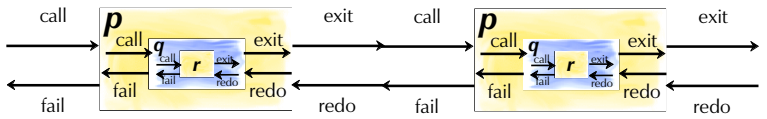


Verschachtelte Konjunktion…

Verschachteln und Hintereinanderstellen kombiniert

```
?- p, p.
```

```
?- p,p.
   1  1  Call: p ?          4  1  Call: p ?
   2  2  Call: q ?          5  2  Call: q ?
   3  3  Call: r ?          6  3  Call: r ?
   3  3  Exit: r ?          6  3  Exit: r ?
   2  2  Exit: q ?          5  2  Exit: q ?
   1  1  Exit: p ?          4  1  Exit: p ?
```



Zahlendeutung

Was bedeuten die Zahlen vor den *Ports*?

```
4  1  Call: p ?
5  2  Call: q ?
```

1. Zahl

- Bei *Call-Ports*: Anzahl durchschrittener *Call*-Eingänge, seit Beginn der Anfrage
- Bei andern Ports: Bezug auf den entsprechenden *Call-Port*
  - Nummern identifizieren Aufrufe (für Version < 3.7: Kästchen) eindeutig!

2. Zahl

- Verschachtelungstiefe beim Beweisen

Fragezeichen vor 1. Zahl bedeutet Entscheidungpunkt

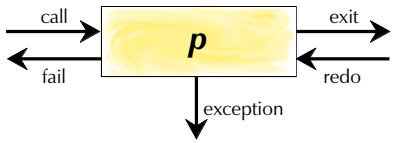
debug/0, nodebug/0, spy/1, nospy/1

Manchmal ist es mühsam, *alle* Prädikate zu *tracen*…

- Debug-Modus: Einschalten mit `debug`, Ausschalten mit `nodebug`
- Prolog zeigt zunächst nur den Trace von Prädikaten, auf die mit `spy/1` ein *spy*-Punkt gesetzt wurde (+)
  - Mit RET oder c (*creep*) kriecht man wie beim trace-Modus weiter
  - Mit l (*leap*) springt man zum nächsten Port eines Prädikats mit *spy*-Punkt
  - Mit n (*nodebug*) wird die Anfrage ohne Tracing beendet
  - Mit h (*help*) gibt's eine Menüübersicht
- Löschen eines *spy*-Punkts mit `nospy/1`

```
| ?- debug.
{The debugger will first leap -
- showing spypoints (debug)}
yes
{debug}
| ?- spy(q/0).
{Spypoint placed on user:q/0}
yes
{debug}
| ?- p, p.
 + 2  2  Call: q ?
   3  3  Call: r ?
   3  3  Exit: r ? l
 + 2  2  Exit: q ? l
 + 5  2  Call: q ? n
yes
{debug}
```


Die Ausnahme: Ein Notausgang



Notausgang für Ausnahmefälle (*exceptions*)

- Mit den 4 *Ports* muss jede Anfrage bewiesen ("yes") werden oder scheitern ("no").
- Der *Exception*-Ausgang erlaubt es, quer zum Beweisvorgang aus Kästchen herauszukommen. (Weder "no" noch "yes" am Schluss)
- *Exceptions* wandern gegen Aussen, und müssen immer durch den *Exception-Port* (ausser sie werden explizit aufgefangen)

Ausnahmefälle für Fehler

Fehlermeldung durch Ausnahmen

- Moderne Prologs, die sich am ISO-Standard ausrichten, melden Fehler durch *exceptions*.
- Es werden dabei unterschiedliche Klassen von Fehler unterschieden
  - **Existenzfehler** (*existence error*): Aufgerufenes Prädikat existiert nicht
  - **Syntaxfehler** (*syntax error*): Irgendetwas im Programmtext ist syntaktisch falsch
  - **Instantiierungsfehler** (*instantiation error*): Bei einer Anfrage war ein Argument ungenügend instantiiert
  - **Typenfehler** (*type error*): Beim Beweisen war ein Argument vom falschen Typ.
  - **Systemfehler** (*system error*): Es ist ein Systemfehler aufgetreten.
  - …

```
{EXISTENCE ERROR: t: procedure user:t/0 does not exist}
```

- Um Fehler zu beheben, muss man die Fehlermeldung verstehen!

Arithmetik mit Prolog

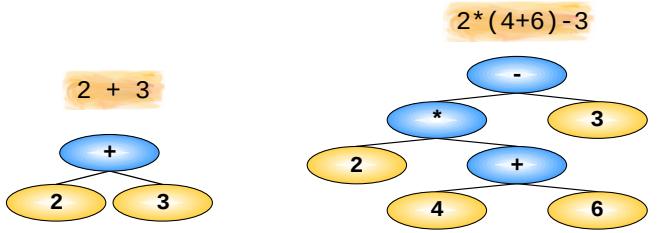
Übersicht

- Arithmetische Ausdrücke
  - Komplexe Namen für Zahlen
- Explizite Evaluation
  - `is/2`
- Arithmetische Operatoren
  - Präzedenz
  - Assoziativität
- Arithmetische Vergleichsprädikate
  - Implizite Evaluation der Argumente

Arithmetische Ausdrücke

Arithmetische Ausdrücke

- bestehen aus Zahlen und arithmetischen Funktionsnamen.
- sind gewöhnliche, komplexe Terme (in Infix-Schreibweise), die numerische Grössen bezeichnen.
- werden nicht automatisch evaluiert, d.h. als Zahlwert berechnet.



Explizite Evaluation

Das eingebaute Infix-Prädikat is/2 berechnet den Wert arithmetischer Ausdrücke.

- Normale Verwendung: Der Wert des zu berechnenden Ausdrucks wird an die Variable im 1. Argument gebunden.

1. Argument	Operator	2. Argument
Variable	*Prädikatsname*	*Arithmetischer Ausdruck*
`X`	`is`	`2-(4+6)*3`

- Es dürfen keine ungebundenen Variablen im arithmetischen Audruck vorkommen!

```
?- X is 2-(4+6)*3.
X = -28
```

```
?- X is A+2.
{INSTANTIATION ERROR: _36 is _33+2 - arg 2}
```

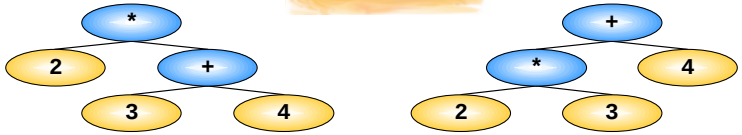

Einige Arithmetik-Operatoren

Operator	Bedeutung	Beispiel
`N + N`	Addition	15 is 10 + 5
`N - N`	Subtraktion	10 is 15 – 5
`N * N`	Multiplikation	15 is 3 * 5
`N / N`	Fliesspunkt-Division	6.5 is 13 / 2
`I // I`	Ganzzahl-Division	6 is 13 // 2
`I mod I`	Modulo (Div.-Rest)	3 is 15 mod 4
`abs(N)`	Absolutwert	3 is abs(–3)
`round(N)`	Runden	4 is round(3.7)

Präzedenz

Welche Struktur besitzt dieser Term?

2 * 3 + 4



+ hat Präzedenz vor *

* hat Präzedenz vor +

Prolog versteht die üblichen Klammerweglass-Konventionen.

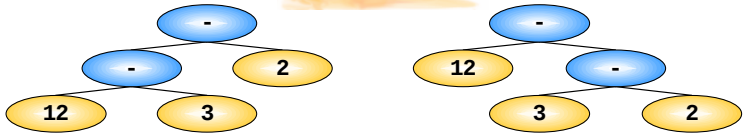
```
?- 2*3+4 = 2*(3+4).
no
```

```
?- 2*3+4 = (2*3)+4.
yes
```


Assoziativität

Welche Struktur besitzt dieser Term?

12 - 3 - 2



linksassoziativ

rechtsassoziativ

Prolog versteht die üblichen Klammerweglass-Konventionen.

```
?- 12-3-2 = (12-3)-2.
yes
```

```
?- 12-3-2 = 12-(3-2).
no
```


Arithmetische Vergleichsprädikate

Prädikate	Bedeutung	Beispiel
<	kleiner als	2 + 3 < 9 * 9
>	grösser als	170 > 5 * 5
=<	kleiner oder gleich	10 =< 17
>=	grösser oder gleich	8 + 10 >= 18
=:=	gleich	2 * 3 =:= 5 + 1
=\=	ungleich	17 =\= 10

Implizite Evaluation

Die arithmetischen Vergleichsprädikate evaluieren beim Beweisen implizit ihre Argumente.

```
zwischen(Mitte, Unten, Oben) :-
  Mitte =< Oben,
  Mitte >= Unten.
```

```
?- zwischen(1, 0+1, 3*4).
yes
```

Keines der Argumente darf im Moment des Evaluierens eine Variable sein oder enthalten!

```
?- X < 3.
{INSTANTIATION ERROR: _62<3 - arg 1}
```

```
?- X = 2, X < 3.
X = 2 ?
yes
```

Übersicht

Operatoren…

… sind Funktoren, an die nicht direkt eine Klammer folgt.

- Infix-, Präfix-, Postfix-Schreibweise
- Präzedenz, Assoziativität, Position
- Vordefinierte und selbstdefinierte Operatoren
- Operator- vs. Funktor-Argument-Schreibweise
- Klauseln sind Terme!

Infix-Schreibweise

Manchmal ist der Kode lesbarer, wenn die Funktor-Argument-Schreibweise durch Operator-Schreibweise ersetzt wird.

- Zum Bespiel Infix-Schreibweise in Arithmetik

```
-(12,-(3,2))        +(*(2,3),4)
12-3-2              2*3+4
```

- Allerdings bringen Operatoren zusätzlich das Problem der Präzedenz und Assoziativität mit sich!

Operatoren: Man nehme ...

Zu jeder Definition eines Operators gehört

- **Präzedenz** *(precedence; priority)*
  - Zahl zwischen 0 und 1200
  - Je kleiner die Zahl, desto höher die Präzedenz (!).
- **Position**
  - Präfix-Operator: Operator steht *vor* Argument.
  - Infix-Operator: Operator steht *zwischen* Argumenten.
  - Postfix-Operator: Operator steht *nach* Argument.
- **Assoziativität** *(associativity)*
  - linksassoziativ: a • b • c = (a • b) • c
  - rechtsassoziativ: a • b • c = a • (b • c)
  - nicht schachtelbar: a • b • c ist kein zulässiger Term!
- **Name** – Atom

Präzedenz und Assoziativität

f: Funktor

x: nicht-assoziativ

y: assoziativ

Angabe	Bedeutung
`fx`	Präfix, nicht schachtelbar
`fy`	Präfix, rechtsassoziativ
`xf`	Postfix, nicht schachtelbar
`yf`	Postfix, linksassoziativ
`xfx`	Infix, nicht schachtelbar
`xfy`	Infix, rechtsassoziativ
`yfx`	Infix, linksassoziativ

Vordefinierte Operatoren nach ISO-Standard

Präzedenz	Position/Assoz.	Operatoren
1200	xfx	:- -->
1200	fx	:- ?-
1100	xfy	;
1050	xfy	->
1000	xfy	,
900	fy	\+ spy
700	xfx	= \= == \== @< @=< @> @>= is =:= =\= < > >= > =..
500	yfx	+ - / \ \ /
400	yfx	* / // rem mod << >>
200	xfx	^
200	fy	\ -

Selbstdefinierte Operatoren

Selbstdefinierte Operatoren brauchen Deklaration…

```
:- op(600, xfx, vater).
```

Präzedenz
Infix, nicht assoziativ
Name

```
:- op(600, xfx, vater).
hans vater gabi.
hans vater kevin.
```

Mit current_op/3 lassen sich die gegenwärtig definierten Operatoren ausgeben.

```
?- current_op(600, A, F).
A = xfx
F = vater
yes
```


Operator vs. Funktor-Argument-Notation

Die Operator-Schreibweise ist ein syntaktischer Zucker (*syntactic sugar*).

- Für jede Operator-Schreibweise eines Terms gibt es eine Schreibweise in der Funktor-Argument-Form.
- Das eingebaute Prädikat write_canonical/1 schreibt die Funktor-Argument-Notation jedes Terms heraus:

```
?- write_canonical(12/3/2).
/(/(12,3),2)
```

- Alle Operatoren-Prädikate sind auch in der Funktor-Argument-Form aufrufbar:

```
?- =(A,a).
A = a ?
```


Regelklauseln sind Terme

Regel-Klauseln in Funktor-Argument-Notation

- Die Symbole für "falls" `:-`, "und" `,` und "oder" `;` sind vordefinierte zweistellige Operatoren.

```
?- write_canonical((p :- q, r, s)).
:-(p,','(q,','(r, s)))
yes
?- write(:-(p,','(q,','(r,s)))).
p:-q,r,s
yes
```

- Nur der Punkt am Schluss von Fakten und Regeln ist kein Term!
  - Er ist das Zeichen, das Term-Enden markiert!

Daten- und Kontrollfluss

Übersicht

- Prozedurale vs. deklarative Semantik
- Datenfluss durch Variablen und Unifikation
- Kontrollfluss
  - Abstraktion, Sequenz, Alternation
- Elimination der Disjunktion
- Spezielle Lenkung des Kontrollflusses
  - Programmiertes Scheitern: fail/0
  - Nicht-Beweisbarkeit: \+/1
  - Suchbäume stutzen: !/0
  - Aufruf: call/1

Prozedurale und deklarative Semantik

Deklarativ: Das Prädikat p ist …

- wahr, falls q und r wahr ist.
- beweisbar, falls q und r beweisbar sind.

```
p :-
   q,
   r.
```

Prozedural: Um das Ziel/die Prozedur p …

- zu erfüllen (*satisfy*), erfülle zuerst q und dann r.
- abzuarbeiten, rufe (*call*) zuerst q und dann r auf.

Prozedurale Interpretation

- hat relevante Reihenfolge!
- ist das, was Prolog-Interpreter kennt/berechnet.

Datenfluss und Modus

Durch textuelle Variablen fliessen Ein- und Rückgabe-werte zwischen Prozeduren hin und her.

- Eingabe-Wert (*Input*): Beim Aufruf instantiierte Variablen
  - Modusdeklaration: **+**
- Rückgabe-Werte (*Output*): Nach Aufruf instantiierte Variablen
  - Modusdeklaration: **–**
- Wenn sowohl Ein- wie Rückgabe möglich ist:
  - Modusdeklaration: **?**

```
% max(+Num,+Num,?Num)
max(X, Y, X) :-
   X >= Y.
max(X, Y, Y) :-
   X < Y.
```

```
?- max(2, 4, Max).
Max = 4

?- max(2, 4, 2).
no
```


Datenfluss und Unifikation

Doppelfunktion von Unifikation beim Beweisen

- **Filter**: Unifikation macht Fallunterscheidungen!
  - Nennt man auch *Pattern Matching* (Mustervergleich)
- **Konstruktor**:Unifikation macht automatischen Aufbau von Datenstrukturen!

Die Instantiierung beim Aufruf entscheidet, ob ein Argument Filter oder Konstruktor ist!

```
belegt(ida, vorlesung(ecl1)).
belegt(ida, uebung(pcl1)).
belegt(ida, vorlesung(pcl1)).
belegt(udo, vorlesung(ecl1)).
```

```
?- belegt(ida, vorlesung(X)).
X = ecl1 ;
X = pcl1 ;
no
```

```
?- belegt(X, vorlesung(ecl1)).
X = ida ;
X = udo ;
no
```

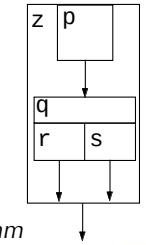
Strukturierung des Kontrollflusses

Prädikatsdefinition, konjugierte Prädikate und **mehrfache Klauseln** stellen die elementaren, prozeduralen Kontrollen zur Verfügung:

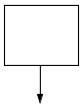
```
z :-
    p,
    q.

q :- r.
q :- s.

p.

r.

s.
```

Abstraktes Programm

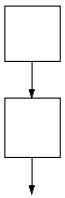
```
?- z.
```



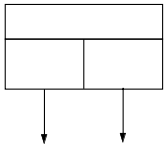
Abstraktion
(Kapselung)



Sequenz
(Abfolge)



Alternation
(Fallunterscheidung)



Disjunktion

```
grossvater(Opa, Enkel) :-
    vater(Opa, Person),
    ( mutter(Person, Enkel)
    ; vater(Person, Enkel)
    ).
```

⇔

```
grossvater(Opa, Enkel) :-
    vater(Opa, Person),
    elter(Person, Enkel).

elter(Person, Kind) :-
    mutter(Person, Kind).
elter(Person, Kind) :-
    vater(Person, Kind).
```

Elimination der Disjunktion

- Disjunktionen können immer ersetzt werden, indem die disjunktiv verknüpften Terme zu Klauseln eines neuen Prädikats werden
- Disjunktionen sind wie Klauseldefinitionen: Es findet Backtracking statt inklusive Rückgängigmachen von Variablenbindungen!

Disjunktion und Backtracking

```
person(hans).
person(klara).
person(gabi).
person(kevin).
```

Programm

```
?- person(X).
X = hans    ;
X = klara   ;
X = gabi    ;
X = kevin   ;
no
```

Anfrage

Wie gehen wir vor, um alle Lösungen eines Ziels zu erhalten?

- bereits bekannt: Manuelle Eingabe eines Semikolons
- Anstatt der manuellen Nachfrage möchte man jedoch einen **programmierbaren Mechanismus** haben.

Programmiertes Backtracking: fail/0

Das eingebaute Prädikat fail/0 kann nie erfüllt werden!

- Backtracking kann damit programmiert werden
- Wichtig für Programmiertechnik *Failure-Driven-Loop*
  - Durch Scheitern lassen sich alle möglichen Lösungen ausgeben!

```
person(hans).
person(klara).
person(gabi).
person(kevin).
alle :-
  person(X),
  write(X), nl,
  fail.
```

Programm

```
?- alle.
no
```

Anfrage

```
hans
klara
gabi
kevin
```

Ausgabe

Failure-Driven Loop

Das Fehlschlagen der Anfrage: Ein behebarer Makel

- Eine zusätzliche, bedingungslos erfüllbare Klausel, die erst dann zum Zug kommt, wenn es keine weiteren Lösungen mehr gibt.

```
person(hans).
person(klara).
person(gabi).
person(kevin).
alle :-
  person(X),
  write(X), nl,
  fail.
alle.
```

Programm

```
?- alle.
yes
```

Anfrage

```
hans
klara
gabi
kevin
```

Ausgabe

Nicht-Beweisbarkeit: \+/1

Bisher konnten wir nur fragen, ob Prolog etwas beweisen kann:

```
person(hans).
person(klara).
```

- Ist Klara eine Person?

```
?- person(klara).
```

Der Präfix-Operator \+ gelingt, falls sein Argument *nicht* bewiesen werden kann:

- Ist es nicht der Fall, dass Gerda eine Person ist?

```
?- \+ person(gundula).
yes
```

- Gibt es niemanden, der eine Person ist?

```
?- \+ person(Jemand).
no
```


Nicht-Beweisbarkeit und Negation

```
weiblich(X) :-
  \+ maennlich(X).
maennlich(hans).
```

```
?- weiblich(hans).
no
?- weiblich(gabi).
yes
```

Beachte: \+ ist keine logische Negation!

- Es gibt keine positive Information, dass Gabi weiblich ist.
- Gabi ist genauso weiblich wie Hermann nach diesem Programm.
- Je vollständiger ein Prädikat definiert ist, umso mehr nähert sich die Nicht-Beweisbarkeit der Negation an (*closed world assumption*).

```
?- weiblich(Wer).
no
```

```
?- weiblich(hermann).
yes
```


Zwecklose Alternativen

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

```
?- max(3, 2, Max).
Max = 3 ;
no
```

Was geschieht bei der Anfrage?

- Beweisversuch mit erster Klausel, der auch gelingt
- Prolog merkt sich als Entscheidungspunkt die zweite Klausel. Es weiss nicht, dass die beiden Klausel nie gleichzeitig erfüllbar sind. Es weiss nicht, dass die beiden Klauseln *deterministisch* sind!

Cut !/0: Exklusive Fallunterscheidung

```
max2(X, Y, X) :-
  X >= Y,
  !.
max2(X, Y, Y) :-
  X < Y,
  !.
```

Mit dem eingebauten Prädikat !\0 lassen sich Klauseln als exklusive Fallunterscheidungen markieren.

- *Der letzte Cut ist eigentlich unnötig!*

Wirkung des Cut

A. Wegschneiden aller alternativen Prädikats-Klauseln *unterhalb* jener, die den Cut enthält

B. Wegschneiden aller alternativen Lösungen für Ziele, die in derselben Klausel *links* vom Cut stehen.

Wirkung des Cuts: Ausgangsprogramm

Abstraktes Beispielprogramm

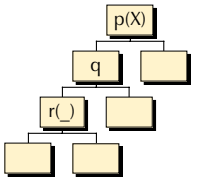
- Die Anfrage hat 4 Lösungen.

```
p(1) :- q.
p(2).

q :- r(_).
q.

r(1).
r(2).
```

Programm

```
?- p(X).
X = 1 ? ;
X = 1 ? ;
X = 1 ? ;
X = 2 ? ;
no
```

Anfrage

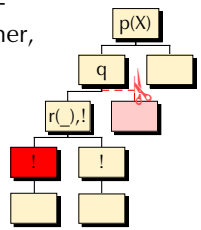


Such-/Beweisbaum

Wirkung des Cuts (A)

cut/0 gelingt immer, aber als Nebeneffekt wird Suchbaum gestutzt.

A. Wegschneiden aller alternativen Prädikats-Klauseln *unterhalb* jener, die den Cut enthält.



(A) im Such-/Beweisbaum

```
p(1) :- q.
p(2).

q :- r(_), !.
q.

r(1).
r(2).
```

(A) im Programm

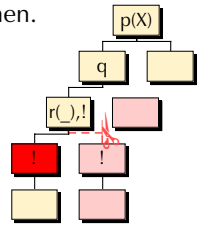
Wirkung des Cut (B)

cut/0 gelingt immer, aber als Nebeneffekt wird Suchbaum gestutzt.

B. Wegschneiden aller alternativen Lösungen für Ziele, die in derselben Klausel *links* vom Cut stehen.

```
?- p(X).
X = 1 ? ;
X = 2 ? ;
no
```

Anfrage



(B) im Such-/Beweisbaum

```
p(1) :- q.
p(2).

q :- r(_), !.
q.

r(1).
r(2).
```

(B) im Programm

Grüne vs. rote Cuts

Der Cut kann nur prozedural verstanden werden!

Grüne Cuts

- schneiden Suchäste ohne Lösungen weg.
- machen Programme effizienter (bei gleicher Lösungsmenge).
- zeigen oft Determinismus an (Kommentar `% green cut`).

Rote Cuts

- schneiden auch Suchäste mit unerwünschten Lösungen weg.
- machen Programme effizienter (bei veränderter Lösungsmenge).
- können u.a. Determinismus erzwingen.
- sind oft schlecht verständlich und heikel in der Verwendung! (Kommentar `% red cut`)

Datenstrukturen zu Aufrufen: call/1

Das eingebaute Prädikat call/1 gelingt, falls sein Argument bewiesen werden kann.

- Daten und Prozeduren sind keine strikt getrennten Welten.
- In der Anfrage "?- call(person(hans))." ist call/1 redundant.
- call/1 ist nur sinnvoll, wo das Argument variabel ist:

Zum Beispiel once/1:

Ein Prädikat, das sein Argument aufruft, aber höchstens eine Lösung erzeugt.

```
once(Goal) :-
   call(Goal),
   !.
```


Definition von \+

Unter Verwendung von call/1, !/0 und fail/0 lässt sich Nicht-Beweisbarkeit definieren.

```
:- op(900, fy, \+).
\+ C :- call(C), !, fail.

\+ C.
```

- Damit wird das Antwortverhalten bei weiblich/1 erklärbar:

```
weiblich(X) :-
  \+ maennlich(X).
maennlich(hans).
```

```
?- weiblich(Wer).
no
?- weiblich(hermann).
yes
```

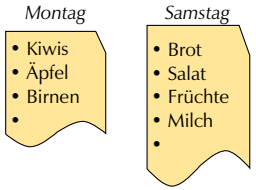
Listen

Übersicht

Listen – Die wichtigste nicht-nummerische Datenstruktur

- beliebige Länge und fixe Reihenfolge
  - Listen vs. n-stellige Terme
- Spezialnotation
  - Klammerschreibweise
  - Listenrest-Strich
- Listen-Unifikation
- Der rekursive Aufbau von Listen
  - Rekursive Datenstruktur – Geschachtelte Termstruktur
  - Die interne Punktdarstellung für Listen
- Listen als Elemente

Einkaufslisten und Wortlisten



Einkaufslisten

Yes.
I know.
Peter saw Mary.

Sätze als Wortlisten

Listen werden geführt für…

- Dinge zum Einkaufen
- Buchstaben eines Wortes
- Wörter in einem Satz
- …

☑ **Beliebige Länge**
  - ▸ Wieviele Elemente?

☑ **Strikte Reihenfolge**
  - ▸ Wie geordnet?

Problem: Erstes Wort von Sätzen

```
satz(yes)
satz(i,know)
satz(peter,saw,mary)
…
```

Datenstruktur für Sätze

```
anfang(satz(W1), W1).
anfang(satz(W1,_), W1).
anfang(satz(W1,_,_), W1).
…
```

Prädikat für Satzanfang

Ansatz: Listenelemente = Argumente komplexer Terme

- Sätze: Term der Stelligkeit *L* repräsentiert Satz der Länge *L*
- Prädikate: Satzverarbeitende Prädikate müssen soviele Fälle berücksichtigen, wie es unterschiedlich lange Sätze gibt.
  - Ein einfaches Prädikat zum Bestimmen des Satzanfangs ist mühsam zum Definieren, da jede Satzlänge eine eigene Klausel braucht!

▸ Schwierigkeit: **Fixe Stelligkeit komplexer Terme**

Gewünschte Eigenschaften

Wunschliste für Listen

- Listen nehmen in Prädikaten und Termen nur ***eine*** Argumentstelle ein!
- Listen können beliebig viele Elemente enthalten.
- Listenelemente sind geordnet.
- Listenelemente können mehrfach vorkommen.

  Beispielliste: Wörter im Satz "Wenn Fliegen hinter Fliegen fliegen, fliegen Fliegen Fliegen nach."
- Listen können auch keine Elemente enthalten.
  - Sie können leer sein (sog. *leere Liste*)

Trick I: Klammerschreibweise

Für Listen gibt es eine eigene Schreibweise

- Elemente sind zwischen eckigen Klammern eingeschlossen
- Elemente sind durch Kommata getrennt
- Elemente sind beliebige Terme

```
[kiwi]
```
Ein Element

```
[]
```
Kein Element
(leere Liste)

```
[apfel,Frucht]
```
Zwei Elemente
(ein Atom und eine Variable)

Die halbe Lösung des Problems

```
satz([yes])
satz([i,know])
satz([peter,saw,mary])
…
```
Datenstruktur für Sätze

```
anfang(satz([W1]), W1).
anfang(satz([W1,_]), W1).
anfang(satz([W1,_,_]), W1).
…
```
Prädikat für Satzanfang

Ansatz: Sätze als Wortlisten

- Sätze: Liste mit *L* Elementen repräsentiert Satz der Länge *L*.
  - *Dank Klammerschreibweise brauchen die Wörter von Sätzen nur noch eine Argumentstelle. Der Funktor satz/1 reicht aus!*
- Prädikate: Satzverarbeitende Prädikate müssen weiterhin Sätze unterschiedlicher Länge als Fälle unterscheiden.
  - *Definition für Satzanfangsprädikat bleibt mühsam, da jede Satzlänge weiterhin eine eigene Klausel braucht!*
- ▶ Schwierigkeit: **Fixe Länge von Listen**

Trick II: Der Listenrest-Strich

Der Listenrest-Strich dient dazu, Listen *beliebiger* Länge zu bezeichnen.

- Vor dem Strich steht mindestens ein Anfangselement.
- Nach dem Strich steht der Listenrest.

Der Listenrest

- ist normalerweise eine Variable, die instantiiert werden kann!
- ist selbst eine Liste, die Restliste!

```
[kiwi,apfel|Fruechte]
```
Mindestens zwei Elemente
(zwei Atome und eine Variable als Listenrest)

Die ganze Lösung des Problems

```
satz([yes])
satz([i,know])
satz([peter,saw,mary])
…
```
Datenstruktur für Sätze

```
anfang(satz([W1|_]), W1).
```
Prädikat für Satzanfang

Listenrest-Strich und Klammerschreibweise lösen das Problem des Satzanfangs mit *einer* einzigen Klausel!

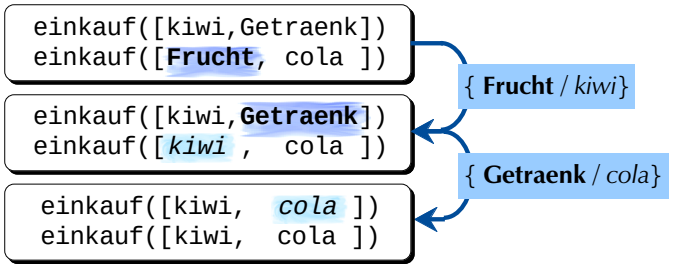
- *Pattern Matching* mit Listenrest-Strich extrahiert aus Sätzen verschiedenster Länge das erste Element!

```
?- anfang(satz([yes]), Anfang).
Anfang = yes
?- anfang(satz([mary,saw,peter]), Anfang).
Anfang = mary
```

Unifikation von Listen

Zwei Listen sind unifizierbar, falls

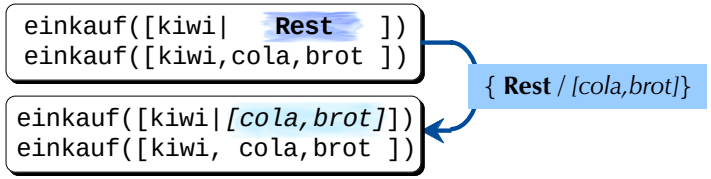
- die einzelnen Elemente paarweise unifizierbar sind
- die Länge beider Listen übereinstimmt



Unifikation von Listen

Eine Liste mit variablem Listenrest und eine Liste ohne eine Listenrest sind unifizierbar, falls

- die Anfangselemente paarweise unifizierbar sind
- der Listenrest mit der Liste der restlichen Elemente unifiziert wird



Listenrest-Strich und Unifikation

Mit | ist der Rest einer Liste erreichbar.

- Extrem wichtig für *Pattern Matching*!

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

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

```
?- [Anfang | Rest] = [a, b | [c, d]].
Anfang = a, Rest = [b, c, d]
```


Erklärung

Wir wissen:

- Prolog-Programme bestehen aus Termen.
- Listen können in Prolog wie Terme verwendet werden.

Aber:

- Die Klammerschreibweise mit [und] entspricht **keiner** Term-Notation!

Intern sind Listen durch rekursiv geschachtelte Terme aufgebaut.

- Die Klammerschreibweise ist eine Kurznotation.

Listen als rekursive Datenstruktur

Rekursive Definition von Listen

Rekursionsfundament

- Die leere Liste bildet eine Liste.

Rekursionsschritt

- Die Verknüpfung eines Elements mit einer Liste bildet wiederum eine Liste.

Am einfachsten konstruktiv betrachtet…

Um eine neue Liste zu bauen, nimm ein Element und hänge eine beliebige vorhandene Liste (leer oder nicht leer) daran.

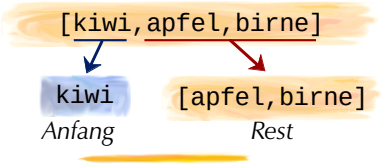
Listen in Prolog

Rekursionsfundament

- Die leere Liste ist das Atom `[]`.

Rekursionsschritt

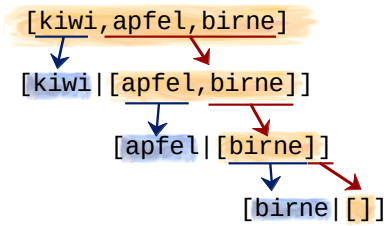
- Jede nicht-leere Liste besteht aus der Verknüpfung
  - eines beliebigen Terms als Anfang (Kopf, *Head)*
  - und einer Liste als Rest (Schwanz, *Tail).*
    - Die Liste kann leer oder nicht leer sein!



Rekursiv geschachtelte Struktur

Listenreststrich als Verknüpfung

- Der rekursive Aufbau wird durch die Notation mit Listenreststrich sichtbar.
- Klammerschreibweise ist eine verflachte Notation.

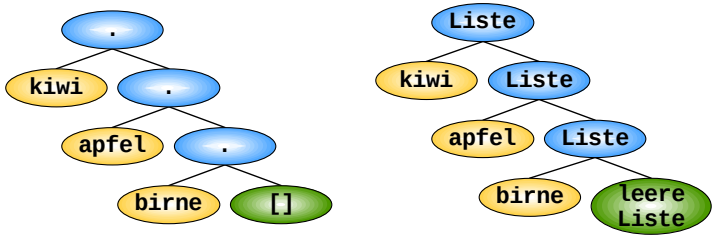



```
?- [kiwi,apfel,birne] = [kiwi|[apfel|[birne|[]]]].
yes
```


Interne Repräsentation

Rekursive Struktur heisst geschachtelte Terme

- Intern repräsentiert Prolog die Verknüpfung von Kopf und Restliste durch den Funktor '.'/2



Interne Term-Struktur *Logische Struktur*

Listen als normale Terme

Listen sind intern ganz normale Terme

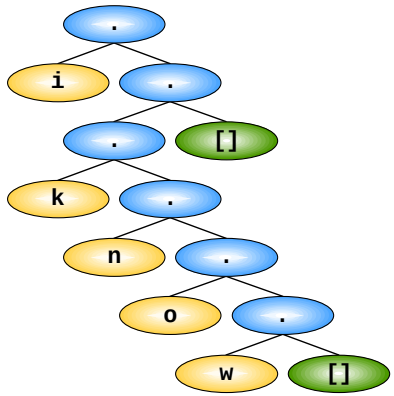
```
?- write_canonical([kiwi,apfel,birne]).
'.'(kiwi,'.'(apfel,'.'(birne,[])))
```

- Diese Punkt-Notation (*Dot-Notation*) mit dem Funktor ./2 wäre aber zu unübersichtlich, um brauchbar zu sein!
- Das 2. Argument des Punkt-Funktors ist die Liste hinter dem Listenrest-Strich.

```
?- '.'(kiwi,Fruechte) = L.
L = [kiwi|Fruechte]
```


Listen als Elemente

Wenn Listen normale Terme sind, können sie auch Elemente von Listen sein…




```
[i,[k,n,o,w]]
```


Das Listenprädikat

Datenstruktur Liste als Prädikat?

- Die Datenstruktur Liste tritt – wie alle Datenstrukturen – nur als Argument von Prädikaten auf!
- Als Prädikat verwendet bedeutet die Listenschreibweise dasselbe wie consult/1.
  - Die listenförmige consult/1 wird meist in grösseren Programmen verwendet, um Programmteile hineinzuladen, die in anderen Dateien vorhanden sind,

```
:- [datei1].
:- [datei1,datei2].
```

```
:- consult(datei1).
:- consult([datei1,datei2]).
```

Rekursive Listenverarbeitung

Übersicht

Rekursion ist die wichtigste Programmiertechnik in Prolog!

- Rekursive Datenstrukturen
  - Einfache und rekursiv gebildete Strukturen
- Rekursive Datenstrukturen und rekursive Prädikate
  - Eine natürliche Kombination…
- Aufbau rekursiver Prädikate
  - Abbruchbedingung
  - Rekursionsschritt
- Rekursive Programmiertechniken mit Listen
  - Länge von Listen: laenge/2
  - Suche nach Elementen: member/2
  - Abbilden von Listen: papagei/2

Listen: Rekursive Datenstrukturen

Bei Listen – wie bei allen rekursiven Datenstrukturen – gibt es 2 Arten von Definitionsregeln.

I. Regeln für **einfache Strukturen**

- *Die leere Liste ist eine Liste.* `[]`
- *Atomaren Terme sind Prolog-Terme. Variablen sind Prolog-Terme.*

II. Regeln für **rekursiv aufgebaute, komplexe Strukturen**

- *Nicht-leere Listen bestehen aus einem Element und einer Liste als Rest.*

`[e|Liste]`

- *Komplexe Terme sind Prolog-Terme, die aus einem Funktor und dessen Argumenten bestehen, die Prolog-Terme sind.*

Problem: Was sind wahre Listen?

`[a,b,c|[d]]`

`[a,b,c|d]`

Listen sind eine bestimmte Sorte von Termen in Prolog

- Definiere ein Prädikate is_list/1, das genau dann wahr ist, wenn das Argument eine Liste ist!

Problem

- Da Listen beliebig viele Elemente enthalten können, müssen beliebig viele Klauseln für is_list/1 geschrieben werden!

```
is_list([]).
is_list([_,_]).
is_list([_,_,_]).
...
is_list([_,_,…,_]).
```


Rekursive Lösung: Wahre Listen

Rekursive Datenstrukturen + Rekursive Prädikate

- Passe die Strategie des Problemlösens der Struktur des Problems an!

Also

- Prädikatsklausel für **einfache Struktur: Leere Liste**

```
is_list([]).
```

"Die leere Liste ist eine Liste."

- Prädikatsklausel für **rekursiven Strukturen: Nicht-Leere Liste**

```
is_list([E|Rest]) :-
  is_list(Rest).
```

"Nicht-leere Listen bestehen aus einem Element und einer Liste als Rest."

Rekursive Dekomposition

Rekursive Datenstrukturen

a. enthalten Teilstrukturen, die mit denselben Definitionsregeln aufgebaut wurden.

Teilstrukturen sind immer weniger komplex als die sie enthaltenden Strukturen.

b. haben **elementare** Strukturen und **rekursive** Strukturen.

Rekursive Prädikate

a. lösen ein Problem, indem sie es auf Teilprobleme gleicher Art reduzieren, die deshalb mit dem gleichen Prädikat erschlagen werden können.

Teilprobleme sind weniger komplex als das Problem, dessen Teil sie sind.

b. haben Klauseln für elementare oder abschliessende Fälle (**Abbruchbedingung**) und rekursive Fälle (**Rekursionsschritt**).

Der Bau rekursiver Prädikate

Für den Aufbau und das Schreiben rekursive Prädikate kann oft ein gemeinsames Schema verwendet werden.

- **Zuerst**: Klauseln für **Abbruchbedingung**
  - Terminiere den Beweis, falls die Abbruchbedingung erfüllt ist.
  - Oft einfach zu finden und zu programmieren!
- **Danach**: Klauseln für **Rekursionsschritte**
  - Löse das Problem für einen einzelnen Schritt und wende auf das Restproblem dasselbe Prädikat rekursiv an.
  - Oft erstaunlich einfache Definition, aber schwierig zu finden!

Gefahr: Rekursive Prädikate ohne Abbruchbedingungen verhalten sich wie zirkulär definierte Prädikate!

```
huhn :- ei.
ei :- huhn.
```


Beispiel I: Länge von Listen

Schreibe ein rekursives Prädikat laenge/2, das die Länge einer Liste bestimmt.

- Erstes Argument (*Input*): Liste, deren Länge zu bestimmen ist
- Zweites Argument (*Output*): Die Länge (d.h. Anzahl der Elemente)

```
?- laenge([], X).
X = 0
yes
```

```
?- laenge([a,b,c], X).
X = 3
yes
```

Wie immer bei rekursiven Prädikaten unterscheiden wir

- Abbruchbedingung
- Rekursionsschritt

Definition: Länge von Listen

Abbruchbedingung

- Die Länge der leeren Liste ist 0.

```
laenge([], 0).
```

Rekursionsschritt

- Die Länge einer nicht-leeren Liste ist die Länge ihres Rests plus 1.

```
laenge([_|Rest], Ergebnis) :-
  laenge(Rest, RestLaenge),
  Ergebnis is RestLaenge + 1.
```

Hinweis

In SICStus Prolog gibt es ein eingebautes Prädikat length/2, dass die Länge einer Liste berechnen kann wie unser Prädikat laenge/2. Zusätzlich kann es aber noch verwendet werden, um Listen bestimmter Länge zu generieren.

Beispiel II: Suche nach einem Element

Schreibe ein Prädikat member/2, das wahr ist, falls ein Term Element einer Liste ist.

```
?- member(sittich, [spitz,dackel,terrier]).
no
```

```
?- member(dackel, [spitz,dackel,terrier]).
yes
```

```
?- member(X, [spitz,dackel,terrier]).
X = spitz ;
X = dackel ;
X = terrier ;
no
```


Suche: Rekursive Dekomposition…

Das gesuchte Element ist das erste Element der Liste.

- **Abbruchbedingung:** Das vorderste Element ist mit dem gesuchten Term unifizierbar.

```
member(dackel, [dackel,terrier])
```

Das gesuchte Element befindet sich vielleicht im Rest der Liste.

- **Rekursionsschritt:** Suche im Listenrest weiter (d.h. ohne das Anfangs-Element)

```
member(dackel, [mops,dackel,terrier])
```


Abbruchbedingung: Gefunden!

```
member(dackel, [dackel,terrier])
```

Die Klausel für die erfolgreiche Suche…

- *X* ist Element der Liste, wenn *X* das erste Element in der Liste ist.

```
member(X, Liste) :-
  Liste = [X|IrgendeinRest].
```

- Einfacher geschrieben:

```
member(X, [X|IrgendeinRest]).
```

- Eigentlich interessiert uns der Rest gar nicht:

```
member(X, [X|_]).
```


Rekursiver Fall: Weitersuchen!

```
member(dackel, [mops,dackel,terrier])
```

Das Element könnte noch im Rest enthalten sein, d.h. im Rest weitersuchen!

- Nimm den Rest der Liste und prüfe, ob *X* im Rest enthalten ist.

```
member(X, [Anfang|Rest]) :-
  member(X, Rest).
```

- Eigentlich interessiert uns der Anfang gar nicht.

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

Rekursive Suche: member/2

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

Deklarative Verdeutschung

- Ein Term ist Element einer Liste, falls der Term Kopf der Liste ist.
- Ein Term ist Element einer Liste, falls der Term Element des Rests der Liste ist.

Listen: Analyse und Konstruktion

Listen-Analyse

- In rekursiven Listenprädikaten wird meist eine Eingabe-Liste rekursiv auseinandergenommen (analysiert)

Listen-Konstruktion

- In rekursiven Listenprädikaten wird oft zugleich eine Ausgabe-Liste rekursiv aufgebaut (konstruiert), die das gewünschte Resultat enthält

Für Dekonstruktion wie Konstruktion wird Unifikation verwendet!

- *Pattern Matching spielt* oft *Doppelrolle der analytischen Fallunterscheidung und Resultatskonstruktion!*

Beispiel III: Abbilden *(Mapping)*

Listenmapping mit Prädikat `papagei/2` ist Beispiel für gleichzeitige Listen-Analyse und -Konstruktion.

- **Eingabe**: Liste
- **Ausgabe**: Eingabeliste, in der bestimmte Element ersetzt sind

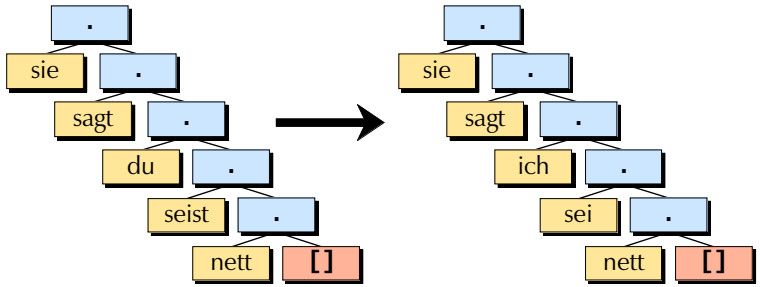
```
?- papagei([du,bist,nett], Echo).
Echo = [ich,bin,nett]

?- papagei([sie,sagt,du,seist,nett], Echo).
Echo = [sie,sagt,ich,sei,nett]
```

Im Beispiel ist die Ausgabe gleich der Eingabe, ausser für die Listenelemente *du, ich, bist, bin, sei, seist*.

Abbilden der Liste

Vorgehen: Analyse und Konstruktion

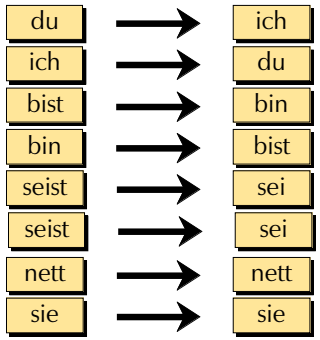
- Gehe schrittweise durch die Liste (»traversieren«)
- Bilde jedes einzelne Element auf das entsprechende Ergebnis ab.



Abbilden der Elemente: map/2

Zum Austauschen der einzelnen Listenelemente definieren wir das Hilfsprädikat map/2.

```
map(du, ich).
map(ich, du).
map(bist, bin).
map(bin, bist).
map(seist, sei).
map(sei, seist).

map(X, X).
```



Zuerst Abbruchbedingung…

Das Prädikat papagei/2 bildet eine Liste in eine andere ab.

Abbruchbedingung

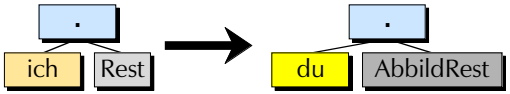
- Bilde die leere Liste auf die leere Liste ab.

  Da kein Element vorhanden ist, muss keines ausgetauscht werden.

```
papagei([], []).
```

- Die Abbruchbedingung steht normalerweise *vor* dem rekursiven Fall.

… dann Rekursionsschritt



Rekursionsschritt

- Nimm Anfangselement und bilde es mit map/2 ab.
- Rufe papagei/2 rekursiv auf, um den Rest der Liste abzubilden
- Konstruiere die Resultatsliste, die besteht aus
  - einem Anfangs-Element: der Abbildung des ersten Elements
  - einer Restliste: der Abbildung des Rests

```
papagei([E|Rest], [AbbE|AbbRest]) :-
  map(E, AbbE),
  papagei(Rest, AbbRest).
```


Abbilden im Überblick: papagei/2

```
% Abbildung bestimmter Terme.
map(du, ich).
map(bist, bin).
map(ich, du).
map(bin, bist).
map(seist, sei).
map(sei, seist).

% Wenn es keiner der obigen Terme ist, ist
% das Abbild gleich dem Original.
map(X, X).

% Die Abbildung einer leeren Liste ergibt
% eine leere Liste.
papagei([], []).

% Die Abbildung eines Terms, gefolgt von einer Rest-Liste
% ist die Abbildung des Terms, gefolgt von der Abbildung
% der Rest-Liste.
papagei([E|Rest], [AbbE|AbbRest]) :-
  map(E, AbbE),
  papagei(Rest, AbbRest).
```

Ein- und Ausgabe

Übersicht

- Wie werden Schriftzeichen kodiert?
- Ein- und Ausgabe von ASCII-Codes
  - Eingabe: get/1, get0/1
  - Ausgabe: put/1, nl/0, tab/1
  - Konvertierung: name/2, atom_codes/2, number_codes/2
  - Zeichenketten
- Ein- und Ausgabe von Termen
  - write/1, read/1, write_canonical/1
- Umlenken von Ein- und Ausgabe in Dateien
  - tell/1, telling/1, told/0
  - see/1, seeing/1, seen/0
- Dateienden und -Verarbeitung

Buchstaben als Zahlen: Kodierung

Buchstaben können als Zahlen angesehen werden.

- Eine *Kodierung* legt fest, welcher Buchstabe mit welcher Zahl gemeint ist.

Willkürliche Zuordnung 1	*Willkürliche Zuordnung 2*
1 = A	65 = A
2 = B	66 = B
…	

- Die Zahlen selbst wurden 'traditionell' durch 8-stellige 0/1-Folgen (*byte*) dargestellt. D.H. 256 mögliche unterschiedliche Werte

Willkürliche Zuordnung 1	*Willkürliche Zuordnung 2*
00000001 = A	01000001 = A
00000010 = B	01000010 = B
…	

Kodierungsstandards

Verschiedene Standards/Konventionen

- ***A**merican **S**tandard **C**ode for **I**nformation **I**nterchange* (ASCII)
  - anderer Name: *International Alphabet* 5 (IA5)
  - regelt Codes fürs englische Alphabet (A – Z; a – z; 0 – 9) und einige Sonderzeichen wie @, {, /, (, %, [.
- **ISO 8859-1** erweitert ASCII um Codes für die Schriftzeichen der meisten westeuropäischen Sprachen, z.B. ä, ß, É, Ò
  - UNIX- und WIN-Systeme verwenden oft ISO-8859-1 (ANSI). MacOS und DOS nicht.
- Dutzende andere Konventionen

Probleme

- Manche Konventionen widersprechen sich (MacOS vs. WIN)
- Nur ASCII ist wirklich weit verbreitet, umfasst aber wenig Zeichen

ASCII-Codetabelle (Zeichensatz)

Ausschnitt aus den 128 Zeichen der ASCII-Tabelle

10/13	*neue Zeile*	47	/	63	?	79	O	95		111	o
32	*Leerschlag*	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~

Ein Ansatz zur Vereinheitlichung

Unicode Version 3.0.1 (Dezember 2000)

- Prinzip: Eine eindeutige Zahl für jedes Zeichen!
- Codes für alle gegenwärtig verwendeten Schriftzeichen (*glyphs*) und Symbole in (fast) allen Sprachen der Welt (49'194 Einträge)
- Codes für Zeichen einiger ausgestorbene Sprachen
- UTF-16 Kodierung mit 16-stelligen 0/1-Folgen (Zahlen von 0 bis 65535. D.H. maximal 65536 verschiedene Zeichen)
  - Konform zur ISO/IEC-Normierung 10646
  - Heute: Unterstützung durch Java, Windows NT, MacOS 8, …
  - Nahe Zukunft: WWW-Dokumente in Unicode (ab HTML 4)
  - UTF-32 Kodierung erlaubt sogar 32-stellige 0/1-Folgen
- Codetabellen und Infos unter `http://www.unicode.org`

Ausschnitte Unicode-Codetabellen

iso-8859-1

+	0	1	2	3	4	5	6	7	8	9
160		¡	¢	£	¤	¥	¦	§	¨	©
170	ª	«	¬	-	®	¯	°	±	²	³
180	´	µ	¶	·	¸	¹	º	»	¼	½
190	¾	¿	À	Á	Â	Ã	Ä	Å	Æ	Ç
200	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ
210	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û
220	Ü	Ý	Þ	ß	à	á	â	ã	ä	å
230	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù
250	ú	û	ü	ý	þ	ÿ				

Unicode enthält ISO-Latin 8859-1 zwischen 0 und 255

Exotischer Code

Zeichen ausgeben: put/1

- **put/1** gibt ein einzelnes Zeichen aus. Das Argument ist der ASCII-Code des Zeichens.

```
?- put(72), put(97), put(108), put(108), put(111).
Hallo
```

Allerdings definiert ASCII keine Codes für die Zeichen exotischer Sprachen.

- Deutsch ist wegen Ä, Ö, Ü, ß etc. eine exotische Sprache
- Ergebnis von put(138) oder put(5000): nicht normiert!

Druckbare Zeichen einlesen: get/1

- **get/1** wartet, bis der Benutzer ein einzelnes druckbares Zeichen auf der Tastatur eingibt.
  - Danach unifiziert das Argument von get/1 mit dem ASCII-Code des eingegebenen Zeichens.

```
?- get(X).
|: A
X = 65 ?
yes
```

Benutzer hat A eingegeben

63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H

Als **nicht-druckbar** gelten die ASCII-Zeichen von 0-32 und 127. Darunter sind Zeilenende (10/13), Tabulator (9) und Leerzeichen (32).

Beliebige Zeichen einlesen: get0/1

- **get0/1** wartet, bis der Benutzer ein *beliebiges* Zeichen auf der Tastatur eingibt. (Eingabe durch Return!)
  - Danach unifiziert das Argument von get0/1 mit dem ASCII-Code des eingegebenen Zeichens.

```
?- get0(32).
yes
```

Benutzer hat Leerschlag (*space*) eingegeben!

32	*Leerschlag*
33	!
34	"
35	#

Benutzer hat anderes Zeichen als Leerschlag eingegeben!

```
?- get0(32).
no
```


Zeilenende nl/0 und Leerzeichen tab/1

- **Zeilenenden** werden auf unterschiedlichen Betriebssystemen durch unterschiedliche ASCII-Codes repräsentiert.

Betriebssystem	ASCII
UNIX	10
MacOS	13
DOS/WIN	10 + 13

10 = Zeilenvorschub (*linefeed, LF*)
13 = Wagenrücklauf (*carriage return, CR*)

  - **nl/0** gibt betriebsystemabhängig die richtigen ASCII-Codes für Zeilenende aus!
- **Mehrere Leerzeichen** werden gerne mit **tab/1** ausgegeben:
  - Das Argument gibt die Anzahl der auszugebenden Leerzeichen an.

```
?- tab(1+2), put(33).
   !
yes
```


Wie beweist Prolog Ein- und Ausgabe?

Eingabeprädikate sind bewiesen, wenn eine entsprechende Eingabe erfolgt ist.

- *Interaktive Eingabe "blockiert" den Prolog-Interpreter bis Input erfolgt.*
  - Bei Backtracking werden allfällige Variablenbindungen rückgängig gemacht, aber es erfolgt keine weitere Eingabeaufforderung!

Ausgabeprädikate sind bewiesen, wenn eine entsprechende Ausgabe als Seiteneffekt erfolgt ist.

- *Die Ausgabe selbst hat auf den Beweis keinen Einfluss.*
  - Bei Backtracking bleibt der Seiteneffekt (die Ausgabe) bestehen, kein Backtracking!

Ein- und Ausgabe gelingen höchstens 1-Mal!

ASCII-Codes und atomare Terme

- **Das eingebaute Prädikat name/2**
  - gibt den Namen eines nicht-variablen atomaren Terms als Liste von ASCII-Codes zurück

```
?- name(bla, L).
L = [98,108,97]
```

```
?- name(27, L).
L = [50,55]
```

Modus: name(+Atomar, ?Liste)

  - oder erzeugt umgekehrt einen atomaren Term aus einer Liste von ASCII-Codes

```
?- name(A, [98,108,97]).
A = bla
```

Modus: name(?Atomar, +Liste)

Das Problem mit name/2…

Zahl oder Atom?

- Falls die ASCII-Code-Liste eine Prolog-Zahl beschreibt, wird sie **immer nur** als Zahl instantiiert.

```
?- name(N, [50,55]).
N = 27 ;
no
```

```
?- name('27', [50,55]).
yes
```

Anstelle von name/2 sollten die konsistenten ISO-Prolog-Prädikate **atom_codes/2** und **number_codes/2** verwendet werden, die eine ASCII-Liste konsequent in Atome oder Zahlen umsetzen.

- Leider verwendet SICStus Prolog in älteren Versionen anstelle von atom_codes/2 atom_chars/2 und anstelle von number_codes/2 number_chars/2 …

ASCII-Codes als Zeichenketten

Eine beliebige Zeichenkette (*string*), die zwischen zwei " (doppelte Hochkommata) eingeschlossen ist, wird als Liste der ASCII-Codes dargestellt.

```
?- Kette = "Hallo Du".
Kette = [72,97,108,108,111,32,68,117]
```

32	Leerschlag
68	D
72	H
97	a

108	l
111	o
117	u

Ein-/Ausgabe von Prolog-Termen

Prolog hat vordefinierte Ein-/Ausgabe-Prädikate für Prolog-Terme

- Vorteil: Komplexe Ausdrücke müssen nicht als Einzelzeichen eingelesen und mühsam zusammengesetzt werden
- Nachteil: Die Prolog-Term-Syntax muss beachtet werden
  - *Jeder Term muss bei der Eingabe mit einem Punkt beendet werden!*

Interaktion kann mit der Aussenwelt nach einem einfachen Muster erfolgen.

```
interaktion :-
    read(Input),
    verarbeite(Input, Output),
    write(Output).
```


Terme einlesen: read/1

- **read/1** liest einen Term ein.

```
?- read(Eingabe), write(Eingabe).
|: bla.
bla
Eingabe = bla ? ;
no
```

Eingabe des Benutzers

Ausgabe von Prolog

- **Beachte:** Der Punkt beendet den Term, gehört aber selbst nicht dazu!

```
?- read(Eingabe), write(Eingabe).
|:bla bla.
{SYNTAX ERROR...
```

- **Beachte:** Die Syntaxregeln für Prolog-Terme müssen beachtet werden!

Termausgabe: write/1, write_canonical/1

- **write/1** gibt einen einzelnen Term aus.

```
?- write(1 + 2 =:= 3 -0).
1+2=:=3-0
?- write([bla,bli,blu]).
[bla,bli,blu]
```

  - write/1 respektiert die beim Aufruf definierten Operatoren und Spezialsyntax.

- **write_canonical/1** ignoriert Spezialsyntax und Operatoren

```
?- write_canonical([bla,bli,blu] = 2).
=('.'(bla,'.'(bli,'.'(blu,[]))),2)
```


Schreiben in eine Datei

- **tell/1 leitet die Ausgabe der Prädikate**
  - put/1
  - nl/0, tab/1
  - write/1, write_canonical/1

  in eine Datei um.

```
?- tell(hans),
   write(hallo), nl,
   write(du), nl,
   told.
```

- **told/0**
  - beendet die Umleitung (schliesst die Datei!)
  - sorgt dafür, dass zukünftige Ausgaben wieder auf dem Bildschirm erscheinen.

Schreiben in eine Datei

- **telling/1** gibt an, in welche Datei die Ausgabe zur Zeit gerade geleitet wird.
  - `user` steht für den Bildschirm, der als abstrakte Datei betrachtet wird.

```
?- telling(Zuerst),
   tell(hans), write(hallo),
   telling(Mitte),
   told,
   telling(Zuletzt).
Zuerst = user,
Mitte = hans,
Zuletzt = user
```


Lesen aus einer Datei

- **see/1 nimmt die Eingabe für die Prädikate**
  - get/1, get0/1
  - read/1

  aus einer Datei. (öffnet die Datei!)

- **seen/0**
  - beendet die Umleitung (schliesst die Datei!)
  - sorgt dafür, dass zukünftige Eingaben wieder vom Benutzer abgefragt werden.

```
?- see(hans),
   get0(_),
   get0(Y), put(Y), nl,
   seen.
a
```

Lesen aus einer Datei

- **seeing/1** gibt an, aus welcher Datei die Eingabe zur Zeit gerade genommen wird.
  - `user` steht für die Tastatur, die als abstrakte Datei betrachtet wird.

```
?- seeing(Zuerst),
   see(hans),
   seeing(Mitte),
   seen,
   seeing(Zuletzt).
Zuerst = user,
Mitte = hans,
Zuletzt = user
```


Das Dateiende

- **Wie kann beim Einlesen das Erreichen des Dateiendes erkannt werden?**

  Dateiende wird als spezielles Element repräsentiert:
  - **Eingabe mit ASCII-Kodes**
    - get/1, get0/1 liefern die Zahl -1 zurück.
  - **Eingabe mit Termen**
    - read/1 liefert das Atom end_of_file zurück.

- **Nachfrage: Wie wird beim Ausgeben das Dateiende herausgeschrieben?**
  - Prolog macht das automatisch beim Beweis von told/0.

Verarbeitung eines Dateiinhalts

Schema für Verarbeiten einer Datei mit Prolog:

```
process_file(F) :-
  see(F),               % Oeffne Datei
  repeat,
    read(T),            % Term einlesen
    process_term(T),    % Term verarbeiten
    T == end_of_file, % Fertig, falls
  !,                    % Dateiende
  seen.                 % Schliesse Datei
```

- repeat/0 ist ein eingebautes Prädikat, das beliebig oft gelingt.

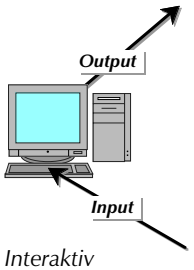
```
repeat.
repeat :- repeat.
```


Wer ist der user?

Die abstrakten Dateien "user"
- liefern Input von der Tastatur
- schreiben Output auf den Bildschirm

im interaktiven Betrieb!

Aber: *Der Input kann auch von einem andern Programm kommen und an ein anderes ausgegeben werden!*



Interaktiv

Prozesskommunikation

Rekursive Programmiertechniken

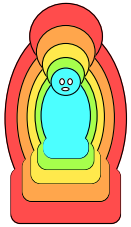
Übersicht

- Linksrekursion
- Transitive Relationen berechnen
  - Hierarchische Beziehungen: Hyponymie
- Dekomposition eines (von mehreren) Arguments
  - Listen verketten: append/3
- Akkumulatoren mit Zwischenresultaten gegen Ineffizienz
  - Listen umkehren: Naive vs. effiziente Version
- Doppelte Rekursion
  - verschachtelte Listen verflachen: flatten/2

Einfachste Rekursion mit Babuschka

Sinnvolle Definitionen von rekursiven Prozeduren müssen mindestens zwei Fälle abdecken.

- Abbruchbedingung(en)
- Rekursionsschritt(e)



Mit Babuschka spielen:

Die innerste Puppe kann nicht zerlegt werden.

Abbruchbedingung

Öffne die Puppe, lege Teile auf die Seite, spiele mit der enthaltenen Puppe weiter.

Rekursionsschritt

Linksrekursion mit Babuschka…

Linksrekursion liegt vor, wenn

- das erste Konjunkt im Rumpf dasselbe Prädikat ist wie im Kopf.

```
spiel(babuschka(Babuschka)):-
    spiel(Babuschka).
spiel(babuschka).
```

Gefahr

- Prolog-Beweiser kommt in unendlich tiefen Suchast!

```
?- spiel(Babuschka).
{ERROR: Memory allocation failed}
{Execution aborted}
```

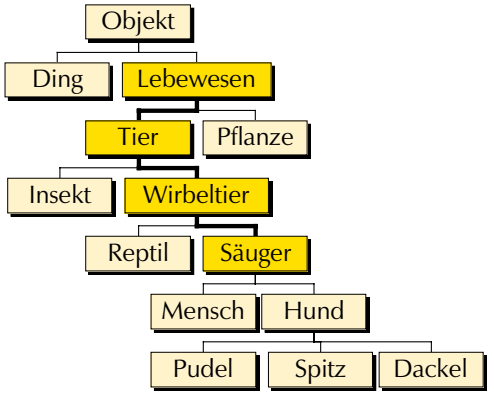
Ausweg

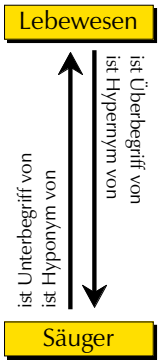
- Abbruchbedingungen vor die linksrekursive Klausel schreiben!

```
spiel(babuschka).
spiel(babuschka(Babuschka):-
   spiel(Babuschka).
```

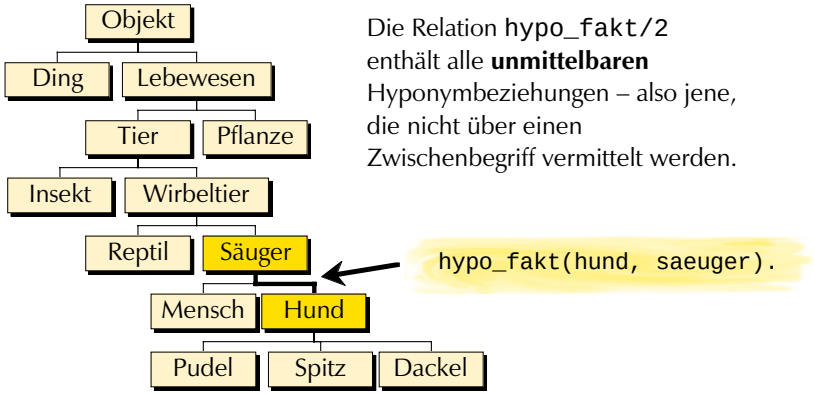
```
?- spiel(Babuschka).
Babuschka = babuschka ? ;
Babuschka = babuschka(babuschka) ? ;
...
```


Hyponymie: Unterbegriffshierarchie



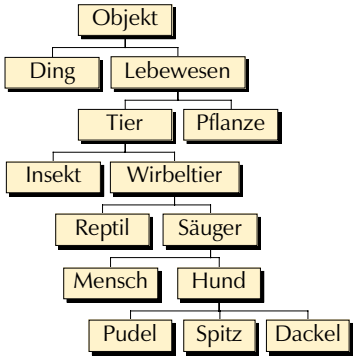


Unmittelbare Beziehung



Die Relation `hypo_fakt/2` enthält alle **unmittelbaren** Hyponymbeziehungen – also jene, die nicht über einen Zwischenbegriff vermittelt werden.

hypo_fakt/2




```
hypo_fakt(ding, objekt).
hypo_fakt(lebewesen, objekt).
hypo_fakt(tier, lebewesen).
hypo_fakt(pflanze, lebewesen).
hypo_fakt(insekt, tier).
hypo_fakt(wirbeltier, tier).
hypo_fakt(reptil, wirbeltier).
hypo_fakt(saeuger, wirbeltier).
hypo_fakt(mensch, saeuger).
hypo_fakt(hund, saeuger).
hypo_fakt(pudel, hund).
hypo_fakt(spitz, hund).
hypo_fakt(dackel, hund).
```

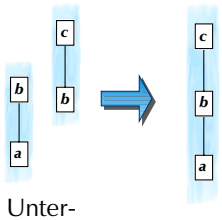

Unmittelbar und mittelbar hyponym

Problem: "Dackel" ist ein Hyponym von "Tier"

- Wie modellieren wir alle mittelbaren (=nicht unmittelbaren) Hyponymiebeziehungen, ohne sie alle einzeln aufzuzählen?

Idee: Hyponymie als Transitive Relation

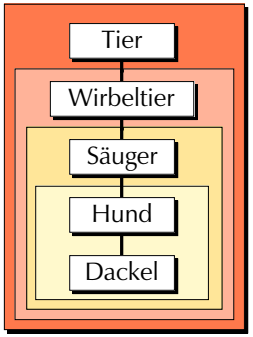
- Eine Relation *R* ist **transitiv**, falls gilt: Wenn *R*(*a*,*b*) und *R*(*b*,*c*) besteht, dann besteht *R*(*a*,*c*).



- **Also**: Wenn *a* Unterbegriff von *b* ist, und *b* ist Unterbegriff von *c*, dann ist auch *a* Unterbegriff von *c*.

hypo = hypo1 + hypo2 + hypo3 + …

Stufen der Hyponymie




```
hypo1(X, Y) :-
    hypo_fakt(X, Y).
```

```
hypo2(X, Y) :-
    hypo_fakt(X, A),
    hypo_fakt(A, Y).
```

```
hypo3(X, Y) :-
    hypo_fakt(X, A),
    hypo_fakt(A, B),
    hypo_fakt(B, Y).
```

```
hypo4(X, Y) :-
    hypo_fakt(X, A),
    hypo_fakt(A, B),
    hypo_fakt(B, C),
    hypo_fakt(C, Y).
```

```
hypo1(X, Y) :-
    hypo_fakt(X, Y).
```

```
hypo2(X, Y) :-
    hypo_fakt(X, A),
    hypo1(A, Y).
```

```
hypo3(X, Y) :-
    hypo_fakt(X, A),
    hypo2(A, Y).
```

```
hypo4(X, Y) :-
    hypo_fakt(X, A),
    hypo3(A, Y).
```

… hypo rekursiv

hypo2, ..., hypo4 liegt *ein* Schema zugrunde:

```
hypo1(X, Y) :-
  hypo_fakt(X, Y).
```

```
hypo2(X, Y) :-
  hypo_fakt(X, A),
  hypo1(A, Y).
hypo3(X, Y) :-
  hypo_fakt(X, A),
  hypo2(A, Y).
hypo4(X, Y) :-
  hypo_fakt(X, A),
  hypo3(A, Y).
```

}

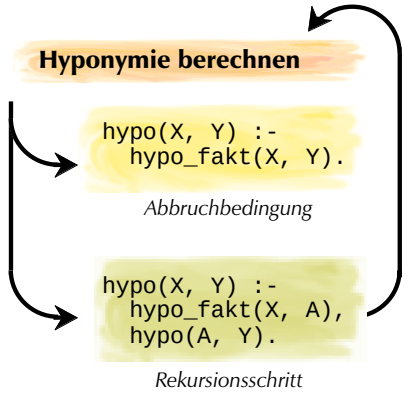
```
hypo(X, Y) :-
  hypo_fakt(X, Y).
```

```
hypo(X, Y) :-
  hypo_fakt(X, A),
  hypo(A, Y).
```


Bestandteile von hypo/2

Die rekursive Definition von hypo/2:

- Abbruchbedingung
  - *Unmittelbare Hyponymie*
- Rekursionsschritt
  - *Hyponymie, die über mehrere unmittelbare Stufen geht*
  - *Die Transitive "Hülle" der unmittelbaren Hyponymie wird rekursiv berechnet, d.h. die normale transitive Hyponymie-Relation.*



Listen verketten: append/3

[a, b, c] + [d, e] = [a, b, c, d, e]

Das Verketten zweier Listen wird häufig gebraucht.

- Das Prädikat `append/3` drückt die Verkettungsbeziehung aus.
  - Manchmal heisst es auch concat/3 in Programmbibliotheken.

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


Rekursive Dekomposition

Abbruchbedingung

- Die Verkettung der leeren Liste mit einer Liste *L* ergibt wieder *L*.

```
append([], L, L).
```

Rekursionsschritt

- Um eine nicht leere Liste [*X*|*L1*] mit einer Liste *L2* zu verketten, verkette *L1* mit *L2* zu *L3* und stelle *X* als Anfangselement zu *L3*.

```
append([X|L1], L2, [X|L3]) :-
  append(L1, L2, L3).
```

Grammatikregeln als Listenverkettung

Grammatikregeln lassen sich als Listenverkettungen von Wörtern formulieren.

$$S \rightarrow NP \quad VP$$

```
[fido,frisst] ⟶ [fido] [frisst]
```

- Eine Wortliste *X* ist ein syntaktisch korrekter Satz, falls sie aus 2 Teillisten *Y* und *Z* besteht, so dass gilt:
  - *X* ist die Verkettung der Liste *Y* und *Z* (in dieser Reihenfolge)
  - die Wortliste *Y* ist eine syntaktisch korrekte Nominalphrase
  - die Wortliste *Z* ist eine syntaktisch korrekt Verbalphrase.

Parsen mit append/3

Die Grammatikregeln lassen sich mit append/3 direkt formulieren und funktionieren als *Parser* (Programm zur syntaktischen Analyse).

```
% S --> NP VP
s(X) :-
   append(Y, Z, X),
   np(Y),
   vp(Z).

% NP --> Eigenname
np([fido]).

% VP --> Vintrasitiv
vp([frisst]).
```

```
?- s([fido,frisst]).
yes
```

"Fido frisst" ist ein syntaktisch korrekter Satz.

```
?- s([frisst,frisst]).
no
```

"frisst frisst" ist kein syntaktisch korrekter Satz.

Listen umkehren

[k, a, n, u] → [u, n, a, k]

Gelegentlich ist es nötig, eine Liste umzudrehen.

- Version I `naive_reverse/2`: »naives« Umkehren
  - Laufzeit verhält sich kubisch zur Listenlänge (n^3)!
  - Um eine Liste der Länge 100 umzudrehen, brauchts etwa 1'000'000 Schritte.
- Version II `reverse_akku/3`: Akkumulator für Zwischenresultat
  - Laufzeit verhält sich linear zur Listenlänge (n)!
  - Um eine Liste der Länge 100 umzudrehen, brauchts etwa 100 Schritte.

Akkumulatoren sind ein wichtiges Mittel zur Effizienzsteigerung!

Listen umkehren: »Naive« Version

Abbruchbedingung

- Die Umkehrung der leeren Liste ist die leere Liste.

```
naive_reverse([], []).
```

Rekursionsschritt

- Um eine nicht leere Liste [*X*|*Rest*] umzukehren, kehre den *Rest* um und verkette ihn mit der Einerliste [*X*].

```
naive_reverse([X|Rest], Ergebnis) :-
   naive_reverse(Rest, RevRest),
   append(RevRest, [X], Ergebnis).
```

Listen umkehren: Akkumulator

Akkumulatoren dienen dem Festhalten und Weitergeben von Zwischenergebnissen bei rekursiven Prädikaten.

- Es muss eine Argumentstelle für den Akkumulator geschaffen werden.

```
reverse(Liste, UmgekehrteListe) :-
   reverse_akku(Liste, [], UmgekehrteListe).
```

- reverse/2 wird auf reverse_akku/3 reduziert.
- Am Anfang ist der Akkumulator leer, d.h. die leere Liste.

Akkumulator: Abbruchbedingung

Abbruchbedingung

- Bei der Umkehrung der leeren Liste enthält das akkumulierte Zwischenresultat das Endresultat.

```
reverse_akku([], Akku, Akku).
```

- Für leere Listen wird reverse/2 korrekt durch reverse_akku/3 abgebildet…

```
?- reverse([], RL).
RL = []
```

⇨

```
?- reverse_akku([], [], RL).
RL = []
```


Akkumulator: Rekursionsschritt

Rekursionschritt

- Um eine nicht leere Liste [*X*|*Rest*] umzukehren, kehre den *Rest* mit dem neuen Zwischenresultat [*X*|*Akku*] um.

```
reverse_akku([X|Rest], Akku, Ergebnis):-
   reverse_akku(Rest, [X|Akku], Ergebnis).
```

- Die Eingabeliste schrumpft mit jedem rekursiven Aufruf.
- Die Akkumulatorliste wächst mit jedem rekursiven Aufruf und stellt dann beim Abbruch das Ergebnis dar.
- Metapher: Die Elemente der Eingabeliste werden im Akkumulator umgestapelt!

Programmiertechnik Akkumulatoren

Akkumulatoren werden normalerweise

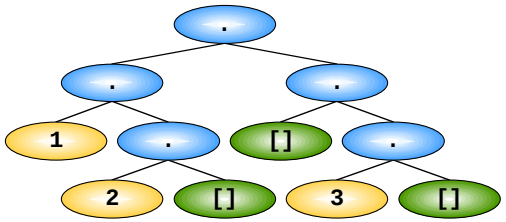
- **initialisiert** beim ersten Aufruf
- **akkumuliert** bei rekursiven Schritten
- **unifiziert** zum Endresultat beim Erreichen der Abbruchbedingung

```
reverse(Liste, UmgekehrteListe) :-
   reverse_akku(Liste, [], UmgekehrteListe).

reverse_akku([], Akku, Akku).

reverse_akku([X|Rest], Akku, Ergebnis):-
   reverse_akku(Rest, [X|Akku], Ergebnis).
```

Doppelrekursion: Verschachtelte Listen

Da Listen normale Terme sind, kann eine Liste auch Element einer anderen Liste sein.

```
?- [[1,2],[],3] = '.'('.'(1,'.'(2,[])),'.'([],'.'(3,[]))).
yes
```



Verschachtelte Listen verflachen

```
?- flatten([[1,2],[],3], Liste).
Liste = [1,2,3]
```

Das Prädikat flatten/2 wandelt eine verschachtelte Liste in flache Listen um.

- Was ist eine flache Liste?

  Eine flache Liste ist eine Liste, deren Elemente keine Listen sind.

- Was ist eine verschachtelte Liste?

  Eine verschachtelt Liste ist eine Liste, deren Elemente zum Teil aus Listen bestehen.

  - Listen sind verschachtelte Datenstrukturen, also sind verschachtelte Listen verschachtelte verschachtelte Datenstrukturen…

Definition von flatten/2

Abbruchbedingung "leere Liste"

```
flatten([],[]).        % Leere Liste ist flach
```

Abbruchbedingung "keine Liste"

```
flatten(X,[X]):-       % Terme in flache Listen packen,
   \+ is_list(X).      % die keine Listen sind
```

Rekursionsschritt "Doppelte Rekursion"

```
flatten([Kopf|Rest], FlacheListe):-
   flatten(Kopf, FlacherKopf),          % Kopf verflachen
   flatten(Rest, FlacherRest),          % Rest verflachen
   append(FlacherKopf, FlacherRest, FlacheListe).
                                        % und verketten
```

Definite Clause Grammar (DCG)

Übersicht

- Formale Sprachen
- Kontextfreie Grammatiken (*Context-Free Grammars*)
  - Terminale, Nichtterminale und Regeln
- Kontextfreie Grammatik in Form von DCGs
- Formales Ableiten von Satzformen und Sätzen
- phrase/2: DCG-Standardparser in Prolog
- Prologinterne Repräsentation und Verabeitung von DCGs
- Parsing-Strategie: Top-Down und Left-Right
- Problem: Linksrekursive Grammatiken
  - Linguistische Motivation für linksrekursive Grammatiken
  - Abhilfen

Formale Sprachen

Vokabular *T* einer Sprache (Terminale)

- $T_{Englisch}$ = {aardvark,…,cat,…,woman,…,zymurgy}

*T** ist die Menge aller endlichen Folgen des Vokabulars *T*.

- $T_{Englisch}^*$ =
  { ε,
  a,aardvark,cat,woman,…
  a cat, cat a, peter sleeps,…
  a a a, a cat sleeps, woman a cat,…
  …}

Folge aus 0 Elementen (epsilon)
Folgen aus 1 Element
Folgen aus 2 Elementen
Folgen aus 3 Elementen
Folgen aus n Elementen

Sprache *L* über Vokabular *T* ist Teilmenge von *T**

- $L_{Englisch}$ = {yes,…,peter sings,…,a cat sleeps,…,the man loves her,…}

▸ Mit Grammatiken kann die gewünschte Teilmenge von T* formal und elegant spezifiziert werden.

Kontextfreie Grammatiken (CFG)

Beispiele für Regeln einer kontextfreien Grammatik:

S → NP VP	*Det → the*	*V → loves*
VP → V NP	*Det → a*	*V → sings*
VP → V	*N → cat*	*V → sees*
	N → woman	*V → thinks*

Gemäss diesen Grammatikregeln sind etwa folgende Sätze (S) erlaubt bzw. nicht erlaubt (*)

- a cat sings
- *woman sees cat
- the woman loves a cat

Kontextfreie Regeln

Bestandteile einer kontextfreien Regel:

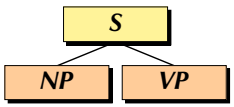
S → *NP VP*

linke Seite
left-hand side

rechte Seite
right-hand side

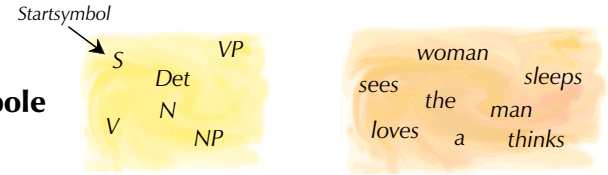
Lesarten:

- Ein S besteht aus einer NP und einer VP.
- Eine NP gefolgt von einer VP ergibt ein S.
- S expandiert zu NP und VP.



CFG: Symbolisch-Graphisch…

Symbole



Nicht-Terminal-Symbole
"Syntaktische Kategorien"

Terminal-Symbole
"Lexikon"

Regeln

$S \rightarrow NP\ VP$	$Det \rightarrow the$	$V \rightarrow loves$
$NP \rightarrow Det\ N$	$Det \rightarrow a$	$V \rightarrow sings$
$VP \rightarrow V\ NP$	$N \rightarrow cat$	$V \rightarrow sees$
$VP \rightarrow V$	$N \rightarrow woman$	$V \rightarrow thinks$

CFG: Mathematisch…

Jede **kontextfreie Grammatik** lässt sich durch ein 4-Tupel (N, T, R, S) beschreiben, wobei gilt:

- N ist endliche Menge von **Nicht-Terminal-Symbolen**
- T ist endliche Menge von **Terminalsymbolen** ($T \cap N = \emptyset$)
- R ist endliche Menge von **Regeln** ($R \subseteq N \times (N \cup T)^*$)
- S ist das **Startsymbol** ($S \in N$)

Pfeilnotation $A \rightarrow \alpha$ für Regeln

- ist lesbarere Schreibvariante für Tupel (A, α), mit $A \in N$ und $\alpha \in (N \cup T)^*$

CFG im DCG-Formalismus

Regeln mit Nicht-Terminalen auf der rechten Seite:

- Syntaktische Regeln

```
s --> np, vp.
np --> det, n.
```

```
vp --> v.
vp --> v, np.
```

Regeln mit Terminal-Symbolen auf der rechten Seite:

- Lexikalische Regeln

```
det --> [the].
det --> [a].
```

```
n --> [cat].
v --> [sees].
v --> [sings].
```

- Verwendung von Terminalen und Nicht-Terminalen in der RHS wäre möglich, macht die Sache aber unübersichtlicher.

Ableiten von Sätzen

Satzformen einer Grammatik *G*

- S ist eine **Satzform**, falls S das Startsymbol von G ist.
- S ist eine **Satzform** der Form $\alpha\delta\gamma$, falls gilt
  - es gibt eine Satzform der Form $\alpha B\gamma$, und
  - es gibt eine Regel der Form $B \rightarrow \delta$.

$S \Rightarrow NP\ VP \Rightarrow Det\ N\ VP \Rightarrow Det\ N\ V \Rightarrow$ the N V $\Rightarrow$ the cat V $\Rightarrow$ the cat sings

Sätze einer Grammatik *G*

- S ist ein **Satz** von G, falls gilt:
  - S ist eine Satzform von G, und
  - S enthält nur Terminalsymbole von G.

DCG-Parsen mit phrase/2

Das eingebaute Prädikat phrase/2 überprüft, ob von einem Nicht-Terminal eine Liste von Terminalsymbolen abgeleitet werden kann.

```
?- phrase(s, [the,cat,sings]).
yes
```

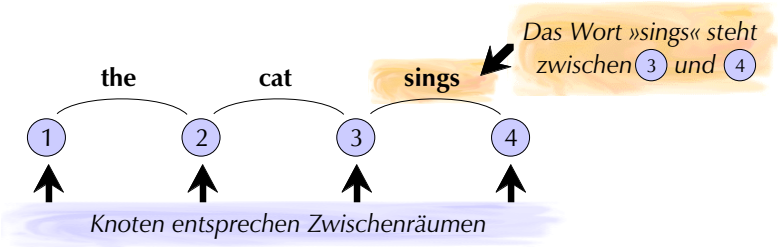
```
?- phrase(np, [a,cat]).
yes
```

```
?- phrase(s, [a,cat]).
no
```

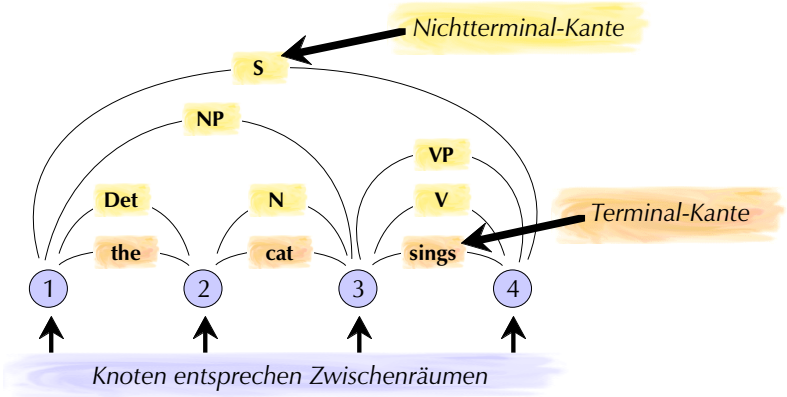

Graphische Veranschaulichung

Anhand einer Grafik lässt sich die Grundidee zeigen, wie der in Prolog eingebaute Parser für DCGs funktioniert

- Die Start-/Zwischen-/Endpositionen werden vergegenständlicht!
- Prologs Parser speichert seine Daten jedoch *nicht* numerisch!



Graphische Veranschaulichung



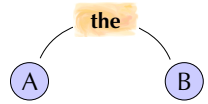
Einlesezauber I: DCG zu Prolog

Beim Einlesen (»Konsultieren«) eines Programms werden DCG-Regeln in gewöhnliche Prolog-Klauseln übersetzt.

- Regel mit Terminal-Symbolen auf der rechten Seite:

```
det --> [the].
```

```
det(A, B) :-
  'C'(A, the, B).
```



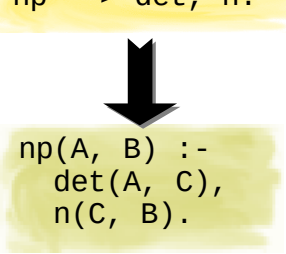
Die Knoten A und B sind durch »the« verbunden.

Einlesezauber II: DCG zu Prolog

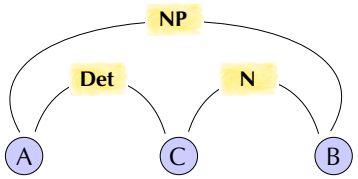
Beim Einlesen (»Konsultieren«) eines Programms werden DCG-Regeln in gewöhnliche Prolog-Klauseln übersetzt.

- Regel mit Nichtterminalen auf der rechten Seite:

```
np --> det, n.
```



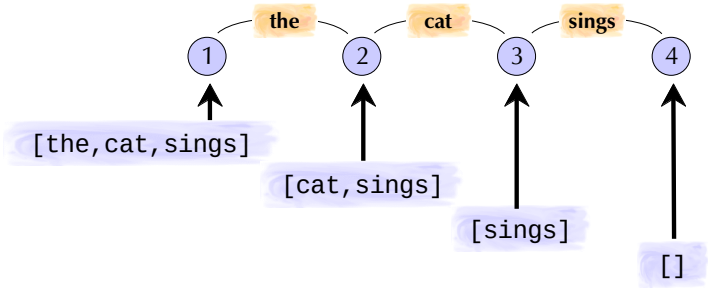
```
np(A, B) :-
  det(A, C),
  n(C, B).
```



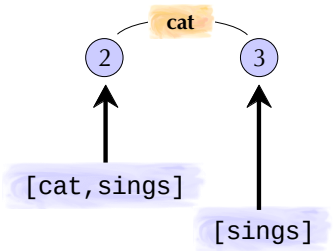
Damit zwischen A und B eine NP steht, braucht es zwischen A und einem Knoten C ein Det, sowie zwischen C und B ein N.

Repräsentation der Zwischenknoten

Für Knoten stehen Listen mit dem Rest des Satzes.



'C'/3: C heisst *connect*



In Prolog vordefiniert

```
'C'([Word|Rest], Word, Rest).
```

Der allgemeine Fall: Wann verbindet ein Wort zwei Knoten?

```
'C'([cat,sings], cat, [sings]).
```

»cat« verbindet Knoten 2 und 3.

Einlesezauber durch Termexpansion

Beim Einlesen (»Konsultieren«) eines Programms werden DCG-Regeln in gewöhnliche Prolog-Klauseln übersetzt.

```
s --> np, vp.
np --> det, n.
vp --> v.
vp --> v, np.
```

```
s(A,B)  :- np(A,C), vp(C,B).
np(A,B) :- det(A,C), n(C,B).
vp(A,B) :- v(A,B).
vp(A,B) :- v(A,C), np(C,B).
```

```
det --> [the].
det --> [a].
n --> [cat].
v --> [sees].
v --> [sings].
```

```
det(A,B) :- 'C'(A, the, B).
det(A,B) :- 'C'(A, a, B).
n(A,B)   :- 'C'(A, cat, B).
v(A,B)   :- 'C'(A, sees, B).
v(A,B)   :- 'C'(A, sings, B).
```

DCG-Atome als Prädikate

An das Ergebnis dieser Übersetzung können Prolog-Anfragen gestellt werden:

```
?- det([the,cat], [cat]).
yes
```

Terminale

```
det(A,B) :- 'C'(A, the, B).
det(A,B) :- 'C'(A, a, B).
n(A,B)   :- 'C'(A, cat, B).
v(A,B)   :- 'C'(A, sees, B).
v(A,B)   :- 'C'(A, sings, B).
```

```
?- np([the,cat], []).
yes

?- s([the,cat,sings], []).
yes
```

Nichtterm.

```
s(A,B)  :- np(A,C), vp(C,B).
np(A,B) :- det(A,C), n(C,B).
vp(A,B) :- v(A,B).
vp(A,B) :- v(A,C), np(C,B).
```

vordef.

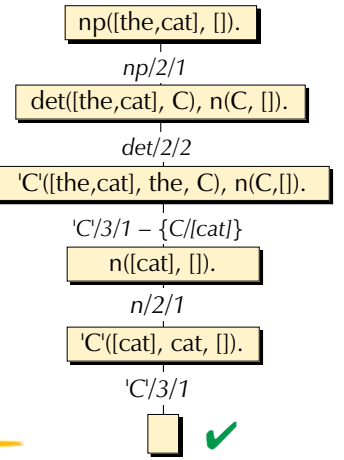
```
'C'([Word|Rest], Word, Rest).
```


DCG: Parsen heisst Beweisen

Schritte zum Beweis von

```
?- np([the,cat], []).
yes
```

Wegen der *Top-Down*- und *Left-Right*-Strategie des Prolog Beweisers arbeitet der DCG-Parser die zu analysierende Kette ebenfalls *Top-Down* und *Left-Right* ab.



Linksrekursive Grammatiken

```
np --> np, conj, np.
```

```
conj --> [and].
```

```
np(A, B) :-
   np(A, C),
   conj(C, D),
   np(D, B).
```

```
conj(A, B) :-
   'C'(A, and, B).
```

Was geschieht bei folgender Anfrage?

```
?- phrase(np,[a,cat,and,a,cat]).
   1  1  Call: phrase(user:np,[a,cat,and,a,cat]) ?
   2  2  Call: np([a,cat,and,a,cat],[]) ?
   3  3  Call: np([a,cat,and,a,cat],_1210) ?
   4  4  Call: np([a,cat,and,a,cat],_1616) ?
...
```


Linksrekursive Grammatiken

Wo liegt das Problem?

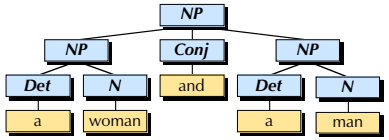
- der Parser gerät in einen endlose Schleife
- weil von einem Nicht-Terminal eine Kette abgeleitet werden kann, die wiederum mit demselben Nicht-Terminal beginnt
- der Parser springt von einem Nichtterminal zum nächsten, ohne ein Terminalsymbol zu konsumieren

Derartige Grammatiken heissen *links-rekursiv*.

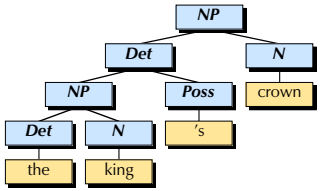
- Links-Rekursion ist für Top-Down-Parser, die von links nach rechts arbeiten, ein Problem.

Linksrekursive Grammatiken

LinguistInnen brauchen linksrekursive Grammatiken.



$NP \rightarrow NP \ Conj \ NP$



$NP \rightarrow Det \ N$

$Det \rightarrow NP \ Poss$

Linksrekursive Grammatiken

Mögliche Abhilfen

- linksrekursive Grammatiken verbieten
- Grammatik so umwandeln, dass sie nicht mehr linksrekursiv ist
  - dies ist für jede kontextfreie Grammatik möglich
  - aber die den Sätzen zugewiesene Struktur ist dann nicht mehr so, wie sich das die LinguistInnen wünschen
- ein anderes Parsing-Verfahren verwenden, das mit links-rekursiven Grammatiken zurecht kommt

In der Praxis wird häufig die dritte Variante gewählt

- denn andere Parsing-Verfahren sind effizienter als reine Top-Down-Algorithmen (hängt sehr stark von Implementation ab)

Definite Clause Grammar II

Übersicht

- DCG-Formalismus vs. DCG-Standard-Parser
- DCG zu Prolog-Übersetzung: Light-Version
- Erweiterungen: Komplexe Nicht-Terminal-Symbole
  - Modellierung von Kongruenz
  - Aufbau von Syntaxstrukturen
  - Abdeckungsgrad von DCGs
- Akzeptor vs. Parser
- Eingebettete Prolog-Klauseln
- Fazit: DCG und Prolog

DCG-Formalismus vs. Standard-Parser

DCG als Grammatik-Formalismus

- *Definite Clause Grammars* erlauben eine systematische Beschreibung der formalen Regularitäten einer Sprache.

```
s --> np, vp.
np --> det, n.
```

```
det --> [the].
det --> [a].
```

Der Standard-DCG-Parser von Prolog

- In den meisten Prologs gibt es einen Mechanismus, der DCGs automatisch in ein Prolog-Programm übersetzt, das die Sätze der von der DCG beschriebenen Sprache analysieren kann.
- Dieser Parser hat Probleme mit linksrekursiven Regeln.

DCG zu Prolog: Light-Version

Wie werden DCGs zu Prolog-Programmen übersetzt?

- *Im Folgenden: Vereinfachte Version der DCG-Übersetzung*
- translate/2 nimmt eine DCG-Regel und übersetzt (compiliert) sie in eine Prolog-Klausel.

```
translate((LHS --> RHS), (Head :- Body)) :-
   left_hand_side(LHS, Start, End, Head),
   right_hand_side(RHS, Start, End, Body).
```

```
np --> det, n.
```

```
np(Start, End) :-
  det(Start, Middle),
  n(Middle, End).
```


DCG zu Prolog: LHS

Wie wird die *Left-Hand-Side* übersetzt?

- left_hand_side/4 setzt die *Start*- und *End*-Variable
- Nicht-Terminal *NT* muss ein Atom sein (atom/1)
- *Head* wird mittels =../2 als komplexer Term zusammengesetzt

```
left_hand_side(NT, Start, End, Head) :-
   atom(NT),
   Head =.. [NT,Start,End].
```

```
np --> det, n.
```

```
np(Start, End) :-
  det(Start, Middle),
  n(Middle, End).
```

DCG zu Prolog: RHS mit Terminal

Wie wird die *RHS* übersetzt?

- Falls ein Terminal-Symbol (listenförmig!) vorliegt
  - Abbruchbedingung
  - 'C'/3-Aufruf einsetzen mit Terminal-Symbol als 2. Argument

```
right_hand_side([T], Start, End, 'C'(Start,T,End)).
```

```
det --> [der].
```

```
det(Start, End) :-
  'C'(Start, der, End).
```


DCG zu Prolog: RHS rekursiv

Wie wird die *Right-Hand-Side* (*RHS*) übersetzt?

- Falls mehrere Symbole auf der rechten Seite sind
  - Doppelte Rekursion
  - Erzeugen der Zwischenposition *Middle*

```
nt --> s1,s2,s3.
```

```
nt --> (s1,(s2,s3)).
```

```
right_hand_side((S1,S2), Start, End, (Body1, Body2)) :-
  right_hand_side(S1, Start, Middle, Body1),
  right_hand_side(S2, Middle, End, Body2).
```

```
np --> det, n.
```

```
np(Start, End) :-
  det(Start, Middle),
  n(Middle, End).
```


DCG zu Prolog: Light-Version

Wie wird die *Right-Hand-Side* übersetzt?

- Falls Nicht-Terminal-Symbol
  - NT-Symbol muss ein Atom sein
  - Rumpfteil wird als komplexer 2-stelliger Term zusammengesetzt mit =../2
  - Abbruchbedingung für die Rekursion

```
right_hand_side(NT, Start, End, Body) :-
  atom(NT),
  Body =.. [NT,Start,End].
```

```
vp --> v.
```

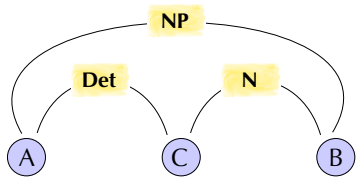
```
vp(Start, End) :-
  v(Start, End).
```


DCG zu Prolog: Light-Version

Die rekursive Dekomposition von right_hand_side/4 fügt die Aufrufe mit den jeweiligen Position korrekt ein:

```
?- translate((np --> det, n), X).
X = np(A,B):-det(A,C),n(C,B) ? ;
no
```

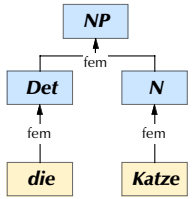
Eine NP besteht zwischen A und B, falls zwischen A und C ein Det besteht und zwischen C und B ein N.



Komplexe Nicht-Terminal-Symbole

DCG mit komplexen NT-Symbolen

- Zusätzliche Information kann repräsentiert und unifiziert werden!
- Beispiel: **Kongruenz** *(Agreement)*
  - Im Deutschen müssen Genus von Artikel und Nomen muss übereinstimmen!
  - Merkmale werden hinaufunifiziert!




```
np(Genus) -->
   det(Genus),
   n(Genus).
n(fem) --> [katze].
det(fem) --> [die].
```

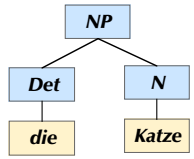
```
np(Genus, A, B) :-
   det(Genus, A, C),
   n(Genus, C, B).
n(fem, A, B) :- 'C'(A, katze, B).
det(fem, A, B) :- 'C'(A, die, B).
```


Konstruktion eines Syntaxbaums

DCG mit komplexen NT-Symbolen: Syntaxstrukturen

- In der LHS jeder Grammatikregel wird die der Regel entsprechende syntaktische Datenstrukutur im Zusatzargument aufgebaut.

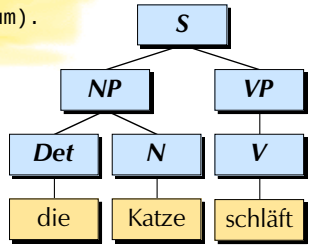
```
                          n(n(katze)) --> [katze].
np(np(Det,N)) --> det(Det),  n(N).
                          det(det(die)) --> [die].
```



- Die Grammatiksymbole wie S, NP, Det werden dabei gerne mehrdeutig verwendet:
  - als Parseprädikate, Datenstrukturen und Variablen
    - *Man könnte auch jeweils andere Bezeichner verwenden!*

Parsen mit komplexen Symbolen

```
s(s(NP,VP)) --> np(NP,nom,Num), vp(VP,Num).
np(np(Det,N),Kas,Num) -->
   det(Det,Gen,Kas,Num), n(N,Gen,Kas,Num).
vp(v(V),Num) --> v(V,Num).

n(n(katze),fem,Kas,sg) --> [katze].
v(v(schlaeft),sg) --> [schlaeft].
v(v(schlafen),pl) --> [schlafen].
det(det(die),fem,nom,sg) --> [die].
det(det(der),fem,gen,sg) --> [der].
...
```




```
?- phrase(s(Baum), [die,katze,schlaeft]).
Baum = s(np(det(die),n(katze)),vp(v(schlaeft)))
```


Akzeptor – Parser

Ein Programm zur syntaktischen Analyse

- nimmt eine Kette von Wörtern entgegen.
- beurteilt, ob die Eingabe gemäss den Regeln einer Grammatik zulässig ist.

Akzeptoren

- antworten nur mit »zulässig« bzw. »nicht zulässig«.

Parser

- geben bei zulässigen Eingaben zusätzlich die mögliche(n) syntaktische(n) Struktur(en) der Eingabekette aus.

Mächtigkeit von DCG

Komplexe Nicht-Terminal-Symbole beeinflussen die mathematischen Eigenschaften des Formalismus.

- Die modellierte Sprache kann ausserhalb der Klasse der kontextfreien Sprachen liegen!

Grammatik für $\{a^n\, b^n\, c^n \mid n \geq 1\}$

```
s --> as(N), bs(N), cs(N).
as(1)       --> [a].
as(succ(A)) --> [a], as(A).
bs(1)       --> [b].
bs(succ(B)) --> [b], bs(B).
cs(1)       --> [c].
cs(succ(C)) --> [c], cs(C).
```

Anfragen

```
?- phrase(s, [a,b,c]).
yes

?- phrase(s, [a,a,b,c]).
no
```


Eingebettete Prolog-Klauseln

Prädikatsaufrufe innerhalb geschweifter Klammern

- werden bei der Übersetzung von DCGs unverändert übernommen.
- Dadurch können beliebige Prolog-Programme in die Grammatik eingebettet werden.

```
lex(cat, n).
lex(dog, n).
lex(cow, n).
```

```
n(n(N)) -->
   [N],
   {lex(N, n)}.
```

```
n(n(A), B, C) :-
   'C'(B, A, C),
   lex(A, n).
```

Beispiel
Anstelle von lexikalischen DCG-Regeln wird ein Lexikon aus lex/2-Fakten verwendet mit Lexem und lexikalischer Kategorie als Argumenten.

Eingebettete Prolog-Klauseln

Vorteile der Prolog-Einbettung

- manchmal kann die Effienz gesteigert werden
  - Es darf auch der cut/0 eingesetzt werden. (Sogar ohne geschweifte Klammern!)
- kontextsensitive Sprachen können erkannt werden

Nachteile der Prolog-Einbettung

- keine deklarative Spezifikation der Grammatik
- Grammatiken werden schnell unübersichtlich
- Reine DCGs können als Formalismus auch von anderen Programmiersprachen/Grammatikkompilierern verarbeitet werden; mit eingebettetem Prolog wird eine Prolog-Abhängigkeit geschaffen

Fazit Prolog und DCGs

Vorteile von DCGs

- in Prolog ist ein einfacher Top-Down-Parser bereits eingebaut
- DCGs sind in Prolog Grammatik und Parser-Programm gleichzeitig
- nützlich zum schnellen Spezifizieren/Ausprobieren einer Mini-Grammatik
- als reiner Formalismus unabhängig von Umsetzung in Parser-Programm

Nachteile vom Standard-Prolog-DCG-Verarbeitung

- »Aufhängen« bei linksrekursiven Grammatiken
- eher ineffizientes Verfahren
  - bei mehrdeutigen Grammatiken werden unter Umständen Teile des Satzes mehrmals analysiert
  - lexikalische Regeln werden ineffizient für grossen Lexika

- **Für richtige Sprachverarbeitungsprojekte kaum brauchbar.**

Term-Prädikate

Übersicht

Eingebaute Term-Prädikate

- Termsynthese und Termanalyse zur Laufzeit
  - =../2, functor/3, arg/3
  - name/2 *(siehe Folien zu Ein- und Ausgabe)*
  - atomic/1, atom/1, var/1, nonvar/1, compound/1, number/1, integer/1, float/1 (*siehe Folien zur Syntax*)
- Identität und Nicht-Identität von Termen
  - ==/2, \==/2
- Standardordnung, Vergleichsprädikate und Sortierung von Termen
  - @</2, @=</2, @>/2, @>=/2
  - sort/2

Termsynthese und -analyse: =../2

- **Das Prädikat =. . /2 (*univ*) verwandelt einen Term in eine Liste,**
  - deren erstes Element gleich dem Funktor des Terms ist,
  - und deren restliche Elemente gleich den einzelnen Argumenten des Terms sind.

```
?- f(arg1,arg2) =.. Liste.
Liste = [f,arg1,arg2]
```

=. . /2 kann auch aus einer Liste einen Term bauen:

```
?- Term =.. [f,arg1,arg2].
Term = f(arg1,arg2)
```


Termsynthese und -analyse: functor/3

- **Das Prädikat functor(*Term*, *F*, *S*) gelingt, falls gilt**
  - *Term* ist komplexer Term mit Funktor *F* und Stelligkeit *S*,

```
?- functor(hund(fido), Funktor, Stelligkeit).
Funktor = hund, Stelligkeit = 1
```

  - oder *Term* ist ein Atom/eine Zahl, wobei *Term* = *F* und *S* = 0

```
?- functor(a, F, S).
F = a, S = 0
```

Mit functor/3 lassen sich neue komplexe Terme bilden:

```
?- functor(Neu, b, 2).
Neu = b(_22, _23)
```


Termsynthese und -analyse: arg/3

- **arg(*N*, *Term*, *Arg*) gelingt, wenn *Arg* das *N*-te Argument von *Term* ist.**

```
?- arg(3, f(a,b,c), Was).
Was = c
yes
```

```
?- arg(1, f(a,b,c), a).
yes
```

```
?- arg(2, f(a,b,c), d).
no
```

  - Weder *N* noch *Term* dürfen freie Variablen sein!

```
?- arg(N, f(a,d), d).
{INSTANTIATION ERROR: arg(_36,f(a,d),_38) - arg 1}
?- arg(2, Term, d).
{INSTANTIATION ERROR: arg(2,_34,_35) - arg 2}
```

Identität und Nicht-Identität

- **Das Prädikat ==/2 gelingt, wenn die zwei Argumentsterme identisch sind.**

```
?- hund(f) == hund(f).
yes
```

```
?- hund(Y) == hund(X).
no
```

- **Das Prädikat \==/2 gelingt, wenn die zwei Argumentsterme nicht identisch sind. Variablen werden nicht gebunden!**

```
?- hund(f) \== hund(X).
yes
```

```
Term1 \== Term2 :-
   \+ Term1 == Term2.
```

Mögliche Definition von `\==/2`

Standardordnung von Termen

- **Das Prädikat @</2 gelingt, wenn das erste Atom in alphabetischer Ordnung vor dem zweiten Atom steht.**

```
?- adam @< eva.
yes
```

```
?- adam @< adam.
no
```

Aber: Für *alle* Terme ist eine Standardordnung festgelegt

I. Variablen (nach Alter) – *Nicht nach Variablennamen!*
II. Fliesskommazahlen (nach numerischem Wert)
III. Ganzzahl (nach numerischem Wert)
IV. Atome (nach Zeichensatz-Kode)
V. Komplexe Terme (nach Stelligkeit, Name des Funktors, Standardordnung der Argument von links nach rechts)

```
?- C @< 1.2.
yes
```


Weitere Ordnungprädikate

Neben dem Term-Vergleichsprädikat @</2 (*echt kleiner*) gibt es wie in der Arithmetik noch andere Vergleichsprädikate, die sich auf diese Standardordnung beziehen:

- **@=</2 (*gleich oder kleiner*)**
- **@>=/2 (*grösser oder gleich*)**
- **@>/2 (*echt grösser*)**

Achtung: Es gibt keine Prädikate @<=/2 und @=>/2…

Standardsortierung von Termen

- **Das eingebaute Prädikat sort/2 bringt eine beliebige Termliste in die Standardordnung:**

```
?- sort([fo(0,2),X,fo,A,-9,-1.0,1,fi,fi(1,1,1),X = Y], L).
L = [X,A,-1.0,-9,1,fi,fo,X=Y,fo(0,2),fi(1,1,1)]
```

Identische Terme dürfen im 2. Argument nur einmal erscheinen.

- *Die Liste im 2. Argument ist also eine geordnete Menge!*

```
?- sort([1,'1',1], L).
L = [1,'1']
yes
```

```
?- sort([1,1], [1,1]).
no
```

Shift-Reduce-Parsing

Übersicht

- Parsing-Richtungen
  - Top-Down: hypothesengesteuert
  - Bottom-Up: datengesteuert
- Shift-Reduce-Parsing als Bottom-Up-Verfahren
  - Stapel als Datenstruktur
  - Der Algorithmus: Shift- und Reduce-Schritte
- Implementation in Prolog
  - Eingabekette konsumieren und Stapel aufschichten: shift/4
  - Grammatikregeln und Lexikon: brule/2 und word/2
  - Reduktion mittels Grammatikregeln: reduce/2
- Parsing-Algorithmus: shift_reduce/3
  - Terminierungsprobleme
  - Tilgungsregeln und zyklische Regeln

Top-Down

Ein Top-Down-Parser für eine kontextfreie Grammatik

- fängt mit dem Startsymbol an.
- führt wiederholt Ableitungsschritte durch.
  - expandiert jeweils LHS durch RHS.
- Ziel: Ableiten der zu analysierenden Kette

S → NP VP	*Det → a*
NP → Det N	*N → man*
VP → V	*V → sleeps*

S ⇒ ***NP*** *VP*
⇒ ***Det*** *N* VP
⇒ a **N** VP
⇒ a *man* **VP**
⇒ a man ***V***
⇒ a man *sleeps*

Legende

fett: Nicht-Terminal-Symbol, das im nächsten Schritt expaniert wird.

kursiv: Symbol, das durch letzten Schritt expandiert wurde

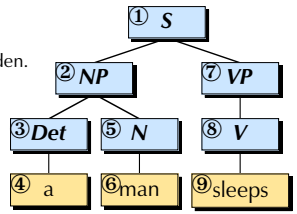
Top-Down

Vorgehen beim Top-Down-Parsing

- Ich suche ein S.
- Um S *(1)* zu erhalten, brauche ich eine NP und eine VP.
- Um NP *(2)* zu erhalten, brauche ich ein Det und ein N.
- Um Det *(3)* zu erhalten, kann ich das Wort »a« *(4)* verwenden.
- Um N *(5)* zu erhalten, kann ich das Wort »man« *(6)* verwenden.
- Damit ist die NP vollständig.
- Um VP *(7)* zu erhalten, brauche ich ein V.
- Um V *(8)* zu erhalten, kann ich das Wort »sleeps« *(9)* verwenden.
- Damit ist die VP vollständig.
- Damit ist das S vollständig.

S → NP VP	*Det → a*
NP → Det N	*N → man*
VP → V	*V → sleeps*

➥ *hypothesengesteuert*



Bottom-Up

Ein Bottom-Up-Parser für eine kontextfreie Grammatik

- fängt mit der zu analysierenden Kette an.
- führt wiederholt Ableitungsschritte »rückwärts« durch.
  - reduziert jeweils RHS auf LHS.
- Ziel: Erreichen des Startsymbols

S → NP VP	*Det → a*
NP → Det N	*N → man*
VP → V	*V → sleeps*

S ⇐ **NP** ***VP***
⇐ NP ***V***
⇐ *NP* **sleeps**
⇐ **Det** ***N*** sleeps
⇐ *Det* **man** sleeps
⇐ **a** man sleeps

Legende

fett: Nicht-Terminal-Symbol, das im nächsten Schritt reduziert wird

kursiv: Symbol, das durch letzten Schritt reduziert wurde

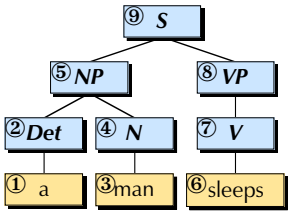
Bottom-Up

Vorgehen beim Bottom-Up-Parsing

- Nimm ein Wort — es ist »a« *(1)*.
- »a« ist ein Det *(2)*.
- Nimm ein weiteres Wort — es ist »man« *(3)*.
- »man« ist ein N *(4)*.
- Det und N bilden zusammen eine NP *(5)*.
- Nimm ein weiteres Wort — es ist »sleeps« *(6)*.
- »sleeps« ist ein V *(7)*.
- V bildet (für sich alleine) eine VP *(8)*.
- NP und VP bilden zusammen ein S *(9)*.

➥ *datengesteuert*

S → NP VP *Det → a*
NP → Det N *N → man*
VP → V *V → sleeps*



Shift-Reduce-Parsing

Das Shift-Reduce-Parsing ist ein einfaches Bottom-Up-Verfahren

- **Daten**
  - Eingabekette: Was muss noch verarbeitet werden? – **Liste**
  - Abarbeitungs-Stapel: Was haben wir alles schon erkannt? – **Stapel**
- **Aktionen**
  - Eingabekette konsumieren – **shift**
    - *Nimm ein Wort*
  - Grammatische Regeln anwenden – **reduce**
    - Lexikalische Regeln (*»a« ist ein Det*)
    - Syntaktische Regeln (*Det und N bilden zusammen eine NP*)

Shift-Reduce-Parsing

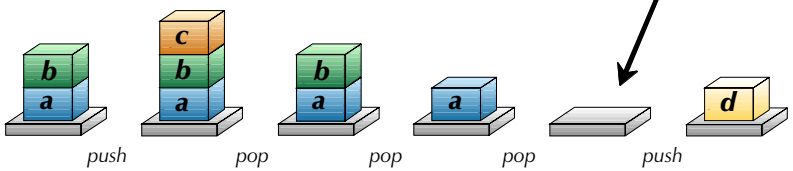
Schritt	Aktion	Stapel	Eingabekette
0	—	*ε*	*the man sleeps*
1	shift	*the*	*man sleeps*
2	reduce	*Det*	*man sleeps*
3	shift	*Det man*	*sleeps*
4	reduce	*Det N*	*sleeps*
5	reduce	*NP*	*sleeps*
6	shift	*NP sleeps*	*ε*
7	reduce	*NP V*	*ε*
8	reduce	*NP VP*	*ε*
9	reduce	*S*	*ε*

Die Daten: Stapel

Die Datenstruktur "Stapel" (auch: Keller), engl. *Stack*

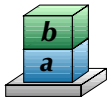
- zwei Operationen
  i. ***push*** — ein Element auf den Stapel darauflegen
  ii. ***pop*** — oberstes Element vom Stapel wegnehmen
- Zugriff (Einfügen und Wegnehmen) immer von oben
- Stapel im Alltag: Bücherstapel, Mensa-Tellerwärmer

leerer Stapel
empty stack



Stapel als Listen

Stapel können als Listen betrachtet werden.



$= [\ b, a\ ]$

oberstes Element auf Stapel
= erstes Element in Liste

$= [\ ]$

leerer Stapel
= leere Liste

push und pop als Prädikate über Listen

```
?- push(c, [b,a], Stack)).
Stack = [c,b,a]
```

```
?- pop(E, [c,b,a], New)).
E = c
New = [b,a]
```


Die Aktionen

In jedem Schritt führt ein Shift-Reduce-Parser eine von zwei möglichen Aktionen durch

- **Shift**
  - »Nimm ein Wort«
  - verschiebe ein Wort auf den Stapel (*schiebe*)
- **Reduce**
  - »X und Y bilden zusammen ein Z«
  - »X bildet (für sich alleine) ein Z«
  - »X ist ein Z«
  - wenn die obersten Stapelelemente gleich der rechten Seite einer Regel sind, ersetze sie durch die linke Seite der Regel (*reduziere*)

Shift-Reduce-Algorithmus

Ablauf beim Shift-Reduce-Parsing

I. **Shift**: Verschiebe ein Wort von der Eingabekette auf den Stapel

II. **Reduce**: Reduziere den Stapel so lange mit Hilfe der lexikalischen und syntaktischen Regeln, bis keine weiteren Reduktionen mehr möglich sind.

III. Sind noch mehr Wörter in der Eingabekette?
  - ja: Gehe zum Schritt I.
  - nein: Stop.

Das Resultat der syntaktischen Analyse befindet sich auf dem Stapel.

Implementierungstechniken I

Implementation des Stapels und der Eingabekette

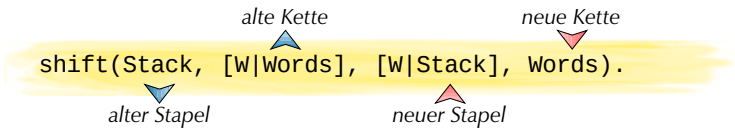
- Stapel als Liste darstellen
  - Notationswechsel: Als Liste wächst Stapel nach links: *Det man* ≡ `[man,det]`
- Eingabekette ebenfalls als Liste (Stapel) darstellen

Schritt	Aktion	Stapel	Eingabekette
0	—	`[]`	`[the,man,sleeps]`
1	shift	`[the]`	`[man,sleeps]`
2	reduce	`[det]`	`[man, sleeps]`
3	shift	`[man,det]`	`[sleeps]`
...	...	...	...
9	reduce	`[s]`	`[]`

shift/4

Ein einzelner Shift-Schritt

- nimmt das erste Wort von der Eingabekette weg (*pop*)
- setzt es zuoberst auf den Stapel (*push*)




```
?- shift([det], [man,sleeps], NewStack, NewString).
NewStack = [man,det],
NewString = [sleeps]
```

Schritt	Aktion	Stapel	Eingabekette
2		[det]	[man,sleeps]
3	shift	[man,det]	[seeps]

Implementierungstechniken II

Effiziente Implementierung der Grammatikregeln

- RHS wird "rückwärts" als offene Liste notiert, damit sie mit Stapel unifiziert! (***b**ackward **rule***)

NP → Det N ←

```
brule([n,det|X], [np|X]).
```

Aktion	Stapel
reduce	*Det N*
reduce	*NP*

Aktion	Stapel
reduce	[n,det]
reduce	[np]

- Aufruf der Regeln reduziert Stapel!

```
?- brule([n,det|Rest], NewStack).
NewStack = [np|Rest]
```


Syntax und Lexikon

Syntaktische Regeln

```
brule([vp,np|X], [s|X]).
brule([n,det|X], [np|X]).
brule([v|X], [vp|X]).
```

S → NP VP
NP → Det N
VP → V

- Lexikonregel

```
brule([Word|X], [Cat|X]) :-
   word(Word, Cat).
```

Lexikon

```
word(a,det).
word(man, n).
word(sleeps, v).
```

Det → a
N → man
V → sleeps

Reduktion: reduce/2

Rekursionsschritt

- Reduziere den Stapel so oft mit einer passenden Grammatikregel, wie es geht.

```
reduce(Stack, ReducedStack) :-
   brule(Stack, Stack2),
   reduce(Stack2, ReducedStack).
```

Abbruchbedingung

- Wenn keine Regel passt, lass den Stapel unverändert.

```
reduce(Stack, Stack).
```

"Catch-All"-Klausel

reduce/2 berechnet die transitiv-reflexive Hülle der brule-Relation!

Parsen mit shift_reduce/3

Abbruchbedingung

- Eingabekette ist leer

```
shift_reduce([], Stack, Stack).
```

Rekursionsschritt

- führt einen einzelnen Shift-Schritt mit shift/2 durch
- benutzt reduce/2, um den Stapel so weit wie möglich zu reduzieren

```
shift_reduce(String, Stack, Result) :-
   shift(Stack, String, NewStack, NewString),
   reduce(NewStack, ReducedStack),
   shift_reduce(NewString, ReducedStack, Result).
```


Problematische Regeln

Terminierungsprobleme von Bottom-Up-Verfahren wie dem Shift-Reduce-Parsing

- bei **Tilgungsregeln** $X \to \varepsilon$

```
brule(Rest, [x|Rest]).
```

  - Regel kann immer angewendet werden
  - Keller wächst immer weiter

- bei **zyklischen Regeln** $A \to B$, $B \to A$

```
brule([b|Rest], [a|Rest]).
brule([a|Rest], [b|Rest]).
```

  - Regeln können zyklisch folgend immer wieder angewendet werden
  - der Keller wird nie kleiner (allenfalls bleibt er immer gleich gross)

Mengenprädikate

Übersicht

- Lösungsmengen als Daten
- Alle Lösungen für ein Ziel erhalten (*all solutions*)
  - findall/3 – Ohne Backtracking und Variablenbindung
  - Mit Backtracking auf ungebundene Variablen
    - bagof/3 – Lösungsliste kann Duplikate enthalten
    - setof/3 – Lösungsliste ist Menge
  - Explizites Binden von Variablen durch Existenzquantor ^/2
- Anwendungen von Mengenprädikaten

findall/3 – Lösungslisten finden

```
word(cat, n).
word(ring, n).
word(do, v).
word(ring, v).
```

```
?- findall(Word, word(Word, Cat), Words).
Words = [cat,ring,do,ring] ? ;
no
```

Finde alle Wörter, egal von welcher Kategorie.

Das eingebaute Prädikat findall/3

- berechnet alle Lösungen eines Ziels.
- liefert gewünschte Teile daraus als Elemente einer Liste zurück.
  - Die Reihenfolge der Elemente entspricht der Reihenfolge, in der die Lösungen gefunden werden.
- ist wichtig, wenn alle Lösungen als Ganzes weiter verarbeitet werden sollen.

findall/3

```
?- findall(Word/Cat,
           (word(Word,Cat), word(Word,Cat2), Cat \== Cat2),
           Ambig).
Ambig = [ring/n,ring/v] ? ;
no
```

Finde alle kategoriell ambigen Wörter mit ihren möglichen Kategorien.

findall(*Term*, *Ziel*, *Liste*)

- *Term* – wird für jede Lösung von *Ziel* zu *Liste* hinzugefügt
- *Ziel* – Ziel, das zu beweisen ist
- *Liste* – enthält für jede Lösung von *Ziel* die entsprechende **Instanz** von *Term*

findall/3 – Kontrollverhalten

```
?- findall(X, fail, Resultat).
Resultat = [] ? ;
no
```

Kontrollverhalten

- falls das *Ziel* einfach fehlschlägt, ist die *Liste* leer
- freie Variablen in *Term* und *Ziel* werden **nie** gebunden!
- findall/3 terminiert nur, wenn der Suchbaum von *Ziel* endlich ist
- findall/3 gelingt höchstens einmal
- findall/3 kann nur scheitern, wenn die *Liste* instantiiert aufgerufen wird!

```
?- findall(Cat, word(Word, Cat), [n,v,a]).
no
```

bagof/3

```
word(cat, n).
word(ring, n).
word(do, v).
word(ring, v).
```

```
?- bagof(Word, word(Word,Cat), Words).
Cat = n, Words = [cat,ring] ? ;
Cat = v, Words = [do,ring] ? ;
no
```

Finde für jede Kategorie alle Wörter.

Das eingebaute Prädikat bagof(*Term*, *Ziel*, *Liste*) funktioniert wie findall/3, aber

- alle freien Variablen in *Ziel*, die nicht in *Term* vorkommen, werden gebunden, und *Liste* jeweils für eine unterschiedliche Bindung berechnet.
- falls *Ziel* nicht erfüllt werden kann, scheitert bagof/3.

Existenzquantor ^/2

```
word(cat, n).
word(ring, n).
word(do, v).
word(ring, v).
```

```
?- bagof(Word, Cat^word(Word,Cat), Words).
Words = [cat,ring,do,ring] ? ;
no
```

Finde alle Wörter, von welcher Kategorie auch immer.

Der Existenzquantor ^ bindet freie Variablen in *Ziel*.

- ^/2 erlaubt es, die Lösungen für alle möglichen Belegungen für die Variable auf der linken Seite des Operators zu berechnen.
- ^/2 ist ein rechts-assoziativer Operator

```
?- bagof(W, C1^C2^(word(W,C1),word(W,C2),C1 \== C2), W).
```


setof/3 – sortierte Lösungsmenge

```
word(cat, n).
word(ring, n).
word(do, v).
word(ring, v).
word(nice, a).
```

```
?- setof(Cat, Word^word(Word,Cat), Cats).
Cats = [a,n,v] ? ;
no
```

Finde die Menge aller Kategorien von welchen Wörtern auch immer.

Das eingebaute Prädikat setof(Term, Ziel, Liste) funktioniert wie bagof/3, aber

- die Liste enthält keine Duplikate.
- die Liste ist entsprechend der Standardordnung für Terme sortiert.

```
setof(Term, Ziel, Menge) :-
    bagof(Term, Ziel, Liste),
    sort(Liste, Menge).
```

Mögliche Definition von setof/3

Anwendungen von Mengenprädikaten

Mengenprädikate

- sprengen den Rahmen der Prädikatenlogik erster Stufe.
- erlauben Dinge zu berechnen, die uns bis anhin unmöglich waren.
  - Wie viele Lösungen hat eine Anfrage?
  - Verschiedene Lösungen einer Anfrage vergleichen.
- Beispiel: Welcher Anteil von Wörtern im Lexikon ist ambig?

```
ambig(Anteil) :-
   setof(W, Cat^word(W,Cat), Ws),
   length(Ws, AnzahlWoerter),
   setof(A, C1^C2(word(A,C1),word(A,C2),C1 \== C2), As),
   length(As, AnzahlAmbige),
   Anteil is AnzahlAmbige / AnzahlWoerter.
```