Übersicht

- Sprachverarbeitung ohne Syntax?
- System erkennt einzelne Stichwörter, ignoriert den Rest
  - Quelle: [Covington, 1994]
  - Beantworten von natürlichsprachlichen (englischen) Anfragen zu einer Datenbank
- Datenbank-Konzepte
- Aufbau des Programms
  - Lambda-Abstraktion und Beta-Reduktion in Prolog
  - einzelne Prädikate

Zweck

- Vorstellen des Programms; *kein Prüfungsstoff*

Stichwort-Erkennung

Viele Systeme versuchen gar nicht erst, die Struktur der Eingabesätze herauszufinden.

Stattdessen erkennen sie einige wenige Stichwörter *(keyword spotting)*.

- sehr effizient
- kein grosser Entwicklungsaufwand
- robust: Auch unerwartete/fehlerhafte/… Eingaben bringen das System nicht durcheinander

- aus linguistischer Sicht völlig unangemessen
- grosse Gefahr, dass Eingaben falsch verarbeitet werden

Beispiel: Datenbank-Zugriff

Als Beispiel für ein Keyword-Spotting-System bauen wir eine natürlich-sprachliche Schnittstelle zu einer Datenbank.

Is anyone's salary under 25000?

Which women work for our company?

Who is the boss?

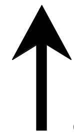
Fire the under-paid men!

Please tell me whether there is any male programmer whose age is over 40.

Relationale Datenbanken

Relationale Datenbanken bestehen aus Tabellen:

ID	Name	Age	Sex	Title	Salary
1001	Doe, John	47	m	President	100 000
1002	Smith, Mary	28	f	Programmer	60 000
1003	Zimmer, Fred	35	m	Sales Representative	55 000
1004	Smith, Mark	48	m	Programmer	15 000
1005	Russell, Ann	31	f	Programmer	17 000



Schlüssel-Attribut: Eindeutige Identifikation eines Eintrags

Relationale Datenbanken

Typischerweise umfasst eine relationale Datenbank mehrere Tabellen.

ID	Name	Age	Sex	Title	Salary
1001	Doe, John	47	m	President	100 000
1002	Smith, Mary	28	f	Programmer	60 000
1003	Zimmer, Fred	35	m	Sales Representative	55 000
1004	Smith, Mark	48	m	Programmer	15 000
1005	Russell, Ann	31	f	Programmer	17 000

Chef	Untergeben
1001	1002
1001	1003
1002	1004
1002	1005

Wir beschränken uns hier der Einfachheit halber jedoch auf eine einzige Tabelle.

Relationale Datenbanken

Die meisten relationalen Datenbanken werden über die *Abfragesprache SQL* angesteuert.

```
SELECT NAME, AGE
  FROM EMPLOYEE
  WHERE SEX = 'm'
    AND TITLE = 'Programmer'
    AND AGE > 40
```

Statt mit SQL zu arbeiten, basteln wir uns schnell eine eigene (sehr stark vereinfachte) Datenbankverwaltung in Prolog.

»Datenbank« in Prolog

Der Einfachheit halber repräsentieren wir die einzige Tabelle als Prolog-Fakten:

ID	Name	Age	Sex	Title	Salary
1001	Doe, John	47	m	President	100 000
1002	Smith, Mary	28	f	Programmer	60 000
1003	Zimmer, Fred	35	m	Sales Representative	55 000
1004	Smith, Mark	48	m	Programmer	15 000
1005	Russell, Ann	31	f	Programmer	17 000

```
employee(1001, 'Doe, John',    47, m, 'President', 100000).
employee(1002, 'Smith, Mary',  28, f, 'Programmer', 60000).
employee(1003, 'Zimmer, Fred', 35, m, 'Sales Rep',  55000).
employee(1004, 'Smith, Mark',  48, m, 'Programmer', 15000).
employee(1005, 'Russell, Ann', 31, f, 'Programmer', 17000).
```

»Datenbank« in Prolog

```
employee(1001, 'Doe, John',    47, m, 'President', 100000).
employee(1002, 'Smith, Mary',  28, f, 'Programmer', 60000).
employee(1003, 'Zimmer, Fred', 35, m, 'Sales Rep',  55000).
employee(1004, 'Smith, Mark',  48, m, 'Programmer', 15000).
employee(1005, 'Russell, Ann', 31, f, 'Programmer', 17000).
```

Die Abfrage geschieht über Prolog-Regeln:

```
salary(ID, Salary) :-
  employee(ID, _, _, _, _, Salary).
```

```
age(ID, Age) :-
  employee(ID, _, Age, _, _, _).
```

»Datenbank« in Prolog

```
:- dynamic(employee/6).
```

Das 6stellige employee kann *während des Programmablaufs* verändert werden.

```
employee(1001, 'Doe, John',    47, m, 'President', 100000).
employee(1002, 'Smith, Mary',  28, f, 'Programmer', 60000).
employee(1003, 'Zimmer, Fred', 35, m, 'Sales Rep',  55000).
employee(1004, 'Smith, Mark',  48, m, 'Programmer', 15000).
employee(1005, 'Russell, Ann', 31, f, 'Programmer', 17000).
```

Einträge werden mittels *retract* gelöscht:

```
retract(employee(1003,_,_,_,_,_)).
```

In unserem Beispiel werden nie Einträge geändert oder neu eingefügt.

Arbeitsweise

Please tell me whether there is any male programmer whose age is over 40.

Die Wörter lassen sich in verschiedene Kategorien unterteilen:

- Aktionen — *show, delete, fire, tell*
- Eigenschaften/Tests — *male, woman, underpaid, programmer*
- Vergleiche — *over, under*
- Argumente für Vergleiche — *salary, age, 30000*
- Unbekannte Wörter — *please, me, whether, elephant*

Arbeitsweise

Please tell me whether there is any male programmer whose age is over 40.

Tokenizer

```
[please,tell,me,whether,there,
 is,any,male,programmer,whose,
 age,is,over,40,'.']
```

filter

```
[action(display_records),
 test(A^sex(A,m)),
 test(B^title(B,Programmer)),
 argument(C^D^age(C,D)),
 comparison(E^F^(E>F)),
 argument(_^G^(G=40))]
```

translate

```
sex(X,m),
title(X,'Programmer'),
age(X,Y), Z = 40, Y > Z,
employed(X)
```

findall

```
[1004,1009]
```

```
[display_records]
```

=..

```
display_records(
  [1004,1009])
```

Arbeitsweise

Das Programm:

- bittet den Benutzer, eine Anfrage zu stellen
- liest die Benutzereingabe ein und zerlegt sie in einzelne Tokens
- filtert alle unbekannten Wörter heraus
  $\Rightarrow$ »Bedeutung« der bekannten Stichwörter
- geht die Liste der »Bedeutungen« durch und bestimmt
  - welchen Bedingungen ein X genügen muss, um zur Lösungsmenge zu gehören
  - welche Aktionen auf diese X angewandt werden
- führt die Aktionen für die Lösungsmenge aus
- testet, ob die Eingabe »quit« war
  - nein: Backtracking bis ganz an den Anfang
  - ja: Ende

Arbeitsweise

Unbekannte Wörter werden einfach ignoriert.

- Das ist nicht unproblematisch:
  “Paul is happy“ ≠ “Paul is not happy”

Bekannten Wörtern wird eine Bedeutung zugewiesen.

- Was *ist* die Bedeutung eines Wortes?
- Was ist mit Mehrdeutigkeiten?

Die Struktur des Satzes wird völlig ignoriert.

Programm-Rumpf

```
:- ensure_loaded(tokenizer).

process_queries :-
   repeat,
      write('Query: '),
      read_atomics(Words),
      process_query(Words),
   Words == [quit],
   !.

process_query(Query) :-
   filter(Query, Filtered),
   translate(Filtered, X, TestsOnX, Actions),
   findall(X, TestsOnX, MatchingEmployees),
   execute_actions(Actions, MatchingEmployees).
```

Lädt das Programm in tokenizer.pl; die dort definierten Prädikate sind danach benutzbar.

Konstruktion der Abfrage

Please tell me whether there is any male programmer whose age is over 40.

Welche Tupel sollen nun angezeigt werden?

- die männlichen Programmierer, deren Alter über 40 liegt
- jene X
  - deren Geschlecht *männlich* ist
  - deren Titel *Programmierer* ist
  - und deren Alter Y grösser als 40 ist
-
```
sex(X,m), title(X,programmer),
age(X,Y), Z = 40, Y > Z
```

Lambda-Abstraktion

Im Lambda-Kalkül können Formeln in Eigenschaften umgewandelt werden.

AUSSAGE

sterblich(fido)

Fido ist sterblich

sterblich(x)

X ist sterblich

Abstraktion von einem konkreten Objekt →

EIGENSCHAFT

λx sterblich(x)

die Eigenschaft, sterblich zu sein

Lambda-Abstraktion

Weil ein Lambda-Ausdruck als Funktion interpretiert wird, kann er ein Argument erhalten und liefert ein Ergebnis:

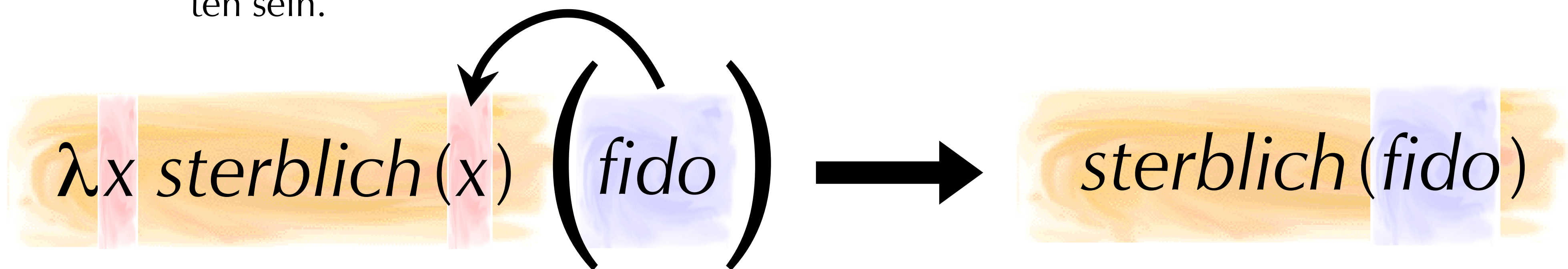
$[\![\lambda x \; sterblich(x)]\!] = \{ \langle fido, wahr \rangle, \langle limmat, falsch \rangle, \langle sokrates, wahr \rangle \}$

$[\![(\lambda x \; sterblich(x))(fido)]\!] = wahr$

Beta-Reduktion

Das Argument kann auch rein syntaktisch eingesetzt werden (sog. Beta-Reduktion):

- Jedes Vorkommen der abstrahierten Variable (hier: *x*) wird durch das Argument (hier: *fido*) ersetzt.
- Beide Ausdrücke werden gleich interpretiert.
  - Denn die Interpretation von Lambda-Ausdrücken ist gerade so definiert, dass dieser Konversionsschritt möglich wird.
  - Die abstrahierte Variable (hier: *x*) darf jedoch nicht im Argument (hier: *fido*) enthalten sein.



Lambda-Abstraktion in Prolog

Mögliche Darstellungen in Prolog:

- Prolog kennt (anders als z.B. Lisp) keine Lambda-Ausdrücke als Sprachelemente ⇒ jede andere Darstellung ginge genausogut
- Die Funktoren könnten auch einen anderen Namen tragen

$$\lambda x\ sterblich(x)$$

darzustellender λ-Ausdruck

```
lambda(X, sterblich(X))
```

Darstellung mit Funktor »lambda«

```
X^sterblich(X)
```

Darstellung mit Infix-Operator »^«

Beta-Reduktion in Prolog

Die Beta-Reduktion kann in Prolog durch das Unifizieren von Termen simuliert werden:

- Die Simulation stimmt nicht ganz immer; dies würde jedoch hier zu weit führen.

```
X^sterblich(X)        fido^Ergebnis
```

X / fido
Ergebnis / sterblich(fido)

```
fido^sterblich(fido)
```

»Bedeutung« der Stichwörter

Wir notieren die »Bedeutung« der Stichwörter als Lambda-Ausdrücke:

- Aktionen

```
keyword(show, action(display_records))
```

- Tests

```
keyword(man, test(X^sex(X, m)))
```

- Vergleiche

```
keyword(under, comparison(X^Y^(X < Y)))
```

- Argumente für Vergleiche

```
keyword(salary, argument(X^Y^salary(X, Y)))
```

Arbeitsweise: process_query/0

process_query/0

- Ruft Tokenizer auf
- Ruft Hilfsprädikat `process_query/1` auf
  (für Fehlersuche: `process_query_chattering/1`)
  - Herausfiltern unbekannter Wörter — `filter/2`
  - Übersetzen der »Bedeutungen« in auführbare Prolog-Terme — `translate/4`
    - → Liste mit Tests, welche *X* erfüllen muss, um zur Lösungsmenge zu gehören
    - → Liste der Aktionen, die auf die Lösungsmenge angewandt werden sollen
  - Bestimmen der Lösungsmenge — `findall/3`
  - Anwenden der Aktionen auf Lösungsmenge — `execute_query/2`
- Nochmals von vorne, ausser die Eingabe war »quit«

Arbeitsweise: filter/2

Das Prädikat filter/2 besteht aus Klauseln für:

- Abbruchbedingung
- Bekanntes Stichwort
  - »Bedeutung« nachschlagen (mit `keyword/2`) und zurückliefern
  - rekursiver Aufruf
- Synonym eines bekannten Stichworts
  - Das synonyme Stichwort herausfinden (mit synonym/2)
  - Dessen »Bedeutung« nachschlagen (mit `keyword/2`) und zurückliefern
  - rekursiver Aufruf
- Unbekanntes Wort
  - rekursiver Aufruf (d.h. das unbekannte Wort wird einfach ignoriert)

Arbeitsweise: translate/4

Das Prädikat translate/4 besteht aus Klauseln für:

- Abbruchbedingung
- Aktion
  - zur Liste der auszuführenden Aktionen hinzufügen, z.B. `display_records`
  - rekursiver Aufruf
- test(λv Test)
  - beta-reduzierten Test zur Liste der Bedingungen hinzufügen, z.B. `underpaid(X)`
  - rekursiver Aufruf
- argument, comparison, argument
  - mehrere Beta-Reduktions-Schritte
  - zur Liste der Bedinungen hinzufügen, z.B. `salary(X, Y), Z=30000, Y < Z`
  - rekursiver Aufruf

Arbeitsweise: execute_query/2

Das Prädikat execute_query/2:

- für eine leere Liste von Aktionen: `display_records`
  - Wenn keine Aktion vom System erkannt wurde, ist Anzeigen wohl am ehesten gemeint
  - Aussderm ist dies sicherer als »im Zweifelsfall entlassen«
- für eine nicht-leere Liste von Aktionen:
  - Für jede Aktion: Aufruf eines passenden Prädikats mit der Lösungsmenge als einzigem Argument
  - Dadurch werden Eingaben mit mehreren Aktionen korrekt verarbeitet, z.B. *Please display all my programmers, and fire them afterwards.*

keyword.pl

Programmiertechniken der Computerlinguistik 2 · Sommersemester 1999
Online unter http://www.coli.uni-sb.de/~brawer/prolog/keyword/

```
% ============================================================
% keyword.pl
%
% A keyword-based natural-language interface to a database
% with employees in a company.
%
% Usage:
%  1. ?- [keyword].              % consult this file
%  2. ?- process_queries.        % start
%  3. who works as a programmer? % enter your queries
%     fire all underpaid men     % etc.
%  4. quit                       % stop by entering "quit"
%
% Taken from [Covington, 1994], but with large modifications.
% ============================================================

% Ensure that the tokenizer is loaded.
:- ensure_loaded('tokenizer.pl').

% ------------------------------------------------------------
% process_queries
% ------------------------------------------------------------
% Asks the user to enter a query and processes the entered
% query.

process_queries :-
   repeat,
     write('Query: '),
     read_atomics(Words),
     process_query(Words),
   Words == [quit],
   !.

% ------------------------------------------------------------
% process_query(+Query)
% ------------------------------------------------------------
% Processes a single query. Call process_query_chattering in
% order to generate output which can be useful for debugging.
%
% Query: A list of tokens, e.g. [tell,me,who,is,the,boss,'.']

process_query(Query) :-
   filter(Query, Filtered),
   translate(Filtered, X, TestsOnX, Actions),
   findall(X, TestsOnX, MatchingEmployees),
   execute_query(Actions, MatchingEmployees).
```

```
process_query_chattering(Query) :-
   filter(Query, Filtered),
   write('Filtered: '), write(Filtered), nl,
   translate(Filtered, X, TestsOnX, Actions),
   write('Tests on '), write(X), write(': '), write(TestsOnX), nl,
   write('Actions: '), write(Actions), nl,
   findall(X, TestsOnX, MatchingEmployees),
   write('Matching Employees: '), write(MatchingEmployees),
   execute_query(Actions, MatchingEmployees).


% ------------------------------------------------------------
% execute_query(+Actions, +Employees)
% ------------------------------------------------------------
% If one or more action keywords were found, execute these
% actions by calling execute_actions/2. Otherwise, display
% the matching employees.
%
% Actions:   A list of actions, e.g. [display_records]
% Employees: A list with the IDs of matching employees,
%            for instance [1001, 1004, 1009]

execute_query([], Employees) :-
   display_records(Employees), !.

execute_query(Actions, Employees) :-
   execute_actions(Actions, Employees).


% ------------------------------------------------------------
% execute_actions(+Actions, +Employees)
% ------------------------------------------------------------
% For each element of the Actions list, a term is constructed
% which has that element as its functor and the entire Employee
% list as its single argument.  Then, call is used to invoke
% that predicate.
%
% Example: execute_actions([display_records, remove_records],
%                          [1001,1004,1009])
% call -----> display_records([1001,1004,1009])
% call -----> remove_records([1001,1004,1009])

execute_actions([], _).
execute_actions([Action|Actions], Employees) :-
   Command =.. [Action, Employees],
   call(Command),
   execute_actions(Actions, Employees).
```

```
% ------------------------------------------------------------
% translate(+KeywordSemantics, ?Variable, -Tests, -Actions)
% ------------------------------------------------------------
% Translates a list of keyword semantics into a number of
% tests over Variable and into a number of actions to be
% performed on the employees which pass the tests.
%
% KeywordSemantics: List of keyword semantics
% Tests:            Single Prolog term which can be passed
%                   to findall in order to determine the
%                   matching employees.
% Actions:          Actions to be performed on the set of
%                   matching employees.

translate([], X, employed(X), []).

translate([action(Action) | Items], X, Tests, [Action|Actions]) :-
   translate(Items, X, Tests, Actions).

translate([test(X^Test) | Items], X, (Test,Tests), Actions) :-
   translate(Items, X, Tests, Actions).

translate([argument(X^Y^T1),
     comparison(Y^Z^T3),
     argument(X^Z^T2) | Items],
     X, (T1, T2, T3,Tests), Actions) :-
   translate(Items, X, Tests, Actions).


% ------------------------------------------------------------
% filter(+Token, -KeywordSemanticsList)
% ------------------------------------------------------------
% Takes a list of tokens and extracts those known as keyword
% (or as a synonym of a keyword).  The result is a list of
% the "semantics" of the recognized keywords.

% Termination
filter([], []).

% Known keyword
filter([Word|Words], [Translation|Translations]) :-
   keyword(Word, Translation),
   !,
   filter(Words, Translations).

% Synonym
filter([Word|Words], [Translation|Translations]) :-
   synonym(Word, Synonym),
   keyword(Synonym, Translation),
   !,
   filter(Words, Translations).

% Unknown word
filter([_|Words], Translations) :-
   filter(Words, Translations).
```

```
% ------------------------------------------------------------
% keyword(+Keyword, -Semantics)
% ------------------------------------------------------------
% Associates keywords with their semantics.

keyword(show, action(display_records)).
keyword(delete, action(remove_records)).
keyword(man, test(X^sex(X, m))).
keyword(woman, test(X^sex(X, f))).
keyword(president, test(X^title(X, 'President'))).
keyword(programmer, test(X^title(X, 'Programmer'))).
keyword(salesrep, test(X^title(X,'Sales Representative'))).
keyword(employee, test(X^title(X, _))).
keyword(underpaid, test(X^underpaid(X))).
keyword(friendly, test(X^friendly(X))).
keyword(unfriendly, test(X^unfriendly(X))).
keyword(over, comparison(Y^Z^(Y>Z))).
keyword(under, comparison(Y^Z^(Y<Z))).
keyword(salary, argument(X^Y^salary(X, Y))).
keyword(age, argument(X^Y^age(X, Y))).
keyword(N, argument(_^Y^(Y=N))) :- number(N).


% ------------------------------------------------------------
% synonym(+Synonym, -Keyword)
% ------------------------------------------------------------
% Associates synonyms with synonymous keywords.

synonym(boss, president).
synonym(programmers, programmer).
synonym(men, man).
synonym(male,man).
synonym(guy, man).
synonym(women, woman).
synonym(employees, employee).
synonym(female,woman).
synonym(display, show).
synonym(present, show).
synonym(give, show).
synonym(tell, show).
synonym(remove, delete).
synonym(fire, delete).
```

```
% -----------------------------------------------------------
% display_records(+EmployeeIDList)
% -----------------------------------------------------------
% Displays employees to the user.
%
% EmployeeIDList: A list of employee IDs, e.g. [1001,1004]

display_records([]).
display_records([ID|IDs]) :-
   write(ID), write(': '),
   full_name(ID, Name), write(Name),
   write(', '), title(ID, Title), write(Title),
   write(', salary '), salary(ID, Salary), write(Salary),
   write('.'), nl,
   display_records(IDs).


% -----------------------------------------------------------
% remove_records(+EmployeeIDList)
% -----------------------------------------------------------
% Removes employees from the database.
%
% EmployeeIDList: A list of employee IDs, e.g. [1001,1004]

remove_records([]).
remove_records([ID|IDs]) :-
   retract(employee(ID, _, _, _, _, _, _)),
   remove_records(IDs).


% -----------------------------------------------------------
% Database Predicates
% -----------------------------------------------------------
% Represents the database as Prolog clauses.  A real system
% would access an external database management system, e.g.
% by passing SQL queries to a DB server.

:- dynamic(employee/7).

employee(1001, 'Doe, John',    47, m, friendly,   'President', 100000).
employee(1002, 'Smith, Mary',  28, f, friendly,   'Programmer', 60000).
employee(1003, 'Zimmer, Fred', 35, m, unfriendly, 'Sales Representative',
                                                              55000).
employee(1004, 'Smith, Mark',  48, m, friendly,   'Programmer', 15000).
employee(1005, 'Russel, Ann',  31, f, unfriendly, 'Programmer', 17000).


full_name(ID, Name) :- employee(ID, Name, _, _, _, _, _).
age(ID, Age) :- employee(ID, _, Age, _, _, _, _).
sex(ID, Sex) :- employee(ID, _, _, Sex, _, _, _).
friendly(ID) :- employee(ID, _, _, _, friendly, _, _).
unfriendly(ID) :- employee(ID, _, _, _, unfriendly, _, _).
title(ID, Title) :- employee(ID, _, _,_,_, Title, _).
salary(ID, Salary) :- employee(ID, _, _,_,_, _, Salary).
underpaid(ID) :- salary(ID, Salary), Salary < 40000.
employed(ID) :- employee(ID, _,_,_,_,_,_).
```