# Multi-Agent Systems

Albert-Ludwigs-Universität Freiburg

Bernhard Nebel, Felix Lindner, and Thorsten Engesser

Winter Term 2018/19

## Logical Programming for BDI-Agents: Motivation

- Basic Assumption: Agents maintain mental states (knowledge, belief, desires, intentions) in some knowledge base.
- Knowledge Base: Is a set of formulae written in some formal language.
- Requirement: A computational system for maintaining and querying a knowledge base.
- GOAL employs Prolog for that pupose.



## A Simple Prolog Program

- The simplest Prolog programs are just a list of facts:

```
student(eva).
student(hans).
subject(eva, cs).
subject(hans, phil).
```



## A Simple Prolog Program

**Prolog program**

```
student(eva).
student(hans).
subject(eva, cs).
subject(hans, phil).
```

**Queries**

```
?- student(eva).
yes
?- student(christian).
no
?- subject(eva, cs).
yes
?- professor(hugo).
ERROR: Undefined procedure:
professor/1
```

# A Simple Prolog Program

**Prolog program**

```
student(eva).
student(hans).
subject(eva, cs).
subject(hans, phil).
```

**Queries**

```
?- student(X).
X = eva ;
X = hans ;
no
?- subject(X, cs).
X = eva;
no
?- subject(X, X).
no
```



# Prolog Syntax: Atoms

- Atoms
  - Terms that consists of letters, numbers, and the underscore, and which start with a non-capital letters: eva, cs, dr_who, hal2000
  - Terms that are enclosed in single quotes: 'President Trump', '(@*+'
  - Certain special symbols like +, , , :-



# Prolog Syntax: Variables

- Variables
  - Terms that consist of letters, numbers, and the underscore, and which start with a capital letter or an underscore: X, Prof, _x
  - _ is an anonymous variable: two occurences of _ are different variables
    - Program: p(a, a). Queries: ?- p(X, X). vs. ?- p(_, _).



# Prolog Syntax: Complex Terms

- Complex Terms
  - Terms of the form: functor(argument1, . . . , argumentN)
  - Functors have to be atoms
  - Arguments can be any kind of Prolog term. Examples: subject(eva, X), f(a, X, g(Y, h(Z)), c)

## Prolog Syntax: Facts

- Facts are complex terms followed by a full stop:
  student(eva). subject(hans, phil).
- Queries are also complex terms, or sequences of complex terms separeted by comma, followed by a full stop.



## Prolog Program with Rules

```
student(eva).
student(hans).
student(laura).
subject(eva, cs).
subject(hans, phil).
subject(laura, eng).
logician(X) :- subject(X, cs).
logician(X) :- subject(X, phil).
```

- :- is read as if...then... (but from right to left): If X's subject is cs, then X is a logician. Or: X is a logician, if X's subject is cs.



## Prolog Program with Rules

**Prolog program**

```
student(eva).
student(hans).
student(laura).
subject(eva, cs).
subject(hans, phil).
subject(laura, eng).
logician(X) :- subject(X, cs).
logician(X) :- subject(X, phil).
```

**Queries**

```
?- logician(eva).
yes
?- logician(laura).
no
?- logician(X).
X = eva ;
X = hans ;
no
```



## Prolog Syntax: Rules

- Rules
  - Rules are of the Form Head :- Body.
  - Like facts and queries, they have to be followed by a full stop.
  - Head is a compex term.
  - Body is a complex term or a sequence of complex terms separated by commas.

## Prolog Program with Rules

**Prolog program**

```
student(eva).
student(hans).
student(laura).
subject(eva, cs).
subject(hans, phil).
subject(laura, eng).
logician(X) :- subject(X, cs).
logician(X) :- subject(X, phil).
double_logician(X) :- subject(X, cs), subject(X, phil).
```

**Queries**

```
?- double_logician(X).
no
?- student(X), subject(X, eng).
yes
```



## Proof Search: Matching

- Two atoms match if they are the same: *eva* = *eva*, *eva* \= *laura*
- A variable matches any other term. The variable then gets instantiated with that term.
- Two complex terms match if they have the same functor of equal arity and if all pairs of arguments in the same position match.
  - Match: subject(X, cs) = subject(eva, cs)
  - No Match: subject(eva, cs) = subject(X, X)



## Proof Search: Example I

```
student(eva).
student(hans).
subject(eva, cs).
subject(eva, phil).
subject(hans, phil).
logician(X) :- subject(X, cs).
logician(X) :- subject(X, phil).
double_logician(X) :- subject(X, cs), subject(X, phil).
```

- Query: ?- student(X).
  - Prolog checks for facts that match the query starting from the top of the knowledge base (yep, order matters).
  - The procedure finds two matching facts. Typing ; forces Prolog to search for more possibilities.



## Proof Search: Example II

```
student(eva).
student(hans).
subject(eva, cs).
subject(eva, phil).
subject(hans, phil).
logician(X) :- subject(X, cs).
logician(X) :- subject(X, phil).
double_logician(X) :- subject(X, cs), subject(X, phil).
```

- Query: ?- double_logician(X).
  - Matches with double_logician(X) :- subject(X, cs), subject(X, phil).
  - What if the two subgoals in the body changed position?

## Cut !

- ! is a goal that always succeeds and which blocks backtracking. Compare ?- double_logician(X). for these two programs:

```
student(eva).
student(hans).
subject(hans, cs).
subject(eva, cs).
subject(eva, phil).
double_logician(X) :- subject(X, cs), !, subject(X, phil).
```

```
student(eva).
student(hans).
subject(eva, cs).
subject(hans, cs).
subject(eva, phil).
double_logician(X) :- subject(X, cs), !, subject(X, phil).
```



## Some more Notes

- Only positive facts are allowed, and no disjunctive facts.
- Negation is allowed in the body of a rule.
  - Negation as Failure: the goal not(b) is true if b cannot be proven true.
  - Thus the program a :- not(b). means: if b cannot be proven, then a is true.
- Disjunction is allowed in the body of a rule.
  - Program a :- b;c. is equal to a :- b. a :- c.
- Proves may not terminate: a :- b. b :- a.
- Prolog has inbuilt arithemtics: X is 1, Y is X + 3.
- Prolog lacks a model-theoretic semantics, often feels rather procedural, is a Turing-complete programming language.



## Lists

- Prolog comes with a very powerful mechanism for list processing.
- Lists are a special kind of Prolog terms.
- The empty list: []
- Non-empty list: .(Head, Tail)
  - Head is an atom, a variable, a complex term, a number, or a list
  - Tail is either the empty list or a non-empty list of the form .(Head, Tail)



## Lists: Examples

- .(a, []): List with one element a
- .(a, .(b, [])): List with two elements a, b
- .(.(a, []), .(b, [])): List with two elements: First being the singleton list containing a, the other one being the singleton list containing b

## Lists: Notation

- .(a, Tail) = [a | Tail]
- .(a, .(b, Tail)) = [a, b | Tail]
- .(a, .(b, .(c, []))) = [a, b, c]
- .(.(a, []), .(b, [])) = [[a], [b]]



## Lists: Example I

```
trans([], []).
trans([a | T1], [b | T2]) :- trans(T1, T2).
```

- ?- trans([a, a], X).
- –> Proof tree at the blackboard.
- Works in both directions!



## Lists: Example II

```
element_of(X, [X | Tail]).
element_of(X, [_ | Tail]) :- element_of(X, Tail).
```

- ?- element_of(b, [a, b, c]).
- In SWI-Prolog you can also use the inbuilt predicate member/2.
- Try: Get all numbers smaller than 5.
- Try: Get all lists, of which 3 is a member.
- Try: Is 7 a member of a given list?



## Final Practical Notes

- SWI-Prolog: `http://www.swi-prolog.org`
- Start program from command line: swipl -s <file> -g <query> -t halt.
- Collecting all answers to a query in a list: Findall/3.
- Output to command line: writeln/1