Prolog Tutorial

Kostiantyn Kucher

Department of Computer Science and Media Technology,
Linnaeus University

Table of Contents

Motivation

- Can we apply *predicate logic* for some software development tasks?
  - Model checking
  - Expert systems
  - Planning
  - AI in games
- We could either rely on third-party libraries for general-purpose languages such as Java or Python...
- Or explore the possibility of supporting these concepts natively at the level of the programming language $\Rightarrow$ enter *Prolog*.

A Brief History of Prolog

- Prolog ("programming in logic") was designed in France in early 1970s.
- Many implementations are available, and the recent ones provide integration with other languages and support for tasks such as web development.
- While many dialects of Prolog exist, we will use *SWI-Prolog*.
    - URL: `http://www.swi-prolog.org/download/stable`
    - Reference manual for version 8.0.2 includes 619 pages!
    - Online version (*SWISH*): `https://swish.swi-prolog.org/`

Main Features of Prolog

- Prolog follows the *logic programming* paradigm:
    - Typically, a program formulates a number of facts and relations about some problem.
    - Based on these facts and relations, queries can be answered using formal logic proofs.
- Logic programming can be considered a subclass of the *declarative programming* paradigm, which focuses on describing the problem (*what*) rather than the control flow and exact computational operations (*how*).
- However, Prolog is not limited to purely declarative interpretation and it can be useful in various applications.
    - In addition to the notation and examples in this tutorial and the assignment, it is possible, for instance, to use an alternative Prolog notation (*definite clause grammars*) to develop parsers.

Table of Contents

Overall Workflow

- First, describe the *facts* and *rules* about the problem by using the building blocks discussed in the next slides ("database" or "knowledge base" in Prolog terminology).
  - **Note:** the order of definitions in the knowledge base matters!
- Then formulate queries with one or several *goals* which Prolog will evaluate.
- In some cases, if several solutions are available (e.g., with several possible variable values), Prolog can enumerate them until it runs out of valid solutions.
  - **Note:** the order of goals in the queries matters!

- *Closed world assumption*: if some data is not provided in the program and Prolog cannot prove it based on the program, it will consider it to be **false**.

Editing and Executing Programs

- For regular programs:
  1. Type your program source code in some text editor and save it in a file (traditionally, the extension is *.pl*).
  2. Open/load the program from Prolog by using the `consult` predicate (command).
  3. Pose the queries.
  4. In SWI-Prolog, you can use the `make` predicate (invoke it as "`make.`") to reload your program on the fly if you have edited it.
- **Note:** *clauses* (statements) are ended with periods.
- **Also:** capitalization is important in Prolog!
- **Also:** Prolog *predicates* typically refer to built-in commands, and the number of arguments (*arity*) is added after a slash, e.g., `make/0`, `consult/1`

Editing and Executing Programs (cont.)

Example: a source code file at ~/prolog-files/helloworld.pl

```
hello(world).  % A comment here
/*
A multiline comment here
*/
```

Example: an SWI-Prolog session

```
?- consult("~/prolog-files/helloworld.pl").
true.
?- hello(world).
true.
?- hello(anotherworld).
false.
```

Editing and Executing Programs (cont.)

- For entering statements directly in Prolog prompt:
  - Type `[user].` to read statements from standard input, and finish with Ctrl-D.

Example

```
?- [user].
|:  good(coffee).
|:  % Ctrl-D pressed here!
% user://2 compiled 0.00 sec, 1 clauses
true.
?- good(coffee).
true.
```

- For SWISH:
  - Create and edit a program (on the left) and run queries from a console (on the bottom right).

Data Types / Terms

- The basic data type in Prolog is a *term* which in general is expressed in form `name(arg1, arg2, ...)`.
- A term with zero arguments is called an *atom*.
  - Think of the atoms as constants in predicate logic, or enumerated keywords from C or Java (also, as string keywords).
  - Should start with a lower case letter, or could be quoted in case of strings with spaces.
  - Examples: `x`, `'A string atom'`
- Integer and float *numbers* are also considered to be terms.
- *Variables* are similar to variables in predicate logic.
  - **Note:** in Prolog, variable names should start with an uppercase letter!
  - Examples: `X`, `My_Variable111`
  - _ (underscore) is a special anonymous or “don't-care” variable matching anything (see more about variable binding below).

Data Types / Terms (cont.)

- *Compound terms* consist of a *functor* atom and a number of *argument* terms.
  - Example: `date(2019, 05, 20)`
  - Note that the example above (which defines a data *structure*) can also be interpreted as a *relation* or a *predicate*.
  - *Lists* are also compound terms, for example: `[]`, `[1, 2]`, `[head | rest_list]`

- Prolog provides a number of tests for various term types: `atom(X)`, `number(X)`, `compound(X)`, `var(X)` (for uninstantiated variables), `nonvar(X)` (instantiated variables, or not variables at all), etc.

Facts

- By specifying predicates (and ending the lines with periods to form proper clauses), we can add *facts* to our knowledge base in Prolog.

Example: facts

```
hello(world).
mix_well(coffee, sugar).
mix_well(coffee, milk).
mix_well(tea, milk).
```

Queries and Variable Binding

- If we run queries over the knowledge base, Prolog will try to *unify* (match) the query predicate with each fact.
- If the query includes a variable, Prolog will try to *bind* this variable to the corresponding value.

Example: given the mix_well facts from the previous slide. . .

```
?- mix_well(coffee, X).
X = sugar  % After this, I have pressed Enter, and the line
was ended with a period.
% This time, press ';' on prompt, and Prolog will output
further solutions
?- mix_well(coffee, X).
X = sugar
X = milk.
```

Queries and Variable Binding (cont.)

Example (cont.)

```
?- mix_well(Y, Z).
Y = coffee,
Z = sugar
Y = coffee,
Z = milk
Y = tea,
Z = milk.
```

Queries and Variable Binding (cont.)

Example (cont.)

```
?- mix_well(X, X).
false.
?- mix_well(_, sugar).
true.
?- mix_well(_, _).
true.
```

- Remember that _ matches anything, and several _ do **not** have to refer to the same value!

Goals and Backtracking

- Prolog supports the *AND* and *OR* connectives as a comma and a semicolon, respectively.
- We can use them (as well as parentheses) to specify more than one *goal* for the query.

Example (cont.)

```
?- mix_well(X, sugar), mix_well(X, milk).
X = coffee.
?- mix_well(coffee, X); mix_well(tea, sugar); mix_well(Y,
milk).
X = sugar % I have pressed ';' several times until Prolog
% ran out of solutions
X = milk
Y = coffee
Y = tea.
```

Goals and Backtracking (cont.)

- If the variable bindings do not satisfy the goals, Prolog *backtracks* and looks for other bindings; the same happens when additional solutions are requested with ';'.
- In the example below, note internal variables (_922 and _932) created by Prolog in the process of unification and backtracking.

Example (cont.; this time with tracing)

```
?- trace.
true.

[trace] ?- mix_well(coffee, X); mix_well(tea, sugar);
mix_well(Y, milk).
    Call:  (9) mix_well(coffee, _922) ?  creep  % 'creep'
    % indicates Enter key presses
    Exit:  (9) mix_well(coffee, sugar) ?  creep
X = sugar  % I have pressed ';' here
```

Goals and Backtracking (cont.)

Example (cont.)

```
    Redo:  (9) mix_well(coffee, _922) ?  creep
    Exit:  (9) mix_well(coffee, milk) ?  creep
X = milk  % I have pressed ';' here
    Call:  (9) mix_well(tea, sugar) ?  creep
    Fail:  (9) mix_well(tea, sugar) ?  creep
    Call:  (9) mix_well(_932, milk) ?  creep
    Exit:  (9) mix_well(coffee, milk) ?  creep
Y = coffee  % I have pressed ';' here
    Redo:  (9) mix_well(_932, milk) ?  creep
    Exit:  (9) mix_well(tea, milk) ?  creep
Y = tea.

[trace] ?- notrace.
true.
```

Goals and Backtracking (cont.)

- It is possible to include a special goal, `!`, in the query, which is called the *cut*.
- It prevents unwanted backtracking, which could be used, e.g., during debugging or to prevent Prolog from finding alternative solutions for efficiency purposes.

Example (cont.)

```
?- mix_well(coffee, X), mix_well(tea, X).
X = milk.
?- mix_well(coffee, X), !, mix_well(tea, X).
false.
```

- In the second query, `X` is initially bound to `sugar` according to the knowledge base, but then after the last goal `mix_well(tea, sugar)` fails, it is impossible to backtrack past the cut to the first goal, and the query fails.

Rules

- Besides facts, we can specify *rules* each consisting of a *head* and a *body*: the head is true if the body is true.
  - In Prolog, it looks like `Head :- Body`.
  - From the point of view of logic, it reads: $\texttt{Body} \vdash \texttt{Head}$
- Facts can be considered a special case of rules with no bodies.
- In contrast to facts, definitions of rules can make use of variables!

Rules (cont.)

Example: the mandatory family tree example

```
is_woman(alice).
is_woman(bridgit).
is_man(connor).
is_man(dave).
parent_of(alice, dave).
parent_of(connor, dave).
parent_of(dave, bridgit).

father_of(X, Y) :- parent_of(X, Y), is_man(X).
```

Rules (cont.)

Example (cont.)

```
?- father_of(X, dave).
X = connor.

?- father_of(X, Y).
X = connor,
Y = dave
X = dave,
Y = bridgit.
```

Math and Operators

- Prolog supports common arithmetic operations as well as some operators which can be rather confusing.
- = denotes unification, e.g., complex terms unify if they have the same functor and their arguments unify (in a recursive fashion).
- `is` forces evaluation of its right-hand side argument as an arithmetic expression, e.g., `1 + 2` would be evaluated to 3.
  - The left-hand side argument must be either a variable, or a numeric constant, not an expression!
  - All the variables in the expression (the right-hand side) must be instantiated in order to evaluate the expression, which affects the order of goals where this operator is used.
  - Attempts to use this operator to change the binding of variables such as `X is X + 1` will not succeed as expected!

Math and Operators (cont.)

Example: unification, evaluation, comparisons

```
?- X = 1+2*3.
X = 1+2*3.

?- X is 1+2*3.
X = 7.

?- 10*10 > 5*5.  % Comparison operators evaluate their
arguments
true.

?- 4*3 =:= 6+6.  % Equality check for expression values
true.
```

Recursion

- Prolog supports a powerful mechanism of *recursive* rules, where the body includes a goal which refers to the head.

Example: extensions for the family tree example

```
ancestor_of(X, Y) :- parent_of(X, Y).
ancestor_of(X, Y) :- parent_of(X, Z), ancestor_of(Z, Y).

?- ancestor_of(X, bridgit).
X = dave
X = alice
X = connor
false.
?- ancestor_of(alice, X).
X = dave
X = bridgit
false.
```

Recursion (cont.)

- Note that the output in the previous slide ended with `false.` after several proposed solutions: SWI-Prolog does that if it *cannot find* more solutions.
  - In contrast, in the previous examples the output was simply ended by a period after the last solution, since Prolog *knew* that there were no further solutions.

- When defining recursive rules, define the stop predicate (non-recursive one) first.
- If possible, try to keep the recursive predicate close to the end of the goal sequence.
  - Similar to other declarative programming languages, Prolog uses a technique called *tail call optimization* which can allow recursive computations to be comparable with iterative / loop-based approaches, but only if the recursive predicate is specified as the last goal of the rule.

Negation as Failure

- *Negation as failure*: if an exhaustive search to prove p fails, then assert $\neg p$.
- In Prolog, `\+ Goal` is true if `Goal` cannot be proven.
- **Important:** negation as failure is **not** the same as logical negation! When using it, pay special attention to the order of goals in the rule/query.

Table of Contents

Summary

- Prolog demonstrates how the concepts of predicate logic can be applied to software development by following the declarative programming paradigm.
- The workflow of Prolog consists of defining terms, facts, and rules, and formulating queries to the resulting knowledge base.
- Prolog supports a number of standard and advanced programming techniques, including recursion and complex data structures.

Development Tools

- The example for backtracking used `trace`/`notrace` functionality of the SWI-Prolog interpreter, but other tools aiding the development and debugging exist, too.
- SWI-Prolog provides a source-level graphical debugger/tracer: `http://www.swi-prolog.org/gtrace.html`
- Prolog Development Tool (PDT) is an addon for Eclipse that provides a source code editor, integration of the graphical debugger, and additional tools for gaining overview of the code: `https://sewiki.iai.uni-bonn.de/research/pdt/docs/start`

Further Reading and Resources

Software and tutorials:

- *SWI-Prolog* (and its documentation): http://www.swi-prolog.org/
- List of editor and IDE options for SWI-Prolog:

  http://www.swi-prolog.org/IDE.html

- *Prolog* at Wikibooks: https://en.wikibooks.org/wiki/Prolog
- *The Prolog Dictionary* by Bill Wilson:

  http://www.cse.unsw.edu.au/~billw/dictionaries/prolog/prologdict.html
- *Guide to Prolog Programming* by Roman Barták:

  http://kti.ms.mff.cuni.cz/~bartak/prolog/contents.html
- *Real World Programming in SWI-Prolog*: http://www.pathwayslms.com/swipltuts/

Further Reading and Resources (cont.)

Books:

- Peter Flach. *Simply Logical: Intelligent Reasoning by Example*. https://book.simply-logical.space/ (**open access!**)
- Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!* http://lpn.swi-prolog.org/lpnpage.php?pageid=online (**open access!** But the online version seems to have issues. . .)
- Ivan Bratko. *Prolog Programming for Artificial Intelligence*. http://catalogue.pearsoned.co.uk/catalog/academic/product?ISBN=9780321417466
- Leon S. Sterling and Ehud Y. Shapiro. *The Art of Prolog*. https://mitpress.mit.edu/books/art-prolog-second-edition (**open access!**)