Roland Schäfer

Prolog for Natural Language Semantics Göttingen/Summer Course 2005

References

[BB] Blackburn, P. & Bos, J. 2005. *Representation and Inference for Natural Language*. Stanford: CSLI.

[CM] Clocksin, W. F. & Mellish, Ch. M. 1995. *Programming in Prolog.* Berlin: Springer.

[Cov] Covington, M. A. 1994. *Natural Language Processing for Prolog Programmers*. Upper Saddle River: Prentice Hall.

[LPN] Blackburn,P. & Bos, J. & Striegnitz, K. 2001. *Learn Prolog Now!*. http://www.coli.uni-sb.de/~kris/learn-prolog-now.

Dates

Day 1, *Prolog Recap*
Wed, 2005/07/20
Afternoon Session @ **13.15-14.45**
Evening Session @ **16.15-17.45**

Day 2, *Model Checking*
Thu, 2005/07/21
Morning Session @ **10.15-11.45**
Afternoon Session @ **14.15-15.45**

Day 3, *Lambda*
Fri, 2005/07/22
Morning Session @ **10.15-11.45**

Day 4, *Underspecification*
Mon, 2005/07/25
Morning Session @ **10.15-11.45**
Afternoon Session @ **14.15-15.45**

This course just gives you a first overview. Most of the semantics implementation we won't do ourselves but copy from BB. If you like to play with Prolog, though, keep playing! No matter what you think about implementations of grammatical theories (or of grammatical theories, for that matter), ***Prolog is good for your brain!***

Day 1: Prolog Recap

1 Afternoon session
i Basics

1. Prolog programs are called knowledge bases. We speak of the declarative and the procedural meaning of definitions in the knowledge base. Knowledge bases basically contain simple facts [`sleeps(mary).`] and conditional rules [`sleeps(mary) :- isnt_awake(mary).`]. Both types are called predicates with functors and arguments (in round brackets, separated by commas). Functor arity (their number of arguments) is given as `/n`.

2. Prolog searches through knowledge bases from the top to the bottom. This is highly relevant for the procedural, not for the declarative side of your work.

3. The basic symbols used in Prolog fall into the classes listed below. Notice that within the naming restrictions given, Prolog processes strings of any complexity; strings containing spaces and protected characters have to be put in '...', though. An atom could be `mary`, but also `'mary'`, and even this would be an atom, even though it contains spaces and the protected symbol `:-` which is interpreted as a literal string since it is within '...': `'look for some :-'`.

Atoms are 1. protected sequences of characters (such as `.` `,` `;` `:-` etc.) with predefined meaning (e.g., logical constants, arithmetic operators),
2. arbitrary strings with initial lower-case letter or any initial character, but enclosed in '...'; can be used as predicates or arguments.

Variables are any string that starts with a capital letter or an underscore `_` without '...'. Prolog internally only uses variables of the format `_n` (like `_7354` etc.). All variables are existentially quantified over in Prolog. One speaks of *binding* a variable to a value or of *instantiating* a variable with a value (means the same thing).
The anonymous variable `_` unifies with any structure without being instantiated.

1. Rules contain a head (basically 'what is to be proved') and one or several goals in their body after the logical constant for implication `:-` (the *if*). Goals are the facts which have to be proven to prove the head, the conditions; they're separated by the comma, which is the logical $\wedge$:

   `is_happy(mary) :- drinks_coffee(mary), attending_a_prolog_class(mary).`

   e.g., $A(m) \wedge D(m) \rightarrow H(m)$

2. The conditional/implication is not the biconditional (*iff*), such that we can have several rules with the same predicates in their heads which have the same arity, but even with different arity. Having two rules with identical heads (and thus the same arity) but different bodies is equal to having one rule with two disjunct goals. Disjunction (logical *or*) is expressed by ;

```
something(X) :- condition1(X); condition2(X).
```

equally:

```
something(X) :- condition1(X).
something(X) :- condition2(X).
```

1. The prototypical way of 'using' a Prolog program is by querying. Enter any term (e.g., a predicate-argument structure), and Prolog tries to prove it given the restrictions of the knowledge base.

2. Prolog basically uses matching (like unification) and deductive modus ponens to prove a query. Prolog instantiates variables to anything (if they have not been instantiated yet) to answer `yes`. It also tries to descend into endless recursive predicates (no *occurs* check).

3. If two Prolog expressions can be unified, the equality predicate $=/2$ is true for the two expressions. It has a prefix form `=(X,Y)` and an infix form `X=Y` . The negated form is `\=` .The backslash is not a universal logical negation (which doesn't exist as a simple constant in Prolog).

4. The logic of a knowledge base (the declarative meaning) can force Prolog to check various subgoals, subsubgoals, etc., when it is trying to prove a query. In the trace you see a `Call: (n)` for every (sub-)goal it picks to prove, and an `Exit: (n)` for successful or a `Fail: (n)` for unsuccessful proof. The `n` is an integer which will help you to identify the (possibly non-adjacent) calls and exits for a predicate. If Prolog fails on some subgoal, it steps back an checks whether there is an alternative to prove (and then tries to prove that alternative). This is called backtracking. We're going to use backtracking in our model checker.[1]

ii Recursion, Lists

1. In Prolog, the task is to formulate a declarative knowledge which forces Prolog to procedurally go into a loop while proving some (recursive) goal. This is always achieved by a rule the head of which is called from its body somewhere (or in a body of some rule which will be called when some goal from the body is being proved, and so forth). Left recursion (endless looping) occurs if no steps have been taken to stop the loop at some point.

[1] SWI says `creep` after every trace line. This is just the spell out of the command you give by pressing the return key (you're telling the interpreter to go on). You could also press 'a' to abort.

2. To avoid left recursion, every recursive rule needs a boundary condition. Typically, the boundary condition has a head identical to the recursive condition, but it gives non-recursive conditions for its success. If the recursive condition reaches a point where the boundary condition can be proven, your program terminates successfully.[2] Therefore, the boundary condition must always precede the recursive rule in the knowledge base.

3. A list is a Prolog structure that holds as its ordered elements any other Prolog structure (also other lists). A list is enumerated within `[ ]` (anything you put in rectangular brackets is regarded as a list), and the elements are separated by commas. Elements can repeat.

4. Every non-empty list is structured in head and tail. The head is the first element (or a list of elements occurring at the left margin of the list), the tail the rest of the list. One can access the head and tail of a list by using the pipe `[Head|Tail]` (naming of the variables is arbitrary as usual).

5. Empty lists are unstructured, i.e., they neither have a head nor a tail. This will cause predicates which work recursively on lists (by separating head and tail) to stop when the empty list has been reached. Note also that logically a head can never be empty, whereas a tail can.

6. `append/3` is built in and is true when *arg1* is a list, *arg2* is a list, and *arg3* is a list composed of the first list and the second list. It can be used to concatenate two lists.

7. To split two lists in prefixes and suffixes one can use `append/3` with two variables for the 'input' lists. Like so:

   ```
   append(X,Y,[l,i,s,t]).
   ```

8. `reverse/2` is built in. It is true when its two *args* are lists and have the same members, only in exactly reversed order. It can be effectively implemented using accumulators.

9. An accumulator is a sort of placeholder which makes back-to-front processing for lists easier. If you start with an empty list as your accumulator and one 'input' list you want to process, you will usually (in every recursive step) take the head off the input list (`[Head|Tail]`) and get a new accumulator by unification `[Head|Accu]` which is passed to the next recursive call as the accumulator. Until now we used to work down a list by taking heads off; with accumulators we build a new, growing list in the process.

[2] In its procedural meaning, a Prolog program can be interpreted like an algorithm; and an algorithm must always (regardless of the input) reach a point where it stops, where it 'is done'. Hence, 'termination'.

2 Evening session: DCGs, difference lists, cut
i Difference lists and definite Clause Grammars (DCGs)

1. DCGs are a handy way of implementing context-free grammars (which produce context-free languages) in Prolog. Context-free phrase structure rules and the mapping of terminal symbols (words or the alphabet, the lexical stock) can be described in a direct way.

2. CFG recognizers are implemented efficiently with difference lists. Instead of using `append/3` to concatenate phrase structures, we can define list structures and what should be left over when we unify them. Thus, we can process phrase structures (and similar structures) by simple unification (which is computationally much better than append operations).

3. Take the clause from LPN:99 `s(X,Z):-np(X,Y),vp(Y,Z).` It is designed so as to check a sentence as grammatical when `s([a,man,loves,a,woman],[]).` is queried. A sentence (`X`) should be constructed to leave nothing (`[]`) over. An NP (`[X,Y]`) or better its 'leftover' is 'chained' with a VP (`[Y,Z]`), leaving the leftover (`Z`), which, given the right definitions of lexical entries, is (`[]`). You should watch traces of a difflist vs. an append solution.

4. DCGs abstract away from the difflist architecture, but is just a translation method. Write `xcat-->ycat,zcat.` for non-terminal production rules, and `cat-->[word].` to fill your lexicon.

5. You usually don't want to use recursive rules like `s-->s,conj,s.` with `s-->np,vp.` (we're going to see why, or check LPN:102). Use new symbols for the recursive case.

6. Since DCGs only abstract away from the difflist architecture, we can add extra arguments to DCG clauses which are interpreted by Prolog like extra goals in the difflist interpretation. Obviously, `s-->np(subj),vp.` is the same as the code below. Use this for simple feature-enriched grammars.

   ```
   s(A,B) :-  np(subject,A,C),
   vp(C,B).
   ```

7. Additionally, we can specify extra tests in curly brackets. Such extra tests are goals that must be proved for the ps rule to successfully apply. Take the following code which says (in the second line) that a determiner is anything (`X`), given that `lex(X,det).` is true for `X` and the category `det`. The first line tells us that this is true for 'the'.

   ```
   lex(the,det).
   det-->[X],{lex(X,det)}.
   ```

ii Cut and fail!

1. The cut is a special atom which either (intentionally) messes up the backtracking process or limits it to increase efficiency. It is procedural in nature; when we use a cut which changes the declarative meaning of our knowledge base, it's called a red cut, otherwise it's a green cut. In general, red cuts should be avoided.

2. The cut (`!`) does the following: when it is reached (as a subgoal of some clause), it always succeeds, it freezes all variable assignments, and it blocks backtracking to any decision node above itself. It's like saying: Ok, we've successfully gotten this far, keep the success and just try to prove the rest on the basis of what has already been proved.

3. The bad news: Prolog is not a fully fledged logic programming language. Not only is there no built-in logical negation operator, it is also impossible to emulate a decent logical negation. And the negation we can emulate has a procedural side and makes use of a red cut. Negation is implemented as failure using the cut plus the built in predicate `fail/0` which (as the name tells us) always fails.

   Look at the code below, which contains a cut-fail combination . Suppose you want to say something like $p(X) \rightarrow \neg q(X)$. In Prolog, the consequent $q(X)$ turns up as the head of a clause, then the antecedent goal $q(X)$ which, if successfully proved, leads to the negation of the consequent, is added as the first goal. Then, we forbid backtracking and tell Prolog that so far, if we were successful, we never want to go back behind that success. Then, however, we explicitly tell Prolog that the whole business should fail. Dead end. Every time, Prolog tries to prove $q(X)$, then succeeds in proving $p(X)$, we make it fail. That's the closest we can get to $p(X) \rightarrow \neg q(X)$.

   ```
   q(X) :- p(X), % remove the log.  negation
   !, % red cut
   fail.  % tell Prolog to fail
   q(X) :- r(X).
   ```

4. Negation as failure is implemented as the constant `\+` . We will do some exercises showing you that one needs to know what it actually resolves to internatlly for successful usage.

Useful built-in predicates and commands	
`chdir/1`	Changes to the directory given as *arg*.
`pwd/0`	Displays the current working directory.
`consult/1`	Consults the file given as arg1 (`.pl` can be omitted).
`[`*n*`]`	Abbreviation for `consult/1` with *n* as arg1.
`trace/0`	Switch tracing on.

`notrace/0`	Switch tracing off.
`debug/0`	Go into debug mode. (In debug mode, Prolog will only display a trace if an exception has occurred. An exception, quite generally, occurs when the interpreter or a processor fails in executing a command that a program tries to get executed.)
`nodebug/0`	Leave debug mode.
`listing/0`	Display current knowledge base.
`listing/1`	Displays the definition of the predicate given as *arg*.
`apropos/1`	Displays help on the topic given as *arg*.
Ctrl+c	Break/interrupt... then press: *a* to stop processing (left recursion!), *e* to leave Prolog, *c* to continue processing,

`assert`
`asserta`
`assertz`
`retract`
	Used to manipulate the database.
	We're not using these, but you're advised to read LPN:159ff.

`atom/1`	Is *arg* atomic (constant, special symbol, integer)?
`var/1`	Is *arg* a variable?
`nonvar/1`	Is *arg* **not** a variable?
`functor/3`	*Arg1* is a term, *arg2* the functor, *arg3* the integer representing the terms arity.
`arg/3`	*Arg1* is an integer, *arg2* a complex term, arg3 the arg1-th argument of
`p(x) =..` `[p,x]`	The predicate =.. takes a term and a list as *args*. The list is composed of the functor and all the arguments of the term as elements.