Introduction to Prolog

Constructive Logic (15-317)
Instructor: Giselle Reis
Lecture 13

In this class we start learning how to program again. This time using the logic programming language Prolog. I know of two main implementations: gprolog and SWI prolog. All the code developed here was tested in gprolog\textsuperscript{1}. Although they implement the same language and a lot of the same built-in predicates, there are some small differences. Keep this in mind.

1 Basics

Prolog is a logic programming language implementing backward chaining on Horn clauses. It is somehow different from a “usual” programming language in the sense that it has no keywords (only a few pre-defined predicates) and a program consists simply of a set of clauses. A clause may be a fact we know about the world. For example:

\begin{verbatim}
mother(marge, lisa).
mother(marge, maggie).
mother(marge, bart).
father(homer, lisa).
father(homer, maggie).
father(homer, bart).
\end{verbatim}

Each of those is an atom and they are interpreted as a logical formula: e.g., the first fact is $mother(marge, lisa) \supset \top$. Clauses may also be rules in the shape:

\begin{verbatim}
head :- body
\end{verbatim}

meaning that head is true if body is true. The head is an atom, but the body can be a conjunction of atoms (this is in accordance to the definition of Horn clauses). We can have the following rules for family relations, for example:

\textsuperscript{1}Documentation here: http://www.gprolog.org/manual/html_node/index.html.
Variables are written starting by upper-case letters in Prolog. Constants start with lower case letters and we can use _ for wildcards (i.e., variables we do not care about). Clauses also have a representation as logical formulas. For example, the last clause corresponds to the formula: \( \forall x. \forall y. \forall z. (parent(z, x) \land parent(z, y) \supset sibling(x, y)) \).

Logic programming is a declarative paradigm (so are functional programming, SQL and regular expressions, for example). In a declarative paradigm, we do not tell the computer how to solve the problem but rather what constitutes a solution. Let’s say we want to come up with a clause for deciding the grandparent relation. Well, \( X \) is a grandparent of \( Y \) if \( X \) is the parent of a \( Z \) who in turn is the parent of \( Y \). In Prolog:

\[
\text{grandparent}(X, Y) :- \text{parent}(X, Z), \text{parent}(Z, Y).
\]

Type all this code into a file named intro.pl, start gprolog from the same directory where this file is located and load it via:

```
| ?- [intro].
```

Don’t forget to add periods indicating the end of each clause and the loading command. At this point, you can issue queries to your program. Ask, for example, who is Maggie’s parent:

```
| ?- parent(X, maggie).
```

As we have seen last class, using the backchaining strategy Prolog will search for clauses in the program whose head unifies with this goal. The first one found is \( \text{parent}(X, Y) :- \text{mother}(X, Y) \) by instantiating \( Y \) with maggie. In order to conclude the head, we need to show the body, so the new goal becomes \( \text{mother}(X, \text{maggie}) \). Following the same procedure, Prolog finds \( \text{mother} \) (marge, maggie) and unifies \( X \) with marge. This is the first answer shown in the prompt. Since there was another clause matching with \( \text{parent}(X, \text{maggie}) \), it asks if you want it to continue searching. Type ; (semi-colon) to continue. Now Prolog will try the second clause and come up with the answer homer for \( X \).

Gprolog has a nice debugging tool called trace. If this is turned on, you can see each step and each clause called. You can turn it on via \text{trace}. and off via \text{notrace}. (notice the periods). Type \text{h} for a list of commands in the debugging mode.

Keep in mind that everything in Prolog are predicates. There is no notion of returning value or variable assignment, only of unification.
2 Lists

List is a very basic and useful datatype in Prolog. In many problems we will see, using a list as a representation facilitates the recursion. The inductive definition of a list is \([\]\) for the empty list and \([X|L]\) for a list where the first element is \(X\) (the same as \texttt{cons}: \(\texttt{X :: L}\) in SML). Elements in a list are separated by commas: \([1,2,3,4]\). Also, they do not need to be all of the same type: \([1, \text{'a string'}, 1.5]\) is a list.

We start by defining a predicate \texttt{prefix1/2}\(^2\) that decides if the first list is a prefix of the second list. We do this recursively on the prefix: the empty list is a prefix of any list. If the prefix starts with an element \(X\), then the list must also start with \(X\) and the rest of the prefix must be a prefix of the rest of the list. In Prolog code:

\[
\text{prefix1([], L).} \\
\text{prefix1([X|P], [X|L]) :- prefix1(P, L).}
\]

Loading this code in gprolog results in a warning message about singleton variables. This is because the \(L\) in the first clause is unimportant and thus can be replaced by a wildcard:

\[
\text{prefix1([], _).} \\
\text{prefix1([X|P], [X|L]) :- prefix1(P, L).}
\]

Analogously, we can define the predicate \texttt{suffix1/2}. Since we can only deconstruct a list from its front, we induct on the list and not on the suffix: by removing elements from the front of the list (whose values we do not need to know), we need to get to something equal to the suffix.

\[
\text{suffix1(L, L).} \\
\text{suffix1(S, [\_|L]) :- suffix1(S, L).}
\]

We now define a predicate \texttt{append1/3} that decides if the third list is the result of appending the first two. We induct on the first list: if \(Xs\) is an empty list, then appending it to \(Ys\) is \(Ys\) itself (the empty list is the neutral element of append). If it has at least one element, this must be an element in the resulting list and, moreover, \(L\) must be the result of appending \(Xs\) and \(Ys\).

\[
\text{append1([], Ys, Ys).} \\
\text{append1([X|Xs], Ys, [X|L]) :- append1(Xs, Ys, L).}
\]

Using \texttt{append1/3} we can implement the prefix and suffix predicates more easily:

\[
\text{prefix2(P, L) :- append1(P, _, L).} \\
\text{suffix2(S, L) :- append1(_, S, L).}
\]

\(^2\)The \(/2\) indicates that this is a binary predicate. Also, we are using \texttt{prefix1}\(^2\) because the predicate \texttt{prefix} is predefined in gprolog.
3 Arithmetic

In previous lectures we have seen ways to define natural numbers and operations on them recursively. Luckily, Prolog provides built-in numbers and arithmetic operations so we don’t have to deal with \( s(s(s(s(z)))) \) for number 5. Let us use arithmetic to compute the length of a list:

```prolog
length1([], 0).
length1([_|L], N) :- length1(L, M), N is M+1.
```

Here we use the construct `is/2`, which tests if the left side is unifiable with the evaluation of the right side. Typically, the left side will be a variable while the right side is an expression. Try a few example on the prompt to see how this works.

Note that we cannot use arithmetic expressions as terms inside a predicate. If we need to use, say, \( p(X+1) \), we need to first do \( Y \leftarrow X+1 \) and then use \( Y \) in \( p(Y) \).

gprolog supports not only integers but also floats. The arithmetic comparison operators are: \( >, <, =<, >= \) (the operators that use the equality symbol should not form an arrow). Arithmetic equality and disequality are denoted by `\=:=` and `\=/=` respectively. When using those operators, both sides must be instantiated.

**Remark 1.** Equality and disequality between terms is denoted by `\=/=` and `\=:=` respectively, and interpreted as unifiability. Equality succeeds if the terms can be unified and disequality succeeds if they cannot be unified. Observe its behavior compared to arithmetic disequality:

```
| ?- 3 \=:= 2+1.
  no
| ?- 3 \=/= 2+1.
  yes
| ?- 3 \=/= 2+X.
  yes
| ?- 3 =:= 2+1.
  yes
| ?- 3 = 2+1.
  no
| ?- 2+1 = 2+X.
  X = 1
```
yes

With the information you have so far, you should be able to implement quicksort. Try it out before looking at the answer at the end of these notes.

4 Cut

Attention! This is not the same as cut in sequent calculus.

Prolog’s cut is an artifact to control backtracking. It is denoted by the exclamation mark ! and used in bodies of clauses. If the program flow reaches a cut, then all choices made so far are committed to and the backtracking stack is cleared. Using cut we can implement ifthenelse/3 for example:

% If A then B else C
ifthenelse(A, B, C) :- A, !, B.
ifthenelse(A, B, C) :- C.

When queried with an ifthenelse clause, Prolog will first match it with the first clause and try to resolve A. If it succeeds, it gets to the cut, which means every backtracking point during the search for A and the choice of the first ifthenelse clause in the first place are cleared. It will then continue with B, keeping its backtracking points and failing if it fails (the second ifthenelse clause is not considered because of the cut).

Cuts can be a very useful tool to prune the search space, but as with all pruning, one needs to be careful not to change the meaning of the program or prune too much.

5 Negation as failure

Even though there is no logical negation in Prolog, it allows for negation as failure, denoted by \+. As the name suggests, the negation of a predicate will hold if it is not possible to derive it. We can implement this using cut:

\+(A) :- A, !, fail.
\+(A).

Constructively speaking, failure to derive is not the same thing as logical negation. Indeed this is considered as an extra-logical construct in Prolog and should be avoided in general. If the predicate contains variables, negation as failure can generate unsoundness. Observe the following example:

\textsuperscript{3}See a nice explanation of good and bad cuts here: http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse44.
\begin{verbatim}
| ?- \+(X = 1), X = 2.
  no
| ?- X = 2, \+(X = 1).
  X = 2
  yes

  In fact, whenever we are using variables and arithmetic or non-logical constructs, it is
good to keep in mind that a variable being instantiated or not may make a difference on
the success or failure of these predicates!

A Quicksort implementation

quicksort([], []). quicksort([X|Xs], Ys) :- partition(Xs, X, Ls, Gs),
  quicksort(Ls, Sl),
  quicksort(Gs, Sr),
  append(Sl, [X|Sr], Ys).

partition([], _, [], []). partition([X|L], P, [X|Ls], Gs) :- X < P,
  partition(L, P, Ls, Gs).
partition([X|L], P, Ls, [X|Gs]) :- X >= P,
  partition(L, P, Ls, Gs).
\end{verbatim}