CSC384: Intro to Artificial Intelligence

A Brief Introduction to Prolog

Part 2/2:

- Debugging Prolog programs
- Passing predicates as arguments and constructing predicates dynamically (on-the-fly).
- Efficient lists processing using accumulators
- Negation as failure (NAF)
- Cut (controlling how Prolog does the search)
- if-then-else

Please read also:

http://cs.union.edu/~striegnk/learn-prolog-now/html/node87.html#lecture10
Debugging Prolog Programs

- **Graphical Debugger (gtrace):**
  - On CDF: run “xpce” instead of “prolog”. Then, to debug:
    - `?- gtrace, father(X, john).`
  - Very easy to use. see “http://www.swi-prolog.org/gtrace.html”

- **Text-based debugger:**
  - You can put or remove a breakpoint on a predicate:
    - `?- spy(female).`
    - `?- nospy(female).`
  - To start debugging use “trace” before the query:
    - `?- trace, male(X).`
  - While tracing you can do the following:
    - `creap`: step inside the current goal (press c/enter/or space)
    - `leap`: run up to the next spypoint (press l)
    - `skip`: step over the current goal without debugging (press s)
    - `abort`: abort debugging (press a)
  - And much more… press h for help
Text Debugger Example

?-spy(female/1).

yes

?-mother(X,Y).  %starts debugging!
Call: (9) female(albert) ?

?-nospy(female/1).
% Spy point removed from female/1

trace, father(X,Y).  %let’s debug!
Call: (9) father(_G305, _G306) ?

Simple Exercise: debug this program both in the text and graphical debugger.
We can pass a predicate as an argument to a rule:

```prolog
test(X) :- X.
?- test(male(john)). %succeeds if male(john) holds.
?- test(parent(carrot,4)). %fails.
```

What if we want to pass arguments of the predicate separately?

```prolog
test(X,Y) :- X(Y). %this is syntax error!
?- test(male, john).
```

Unfortunately the above does not work, we cannot write \( X(Y) \) !

`..` is used to build predicates on the fly:

```prolog
test(X,Y) :- G =.. [X,Y], G. %builds predicate \( X(Y) \) dynamically and calls it
?- test(male, john).
```

In general, \( G =.. [P,X1,X2,...,Xn] \) creates \( P(X1,X2,...,Xn) \). E.g:

```prolog
?- G =.. [parent, john, X].
G = parent(john, X)
```
Adding/Deleting Rules/Facts Dynamically

A program can add or delete facts/rules dynamically:

- **assert(term)**  %adds the given rule or fact
  - `assert(male(john)).`
  - `assert((animal(X) :- dog(X))).`

- **retract(term)**  %deletes the first fact/rule that unifies with the given term
  - `retract(animal(_)).`
  - `died(X), retract(parent(john,X)).`

- **retractrtall(term)**  %deletes ALL facts/rules that unify
  - `retractall(parent(_,_)).`

- There is also **assertz(term)** that adds the fact/rule to the end rather than beginning
More Lists Processing in Prolog

- Much of Prolog’s computation is organized around lists.

- Many built-in predicates: member, append, length, reverse.

- List of lists:
  - `[[1,2], [a, b, c(d), 4], []]`

- Can define a matrix, e.g. 3x2: `M=[[1,2], [-1,4], [5,5]]`

- Elements can be structures: `M2= [[[1.5,a), (3.2,b)], [(0,c), (7.2,d)]]` is a 2x2 matrix. Then if write `M2=[H|_], H=[_,(Cost, Name)]`. It succeeds and we get Cost=3.2 and Name=b.

- Exercise: write a predicate `getElm(M,R,C,E)` which holds if element E is at M[R][C]. Assume the matrix is KxL and I,J>=0 and in range. Note that M[0][0] is the element at 1st row 1st column.
Lists: Extracting Desired Elements

- Extracting all elements satisfying a condition:
  e.g. extract(male, [john, alice, 2, sam], Males)

- Generally:
  extract(+Cond, +List, ?Newlist)

  %Note: +, -, ? are not actual Prolog symbols, just used as convention!

  extract(_ ,[ ], [ ]).  
  extract(Condname, [X | Others], [ X | Rest]):-  
    Cond =.. [Condname,X],  %building cond predicate CondName(X)  
    Cond,  %now, calling Cond predicate to see if it holds  
    extract(Condname, Others, Rest).

  extract(Condname, [X | Others], Rest):-  
    Cond =.. [Condname,X],  
    \+ Cond, extract(Condname, Others, Rest).

- \+ is negation as failure. We can also simplify the above using if-then-else. We will get back to these in a couple of slides.
Lists: append

- Appending two lists (challenge: no assumption on args):
  append(?X, ?Y, ?Z)

  Holds iff Z is the result of appending lists X and Y.

Examples:

- `append([a,b,c],[1,2,3,4],[a,b,c,1,2,3,4])`
- Extracting the third element of L: `append([_,_,X],_,L)`
- Extracting the last element of L: `axppend(_, [X], L)`
- Finding two consecutive elements X & Y in L:
  `append(_, [X, Y |_], L)`
Implementing append

definition:  append(\(X, Y, Z\))

append([], L, L).
append([H|T], L, [H|L2]):-
               append(T, L, L2).

- What are all the answers to append(\(_, [X, Y], [1, 2, 3, 4, 5] \)) ?
- What are all the answers to append(X, [a], Y) ?
- What is the answer to append(X, Y, Z)? How many answers?
Lists: reversing

- Reversing a list: [1,2,[a,b],3] -> [3,[a,b],2,1]
  \[\text{reverse}(\text{L},\text{RevL})\]

  reverse([], []).  
  reverse([H|T],RevL):-  
    reverse(T,RevT), append(RevT,[H],RevL).

- This is not efficient! Why? \(O(N^2)\)
Efficiency issues: Fibonacci

- Fibonacci numbers:
  \[ \begin{align*}
  f(n) = & \begin{cases} 
  0 & n = 0 \\
  1 & n = 1 \\
  f(n-1) + f(n-2) & n > 1
  \end{cases}
  \end{align*} \]

- Consider the following implementation:
  
  fib(0,0).
  fib(1,1)
  fib(N,F):- N>1,
  N1 is N-1, fib(N1,F1),
  N2 is N-2, fib(N2, F2),
  F is F1+F2.

- This is very inefficient (exponential time)! Why?
- Solution: use accumulator!
Fibonacci using accumulators

Definition: \texttt{fibacc(+N,+Counter,+FibNminus1,+FibNminus2,-F)}

We start at counter=2, and continue to reach N. \texttt{FibNminus1} and \texttt{FibNminus2} are accumulators and will be update in each recursive call accordingly.

\begin{align*}
\texttt{fibacc(N,N,F1,F2,F):-} & \quad \%\text{basecase: the counter reached N, we are done!} \\
& \quad \text{F is F1+F2.} \\
\texttt{fibacc(N,I,F1,F2,F):-} & \quad I<N, \quad \%\text{the counter < N, so updating F1\&F2} \\
& \quad I\text{pls1 is I +1, F1New is F1+F2, F2New is F1,} \\
& \quad \text{fibacc(N,I\text{pls1},F1\text{New},F2\text{New},F).}
\end{align*}

\begin{itemize}
\item This is \textit{O(N)}. \\
\item Now we define \texttt{fib(N,F)} for \texttt{N>1} to be \texttt{fibacc(N,2,1,0,F)}.
\end{itemize}
Accumulators: reverse

- Efficient List reversal using accumulators: \( O(n) \)

```prolog
reverse(L,RevL):-
    revAcc(L,[ ], RevL).

revAcc([ ],RevSoFar, RevSoFar).
revAcc([H | T],RevSoFar, RevL):-
    revAcc(T,[H | RevSoFar], RevL).
```
Negation As Failure

- Prolog cannot assert something is false.
- Anything that cannot be proved from rules and facts is considered to be false (hence the name Negation as Failure)
- Note that this is different than logical negation!
- In SWI it is represented by symbols `\+`
  - `\+ member(X,L)` %this holds if it cannot prove X is a member of L
  - `\+(A<B)` %this holds if it cannot prove A is less than B
- `X \= Y` is shorthand for `\+ (X=Y)`
NAF examples

Defining disjoint sets:

\[
\text{overlap}(S_1, S_2):=\%S_1 \& S_2 \text{ overlap if they share an element.}
\text{member}(X, S_1), \text{member}(X, S_2).
\text{disjoint}(S_1, S_2):= \neg \text{ overlap}(S_1, S_2).
\]

?- disjoint([a,b,c],[2,c,4]).
no
?- disjoint([a,b],[1,2,3,4]).
yes
?- disjoint([a,c],X).
No \leftarrow \text{this is not what we wanted it to mean!}
Proper use of NAF

- $\\alg{+ G}$ works properly only in the following two cases:
  
  - When $G$ is fully instantiated at the time of processing $\\alg{+}$. In this case, the meaning is straightforward.
  
  - When there are uninstantiated variables in $G$ but they do not appear elsewhere in the same clause. In this case, it means there are no instantiations for those variable that makes the goal true. e.g. $\\alg{+G(X)}$ means there is no $X$ such that $G(X)$ succeeds.
If-then-else

Let’s implement \( \text{max}(X,Y,Z) \) which holds if \( Z \) is maximum of \( X \) and \( Y \). using NAF:

\[
\begin{align*}
\text{max}(X,Y,Z) & : - X =< Y , Z = Y. \\
\text{max}(X,Y,Z) & : - \ + (X =< Y) , Z = X.
\end{align*}
\]

This is a simple example. But shows a general pattern: we want the second rule be used only if the condition of the 1\(^{st}\) rule fails.: it’s basically an if-then-else:

\[
\begin{align*}
p & : - A, B. \\
p & : - \ + A , C.
\end{align*}
\]

SWI has a built-in structure that simplifies this and is much more efficient: ->
If-then-else

- In Prolog, “if A then B else C” is written as \((A \rightarrow B ; C)\).
- To Prolog this means:
  - try A. If you can prove it, go on to prove B and ignore C. If A fails, however, go on to prove C ignoring B.
- Let’s write max using \(\rightarrow\):
  ```prolog
  max(X,Y,Z) :-
      (X =< Y \rightarrow Z = Y ;
        Z = X
    ).
  ```
- Note that you may need to add parenthesis around A, B, or C themselves if they are not simple predicates.
Guiding the Search Using Cut!

- The goal “!”, pronounced cut, always succeeds immediately but just once (cannot backtrack over it).

- It has an important side-effect: once it is satisfied, it disallows (just for the current call to predicate containing the cut):
  - backtracking before the cut in that clause
  - Using next rules of this predicate

- So, below, before reaching cut, there might be backtracking on b1 and b2 and even trying other rules for p if b1&b2 cannot be satisfied.

  p:- b1,b2,!,a1,a2,a3. %however, after reaching !, no backtracking on b1&b2
  p:- r1,…,rn. %also this rule won’t be searched
  p:- morerules. %this one too!

- See the following link for more details and examples:
  http://cs.union.edu/~striegnk/learn-prolog-now/html/node88.html#sec.l10.cut
Implementing \(+\) and \(-\rightarrow\) using cut

- \textbf{fail} is a special symbol that will immediately fail when Prolog encounters it.
- We can implement NAF using cut and fail as follows:

\[
\text{neg}(\text{Goal}) :\ - \ \text{Goal}, !, \text{fail}.
\]
\[
\text{neg}(\text{Goal}).
\]

- \textbf{neg} will act similarly to \(+\). Why?

- We can implement “\(p :\ - A \rightarrow B ; C\)” using cut:

\[
\begin{align*}
\text{p} & :\ - A,!,B. \\
\text{p} & :\ - C.
\end{align*}
\]

- If A can be proved, we reach the cut and the 2\textsuperscript{nd} rule will not be tried.
- If A cannot be proved we don’t reach cut and the 2\textsuperscript{nd} rule is tried.