Logic Programming and Prolog (XSB)

CSE 307 – Principles of Programming Languages
Stony Brook University
http://www.cs.stonybrook.edu/~cse307
The XSB Prolog System

- [http://xsb.sourceforge.net](http://xsb.sourceforge.net)
- Developed at Stony Brook by Prof. Warren and many contributors
Variables: Any alphanumeric symbol that starts with a capital letter or a _ (underscore)
- Examples: Abc, ABC2, _abc34
- Non-examples: 123, abc, aBC

Each occurrence of a singleton symbol _ is treated as a new variable, which was never seen before:
- Example: p(_,abc), q(cde,_): the two _’s are treated as completely different variables
- But the two occurrences of _xyz in p(_xyz,abc), q(cde,_xyz) refer to the same variable

Relation names and constants:
- must either start with a lowercase letter (and include only alphanumerics and _)
  - Example: abc, aBC123, abc_123
- or be enclosed in single quotes
  - Example: 'abc &% (, foobar1'
  - Note: abc and 'abc' refer to the same thing
Numbers can be floats or integers.

A compound term is composed of an atom called a "functor" and a number of "arguments", which are again terms: tree(node(a),tree(node(b),node(c)))

Special cases of compound terms:
- Lists: ordered collections of terms: [], [1,2,3], [a,1,X|T]
- Strings: A sequence of characters surrounded by quotes is equivalent to a list of (numeric) character codes: “abc”, “to be, or not to be”
XSB Prolog

• Negation:
  • **not**: negation-as-failure

• Another negation called **tnot**
  • Use: \( \ldots \leftarrow \ldots, tnot(foobar(X)). \)
  • All variables under the scope of **tnot** must also occur to the left of that scope in the body of the rule in other **positive** relations:
    • Ok: \( \ldots \leftarrow p(X,Y), tnot(foobar(X,Y)), \ldots \)
    • Not ok: \( \ldots \leftarrow p(X,Z), tnot(foobar(X,Y)), \ldots \)

• XSB also supports Datalog:
  \( :- \text{auto_table}. \)
  at the top of the program file
Overview of Installation

• Unzip/untar; this will create a subdirectory XSB
• Windows: you are done
• Linux:
  
  cd XSB/build
  ./configure
  ./makexsb

  That’s it!
• Cygwin under Windows: same as in Linux
Use of XSB

- Put your ruleset and data in a file with extension .p (or .pl)
  
  \[
  \begin{align*}
  p(X) & : - q(X,\_). \\
  q(1, a). \\
  q(2, a). \\
  q(b, c). \\
  ?- p(X).
  \end{align*}
  \]

- Don’t forget: all rules and facts end with a period (.)

- Comments: /*…*/ or %…. (% acts like // in Java/C++)

- Type
  
  ```
  .../XSB/bin/xsb
  ...
  
  \text{where ... is the path to the directory where you downloaded XSB}
  ```

- You will see a prompt
  
  ```
  | ?-
  ```

  and are now ready to type queries

(c) Paul Fodor (CS Stony Brook)
Use of XSB

• Loading your program, myprog.P
  
  | ?- [myprog].

  XSB will compile myprog.P into myprog.xwam and load it.

• Now you can type further queries, e.g.
  
  | ?- p(X).
  | ?- p(1).

  Etc.
Some Useful Built-ins

• `write(X)` — write whatever X is bound to
• `writeln(X)` — write then put newline
• `nl` — output newline
• Equality: `=`
• Inequality: `\=`

http://xsb.sourceforge.net/manual1/index.html (Volume 1)
If you need it: use the builtin *is*

\[ p(1). \ p(2). \]

\[ q(X) \ :- \ p(Y), \ X \ is \ Y*2. \]

Now \( q(2) \), \( q(4) \) will become true.

**Note:**

\[ q(2*X) \ :- \ p(X). \]

will not do what you might think it will do.

It will make \( q(2*1) \) and \( q(2*2) \) true, but \( 2*1 \) and \( 2*2 \) are treated completely differently from \( 2 \) and \( 4 \) (no need to get into all that for now).
Some Useful Tricks

• XSB returns only the first answer to the query. To get the next, type \code{; <Return>}. For instance:

\begin{verbatim}
?~ q(X). <Return>
X = 2 <Return>
X = 4 <Return>
yes
?~
\end{verbatim}

• Usually, typing the \code{;}’s is tedious. To do this programmatically, use this idiom:

\begin{verbatim}
?~ (q(_X), write('X='), writeln(_X), fail; true).
\end{verbatim}

\_X here tells XSB to not print its own answers, since we are printing them by ourselves. (XSB won’t print answers for variables that are prefixed with a \_.)
Aggregates in XSB

- `setof(?Template, +Goal, ?Set)`: ?Set is the set of all instances of ?Template such that Goal is provable.
- `bagof(?Template, +Goal, ?Bag)`: has the same semantics as `setof/3` except that the third argument returns an unsorted list that may contain duplicates.
- `findall(?Template, +Goal, ?List)`: is similar to predicate `bagof/3`, except that variables in Goal that do not occur in Template are treated as existential, and alternative lists are not returned for different bindings of such variables.
- `tfindall(?Template, +Goal, ?List)`: is similar to predicate `findall/3`, but the Goal must be a call to a single tabled predicate.
A list is handled as binary tree in Prolog

[Head | Tail] OR
.(Head, Tail)

Where Head is an atom and Tail is a list

We can write [a, b, c] or .(a, .(b, .(c, []))).
Matching

• Given two terms, they are identical or the variables in both terms can have same objects after being instantiated.

  \[ \text{date}(D,M,2006) \quad \text{unification} \quad \text{date}(D1,\text{feb},Y1) \]
  \[ D=D1, \quad M=\text{feb}, \quad Y1=2006 \]

• General Rule to decide whether two terms, \( S \) and \( T \) match are as follows:
  
  • If \( S \) and \( T \) are constants, \( S=T \) if both are same object
  
  • If \( S \) is a variable and \( T \) is anything, \( T=S \)
  
  • If \( T \) is variable and \( S \) is anything, \( S=T \)
  
  • If \( S \) and \( T \) are structures, \( S=T \) if
    
    • \( S \) and \( T \) have same functor
    
    • All their corresponding arguments components have to match
Declarative and Procedural Way

- Prolog programs can be understood two ways: declaratively and procedurally.

\[ P : - Q, R \]

- **Declarative Way**
  - P is true if Q and R are true

- **Procedural Way**
  - To solve problem P, first solve Q and then R (or) To satisfy P, first satisfy Q and then R
  - Procedural way does not only define logical relation between the head of the clause and the goals in the body, but also the order in which the goal are processed.
Evaluation

mother_child(trude, sally).
father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
sibling(X, Y) :- parent_child(Z, X),
             parent_child(Z, Y).

?- sibling(sally, erica).
Yes  (by chronological backtracking)
Evaluation

• \texttt{?- father\_child(Father, Child).} enumerates all valid answers on backtracking.

• \texttt{?- sibling(S1, S2).} enumerates all valid answers on backtracking.
Append example

```
append([],L,L).
append([X|L], M, [X|N]) :- append(L,M,N).
```

append([1,2],[3,4],X)?
Append example

append([], L, L).
append([X|L], M, [X|N]) :- append(L, M, N).

`append([1,2],[3,4],X)`

\[ X=1, L=[2], M=[3,4], A=[X|N] \]
Append example

append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).

append([2],[3,4],N)?
append([1,2],[3,4],X)? X=1, L=[2], M=[3,4], A=[X|N]
append([], L, L).
append([X|L], M, [X|N']) :- append(L, M, N').

append([2], [3, 4], N)?
X=2, L=[], M=[3, 4], N=[2|N']

append([1, 2], [3, 4], X)?
X=1, L=[2], M=[3, 4], A=[1|N]
Append example

append([],L,L).
append([X|L],M,[X|N']) :- append(L,M,N').

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>append([], [3, 4], X)</td>
<td>X=2, L=[], M=[3, 4], N=[2</td>
</tr>
<tr>
<td>append([2], [3, 4], X)</td>
<td>X=1, L=[2], M=[3, 4], A=[1</td>
</tr>
<tr>
<td>append([1, 2], [3, 4], X)</td>
<td></td>
</tr>
</tbody>
</table>
Append example

\begin{align*}
\text{append}([], L, L). \\
\text{append}([X|L], M, [X|N']) & : - \text{append}(L, M, N').
\end{align*}

<table>
<thead>
<tr>
<th>Expression</th>
<th>Unification</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{append}([], [3,4], N')?</td>
<td>L = [3,4], N' = L</td>
<td></td>
</tr>
<tr>
<td>\text{append}([2], [3,4], N)?</td>
<td>X=2, L=[], M=[3,4], N=[2</td>
<td>N']</td>
</tr>
<tr>
<td>\text{append}([1,2], [3,4], X)?</td>
<td>X=1, L=[2], M=[3,4], A=[1</td>
<td>N]</td>
</tr>
</tbody>
</table>
append([], L, L).
append([X|L], M, [X|N']) :- append(L, M, N').

append([], [3, 4], N')?

L = [3, 4], N' = L

append([2], [3, 4], N)?

X=2, L=[], M=[3, 4], N=[2|N']

append([1, 2], [3, 4], X)?

X=1, L=[2], M=[3, 4], A=[1|N]

Answer: A = [1, 2, 3, 4]
More Examples

member(X,[X|R]).
member(X,[Y|R]) :- member(X,R)

• *X is a member of a list whose first element is X.*
• *X is a member of a list whose tail is R if X is a member of R.*

?- member(2,[1,2,3]).
Yes

?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
No
More Examples

select(X,[X | R],R).

select(X,[F | R],[F | S]) :- select(X,R,S).

- When X is selected from [X | R], R results.
- When X is selected from the tail of [X | R], [X | S] results, where S is the result of taking X out of R.

?- select(X,[1,2,3],L).
X=1  L=[2,3] ;
X=2  L=[1,3] ;
X=3  L=[1,2] ;
No
More Examples

reverse([X|Y],Z,W) :- reverse(Y,[X|Z],W).
reverse([],X,X).

?- reverse([1,2,3],[],X).
X = [3,2,1]
Yes
perm([],[]).

perm([X | Y], Z) :- perm(Y, W), select(X, Z, W).

?- perm([1,2,3], P).
P = [1,2,3] ;
P = [2,1,3] ;
P = [2,3,1] ;
P = [1,3,2] ;
P = [3,1,2] ;
P = [3,2,1]
Recursion

• Transitive closure:

edge(1, 2).
edge(2, 3).
edge(2, 4).

reachable(X, Y) :- edge(X, Y).
reachable(X, Y) :- edge(X, Z), reachable(Z, Y).
?- reachable(X,Y).

X = 1
Y = 2; Type a semi-colon repeatedly

X = 2
Y = 3;

X = 1
Y = 3;

X = 1
Y = 4;

no

?- halt. Command to Exit XSB
Cut (! in Prolog) is a goal which always succeeds, but cannot be backtracked past.

**Green cut**

```prolog
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X), \+ gotmoney(X).
```

Cut says “stop looking for alternatives”

by explicitly writing \+ gotmoney(X), it guarantees that the second rule will always work even if the first one is removed by accident or changed

**Red cut**

```prolog
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X).
```
Definite clause grammar (DCG)

• A **DCG** is a way of expressing grammar in a logic programming language such as Prolog

• The definite clauses of a DCG can be considered a set of axioms where the fact that it has a parse tree can be considered theorems that follow from these axioms
DCG grammar for arithmetic expr.

`expr` --> `term`, `addterm`.

`addterm` --> `[+], expr`.

`term` --> `factor`, `multfactor`.

`multfactor` --> `[*], term`.

`factor` --> `[I], {integer(I)}`.

`factor` --> `['()], expr, ['']`.

% xsb

| ?- `expr([4,*5,+1],[])`.

yes

| ?- `expr([1,+3,*,'(',2,+,4,')'],[])`.

yes
DCG grammar for arithmetic expr.

:- table expr/3, term/3.

expr(Val) --> expr(Eval), [+], term(Tval), \{Val is Eval+Tval\}.
expr(Val) --> term(Val).

term(Val) --> term(Tval), [*], primary(Fval), \{Val is Tval*Fval\}.
term(Val) --> primary(Val).

primary(Val) --> ['('], expr(Val), [')'].

primary(Int) --> [Int], \{integer(Int)\}.

%xsb

| ?- [grammar].
| ?- expr(Val,[1,+,*],*,',',[4,+,*],[]).

Val = 55
The cat scares the mouse.
A SIMPLE NATURAL LANGUAGE DCG

sentence --> noun_phrase, verb_phrase.

verb_phrase --> verb, noun_phrase.

noun_phrase --> determiner, noun.

determiner --> [ the].

noun --> [ cat].

noun --> [ cats].

noun --> [ mouse].

verb --> [ scares].

verb --> [ scare].

?- sentence(X,[]).
THIS GRAMMAR GENERATES

[ the, cat, scares, the, mouse]

[ the, mouse, scares, the, mouse]

[ the, cats, scare, the, mouse]

CONTEXT DEPENDENT!
NUMBER AGREEMENT CAN BE FORCED BY ARGUMENTS

sentence(Number)  -->
  noun_phrase(Number), verb_phrase(Number).

verb_phrase(Number)  -->
  verb(Number), noun_phrase(Number1).

noun_phrase(Number)  -->
  determiner(Number), noun(Number).

noun(singular)  -->  [mouse].

noun(plural)  -->  [mice].

verb(singular)  -->  [scares].

verb(plural)  -->  [scare].

?- sentence(X,Number,[]).
DCG with Parse tree

sentence(s(NP,VP)) --> noun_phrase(NP), verb_phrase(VP).

noun_phrase(np(D,N)) --> det(D), noun(N).

verb_phrase(vp(V,NP)) --> verb(V), noun_phrase(NP).

det(d(the)) --> [the].

det(d(a)) --> [a].

noun(n(bat)) --> [bat].

noun(n(cat)) --> [cat].

verb(v(eats)) --> [eats].

?- sentence(Parse_tree, [the,bat,eats,a,cat], []).

Parse_tree = s(np(d(the),n(bat)),vp(v(eats),np(d(a),n(cat)))).
% special rule syntax
s --> np, vp.
np --> det, n.
vp --> tv, np.
v --> v.
det --> [the].
det --> [a].
det --> [every].
n --> [man].
n --> [woman].
n --> [park].
tv --> [loves].
tv --> [likes].
v --> [walks].

% Grammar in pure Prolog
s(S0,S) :- np(S0,S1), vp(S1,S).
np(S0,S) :- det(S0,S1), n(S1,S).
v(S0,S) :- v(S0,S).
v(S0,S) :- tv(S0,S1), np(S1,S).
det(S0,S) :- S0=[the|S].
det(S0,S) :- S0=[a|S].
det(S0,S) :- S0=[every|S].
n(S0,S) :- S0=[man|S].
n(S0,S) :- S0=[woman|S].
n(S0,S) :- S0=[park|S].
tv(S0,S) :- S0=[loves|S].
tv(S0,S) :- S0=[likes|S].
v(S0,S) :- S0=[walks|S].

?- s([a,man,loves,the,woman],[]).

yes