CSI 2165 Winter 2006

Diana Inkpen
SITE
University of Ottawa

Part IV

Applications: Artificial Intelligence

- Natural language processing
- Games
- Meta-interpreters
- Theorem proving

Natural Language Processing in Prolog

- Prolog is especially well-suited for developing natural language systems.
- We implement a natural language parser for a simple subset of English, that includes sentences such as:
  - The dog ate the bone.
  - The big mouse chases a lazy cat.

Grammar rules

- This grammar can be described with the following grammar rules:

<sentence> ::= <nounphrase> <verbphrase>
<nounphrase> ::= <determiner> <noun> |
<determiner> <adjective> <noun>
<verbphrase> ::= <verb> <nounphrase>

<determiner> ::= the | a
<noun> ::= dog | bone | mouse | cat
<verb> ::= ate | chases
<adjective> ::= big | brown | lazy

Accepting sentences

- To begin with, we will simply determine if a sentence is a legal sentence according to the grammar.
- A predicate called sentence, which will determine if its first argument is a sentence.
- The sentence will be represented as a list of words. Examples:
  - [the,dog,ate,the,bone]
  - [the,brown,mouse,chases,a,lazy,cat]

One strategy for parsing

- **Generate-and-test strategy**: the list to be parsed is split in different ways; test to see if the splittings are components of a legal sentence.
- We can use append to generate all the splittings of a list.
- The top-level rule would be:

```
sentence(L) :- append(NP, VP, L),
               nounphrase(NP),
               verbphrase(VP).
```

- The append predicate will generate possible values for the variables NP and VP, by splitting the original list L. The next two goals test each of the portions of the list to see if they are grammatically correct. If not, backtracking into append causes another possible splitting to be generated.

The lexicon

- The clauses for nounphrase and verbphrase are similar to sentence, and call further predicates that deal with smaller units of a sentence, until the word definitions are met, such as:

```
verb([ate]).
verb([chases]).
noun([mouse]).
noun([dog]).
```


Using Difference Lists

- The previous strategy, is extremely slow because of the constant generation and testing of trial solutions that do not work. Furthermore, the generating and testing is happening at multiple levels.
- The more efficient strategy is to skip the generation step and pass the entire list to the lower level predicates, which in turn will take the grammatical portion of the sentence they are looking for from the front of the list and return the remainder of the list.
- To do this, we use a structure called a difference list. It is two related lists, in which the first list is the full list and the second list is the remainder. The two lists are the last two arguments in a predicate.

Difference Lists

- Here is the first grammar rule using difference lists.
- A list S is a sentence if we can extract a noun phrase from the beginning of it, with a remainder list of S1, and if we can extract a verb phrase from S1 with the empty list as the remainder.

```
sentence(S,[]):- nounphrase(S,S1),
                 verbphrase(S1,[]).
```


The rest of the grammar rules

```
nounphrase(NP,X):- determiner(NP,S1),
                   noun(S1,X).
nounphrase(NP,X):- determiner(NP,S1),
                   adjective(S1,S2),
                   noun(S2,X).
verbphrase(VP,X):- verb(VP,S1),
                   nounphrase(S1,X).
```


The lexicon

- The lowest level predicates that define the actual words. They too must be difference lists. If the head of the first list is the word, the remainder list is simply the tail.

```
noun([dog|X],X).
noun([cat|X],X).
noun([mouse|X],X).
verb([ate|X],X).
verb([chases|X],X).
adjective([big|X],X).
adjective([brown|X],X).
adjective([lazy|X],X).
determiner([the|X],X).
determiner([a|X],X).
```


Testing

```
?- sentence([the,lazy,mouse,ate,a,dog],[]).
   yes
?- sentence([the,dog,ate],[]).
   no
?- sentence([a,big,cat,chases,a,lazy,dog],[]).
   yes
?- sentence([the,cat,jumps,the,mouse],[]).
   no

?- noun([dog,ate,the,bone],X).
   X = [ate,the,bone]
?- verb([dog,ate,the,bone],X).
   no
```

Definite Clause Grammar

- The use of difference lists for parsing is so common in Prolog, that most Prologs contain additional syntactic sugaring that simplifies the syntax by hiding the difference lists from view. This syntax is called Definite Clause Grammar (DCG), and looks like normal Prolog, only the symbol :- is replaced with an arrow -->.
- The DCG representation is parsed and translated to normal Prolog with difference lists.
- Using DCG, the sentence predicate developed earlier would be phrased:

```
sentence --> nounphrase, verbphrase.
```


DCG

```
sentence --> nounphrase, verbphrase.
```

- This would be translated into normal Prolog, with difference lists. The above example would be translated into the following equivalent Prolog.

```
sentence(S1, S2):-  nounphrase(S1, S3),
                    verbphrase(S3, S2).
```

- Thus, if we define 'sentence' using DCG we still must call it with two arguments, even though the arguments were not explicitly stated in the DCG representation.

```
?- sentence([the,dog,chases,the,cat], []).
   yes
```


The DCG Lexicon

- The DCG vocabulary is represented by simple lists.

```
noun --> [dog].
verb --> [chases].
```

- These are translated into Prolog as difference lists.

```
noun([dog|X], X).
verb([chases|X], X).
```

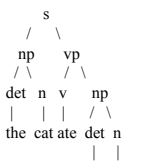

The parser (accepts correct sentences)

```
sentence --> nounphrase, verbphrase.
nounphrase --> determiner, noun.
nounphrase --> determiner, adjective, noun.
verbphrase --> verb, nounphrase.

determiner --> [the].
determiner -->[a] .
noun --> [dog].
noun --> [bone].
noun --> [mouse].
noun --> [cat].
verb --> [ate].
verb --> [chases].
adjective --> [big].
adjective --> [brown].        adjective --> [lazy].
```


Parser (produces parse trees)

```
?- sentence(Tree,[the,cat,ate,the,mouse],[]).
   Tree = s(np(det(the),n(cat)),vp(v(ate),np(det(the),n(mouse))))
```

- **Parse tree** (the names of the functors are the names of the nonterminals in the CFG grammar or short notations for them).



The final parser

```
sentence(s(NP,VP)) --> nounphrase(NP), verbphrase(VP).
nounphrase(np(D,N)) --> determiner(D), noun(N).
nounphrase(np(D,A,N)) -->determiner(D),adjective(A), noun(N).
verbphrase(vp(V,NP))--> verb(V), nounphrase(NP).

determiner(det(the)) --> [the].
determiner(det(a)) -->[a].
noun(n(dog)) --> [dog].
noun(n(bone)) --> [bone].
noun(n(mouse)) --> [mouse].
noun(n(cat)) --> [cat].
verb(v(ate)) --> [ate].
verb(v(chase)) --> [chases].
adjective(adj(big)) --> [big].
adjective(adj(brown)) --> [brown].
adjective(adj(lazy)) --> [lazy].
```

Another example: parsing numbers

<real-number> ::= <part> . <part>
<part> ::= <digit> | <digit> <part>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- 3.519 is an example "sentence" in the language. Draw its parse tree.
- In our parser code below, we will assume that the sentence has been read and stored as a list of atoms, such as [3, ., 5, 1, 9].

Using DCG notation for difference lists

```
real_number --> part, [.], part.
part --> digit.
part --> digit, part.

digit --> [H],{ member(H, [0,1,2,3,4,5,6,7,8,9]) }.
```

- Or equivalently

```
digit([H|Rest], Rest):- member(H, [0,1,2,3,4,5,6,7,8,9]).
```

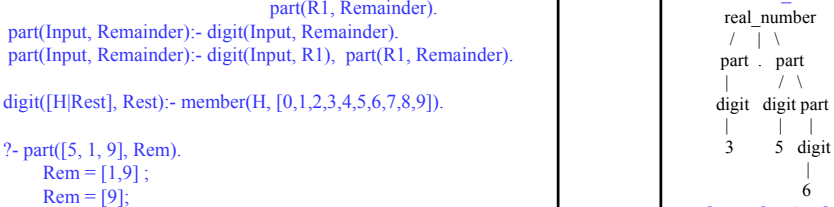
- Note: Between curly brackets you can put normal Prolog code.

Equivalent Prolog code

```
real-number(Input, Remainder):- part(Input, [.|R1]),
                                part(R1, Remainder).
part(Input, Remainder):- digit(Input, Remainder).
part(Input, Remainder):- digit(Input, R1),  part(R1, Remainder).

digit([H|Rest], Rest):- member(H, [0,1,2,3,4,5,6,7,8,9]).

?- part([5, 1, 9], Rem).
      Rem = [1,9] ;
      Rem = [9];
      Rem = [];
      no
```


Adding the parse trees

```
?- real_number(Tree, [3,.,5,6],[]).
  Tree = real_number(part(digit(3)), '.',part(digit(5), part(digit(6))))
```




```
real_number(real_number(P1,.,P2)) --> part(P1), [.], part(P2).
part(part(D)) --> digit(D).
part(part(D,P)) --> digit(D), part(P).
digit(digit(H)) --> [H],{member(H, [0,1,2,3,4,5,6,7,8,9])}.
```


Testing

```
?- real_number(Tree, [3,.,5,6],[]).
  Tree = real_number(part(digit(3)), '.',part(digit(5), part(digit(6))))

?- part(Str, [5, 1, 9], Rem).
Str = part(digit(5))
Rem = [1, 9] ;

Str = part(digit(5), part(digit(1)))
Rem = [9] ;

Str = part(digit(5), part(digit(1), part(digit(9))))
Rem = [] ;

No
```

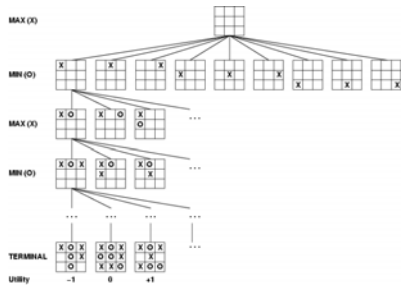

Games

- Two persons, adversarial games
- One person and the computer
- Examples: tic-tac-toe, chess, checkers, go
- Optimal decisions
- MiniMax algorithm, α-β pruning
- Imperfect, real-time decisions

Games vs. search problems

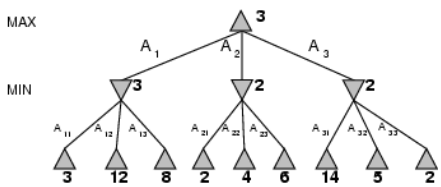
- "Unpredictable" opponent → specifying a move for every possible opponent reply
- Time limits → unlikely to find goal, must approximate

Game tree (2-player, deterministic, turns)



Minimax

- Perfect play for deterministic games
- Idea: choose move to position with highest minimax value
  = best achievable payoff against best play
- E.g., 2-ply game:



The minimax principle

```
% minimax( Pos, BestSucc, Val):
%    Pos is a position, Val is its minimax value;
%    best move from Pos leads to position BestSucc

minimax( Pos, BestSucc, Val)  :-
  moves( Pos, PosList), !,   % Legal moves in Pos produce PosList
  best( PosList, BestSucc, Val)
   ;
  staticval( Pos, Val).  % Pos has no successors: evaluate statically

best( [ Pos], Pos, Val)  :-  minimax( Pos, _, Val), !.
```


The minimax principle (cont.)

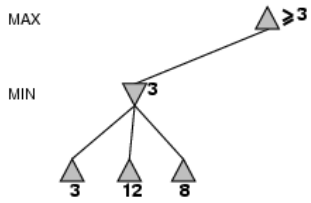
```
best( [Pos1 | PosList], BestPos, BestVal)  :-
    minimax( Pos1, _, Val1),    best( PosList, Pos2, Val2),
    betterof( Pos1, Val1, Pos2, Val2, BestPos, BestVal).

betterof( Pos0, Val0, Pos1, Val1, Pos0, Val0)  :-
                                % Pos0 better than Pos1
  min_to_move( Pos0),           % MIN to move in Pos0
  Val0 > Val1, !                % MAX prefers the greater value
  ;
  max_to_move( Pos0),       % MAX to move in Pos0
  Val0 < Val1, !.           % MIN prefers the lesser value

betterof( Pos0, Val0, Pos1, Val1, Pos1, Val1).
% Otherwise Pos1 better than Pos0
```


Properties of minimax

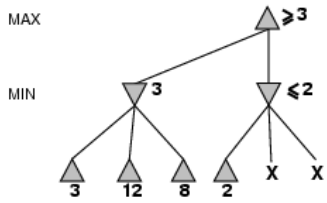
- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? $O(b^m)$
- Space complexity? O(bm) (depth-first exploration)

- For chess, b ≈ 35, m ≈100 for "reasonable" games
  → exact solution completely infeasible

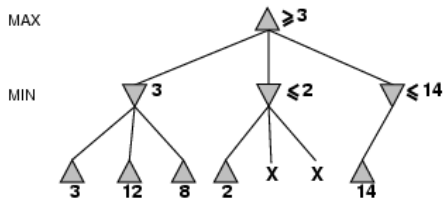
α-β pruning example



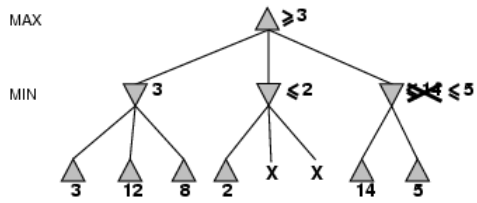
α-β pruning example



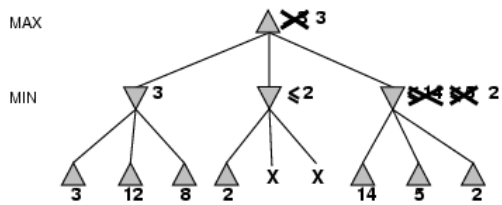
α-β pruning example



α-β pruning example



α-β pruning example



Resource limits

Suppose we have 100 secs, explore 10^4 nodes/sec
→ 10^6 nodes per move

Standard approach:

- cutoff test:
  e.g., depth limit
- evaluation function
  = estimated desirability of position

Cutting off search

Does it work in practice?
$b^m = 10^6$, b=35 → m=4

4-ply look-ahead is a hopeless chess player!
- 4-ply ≈ human novice
- 8-ply ≈ typical PC, human master
- 12-ply ≈ Deep Blue, Kasparov

Deterministic games in practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.
- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.
- Go: human champions refuse to compete against computers, who are too bad. In go, *b > 300*, so most programs use pattern knowledge bases to suggest plausible moves.

Meta-interpreters

```
% A basic Prolog meta-interpreter
% A Prolog interpreter written in Prolog

prove(true).

prove( (Goal1, Goal2)) :- prove(Goal1),
                          prove(Goal2).

prove(Goal) :- clause(Goal, Body),
               prove(Body).

% Useful to add tracing and proof trees
```


Example

```
?- trace((member(X,[a,b]),member(X,[b,c]))).
Call: member(_0085,[a,b])
Exit: member(a,[a,b])
Call: member(a,[b,c])
   Call: member(a,[c])
      Call: member(a,[])
      Fail: member(a,[])
   Fail: member(a,[c])
Fail: member(a,[b,c])
Redo: member(a,[a,b])
   Call: member(_0085,[b])
   Exit: member(b,[b])
Call: member(b,[b,c])
Exit: member(b,[b,c])
```


A Prolog meta-interpreter for tracing programs

```
% trace( Goal): execute Prolog goal Goal displaying trace information

trace( Goal) :- trace( Goal, 0). % traces Goal at once
                                 % no need to trace step by step
trace( true, Depth) :- !.                % Cut; Depth = depth of call

trace( ( Goal1, Goal2), Depth) :- !,     % Cut
  trace( Goal1, Depth),
  trace( Goal2, Depth).

trace( Goal, Depth) :-
  display( 'Call: ', Goal, Depth),
  clause( Goal, Body),
  Depth1 is Depth + 1,
  trace( Body, Depth1),
  display( 'Exit: ', Goal, Depth),
  display_redo( Goal, Depth).
```


A Prolog meta-interpreter for tracing programs (cont.)

```
trace( Goal, Depth) :-                   % All alternatives exhausted
  display( 'Fail: ', Goal, Depth),
  fail.

display( Message, Goal, Depth) :-
  tab( Depth), write( Message),
  write( Goal), nl.

display_redo( Goal, Depth) :-
  true                                   % First succeed simply
  ;
  display( 'Redo: ', Goal, Depth),       % Then announce backtracking
  fail.                                  % Force backtracking
```

Theorem proving

- Given a formula show that is it true
- Resolution principle: show that negating the theorem leads to a contradiction
- Propositional calculus
- Examples:
  - `p V ~p`
  - `(a=>b) & (b=>c) => (a=>c)`

Conjunctive normal form

- Transform into conjunction of clauses

```
~((a=>b) & (b=>c) => (a=>c))
= ~(~((a=>b) & (b=>c)) V (a=>c))
= ((a=>b) & (b=>c)) & ~(a=>c)
= (~a V b) & (~b V c) & ~(~a V c)
= (~a V b) & (~b V c) & a & ~c
```


Translating a propositional calculus formula into a set of (asserted) clauses

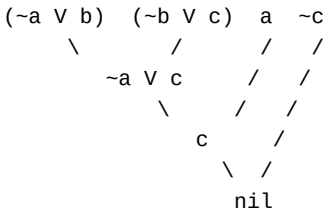
```
:- op( 100, fy, ~).      % Negation
:- op( 110, xfy, &).     % Conjunction
:- op( 120, xfy, v).     % Disjunction
:- op( 130, xfy, =>).    % Implication

% translate( Formula): translate propositional Formula
%   into clauses and assert each resulting clause C as clause( C)

translate( F & G) :-     !,          % Translate conjunctive formula
  translate( F),  translate( G).

translate( Formula) :-
  transform( Formula, NewFormula),   !,   % Transformation step on Formula
  translate( NewFormula).

translate( Formula) :-               % No more transformation possible
  assert( clause( Formula)).
```


Translating formulas (cont.)

```
% Transformation rules for propositional formulas
% transform( Formula1, Formula2) if
%   Formula2 is equivalent to Formula1, but closer to clause form

transform( ~(~X), X).                    % Eliminate double negation
transform( X => Y, ~X v Y).              % Eliminate implication
transform( ~ (X & Y), ~X v ~Y).          % De Morgan's law
transform( ~ (X v Y), ~X & ~Y).          % De Morgan's law
transform( X & Y v Z, (X v Z) & (Y v Z)).  % Distribution
transform( X v Y & Z, (X v Y) & (X v Z)).  % Distribution
transform( X v Y, X1 v Y) :- transform( X, X1).  % Transform subexpression
transform( X v Y, X v Y1) :- transform( Y, Y1). % Transform subexpression
transform( ~ X, ~ X1) :- transform( X, X1).      % Transform subexpression
```


Resolution

- p V q and ~p V r gives q V r
- Contradiction: p & ~p



A small interpreter for pattern-directed programs

```
:- op( 800, xfx, --->).
% run: execute production rules of the form
% Condition ---> Action until action `stop' is triggered
run :-  Condition ---> Action,       % A production rule
        test( Condition),            % Precondition satisfied?
        execute( Action).

% test( [ Condition1, Condition2, ...])  if all conditions true
test( []).                           % Empty condition
test( [First|Rest]) :- call( First), test( Rest). % Test conjunctive condition

% execute( [ Action1, Action2, ...]): execute list of actions
execute( [ stop]) :- !.              % Stop execution
execute( []) :-  run.                % Empty action (execution cycle completed)
                                     % Continue with next execution cycle
execute( [First | Rest]) :-  call( First),  execute( Rest).

replace( A, B) :- retract( A), !, assert( B).
% Replace A with B in database, Retract once only
```

Theorem prover

```
:- dynamic clause/1, done/3.

% Production rules for resolution theorem proving
% Contradicting clauses

[ clause( X), clause( ~X) ] ---> [ write('Contradiction found'), stop].

% Remove a true clause
[ clause( C), in( P, C), in( ~P, C) ]  ---> [ retract( C) ].

% Simplify a clause
[ clause( C), delete( P, C, C1), in( P, C1) ]  --->[ replace( clause( C), clause( C1)) ].

% Resolution step, a special case
[ clause( P), clause( C), delete( ~P, C, C1), not done( P, C, P) ] --->
[ assert( clause( C1)), assert( done( P, C, P)) ].
```


Theorem prover (cont.)

```
% Resolution step, general case
[ clause(C1), delete( P, C1, CA),
  clause(C2), delete( ~P, C2, CB),  not done( C1,C2,P) ]  --->
[ assert( clause(CA v CB)), assert( done( C1, C2, P)) ].

% Last rule: resolution process stuck
[] ---> [ write('Not contradiction'), stop].

% delete( P, E, E1)  if  deleting a disjunctive subexpression P from E gives E1
delete( X, X v Y, Y).
delete( X, Y v X, Y).
delete( X, Y v Z, Y v Z1)  :-  delete( X, Z, Z1).
delete( X, Y v Z, Y1 v Z)  :- delete( X, Y, Y1).

% in( P, E)  if P is a disjunctive subexpression in E
in( X, X).
in( X, Y)  :-   delete( X, Y, _).
```