Prolog (part 2)

Andrea Sterbini – sterbini@di.uniroma1.it

Recall:

FACT

```
term(arg1, arg2, arg3 …).          % rule always true
```

RULE

```
head(arg1, arg2, …) :-             % to prove this head
    body1(…),                      % we must prove this
    body2(…),                      % AND this
    …                                  % …
    bodyN(…).                      % AND this
                                   % (SEQUENTIALLY)
```

Details on rule execution

To prove a predicate (e.g. a prolog term) we must search for either:

1) a rule with the same head (should unify with the term to prove)
2) or a fact with same term (which also should unify)

i.e.:

- the term functor must be the same
- the number of arguments must be the same
- each argument must recursively unify with the corresponding argument

This is generally used to selectively match the predicate clauses

Arguments can be used both as Input or as Output depending on their binding

There is no return value, you can use any argument as output

Lists (dynamic, heterogeneous)

```
List = [ one, two, three, four ]   % list syntax

[ Head | Tail ]    = List          % how to extract the first element
    Head           = one           % fails if the list is empty
    Tail           = [ two, three, four ]

[ First, Second | Rest ] = List    % extracting first and second element
    First          = one           % fails if the list has less than 2 elements
    Second         = two
    Rest           = [ three, four ]

EmptyList = []                     % the empty list

is_empty([]).                      % test for empty list through unification

length( [], 0).                    % base case: an empty list has length 0
length( [H|Tail], N1) :-           % recursive case: if there is at least 1 element
    length(Tail,N),                % N = length of a list with one element less
    N1 is N + 1.                   % plus 1
```

Predicates on lists

```
% list concatenation (or split if used backward)
append([], B, B).                    % B if A is empty
% else attach first element in front of B appended to the rest of A
append([ Head | Tail], B, [Head | C] ) :-
    append(Tail,B,C).

% member check / generation
member( A, [ A | _ ] ).              % A is member if is the first element
member( A, [ _ | Tail] ) :-
    member(A, Tail).                 % or if is member of the rest
        % NOTICE: member should fail if the list is empty
```

Functional programming

Predicates can be used as if they were functions or to test values
You just add an argument to collect the result

```
square( X, Result ) :- Result is X * X.          % function
is_odd(X) :- 1 is X mod 2.                       % test=compute+unify
```

You can map functions over lists

```
List = [ 1, 2, 3, 4 ], maplist( square, List, List1 ).
    =>  List1 = [ 1, 4, 9, 16 ]
```

Or get all elements satisfying some property

```
List = [1, 2, 3, 4], include(is_odd, List, Odd).
    =>  Odd = [1, 3]
List = [1, 2, 3, 4], partition(is_odd, List, Odd, Even).
    =>  Odd = [1, 3]          Even = [2, 4]
```

How partition could be defined

```
% if there are no elements we produce two empty lists
part( _Predicate, [], [], [] ).

part( Predicate, [H|T], [H|T1], T2 ) :-
      call(Predicate, H),             % if the H satisfies the Predicate
      part(Predicate, T, T1, T2).     % H is added in front of the first list

part( Predicate, [H|T], T1, [H|T2] ) :-
      not(call(Predicate,H)),         % else
      part(Predicate, T, T1, T2).     % H is added in front of the second list
```

Notice: this predicate can be used both to partition and to join list … why?

What if predicates are used “backward”?

```
% find a list X that is partitioned this way
part(is_odd, X, [1,3], [2,4]).

[1,3,2,4] ; [1,2,3,4] ; [1,2,4,3] ; [2,1,3,4] ; [2,1,4,3] ; [2,4,1,3] % 6 possible lists!!

% What if we use maplist “backward”?
maplist(square, X, [1, 4, 9]).          % is cannot be used “backward” in square
                    Arguments are not sufficiently instantiated
                    In:  [3] 1 is _1680*_1682
% We need a better definition of square(N,N2) that works forward and backward
square(N, N2) :-
      nonvar(N),                         % if N is known
      N2 is N*N.                         % compute N2=N*N
square(N, N2) :- var(N),                 % else if N is a variable
      between(0,N2,N),                   % look for some integer N between 0 and N
      N2 is N*N.                         % such that N*N = N2
```

Meta-programming

You can build terms from lists and viceversa with =..

term(1, two, three) =.. [term, 1, two, three]

You can apply / call predicates by adding other arguments

apply(Predicate, AdditionalArgsList) OR

call(Predicate, AdditionalArg1, Arg2, …)

(this allows us to use partial predicates)

You can add/remove new facts or clauses to/from rule memory

```
% add at the beginning              % add at the end
asserta( Head :- Body )             assertz( Head :- Body )
asserta( Fact )                     assertz( Fact )
    retract( FactOrClause )         % delete FIRST matching rule
    retractall( FactOrClause )      % delete ALL matching rules
```

Definite Clause Grammars (DCG)
an alternative syntax to write parsers/generators

Two arguments are added to each grammar rule head / body:
- the <u>list of input tokens</u> to be recognized
- the <u>remaining list of tokens</u> not consumed yet

RULE READ FROM FILE

```
sentence -->
      subject,
      verb,
      complement.

%special: terminal tokens as lists
verb --> [ run ].
```

IS TRANSFORMED TO

```
sentence( Words, Rest3 ) :-
            subject( Words, Rest1 ),
            verb( Rest1, Rest2 ),
            complement(Rest2, Rest3).

% are simply expected as next token
verb( [ run | Rest ], Rest ).
```

Grammar example (with gender agreement)

```
sentence          --> subject, verb, direct_object.
subject           --> article(Gender),
                       actor(Gender).    % same gender for article & actor
direct_object     --> article(Gender),
                       object(Gender).   % same gender for article & object
article( female ) --> [ la ].            % female article
article( male )   --> [ il ].            % male article
actor( _ )        --> [ chirurgo ].      % surgeon is male/female in Italian
actor( female )   --> [ elefantessa ].   % female elephant
actor( male )     --> [ elefante ].      % male elephant
verb              --> [ mangiava ].      % was eating
verb              --> [ guardava ].      % was looking
object( female )  --> [ insalata ].      % salad is female in Italian
object( male )    --> [ cavolfiore ].    % cauliflower is male in Italian
```

We can use the grammar to generate all possible sentences
?- phrase(sentence, WordList)

[la, chirurgo, mangiava, la, insalata] % the female surgeon was eating the salad
[la, chirurgo, mangiava, il, cavolfiore] % the female surgeon was eating the cauliflower
[la, chirurgo, guardava, la, insalata] % ... looking the salad
[la, chirurgo, guardava, il, cavolfiore] % … looking the cauliflower
[la, elefantessa, mangiava, la, insalata] % the female elephant was eating the salad
[la, elefantessa, mangiava, il, cavolfiore] % the f. elephant was eating the cauliflower
[la, elefantessa, guardava, la, insalata]
[la, elefantessa, guardava, il, cavolfiore]
[il, chirurgo, mangiava, la, insalata]
[il, chirurgo, mangiava, il, cavolfiore]
[il, chirurgo, guardava, la, insalata]

… %TASK: how can we add also singular/plural constraints?

Or to parse (recognize) a sentence and get the parse tree (DEMO)

English grammar example with singular / plural agreement

% a sentence is a NounPart followed by a VerbPart with the same Number
s(s(NP,VP)) --> np(NP, Num), vp(VP, Num).

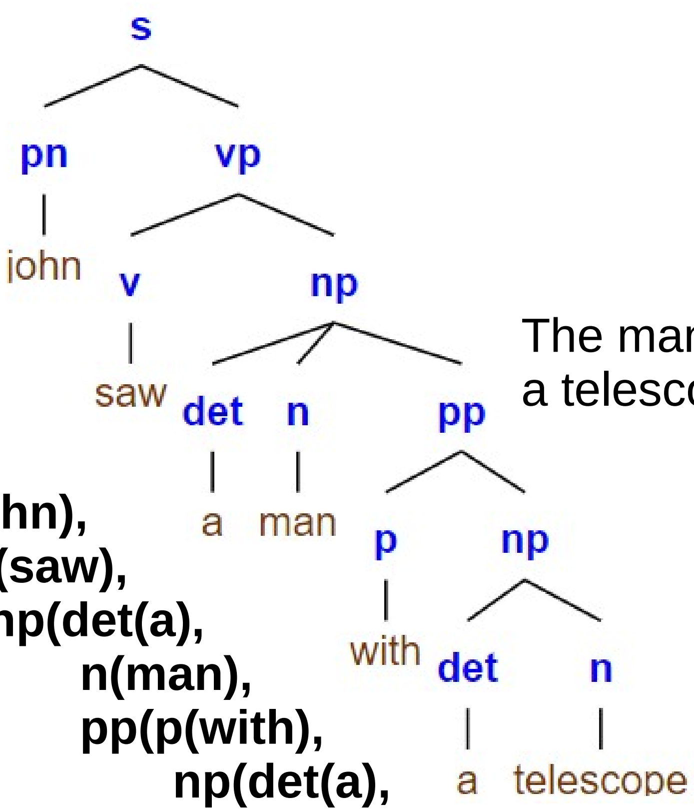
% a NP could be a PersonName
np(NP, Num) --> pn(NP, Num).
% or an Article followed by a Name with the same Number
np(np(Det,N), Num) --> det(Det, Num), n(N, Num).
% or an Article, a Name and a PredicatePart with the same Number
np(np(Det,N,PP), Num) --> det(Det, Num), n(N, Num), pp(PP).

% a VerbPart can be a Verb followed by a NounPart
vp(vp(V,NP), Num) --> v(V, Num), np(NP, _).
% or a Verb followed by a NounPart and a PredicatePart
vp(vp(V,NP,PP), Num) --> v(V, Num), np(NP, _), pp(PP).

Or to parse (recognize) a sentence and get the parse tree (DEMO)

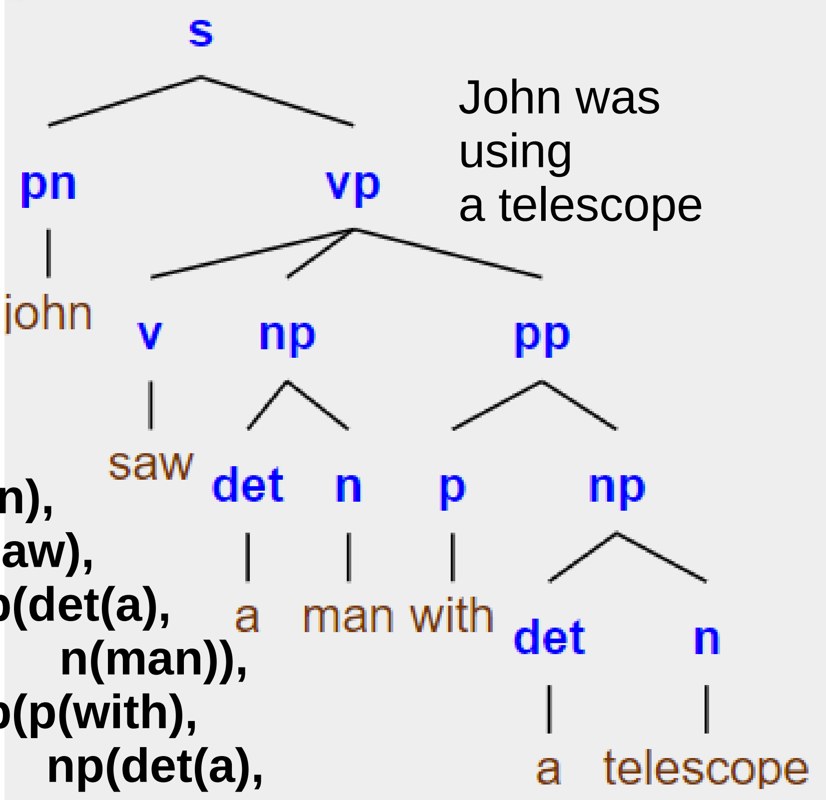
```
% a PredicatePart is a Preposition followed by a NounPart
pp(pp(P,NP)) --> p(P), np(NP, _).

det(det(a), sg) --> [a].                      % singular article
det(det(the), _) --> [the].                   % article

pn(pn(john), sg) --> [john].                  % person name (singular)

n(n(man), sg) --> [man].                      % singular name
n(n(men), pl) --> [men].                      % plural name
n(n(telescope), sg) --> [telescope].          % ...

v(v(sees), sg) --> [sees].                    % singular verb
v(v(see), pl) --> [see].                      % plural verb
v(v(saw), _) --> [saw].                       % verb

p(p(with)) --> [with].                        % preposition
```

Two possible parse trees for the same sentence:
?- phrase(s(Tree), [john, saw, a, man, with, a, telescope]).



The man had a telescope

```
s(pn(john),
   vp(v(saw),
        np(det(a),
             n(man),
             pp(p(with),
                  np(det(a),
n(telescope))))))
```



John was using a telescope

```
s(pn(john),
   vp(v(saw),
        np(det(a),
             n(man)),
        pp(p(with),
             np(det(a),
n(telescope)))))
```

Common extensions

Grammars

grammar rules map easily to Prolog predicates,
both for parsing and for text generation

Constraints

the domain of the possible values of a variable
can be constrained in many ways (e.g. the sudoku game)

OOP

terms could represent objects and their properties
rules could represent methods

GUI

widgets, events, callbacks and so on

Examples

Limericks

Grammar

Constraints (Sudoku)

Algebraic simplification?

Algebraic derivatives?