

## Advanced NLP in Prolog

### Metaparser in Prolog

It is sometimes convenient to make a parser that handles the parse trees and other information as a side effect, i.e. it is not necessary to explicitly have the syntax tree in the grammar. This motivates the concept of a *Metaparser*, where the grammar is represented as a set of Prolog facts, and where a separate metaparser program is interpreting this grammar dynamically. (Not as in DCG, where Prolog translates it into executable Prolog). The price is a layer of inefficiency, but for demonstrative purposes, it is an ideal tool.

We shall describe the principles using only a simple DCG Metainterpreter, but shall later use the same principle to implement more general language interpreters:

- Parsers for extended language formalisms (Categorical Grammars)
- Parsers that uses other strategies (Bottom up parsers, Chart Parsers)

A grammar that is represented in this way is called an M-grammar (M for meta, but metagrammar means something different).

To distinguish these grammars from DCG grammars, we use another production symbol

`' ' ---> ' '` which has to be declared by

```
:-op(1100,xfy,--->)
```

The grammar above is then stated as an M-grammar as

```
s ---> np, vp.
```

```
np ---> proper_noun.
```

```
vp ---> verb, np.
```

```
verb ---> txt(loves).
```

```
proper_noun ---> [john].
```

```
proper_noun ---> [mary].
```

The metaparser can now be formulated

```
:-op(1100,xfy,--->).
```

```
metaparse(S,List):-  
    parse(S,PSTree,List,[]),  
    prettyprint(PSTree).
```

```
parse(S,prod(S,PSTree),X0,X1):-  
    (S ---> PS),  
    parse(PS,PSTree,X0,X1).
```

```
parse((First,Rest),(FirstTree,RestTree),X0,X2):-  
    parse(First,FirstTree,X0,X1),  
    parse(Rest,RestTree,X1,X2).
```

```
parse([Word],[Word],[Word|Rest],Rest).
```

(The prettyprint will be shown later.)

We get a formatted parse tree by the call

```
?-metaparse(s,[john,loves,mary]).
```

```
==>
```

```
s
  np
    proper_noun
      [john]
  vp
    verb
      [loves]
    np
      proper_noun
        [mary]
```

## Metagrammar in DCG

It is an amusing consequence that the above parser can be formulated as a DCG grammar. In a way, we have implemented DCG in DCG, because Prolog translates this metagrammar exactly to the parser program above. A grammar to describe a grammar formalism is called a *metagrammar*.

```
parse((First,Rest),(FirstTree,RestTree)) -->
    parse(First,FirstTree),
    parse(Rest,RestTree).
```

```
parse(S,prod(S,PSTree)) -->
    {S ---> PS},
    parse(PS,PSTree).
```

```
parse([Word],[Word]) --> [Word].
```

## Parsing with Ambiguities

As another amusing example, consider the sentence

Time flies like an arrow .

This innocuous little example is sometimes used to show the futility of trying to analyse natural language sentences, because it has so many interpretations:

- Time passes very quickly
- The time is flying with the same speed as an arrow.
- The time is flying in a way similar to arrows.
- Some time-related flies are fond of arrows. (as fruit flies like a banana )
- Some flies called "Time" flies are fond of arrows.
- Take the time of flies quickly !
- Take the time of flies in a way similar to how an arrow would do it!
- Take the time of flies with arrow form !
- Take the time of flies in the same way as you would take the time of an arrow !
- etc.

A grammar for this may not necessarily be unambiguous:

Consider the grammar, which we represent as a M-grammar.

```
sentence      ---> noun_phrase, verb_phrase .
sentence      ---> verb, noun_phrase, comparison .
noun_phrase   ---> noun .
```

noun\_phrase ---> proper\_noun.  
noun\_phrase ---> article,noun.  
noun\_phrase ---> composite\_noun.  
verb\_phrase ---> verb,object.  
verb\_phrase ---> verb,comparison.  
object ---> noun\_phrase.  
comparison ---> comparator,noun\_phrase.  
composite\_noun ---> proper\_noun,noun.

verb ---> [time].  
verb ---> [flies].  
verb ---> [like].  
comparator ---> [like].  
noun ---> [time].  
noun ---> [flies].  
noun ---> [arrow].  
proper\_noun ---> [time].  
article ---> [an].

?-metaparse(sentence,[time,flies,like,an,arrow]),fail.

sentence  
  noun\_phrase  
    noun  
      [time]  
  verb\_phrase  
    verb  
      [flies]  
    comparison  
      comparator  
      [like]  
    noun\_phrase

sentence  
  noun\_phrase  
    proper\_noun  
      [time]  
  verb\_phrase  
    verb  
      [flies]  
    comparison  
      comparator  
      [like]  
    noun\_phrase

article [an] noun [arrow]	article [an] noun [arrow]
sentence noun_phrase composite_noun proper_noun [time] noun [flies] verb_phrase verb [like] object noun_phrase article [an] noun [arrow]	sentence verb [time] noun_phrase noun [flies] comparison comparator [like] noun_phrase article [an] noun [arrow]

While ambiguity in programming languages is considered an unacceptable fault of the language itself, we have to live with it in natural language, because it is inherently ambiguous at all levels. A later semantic analysis will know the ambiguity of "like", and can check in a semantic net what attributes are meaningful, i.e

Time can fly    (in a certain sense)

A fly has no attribute which can have a value Time

It is not relevant to take the time of flies.

etc.

It is important to find the right sense of the word. This is often possible by taking into account the grammar, and also the semantic classes of the nouns involved. In general, finding the right sense also gives the right semantic code, i.e. we can safely forward the problem to the pragmatic processing.

## Consensical Grammar

TUC's grammar formalism is called *Consensical Grammar* which means

Context Sensitive Categorical Attribute Logic Grammar

and is a reminder of all the extensions of context free grammars that are needed to cope with natural language parsing. They are

- Context sensitive

The grammar formalism is in effect an extension of context free grammars, and implements at least a proper subset of context sensitive grammars, as will be shown below

- Categorical

Categorical grammar is a name of a grammar type that uses operators to manipulate the combination of subphrases into larger phrases. Consensical grammar borrows some of its notation from classical categorical grammar.

- Attribute

The classical context sensitive grammars use names of syntactic categories without arguments. However, this grammar allows attributes and attribute variables to be embedded in the grammar besides translation schemes in the form of extra conditions on the attributes.

- Logic

The grammar is based on Definite Clause Grammars, and inherits all the logic machinery that is

imbedded in Prolog. However, the grammar formalism as such is not dependent on a top down parsing method.

Consensical grammar relies heavily on the principles of the Extraposition grammars which was used to implement the CHAT-80 systems.

The examples below show that this categorial grammar has the generating capacity that goes beyond context free grammars.

## Language Hierarchy

The languages, both formal and natural, can be classified into a hierarchy that was devised by Chomsky. The hierarchy sets up the relation between the languages defined as sets of strings, the grammar formalism and the kind of automata that are able to accept these strings, and no others.

## Regular Languages And Grammars

The simplest languages are regular languages. Regular languages form the model for low level morphological or lexical analysis, both in programming and natural languages.

(Regular language are accepted by Finite State Automata). Regular grammars can be made into a special case of context free languages in that all productions are of the form

$$\begin{aligned} S_1 &\rightarrow t \\ S_1 &\rightarrow t S_2 \end{aligned}$$

where  $t$  denotes a terminal symbol,  $S_1, S_2$  are non-terminals, and all the nonterminals appear at the end of the right sides.

An example of a regular language is the set of strings  $a^* b^*$  which means means none or more repetitions of a and b. We could say that this represents the set of strings  $a^m b^n$  for various unrestricted integers m and n.

S  $\rightarrow$  A B  
A  $\rightarrow$  a A  
A  $\rightarrow$   
B  $\rightarrow$  b B  
B  $\rightarrow$

There is a limitation in regular languages, that is due the lack of memory in FSA. For example the following language is not regular:

The set of strings  $a^n b^n$  for any n, which means a repetition of n a's followed by an equal number of b's.

(We can say that regular grammars cannot count).

## Context Free Languages And Grammars

Context free languages have been introduced already. They are defined by productions that contains single left side nonterminal symbols, and unrestricted right sides.

(They are accepted by (one-stack) Pushdown automata (PDA)). Context free languages are larger than regular languages. For example, the non regular language above is easily defined by the grammar

$$S \rightarrow a S b$$
$$S \rightarrow$$

(We can say that context free languages can count, but only two things.)

## Context Sensitive Languages And Grammars

Context sensitive grammars allow the left hand sides to contain more than one symbol, so that a production is only allowed when all the left side symbols match with the text. (Context sensitive languages can be accepted by a two-stack Pushdown automaton).

$$\begin{array}{l} X A \rightarrow X a \\ X \rightarrow \end{array}$$

'A' can produce 'a', but only allowed in the context of X, and X shall be unchanged.

An example of a language that is not context free is

$$a^n b^n c^n$$

where all the n's denote the same number of repetitions. However, this language is context sensitive. (We can say that context sensitive languages can count more than two things.)

$$S \rightarrow S1$$
$$S1 \rightarrow X S1 S2$$
$$S1 \rightarrow$$
$$S2 \rightarrow A b S2 c$$
$$S2 \rightarrow$$
$$X A \rightarrow a$$

The latter grammar can be reformulated into categorial grammar as follows. We note that the purpose of X is to push an A on a stack (waiting list). This is expressed as follows, using the extraposition operator.

Instead of

$X \ A \ \rightarrow \ a$

we write

$X \ \rightarrow \ []-A \ a$

In Prolog form, the grammar is

```
s ----> s1.

s1 ----> x,s1,s2.
s1 ----> [].

s2 ----> a1,[b],s2,[c].
s2 ----> [].

x ----> []-a1,[a].
```

The run examples indicate that grammar copes with the counting.

E: a a b b c c

```
s
  s1
    x
      []-a1
      [a]
    s1
      x
        []-a1
        [a]
      s1
        []
      s2
        a1-a1
        []
        [b]
      s2
        a1-a1
```

```
          []
         [b]
        s2
         []
        [c]
       [c]
      s2
     []
```

==> yes

E: a a b b c .

==> no

## Consensical Grammar Applied to NL

In the following is an example of a grammar, a sentence and a parse tree in Consensical grammar.

```
sentence ---> statement,['.'].
sentence ---> question,['?'].

statement ---> np,vp.

question ---> [which], statement \ det.

np ---> det,noun,relclause.
np ---> pnoun.

vp ---> tv,np.
vp ---> iv.

relclause ---> [that],
               statement // np. %% NB

relclause ---> [].

begin ---> []-[lock].
end ---> [lock].
```

## % Lexicals

det ---> [a].  
det ---> [the].  
det ---> [every].

noun ---> [man].  
noun ---> [woman].  
noun ---> [dog].  
noun ---> [cat].  
noun ---> [mouse].  
noun ---> [fish].

iv ---> [dies].  
iv ---> [lives].  
iv ---> [squeaks].

tv ---> [catches].  
tv ---> [chases].  
tv ---> [kills].  
tv ---> [kisses].  
tv ---> [likes].  
tv ---> [loves].

pnoun ---> [fred].  
pnoun ---> [john].  
pnoun ---> [mary].

## Sample sentences

The mouse that the cat that likes a fish chases squeaks.

The mouse that the cat that chases likes a fish squeaks.

Which mouse that a cat chases dies ?

E: the mouse that the cat that likes a fish chases sq

```
sentence
  statement
    np
      det
        [the]
      noun
        [mouse]
      relclause
        [that]
      statement-np
        np
          det
            [the]
          noun
            [cat]
          relclause
            [that]
          statement-np
            np-np          % the cat (gap)
              []
            vp
              tv
                [likes]
              np
                det
                  [a]
                noun
                  [fish]
              relclause
```

```

                                []
                                vp
                                tv
                                [chases]
                                np-np
                                []
                                % the mouse (gap)

vp
  iv
  [squeaks]
[.]

==> yes

```

The next example shows an example where a relative clause cannot be formed.

E: the mouse that the cat that chases likes a fish squ

==> no

## Restriction on Movements

### Forward Application Move Restriction

A definition of a relative clause would be defined using the - operator:

```
relclause ---> [that],
                statement - np.
relclause ---> [].
```

However, this definition would accept the following sentence:

The mouse that the cat that chases likes a fish squeaks.  
(although with the same reasonable analysis as for the original sentence.). In order to avoid this, we used the // operator instead.

```
relclause ---> [that],
                statement // np.
relclause ---> [].
```

The difference here is that the inward application // will not succeed unless the np is used by the same statement, while the - would allow the np to be stacked and used ad lib.

### Non Conjunction Move Restriction

We can easily extend the grammar above to allow more than one noun phrase instead of a simple one. We can do that by extending the grammar:

np ---> np1, andnps.

andnps ---> [and], np.

andnps ---> [ ].

np1 ---> det, noun, relclause.

np1 ---> pn.

This will allow the sentences

John and Fred lives.

From earlier, we can recognise a sentences

Fred loves Mary.

A man that lives loves Mary.

This is grammatical. It means that there is woman X, a man Y, Y lives and Y loves X.

Now consider the sentence.

A man that John and ( ) lives loves Mary.

Here, the gap is placed inside the conjunction

John and ( )

The logical meaning is that there is an X (e.g. Fred) so that John and X live, and X loves Mary. However, this construction is strictly forbidden in English.

Again, the grammar can be remedied, but we shall not give the explicit details since this overacceptance seems to be rather harmless for language understanding.

## Consensical Grammar Metaparser

The following program implements the elements of syntactic analysis of context sensitive categorial grammars. It is listed as an example of a *Metagrammar parser* which dynamically interprets the grammar rules, and perform a parse accordingly. The parser needs a lot of explanation, but even if you don't understand it now, it is an executable program that can be used for experimentation. In principle, it is the same parser as is used by TUC, although we have not incorporated agreement checks and code generation.

The program also appears in the appendix "Metaparser for phrase structured languages" together with a program 'readin' that scans input from user.

```
%% Categorial Grammar Meta Interpreter

:-op(1100,xfy,'--->'). %% Metagrammar Production
:-op(500,xfy, \ ).    %% Backwards application
:-op(500,xfy, / ).    %% Forwards application
:-op(400,yfx, // ).   %% Inwards application
:-op(500,yfx, - ).    %% Outwards application

% parse(PhraseStructure,Syntaxtree,GapStack1,GapStack2).

parse([X],[X],GS,GS) --> [X], %% read word from list
    {\+ frontgap(GS)}.        %% unless front gap

parse([],[],GS,GS) --> [].
```

```

parse( (X,Y) , (TX,TY) ,GS1,GS3) -->
    !,
    parse(X,TX,GS1,GS2) ,
    parse(Y,TY,GS2,GS3) .

parse(X \ U ,T-U,GS1,GS2)    -->
    parse(X,T,[gap(U,first)|GS1],GS2) .

parse(X - U ,T-U,GS1,GS2)    -->
    parse(X,T,[gap(U,free)|GS1],GS2) .

parse(X // U ,T-U,GS1,GS4)    -->
    {lock(GS1,GS2)} ,
    parse(X,T,[gap(U,free)|GS2],GS3) ,
    {unlock(GS3,GS4)} .

parse(X,prod(X,YT) ,GS1,GS2)  -->
    {X ---> Y} ,
    parse(Y,YT,GS1,GS2) .

parse(X,prod(X-X,[ ]),[gap(X,_)|Y],Y)  --> [ ] .

%% Runtime utility

lock(GS,[gap(lock,last)|GS]) .

unlock([gap(lock,last)|GS],GS) .

frontgap( [gap(_,first)|_] ) .

```

```

run :-
    scanner(Wordlist),
    syntax(Wordlist,Syntaxtree),
    prettyprint(Syntaxtree).

syntax(List,ST) :-
    parse(sentence,ST,[],[],List,[]).
    %% 2 extra arguments for compatibility with DCG expansion

prettyprint(ST):-
    nl,
    pretty(0,ST),
    nl.

pretty(N,prod(X,Y)-U):- !,
    tab(N),write(X-U),nl,N2 is N+3,
    pretty(N2,Y).

pretty(N,prod(X,Y)):- !,
    tab(N),write(X),nl,N2 is N+3,
    pretty(N2,Y).

pretty(N,(X,Y)):- !,
    pretty(N,X),
    pretty(N,Y).

pretty(N,ST):-tab(N),write(ST),nl.

:-compile(readin).

```

## EXERCISES

The exercises here are advanced.

1. Make an analysis of the classical Context Sensitive with Consensical Grammar, and find out if they are
2. Try to combine feature unification with consensical i.e. using feature unification as the main argument mechanism.
3. Try to combine chart parsing with consensical grammar
4. Try to combine consensical grammar and feature unification with chart parsing.