Introduction to AI: Prolog and Grammars

François Jacquenet

Laboratoire Hubert Curien
Université de Saint-Etienne
Francois.Jacquenet@univ-st-etienne.fr

November, 2020

UNIVERSITÉ DE LYON

UNIVERSITÉ JEAN MONNET SAINT-ÉTIENNE

Outline

Syntax - Grammar

- Consider a sentence
  - "The black cat watches the big dog"
  - It seems correct
- Consider another one
  - "cat dog black The big the watches"
  - It doesn't seem correct

The first one respects the english grammar, this not the case of the second one

Formal Grammar

A formal grammar is a tuple (V_n, V_t, R, S) where:

- V_n is a finite set of non-terminals
- V_t is a finite set of terminals
- R is a set of rewriting rules $X \to Y$ where X and Y are finite sequences of elements of V_n and V_t
- S is the start symbol of the grammar, it is an element of V_n

Formal Grammar

Example

- $V_n = \{A, B, D\}$
- $V_t = \{a, b, d\}$
- $R = \begin{cases} A \rightarrow a\,B \\ A \rightarrow a\,A \\ B \rightarrow b\,D \\ B \rightarrow b\,B \\ D \rightarrow d \\ D \rightarrow d\,D \end{cases}$
- $S = A$

Formal Grammar

Noam Chomsky, an American linguist (born in 1928) has designed the Chomsky hierarchy.
It is a classification of formal grammars into 4 classes

Grammar	Language Name	Rule Contraints
Type-0	Recursively enumerable	$\alpha \rightarrow \beta$ (α contains at least one non-terminal)
Type-1	Context-sensitive	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	$A \rightarrow \gamma$
Type-3	Regular	$A \rightarrow a$ and $A \rightarrow aB$

A and B are non-terminals
α, β and γ are sequences of terminals, non-terminals or empty
a is a terminal

Formal Grammar

Grammar	Language Name	Rule Contraints
Type-0	Recursively enumerable	$\alpha \rightarrow \beta$ (α contains at least one non-terminal)
Type-1	Context-sensitive	$\alpha A \beta \rightarrow \alpha\gamma\beta$
Type-2	Context-free	$A \rightarrow \gamma$
Type-3	Regular	$A \rightarrow a$ and $A \rightarrow aB$

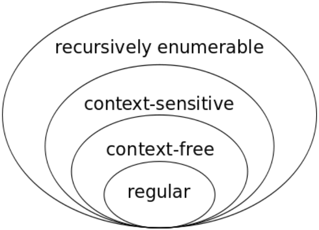

Figure 1: Inclusion of classes of grammars

Context-Free Grammar

A context-free grammar is a tuple (V_n, V_t, R, S) where:

- V_n is a finite set of non-terminals
- V_t is a finite set of terminals
- R is a set of rewriting rules $X \to Y$ where X is an element of V_n and Y is a finite sequence of elements of V_n and V_t
- S is the start symbol of the grammar, it is an element of V_n

Context-Free Grammar

The grammar below is an example of a Context-Free grammar

- $V_n = \{X\}$
- $V_t = \{a, b\}$
- $R = \left\{ \begin{array}{l} X \rightarrow a\,b \\ X \rightarrow a\,X\,b \end{array} \right.$
- $S = X$

Context-Free Grammar

Context-Free grammars can be used to model programming languages.
Consider the following small C programs

```
int a ;
float x ;
int f(int y)
{
    a = 42 ;
    if (a == y) {...}
    else {...}
    x = 3.14 ;
}
...
```

```
char word[256] ;
void p(int x ; float y)
{
    int a,b ;
    while (x < 0)
    {
        ...
    }
    ....
}
...
```

Context-Free Grammar

Those programs must respect the C programming language grammar

$$\begin{cases} Program \rightarrow listOfDeclarations\ listOfInstructions \\ \\ listOfDeclarations \rightarrow oneDeclaration \\ listOfDeclarations \rightarrow oneDeclaration\ listOfDeclarations \\ \\ listOfInstructions \rightarrow oneInstruction \\ listOfInstructions \rightarrow oneInstruction\ listOfInstructions \\ \\ oneInstruction \rightarrow assignment \\ oneInstruction \rightarrow conditional \\ oneInstruction \rightarrow loop \\ \\ \ldots \end{cases}$$

Context-Free Grammar

A grammar may be used to

- Generate sentences from the start symbol
- Parse a sentence to know if it is syntactically correct or not

For example, consider again the grammar below

$$\begin{cases} X \rightarrow a\,b \\ X \rightarrow a\,X\,b \end{cases}$$

It can be used to

- generate the sentences made up of n times the symbol 'a' followed by n times the symbol 'b' (n >= 1)
- parse a sentence to know if it respects the grammar

An example

Consider for example a Context-Free grammar that models a subset of the english language

$$
\begin{cases}
sentence \rightarrow noun\_phrase\ verbal\_phrase \\
\\
noun\_phrase \rightarrow determiner\ noun \\
\\
verbal\_phrase \rightarrow verb\ noun\_phrase \\
verbal\_phrase \rightarrow verb \\
\\
determiner \rightarrow the \\
determiner \rightarrow a \\
\\
noun \rightarrow cat \\
noun \rightarrow dog \\
\\
verb \rightarrow watches \\
\ldots
\end{cases}
$$

A first way to code in Prolog

We can write a Prolog program to model this grammar

We can do it with what we already know about Prolog:

```
sentence(S) :-
   noun_phrase(NP), verbal_phrase(VP), append(NP,VP,S).
noun_phrase(NP):-
   determiner(D), noun(N), append(D,N,NP).
verbal_phrase(VP):-
   verb(V), noun_phrase(NP), append(V,NP,VP).
verbal_phrase(VP):-
   verb(VP).
determiner([the]).
determiner([a]).
noun([cat]).
noun([dog]).
verb([watches]).
```

A first way to code in Prolog

If we want to know if a sentence is syntactically correct:

```
?- sentence([a, cat, watches, the,dog]).
```

If we want to generate all the possible sentences:

```
?- sentence(S).
```

We get:

```
S = [the, cat watches, the, cat]
S = [the, cat watches, the, dog]
S = [the, cat watches, a, cat]
S = [the, cat watches, a, dog]
S = [the, cat watches]
...
```

Weakness of this approach

While parsing a sentence, the process is not guided by the sentence.

Let us trace the resolution of the goal

```
?- noun_phrase([a,dog]).
```

and look at what happens

Weakness of this approach

```
[trace]  ?- noun_phrase([a,dog]).
   Call: (8) noun_phrase([a, dog]) ? creep
   Call: (9) determiner(_11762) ? creep
   Exit: (9) determiner([the]) ? creep
   Call: (9) noun(_11768) ? creep
   Exit: (9) noun([cat]) ? creep
   Call: (9) lists:append([the], [cat], [a, dog]) ? creep
   Fail: (9) lists:append([the], [cat], [a, dog]) ? creep
   Redo: (9) noun(_11768) ? creep
   Exit: (9) noun([dog]) ? creep
   Call: (9) lists:append([the], [dog], [a, dog]) ? creep
   Fail: (9) lists:append([the], [dog], [a, dog]) ? creep
```

Weakness of this approach

```
   Redo: (9) determiner(_11762) ? creep
   Exit: (9) determiner([a]) ? creep
   Call: (9) noun(_11768) ? creep
   Exit: (9) noun([cat]) ? creep
   Call: (9) lists:append([a], [cat], [a, dog]) ? creep
   Fail: (9) lists:append([a], [cat], [a, dog]) ? creep
   Redo: (9) noun(_11768) ? creep
   Exit: (9) noun([dog]) ? creep
   Call: (9) lists:append([a], [dog], [a, dog]) ? creep
   Exit: (9) lists:append([a], [dog], [a, dog]) ? creep
   Exit: (8) noun_phrase([a, dog]) ? creep
true.
```

A second way to code in Prolog

We can use append/2 at the beginning

```
sentence(S) :-
   append(NP,VP,S),noun_phrase(NP), verbal_phrase(VP).
noun_phrase(NP):-
   append(D,N,NP),determiner(D), noun(N).
verbal_phrase(VP):-
   append(V,NP,VP),verb(V), noun_phrase(NP).
verbal_phrase(VP):-
   verb(VP).
determiner([the]).
determiner([a]).
noun([cat]).
noun([dog]).
verb([watches]).
```

It remains inefficient!!!

A second way to code in Prolog

Let us trace the resolution of the goal

```
?- noun_phrase([a,dog]).
```

and look at what happens

```
?- noun_phrase([a,dog]).
   Call: (8) noun_phrase([a, dog]) ? creep
   Call: (9) lists:append(_2236, _2238, [a, dog]) ? creep
   Exit: (9) lists:append([], [a, dog], [a, dog]) ? creep
   Call: (9) determiner([]) ? creep
   Fail: (9) determiner([]) ? creep
   Redo: (9) lists:append(_2236, _2238, [a, dog]) ? creep
   Exit: (9) lists:append([a], [dog], [a, dog]) ? creep
   Call: (9) determiner([a]) ? creep
   Exit: (9) determiner([a]) ? creep
   Call: (9) noun([dog]) ? creep
   Exit: (9) noun([dog]) ? creep
   Exit: (8) noun_phrase([a, dog]) ? creep
true
```

A second way to code in Prolog

```
   Redo: (9) lists:append([a|_2224], _2244, [a, dog]) ? creep
   Exit: (9) lists:append([a, dog], [], [a, dog]) ? creep
   Call: (9) determiner([a, dog]) ? creep
   Fail: (9) determiner([a, dog]) ? creep
   Redo: (9) lists:append([a, dog|_2230], _2250, [a, dog]) ? cre
   Fail: (9) lists:append(_2236, _2238, [a, dog]) ? creep
   Fail: (8) noun_phrase([a, dog]) ? creep
false.
```

A more efficient way to code in Prolog

Instead of using one list, we use TWO lists

```
sentence(Sin,Sout) :-
    noun_phrase(Sin,NPout), verbal_phrase(NPout,Sout).
noun_phrase(NPin,NPout):-
    determiner(NPin,Dout), noun(Dout,NPout).
verbal_phrase(VPin,VPout):-
    verb(VPin,Vout), noun_phrase(Vout,VPout).
verbal_phrase(VPin,VPout):-
    verb(VPin,VPout).
determiner([the|Tail],Tail).
determiner([a|Tail],Tail).
noun([cat|Tail],Tail).
noun([dog|Tail],Tail).
verb([watches|Tail],Tail).
```

The pairs of lists are called difference lists

A more efficient way to code in Prolog

Which goal must we solve now?

To parse a sentence:

```
?- sentence([a,cat,watches,the,dog],[]).

?- sentence([a,cat,watches,dog],[]).
```

To generate sentences:

```
?- sentence(X,[]).
```

A more efficient way to code in Prolog

Let us trace a goal

```
[trace]  ?- noun_phrase([a,dog],[]).
   Call: (8) noun_phrase([a, dog], []) ? creep
   Call: (9) determiner([a, dog], _5038) ? creep
   Exit: (9) determiner([a, dog], [dog]) ? creep
   Call: (9) noun([dog], []) ? creep
   Exit: (9) noun([dog], []) ? creep
   Exit: (8) noun_phrase([a, dog], []) ? creep
true.
```

A more efficient way to code in Prolog

There exists the built-in predicate `phrase/2` that allows the user to transparently use the grammar without knowing how the Prolog interpreter translates a DCG to a Prolog program

To parse a sentence:

```
?- phrase(sentence,[a,cat,watches,the,dog]).

?- phrase(sentence,[a,cat,watches,dog]).
```

To generate sentences:

```
?- phrase(sentence,X).
```

A more efficient way to code in Prolog

Let us trace (1/2)

```
trace]  ?- phrase(sentence,[the,cat, watches,the,dog]).
   Call: (8) sentence([the, cat, watches, the, dog], []) ? creep
   Call: (9) noun_phrase([the, cat, watches, the, dog], _5336) ?
   Call: (10) determiner([the, cat, watches, the, dog], _5336) ?
   Exit: (10) determiner([the, cat, watches, the, dog], [cat, wa
   Call: (10) noun([cat, watches, the, dog], _5336) ? creep
   Exit: (10) noun([cat, watches, the, dog], [watches, the, dog]
   Exit: (9) noun_phrase([the, cat, watches, the, dog], [watches
   Call: (9) verbal_phrase([watches, the, dog], []) ? creep
   Call: (10) verb([watches, the, dog], _5336) ? creep
```

A more efficient way to code in Prolog

Let us trace (2/2)

```
   Exit: (10) verb([watches, the, dog], [the, dog]) ? creep
   Call: (10) noun_phrase([the, dog], []) ? creep
   Call: (11) determiner([the, dog], _5336) ? creep
   Exit: (11) determiner([the, dog], [dog]) ? creep
   Call: (11) noun([dog], []) ? creep
   Exit: (11) noun([dog], []) ? creep
   Exit: (10) noun_phrase([the, dog], []) ? creep
   Exit: (9) verbal_phrase([watches, the, dog], []) ? creep
   Exit: (8) sentence([the, cat, watches, the, dog], []) ? creep
true .
```

DCG

DCG introduces "syntactic sugar"

```
sentence -->
   noun_phrase, verbal_phrase.
noun_phrase -->
   determiner, noun.
verbal_phrase -->
   verb, noun_phrase.
verbal_phrase -->
   verb.
determiner --> [the].
determiner --> [a].
noun --> [cat].
noun --> [dog].
verb --> [watches].
```

DCG

Let us have a look at how Prolog translates it (listing)

```
sentence(A, C) :-
  noun_phrase(A, B),
  verbal_phrase(B, C).
noun_phrase(A, C) :-
  determiner(A, B),
  noun(B, C).
verbal_phrase(A, C) :-
  verb(A, B),
  noun_phrase(B, C).
verbal_phrase(A, B) :-
  verb(A, B).
determiner([the|A], A).
determiner([a|A], A).
noun([cat|A], A).
noun([dog|A], A).
verb([watches|A], A).
```

Actions in DCG

We can insert actions in a DCG.
For example, consider the context-free grammar defined by the following rules:

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

The DCG associated with this context-free grammar is:

```
s --> [a], [b].
s --> [a], s, [b].
```

Actions in DCG

We can use this DCG to solve various goals

```
?- phrase(s,[a, a, a, b, b, b]).
true

?- phrase(s,[a, a, b, b, b]).
false

?- phrase(s,L).
L = [a, b]
L = [a, a, b, b]
L = [a, a, a, b, b, b]
...
```

Actions in DCG

Now, suppose that while verifying if a sentence (a list of words) is syntactically correct, we want to count the number of "a" in this sentence.

We want to be able to run:

```
?- phrase(s(N),[a, a, a, b, b, b]).
N = 3
```

Actions in DCG

Of course we also want to get those results:

```
?- phrase(s(N),[a, a, b, b, b]).
false

?- phrase(s(N),L).
N = 1
L = [a, b]

N = 2
L = [a, a, b, b]

N = 3
L = [a, a, a, b, b, b]

...
```

Actions in DCG

To do so, we have to write the following DCG:

```
s(1) --> [a], [b].
s(M) --> [a], s(N), [b], {M is N + 1}.
```

Here we can see :

- the nonterminal symbol s now has one parameter
- we added an action: {M is N + 1}

Actions in DCG

Remember that while loading a DCG in the interpreter, it is converted, in a transparent way, to a pure Prolog program.

- each DCG rule is converted to a Prolog rule.
- each non terminal symbol that has n parameters is converted to a Prolog predicate of arity $n + 2$ parameters
- everything between "{" and "}" is left unchanged (modulo a variable renaming).

Actions in DCG

```
s(1) --> [a], [b].
s(M) --> [a], s(N), [b], {M is N + 1}.
```

is converted to:

```
?- listing(s).
s(1, [a|A], B) :-
  A=[b|B].
s(C, [a|A], E) :-
  s(D, A, B),
  B=[b|F],
  C is D+1,
  E=F.
```