Introduction to AI with Prolog

Definite Clause Grammars

Alexey Sery
a.seryj@g.nsu.ru

Novosibirsk State University
*THE REAL SCIENCE

НГУ ФИТ

Department of Information Technologies
Novosibirsk State University

October 20, 2020

Goals of the lesson

- To introduce context free grammars (CFGs) and some related concepts.
- To introduce definite clause grammars (DCGs), an inbuilt Prolog mechanism for working with context free grammars.

Context free grammars

Prolog's inventor, Alain Colmerauer, was a computational linguist, and computational linguistics remains a classic application for the language. SWI Prolog offers a number of tools which make life easier for computational linguists, and today we are going to start learning about one of the most useful of these: Definite Clauses Grammars, or DCGs as they are usually called.

In formal language theory, a **grammar** describes how to form strings from a language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context—only their form.

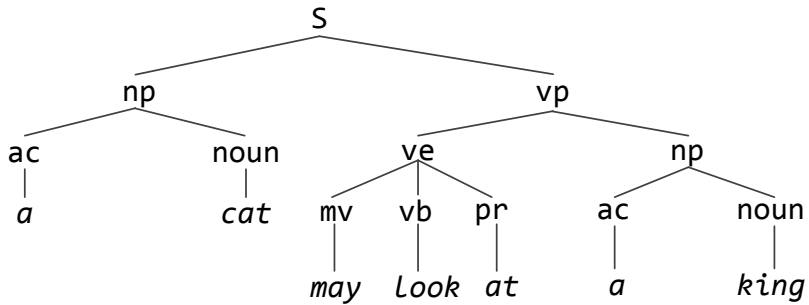
A **formal grammar** is defined as a set of production rules for strings in a formal language.

Context free grammars

S → nounPhrase verbPhrase
nounPhrase → article noun
verbPhrase → verbExpr nounPhrase
verbExpr → modalVerb verb prep
article → a
article → the
noun → cat
noun → king
modalVerb → may
verb → look
prep → at

- This grammar contains eleven rules.
- A context free rule consists of a single nonterminal symbol, followed by $\rightarrow$, followed by a finite sequence made up of terminal and/or non-terminal symbols. The rules tell us how different grammatical categories can be built up. Read $\rightarrow$ as **can consist of**, or **can be built out of**.
- `S`, `nounPhrase`, `verbPhrase`, `article`, `noun`, `verbExpr`, `modalVerb`, `verb`, `prep` are non-terminal symbols here.
- *a, the, cat, king, may look at* are terminal symbols. The entirety of terminal symbols is also called **an alphabet**.

Consider the string of words *A cat may look at a king*. Let us find out if it is grammatical according to our grammar, and if it is, what structure it has. The following tree answers both our questions:



- A string of non-terminal symbols (words) is **grammatically correct** according to our grammar if we can build a parse tree for it. Having a grammar like one above we can implement **a recognizer** — a program that correctly classifies strings as being grammatical or not.
- In addition to knowing whether a string is grammatical or not, we are interested in why it is grammatical. More precisely, we often want to know what its structure is, and this is exactly the information a parse tree gives us. This kind of information would be important if we were using this sentence in some application and needed to say what it actually meant. A program that correctly answers if a string is grammatical and also builds a parse tree is called **a parser**.

Context free grammars

- A context free language is a language that can be generated by a context free grammar.
- It is proved that some natural languages are context free (for example English and French).
- Many programming languages are context free languages.
- For this reason context free grammars are often used in compiler development.

Let us put it in practice.

```
sentense(S) :- nounPhrase(NP), verbPhrase(VP), append(NP,VP,S).
nounPhrase(NP) :- article(A), noun(N), append(A,N,NP).
verbPhrase(VP) :- verbExpr(VE), nounPhrase(NP), append(VE,NP,VP).
verbExpr(VE) :- modalVerb(MV),verb(V),prep(P),append([MV,V,P],VE).
article([A]) :- lexicon(''article'', A).
noun([N]) :- lexicon(''noun'',N).
modalVerb([MV]) :- lexicon(''modal verb'',MV).
verb([V]) :- lexicon(''verb'',V).
prep([P]) :- lexicon(''prep'',P).
```

DCG in Prolog

Predicate `lexicon/2` is used to define an alphabet (a set of terminal symbols).

```
lexicon(''article'',''a'').
lexicon(''article'',''the'').
lexicon(''noun'',''cat'').
lexicon(''noun'',''king'').
lexicon(''verb'',''look'').
lexicon(''modal verb'',''may'').
lexicon(''prep'',''at'').
```

Then add two predicates that are handy when you start a program. Predicate `recognize` asks user to input string and starts recognizer, `generate` starts language generator that returns all phrases that are grammatically correct according to grammar.

```
recognize :- write('Enter the phrase:  '),
             current_input(In),
             read_string(In, '\n', '\r\t', _, Phrase),
             split_string(Phrase, " ", "", ListPhrase),
             sentense(ListPhrase),!.

generate :- sentense(Phrase),
             atomics_to_string(Phrase,' ',String),
             write(String).
```

This program works, but it is not effective.

- The program tries to "guess" parts of a phrase and then afterwards checks whether these can be combined to form the sentence.
- Posing a query `sentense(['a','cat','may','look','at','a','king'])` will cause the program to check all possible sentences until one of them match the required one.
- The problem obviously is, that the goals are called with uninstantiated variables as arguments.

We could try and solve the third problem by changing the rules in such a way that append becomes the first goal, but still our program would remain highly ineffective.

More to it, if we place append to the front generation rules will break.

```
sentense(S) :- append(NP,VP,S), nounPhrase(NP), verbPhrase(VP).
nounPhrase(NP) :- append(A,N,NP), article(A), noun(N).
verbPhrase(VP) :- append(VE,NP,VP), verbExpr(VE), nounPhrase(NP).
verbExpr(VE) :- append([MV,V,P],VE),
                modalVerb(MV),
                verb(V),
                prep(P).
```

Recognition using difference lists

Consider:

```
Ls = [x, y, z| Rs].
```

% Ls is a partial list, because Rs is uninstantiated

Recognition using difference lists

Consider:

```
Ls = [x, y, z| Rs].
```

Symbolically: $Ls = [x, y, z] + Rs$

% `Ls` is a partial list, because `Rs` is uninstantiated
$\Leftrightarrow$ `append([x, y, z], Rs, Ls)`

Recognition using difference lists

Consider:

```
Ls = [x, y, z| Rs].
```

Symbolically: $Ls = [x, y, z] + Rs$

Symbolically: $Ls - Rs = [x, y, z]$

% `Ls` is a partial list, because `Rs` is uninstantiated
$\Leftrightarrow$ `append([x, y, z], Rs, Ls)`
`[x, y, z]` is the **list difference** between `Ls` and `Rs`

Recognition using difference lists

Consider:

```
Ls = [x, y, z| Rs].
```

Symbolically: $Ls = [x, y, z] + Rs$

Symbolically: $Ls - Rs = [x, y, z]$

```
Rs = [].
Rs = [a, b]
```

% `Ls` is a partial list, because `Rs` is uninstantiated

$\Leftrightarrow$ `append([x, y, z], Rs, Ls)`

`[x, y, z]` is the **list difference** between `Ls` and `Rs`

```
Ls = [x, y, z]
Ls = [x, y, z, a, b]
```

Recognition using difference lists

Consider:

```
Ls = [x, y, z| Rs].
```

Symbolically: $Ls = [x, y, z] + Rs$

Symbolically: $Ls - Rs = [x, y, z]$

```
Rs = [].
Rs = [a, b]
Ls0 = [x, y| Ls1], Ls1 = [z, k|
Ls2], Ls2 = [...| Ls3]
```

% `Ls` is a partial list, because `Rs` is uninstantiated

$\Leftrightarrow$ `append([x, y, z], Rs, Ls)`

`[x, y, z]` is the **list difference** between `Ls` and `Rs`

```
Ls = [x, y, z]
Ls = [x, y, z, a, b]
```

An efficient concatenation

Recognition using difference lists

Consider:

```
Ls = [x, y, z| Rs].
```

Symbolically: $Ls = [x, y, z] + Rs$

Symbolically: $Ls - Rs = [x, y, z]$

```
Rs = [].
Rs = [a, b]
Ls0 = [x, y| Ls1], Ls1 = [z, k|
Ls2], Ls2 = [...| Ls3]
```

$D_0 = Ls0 - Ls1, D_1 = Ls1 - Ls2, D_2 = Ls2 = Ls3$

% Ls is a partial list, because Rs is uninstantiated

$\Leftrightarrow$ `append([x, y, z], Rs, Ls)`

`[x, y, z]` is the **list difference** between `Ls` and `Rs`

```
Ls = [x, y, z]
Ls = [x, y, z, a, b]
```

An efficient concatenation

$$D_0 + D_1 + D_2 = Ls0 - Ls3$$

Recognition using difference lists

The pair of lists S and D represents a sentence if (a) We can consume S and leave behind a VP, and the pair S and VP represents a noun phrase, and (b) We can then go on to consume VP leaving D behind, and the pair VP, D represents a verb phrase. That is: the sentence we are interested in is the difference between the contents of the two lists.

```
sentense(S,D) :- nounPhrase(S,VP), verbPhrase(VP,D).
nounPhrase(NP,D) :- article(NP,N), noun(N,D).
verbPhrase(VP,D) :- verbExpr(VP,VE), nounPhrase(VE,D).
verbExpr(VE,D) :- modalVerb(VE,MV), verb(MV,V), prep(V,D).
article([A|D],D) :- lexicon(''article'',A).
noun([N|D],D) :- lexicon(''noun'',N).
modalVerb([MV|D],D) :- lexicon(''modal verb'',MV).
verb([V|D],D) :- lexicon(''verb'',V).
prep([P|D],D) :- lexicon(''prep'',P).
```

Recognition using difference lists

DCGs are really "syntactic sugar" for grammars written in terms of difference lists.

```
sentense --> nounPhrase, verbPhrase.
nounPhrase --> article, noun.
verbPhrase --> verbExpr, nounPhrase.
verbExpr --> modalVerb, verb, prep.
article --> [''a''].
article --> [''the''].
noun --> [''cat''].
noun --> [''king''].
modalVerb --> [''may''].
verb --> [''look''].
prep --> [''at''].
```

Recursive rules

Suppose we want to add a conjunctive rule to our little grammar:

sentense → sentense conjunction sentense
conjunction → and
conjunction → or
conjunction → but

Recursive rules

A piece of cake, isn't it?

```
sentense --> sentense, conjunction, sentense.
conjunction --> [''and''].
conjunction --> [''or''].
conjunction --> [''but''].
```

Nice try, but no...

Recursive rules

If we place the recursive rule before the non-recursive rule

```
sentense --> nounPhrase, verbPhrase
```

then the knowledge base will contain the following two Prolog rules, in this order:

```
sentence(A, B) :- sentence(A, C), conjunction(C, D), sentence(D, B).
sentence(A, B) :- nounPhrase(A, C), verbPhrase(C, B).
```

When Prolog tries to use the first rule, it immediately encounters the recursive goal, which it then tries to satisfy using the first rule, whereupon it immediately encounters the same goal and so on infinitely... Prolog goes into infinite loop and does no useful work.

Recursive rules

So, let us add the recursive rule at the end of the knowledge base making Prolog always encounter the non-recursive rule first.

What happens now when we try to recognize grammatically correct sentence? Prolog seems to be able to handle it and gives the right answer. But when we pose an ungrammatical sentence, Prolog again gets into an infinite loop.

Since it is unable to recognize our sentence as consisting of a noun phrase and a verb phrase, Prolog tries to analyze it with a recursive rule again and again, and ends up in the same loop as before.

```
?- sentence(["a", "cat", "may", "look", "at", "the", "king"], [])  true.
?- sentence(["a", "cat"], []).                                     ERROR
```

Recursive rules

Recall, that in the case of ordinary recursive rule (not DCG) the trick was to change goals of the recursive rule so that the recursive goal is not the first one in the body of the rule.

In the case of DCG this is not a solution, since the order of the goals determines the order of the words in the sentence.

```
sentence(A, B) :- sentence(A, C), conjunction(C, D), sentence(D, B). ⇏
sentence(A, B) :- conjunction(C, D), sentence(A, C), sentence(D, B).
```

Recursive rules

What do we do then? The only possible solution we have is to introduce new non-terminal symbols.

```
plainSentense --> nounPhrase, verbPhrase.
sentense --> plainSentense.
sentense --> plainSentense, conjunction, sentense.
```

A simple example of a formal language is $a^n b^n$. There are only two "words" in this language: the symbol a and the symbol b. The language $a^n b^n$ consist of all strings made up from these two symbols that have the following form: the string must consist of an unbroken block of as of length n, followed by an unbroken block of bs of length n, and nothing else. Note that the empty word $\mathcal{E}$ also belongs to $a^n b^n$.

A simple example of a formal language is $a^n b^n$. There are only two "words" in this language: the symbol a and the symbol b. The language $a^n b^n$ consist of all strings made up from these two symbols that have the following form: the string must consist of an unbroken block of as of length n, followed by an unbroken block of bs of length n, and nothing else. Note that the empty word $\mathcal{E}$ also belongs to $a^n b^n$.

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