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

November 17, 2020

Goals of the lesson

- To examine extra arguments and extra tests offered by DCG notation.
- To discuss the limitations of DCGs.

Extra arguments

Recall the grammar we built on the previous class:

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

Extra arguments

As we know this grammar produces sentences like:

- A cat may look at a king
- A cat may look at the king
- A king may look at a cat
- A king may look at the cat
- The cat may look at a king
- The king may look at a cat

And so on, around 20 sentences in total.

Extra arguments

Suppose we wanted to add pronouns, and deal with sentences like:

- He may look at her
- A cat may look at him
- A king may look at her
- She may look at a cat

There are subjective and objective pronouns.

Extra arguments

Hmm, nice, let us add rules for pronouns:

```
pro -->["he"].
pro -->["him"].
pro -->["she"].
pro -->["her"].
nounPhrase -->pro.
```

Extra arguments

Up to a point this grammar works. It recognizes and generates all sentences we wanted it to, but it will also accept a lot of sentences that are clearly wrong.

- Him may look at a cat
- He may look at she
- The king may look at she
- The cat may look at she

Extra arguments

Definite Clause Grammar is just a language model. It possesses no knowledge about English phrases.

That is, accepting a sentence *"A cat may look at she"* it doesn't know that "she" and "he" are subject pronouns and cannot be used in object position.

Moreover, accepting *"Him may look at a cat"* it doesn't know either that "her" and "him" are object pronouns and cannot be used in subject position.

Extra arguments

PERSONS	SINGULAR		PLURAL	
	Subjective Case	Objective Case	Subjective Case	Objective Case
1^{st} person	I	me	we	us
2^{nd} person	you	you	you	you
3^{rd} person	he,she,it	him,her,it	they	them

Extra arguments

Now it is obvious that we have to explicitly state which pronoun can and which cannot occur in a subject position.

```
sentense --> subjNP, verbPhrase.
subjNP --> article, noun.
subjNP --> spr.
objNP --> article, noun.
objNP --> opr.
verbPhrase --> verbExpr, objNP.
spr -->  ["he"].
spr --> ["she"].
opr --> ["him"].
opr --> ["her"].
```

It works now. This solution, however, cannot be considered a good one.

A small addition to the lexicon has led to quite a big change in the grammar. In particular, we've doubled the number of noun phrase rules.

If we wanted to make more changes, let's say it to further extend the lexicon, things would get even worse.

Extra arguments

What we truly need is some elegant programming technique, and here is where the extra arguments come into play.

Consider the following program:

```
sentense --> nounPhrase(subj),verbPhrase.
nounPhrase(_) --> article, noun.
nounPhrase(P) --> pro(P).
verbPhrase --> verbExpr, nounPhrase(obj).
verbExpr --> modalVerb, verb, prep.
modalVerb --> ["may"].
verb --> ["look"].
prep --> ["at"].
```

```
article --> ["a"].
article --> ["the"].
noun --> ["cat"].
noun --> ["king"].
pro(subj) --> ["he"].
pro(subj) --> ["she"].
pro(obj) --> ["him"].
pro(obj) -> ["her"].
```

Extra arguments

We already know (or at least I hope we do) that any DCG rule is just a syntactic sugar for an ordinary Prolog rule. So, what `sentence --> nounPhrase(subj)` translates into?

```
sentense --> nounPhrase,verbPhrase.
sentense(A,B) :- nounPhrase(A,C),verbPhrase(C,B).
```

Extra arguments

We already know (or at least I hope we do) that any DCG rule is just a syntactic sugar for an ordinary Prolog rule. So, what `sentence --> nounPhrase(subj)` translates into?

```
sentense --> nounPhrase,verbPhrase.
sentense(A,B) :- nounPhrase(A,C),verbPhrase(C,B).
```

```
sentense --> nounPhrase(subj),verbPhrase.
sentense(A,B) :- nounPhrase(subj,A,C),verbPhrase(C,B).
```

Extra arguments

One final remark: don't be misled by this simplicity of our example. Extra arguments can be used to cope with some complex syntactic problems.

DCGs are no longer the state-of-art grammar development tools they once were, but they're not toys either.

Once you know about writing DCGs with extra arguments, you can write some fairly sophisticated grammars.

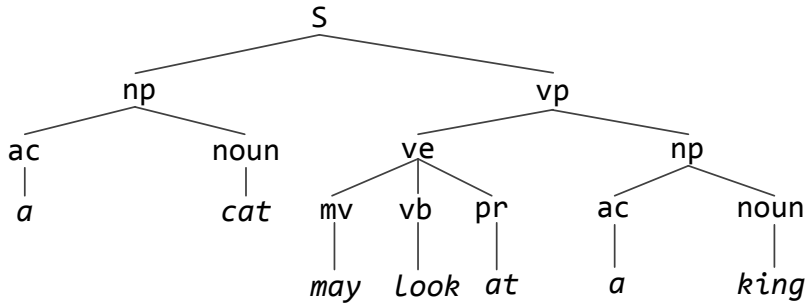
Making parsers

So far, our programs have been able to answer *"yes"* or *"no"* whether the input sentence correct according to the grammar, and produce grammatical output.

That is, they all were *recognizers*.

But what if we would like them to not only tell us which sentences are grammatical, but also to analyze their structures. For example we would like to see the parse trees.

Considering the tree:



We could make a following DCG:

```
sentense(s(NP,VP)) --> nounPhrase(NP), verbPhrase(VP).
nounPhrase(np(A,N)) --> article(A), noun(N).
verbPhrase(vp(VE,NP)) --> verbExpr(VE), nounPhrase(NP).
verbExpr(ve(MV,V,P)) --> modalVerb(MV), verb(V), prep(P).
article(art("a")) --> ["a"].
article(art("the")) --> ["the"].
noun(n("cat")) --> ["cat"].
noun(n("king")) --> ["king"].
modalVerb(mv("may")) --> ["may"].
verb(v("look")) --> ["look"].
prep(p("at")) --> ["at"].
```

Corresponding to our proverb we could have the following result:

```
?- sentence(T, [''a'', ''cat'', ''may'', ''look'', ''at'', ''a'', ''king''], []).

T = s(np(art(''a''), n(''cat'')), vp(ve(mv(''may''), v(''look''), p(''at'')),
np(art(''a''), n(''king'')))).
```

It may not look nice, but it contains all the information from the picture. Depending on the task, we can design the output to our liking.

Extra goals

When haven't used DCG notations we introduced a special predicate `lexicon/2` to define our alphabet.

```
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).
```

Extra goals

When haven't used DCG notations we introduced a special predicate `lexicon/2` to define our alphabet.

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

Extra goals

When haven't used DCG notations we introduced a special predicate `lexicon/2` to define our alphabet.

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

Why haven't we done the same thing in DCG notation then? **Is it possible?**

Extra goals

Why, of cause it is!

```
sentense --> nounPhrase, verbPhrase.
nounPhrase --> article, noun.
verbPhrase --> verbExpr, nounPhrase.
verbExpr --> modalVerb, verb, prep.

article --> {lexicon("article",A)},[A].
noun --> {lexicon("noun",Noun)},[Noun].
modalVerb --> {lexicon("modal verb",Mv)},[Mv].
verb --> {lexicon("verb",Verb)},[Verb].
prep --> {lexicon("prep",Prep)},[Prep].
```

The extra goals can be written anywhere on the right side of a DCG rule, but **must stand between curly brackets**. When Prolog encounters such curly brackets while translating a DCG into its internal representation, it just takes the extra goals specified between the curly brackets over into the translation.

The extra arguments give us the tools for coping with any computable language whatsoever, not just context free languages.

For example, the formal language $a^n b^n c^n - \{\varepsilon\}$ is not a context free language. It could be proved by contradiction that you will not succeed in writing a context free grammar that generates precisely these strings.

Extra goals

What we need to make a DCG for $a^n b^n c^n - \{\varepsilon\}$ is a symbol counter. Any correct string has to have a block of as followed by a block of bs followed by a block of cs, and all blocks are to have the same length.

```
as(0) -->  [].
as(Len) -->  [a],as(Prev), {Len is Prev + 1}.
bs(0) -->  [].
bs(Len) -->  [b],bs(Prev), {Len is Prev + 1}.
cs(0) -->  [].
cs(Len) -->  [c],cs(Prev), {Len is Prev + 1}.
```

The second rule for the non-terminal as would be translated as follows:

```
as(Len,A,B) :- 'C'(A, a, C),
               as(Prev, C, B),
               Len is Prev + 1.
```

Conclusions

For the most part, DCGs are presented as a simple tool for developing context free grammars. But as we saw it was possible to extend beyond that. DCGs are programming language in their own right. They are Turing complete.

But they have rather significant drawbacks. They are tendent to loop when goal ordering is wrong. If we need to use lots of extra arguments, they become a rather clumsy mechanism. Many of this problems, however, DCG notation inherits from the Prolog itself. It is well-known that all top-down parsers loop on left-recursive grammars.

DCGs as we saw them, are a nice notation for context free grammars enhanced with some features that comes with a free parser/recognizer, are probably best viewed as a convenient tool for testing new grammatical ideas, or for implementing reasonably complex grammars for particular applications.

With a conventional programming language (such as C++ or Java) it simply wouldn't be possible to reach this stage so soon. Things would be easier in functional languages (such as LISP, SML, or Haskell), but even so, it is doubtful whether beginners could do so much so early.