Definite Clause Grammars

Paul Bailey
paulba@dai.ed.ac.uk
Room F12, 80 South Bridge
Division of Informatics
University of Edinburgh

Contents

- Definite Clause Grammars
- Grammar rules
- Terminals and non-terminals
- Grammar rules in Prolog
- How Prolog uses grammar rules
- A very simple grammar
- Adding arguments
- Adding Prolog goals

Definite Clause Grammars

- Prolog provides some built-in facilities for defining *grammars*.

- A *grammar* is a precise definition of which sequences of words or symbols belong to some *language*.

- In Prolog, these grammars are called *Definite Clause Grammars* (DCGs).

- Grammars are particularly useful for natural language processing, which is the computational processing of human languages, like English.

- But they can be used to process any precisely defined ’language’, such as the commands allowed in some human-computer interface.

Grammar rules

- In general, a grammar is defined as a collection of *grammar rules*. These are sometimes called *rewrite rules*, since they show how we can rewrite one thing as something else.

- In linguistics, a typical grammar rule for English might look like this:

```
sentence → noun_phrase, verb_phrase
```

- This would show that, in English, a *sentence* could be constructed as a *noun phrase*, followed by a *verb phrase*.

- Other rules would then define how a noun phrase, and a verb phrase, might be constructed. For example:

```
noun_phrase → noun
noun_phrase → determiner, noun
verb_phrase → intransitive_verb
verb_phrase → transitive_verb, noun_phrase
```

Terminals and non-terminals

- In these rules, symbols like *sentence*, *noun*, *verb*, etc., are used to show the structure of the language, but they don't go as far down as individual 'words' in the language.

- Such symbols are called *non-terminal symbols*, because we can't stop there.

- In defining grammar rules for *noun*, though, we might be able to say:

```
noun → 'ball'
noun → 'dog'
noun → 'stick'
noun → 'Edinburgh'
```

- Here, 'ball', 'dog', 'stick' and 'Edinburgh' are words in the language itself.

- These are called the *terminal symbols*, because we can't go any further. They can't be expanded any more.

Grammar rules in Prolog

- Grammar rules look very similar to this in Prolog.
- In place of the → arrow, we have a special operator: `-->`.
- So, we might write the same rules as:

```
sentence --> noun_phrase, verb_phrase.
noun_phrase --> noun.
noun_phrase --> determiner, noun.
verb_phrase --> intransitive_verb.
verb_phrase --> transitive_verb, noun_phrase.
```

- Here, each non-terminal symbol is like a predicate with no arguments.
- Terminal symbols are represented as lists:

```
noun --> [ball].
noun --> [dog].
noun --> [stick].
noun --> ['Edinburgh'].
```

How Prolog uses grammar rules

- DCG rules look a lot like conventional Prolog clauses, with a left-hand side, and a right-hand side.

- In fact, Prolog converts DCG rules into an internal representation which makes them conventional Prolog clauses.

- Non-terminals are given two extra arguments, so:

  ```
  sentence --> noun_phrase, verb_phrase.
  ```

  becomes:

  ```
      sentence(In, Out) :-
           noun_phrase(In, Temp),
           verb_phrase(Temp, Out).
  ```

- This means: some sequence of symbols `In`, can be recognised as a sentence, leaving `Out` as a remainder, if a noun phrase can be found at the start of `In`, leaving `Temp` as a remainder, then a verb phrase can be found at the start of `Temp`, leaving `Out` as a remainder.

How Prolog uses grammar rules (2)

- Terminal symbols are represented using the special predicate `’C’`, which has three arguments. So:

  ```
  noun --> [ball].
  ```

  becomes:

  ```
  noun(In, Out) :-
          ’C’(In, ball, Out).
  ```

- This means: some sequence of symbols `In` can be recognised as a noun, leaving `Out` as a remainder, if the atom `ball` can be found at the start of that sequence, leaving `Out` as a remainder.

- The built-in predicate `’C’` is very simply defined:

  ```
  ’C’( [Term|List], Term, List ).
  ```

  where it succeeds if its second argument is the head of its first argument, and the third argument is the remainder.

A very simple grammar

- Here’s a very simple little grammar, which defines a very simple language:

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

noun --> [paul].
noun --> [david].
noun --> [annie].

verb --> [likes].
verb --> [hates].
verb --> [defenestrates].
```

- We can now use the grammar to test whether some sequence of symbols *belongs to* the language:

```
| ?- sentence([paul, likes, annie], []).
yes

| ?- sentence([paul, likes, teaching, iaip], []).
no
```

A very simple grammar (2)

- We might even use the grammar to generate all of the possible sentences in the language:

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

  X = [paul,likes,paul] ?  ;
  X = [paul,likes,david] ?  ;
  X = [paul,likes,annie] ?  ;
  X = [paul,hates,paul] ?  ;
  X = [paul,hates,david] ?  ;
  ```

  and so on.

- What we've implemented here is a *recogniser*. It will tell us whether some sequence of symbols is in a language or not. This has limited usefulness.

- It would be much more useful if we could *do* stuff with a sequence of symbols, such as converting it into some internal form for processing, or converting it into another form, say for language translation.

- We can do this very powerfully with DCGs, by building a *parser*, rather than a recogniser.

Adding arguments

- We can add our own arguments to the non-terminals in DCG rules, for whatever reasons we choose.

- As an example, in English, the *number* (singular or plural) of the subject of a sentence and the *number* of the main verb must agree. We can add this constraint to a grammar very easily:

```
sentence --> noun(Num), verb_phrase(Num).
verb_phrase(Num) --> verb(Num), noun(_).

noun(singular) --> [paul].
noun(plural) --> [students].

verb(singular) --> [likes].
verb(plural) --> [like].
```

- So now:

```
| ?- sentence([paul, likes, students], []).
yes
| ?- sentence([paul, like, students], []).
no
| ?- sentence([students, like, paul], []).
yes
| ?- sentence([students, likes, paul], []).
no
```

Adding Prolog goals

- If we need to, we can add arbitrary Prolog goals to any DCG rule.

- They need to be put inside {} brackets, so that Prolog knows they're to be processed as Prolog, and not as part of the DCG itself.

- Let's say that within some grammar, we wanted to be able to say that some symbol had to be an integer between 1 and 100 inclusive. We *could* write a separate rule for each number:

```
num1to100 --> [1].
num1to100 --> [2].
num1to100 --> [3].
num1to100 --> [4].
...
num1to100 --> [100].
```

- But using a Prolog goal, there's a much easier way:

```
num1to100 --> [X], {integer(X), X >= 1, X =< 100}.
```