Logic programming as a framework for Knowledge Representation and Artificial Intelligence

Addendum: Definite Clause Grammars

Henning Christiansen
Roskilde University

Contents

1 Definite Clause Grammars in Prolog **2**
1.1 A first DCG 2
1.2 Adding attributes 4
1.3 Adding extra conditions 5

1 Definite Clause Grammars in Prolog

Most available Prolog systems include a grammar notation for Definite Clause Grammars, DCGs. You can write your own grammars for, say, natural language (such as Danish, Latin, etc.), programming languages, or (prototype versions) of user interface languages. You can write DCG rules in your Prolog source programs without referring to special libraries, and DCGs are implemented as a kind of syntactic sugar in the sense that they are compiled into Prolog rules when the program file is read in.

Some Prolog versions, such as SICStus [16], include facilities that make it straightforward to implement new notation that can be used in source texts and is translated into Prolog in a way similar to DCG. We will not go into these details here; interested readers may look up the predicate `term_expansion` in the manual [16].

DCGs process Prolog lists which means that you need to do some extra work in order to read in a text file and have it analyzed using your grammar. This phase, often called lexical analysis or tokenization, consists here of reading the file character by character and converting it into a list of Prolog constants. So if, for example, you have a text file saying "`logic programming is fun`", you should write the code that converts to the list `[logic,programming,is,fun]`; this is a straightforward exercise that we do not consider here.

For small experiments, we can ignore the reading-from-file problems by stating the strings to be analyzed as Prolog lists. Assuming you have written the right grammar for a language that includes this sentence, you may have it analyzed by calling the built-in predicate `phrase` as follows.

```
?- phrase(sentence,[logic,programming,is,fun]).
yes
```

In the following we introduce DCGs by a series of more and more advanced examples.

1.1 A first DCG

Here is shown a little context-free grammar that recognizes simple sentences such as "the man eats the apple". Nonterminals in this grammar are `sentence`, `noun_phrase`, `verb_phrase`, `determiner`, `noun`, and `verb`; terminal symbols are those that appear inside square brackets in the grammar rules.

```
sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun.

verb_phrase --> verb.

verb_phrase --> verb, noun_phrase.

determiner --> [the].

noun --> [man].
noun --> [men].
noun --> [woman].
noun --> [women].
noun --> [apple].
noun --> [apples].
```

```
verb --> [eats].
verb --> [eat].
verb --> [loves].
verb --> [love].
verb --> [sings].
verb --> [sing].
```

Notice that a grammar rule is distinguished from a usual Prolog rule by using "`-->`" instead of "`:-`". This signifies to the Prolog systems, that is should run the DCG compiler for each rule to produce a Prolog rule which, then, is treated in the usual way.

Nonterminals are used in the grammar in a way similar to predicates in usual Prolog rules, and we will see later that nonterminals can have additional arguments. The compilation to Prologs adds to arguments to each nonterminal. As an example, the rule for noun phrases is compiled into the following Prolog rule.

```
noun_phrase(S0,S2):- determiner(S0,S1), noun(S1,S2).
```

In principle, we could have used just one argument to hold the list to be analyzed, so that, e.g., an instance of a `noun_phrase` held a list such as `[the,man]`, and then split the string using `append`. However, it can be shown, that such an application would become very inefficient due to immense backtracking as the `append` predicate will generate all possible splittings of a list until the right one is found.

The technique used for DCG is know as *difference lists*, which makes it possible to take one symbol off the list one by one. If, for example, the list `[the,man,eats,the,apple]` is analyzed with the grammar, the `noun_phrase` predicate would be called (for the first noun phrase in the sentence) with the following arguments

```
noun_phrase([the,man,eats,the,apple],_)
```

When this call has finished, the second argument has become instantiated to the list representing what remains when the noun phrase has "consumed" its symbols.

```
noun_phrase([the,man,eats,the,apple],[eats,the,apple])
```

This remaining list will the be given to the continuation which, in this case is the call for analyzing a verb phrase which goes on to a call to `verb`. The rule for the verb "eats" can be compiled into the following Prolog clause.[1]

```
verb([eats|S], S).
```

Similar to what we saw above, we may expect it to be called with the list `[eats,the,apple]` as first argument, and when it has finished, the second argument has been given the value `[the,apple]`, which is given to the continuation which is the analysis of the final noun phrase. The following examples shows some queries and answers.

[1]Some Prolog systems uses an auxiliary predicate called `c` (for "connects" defined by clause `c(X,[X|S],S)`. In that case, the `verb` rule is compiled into `verb(S0,S1):- c(eats,S0,S1).`

```
?- phrase( sentence, [the,man,eats,the,apples]).
yes
?- phrase( sentence, [the,duck,ducks,the,duck]).
no
```

However, this grammar accepts also sentences with number disagreement such as the following.

```
?- phrase( sentence, [the,women,eats,the,apple]).
yes
```

In addition there is no semantics involved, so the grammar accepts also nonsense sentences as this one.

```
?- phrase( sentence, [the,man,sings,the,apple]).
yes
```

The inclusion of semantics to handle the last sort of requires linguistically grounded models that we shall avoid for the moment. However, the number agreement can included in the grammar as shown in the following section.

1.2 Adding attributes

Arguments can be added to nonterminals exactly as arguments to predicates in a Prolog rule; in the case of grammars, these additional arguments are often called *attributes* or *features.* In the following grammar, we have added attributes corresponding to grammatical number to the relevant nonterminals; we expect them to take values `singular` or `plural`.

```
sentence --> noun_phrase(Number), verb_phrase(Number).

noun_phrase(Number) --> determiner(Number), noun(Number).

verb_phrase(Number) --> verb(Number).

verb_phrase(Number) --> verb(Number), noun_phrase(_).

determiner(_) --> [the].

noun(singular) --> [man].
noun(plural)   --> [men].
noun(singular) --> [woman].
noun(plural)   --> [women].
noun(singular) --> [apple].
noun(plural)   --> [apples].

verb(singular) --> [eats].
verb(plural)   --> [eat].
verb(singular) --> [loves].
verb(plural)   --> [love].
verb(singular) --> [sings].
verb(plural)   --> [sing].
```

The following queries and answers indicates that now the grammar suppresses sentences with number problems.

```
?- phrase( sentence, [the,man,eats,the,apple]).
yes
?- phrase( sentence, [the,man,eat,the,apple]).
no
?- phrase( sentence, [the,man,sings,the,apple]).
yes
```

This additional argument is included by the DCG compiler as an argument of the corresponding predicate symbol together with the two arguments that are used for the string.

```
noun_phrase(Number,S0,S2):- determiner(Number,S0,S1), noun(Number,S1,S2).
```

As for other Prolog programs, we may use the in several directions and a grammar can often be used for generation of language. The following query generates on backtracking (when user types a lot of semicolons), the 126 different sentences that are possible in our little language.

```
?- phrase( sentence, S).
S = [the, man, eats];
S = [the, man, loves]:
...
S = [the, man, loves, the, woman];
S = [the, man, loves, the, women];
S = [the, man, loves, the, apple];
S = [the, man, loves, the, apples];
S = [the, man, sings, the, man];
...
S = [the, apples, sing, the, women];
S = [the, apples, sing, the, apple];
S = [the, apples, sing, the, apples];
no
```

1.3 Adding extra conditions

Finally, we can add arbitrary Prolog code to the bodies of grammar rules which is useful when the attributes in the rule depend on each other in complicated ways. This may be relevant, for example, when semantic descriptions are added to a grammar such as the one shown above. Here we will take a simpler example.

The following little grammar describes the command language for a little robot that can walk along a straight line. The grammar uses the curly-bracket notation in order to assign a "meaning" to a command sequence, which is the final position of the robot (assuming that it starts at position `0`).

Curly brackets are useful for conditions that cannot be expressed in an easy way using unification of arguments only.

The grammar is available at file `trip`. Notice that there are two nonterminals called `trip`, but they differ in the number of arguments.

```
trip(To) --> trip(0,To).

trip(Here,Here) --> [].

trip(From,To) -->
   step(HowMuch),
   {NewFrom is From + HowMuch},
   trip(NewFrom,To).

step(1) --> [forward].
step(-1) --> [back].
step(0) --> [think].
```

The following shows a query and answer for this grammar.

```
?- phrase(trip(N), [forward,forward,think,back,think,forward,forward]).
N = 3
```

References

[1] CHR Grammars; official web pages. `http://www.ruc.dk/~henning/chrg`

[2] The programming language CHR, Constraint Handling Rules; official web pages. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`

[3] Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In *Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series*, pages 141–152. Physica-Verlag (Springer), 2000.

[4] Slim Abdennadher and Thom Frühwirth. *Essentials of Constraint Programming*. Springer, 2003.

[5] Ivan Bratko and Stephen Muggleton. Applications of inductive logic programming. *Commun. ACM*, 38(11):65–70, 1995.

[6] H. Christiansen and V. Dahl. Meaning in Context. In Anind Dey, Boicho Kokinov, David Leake, and Roy Turner, editors, *Proceedings of Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05)*, volume 3554 of *Lecture Notes in Artificial Intelligence*, pages 97–111, 2005.

[7] Henning Christiansen. Teaching computer languages and elementary theory for mixed audiences at university level. *Computer Science Education Journal*, 14, 2004. To appear.

[8] Henning Christiansen. CHR Grammars. *Int'l Journal on Theory and Practice of Logic Programming*, 5(4-5):467–501, 2005.

[9] Henning Christiansen and Veronica Dahl. Assumptions and abduction in Prolog. In Elvira Albert, Michael Hanus, Petra Hofstedt, and Peter Van Roy, editors, *3rd International Workshop on Multiparadigm Constraint Programming Languages, MultiCPL'04; At the 20th International Conference on Logic Programming, ICLP'04 Saint-Malo, France, 6-10 September, 2004*, pages 87–101, 2004.

[10] Henning Christiansen and Verónica Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2005.

[11] Peter A. Flach and Antonis C. Kakas, editors. *Abduction and Induction: Essays on their relation and integration*. Kluwer Academic Publishers, April 2000.

[12] Thom Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.

[13] A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming,* vol. 5, Gabbay, D.M, Hogger, C.J., Robinson, J.A., (eds.), Oxford University Press, pages 235–324, 1998.

[14] Michael Negnevitsky. *Artificial Intelligence, A Guide to Intelligent systems*. Addison-Wesley, 2nd edition, 2004.

[15] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf, editors. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science*. Springer, 1997.

[16] Swedish Institute of Computer Science. SICStus Prolog user's manual, Version 3.12. Most recent version available at `http://www.sics.se/isl`, 2006.