Logic Programming and Deductive Databases

Chapter 8: Definite Clause Grammars (DCGs)

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Summer 2021

http://www.informatik.uni-halle.de/~brass/lp21/

Objectives

After completing this chapter, you should be able to:

- use context-free grammars in Prolog.
- compare grammar support in Prolog with parser generators for compilers like yacc/bison.

Contents

Definite Clause Grammars (1)

- In Prolog, one can directly write down grammar rules (of context-free grammars).

  Actually, the grammar formalism is even more expressive, since one can include arbitrary Prolog code as an additional condition for the applicability of a grammar rule.

- A simple preprocessor translates the grammar rules into standard Prolog rules.

- Thus, Prolog has a tool like yacc/bison built-in.

  yacc/bison are standard tools used in compiler construction: Given a context-free grammar (with certain restrictions), they produce a C program that checks the syntactic correctness of the input. One can extend the grammar with program code for processing the input.

Definite Clause Grammars (2)

- However, the goal of the Prolog grammar formalism is not compiler construction, but natural language processing (e.g., for machine translation):
  - There one needs more complicated grammars.
  - E.g., non-deterministic grammars are possible in Prolog, but not in compiler-construction tools.

    In natural language (e.g., English), there are ambiguous words, phrases, and sentences. These can easily be processed with backtracking in Prolog. In programming languages (e.g., C, Prolog), the meaning of every construct must be completely clear.
  - However, the efficiency requirements are not so strong, since the inputs are usually not very long.

Definite Clause Grammars (3)

- Comparison with yacc/bison, continued:
  - In Prolog, arbitrary context-free grammars are possible, in yacc/bison only LALR(1) grammars.

    The condition in compiler construction tools ensures that efficient parsing is possible: The decisions for building the parse tree are done backtrack-free with only a single token lookahead.
  - In Prolog, grammar rules can be mixed with arbitrary program code. This can contain additional checks for selecting a grammar rule.

    In yacc/bison, one can include C code that is executed when a grammar rule is applied. In this way, one can, e.g. generate output. However, the C code cannot influence the parsing decisions, e.g. choose one of several possible grammar rules.

Definite Clause Grammars (4)

- The grammar on the next slide describes the commands of a simple text adventure game:
  - Such games are similar to books, in which the reader can give the main actor commands and influence in this way the storyline.

    Therefore, they are called “interactive fiction”. Titles like “Zork” by Infocom were very popular.
- In principle, the user can input any English sentence. In practice, most commands
  - move the player around in the adventure world
  - apply objects found in certain locations.

Definite Clause Grammars (5)

```
command --> verb, noun_phrase.
command --> [go], direction.
command --> direction.
command --> [quit].

direction --> [north].
direction --> [south].
direction --> [east].
direction --> [west].

verb --> [take].
verb --> [examine].

noun_phrase --> noun.
noun_phrase --> [the], noun.
noun --> [key].
noun --> [lamp].
```

Definite Clause Grammars (6)

- Nonterminal grammar symbols (syntactic categories) are written like standard Prolog predicates.
- Terminal symbols (expected input tokens) are written in `[...]`.
- The syntactic analysis is done with the predicate "`phrase`", e.g.
  - `phrase(command, [take, the, lamp]).` $\longrightarrow$ yes.
  - `phrase(command, [lamp, the, north]).` $\longrightarrow$ no.

  Of course, one needs more than "yes/no". This is done by attributes of the grammar symbols (predicate arguments). See below.

Implementation (1)

- In principle, every nonterminal symbol N is translated into a Prolog predicate that is true for all lists of input tokens that are derivable from N.
- A naive solution (not the Prolog solution) would generate rules like

```
command(X)        :- append(Y, Z, X),
                     verb(Y), noun_phrase(Z).
command([go|X])   :- direction(X).
command(X)        :- direction(X).
command([quit]).
```

Implementation (2)

- The above solution is too inefficient: Especially the arbitrary splitting of the input list with append causes a lot of unnecessary backtracking.
- The real implementation of grammar rules uses a data structure called "difference lists":
  - E.g. the list `[a, b, c]` is represented by a pair of lists `[a, b, c | X]` and `X`.
  - A special case is the pair `[a, b, c, d, e]`, `[d, e]`: This also represents the list `[a, b, c]`.

Implementation (3)

- Thus, every nonterminal symbol is translated into a predicate with two arguments:
  - Input list (total rest of input tokens before the nonterminal symbol is processed).
  - Output list (rest of input after the nonterminal).
- The difference between both lists are the input symbols derivable from the nonterminal symbol:

```
command(X, Z) :- verb(X, Y), noun_phrase(Y, Z).
command(X, Z) :- X = [go|Y], direction(Y, Z).
command(X, Z) :- direction(X, Z).
command(X, Z) :- X = [quit|Z].
```

Implementation (4)

- I.e. every predicate cuts off from the input list the prefix it can process, and hands the rest to the next predicate.
- The syntax analysis is then done by calling the predicate for the start symbol of the grammar with the complete input list and the empty list as the rest:

```
command([take, the lamp], []).
```

- Thus, the predicate `phrase` is defined as:

```
phrase(Start, Input) :-
        Goal =.. [Start, Input, []],
        call(Goal).
```

Attributes (1)

- Usually, it is not sufficient to know that the input is syntactically correct, but one needs to collect data from the input.
- Therefore, the nonterminal symbols can have arguments (which correspond to attributes in attribute grammars).
- The preprocessor for grammar rules simply extends the given literals by two further arguments for the input and output token lists.

  The given arguments are left untouched.

Attributes (2)

```
command(V,O) --> verb(V), noun_phrase(O).
command(go,D) --> [go], direction(D).
command(go,D) --> direction(D).
command(quit,nil) --> [quit].

direction(n) --> [north].
direction(s) --> [south].
direction(e) --> [east].
direction(w) --> [west].

verb(take) --> [take].
verb(examine) --> [examine].

noun_phrase(O) --> noun(O).
noun_phrase(O) --> [the], noun(O).
noun(key) --> [key].
noun(lamp) --> [lamp].
```

Further Possibilities (1)

- One can include arbitrary Prolog code in the syntax rules.
- It must be written in `{...}` in order to protect it from the rewriting done by the preprocessor.
- E.g. it might be easier to store a list of game objects as facts, and to use only a single grammar rule for nouns:

```
noun(O) --> [O], {object(O, _, _)}.
```

The additional arguments of `object` could be the initial location and the description of the object.

Further Possibilities (2)

- The cut `!`, the disjunction (which can also be written `|`), and the if-then symbol `->` do not need to be included in `{...}`.

  One can also use parentheses `(...)` to structure the alternatives.

- For instance, the optional article before the noun can also be encoded in a single rule:

```
noun_phrase(O) --> ([the] | []), noun(O).
```

- The cut can help to improve the efficiency of the syntax analysis.

Further Possibilities (3)

- The left hand side of the syntax rule can contain a "look-ahead terminal", e.g.

  ```
  p, [a] --> q, [a].
  ```

  means that the production `p --> q` can only be applied if `a` is the next token.

  This is translated to `p(X1,X4) :- q(X1,X2), X2=[a|X3], X4=[a|X3]`, i.e. the look-ahead terminal" is inserted back into the input stream after the rule is processed. In the example, `X2 = X4`, thus the `a` is not consumed.

Efficiency Improvements

- Avoid left recursion.

  This is usually not only inefficient, but wrong: At least for incorrect inputs it easily gets into an endless recursion.

- Think about possible cuts, especially before tail recursions.
- It might be possible to use the Prolog index over the first argument.

```
lookahead(Token), [Token] --> [Token].
stmt --> lookahead(Token), stmt(Token).
stmt(if) --> [if], cond, [then], stmt.
...
stmt(id) --> [id], [':='], expression.
```

Formal Syntax

```
g_rule --> g_head, ['-->'], g_alt.
g_head --> non_terminal, ([','], terminal | []).
g_alt  --> g_if, (['|'], g_alt | []).
g_if   --> g_rhs, (['->'], g_rhs | []).
g_rhs  --> g_item, ([,], g_rhs | []).
g_item --> terminal.
g_item --> non_terminal.
g_item --> variable.
g_item --> ['!'].
g_item --> ['('], g_alt, [')'].
g_item --> ['{'], prolog_goal, ['}'].
non_terminal --> any_callable_prolog_term.
terminal --> ['['], (toks | []), [']'].
toks --> any_prolog_term, ([','], toks | []).
```

Contents

Lexical Analysis (1)

- The input to the syntax analysis (parser) is usually a list of word symbols, called tokens.
- Of course, one could also use a list of characters (atoms or ASCII codes).
- However, since the combination of characters to words is simple, a more efficient algorithm (without backtracking) can be used.
- This reduces a long list of characters to a short list of words. Then the more complex algorithm can work on a shorter input.

Lexical Analysis (2)

- The module that is responsible for transforming a sequence of characters into a sequence of word symbols (lexical analysis) is called the scanner.
- The separation of lexical analysis and syntax analysis is also useful because the scanner can suppress
  - white space between tokens

    Usually, any sequence of spaces, tabulator characters, and line ends is permitted.
  - comments.
- This simplifies the syntax analysis.

Lexical Analysis (3)

- The following example program reads input characters until the line end.
- It skips spaces and composes sequences of letters to words. Other characters (punctation marks etc.) are treated as one-character tokens.
- The main work is done by a predicate `scan` that gets the current input character as first argument:

```
scanner(TokList) :-
    get_char(C),
    scan(C, TokList).
```

Lexical Analysis (4)

```
scan('\n', []) :- !.
scan(' ', TokList) :- !,
    get_char(C),
    scan(C, TokList).
scan(C, [Word|TokList]) :-
    letter(C), !,
    read_word(C, Letters, NextC),
    name(Word, Letters),
    scan(NextC, TokList).
scan(C, [Sym|TokList]) :-
    name(Sym, [C]),
    get_char(NextC),
    scan(NextC, TokList).
```

Lexical Analysis (5)

- The predicate `read_word` reads a list of letters starting with character `C`. It returns the character `NextC` that follows after the word:

```
read_word(C, [C|MoreC], NextC) :-
    letter(C), !,
    get_char(C2),
    read_word(C2, MoreC, NextC).
read_word(C, [], C).
```

- The predicate `letter` defines which characters can appear in words:

```
letter('a').
...
```