Programming Paradigms

Paper Assignment: Definite Clause Grammars

May 01, 2024

Thomas Gueutal
s0195095@ad.ua.ac.be

0. Abstract

Definite clause grammars (DCGs) form a first-order logic equivalent of context-free grammars (CFGs). They allow languages to be defined declaritively and are uniquely suited to reasoning with strings and lists. This paper explores the ties between CFGs and DCGs to give a theoretical foundation of DCGs' expressiveness, and then demonstrates DCGs in Prolog as a practical counterpart to the theory.

1. Introduction

The goal of grammars is two-fold. First is testing membership of an input string in the language the grammar describes. Second is generating possible strings in the language. Classical grammars do this by applying derivation and or recursive inference to a set of derivation rules, in relation to an input string, in what is essentially a long sequence replacements of one intermediate substring (sentential string) by another: top-down processes map rules head-to-body, bottom-up processes map rules body-to-head. As a theoretical foundation of DCGs, we first look into how CFGs tie into DCGs.

Provided with this paper should be Prolog scripts to demonstrate the DCGs discussed in this paper, and matching bash scripts that contain example goal queries.

2. Preliminary definitions

2.1. Context Free Grammars (CFGs)

A CFG specifies a language; it describes a collection of strings that all conform to the same pattern. These strings are composed of specific characters (or "words"). Said differently, a CFG defines a set of "sentences" according to "grammatical rules" using a chosen "alphabet".

A CFG G is defined as a 4-tuple $G = (\boldsymbol{V}, \boldsymbol{T}, \boldsymbol{P}, S)$.
For example Figure 2: $G = (\{S\}, \{a, b\}, \boldsymbol{P}, S)$ where $\boldsymbol{P}$ consists of the two rules in the figure.
$\boldsymbol{T}$ is the set of terminals or the "alphabet" of the language.
$\boldsymbol{P}$ is a set of productions or replacement rules that together specify the language's "grammatical pattern". The form of any CFG rule is as seen in Figure 1: a head that is a *single* variable, an arrow and a body. Here ε is the 'empty string' symbol meaning 'no symbols', with $\varepsilon \notin \boldsymbol{T} \cup \boldsymbol{V}$. Next, $\alpha_1, ..., \alpha_k \in (\boldsymbol{T} \cup \boldsymbol{V} \cup \{\varepsilon\})$ and $\beta = \alpha_1 ... \alpha_k$ is a sentential string of length $k \geq 0$. So every rule body β is any permutation of terminals and variables. Note that a rule with variable H as head may still contain H in its body, and $k = 0$ is synonymous with $\beta = \varepsilon$.
$\boldsymbol{V}$ is a set of variables or non-terminals that individually represent the constituent sub-patterns of G's overarching grammatical pattern, in the sense that we can partition $\boldsymbol{P}$ so that each subset contains only rules with the same head. Each subset then forms a self-contained chunk of the pattern of G.
Finally, $S \in \boldsymbol{V}$ is the single initial or start variable. Via a top-down recursive replacement strategy, where we replace variables by the body of its corresponding rules, we can arrive at strings that consist solely of terminals. This process is called derivation. The bottom-up equivalent is recursive inference.

$$H \to \varepsilon$$
$$H \to \alpha_1 ... \alpha_k$$

Figure 1: CFG Productions

$$S \to \varepsilon$$
$$S \to aSb$$

Figure 2: CFG Example: $a^n b^n$

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

Figure 3: Prolog DCG Example: $a^n b^n$

2.2. Definite Clause Grammars (DCGs)

As the name implies, a DCG is a grammar that defines a language using definite clauses (DCs) as rules. We now describe a transformation from a CFG G to an equivalent DCG G'.

A definite clause is a **logical statement** as in Equation 2 and means "P *is true if* Q_1 *is true and ... and* Q_k *is true*". Both the head P and all Q_i in the body are literals. A literal has the form $p(t_1, ..., t_j)$, consisting of a j-arity predicate p and its arguments t_i. An argument is a term, which can be: a constant, a variable or a compound

term of the form $f(t_1, ..., t_m)$ where f is a m-arity functor and its arguments t_n are again terms themselves. [1] This definition of terms reflects the recursive nature of DC rules, also present in CFGs.

$$H \rightarrow \alpha_1 \alpha_2 ... \alpha_k \tag{1}$$

$$P \Leftarrow Q_1 \ \& \ ... \ \& \ Q_k. \tag{2}$$

$$h(S_0, S_k) \Leftarrow a_1(S_0, S_1) \ \& \ a_2(S_1, S_2) \ \& \ ... \ \& \ a_k(S_{k-1}, S_k) \tag{3}$$

Knowing this, any context-free rule as in Equation 1 can be translated into a corresponding definite clause seen in Equation 3. Prolog's predicate semantics match that of these target definite clauses by design; both employ predicates, constants and variables in the body of their predicates. The second argument of each literal is always fed forward as the first argument of the subsequent literal. This is key to using definite clauses for parsing.

The **variables** S_i represent positions *between elements* in the input string we are matching using the DCG. Take for example $w = {}_0the_1cat_2eats_3the_4mouse_5$. So $a_i(S_{i-1}, S_i)$ is read as "the substring from position S_{i-1} until and including position S_i in the input string matches the pattern described by predicate a_i". So the sequence $(S_0, S_1), ..., (S_{k-1}, S_k)$ within the same definite clause refers to a series of subsequent substrings of the input string contained in the position range (S_0, S_k). Unit clauses have implicit access to the input string, to validate it.

In Figure 4, suppose we apply predicate $s(S_0, S_2)$ to w, then $np(S_0 = 0, S_1 = 2)$ matches "*the cat*" and $vp(S_1 = 2, S_2 = 5)$ matches "*eats the mouse*". Both literals np and vp describe subsequent substrings – sub-patterns – of w and evaluate to true, which implies predicate s does as well.

$$\left.\begin{array}{l} S \rightarrow NP \ \ VP \\ NP \rightarrow A \ \ N \\ VP \rightarrow V \ \ NP \end{array}\right\} \Rightarrow$$

$$\begin{array}{r} s(S_0, S_2) \Leftarrow np(S_0, S_1) \ \& \ vp(S_1, S_2). \\ np(S_0, S_2) \Leftarrow a(S_0, S_1) \ \& \ n(S_1, S_2). \\ vp(S_0, S_2) \Leftarrow v(S_0, S_1) \ \& \ np(S_1, S_2). \\ a(0,1). \quad a(3,4). \\ n(1,2). \quad n(4,6). \\ v(2,3). \end{array}$$

Figure 4: CFG to DCG transformation, adapted from [1]
The unit (0-arity) clauses a, n and v are derived explicitly from w

Practical applications of DCGs may replace the use of positions with an equivalent approach for performance reasons or to improve the readability of a created DCG.

"*Given the translation of a CFG G with start symbol S into a set of definite clauses G' with corresponding "start" predicate s, to say that a string $w \in \boldsymbol{T}^{l-1}$ is in the CFG's language is equivalent to saying that the start goal $s(0, l)$ is a consequence of G' and a set of unit (0-arity) clauses or facts that represents w*" [1], such as the last clauses a, n and v in the DCG of Figure 4.

Note that the given definition of DCGs allows each literal to have additional arguments besides its fixed (S_{i-1}, S_i) positions pair; notice that above a literal's arguments are not limited to exactly two. In practice it is easier to omit (S_{i-1}, S_i) in DCG definitions and simply let a "compilation" step add them pre-runtime. Figure 3 depicts such a DCG (in Prolog), where the square brackets embed terminals into the rule and *start* is a predicate or literal that omits its two position arguments. In this case, the predicate *start*/0 would be "compiled" into corresponding definite clauses *start*/2 by adding the missing positions to both its heads and recursively to all literals in its bodies. In actuality, Prolog implements DCGs using difference lists instead of positions, but it does add two extra arguments in "compilation".

You should now be convinced that plain DCGs that only use two parameters (S_{i-1}, S_i) for each literal have the same expressive power as CFGs since there always exists a corresponding DCG, as they were just defined, for any CFG by the translation process above. The same holds in the other direction.

2.3. Earley Deduction Proof Procedure

You may now wonder if in practice it is feasible to match strings using DCGs, since algorithms applicable to CFGs may not work for DCGs anymore due to the inclusion of predicate logic. The Earley parsing algorithm is a famous top-down parsing algorithm that fits into the chart parser category [2]. This by definition gives it expressivity on the level of CFGs [1]. The **Earley deduction proof procedure** [1] is the predicate logic's counterpart to the Earley parsing algorithm for CFGs.

That is to say, there do exist theoretically grounded algorithms to check the membership of a string in the language of a DCG.

3. Main result: DCGs in Prolog

Prolog does not make use of the Earley deduction proof procedure. It instead embeds the **fundamental inference rule for definite clauses** into its search procedure (unification). [1]

"*The fundamental inference rule for definite clauses is the following resolution rule. From clauses Equation 4 and Equation 5 when B and D_i are unifiable by substitution σ, infer Equation 6 as a derived clause and the resolvent of the two.*" [1]

$$B \Leftarrow A_1 \ \& \ \ldots \ \& \ A_n \tag{4}$$

$$C \Leftarrow D_1 \ \& \ \ldots \ \& \ D_i \ \& \ \ldots \ \& \ D_n \tag{5}$$

$$\sigma[C \Leftarrow D_1 \ \& \ \ldots \ \& \ D_{i-1} \ \& \ A_1 \ \& \ \ldots \ \& \ A_n \ \& \ D_{i+1} \ \& \ \ldots \ \& \ D_n] \tag{6}$$

"*A goal clause like Equation 5 is successively rewritten by the resolution rule using clauses from the program Equation 4* ". In other words, Prolog declares a goal clause true if it can recursively prove that the body's literals resolve to true. "*The Prolog proof procedure can be implemented very efficiently, but it has the same theoretical problems of the top-down backtrack parsing algorithms after which it is modeled.*" [1]

We can expand the expressiveness of DCGs by utilizes additional arguments in choice literals. Then **context sensitive grammars can be expressed** as well. More on this in Appendix 1.

As such, Prolog provides a specialized syntax compared to the plain DCG syntax to improve its expressivity from CFGs to CSGs and to preserves Prolog's logical characteristics. Prolog alters DCGs as follows: **square brackets** or double quotes are used to embed terminals, but they have different semantics. Variables can be embedded into brackets as well and allow reasoning with arbitry substring matches. **Curly braces** are used for plain Prolog code. This makes it easy to enforce complex constraints onto definite clauses. **Cuts** are allowed in- and outside curly braces. Figure 5 depicts the Prolog DCG syntax, the annotations on top are substrings that will be matched by each particular piece of code.

```
                          [this]   [105, 115] or [i, s]   [some] or [a]   if X unassigned,          [good, example]   [!] or [?] or [.]
head(X)  --> [this] ,      "is"   , [X], {word(X)},      force X= 'some'   , [good, example],   punct   .
                                                                  !
punct    --> [!] | [?] | [.].
word(X)  :-  X = 'some' ; X = 'a'.
```

Figure 5: Prolog DCG syntax

Prolog makes use of difference lists (L_1, L_2) instead of the position arguments (S_{i-1}, S_i). The pair (L_1, L_2) represents $L_{\text{diff}} = L_1 - L_2$, for example $[\text{the}, \text{diff}] = [\text{the}, \text{diff}, \text{and}, \text{more}] - [\text{and}, \text{more}]$. So we require L_2 to be a tail of any length of L_1. This implementation is for the sake of efficiency, but is equivalent to the positions of the initial DCG definition.

4. Example: Sierpinski Triangles

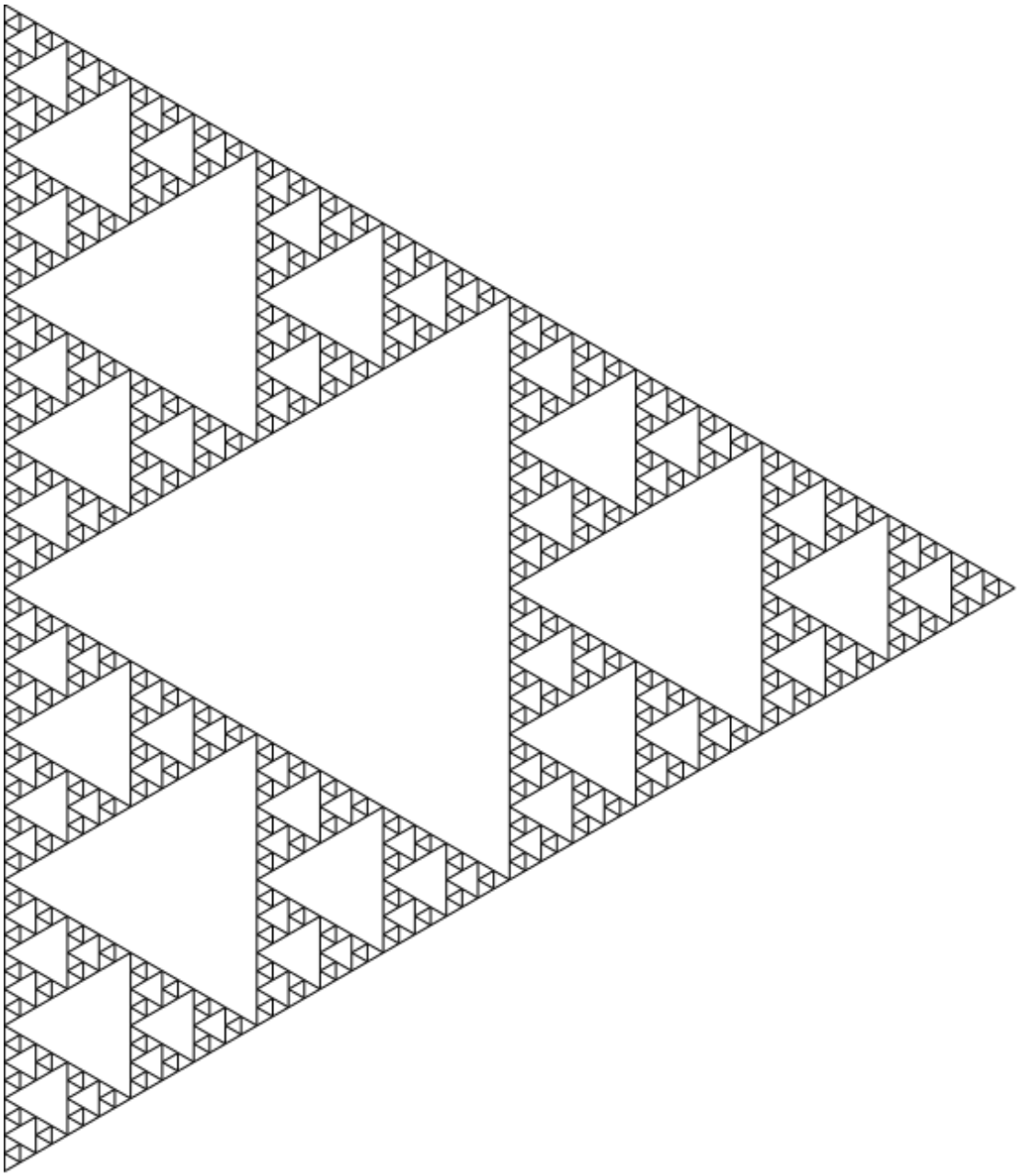
The Prolog scripts and corresponding bash scripts provided with the paper may serve to illustrate the use of DCGs for generation and checking of sequences. To illustrate this usefulness, a short demonstration of the generation of a sierpinski triangle pattern is in order. More detail is also available in the added README file.

Two implementations are provided: one is a pure DCG, the other is a DCG embedded into a predicate.

Calling the bash script will generate a rudimentary SVG image of a sierpinski triangle with the given parameters.

5. Conclusion

The versatility of the Prolog unification procedure built on the fundamental DC resolution rule embedding makes it so that "*DCGs can be used to parse, generate, complete and check sequences manifested as lists.*" [3] This provides a nicely declarative way of defining grammars using predicate logic for all manner of uses.



1. Context Sensitive Grammars (CSGs)

The defining difference between context-**free** grammars (CFG) and context-**sensitive** grammars (CSG) is the head of their productions. A CFG always has a single variable as its head. A CSG may have a string of terminals and non-terminals as a head. This is depicted in Figure 7, where α, β, γ are permutations of zero or more terminals and variables and ε defined as before in Section 2.1.

This gives CSGs additional expressiveness compared to CFGs, because the rules can take advantage of the surrounding terminals and variables as a source of **contextual knowledge** for the substitution during the process of derivation or recursive inference. Let us demonstrate with an example. Remember Figure 2, which defines the language $a^n b^n$ as a CFG. We want to construct a similar grammar for the pattern $a^n b^n c^n$. Note that it is impossible to define a CFG for that language as per the pumping lemma, but this *is* possible using CSGs as seen in Figure 8.

The CSG works as follows: S lets you choose between $n = 0$ or $n > 0$, G generates an equal amount of a's and b's so that the b's are interleaved by the variable C, while every C pushes itself towards the right through the b's until it leaves them and then converts to a terminal c. The terminal e facilitates the very first C to c conversion.

$$H \to \varepsilon$$
$$\alpha H \beta \to \gamma$$

Figure 7: CSG Productions

$$\begin{aligned} S &\to Ge \mid \varepsilon \\ G &\to aGbC \mid abC \\ Cb &\to bC \\ Cc &\to cc \\ Ce &\to c \end{aligned}$$

Figure 8: CSG Example: $a^n b^n c^n$

$$\begin{aligned} \text{start}(N) &\to \text{as}(N), \text{bs}(N), \text{cs}(N). \\ \text{as}(0) &\to [\ ], !. \\ \text{bs}(0) &\to [\ ], !. \\ \text{cs}(0) &\to [\ ], !. \\ \text{as}(N) &\to [a], \text{as}(M), \{N \text{ is } M + 1\}. \\ \text{bs}(N) &\to [b], \text{bs}(M), \{N \text{ is } M + 1\}. \\ \text{cs}(N) &\to [c], \text{cs}(M), \{N \text{ is } M + 1\}. \end{aligned}$$

Figure 9: DCG Example: $a^n b^n c^n$ adapted from [4]

References

[1] F. C. Pereira and D. H. D. Warren, "Parsing as Deduction," in *Annual Meeting of the Association for Computational Linguistics*, 1983. [Online]. Available: https://api.semanticscholar.org/CorpusID:776531

[2] D. Arnold, "Chart Parsing." Accessed: Apr. 21, 2024. [Online]. Available: https://web.archive.org/web/20150221021038/http://webdocs.cs.ualberta.ca/~lindek/650/papers/chartParsing.pdf

[3] "Markus Triska (2024). The Power of Prolog." Accessed: Apr. 16, 2024. [Online]. Available: https://www.metalevel.at/prolog/dcg

[4] "Wikipedia (2024). Definite clause grammar." Accessed: Apr. 16, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Definite_clause_grammar#Non-context-free_grammars