A Practical Implementation of DCGs

JUKKA PAAKKI

Nokia Research Center
P.O.Box 156
SF - 02101 Espoo, Finland
(paakki@rc.nokia.fi)

Abstract

Definite clause grammars (DCGs) are a logic counterpart to context-free grammars, the most widely-used formalism for defining the syntax of languages. The conventional implementation strategy of DCGs is translation into the logic programming language Prolog, giving rise to a parser for the defined language. This translation can be carried out in a rather straightforward way, and that is why DCGs are provided as an enhancement in a number of Prolog systems. Definite clause grammars were originally presented in [PeW80] as a formalism for describing natural languages. The implementation method presented there (and adopted as such in the Prolog systems) thus takes into account the most general structure of natural languages and produces a parser which is inherently nondeterministic.

The appealing combination of a definition formalism (context-free grammars) and a direct operational realization of that formalism (parser) has been the inspiration for applying DCGs also outside their original intended domain. The most notable example is compiler writing for programming languages. While DCGs have been commonly presented as an advanced compiler writing tool (e.g. [StS86], [Szp87]), a more careful analysis of their traditional implementation strategy reveals shortcomings with respect to practical parsing of programming languages. The problems can be immediately noticed when taking a look on the way a DCG is transformed into ordinary Prolog code:

1. Nondeterministic parsing is simulated with Prolog's normal backtracking mechanism. Now the order of the alternative productions has great significance on the efficiency of the parser since they are tried in the order of appearance within the DCG.
2. The parser cannot in general deal with left-recursive productions (nor with regular expressions) and may loop infinitely when trying to apply such a production. This may happen both for syntactically legal inputs as well as for syntactically illegal ones.
3. No recognition of syntax errors is provided, but instead the parser simply fails (or loops) with a syntactically erroneous input.
4. Scanning cannot be interleaved with parsing since the source program is represented as a complete list of tokens. This leads necessarily to at least two passes over the source program just for parsing it.

Since DCGs as a notation are rather compact and elegant, they certainly are a powerful tool in experimental and prototyping programming language implementation. However, when aiming at practical applications, the standard implementation makes DCGs unusable. We have produced an implementation of DCGs that drives at removing the troublespots, at the same time retaining the general and powerful notation. Our system accepts DCGs in their standard form, but the implementation most notably circumvents the problems 1 and 3 mentioned above: in our system a DCG gives rise to a deterministic error-recovering parser that never fails or loops.

The main objective of our DCG implementation is to provide an automatic error handling mechanism. Since syntactic error detection and recovery are most laborious in connection with nondeterministic parsing, and since ambiguous grammars are rarely needed in defining modern programming languages, we have decided to base our DCG facility on deterministic parsing. The particular grammar class is LL(1),

realized in this case as top-down recursive descent parsers; hence a mapping to Prolog's normal execution model is easy to establish.

The most important facility of our DCG implementation, automatic error recovery, is based on the well-known *panic mode* and *phrase level* methods, as described e.g. in [WeM80]. The idea is to keep the parser always in synchron with the input stream. This means that when detecting an error, the parser skips symbols in the input, until a symbol is found that matches the current state of the parser. The parser and the input are synchronized both at entry and at exit of each nonterminal. Synchronization is based, as usual, on the FIRST and FOLLOW sets of symbols (see e.g. [ASU86]).

The system is implemented in Quintus Prolog [QCS86], and it produces parsers in Quintus Prolog. We have analyzed the performance of the generated parsers by comparing three different DCG implementations: the conventional Prolog-translator embedded in Quintus Prolog, a DCG meta-interpreter [PaT90], and the Prolog-translator presented in this paper. Comparison between the translation based implementations shows that our implementation produces slower parsers than the conventional implementation, the factor being typically about 4. The obvious reason to this loss of performance is the introduction of the error recovery mechanism into the parsers: a parser-input synchronization is carried out twice for each production, while the conventional implementation totally ignores such concerns. On the other hand, efficiency has not been the major motivation of our DCG implementation, but instead the emphasis has been on automatic syntactic error detection and recovery. This facility has turned out to be most valuable when parsing programs of reasonable length: our system is always able to analyze a source program and report its illegal constructs, whereas the conventional implementation remains totally silent in such cases. Taking into account the provided error feedback, the performance of the generated parsers is quite acceptable.

Since we have exploited standard syntactic error handling methods, the quality of the generated error recovery mechanism is rather typical for recursive descent parsers: in ordinary cases the parsers are able to detect most of the actual errors, but on the other hand the amount of text that is skipped during synchronization may be rather large. This implies that some actual errors may not be reported at all, while some correct sentences may be incorrectly reported as erroneous.

The DCG notation and its implementation are one central part of the PROFIT system [Paa89], currently under development. PROFIT as a language is based on Prolog and intended especially for compiler writing. Some notable features of the language are determinism, the DCG facility, and functions. Since the main implementation method of the language is translation into Prolog, PROFIT can also be considered as a compiler writing system for the implementation of logical one-pass attribute grammars [Paa90], a logic-based subclass of attribute grammars.

References

[ASU86] Aho A.V., Sethi R., Ullman J.D.: *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Paa89] Paakki J.: A Prolog-Based Compiler Writing Tool. In: *Proc. of the Workshop on Compiler Compiler and High Speed Compilation*, Berlin (GDR), 1988. Report 3/1989, Akademie der Wissenschaften der DDR, 1989, 107-117.

[Paa90] Paakki J.: A Logic-Based Modification of Attribute Grammars for Practical Compiler Writing. In: *Proc. of the 7th Int. Conference on Logic Programming* (D.H.D.Warren, P.Szeredi, eds.), Jerusalem, 1990. The MIT Press, 1990, 203-217.

[PaT90] Paakki J., Toppola K.: An Error-Recovering Form of DCGs. *Acta Cybernetica* (To appear).

[PeW80] Pereira F., Warren D.: Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13, 1980, 231-278.

[QCS86] Quintus Computer Systems, Inc.: *Quintus Prolog Reference Manual*, Version 6, 1986.

[StS86] Sterling L., Shapiro E.: *The Art of Prolog*. The MIT Press, 1986.

[Szp87] Szpakowicz S.: Logic Grammars. *BYTE* 8, 1987, 185-193.

[WeM80] Welsh J., McKeag M.: *Structured System Programming*. Prentice-Hall, 1980.